Mission-driven Software-intensive System-of-Systems Architecture Design

Eduardo Ferreira Silva

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

THESE DE DOCTORAT DE

L'UNIVERSITE BRETAGNE SUD
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Eduardo Alexandre FERREIRA SILVA

Conception d'architecture de système-de-systèmes à logiciel prépondérant dirigée par les missions

Thèse présentée et soutenue à Natal/Vannes, le 17 décembre 2018
Unité de recherche : IRISA (UMR CNRS 6074)
Thèse N° : 509

Rapporteurs avant soutenance :

Khalil DRIRA	Directeur de Recherche CNRS, LAAS-CNRS – Université Toulouse, France
Abdelhak-Djamel SERIAI	Maître de Conférences HDR, LIRMM – Université de Montpellier, France

Composition du Jury :

Président :	Elisa Yumi NAKAGAWA	Professeur « Livre Docente », ICMC – Université de São Paulo, BR
Examinateurs :	Khalil DRIRA	Directeur de Recherche CNRS, LAAS-CNRS – Université de Toulouse, FR
	Abdelhak-Djamel SERIAI	Maître de Conférences HDR, LIRMM – Université de Montpellier, FR
	Marcel OLIVEIRA	Maître de Conférences HDR, Université Fédérale Rio Grande do Norte, BR
Dir. de thèse :	Flavio OQUENDO	Professeur des Universités, IRISA – Université Bretagne Sud, FR
Co-dir. de thèse :	Thais BATISTA	Professeur des Universités, Université Fédérale Rio Grande do Norte, BR

Eduardo Alexandre Ferreira Silva

Mission-driven Software-intensive System-of-Systems Architecture Design

Doctoral dissertation submitted in partial fulfillment of the requirements for the degrees of Doutor em Sistemas e Computação and Docteur en Informatique under the joint supervision agreement between the Universidade Federal do Rio Grande do Norte (UFRN), Brazil, and Université Bretagne Sud (UBS), France.

Supervisors

Profa. Dra. Thais Vasconcelos Batista

Prof. Dr. Flavio Oquendo

Natal, Brazil

December 2018

To Aurora, the brightest light in my life.

Acknowledgment

It is not easy to say thanks in a PhD thesis. Not for the process of acknowledging the participation of thirds in the journey, but for not knowing when to stop doing it. Although my name is the only one in the cover of this manuscript, I like to think I am not the solo responsible for it.

Whatever I've accomplished up to this point, it make me better understand Isaac Newton's words: "*If I have seen further it is by standing on the shoulders of Giants*". This work was written by a thousand hands, but I'll give special attention to those physically close to me, that helped me not only professionally, but also emotionally.

"*Family always comes first*", is something we are used to hear everywhere, and is probably the greater truth people say. Thank you Aline, my beloved wife, and Aurora, my little daughter, for all support and willpower you gave me along these years, I would surely tear apart without you. I also thank my family as a whole: father, mother, grandparents, aunts and uncles, and cousins. I'm sure I am what I am thanks to you all.

I'm deeply indebted to my supervisor: Thais Batista. Everything I've accomplished during all those years, from graduating, mastery and now a PhD: I had you by my side. You gave me all the opportunities that made me become what I am. After ten years you become more than a supervisor and idol to me, but a good friend in whom I can trust completely. Thank you for the understanding, the caring, the attention and the guidance during the whole process.

I was blessed with two amazing supervisors. Flavio Oquendo, my second supervisor, is one of the professionals I respect the most on earth. Thanks for giving me the opportunity to work with you and the researchers of IRISA, for being a important contributor on the most amazing experience I've ever lived. Flavio and Thais are, together, my greatest inspiration.

I am sure the friends we have have a major influence in all aspects of our lifes. Some say we are the average of our five closest friends. Whether this is true or not, I specially thank my five closest friends: Carlos Pinheiro, Marccelo Amaral, Anderson Araújo, João Ribeiro, and David Vasconcelos.

I thank my good friends in the ConSiste laboratory, in UFRN. Particularly Everton Cavalcante, whom worked with me for many years, and Lidiane Santos, who will surely will work with me for a couple years to come. I also thank my friends in IRISA, in UBS, and the friends I made in Vannes. Special thanks to Fadhlallah, Anne, and Cathy, for the experiences we lived together.

Thanks Jean Quilbeuf, Didier Vojtisek and all the team from IRISA Rennes, that helped me in the hardest part of this work. Thanks Gersan Moguerou, for helping me understand better my own work.

Finally, I thank the *Brazilian Coordination for the Improvement of Higher Education Personnel* (CAPES) and the Brazilian government for financial support, that made this work possible.

For the horde!

Mission-driven Software-intensive System-of-Systems Architecture Design

Author: Eduardo Alexandre Ferreira Silva
Supervisor: Thais Batista, PhD
Supervisor: Flavio Oquendo, PhD

Abstract

The formulation of missions is the starting point to the development of Systems-of-Systems (SoS), being used as a basis for the specification, verification and validation of SoS architectures. Specifying, verifying and validating architectural models for SoS are complex tasks compared to usual systems, the inner complexity of SoS relying specially on emergent behaviors, i.e. features that emerge from the interactions among constituent parts of the SoS which cannot be predicted even if all the behaviors of all parts are completely known. This thesis addresses the synergetic relationship between missions and architectures of software-intensive SoS, giving a special attention to emergent behaviors which are created for achieving formulated missions. We propose a design approach for the architectural modeling of SoS driven by the mission models. In our proposal, the mission model is used to both derive, verify and validate SoS architectures. As first step, we define a formalized mission model, then we generate the structure of the SoS architecture by applying model transformations. Later, when the architect specifies the behavioral aspects of the SoS, we generate concrete SoS architectures that will be verified and validated using simulation-based approaches, in particular regarding emergent behaviors. The verification uses statistical model checking to verify whether specified properties are satisfied, within a degree of confidence. The formalization in terms of a temporal logic and statistical model checking are the formal foundations of the developed approach. A toolset that implements the whole approach was also developed and experimented.

Keywords: Software Architecture, Software-intensive Systems-of-Systems, Mission Modeling, Semi-Automated Architecture Design

Mission-driven Software-intensive System-of-Systems Architecture Design

Auteur: Eduardo Alexandre Ferreira Silva
Superviseur: Thais Batista, PhD
Superviseur: Flavio Oquendo, PhD

Résumé

La formulation des missions est le point de départ du développement de systèmes-de-systèmes, étant utilisée comme base pour la spécification, la vérification et la validation d'architectures de systèmes-de-systèmes. Élaborer des modèles d'architecture pour systèmes-de-systèmes est une activité complexe, cette complexité reposant spécialement sur les comportements émergents, c'est-à-dire, des comportements issus des interactions entre les parties constituantes d'un système-de-systèmes qui ne peuvent pas être prédits même si on connaît tous les comportements de tous les systèmes constituants. Cette thèse adresse le lien synergique entre mission et architecture dans le cadre des systèmes-de-systèmes à logiciel prépondérant, en accordant une attention particulière aux comportements émergents créés pour réaliser les missions formulées. Nous proposons ainsi une approche pour la conception d'architecture de systèmes-de-systèmes dirigée par le modèle de mission. Dans notre approche, le modèle de mission sert à dériver et à valider les architectures de systèmes-de-systèmes. Dans un premier temps, nous générons la structure de l'architecture à l'aide de transformations de modèles. Ensuite, lors que l'architecte spécifie les aspects comportementaux, la description de l'architecture résultante est validée à l'aide d'une démarche conjointe qui comprend à la fois la vérification des propriétés spécifiées et la validation par simulation des comportements émergents. La formalisation en termes de logique temporelle et la vérification statistique de modèles sont les fondements formels de l'approche. Un outil mettant en œuvre l'ensemble de l'approche a été également développé et expérimenté.

Mots-clés: Architecture logicielle, systèmes-de-systèmes à logiciel prépondérant, modélisation de missions, conception architecturale semi-automatisée.

List of Figures

List of Tables

List of Acronyms

System-of-Systems – SoS

Software Intensive Systems of Systems – SiSoS

Verification and Validation – V&V

Linear Temporal Logic – LTL

Model-Driven Development – MDD

Model-to-Model – M2M

Atlas Transformation Language – ATL

Query/View/Transformation – QVT

Statistical Model Checking – SMC

Flood Monitoring SoS – FMSoS

Linear Temporal Logic - LTL

eXecutable Domain Specific Modeling Language – xDSML

Kermeta3 – K3

Boolean Satisfiability Problem – SAT

Comprehensive Modeling for Advanced Systems of Systems – COMPASS

Unifying Theories of Programming – UTP

Goal-Oriented Software Architecting – GOSA

Functional Requirements – FR

Non-Functional Requirement – NFR

Business Process Execution Language – BPEL

Computer-Aided Software Engineeering

List of Symbols

$\neg$ Logical operator of negation

$\wedge$ Logical operator AND

$\vee$ Logical operator OR

$\Longrightarrow$ Logical operator IMPLIES

$\Longleftrightarrow$ Logical operator IFF

φ Logical formula

σ Logical formula

$\mathfrak{U}$ UNDEFINED value

Contents

1 Introduction

Software is everywhere. The rapid evolution of electronics allowed us to introduce software components in the various unusual and unexpected elements of our daily life. Such evolution also drastically improved computational power, thereby allowing software systems to become more complex and bigger, at the same time as faster. Altogether, these aspects woke an interest for integrating software systems in cooperation environments, using a group of existing systems to form a larger, more complex, system that is capable of performing new operations.

Many examples of cooperation-based systems as such can be found. One of the most remarkable recent domains is Internet of Things (IoT) (ATZORI; IERA; MORABITO, 2010; ALKHABBAS; SPALAZZESE; DAVIDSSON, 2017), in which the goal is to integrate many "intelligent" *things* towards a cooperation environment to achieve a predetermined functionality. Every *thing* is completely independent from each other and one of the design challenge relies on how to define a cooperation that would allow the integrated *things* to provide the desired properties. One of the IoT applications are the *smart city* projects (LEEM; KIM, 2013), that consists on integrating existing city systems and services to enhance urban life and development, including traffic, public transportation, social services, etc.

One of the most notorious initiatives for system integration and cooperation focus on independent, heterogeneous **constituent systems**, therefore embracing domains as IoT and smart cities. A system-of-systems (SoS) is defined as the result of the interaction among independent heterogeneous **constituent systems** that cooperate to form a larger, more complex system for accomplishing a given mission (MAIER, 1998).

From the system-of-systems perspective, a constituent system is an independent system that is capable of interacting with other systems. Each constituent system has its own objectives it will try to achieve by its own, the so-called **mission**. The SoS as a whole also have its missions, although differently from the **individual missions** of the constituent

systems, the **global missions** of the SoS can only be achieve through cooperation between the constituent parts.

Each global mission rely on an specific behavior that stems on cooperation, the **emergent behavior**, a behavior that is only observable when the systems are interacting. Although some of these behaviors can be *expected*, it is not possible to *predict* them based on the constituent parts. Since the emergent behavior is *more than the sum of the parts* (OQUENDO, 2018), they cannot be calculated based on the behaviors of the constituent systems (BOARDMAN; SAUSER, 2006).

Once emergent behaviors cannot be associated to any single constituent system, neither the global missions that rely on that behavior can. Therefore, the global missions can *never* be achieved by an individual constituent system, being then a characteristic of the SoS as a whole (NAQVI et al., 2010).

Besides emergent behavior, there are other intrinsic characteristics that make SoS distinct from other distributed, complex and large-scale systems. Regarding constituent systems, they have (i) **the operational and managerial independence**, that consists on providing their own functionalities even when they do not cooperate within the scope of the SoS and can be managed independently, and are (iii) geographically distributed. The SoS have an (ii) **evolutionary development**, that establishes that the systems may evolve over time to respond to changes on its execution environment, or on its own missions. Altogether, these characteristics have posed a set of challenges mainly related to the development, dynamicity, and evolution of SoS, thereby making traditional system engineering processes to be no longer suitable for constructing these systems (BOEHM; LANE, 2006; CALINESCU; KWIATKOWSKA, 2010).

As a subset of SoS, Software-Intensive Systems of Systems (SiSoS) are a kind of SoS in which software plays a key role (ISO 42010:2011, 2011). In this kind of systems, the adoption of software engineering processes sightly impacts on development, implantation and maintenance of the systems. The increasing complexity of software systems caused a growing interest for SiSoS within the Software Engineering. Since the solutions for SiSoS requires a complex, software-based integration for the constituent systems to form a SoS, the traditional approaches are often ineffective.

Although this work focus on SiSoS, the term SoS might be found along the text for simplification purposes. However, it is important to clarify that this thesis proposes a solution and uses background specifically for SiSoS.

This Chapter introduces the problem within the context. Section 1.1 gives an overview and discuss the needs of the domain. Section 1.2 presents the research questions and goals of this work. Section 1.3 presents the expected contributions of this work and Section 1.4 gives an overview of our evaluation proposal. Finally, Section 1.5 describes the outline of this thesis.

1.1 Problem Statement

This Section introduces the problem, contextualizing and discussing the aspects related to this work. Section 1.1.1 presents the SoS context and the role missions play in it and the architecture. Section 1.1.2 briefly discusses architectural validation and verification for SoS.

1.1.1 Bridging Missions and Software Architecture in SoS Modeling

An important concern in the design of SiSoS is the systematic modeling of both global and individual missions, as well as all relevant mission-related information. Missions play a key role in the SoS context since they support the identification of required capabilities of constituent systems and the interactions among these systems that may potentially lead to emergent behaviors towards the accomplishment of the global goals of the SoS. Therefore, mission models can be viewed as a potential starting point to be adopted when designing an SoS as they can be used as a basis of the whole development process (SILVA et al., 2014).

In this context, mKAOS (SILVA; BATISTA; OQUENDO, 2015) is a pioneer mission description language, designed for the specificities of SoS. Mission models in mKAOS can be seem as a complimentary requirements model that can be refined to the capability level, expressing the functionalities the systems are able to perform. Such models can express not only missions and capabilities, but also emergent behaviors and environment conditions. Allowing the stakeholders to design and analyze the SoS from the most various viewpoints. It is important to mention that mission models, in mKAOS, do not concern on the implementation or behavior of the involved parts, focusing on the goals and *what* are the potential contributions of each, instead.

In a mission-oriented approach for designing SoS, a next step towards the concretization of the mission model is its refinement to an architectural model, i.e., a model ex-

pressing the SoS software architecture, that will define *how* the desired functions will be implemented. SoS software architectures have been recognized as a significant element for determining the success of these systems and contributing to their quality (GONCALVES et al., 2014; NAKAGAWA M. GONCALVES; OQUENDO, 2013; GUESSI et al., 2015).

Mission models shall be used as a basis for the further elaboration of architectural models by SoS software architects (SILVA et al., 2014), since they specify what the SoS is intended to be. The process to produce an architecture based on the mission model can be classified as a refinement process, since it maintains the coarse-grain-most properties as it introduces new properties that are expected from the fine-grain-most properties. These properties can be used altogether in a verification process, that might automatically detect property violation (COUTO; FOSTER; PAYNE, 2014).

Since such a refinement allows specifying the SoS software architecture in compliance with the mission model, it is possible to establish traceability links between missions and architectural elements. In this context, traceability between missions and architectural elements is fundamental, specifiably due to the unpredictable nature of emergent behavior (OQUENDO, 2018). It is, therefore, necessary to simulate the architectural models to observe which behaviors are emerging and which of those are desired or not. Furthermore, thanks to traceability between models, it is possible to identify the subset of constituent systems that are supporting each behavior. Through the simulation, it is possible to validate the architecture within the mission model.

However, currently, there is a lack of studies that concerns on mission models. Hence, existing architectural definition approaches tends to use traditional requirements engineering. Since constituent systems are **the operational and managerial independent**, i.e. they have their own objectives and are managed by independent entities (MAIER, 1998), such systems often present a **behavioral uncertainty**: the internal function and behavior of these systems are **unknown** or **non-deterministic**. Consequently, traditional architectural approaches are particularly ineffective due to its inability of to cope this kind of circumstance and specially due to the nature of the emergent behavior.

In fact, every technique, framework or methodology we found up to this date completely neglects emergent behavior, focusing on properties as DANSe [1] or interoperability as COMPASS [2]. Further, these approaches rely on traditional architectural description languages, that are often unable to express common characteristics of constituent sys-

[1] /http://www.danse-ip.eu/home
[2] http://www.compass-research.eu/index.html

tems such as the inner dynamism and the behavioral uncertainty. In this context, SosADL (OQUENDO, 2016a) is a novel ADL designed for the specificities of SoS. Formally grounded in π-calculus (OQUENDO, 2016b), the language introduces new constructs that are key on the SoS context, such as coordination elements.

1.1.2 Validation and Verification of Software Architectures for SoS

IEEE ISO 1012-2004 (IEEE ISO 1012-2004, 2005) defines a process for software verification and validation (V&V) that determines whether the products on the development process meet the requirements and therefore the user's needs with a given degree of quality.

Although related, validation and verification are performed at different moments of software production and concerns on different aspects of the system: verification is related to the properties that constraint the specification, permeating between requirements (non-functional requirements) and architectural model (architectural constraints). The verification can be performed at any moment in the implementation process, even with unfinished models. Validation regards the expectations and needs of the stakeholders, therefore it is often performed as a final stage of implementation.

On one hand, validating Software Architectures is a challenge task even for traditional systems, as it aims to guarantee quality degrees for the produced architecture. Therefore, it is an essential part of the development process (MICHAEL; RIEHLE; SHING, 2009).

The validation process consists on checking whether an architecture does what it is supposed to do. The challenge, that normally consists on checking the requirements, is even more complicated for the SoS context. A validation process for SoS must be able to identify when an architecture is capable of achieving the proposed missions, which is a complex concept when compared to requirements. Since global missions depends on emergent behavior, which are unpredictable, the validation process for SoS must rely on simulation, differently from traditional systems.

Most of validation processes for software architecture are mostly manual, in which the architect reads the requirements and identifies whether the system implement it, relying on traceability. In the SoS context, besides the identification of the parts that implement the missions, the architect must identify in which circumstances or contexts the missions are achieved or might fail.

On the other hand, to verify correctness of a system the most popular verification

technique is model checking (CLARKE JR.; GRUMBERG; PELED, 1999; ZHANG; MUCCINI; LI, 2010). Model checking consists on using a system specification in a given notation and a set of properties or constraints, then exhaustively testing the possible states of the system towards the predefined set of properties on each of those states (TSAI; XU, 2000). A model is considered correct, in the verification context, if it complies with the constraints in all possible states.

Traditional model checking, however, relies on building all possible states of a system and are, therefore, subject to the *state space explosion problem* (HOLZMANN, 2002). Hence, when it comes to systems with innate dynamism, uncertainty or intensive concurrency, those traditional techniques becomes obsolete and inefficient, often ineffective. Since SoS are essentially dynamic, concurrent and with some degree of uncertainty regarding the behavior of the constituent systems, traditional model checking is not effective in this context.

Furthermore, model check deeply depends on the notation used to specify the system, since verification techniques requires a notation that is checkable. There are some proposals focused on formalization of architectural models, aiming to allow the architecture to be automatically checked (LICHTNER; ALENCAR; COWAN, 2000). However, a formal background is still one of the most desired features of an architectural description language (ADL), which might support model checking of architectural models made using an built-in formalism. In this context, most of the verification approaches attempts to introduce or use the existing formalism on ADLs, such as EAST-ADL (ENOIU et al., 2012) and Wright♯ (ZHANG et al., 2012).

1.2 Research questions and Goals

Given the problem statement, the main objective of this work is to propose a methodology for developing SoS architectures. This methodology relies on the so-called mission models and includes automated model transformations for producing the architectural model and validation and verification mechanisms for the produced architecture.

We walked through a sandy ground during the identification of the problem to be solved. Many pieces are missing to propose a solid architectural methodology that is based on mission models. First, it was not clear how we could relate the missions and architecture in order to refine the mission model maintaining the properties of the first. Then, expressing an architectural model of SoS has proved to be a tricky activity, specially

when trying to track the mission that would influence in each piece of the architecture. Finally, on verification and validation the problem proved to be tricker, since traditional model checking is not an option and the observation of the emergent behaviors requires simulation due to their unpredictability. We summarized these problems in six Research Questions (RQ):

- RQ1: What are the common concepts that permeate between the mission model's elements and the architectural model?
- RQ2: How can we relate mission model elements with architectural elements?
- RQ3: How to verify mission-related architectural properties in the SoS context?
- RQ4: How to validate an architectural model within a mission model?
- RQ5: How to validate an architecture produced through a mission-based process?
- RQ6: Which kind of architectural validation can be done regarding emergent behaviors?

RQ1 aims to identify some potential trace points that can be useful in a refinement methodology as the one intended by this work. Through the traceability supported by these shared concepts, we can define responsibilities throughout the methodology. RQ2 is complimentary to RQ1, focusing on the bigger picture: "can we relate mission-related elements and properties and architectural elements?". RQ3 and RQ4 concerns on verification and validation, focusing on the techniques and technologies we could use to verify and validate architectures of SoS, considering the properties defined in a mission model. Finally, RQ6 focuses on the emergent behavior, that is often neglected by existing approaches, aiming to find a mean to validate the SoS on the specificity of the emergent behavior.

During the first steps of this study we choose some pieces that showed useful. Specifically, this study is a continuation of a previous study in which we defined mKAOS, a pioneer mission description language (SILVA; BATISTA; OQUENDO, 2015; SILVA; BATISTA; CAVALCANTE, 2015). This language was built over a goal-oriented approach for requirements modeling, adding constructs to represent missions and refine it to the capability level, which represents operations constituent systems provide. Also, we decided to use SosADL (OQUENDO, 2016a): a pioneer formal language for SoS architectural description. Due to the familiarity of the group with those languages and their pioneer nature, we

decided to rely on them to define our methodology. However, we are aware that other languages can emerge on the context, therefore, we briefly discuss how these languages can be renewed in Chapter 8.

1.3 Contributions

This work permeated between many domains of software engineering. Our findings regarding research questions are the main contribution of this study. These conclusions led us to the definition of an architectural process that encompasses all steps of model definition: description, validation, and verification.

- **RQ1**: we identified a set of common concepts that are present in both mission and architectural models. Although these concepts are represented through different constructs and each model focus on a specific facet of such concept, we were able to draw an automatic transformation that would simplify the modeling process. This was a pioneer work, since automatic model transformation was never used in SoS context;
- **RQ2**: the traceability promoted by the common concepts that permeates both mission and architectural model allows us to establish a direct link between missions and the constituent systems that are involved in its achievement;
- **RQ3**: we identified an alternative to traditional model checking that supports the dynamism and behavioral uncertainty that hovers the constituent system in a SoS, allowing us to verify the compliance of the architecture within properties described in the mission model;
- **RQ4**: regarding validation, we propose a simulation-based validation that can be partially automatized to validate the architecture within the mission model. For doing so, we use the verification mechanism to automatically check for mission accomplishment and arrival of emergent behavior;
- **RQ5**: the simulation-based validation can also be used on manual processes of validation, in which the stakeholders can observe how the SoS behaves as a whole, determining whether it complies with their needs;
- **RQ6**: although it is not possible to predict emergent behaviors, we found that it is possible to verify whether an expected emergent behavior is present or not, this

♯	**Contribution**
1	Model-based refinement methodology for SoS architecture
2	mKAOS to SosADL mapping mechanism
3	Simulator of SosADL
4	mKAOS formalism
5	Partial validation mechanism
6	Verification mechanism using PlasmaLab
7	mKAOS textual editor
8	Graphical editor for SosADL

Table 1: Contributions of this work

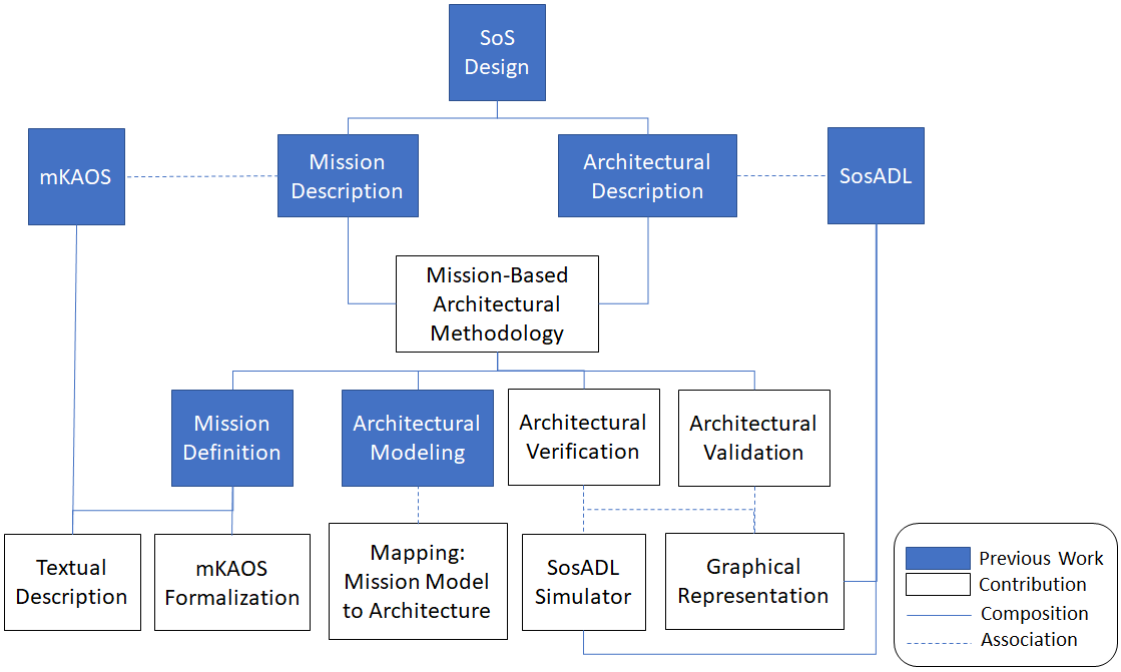


Figure 1: Overview of the contributions

can be automatically checked using the verification mechanism we propose, once the emergent behavior is formally described.

Along the path to answer these research questions, we propose a set of enhancements for the two modeling languages we decided to work with: mKAOS and SosADL. Altogether, these contributions compose a mission-based methodology to design software architectures of SoS. Table 1 summarizes the main contributions of this work, although some additional minor improvements might be found along the manuscript. Figure 1 shows an overview of the contributions, relating it with existing works.

The main contribution of this work is a **pioneer model-based refinement methodology to generate and validate architecture descriptions in SoSADL based on mKAOS mission models**. The generated architecture descriptions are partial in the

sense that they only encompass the structural definition of the involved constituent systems and its topology, and the architect must introduce the behavioral definition of the elements. Whenever the abstract architecture is enhanced with behavior, the methodology provides mechanisms to validation and verification. Further, we implemented a set of tools that partially automatize the process and its steps.

Similarly to the existing approaches for deriving software architectures from requirements, such as KAOS (LAMSWEERDE; LETIER, 2004; LAMSWEERDE, 2001), the proposed methodology relies on a top-down approach that **allows producing SoS software architectures based on a high-level description of the constituent systems**. Such methodology includes a mapping process that takes mKAOS models and partially generates SosADL models with the architecture's topology. Such mapping process ensures traceability between the mission and architectural models as it is based on a model transformation, thereby enabling architects to precisely identify which pieces of the software architecture are responsible for each mission.

The contributions of this work also include a **simulation mechanism for SosADL**, allowing the architect to evaluate the SoS within a controlled environment. This simulation mechanism allow the architect to control the execution, step by step, introducing stimulus or data at will.

Since the simulation mechanism is based on concrete architecture models, our methodology uses a mechanism developed in our research group to derive concrete architectures from abstract architectures. This mechanism consists on producing all possible concrete architectures that conforms to the abstract architecture, given a set of available systems. A generated concrete architecture is used along the methodology, for validation and verification.

Regarding verification and validation, essentially, we propose the formalization mKAOS (SILVA; BATISTA; OQUENDO, 2015), allowing the formal definition of missions, emergent behaviors, and SoS properties or constraints. Since the language from which mKAOS inherits of (KAOS (LAMSWEERDE, 2009)) is formally grounded in Linear Temporal Logic (LTL), we propose the use of the same kind of logic to mKAOS constructs. We adopted DynBLTL (QUILBEUF et al., 2016; CAVALCANTE, 2016), an extension of LTL for dynamic systems that showed promising as an hidden formalism. Using the formal definition of missions and emergent behaviors, we are able to use the SosADL simulator to verify the compliance of an architecture within the SoS properties using a simulation-based process through PlasmaLab (LEGAY; SEDWARDS, 2014).

Finally, the verification mechanism based on PlasmaLab can be used to automatically detect the occurrence of the emergent behaviors and calculate mission feasibility on a given architecture, allowing the automatic validation of the architecture within the mission model. A manual validation is also supported, based on the SosADL simulator, in which stakeholders shall identify whether the SoS meet their needs.

During the evolution of this work, some publications were achieved concerning on the contributions. These publications are listed in Appendix 8.3.

Along the manuscript we also report the problems we faced, such as our attempt on using GEMOC [3] to develop our simulator for SosADL. The experience with these problems might be valuable for the next generation of researchers and groups that work on alternative approaches.

1.4 Evaluation

To evaluate our proposal we ran a case study, common to all mKAOS and SosADL approaches: the Flood Monitoring SoS (FMSoS) (HUGHES et al., 2011). This system is introduced in Chapter 2.8, since it is used along this manuscript as a running example.

Our case study encompasses the steps of the proposed methodology: (i) mission modeling, (ii) mapping to architecture, (iii) architectural behavioral modeling, (iv) verification and validation. At some points of the evaluation, we compare the proposal with alternative approaches, such as an alternative simulator for the verification mechanism.

1.5 Outline

The remainder of this document is structured as follows. Chapter 2 provides all required background fundamental to the understanding of this work, including all involved languages and the running example used to illustrate the proposal. Chapter 3 presents the contributions in the context of the involved languages: mKAOS and SosADL, also presenting a mapping mechanism between both languages. Chapter 4 presents the refinement methodology proposed by this thesis as a whole. Chapter 5 concerns on the implementation of the toolset that promotes the methodology. Chapter 6 presents an evaluation of the proposal through a case study, showing the execution of the methodology along the modeling of a system. Chapter 7 presents the current state of the art and related

[3] `http://www.gemoc.org`

works. Finally, Chapter 8 presents the final remarks: conclusions, threats to validity and limitations.

2 Background

This chapter provides detailed information regarding the concepts and languages used in this thesis. Section 2.1 presents a key concern for this work: systems-of-systems. Section 2.2 briefly introduces Software Architecture, another important concern of this work. Section 2.3 briefly explains Model-Driven Development, an approach used to partially implement the proposal of this work. Sections 2.4 and 2.5 details the two main modeling languages that will be used in this work: mKAOS and SosADL. Section 2.6 introduces linear temporal logic, which is the base for the formalisms of this work. Section 2.7 briefly discusses about statistical model checking. Finally, Section 2.8 describes the running example used in this work: the SoS Flooding Monitor.

2.1 System-of-Systems

The increasing complexity of systems demanded the need for composing existing systems into new ones, aiming to use the features from systems already deployed and under execution, and also providing new features that arise from cooperation between the involved systems. In this context, the study of systems-of-systems (MAIER, 1998) provides solutions to the system composition process. By definition, a system-of-systems (SoS) is a system composed of independent, functional constituent systems that cooperated among themselves to achieve a greater mission.

SoS differs from traditional systems since it has emergent behavior, which is a comportment that emerges from constituent systems' interactions and is only observable during cooperation. It cannot be predicted based on the capabilities of the constituent system as it features functionalities of the architecture as a whole, instead of aggregation or union of individual behaviors. In fact, an emergent behavior is observed to be *more than the sum* of the constituent systems, such as coordination on drone flocks (VASARHELYI et al., 2018), that is a consequence of individual capabilities but cannot be predicted or derived

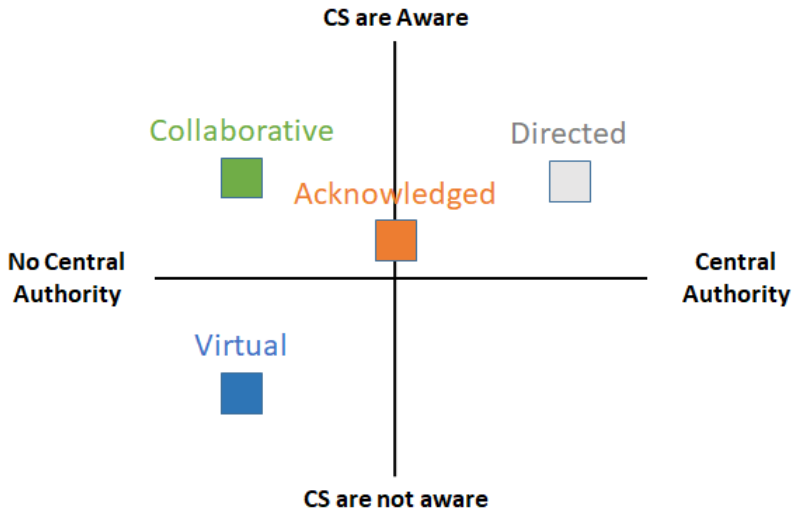


Figure 2: Types of SoS

from these capabilities. The nature of the emergent behavior makes it difficult to model and implement SoS, since many of those behaviors are not predicted at the design time, and some of them are undesirable.

Often found in the literature, the term *system-of-systems* is frequently used to refer to systems that, in fact, are not SoS. An SoS is defined by its (MAIER, 1998) (i) **geographical distribution**, meaning that the constituent systems are distributed in the physical space; (ii) **operational independence**, each constituent system is capable of achieving its own objectives and function by its own; (iii) **managerial independence**, the constituent systems might be managed by different companies with no communication between those; (iv) **evolutionary development**, the constituent systems can, and often will, evolve regardless of the SoS, meeting new requirements and configurations that matters only for the constituent system; and finally, (v) the **emergent behavior**, as aforementioned, a set of behaviors that is only observable when the constituent systems are cooperating among themselves. Furthermore, a SoS can be classified in four kinds (BOEHM; LANE, 2006): (i) *directed*; (ii) *collaborative*; (iii) *acknowledged* and (iv) *virtual*. This classification depends essentially on two factors: (i) the awareness of the constituent systems regarding their participation in an SoS, and (ii) the nature of the authority that manages the SoS. Fig. 2 plots the types of SoS in a authority versus awareness graph.

Directed SoS are systems-of-systems that are managed by a single authority that controls all the constituent systems. The constituent systems are completely aware of their participation within the SoS and often are projected and evolved aiming to better meet the needs of the SoS. This kind of SoS is the most simple to handle, since the

management authority accesses each detail of the constituent systems and can change it anytime. **Acknowledged** SoS are systems-of-systems in which the constituent systems are also aware of their participation and have a central authority, that is defined from mutual agreements between the constituent systems' managers based on recognized objectives and resources. This central authority does not have authority over constituent systems, simply providing guidance to them. In **Collaborative** SoS, all the constituent systems are also aware of their participation and work together to define protocols and contracts to fulfill central purposes. In this kind of SoS, there is no central authority and the collaboration is defined by the constituent systems individually. **Virtual** SoS, are the spontaneous SoS, i.e. SoS whose constituent system are not aware of their participation and there is no central authority. Virtual SoS are systems that are formed when constituent systems shares a common space and interact in order to achieve their own goals. The SoS missions are achieved without any acknowledge of the constituent parts and no control is possible, although some guidelines might be agreeded between the constituent systems. Due to its spontaneous nature, the current technology cannot manage virtual SoS.

When developing Directed SoS, there are not much difference from traditional systems. Since a company or organization controls everything, traditional software development approaches might be effective in this case. However, for collaborative and acknowledged SoS the reality is sightly different, specially due to the potential uncertainty that hovers the SoS, regarding constituent's behavior. Since there might be constituents with unknown behavior, designing these kinds of system with traditional approaches is potentially ineffective. Most of these traditional approaches uses modeling, validation and verification techniques that rely on the behavior of the elements, with an unknown behavior, the results are inconclusive. Therefore, this work focuses on collaborative and acknowledged SoS, in which solutions for modeling, validating and verifying are limited.

An essential concept in the SoS context is **Mission**. In SoS, a mission is a functional objective or feature the system must achieve or provide (SILVA et al., 2014). It can be classified in two types: **individual mission** and **global mission**. An individual mission is a mission that is assigned to a constituent system, which is responsible for achieving it by its own. A global mission, in the other hand, is assigned to the SoS as a whole and cannot be achieved without cooperation between its constituent systems. By definition, no constituent system is able to achieve a global mission by its own.

Missions are closely related to requirements, in the sense that the SoS are designed to achieve it. However, differently from requirements, missions are more related to the

runtime and implementation than to design and might have a priority. Thus, it is not possible to decide if a SoS achieves a mission by design and the SoS may fail to achieve a given mission or choose to achieve a more important mission. Since the dynamic nature of the SoS, global missions might often fail during reconfiguration processes.

2.2 Software Architecture

Software Architecture (GARLAN; SHAW, 1994; PERRY; WOLF, 1992) is a sub-domain of Software Engineering that concerns on the organization of software systems. It consists on designing high-level structures and describing how those structures are related to each other, abstracting some implementation aspects. The main objective is to reason about a system model and solve problems at this level, taking complex and important decisions in an early stage of development. A software architecture is intended to ease communication between the stakeholders, by providing a clear, simple language that can be used across many stages of development.

Essentially, a software architecture is composed of an homonymous document that describes the system in terms of **components** and **connectors**. Components are high-level elements that represent any piece of the system responsible for producing or consuming data, for doing so, components have their interfaces, usually called **ports**. Connectors are communication elements: they carry data from one place to another. As components, connectors have an interface, usually called **role**. Another fundamental part of a software architecture is the **configuration**, that specifies how the components will interact with each other through connectors. Besides structure, software architecture might also concern on other aspects of the system, such as behavior (MAGEE; KRAMER; GIANNAKOPOULOU, 1999) and deployment environment (MIKIC-RAKIC; MEDVIDOVIC, 2002).

Further information can be associated to the software architecture document, the so-called **architectural model**, such as properties to be fulfilled by some component, connectors or configuration itself. The structure of the document promotes modularization and reuse of components, which quality can be measured by objective criteria, such as number of dependencies.

In this context, an **Architectural Description Language** (ADL) is a domain-specific language defined to support the definition of architectural models. An ADL allows the description of all elements of a software architecture and might provide some additional mechanism depending on the domain it is intended for. Most of the ADLs

are designed for a specific domain, such as AADL, that is directed to avionic systems, and Rapide (LUCKHAM, 1996), specific for distributed systems. However, there are some ADLs for general use, such as xADL (DASHOFY; HOEK; TAYLOR, 2001) or SysADL (LEITE; OQUENDO; BATISTA, 2013) and also extensible ADLs such as Acme (GARLAN; MONROE; WILE, 1997).

Although there is no consensus regarding which ADL to use, it is accepted that ADLs must provide first order elements to represent the main concerns of software architecture and might provide additional elements for domain-specific concerns. In this context, most ADLs are semi-formal languages, providing more flexibility to the architect. However, formal ADLs are gaining attention since they allow automatic checking of properties at design time, increasing the degree of confidence of the model.

Checking properties of an architectural model is an important step of the architectural model. It is fundamental to maintain some quality attributes of software architecture (IEC61508-3, 2010; ISO/IEC9126, 1995). In one hand, verification (IEEE ISO 1012-2004, 2005) consists in checking whether an architecture satisfies a set of properties. These properties can be checked even with an incomplete architecture and it is expected to the model to maintain its properties during evolution. In formal languages, these properties can be described using some formalism and the verification might be automatic, performed by some model checker.

On the other hand, it is fundamental to validate the system's architecture (IEEE ISO 1012-2004, 2005). The process of validating an architecture consists of checking whether the architecture does what it is supposed to, therefore it is usually performed at the end of the modeling stage. Usually, validation techniques consists in identifying which elements implements each requirement (KUMAR, 2016). Often, the architecture is only validated at runtime, after all steps of implementation of the system. However, some initiatives suggests an early, continuous validation of the architecture (GOLDSTEIN; SEGALL, 2015), still at design time. For doing so, the architecture must be capable of being simulated, to allow the architect to observe how it behaves.

2.3 Model-Driven Development

Among the issues of developing software, maintaining documentation is certainly one of the most challenging and stressing tasks. Specially software models, among them the software architecture model, often differs from the implementation and some solutions pro-

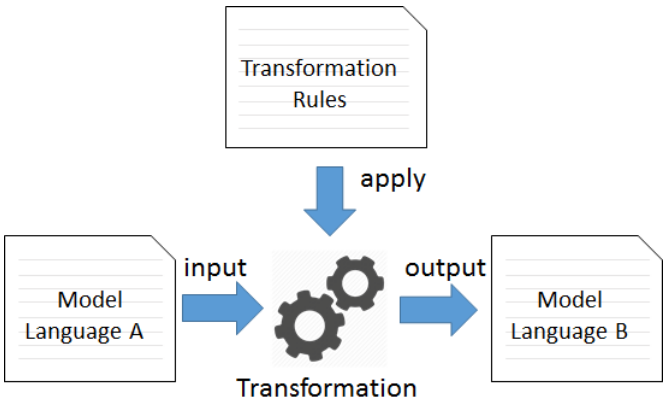


Figure 3: Model-to-model transformation

jected at design time might be lost during the implementation process. To minimize this problem, model-driven development (MDD) (VöLTER T. STAHL; HELSEN., 2006) proposes a visible change of perspective, promoting a model-level problem solving. The approach relies on running a set of automated models transformation to ensure traceability and minimize translation errors. These transformations are usually from one language to another, and might be used in a refinement process, refining a coarse-grain model to a fine-grain model.

An important MDD concept is the so-called **model-to-model transformations** (M2M) . It consists in mapping elements of two languages, based on the meta-models of the involved languages. Fig. 3 illustrates the transformation mechanism, that takes a model in language A and applies a set of transformation rules to produce a model in language B.

There are several tools that provide M2M mechanisms, among them: ATL (ATL, Eclipse.org,) and QVT (QVT, Eclipse.org,). ATL is a model transformation toolkit, with an homonymous language. The toolkit includes the language implementation, an engine to run the transformation and test mechanisms. On the other hand, QVT Operational is also a powerful transformation language, and an OMG standard, part of the QVT toolkit. Although both tools are similar, ATL documentation and community is larger than QVT, thus, we choose ATL for our implementation.

MDD promotes a development methodology that consists in describing software through coarse-grain models and apply several M2M transformations to obtain a fine-grain model. The transformation, that ensures traceability, might involve several kinds of languages, including programming languages. As the mapping is complete, the fine-grain model will certainly reflect all coarse-grain decisions and solutions.

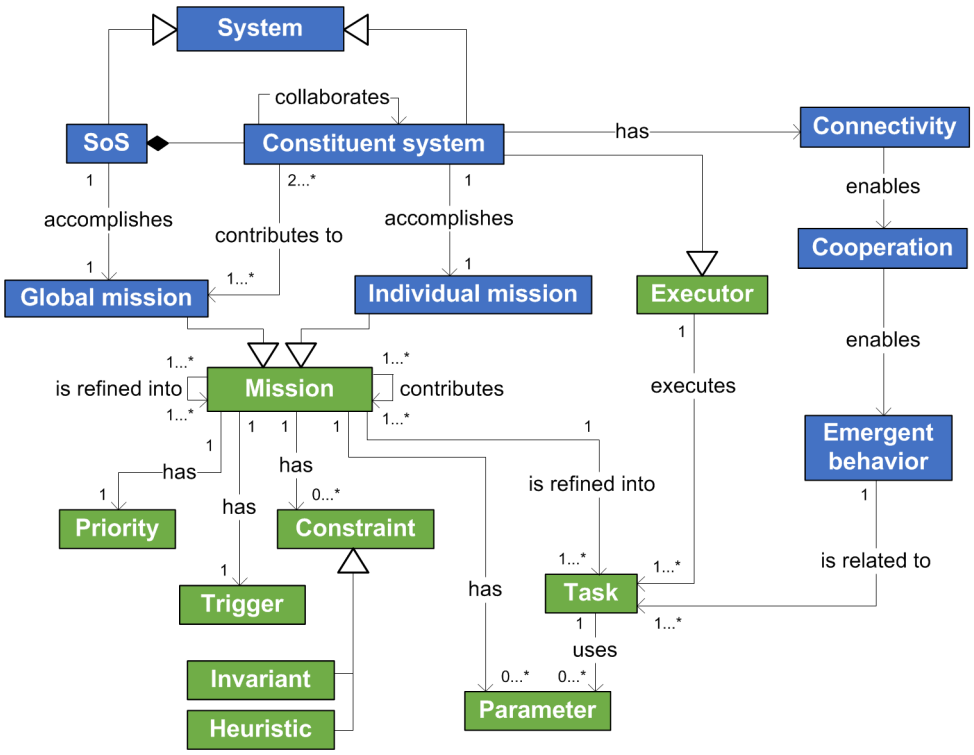


Figure 4: Conceptual Model for Missions in SoS

2.4 mKAOS

In a previous work, we conducted a literature review (SILVA et al., 2014) and proposed a conceptual model for missions in SoS. This conceptual model encompasses specific elements for the SoS domain and relates those with missions. Figure 4 presents the conceptual model in which mKAOS relies on. Its basic unit is the **System**, that may be the **SoS** itself or a **Constituent System**. An SoS is a composition of Constituent Systems. The central element of the model is the **Mission**, that is specialized as **Global**, assigned to the SoS, or **Individual**, assigned to the Constituent Systems. A mission encompasses a set of five elements: (i) **Priority**, that defines the commitment degree of the system with the mission; (ii) a **Trigger**, that defines the circumstances in which the system will pursuit the achievement of the mission; (iii) **Constraint**, in form of **Invariants** and **Heuristic**; (iv) a set of **Parameters**, data that the mission will use or produce as executed; and (v) a set of **Tasks**, operational implementations that execute a functionality. Missions can also be refined into sub-missions, and might contribute to each other.

We identified that the KAOS (DARIMONT; LAMSWEERDE, 1996; LAMSWEERDE; LETIER, 2004; LAMSWEERDE, 2001) language supports several concepts involved in this conceptual model. However, as the language uses requirements of basic unit, an extension is needed to

Conceptual Element	mKAOS Element
SoS	System
Constituent System	Constituent System
Global Mission	Mission
Individual Mission	Mission (leaf)
Priority	Mission attribute
Trigger	Mission attribute
Invariant	Domain Invariant
Heuristic	Domain Hypothesis
Task	Operational Capability
Parameter	Entity + Links
Emergent Behavior	Emergent Behavior
Connectivity	Input/Output Link
Cooperation	Communicational Capability

Table 2: Relation between mKAOS elements and conceptual model

properly represent all the concepts from the model. As a design choice, this extension will not handle implementation details, focusing on the description of mission and constituent systems in terms of tasks, that we abstracted to capabilities.

mKAOS (SILVA; BATISTA; OQUENDO, 2015; SILVA; BATISTA; CAVALCANTE, 2015) is a specialization of KAOS, a requirements engineering language and methodology. The basic elements defined in KAOS are: goals, requirements, conflicts, obstacles, and expectations. KAOS' methodology uses a set of diagrams to ensure that a requirement has at least one operational function implementing it. Due to the existing similarity between the elements defined in KAOS and the ones required to represent mission-related information, mKAOS was derived from such a language aiming at supporting mission modeling in SoS. mKAOS takes advantage of most properties of KAOS, such as its philosophy in terms of separating models according to their respective concerns and overlapping them to have a cross-view of the system. Besides specializing some concepts defined in KAOS, mKAOS creates specific elements suited to the SoS context, such as emergent behaviors and missions. An SoS can be described in mKAOS through six different models, each one with its own syntax and semantics.

Table 2 relates mKAOS elements with the conceptual model's elements. All the elements have its representation in the language, although in some cases we choose to implement a more abstract concept, in order to avoid detailing the implementation.

The main mKAOS model is the **Mission Model**, which describes missions and expectations. The **Responsibility Model** concerns the description of both constituent systems, environment agents, and the assignment of missions/expectations to them. The

mKAOS Model	Model Elements
Mission Model	Mission, expectation
Responsibility Model	Constituent system, environment agent
Object Model	Entity, event, domain hypothesis, domain invariant
Operational Capability Model	Operational capability
Communicational Capability Model	Communicational capability
Emergent Behavior Model	Emergent behavior

Table 3: mKAOS models and respective elements

Object Model specifies objects used by the system for data exchange and physical structures in terms of: (i) **entities**, which represent a data abstraction or physical entity; (ii) **events** that can be handled by the systems; (iii) **domain hypothesis**, desirable features of the system, defined as constraints; and (iv) **domain invariants**, which are constraints that must hold during the whole system execution and further evolution. mKAOS also provides two Capability Models: the **Operational Capability Model** defines a set of operations that each constituent system is able to execute, i.e., their operational capabilities, whereas the **Communicational Capability Model** specifies the possible interactions and cooperation among constituent systems, the so-called communicational capabilities. Finally, the Emergent Behavior Model describes emergent behaviors, specific features that are produced from the interaction between at least two constituent systems. Table 3 summarizes the elements of the mKAOS models.

The **Mission Model** follows a tree structure in which leaf nodes represent individual missions and non-leaf nodes represent global missions, respectively assigned to constituent systems and to the SoS as a whole. In this model, **expectations** represent objectives external to an SoS and that might influence the achievement of its missions. **Refinement** links establish a refinement relationship among missions, so that a given mission can be refined into other sub-missions and/or expectations. The assignment of missions to constituent systems is defined in a corresponding mKAOS Responsibility Model, in which each constituent system must have at least one assigned individual mission and each individual mission must be assigned to exactly one constituent system. In turn, expectations must be assigned to environment agents, which are external agents that somehow interfere on the system. Fig. 5 depicts the overlapping of a **Mission Model** and a **Responsibility Model** representing missions of the flood monitoring SoS. For instance, the *Alert Citizen in Risky Areas* mission is refined into two other missions, namely *Identify Citizens in Risky Area* and *Alert Citizen*. The first one is refined into two more missions, *Calculate Risky Areas* and *Identify Citizen*. The *Identify Citizen* and *Alert Citizen* individual missions are

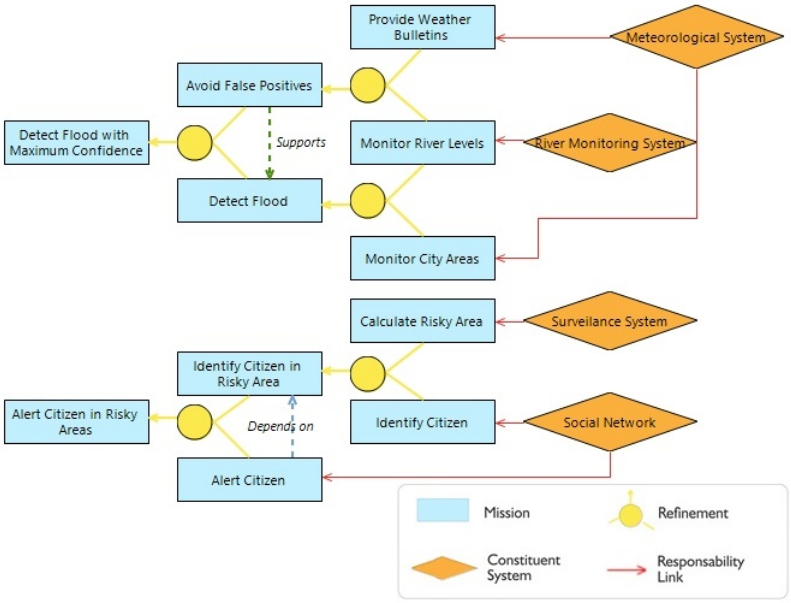


Figure 5: Example of overlapped of mKAOS' Mission Model and Responsibility Model representing missions and constituent systems of the flood monitoring SoS

assigned to the *Social Network system*, while the *Calculate Risky Area* individual mission is assigned to the *Surveillance System*. Fine-grained mission-related information can be expressed in mKAOS by using the other models available in the language.

The notation provided by mKAOS also allows defining relationships among missions. In Fig. 5, the *Alert Citizen* mission depends on the *Identify Citizen in Risky Area* mission, i.e., the first mission can only be achieved after achieving the second one. Another relationship is between the *Avoid False Positives* and *Detect Flood* missions, in which the former facilitates the achievement of the latter.

2.5 SosADL

A proper representation of SoS software architectures is quite important to the success of such systems as it can provide a basis for architectural analysis and guide their evolution. When describing SoS software architectures, it is fundamental to consider: (i) both structural and behavioral definitions for the SoS and its constituent systems; (ii) interactions among constituent systems; (iii) adaptations due to the dynamic scenarios in which an SoS operate; and (iv) properties, constraints, and quality attributes (BATISTA, 2013). To cope with these concerns, SosADL (OQUENDO, 2016a) arises as a formal language to comprehensively describe SoS software architectures while allowing for their automated, rigorous analysis. The formal foundations of SosADL rely on an extension of the π-calculus

process algebra (OQUENDO, 2016b), thereby being a universal model of computation (**??**) enhanced with SoS concerns.

One of the main characteristics of SoS software architectures is that the concrete constituent systems that will be part of the system are partially known or even unknown at design time. For this reason, these systems need to be bound dynamically, thereby making an SoS software architecture concretized only at runtime. To cope with this requirement, SosADL allows the description of SoS software architectures in an intentional, abstract way. This means that the architecture description expresses only the types of constituent systems required to accomplish the missions of the SoS as a whole (at design-time), but the concrete systems themselves will be identified and evolutionarily incorporated into the SoS at runtime. Furthermore, the communication among constituent systems is said to be mediated in the sense that it is not solely restricted to communication (as in traditional systems), but it also allows for coordination.

SosADL uses a set of eleven elements, namely: (i) systems; (ii) gates; (iii) connections; (iv) assumptions; (v) guarantees; (vi) properties; (vii) behavior; (viii) mediators; (ix) duties; (x) coalitions; and (xi) bindings. Despite possible similarities with respect to the elements defined in traditional ADLs, the concepts defined in SosADL are aligned with the terminology adopted in the literature about SoS to fit the semantics required in SoS software architectures.

The system concept is an abstract representation of a constituent system that may be part of the SoS, but that is not under its control due to its operational and managerial independences. A system encompasses **gates**, **assumption**, **guarantees**, **properties**, and **an internal behavior describing its mission**. A **gate** groups interaction points of a system with its environment, encompassing at least one connection. A **connection** is a typed communication channel through which the system sends or receives data. **Assumptions** express properties expected by a gate of a system to be satisfied by the environment, e.g., rules related to provided/required data in gates. **Guarantees** describe properties that must be enforced by the system, thereby being a way of representing specific requirements at the architectural level. A **behavior** represents the functional capabilities of the system and how it interacts with the environment by sending/receiving data. The formally founded constructs for expressing behavior in SosADL are similar to the ones defined in π-ADL (OQUENDO, 2004), another ADL based on π-calculus for formally describing dynamic software architectures of traditional systems under both structural and behavioral viewpoints. Fig. 6 shows a partial example of a system described in SosADL.

```
system Gateway(lps: Coordinate) is {
   …
  gate notification is {
    connection measure is in{MeasureData}
    connection alert is out{MeasureData}
      // alert sent by the gateway
  } guarantee {
     protocol notificationpact is {
        repeat {
          via notification::measure receive any
          repeat {anyaction}
        }
        via notification::alert send any
      }
  }
```

Figure 6: Partial example of a system described in SosADL

The *Gateway* system has a gate called *notification*, which is composed of two connections, *measure* (for receiving data) and *alert* (for sending data). The guarantee for this system defines a protocol stating that the gate receives values via the *measure* input connection and sends values via the *alert* output connection. These actions are performed repeatedly, as expressed by the repeat construct.

In SosADL, a **mediator** is an architectural element under control of the SoS and mediates the communication and coordination among constituent systems, thus also promoting interoperability among them. Mediators differ from traditional connectors as they are used not only as mere communication channels, but also as elements responsible for the coordination among the interacting constituent systems (ISSARNY; BENNACEUR, 2013). Therefore, mediators play a role in terms of allowing the SoS to achieve its missions through emergent behaviors arising from such interactions. Similarly to systems, mediators can be also described abstractly, so that concrete mediators can be synthesized and deployed at runtime in order to cope with the highly dynamic environment of an SoS. A mediator definition encompasses a set of duties, which express obligations to be fulfilled by gates of constituent systems that may interact with the mediator. Moreover, a mediator allows defining assumptions, guarantees, and an internal behavior. Fig. 7 exemplifies a mediator in SosADL, with a partial textual description and an example graphical representation. A mediator is defined with a duty called replicate and a guarantee specifying that the mediator will receive a *Parameter* and simultaneously send it through both connections *destionation1* and *destination2*.

A coalition represents the SoS itself and defines how constituent systems and mediators can be temporarily arranged to compose the SoS. As systems are not under the SoS

```
Mediator Replicator(p:Parameter) is {
  …
  duty replicate is {
    connection source is in{Parameter}
    connection destination1 is out{Parameter}
    connection destination2 is out{Parameter}
  } guarantee {
    protocol senddata is {
      repeat {
        via replicate::source receive data
        via replicate::destination1 send data
        via replicate::destination2 send data
      }
    }
  }
}
```

Figure 7: Partial example of a mediator described in SosADL

control, it is necessary to specify how the mediators can be created and which systems will interact with them to define a concrete SoS. For this purpose, coalitions are composed by a set possible systems, mediators, and bindings that will be realized at runtime. A binding is the construct responsible for establishing dynamic connections between systems and mediators, in particular connections from gates to duties. Such a dynamic nature of bindings is an important aspect for SoS since it is often not possible to foresee which concrete constituent systems will be connected to the mediators at runtime.

It is important to highlight that SosADL focus on the architecture of an SoS as a whole, therefore, the individual architectures of the constituent systems are, although desirable, not mandatory in an SosADL description. This covers one important aspect of the SoS domain: the internal architectures of the constituent systems are often unavailable. The architecture of the SoS, however, strongly depends on the interfaces of each constituent system, defined in terms of gates.

2.6 Linear Temporal Logic

Linear Temporal Logic (LTL) (PNUELI, 1977; EMERSON, 1990) is a modal logic in which the statements refer to time. LTL formulae are composed of proposition variables (PV), logical operators and temporal modal operators. By default, LTL encompasses the logical operators: $\neg$, $\wedge$, $\vee$, $\Longrightarrow$ and $\Longleftrightarrow$.

Regarding temporal operators, LTL proposes the use of five operators, that are extended to seven by some authors.

1. Next(φ): the formula φ must be **true** in the next moment
2. Always(φ): the formula φ must remain **true** during all time
3. Eventually(φ): the formula φ must become **true** in the future
4. Until(φ,σ): condition φ must be **true** until σ becomes **true**, φ must become **true** at some point
5. Release(φ,σ): once φ becomes **true**. σ must be **true**. φ may never become **true**
6. Weak Until(φ,σ): similar to *Until*. σ may never become **true**
7. Strong Release(φ,σ): Similar to *Release*. φ must become **true** at some point

In LTL, a proposition is satisfied by the infinite sequence of evaluations of a formula. That may refer to future paths or states of the system, depending on the temporal modal operators.

An extension of LTL is Bounded Linear Temporal Logic (BLTL) (KAMIDE, 2012), that introduces the notion of **time bound**. In LTL, the propositions must be satisfied during an infinite time sequence, which is often hard to proof. For tackling this issue, BLTL uses *predefined subset of time t*, in which the formulae must be satisfied.

Using the time bound, it is possible to define properties that are maintained during a finite time lapse. The modal temporal operators are enhanced with this aspects, that may use **time units** or **steps** to define the duration of the bound. Using time bounds, the evaluation process of BLTL always rely on a finite set of states.

2.7 Statistical Model Checking

In software architecture, properties or constraints highly influence the design process (GIESECKE; HASSELBRING; RIEBISCH, 2007), since they are limiting factors and often restrict the available options in the decision making process. Architectural constraints typically can be classified as two kinds: (i) technical, that restricts the architecture due to technical factors such as response time or physical infrastructure; and (ii) business, which concerns on specificities of the domain of the system.

However, the most important thing about architectural properties is the possibility of verifying these properties at design-time, decreasing the cost of the implementation

process. In this context, **model checking** is typically adopted as a solution, since it allows the verification of such properties in a simple manner. Model checking (CLARKE JR.; GRUMBERG; PELED, 1999; ZHANG et al., 2012) consists on verifying a model for some predetermined properties expressed in a given notation. As a notable solution for architectural verification, model checking is essential to identify possible faults in the model at design-time, allowing an early correction of those.

Traditional model checking approaches uses the model and a set of properties as input, building a representation of the possible state of the architecture (TSAI; XU, 2000) and identifying whether any of these representations shows a constraint violation. The model is considered *correct* if no violation is found. Otherwise the model checker may present the state in which the property is violated. This approach is susceptible to the *state explosion problem* (HOLZMANN, 2002), i.e., the number of states might grow in such way that makes it impossible to analyze all possible states.

Furthermore, traditional model checking techniques have some limitations. Besides the state explosion problem, the checkers needs to be able to produce states of the architecture, which is particularly hampered by architectural dynamism. When the architecture can change at runtime, producing states may become specially expensive and some times inviable. Also, the exhaustive methods tends to be unfeasible unless the exact number of components is known in advance (QUILBEUF et al., 2016). In the SoS context, the problem becomes even more challenging due to the uncertainty regarding the constituents' behavior: as they may behave in non-deterministic manners, using exhaustive methods may become non-effective.

Alternatively, **statistical model checking** (SMC) (LEGAY; DELAHAYE; BENSALEM, 2010; LEGAY; SEDWARDS, 2014) is gaining a momentum because it provides a probabilistic, simulation-based method for verifying properties on an architecture. SMC uses one or multiple heuristics to estimate the degree of compliance of a system to a set of constraints. Instead of building all possible states, this approach builds the more probable states and rely on simulation. Instead of inferring new states based on available data, statistical model checkers use an external simulator to analyze the effect of an event on a state. Such external simulator might have unknown behavior or use non-determinism machines in its processing.

SMC relies on simulation, using a set of stochastic models derived from the architecture to calculate the probability of each bounded property to be satisfied. Using statistical analysis over the *most probable* states, statistical model checkers can calculate the com-

pliance of the model to the properties with some degree of confidence. Such degree of confidence might be predetermined or a goal for the checker, depending on the heuristic adopted.

It is important to mention that there are some other alternatives to traditional model checking that solves the *state explosion problem*, such as *Adaptive States and Data Abstraction* (DAMS et al., 1994). However, these approaches only solves one of the issues of traditional model checking in SoS context. These approaches still require the behavior of the systems to be known. Using SMC, the only requirement is that the system should be able to be simulated.

2.8 Running Example: Flood Monitoring

Floods are one of the major problems in many countries around the world (HUGHES et al., 2011; DEGROSSI; AMARAL; VASCONCELOS, 2013). In rainy seasons, this type of event can be quite devastating in urban centers traversed by rivers as they may cause material, human, and economic losses. Regardless of their magnitude, floods represent a risk and hence they must be detected as quickly as possible. This is important to ensure a better planning of the management actions required to reduce possible damages caused by the flood, e.g., defining evacuation plans, rearranging traffic in the proximities of flooded areas, and coordinating rescue actions.

In this context, an SoS can foster effective flood monitoring, support timely response from authorities, and contribute to alleviate impacts caused by floods. To achieve these purposes, such an SoS can combine information provided by multiple collaborating independent systems such as river monitoring systems and meteorological systems. Within this SoS, river monitoring systems composed of a network of sensors spread in flood-prone areas near the river can be used to monitor the river water level as an indicator of flooding. In turn, meteorological systems comprising weather stations and satellites can be used to collect and analyze atmospheric parameters (e.g., temperature, humidity, rain amount and intensity, etc.) that also serve as input to the construction of prediction models for supporting weather forecasting.

Despite these systems seem to be enough for enabling the SoS to determine the risk of a potential flood, false positives regarding a flood risk may be caused by biased sensors or other conditions on the river. Aiming at improving the accuracy of the measures collected by the sensor nodes deployed in the monitored river area, a surveillance system based on

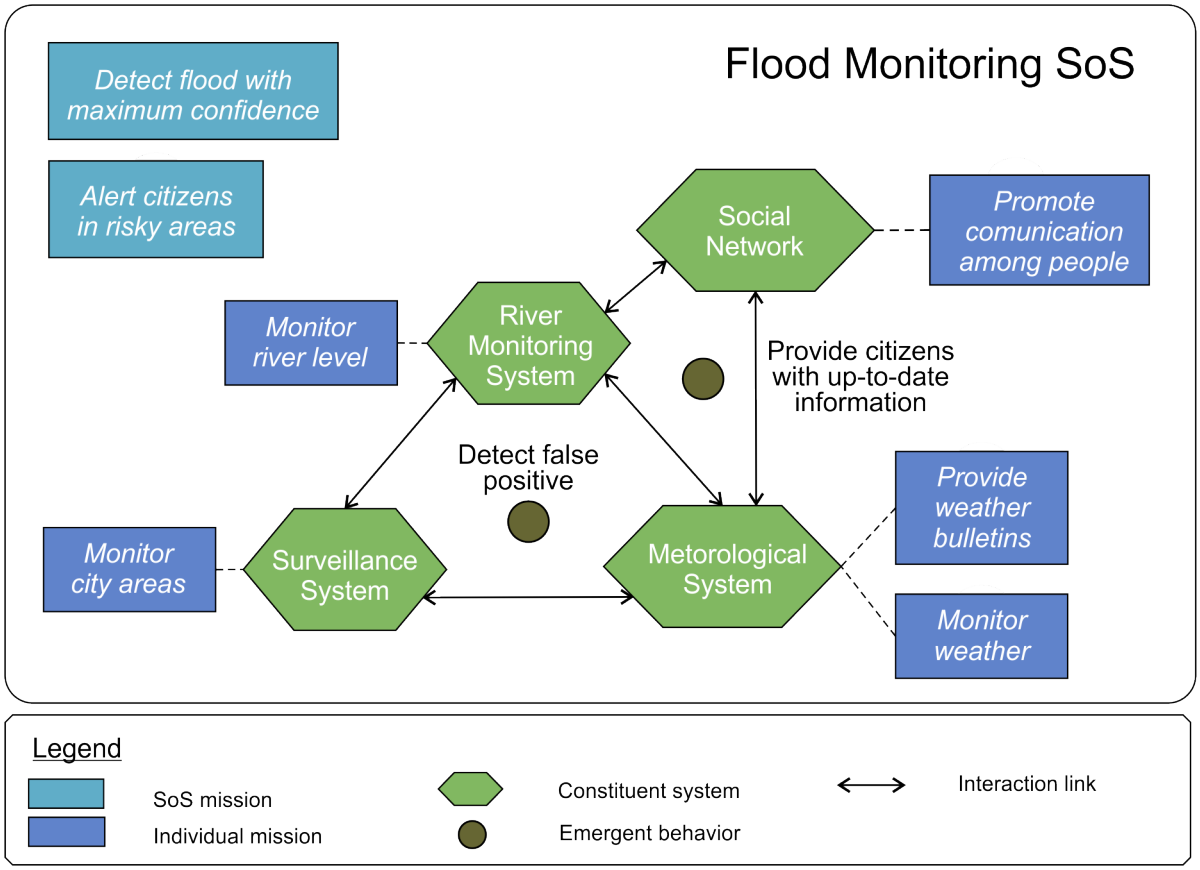


Figure 8: Constituent systems and missions of the flood monitoring SoS

the remote use of drones can be used to provide images of the river for estimating its flow rate. In this scenario, drones endowed with digital cameras can be used to record video and/or capture images of the overflown area. These multimedia data are then processed and combined with data provided by the meteorological systems and data provided by the sensor nodes spread on the river, thus contributing to detect an imminent flood with maximum confidence and avoid false positives.

Fig. 8 depicts the aforementioned constituent systems and its respective missions in the scope of the Flood Monitoring SoS (FMSoS) , whose global missions are (i) to detect flood with maximum confidence and (ii) to alert citizens in risky areas. To accomplish such missions, the river monitoring system, the surveillance system, the meteorological system, and a social network should collaborate among each other. River monitoring systems are responsible for monitoring the river level, meteorological systems can produce weather forecasts indicating future conditions, and surveillance systems are responsible for monitoring the city by recording videos and/or capturing images. Although both river monitoring and meteorological systems are able to independently emit alert messages indicating a critical condition for flooding, only the interaction between these systems allows

avoiding false positives by combining data provided by them. In addition, images provided by the surveillance systems can support the confirmation of the flood risk. Therefore, this emergent behavior resulted from the interaction among such systems enables the flood monitoring SoS to detect floods with confidence and to avoid false positives. It is worth mentioning that all of these constituent systems are operationally independent, i.e., they provide their own functionalities independently from each other and out of the scope of the SoS.

3 Enhancing mKAOS and SosADL

In order to propose a methodology to produce architectural models based on mission models, it is fundamental to enhance the mission modeling language mKAOS and develop a set of tools for the architecture description language SosADL. On one hand, we identified that mKAOS lack a inner formalism that would support the derivation of software architectures and promote verification, also supporting validation of models. On the other hand, due to the importance of simulation in the context of verification and validation, it was necessary to build a simulation engine for SosADL based on the formal semantics defined in π-calculus. This chapter focuses on these aspects of mKAOS and SosADL, providing solutions and enhancements that supports the definition of a mission-based architectural methodology.

In Section 3.1 we discuss the formalism that is needed to mKAOS to allow automatic validation and verification. As a secondary contribution, we introduce a graphical language for SosADL in Section 3.2. Further, as a simulation/execution mechanism is necessary to support validation and verification, we discuss SosADL simulation environment in Section 3.3.

3.1 mKAOS Formalism

Verifying mission-related properties is one of the goals of this work, we investigate the notation used by the mission description language, mKAOS. Since verification of models depends on the notation used to define the properties, we found a lack of formal mechanism, in mKAOS, to describe the mission-related properties.

mKAOS was designed as a simple solution for SoS mission modeling. However, mKAOS relies on several assumptions that might not be satisfied. For instance, mKAOS assumes that an emergent behavior arrives as soon as the required communicational capabilities are present in the system. This assumption is very overweening and this is a potential

point of failure of the definition language, that might compromise all approaches that uses it.

Aware of this fact, we introduced a formalism for mKAOS, based on Linear Temporal Logic (LTL) (MANNA; PNUELI, 1992) to mitigate this limitation. This formalism raises mKAOS to a formal language, from current semi-formal level, that allows the mission-related properties and emergent behaviors to be checked.

To define such formal mechanism for mKAOS, it was necessary to investigate the SoS needs in terms of logical operators. This task was already done by Oquendo et al (OQUENDO, 2016b), Cavalcante (CAVALCANTE, 2016) and Quibeuf et al. (QUILBEUF et al., 2016), in dynamic systems context. However, Oquendo's solution applies π-calculus, a process calculus and might be used as inspiration only. On the other hand, Quibeuf et al. (CAVALCANTE, 2016; QUILBEUF et al., 2016) proposed DynBLTL, a dynamic extension for BLTL (Bounded Linear Temporal Logic) that introduces a new value $\mathfrak{U}$ that represents the **undefined** value, allowing therefore the definition of transitional states in which variables and formulas have its value yet to be defined.

This Section details the formalizing process of mKAOS. In Section 3.1.1 we describe DynBLTL, the formal language we choose to introduce in mKAOS. Section 3.1.2 presents the freeze operator, a new operator we needed to introduce in DynBLTL. Section 3.1.3 describes the mKAOS grammar, produced in the formalization process.

3.1.1 DynBLTL

Verification mechanism, either using traditional model checking or not, deeply depends on the notation used by the properties language. Any method for automatic property checking implements the semantics of one or more property languages, therefore the choice of property notation depends on the required method for verification.

Aware of this fact, we decided to tackle the formal limitation of mKAOS introducing a formalism that allows a model checking technique that is adequate to SoS models. In this context, DynBLTL is a language for expressing the properties in such a manner that they can be used by SMC tools in the verification process. It allows the dynamic bound of operations, allowing the system to maintain execution states with a degree of uncertainness.

DynBLTL's main contribution is the introduction of a third value: $\mathfrak{U}$, that represents undefined or inexistent values. Grounded on a three-value logic, the language supports

```
always eventually before 40 steps {
forall r:allOfType(RiverMonitoringSystem) {
    r.warning != null implies
        exists s:allOfType(SocialNetwork) s.sendWarning == r.warning
}
}
```

Figure 9: Formal Definition in DynBLTL

the expression of properties that depends on variables or states that may not be present during some moment. With that, it is possible to express constraints without the previous knowledge of the current state of the model, allowing dynamism to be supported by the language.

The introduction of the value $\mathfrak{U}$ changes the semantic of the binary operators of BLTL:

- $\neg$ works as usual with boolean values, $\mathfrak{U}$ otherwise
- $\vee$ returns **true** if one of the operands are **true** and **false** otherwise, note that it returns **true** even if the other one is $\mathfrak{U}$. It returns $\mathfrak{U}$ if both operands are $\mathfrak{U}$
- $\wedge \equiv \neg\,(\neg\,\varphi_1 \vee \neg\,\varphi_2)$
- $\Longrightarrow\ \equiv \neg\,\varphi_1 \vee \varphi_2$

Each constraint in DynBLTL is composed of three main constructs: (i) a **quantifier**; (ii) a **temporal bound**; (iii) the **property**. The quantifier determines the variables that will be taken into account for the property, restraining the verification set. The temporal bound determines the time interval that will be considered for the property, in which the variables will be bound and the property verified. Finally, the property encompasses an expression that will be evaluated with the values within the temporal bound. A system complies with a constraint if the evaluation of its property results in **true**, under the overmentioned conditions.

Fig. 9 shows a formal definition in DynBLTL. It defines a rule that specifies that eventually in 40 steps of the system's execution [temporal bound], for each constituent system of type *RiverMonitoringSystem* [quantifier], if there is a *Warning* then there should be a constituent system *SocialNetwork* that will handle this warning [property].

For supporting a proper definition of the properties, DynBLTL also provides a set of

built-in functions that supports the exploration of architectural models. These functions are:

- **allOfAType**(type): returns a set with all components of type *type*;
- **areConnected**(a, b): returns *true* if the components a and b are connected;
- **areLinked**(a.c, b.c): returns *true* if the connection c of node a is connected to the connection c of node b;
- **lastValue**(a.c): returns the last non-undefined value of connection c of node a

3.1.2 The Freeze Operator

During our studies over DynBLTL and mKAOS an important limitation on the constraint language was detected. In fact, since DynBLTL relies on dynamic bound of variables, some values that would be necessary for some future property might be lost in the constraint definition process, due to the lack of mechanism to represent value persistence.

An example of this limitation was found on the specification of an emergent behavior for the FMSoS. This expected behavior establishes that every data produced by a **Sensor** will eventually arrive at the **RiverMonitoringSystem**. With the current version of DynBLTL, it is not possible to define a property for such behavior, therefore the tools are unable to check it.

However, this is a limitation of DynBLTL, not of temporal logics. We identified some studies on temporal logic that suggest the so-called **freeze operator** (DEMRI; SANGNIER, 2010). Such operator implements persistence on values to be bound, allowing these values to be used in future timestamps.

Since DynBLTL is designed to evaluate models that rely on stochastic mechanisms, the language focuses on the non-deterministic behavior of the systems. Therefore, storing values for future use was found unnecessary so far for introducing a degree of complexity the language was not designed to support. However, this emergent behavior of FMSoS brought the need for such operation.

As a result, the freeze operator was introduced in DynBLTL with the following semantics:

- **freeze**(var): returns the current value of *var*, that might be stored for further use

```
always during 100 time units
    forall s:allOfType(RiverSensor) {
       forall x:freeze(s.cl){
          forall rms:allOfType(RiverMonitoringSystem) {
              eventually before 100 time units (x==rms.cl)
          }
       }
```

Figure 10: Freeze Operator in DynBLTL

As an example, an application of this operator specifies the overmentioned emergent behavior. Illustrated by Fig. 10, the formal definition for the overmentioned emergent behavior defines that: each value x in that occurs in *s.c1* must eventually occur in *rms.s1* before 100 time units.

Originally, the freeze operator takes two arguments: (i) *var*, the current value of a connection; and (ii) *time*, a time bound that will define the temporal interval for which the value will be persisted. However, we decided to suppress the time bound, using the time bound of the outermost quantifier, for simplification purposes. In the example of Fig. 10, the value x would be frozen for 100 time units.

3.1.3 mKAOS Grammar

Aiming to introduce formal mechanisms in mKAOS, a set of changes was necessary. First, it was fundamental to define a textual language for the graphical representation. The grammar is based the one presented in Dardennes' work (DARDENNE; LAMSWEERDE; FICKAS, 1993), although some differences might be noticed due to mKAOS-specific constructs. The complete mKAOS' grammar is available in Appendix B.

The central element in the language, a mission, is modeled by the rule presented by Fig. 11 as an extended BNF. A mission essentially has a *name*, a *priority*, a *informal definition* (*informalDef*), a *trigger* that is expressed in terms of a DynBLTL expression. Optionally, it may have a *formal definition* that is also defined as DynBLTL formulas, and a *refinement*.

Fig. 12 shows parts of a textual description of a mission model. In this example, the mission *AlertCitizenInRiskyArea* is refined in two sub-missions: *IdentifyCitizenInRiskyArea* and *AlertCitizen*. The mission *AlertCitizen* depends on the mission *IdentifyCitizenInRiskyArea*.

```
Mission:
   ' Mission' name=ID '{'
         (links+=MissionLink (',' links+=MissionLink)*)?
         & ('resolves' resolve+=[Obstacle|ID]
         | 'conflicts' conflicts+=[Goal|ID]
         | 'concerns' concerns+=[Object|ID])*
         & ('assigned' 'to' assignedTo=[ConstituentSystem|ID])?
         & ('priority' '=' priority=INT
         & 'informalDef' '=' description=STRING
         & 'trigger' '=' trigger=expr
         & ('formalDef' '=' rule=expr)?)
         (refinement=MissionRefinement)?
'}'
```

Figure 11: Grammar rule for Mission

```
Mission AlertCitizenInRiskyArea {
  informalDef = "Alert all citizen in the area in which the
                 flood was detected"
  refinement (all) {
     Mission IdentifyCitizenInRiskyArea {
        informalDef = "Calculate the citizen in the risky area"
        refinement (all) { ... }
     }
     Mission AlertCitizen {
         dependsOn IdentifyCitizenInRiskyArea
         informalDef = "Send an alert to a set of citizens"
         formalDef = ...
     }
  }
}
```

Figure 12: Textual Description in mKAOS

```
Mission MonitorRiverLevels {
   formalDef = always eventually before 40 steps
           exists s1:allOfType(RiverSensor) s1.riverLevel != null
}
```

Figure 13: Formal Mission Description

Formal descriptions of missions are always optional. Although it is important to formally describe each individual mission, i.e. to define the circumstances in which the mission is achieved, often the required information is not available, since the constituent systems might be developed by a different team and have no documentation available.

Fig. 13 shows an example of formal definition for a mission, specifying that the mission *MonitorRiverLevels* will always be accomplished if eventually before 40 steps there exists a *RiverSensor* that is providing the river level information.

For non-individual missions (i.e.: global missions and intermediary missions), the formal definition is often unnecessary. In these cases, it is possible to formally describe how the sub-missions are related to the accomplishment of this mission, which can be done using the newly introduced **Mission Refinement**.

The Mission Refinement tackles one of the limitations of mKAOS. There was no support for the various kinds of refinements, for instance, it was not possible to define a set of sub-missions in which the achievement of some of those are sufficient for the accomplishment of the root-most mission. Previously, the refinement assumed all the sub-missions must be achieved in order to achieve the root-most mission.

We introduced new kinds of refinement to allow the representation of the various types of relations: the **mission refinement**. There are four different types of mission refinements: (i) **all**, in which the mission requires all sub-missions to be accomplished; (ii) **at least one**, in which the mission requires at least one sub-mission to be accomplished; (iii) **alternative**, in which the mission requires exactly one of the sub-missions to be accomplished; (iv) **custom**, in which the user defines a formal rule for achieving the mission based on the status of the sub-missions. Notice that, in this context, **expectations** might take place of sub-missions.

The syntactical definition of a mission refinement is presented by Fig. 14. Custom refinements encompasses a DynBLTL formula that defines the rule for the refinement.

Fig. 15 illustrates a mission refinement. In this description, we use a variation of our

```
MissionRefinement:
   'refinement' '[' (kind=MissionRefinementKind | custom=expr ) ']'
    '{'
         submissions+=Mission*
    '}'
;

enum MissionRefinementKind:
    all='all' | atLeastOne='atLeastOne' | alternative='alternative'
    | custom='custom'
;
```

Figure 14: Grammar rule for Mission Refinement

```
Mission AlertCitizen {
    informalDef = "Send an alert to a set of citizen"
    refinement (atLeastOne) {
        Mission AlertCitizenNotification { ... }
        Mission AlertCitizenSMS { ... }
        Mission AlertCitizenEmail { ... }
    }
}
```

Figure 15: Alternative Mission Refinement Example

running example that uses several kinds of alerts to the citizen in risky areas. At least one of these missions must be accomplished in order to achieve the *AlertCitizen* mission.

To introduce DynBLTL constructs in mKAOS, we choose few elements that might be formally described. All these elements received a *formalDef* attribute, that consists on a DynBLTL formal description. Besides missions, the *formalDef* attribute was introduced into the following elements: (i) **Emergent behavior**; (ii) **Domain Invariant**; and (iii) **Domain Hypothesis**. Fig. 16 shows a partial syntax for constraints in mKAOS (Domain Invariant and Domain Hypothesis), that can be used to define mission-related properties.

Emergent behaviors can also be formally described using DynBLTL formulas. The formal description of an emergent behavior allows the automatic detection of such behaviors when they are expected. Fig. 17 presents the syntax of the emergent behavior in mKAOS, that encompasses a *name*, an *informal def*, a set of *emergence links* that refers to the communicational capabilities that are involved in the behavior and the *formalDef*.

Fig. 18 specifies a expected, desirable emergent behavior that emerges from the interaction between pair of *Sensors*, a sub-systems of *RiverMonitoringSystem*: the com-

```
DomainHypothesis:
  'DomainHypothesis' name=ID
  '{'
    (...)
    'formalDef' '=' rule=expr
  '}'
;

DomainInvariant:
  'DomainHypothesis' name=ID
  '{'
    (...)
    'formalDef' '=' rule=expr
  '}'
;
```

Figure 16: Grammar rule for Domain Invariant and Hypothesis

```
EmergentBehavior:
  'EmergentBehavior' name=ID '{'
    (('informalDef' '=' informal=STRING)?
    & ('formalDef' '=' formal=expr)?
    & 'emergesFrom' emerge+=EmergeLink (',' emerge+=EmergeLink)*)
  '}'
;
```

Figure 17: Grammar rule for Emergent Behavior

municational capability *SensorDataForward*. In this system, a set of sensors is disposed over a river and might use its own communication mechanisms to forward messages from other sensors, avoiding the need of gateways, routers or other communication components. Therefore, a possible and desired behavior is that every information sent by any of the sensors can eventually arrive at a controller, since all the sensors are connected, as a requirement of *RiverMonitoringSystem*.

Finally, the *Domain Invariant* and *Domain Hypothesis* elements have the *formalDef* attribute as mandatory. In fact, we changed the definition mechanism of these elements to consists essentially of the formal definitions using DynBLTL's syntax. Since these elements can be related to any object or capability of mKAOS, the extension of the formalization covers the whole language.

3.2 SosADL Graphical Representation

One of the limitations of SosADL was the lack of a graphical representation for architectural models. Without this representation, the architectural process in the language was harder and more susceptible to human error, since the architect would have to cre-

```
EmergentBehavior SensorsDataTransmission {
    informalDef = "All data from the sensors will
                    eventually arrive at a controller"
    formalDef = always during 100 time units
        forall s:allOfType(RiverSensor) {
            forall x:freeze(s.cl){
                forall rms:allOfType(RiverMonitoringSystem) {
                    eventually before 100 time units (x==rms.cl)
                }
            }
    emergesFrom SensorDataForward[2..*]
}
```

Figure 18: Formal Definition for Emergent Behavior

ate a mental representation for the textual model being build. This impacts not only on the architectural process, but also on validation, since it hampers the identification of the topology of the architecture and therefore the identification of relations between constituent systems.

For tackling this issue, we propose a graphical representation for SosADL, using a widely used framework that is compatible with the existing implementation of the language: Sirius. Sirius is a declarative framework for defining graphical language that integrates with EMF and Xtext, supporting automatic synchronization between graphical and textual models.

Each graphical representation, in Sirius, is specified through a viewpoint that is associated to one or more file extensions. Each viewpoint encompasses a set of diagrams, that are composed by graphical element definitions. Each diagram and element definition is associated to an element in the metamodel of the language, the framework is then responsible for building the graphical representation based on these definitions and the provided model.

The SosADL graphical representation is organized into one Sirius' viewpoint, named SosADL. We developed three diagrams, two definition diagrams and one architecture diagram, to represent the concrete architecture. It is worth highlighting that the architectural models can be made in both graphical or original textual view, since the frameworks are capable of maintaining the correspondence between both views. Figure 19 show the Sir-

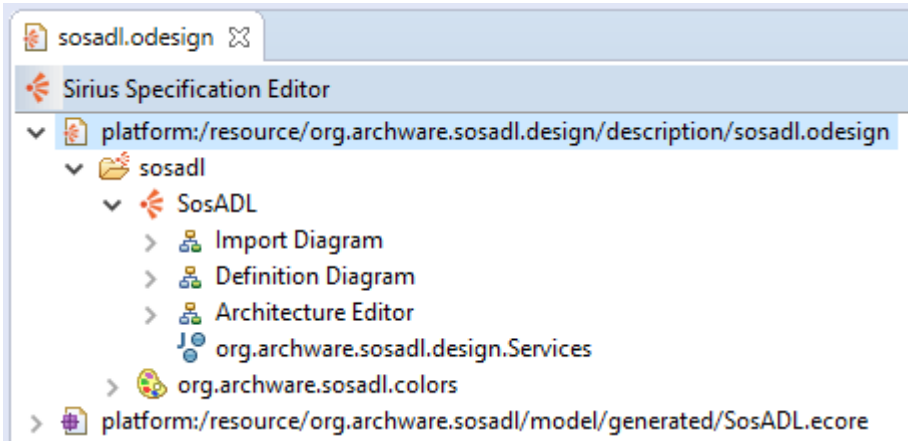


Figure 19: SosADL Sirius' Viewpoint and Diagrams Specification

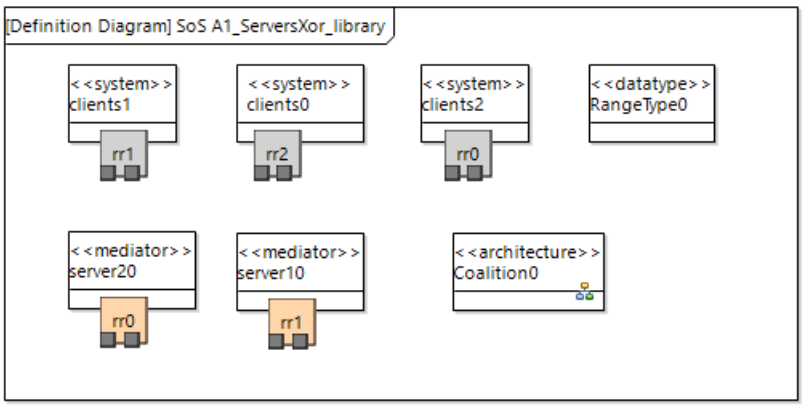


Figure 20: SosADL Definition Diagram

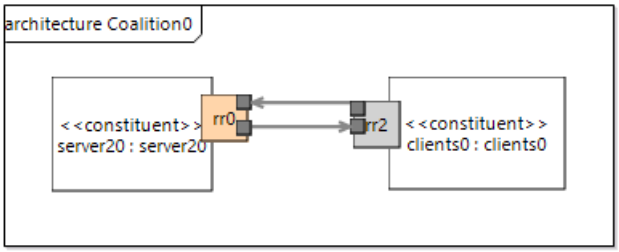
ius definition environment that specifies the viewpoint and the diagrams. This graphical specification is based only on the SosADL meta-model, which is part of the SosADL tool, generated by Xtext.

Based on the graphical specification, Sirius is capable of generating visual representation for SosADL models defined textually. It is also capable of creating new elements and maintaining changes made through the graphical editor. An **Import Diagram** is the simplest diagram in the graphical view. It represents the whole model and its imports. Since SosADL's import mechanism is not complete, this diagram is also incomplete. The **Definition Diagram** is responsible for defining the systems, mediators, gates, types, etc. Figure 20 shows an example of the client-server architecture generated automatically by Guessi's (GUESSI; OQUENDO; NAKAGAWA, 2016) tool. It defines three systems: (i) *clients1*, (ii) *clients2*, and *clients3*; and two mediators: *server20* and *server10*. The type *RangeType0* and the architecture *Coalition0* are also defined in this diagram.

Finally, the **Architecture Diagram** is responsible for representing the concrete ar-

```
architecture Coalition0() is {
    gate unusedGate0 is {
        connection unusedConnection0 is in{RangeType0}
    }
    guarantee {
        protocol allowAll is {
        repeat {
        anyaction
    }
    }
    }
    behavior main is compose {
        server20 is server20
        clients0 is clients0
    }
    binding {
        unify one {clients0::rr2::req2} to one {server20::rr0::req0} and
        unify one {server20::rr0::ack0} to one {clients0::rr2::ack2}
     }
}
```

(a)



(b)

Figure 21: Concrete Architecture in SosADL

chitectures and their topologies. Figure 21 shows the architecture diagram for *Coalition0*. Figure 21(a) presents the textual definition of the system, and (b) shows the correspondent graphical view of this architecture. Both representations were generated by the current tools.

3.3 SosADL Execution

One of the major needs of ADLs for SoS is the possibility of simulation and/or execution. Specially due to the unpredictable nature of the emergent behavior, it is key for the architect to be able to simulate the architecture to observe the behaviors that are present in a given scenario. Simulation is also key for validation, since it allows architects to observe the architecture in a controlled environment, beforehand of implementation.

In this context, SosADL was designed aiming to allow formal analysis and also simulation, with constructs that can only be tested on simulation environments, such as the *mediator*. Therefore, a simulation mechanism is crucial to a design process that involves the ADL. Such mechanism would allow the architect to foresee unpredicted emergent

behaviors, but would also to support the validation process.

However, some considerations are necessary before we start discussing execution/simulation of SosADL models. First of all, SosADL supports modeling of both abstract and concrete architectures, hence, it is fundamental to identify the differences between those kinds of models.

In SosADL, concrete architectures represent a SoS in the context it will be deployed, and abstract architectures represents a group or family of SoS. Hence, concrete architectures should not be executed as an specific architecture. Therefore, concrete architectures are those that must be executed. Guessi et al. (GUESSI; OQUENDO; NAKAGAWA, 2016) worked with the ArchWare team in this context, in which the feasibility of an abstract model is tested through exhaustive generation of concrete architectures. We decided to use her solution to produce concrete architectures. Guessi's solution is further discussed in Section 4.3.1.1.

With the clarification of which model we shall work with, we identified a study that proposes a simulation based on model transformation. Such approach, proposed by Graciano Neto (NETO, 2016) uses a transformation to DEVS (COURETAS; ZEIGLER; PATEL, 1999), an executable formalism for modeling and analyzing systems through statecharts and timed events. This work was enlightening to our proposal and is briefly discussed in Section 3.3.1.

However, the Graciano Neto's approach consists on using an external simulator based on a transformation process. We propose an evolution of such approach, that relies on an integrated simulator for SosADL models.

For proposing so, we identified the SosADL execution semantics, that is presented in Section 3.3.2. The implementation of this semantics in a simulator did not came from a first shot. Our attempts are presented in Sections 3.3.3 and 3.3.4. The first used GEMOC (COMBEMALE; BARAIS; WORTMANN, 2017), an emerging framework for model execution, and did not succeed. However, the lessons learned from this experience were valuable to the later: a model simulator made from scratch over SosADL tools.

3.3.1 Execution through Model-Transformation

Executing SosADL is an under-development feature of the language. Graciano Neto (NETO, 2016; NETO et al., 2018) proposes an execution mechanism for SosADL based on MDD. The approach uses DEVS (COURETAS; ZEIGLER; PATEL, 1999), an executable

SosADL Concept	SosADL	DEVS
Connection	Connection Declaration	DEVS Port
Constituent System	System Declaration	Atomic Model
Data Types	Data Type Declaration	Data Type
Gate	Gate Declaration	DEVS Port
Mediator	Mediator Declaration	Atomic Model
Architecture	Coalition	Coupled Mode

Table 4: SosADL to DEVS Mapping

formalism for modeling and analyzing systems through statecharts and timed events. SosADL models are mapped to DEVS models using a simple MDD instrument, then the produced DEVS model can be executed in specific tools, such as MS4ME (MS4 Systems,). It is important to highlight that Graciano Neto's solution was developed simultaneously to this work and might present some similarities, since both works were produced by the same research team.

Since this proposal relies on model transformation, it is based on a direct mapping identified by the authors. Since both SosADL and DEVS rely on rigorous formalizations, this mapping process preserves the concepts in which the languages are grounded (NETO, 2016).

The mapping process is divided in two steps: (i) the generation of atomic models, and (ii) generation of coupled models. The first step consists essentially in the automatic transformation, that was made using Xtend[1] and Xtext. The elements are transformed using rules based on the correspondence Table 4. The only exception is the coupled mode, that is generated by the second step. The second step requires some processing, and calculates the transitions based on the dynamism and *unify* relations of the SosADL models. After the production of the DEVS model, the model can be executed and analyzed.

Although functional and efficient, even in large scale systems, due to the efficiency of all tools used in the process, the simulation through this method requires some effort from the user. It is necessary to build the SosADL model, transform it to DEVS, execute in MS4ME, and track the results back to the architecture. Therefore using this process to validate systems in constant evolution may be expensive, for requiring several transformation processes and use of multiple tools.

The main issue, however, regards model checking. As we previously discussed, *Statistical Model Checking* is more effective in SoS scenarios, due to its dynamism and behavioral uncertainty. However, SMC tools require an external simulator to execute the models and

[1] `https://www.eclipse.org/xtend/`

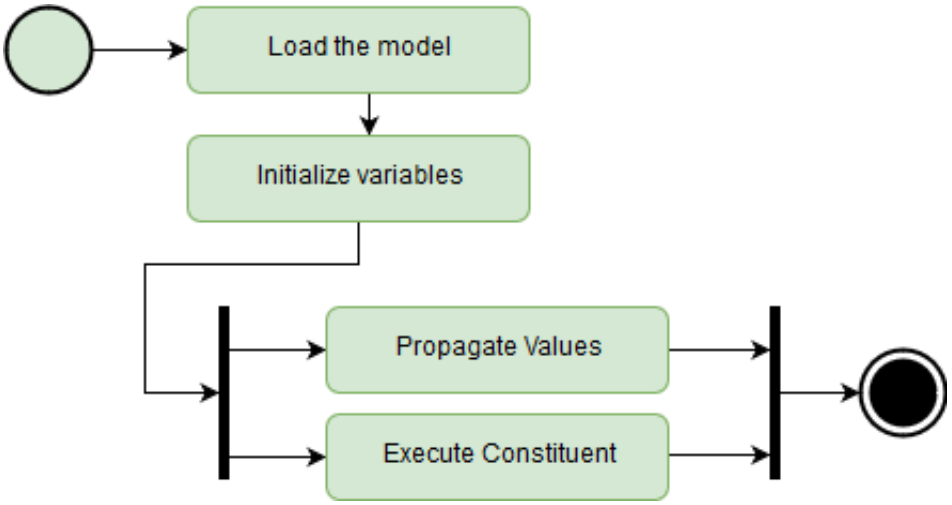


Figure 22: Activities of Execution Workflow

provide feedback of this execution. Using a transformation-based approach may impact on the effectiveness of verification, due to the potential semantic loss during the transformation process.

3.3.2 SosADL Execution Semantics

In order to implement the SosADL simulator, it was necessary to define the execution semantics we would implement. We divide the execution semantics into two scales: (i) **execution workflow**, and (ii) **specific semantics**. The general workflow controls the execution as a whole, establishing the activities that would be executed in order to simulate a SosADL model. On the other hand, the specific semantics rely on the semantics statements and expressions of SosADL, defining how each construct must behave and the impact they have on the execution.

The **execution workflow** specifies the activities the simulator must execute in order to execute the model in macro scale. This workflow is divided in 5 steps, as illustrated by Fig. 22. The first step is **load the model**, in which the simulator must load the architectural model to be executed and enhance it by allowing the *connections* to have values. The next step is **initialize variables**, that consists in initializing the values on the *connections*. Then a step **propagates the values**, must move values from one connection to another, based on the *unify* relations on the model. Simultaneously, the simulator must **execute the constituent systems and mediators**, that will be executed if the asserts are fulfilled and the necessary data is available.

It is important to highlight that the step **propagate the values**, is also responsible

for synchronization mechanisms, ensuring that a value will not be maintained or altered by two different constituents at the same time. When a value is propagated to the unified connections, the origin must be consumed and hence assume the value *empty*.

The **specific semantics** specifies how the constituent systems and mediator perform their operations. Specifically, it defines the execution semantic of the statements and constructs of SosADL. These semantics were defined by Oquendo (OQUENDO, 2016b), encompassing semantics of actions and behaviors in terms of π-calculus.

3.3.3 Execution through xDSML

Alternatively to Graciano Neto's proposal, another possibility is to implement a executable model based on xDSML (eXecutable Domain Specific Modeling Language) frameworks. Among the existing frameworks, GEMOC[2] (COMBEMALE; BARAIS; WORTMANN, 2017) is one of the pioneer projects.

Due to the relevance of GEMOC within the community, we tried to build the xDSML model of SosADL using the framework. However, the use of this approach failed due to several limitations on the GEMOC framework. Nevertheless, we report our advances and the limitations found for further use in this subsection.

GEMOC is a framework to build execution environments for modeling languages. The framework is based on widely used frameworks, such as EMF[3], Sirius[4] and Xtext[5]. It integrates various solutions to allow an easy manipulation and definition of execution environments.

A GEMOC implementation can be divided into three phases, each one is briefly described in this subsection, focusing in our implementation. The framework integrates the results of the phases to produce the execution environment. First phase is the definition of languages, that will be used by final users, this phase is described in Section 3.3.3.1. Second phase is the definition of the aspects, that described the execution semantics of the language, detailed in Section 3.3.3.2. Third phase is the extension of the language, which is optional and consists on producing a new model that encompasses not only the base language definition, but also the execution semantics defined in step three, this phase is detailed in Section 3.3.3.3. Finally, Section 3.3.3.4 presents our conclusions and learning

[2] http://www.gemoc.org
[3] `http://www.eclipse.org/modeling/emf/`
[4] `http://www.eclipse.org/sirius/`
[5] `http://www.eclipse.org/Xtext/`

from the attempt of using this framework.

3.3.3.1 Language Definition

The first phase of definition of a xDSML in GEMOC is the language definition. The framework was built to allow reuse of existing languages, which was helpful since SosADL already have a set of tools.

GEMOC is able to understand abstract models defined in EMF and concrete languages specified with Xtext and Sirius. Since SosADL already had the language definition in EMF and Xtext, and we implemented a graphical language in Sirius, the framework is able to handle SosADL models automatically.

3.3.3.2 Execution Semantics

GEMOC uses Kermeta3 (K3) [6] as action language to define the execution semantics. The framework allowed the extension of existing SosADL classes, injecting methods to some elements such as *Constituent System*, *Mediator* and the architecture itself using *aspects*.

Using K3, GEMOC requires the use of annotations to define three main methods: (i) the *@Main* method, that controls the whole execution; (ii) the *@InitializeModel* method, that is invoked once to initialize the execution model; and (iii) *@step* method, that defines a single step of the execution.

The *InitializeModel* method is responsible for implementing the two first activities of the execution workflow, previously presented in Section 3.3.2. Fig. 23 presents the implementation of such method, in which the load of the model is performed automatically by GEMOC, this methods just needs to invoke the execution semantics of *unify*.

The two remaining activities of the execution workflow are invoked in the *main* method, for parallel computing: (i) *propagate*, responsible for propagating values on the connections, based on the operations of *unify* within the architectural model; and (ii) *executeConstituents*, that verifies the capability to execute each constituent system and mediator. These methods are also defined as steps, to make it easier to use for the final user. The whole K3 aspect file is available at Appendix E

To allow a proper execution of the constituents (constituent systems and mediators),

[6] `http://diverse-project.github.io/k3/`

```
def public void init(EList<String> args) {
  for (GateDecl gate : _self.gates) {
    for (Connection c : gate.connections) {
      ConnectionAspect.value(c, Values.empty) // initialize values
    }
  }
  // unify gates
  ExpressionAspect.performAction(_self.behavior.bindings)
  println("Started")
}
```

Figure 23: K3 *InitializeModel* method

we defined modules for statements and expression interpretation. The execution semantics invokes those modules whenever necessary, providing an abstraction we named *Context*, that encompasses the current *scope* and status of constituents. The *Statement Interpreter* and *Expression Interpreter* are responsible for, based on the current *scope*, calculating the impact of each expression or statement in the execution context updating values on variables whenever necessary.

3.3.3.3 Execution Model

GEMOC does not use the original model to perform the simulation. Instead, it creates a *runtime model* based on the original model and the execution semantics. For doing so, it uses the Melange framework[7] for assembling the EMF metamodel definition and the execution semantics defined using K3.

Melange creates a new metamodel and a new set of classes that implements it, and also a set of adapters that allow automatic adaptation of the models to the newly generated runtime model.

The Melange definition of SosADL specifies which aspects (executions semantics defined in K3) will be used to produce the runtime model. Essentially, we defined aspects for each runtime-relevant element, as observed in Fig. 3.3.3.3. The aspects encompasses new abstractions of methods for *connections*, *constituent systems*, *mediators*, *expressions*, *statements* and *unifies*, describing how each of these elements are executed.

[7] `http://melange.inria.fr/`

```
language XSosadl inherits BaseSosADL {

    // Melange generates an extended ecore in a language runtime
    // project (org.archware.sosadl.gemoc.sosadl)
    // that copies everything needed for the  execution (abstract syntax and aspects),
    with sosADL.aspects.ConnectionAspect
    with sosADL.aspects.SystemDeclAspect
    with sosADL.aspects.MediatorDeclAspect
    with sosADL.aspects.ArchitectureDeclAspect
    with sosADL.aspects.ExpressionAspect
    with sosADL.aspects.StatementAspect
    with sosADL.aspects.UnifyAspect
}
```

Figure 24: Melange file for SosADL

3.3.3.4 Discussion

GEMOC proposes an easy to use framework to define the execution semantics and implement an execution environment for SosADL. Based on other frameworks such as Kermata3 and Melange, GEMOC promotes a separation of concerns that sounds outstanding for our work.

However, the framework is full of limitations. In fact, those limitations forced our team to give up on the framework, due to its current immaturity. Many of the limitations comes from Melange, but GEMOC itself also requires many interceding in the process of developing the execution environment.

We found that Melange is unable to handle external tools, that means that every method that is invoked by K3 aspects must be either in the metamodel or in the aspects itself. For SosADL, this is a major limitation, since the language encompasses an external *type checker* that is responsible for some syntax checking also. Melange was unable to generate runtime models for SosADL, unless we disabled the type checker for the execution environment, which was not possible since this type checker supports the core language. This was a major problem that was reported in `https://github.com/diverse-project/melange/issues/102`.

Also, by that time, Sirius was unable to handle the runtime model simultaneously with the original model, even with definition of additional layers. Therefore, the framework was unable to provide the runtime model in a way Sirius could understand, making it impossible to display a graphical representation of such runtime model. We are not sure whether this is a limitation of Sirius or GEMOC, since the later might be invoking the first incorrectly.

GEMOC provides a mechanism for monitoring the scope, presenting the variables and their current values. However, this mechanism is full of limitations. The most important one is that it is not possible to change the name display or filter the variables, which often becomes hard to read due to the complexity and scale of the models.

Those limitations, among other minor problems[8,9], made unfeasible to persist on the use of the framework. Instead, we chose to implement our own execution engine. Fortunately, we could use or adapt the execution semantics in K3 to pure Java code, easing the implementation process.

All the files and projects we used to implemented SosADL in GEMOC are available at `https://github.com/eduardoafs/sosadl_melange`.

3.3.4 Execution through built-in Simulator

Alternatively from the xDSML approach, we built a simulator in pure Java using the existing plug-ins to provide the necessary infrastructure. For doing so, we were able to reuse code snippets of the designed aspect in Kermetta3 and our learnings from GEMOC.

In this Section, we describe the SosADL simulator that was made using pure Java. Section 3.3.4.1 presents the requirements elicited for the simulator, and Section 3.3.4.2 details the architecture of the plug-in that implements such simulator. Finally, Section 3.3.4.3 briefly discusses the PlasmaLab connector, a key mechanism for verification.

3.3.4.1 Requirements

The SosADL simulator was build aiming for some goals, specially to support statistical model checking and validation of software architectures. That said, we elicited some requirements for the simulator, that we have described using a SysML requirements diagram.

The main requirement of SosADL simulator is **Simulate SosADL models**, presented in Fig. 25. This requirement is a composition of six other requirements: (i) **Load SosADL Models**, (ii) **Initialization of Values**, (iii) **Support Stimuli Generators**, (iv) **Control Execution**, (v) **Execute Model**, and (vi) **Monitor Activities**.

Load SosADL Models is the first requiremente that composes **Simulate SosADL**

[8] `https://github.com/diverse-project/melange/issues/106`
[9] `https://github.com/diverse-project/melange/issues/103`

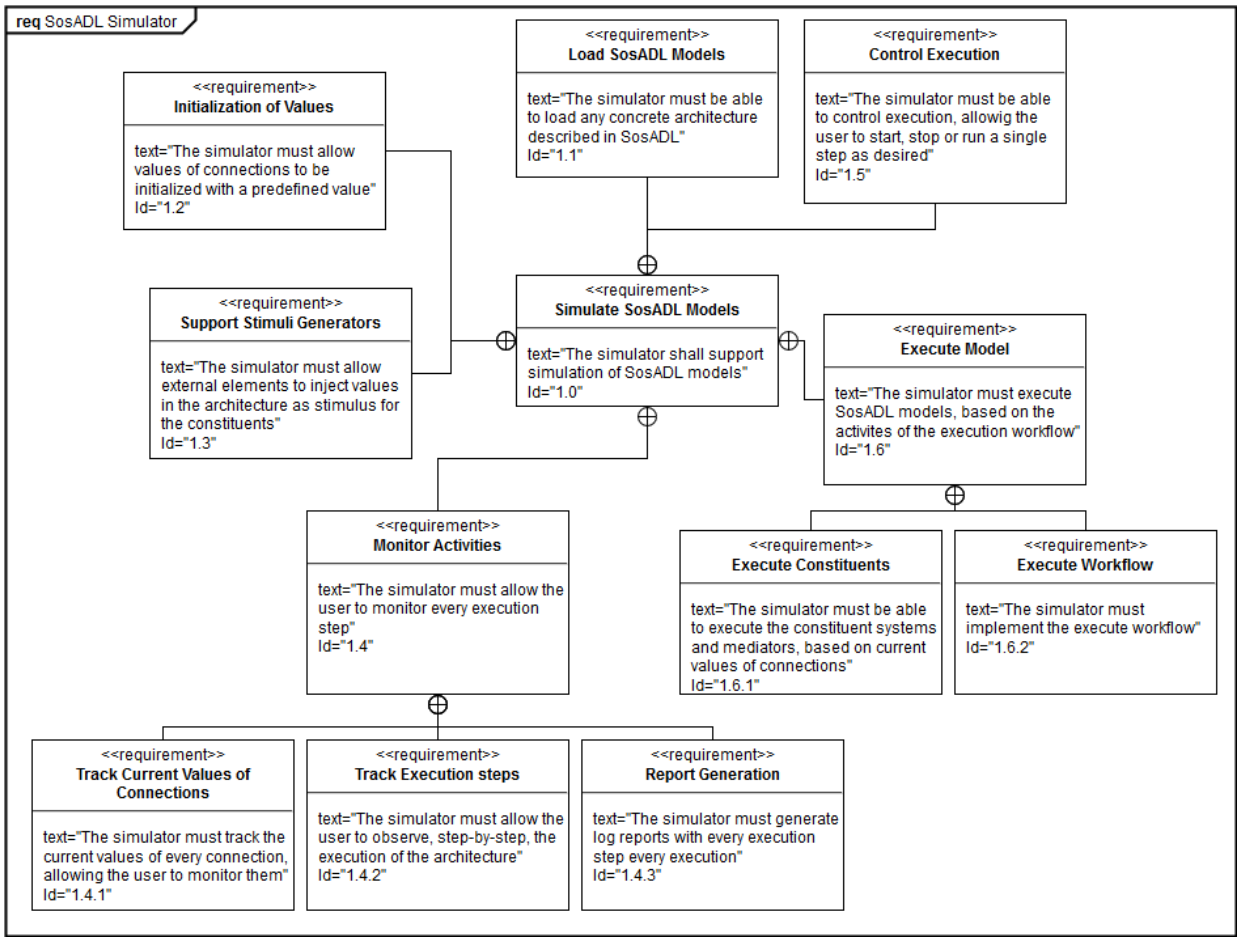


Figure 25: Simulate SosADL Models Requirement

Models, it specifies that the simulator must be able to load any existing SosADL concrete architecture. **Initialization of Values** specifies that the simulator must allow the user to predefine values that will be initialized on connections.

Stimuli generators were first introduced together with SosADL by (NETO et al., 2017), aiming to allow the user to control the environment in which the SoS is. The requirement **Support Stimuli Generators** specifies that the SosADL simulator must support this kind of mechanism, allowing the user to control the simulation environment.

The requirement **Monitor Activities** specifies that the SosADL simulator must allow the user to track every activity on the simulator, which consists of: (i) current values of connections; and (ii) execution steps. The simulator must also produce reports in form of logs, that will detail every execution step.

One of the most important requirements in SosADL simulator, **Execute Model** specifies that the simulator must be able to execute SosADL models, implementing mechanisms for executing the **execution workflow** and the constituents, using SosADL semantics.

Finally, the simulator must allow the user to **Control Execution**. The user must be able to start, stop, restart and run the simulation step-by-step at any moment.

Aside the SosADL Simulator, we propose an additional module to handle verification and validation, using PlasmaLab (LEGAY; SEDWARDS, 2014; LEGAY; SEDWARDS; TRAONOUEZ, 2016) as model checker. The so-called V&V Module bridges the SosADL Simulator and PlasmaLab to support automatic verification and model validation. The requirements diagram for V&V module and further specification details are fully available at `http://eduardoafs.github.io/m2arch`.

3.3.4.2 Simulator Architecture

Based on the overmentioned requirements, we defined an architecture for the SosADL simulator. Such architecture implements a layer-based structure, in which the layer elements can only interact with the layer immediately below. However, elements in the same layer can also interact. Fig. 26 despicts an overview of the architecture.

The components in the first layer, namely *SosADL Base Plugin* and *Value Manager*, correspond to the existing SosADL plugins and a module to control current values of objects. Over these components, in the second layer, the *Context Manager* is built. The *Context Manager* is probably the most important component in the simulator, since it associates SosADL elements (connections, variables, etc) to their current values, within a

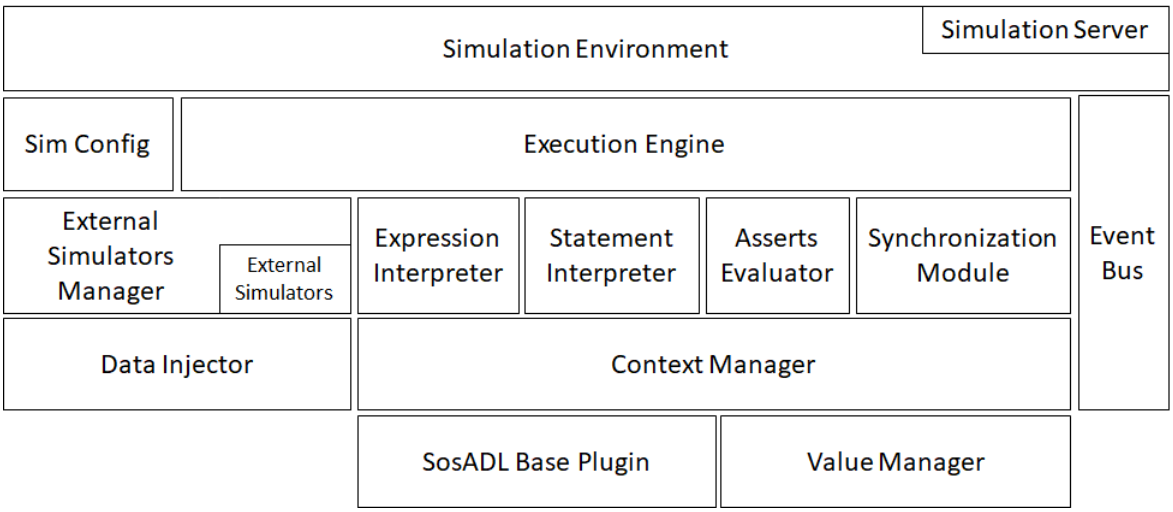


Figure 26: SosADL Simulator Architecture

structure we called *Context*. The second layer also encompasses the *Data Injector*, that is responsible for manipulating the context directly.

The third layer encompasses the components that manipulate context: (i) *Expression Interpreter*, responsible for interpreting arithmetical expressions; (ii) *Statements Interpreter*, responsible for interpreting statements; (iii) *Asserts Evaluator*, that evaluates and checks the asserts; (iv) *Synchronization Module*, that is able to lock/unlock values and controls the parallelism in the execution; and (v) *External Simulator Manager*, that is responsible for loading/unloading external controllers, which are plug-ins that are able to replace an architectural element, allowing implementation of **Stimuli Generators** that manipulate the context directly using the data injectors.

The fourth layer encompasses two elements: (i) the *Simulation Configuration Manager*, that loads **configuration files** and manipulate the external controllers, the configuration manager also contains an external controller that is responsible for directly manipulate the context according to predefined instructions; and the (ii) *Execution Engine*, that controls the whole model execution.

An *Event Manager* is a crosscutting component, that interacts with all components in the architecture, allowing the execution engine to identify precisely what happened in each level of the execution through the manipulation of **Events**. An **Event** can be a (i) *communication event*, in which a constituent or mediator provides or consumes data from another element; a (ii) *synchronization event*, in which a shared information is synchronized or locked/unlocked; (iii) *data events*, like consumption or production of new values; (iv) *structural update*, when the architecture changes for any reason; (v) *execution*

event, which refers to the execution of a constituent system or mediator; (vi) *other*, a non-specific event. The *Event Mananger* creates and organizes the events and might be used to generate simulation reports.

Finally, the *Simulation Environment* layer encompasses a single homonymous component, that provides a user interface and controls a *Simulation Server*, that will be used for *Statistical Model Checking*. Currently, the user interface only provides textual outputs, reporting the events of the simulation according to user-specified configurations.

Details regarding the implementation of SosADL Simulator are further detailed in Chapter 5.

3.3.4.3 Integration with PlasmaLab

Besides simulating SosADL models, the SosADL simulator needs to be capable of integrating with PlasmaLab, for supporting statistical model checking for verification purposes.

PlasmaLab requires a set of four requests to be handled: (i) *init*, in which the tool asks the simulator to initialize; (ii) *new trace*, that consists in requesting a new simulation to start; (iii) *new state*, that consists in the execution of a single execution step; and (iv) *end*, in which the simulation server terminates the execution.

These requests are made in a predefined order to the statistical model checking process, illustrated by Fig. 27. First, the SMC tool will request a *init* once, then requests for *new trace* will be sent eventually to start a new simulation. Once started, several *new state* requests will be made. At the end of the checking process, an *end* request will be sent.

To implement the support for these requests, we decided to implement a **Simulation Server**, PlasmaLab connector. This connector is responsible for bridging SosADL simulator and PlasmaLab, transforming the requests into commands for the simulator and translating the response into the format required by PlasmaLab.

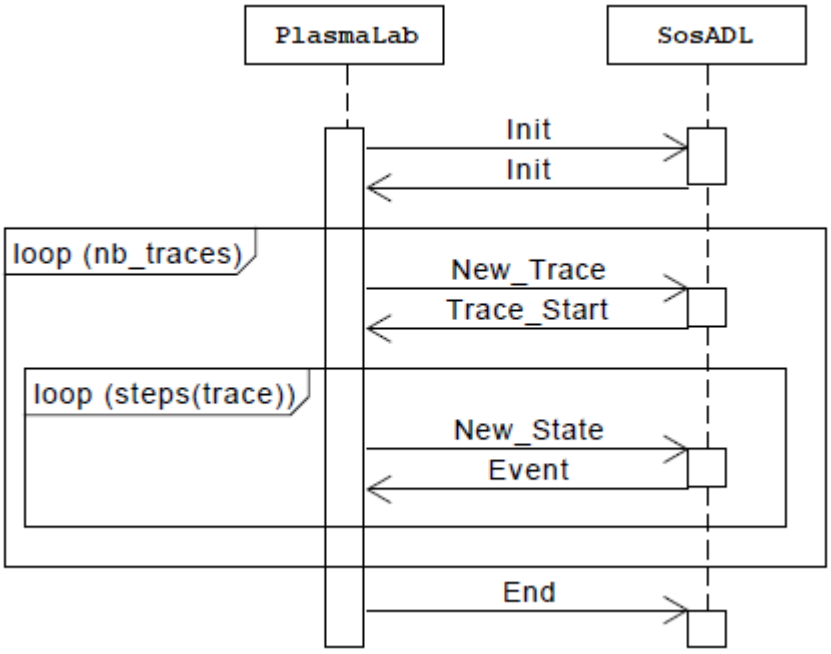


Figure 27: PlasmaLab Interaction with Simulation Server

4 M2Arch: A Mission-Based Methodology for Designing SoS Architectures

Proposing a method to support architectural description of SoS based on mission models is the main objective of this work. In this Chapter, we presents M2Arch, a method that uses mKAOS mission models to produce architectural models. This method is partially automated and encompasses the main activities of software architecture design: (i) modeling, (ii) verification and (iii) validation.

M2Arch gives **special attention to emergent behaviors** and **traceability between missions and architectural elements**. It also encompasses an automatic manner to verify the architecture for domain properties and a semi automatic validation for missions.

The outline of this Chapter is structured as follows: Section 4.1 provides an overview of the proposal, presenting the method as a whole. Each of the following Sections describe a single step of the method: Section 4.2 focuses on the first step: *definition*; Section 4.3 focuses on the properties verification; and finally, Section 4.4 describes the validation mechanism we propose.

4.1 Process Overview

Refining mission models to architectural models demands a significant effort from the architects. Aiming to systematize this process, we propose a method that uses mKAOS models as a basis to produce, in a semi-automatic manner, SosADL models.

The method for designing SoS architectures that is proposed by this work consists of a three-step process. The first step, **Definition**, consists on the definition of all involved

models: (i) the **Mission Model**, and (ii) **Architectural Model**. The development of these models are partially supported by an automatic transformation.

The **Verification** step consists on checking constraints in the derived concrete architectures, using the formalism of the involved models. The verification process is fully automated, using the tools that are associated to M2Arch. In this step, we verify domain-related properties, described in mKAOS as *Constraints*, checking the conformance of the architecture with this set of rules with a certain degree of confidence. Architecture-related properties (such as restrictions of the deployment environment or adopted technologies) can also be verified, however, we briefly describe this activity since it was not the focus of this work.

Finally, the **Validation** step uses some of the generated artifacts from the verification step to support the validation of the produced architecture. This step is semi automatic, since we are able to automatically check the emergence of the emergent behaviors and the achievability of the formally-described missions. Part of the validation, however, consists on the simulation of the architecture and interpretation of the simulation reports, that will indicate whether the system does what it is intended to do. This later activity is essentially manual, since it depends on interpretation of requirements and the stakeholder's needs.

Fig.28 depicts an overview of M2Arch. In the **Definition** step, the mission model will be defined, then submitted to an automatic transformation. Based on the artifact generated by the transformation, the abstract architectural model is produced. The **Verification** step starts with a derivation of a concrete architecture, using an automated process. This concrete architecture is the one submitted to a automated verification process, based on the constraints of the SoS. Finally, the **Validation** is divided in two phases: (i) the automatic validation, supported by our tools, consists on checking the achievability of the missions and the emergence of expected emergent behaviors; (ii) then the simulator can be executed alone, providing detailed information to the architect that can, manually, identify how the SoS behaves. At any point of verification or validation, the architect may identify adjustments to be done in the mission model, returning to the definition step.

4.2 Definition

We propose the first step of the method to be dedicated to the modeling of the missions and the architecture. The main artifacts produced in this step are the mission model and the architectural model.

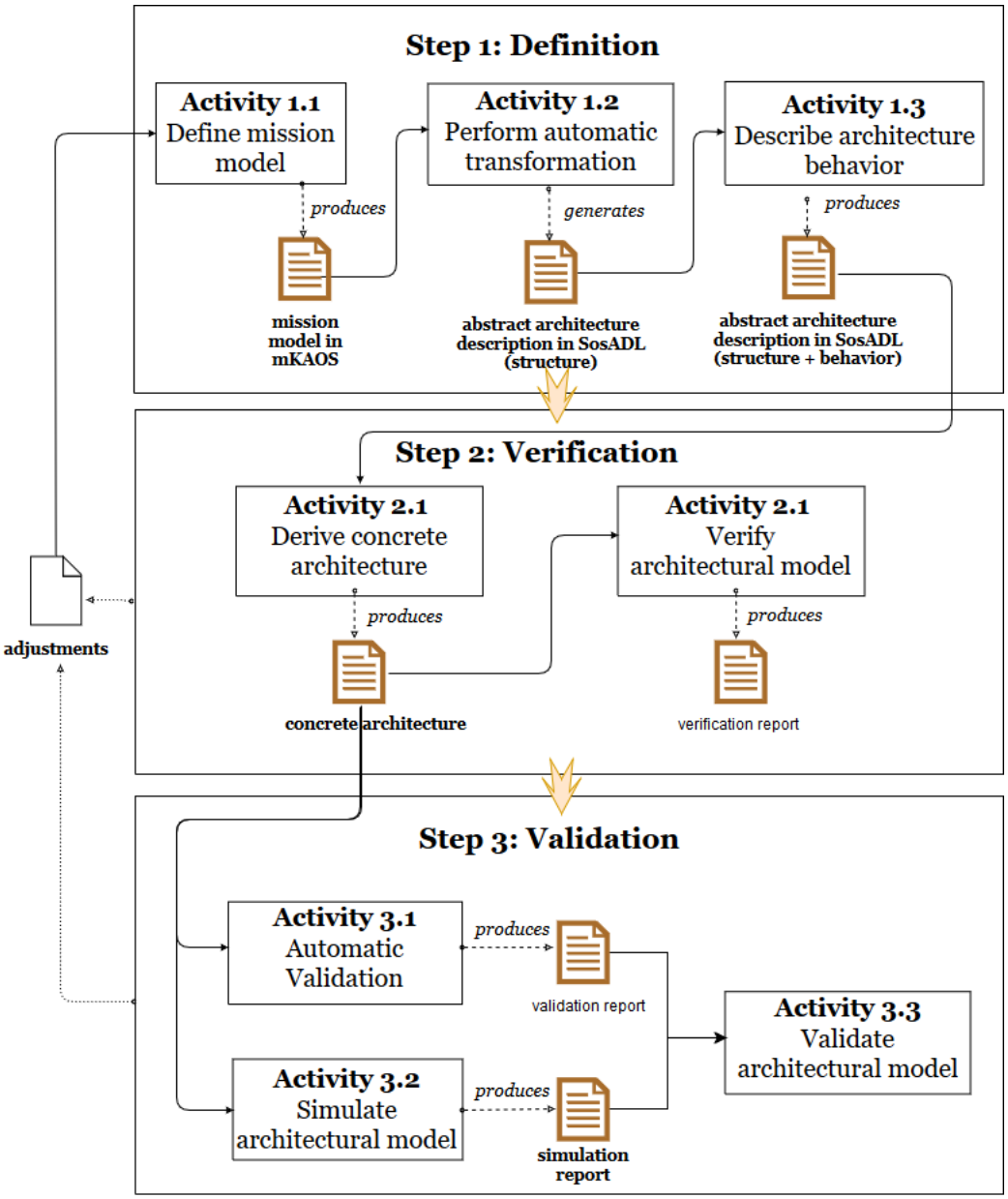


Figure 28: Overview of M2Arch

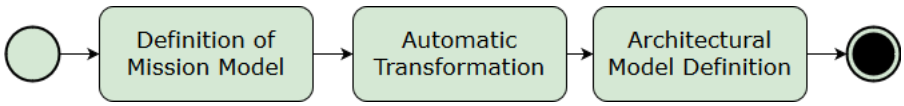


Figure 29: Activities of the Definition Step

The definition step is organized in three activities, as presented by Fig. 29: (i) Mission Model Definition; (ii) Automatic Transformation; (iii) Architectural Model Definition. Each activity produces an artifact that is used as input for the next activity.

As the definition produces an architecture, it is expected to be the most complex step of M2Arch. In this Section, we describe all the activities that are involved in this step, also presenting some guidelines to promote some features we expect the models to contain. Section 4.2.1 describes the starting activity: definition of mission models, which is done using the mission modeling language mKAOS. Section 4.2.2 presents an automatic mapping that is responsible for partially generating the architectural model, based on the mission model. This automatic mapping was implemented based on the equivalent concepts that permeates the mission and architecture models. Section 4.2.3 describes a third activity that uses the generated architecture as input to produce an architectural model that encompasses both structure and behavior of the SoS.

The definition step outputs two artifacts that must be maintained during the whole development of the SoS: the mission model and the architecture model. Thanks to the traceability and the automatic mapping, the changes in one of those models can be automatically reflected in the other, whenever necessary.

4.2.1 Mission Model Definition

Mission models are the core model for our method. Therefore, defining a detailed mission model is the key to the successful use of our approach.

In mKAOS, Mission Models are structured in six models: (i) an homonymous model, **mission model** responsible for describing individual and global missions, as well as expectations from the environment; (ii) **responsibility model**, that describes constituent systems and their responsibilities over the missions; (iii) **operational capability model**, that describes the capabilities of the constituent systems; (iv) **communicational capability model**, responsible for representing the cooperations among the constituent systems; (v) **emergent behavior model**, that defines the expected emergent behavior and the conditions for their emergence; (vi) **object model**, that specifies objects, events and

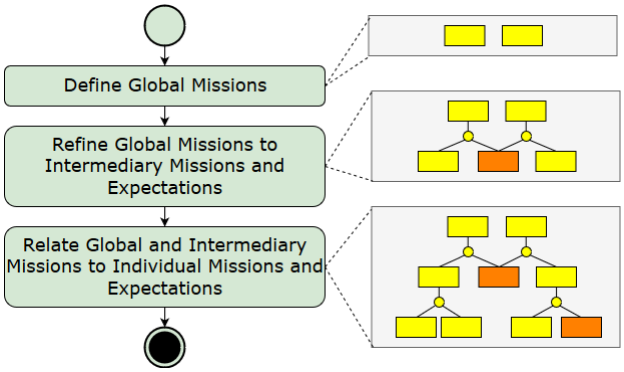


Figure 30: Defining a Mission Model

```
Mission PromoteCommunicationAmongPeople {
formalDef = always during 100 time units
   forall user:allOfType(SNUser) {
      exist server:allOfType(Server) {
          eventually before 100 time units areConnected(server,user)
      }
   }
}
```

Figure 31: Formal Definition of Mission *PromoteCommunicationAmongPeople*

constraints.

We suggest that the definition of the mission model starts by the homonymous model. The stakeholders must be able to express the missions they want the system to achieve and refine those missions to a set of lower level missions and expectations. Fig. 30 depicts on the activity of defining a mission model, that starts by the definition of the global missions. The global missions must be refined to *intermediary missions*, using *Expectations* as needed. Finally, the missions should be associated to individual missions and *Expectations*.

The missions must be detailed as much as possible. A proper use of the *Mission Refinements* allow the stakeholders to express various kinds of refinement relationships. It is important for the individual missions to be formally described, using DynBLTL constructs within the *formalDef* field, as shown by Fig. 31. A formal description of a mission specifies the conditions for the missions to be achieved. In Fig. 31, the mission *PromoteCommunicationAmongPeople* is achieved when exists a *server* connected to each user (*SNUser*). Formally described missions can be automatically checked by M2Arch, easing the validation process.

Based on the missions, the stakeholders might identify the constituent systems that are able to perform the individual missions, describing the *Responsibility Model*. Then, it

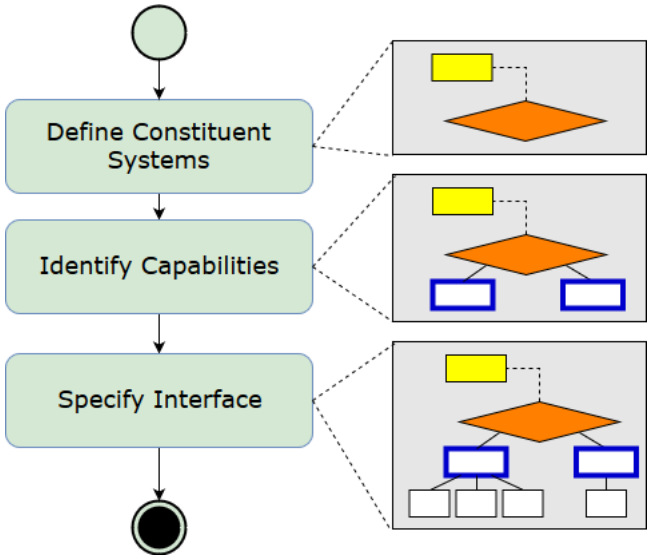


Figure 32: Specifying Capabilities of a Constituent System

is possible to identify the capabilities of the constituent systems that must be described in the *Operational Capability Model*. Since capabilities require an interface, at this moment it is important to have an *Object Model* with all entities that will be exchanged between the constituent systems. Using *input* and *output* links, the designers can define the interface of a capability. Fig. 32 depicts the process to describe the operational capabilities of a *Constituent System*. It starts by the definition of the constituent system, based on the mission it will be responsible for; then, the definition of the capabilities; finally, it is possible to define the interface of each capability using the *input* and *output* links.

Following the definition of operational capabilities, the stakeholders must identify, in the mission model, potential interaction points. Whenever an operational capability produces a data, as in Fig. 33, and a data of same type is used by another operational capability, it is possible to establish a cooperation link between these capabilities. In Fig. 33, the capability *ToProvideHidrologicalModel* produces an *HidrologicalModel*. An object of type *HidrologicalModel* is used as input for the capability *ToSimulateHidrologicalChanges*, from another constituent system. Therefore, this characterizes a possible cooperation point between the involved constituent systems.

It is worth highlighting that this activity consists in specifying **possible interaction points**, regardless of their real use by the constituent systems or not. Each interaction point represents a communicational capability, which implies in a possible cooperation between two or more constituent systems.

The cooperation points (communicational capabilities) allow some emergent behaviors

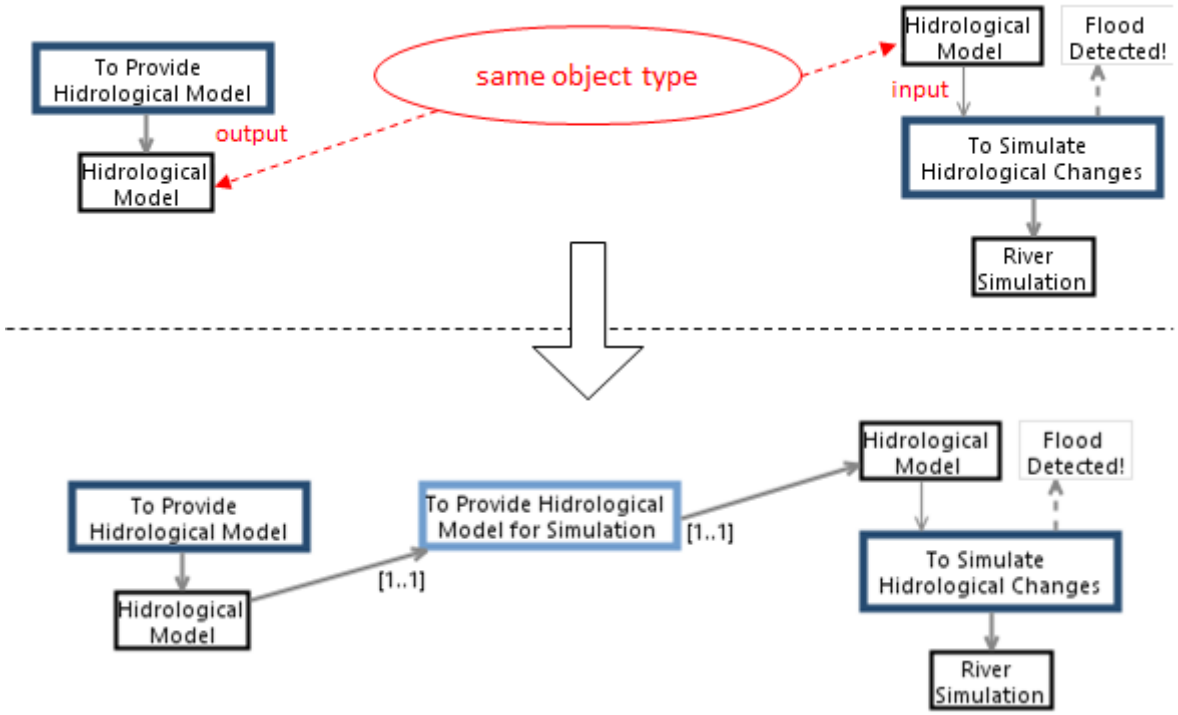


Figure 33: Identifying Communicational Capabilities

to appear in the SoS. Each emergent behavior is defined on one or more communicational capabilities, which is specified in the *Emergent Behavior Model*. Since mKAOS is not concerned with the implementation of the SoS, it is not capable of representing the operationalization of the emergent behaviors. However, it is strongly recommended to describe a formal rule to check the emergence of each behavior, using DynBLTL constructs. Notice that, as communicational capabilities, the stakeholders must identify as many emergent behaviors as possible, no matter if they are desired or not.

Finally, it is possible to define domain rules, *Constraints*, to focus on the SoS as a whole. mKAOS allows two kinds of constraints: (i) **domain invariants**, and (ii) **domain heuristics**. The only difference between them is the required commitment level. Domain invariants are constraints that **must** be fulfilled at every moment. Domain heuristics specify desirable, but not mandatory, properties. Syntactically, both constraint kinds are defined using the same structure, in mKAOS, that consists in a DynBLTL rule.

4.2.2 M2Arch Automatic Mapping Process

mKAOS was designed as a descriptive language for missions in SoS, focusing on what the system must be able to achieve instead of how it will achieve. Nonetheless, the descriptive elements of mKAOS refine mission definitions to the system level, assigning respon-

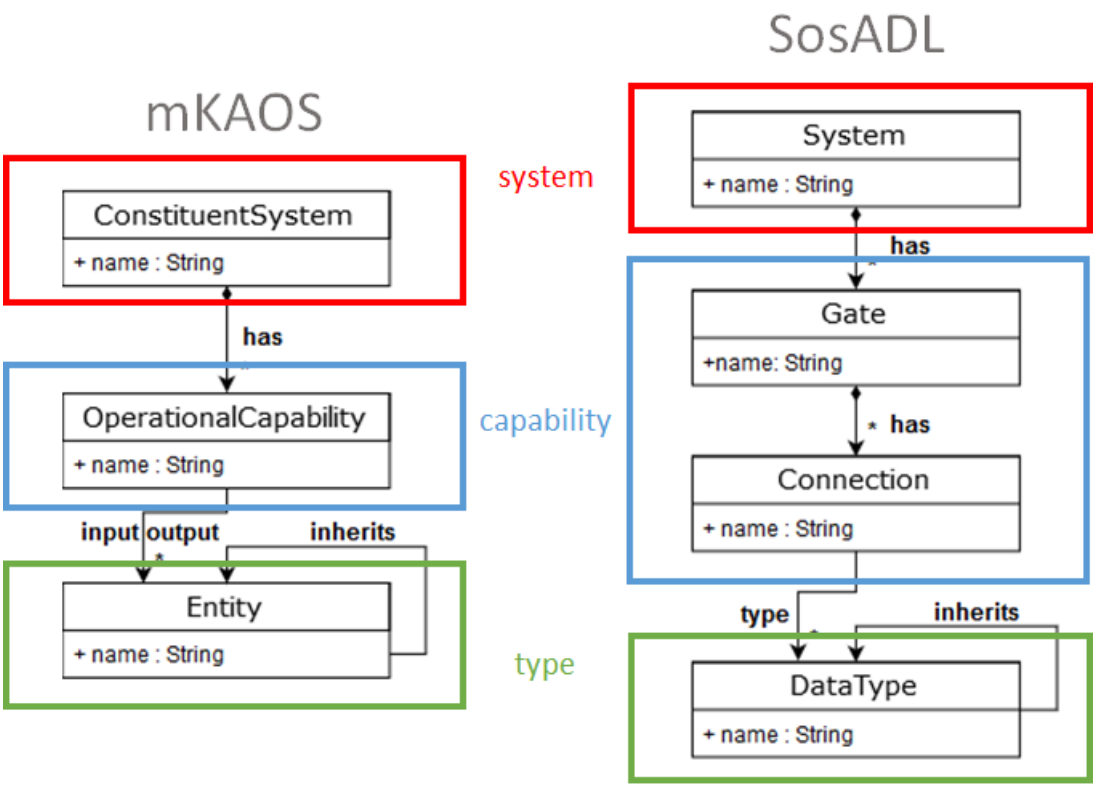


Figure 34: Overview of Equivalent Concepts

sibilities and obligations of each constituent system. At this point, no further description related to how the system will achieve the existing missions is possible in mKAOS. Therefore, the architectural description provides a new level of abstraction by refining mKAOS models to an operational, coarse-grained level. Although the proposed refinement relies on a mapping from missions to architecture, neither the mission model nor the architectural description provides sufficient information to represent the information from each other, some data are not reflected in the architectural description during the refinement process (e.g., missions) and, hence, both models must be maintained for its own purposes.

Considering that mKAOS and SosADL provide different levels of abstraction for the system, the mapping process is based on the equivalent concepts between both languages (SILVA et al., 2016; SILVA; CAVALCANTE; BATISTA, 2017). Fig. 34 presents the association between the equivalent concepts that permeate between both mission and architecture models. The main equivalent concept is the **capability**, which is available in mKAOS models in form of a homonymous element and in SosADL can be represented by the set of *interfaces of constituent systems*: the gates. Capabilities, in mKAOS also, have an interface, therefore a transformation process would rely on the interfaces of these and the constituent systems in SosADL.

In mKAOS, a capability is a first order element whose interface is defined by the composition of the inputs and outputs links of the *Operational Capabilities* and *Communicational Capabilities*. These links are product of the overlapping between the *Capabilities Models* and the *Object Model* and represents the nature of the data that is received or sent by each *Capability*. On the other hand, SosADL defines interfaces as essential, explicit elements for defining the architecture of an SoS. It represents the interfaces as *connections*, which are used for both structural and behavioral specifications. A set of connections form a *gate*, that can be directly related to a capability. *Events* in mKAOS are also mapped to connections, in SosADL and handled as a special kind of data. However, it is possible to represent Events as usual connections.

Since both mKAOS and SosADL provides representations for constituent systems, namely *Constituent System* and *System*, a natural association is found between those elements. In mKAOS, each *Constituent System* is directly associated to a set of capabilities, therefore, it is possible to identify which system implements each capability. On the other hand, in SosADL, a System encompasses a set of gates that represent its interfaces. As aforementioned, we are capable of relating capabilities with gates, which allows a syntactical relation between *Constituent Systems* (mKAOS) and *Systems* (SosADL). Since these elements are already conceptually related, performing such kind of mapping strengths traceability.

However, the representation of the capabilities depends on their nature. In mKAOS, the representation of *Operational Capabilities* have a different semantics compared to those for *Communicational Capabilities*. This difference relies on the fact that *Communicational Capabilities* are better associated to obligations than to interfaces, when comparing to ADL concepts. Therefore, the interface of *Communicational Capabilities* are more similar to channels for communication and cooperation, that specify some kind of *contract*, although it is capable of performing some operations. Moreover, the *Communicational Capabilities* are not associated to *Constituent Systems*, hence they cannot be transformed into *gates* for those Systems, as *Operational Capabilities* do. Due to these characteristics, we found that *Communicational Capabilities* are more related to *mediators* than constituent systems, as they are part of the SoS as a whole.

Further, regarding the mission models, mKAOS specifies that a mission has a priority and the SoS or the constituent systems might choose to achieve one mission instead of another, depending on the available resources. This is a completely normal behavior and must be taken into account when designing the architecture. In this regard, the mediator

tackles this issue since it is resolved at runtime and has an inherently dynamic nature. The mediator allows to specify a connection that may be active or not, depending on the available resources: if a mission depends on this connection, we can associate its achievement with the status of the mediator. Therefore, the mapping of communication capabilities to mediators also supports traceability, such as the mapping of constituent systems to systems.

Based on the relations we found, we defined a mapping process based on model-driven development (MDD) (VöLTER T. STAHL; HELSEN., 2006), an approach that changes the focus of problem solving from programming to abstract modeling. Modern MDD solutions are mainly based on model-to-model (M2M) transformations (SENDALL; KOZACZYNSKI, 2003), which consist in automatically refining models to lower abstraction levels aiming to reflect solutions defined in higher levels. Most M2M implementations are implemented upon Eclipse (ECLIPSE, Eclipse.org, a), in particular relying on the Eclipse Modeling Framework (EMF) (EMF, Eclipse.org, b), a largely used framework that simplifies the creation of modeling tools and languages. As both mKAOS and SosADL implementations are based on EMF, it is easy to establish traceability between models of these languages.

The mapping process is divided into five steps, as illustrated in the diagram depicted in Fig. 35:

1. Identification of the *data types* used in the *Object Model* (*entities* and *events*) and their definition in SosADL;

2. Identification of *constituent systems* from the *Responsibility Model* and their definition as *systems* in SosADL;

3. For each *system*, select the associated *operational capabilities* specified in the *Operational Capability Model* and define a *gate* whose *connections* are defined for each input, output, and event. Input events will result in input *connections* whilst produced events will be mapped to output *connections*;

4. For each communicational capability defined in the *Communicational Capability Model* define a *mediator* whose duties are defined based on the input and outputs for the *capability*, similarly to the *gate* production. Inputs or outputs from *communicational capabilities* that are not used by any *operational capability* are described as inputs/outputs for the SoS as a whole;

5. Connect *constituent systems* and *mediators* using the data association defined by input and output links in mKAOS, thereby establishing bindings in SosADL for

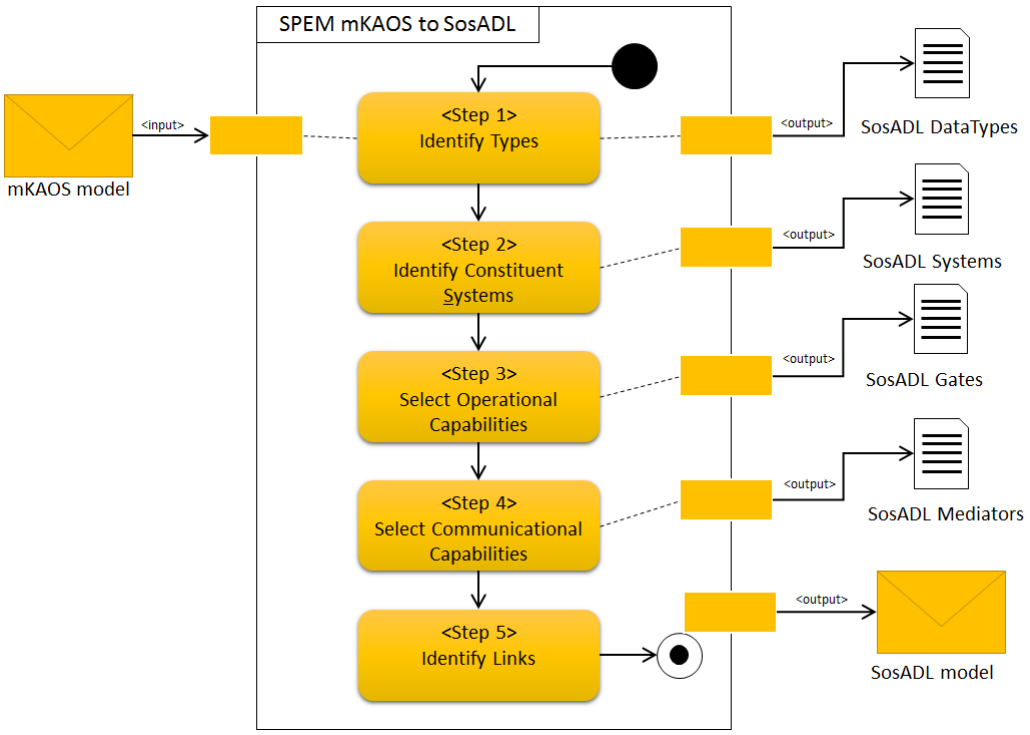


Figure 35: Mapping Process from mKAOS to SosADL

mKAOS	**SosADL**
Constituent system	System
Communicational capability	Mediator
Operational capability	Gate (in system)
Input/output/event	Input/output connection
Entity	Data type
Event	Data type

Table 5: Correspondence Between the Elements of mKAOS and SosADL Languages

each of these links. This last step involves the *Object Model* and both *Operational* and *Communicational Capability Models*, as well as the links between the objects and capabilities;

Table 5 summarizes the correspondences between the mKAOS and SosADL elements, implemented by the mapping process.

4.2.3 Architectural Model Definition

Through the automatic mapping of mission models to architecture, the constituent systems and its interfaces, as well as the mediators and the topology of the architecture will be automatically generated. Therefore, at this stage, the architect focus only on the

```
system Sensor(lps: Coordinate) is {
  …
  behavior sensing is {
    repeat {
      choose{
        via m::sense receive data
        via m::measure send data::convert()
      } or {
        via m::pass receive data
        via m::measure send data
      }
    }
  }
}
```

Figure 36: Behavioral Definition of System Sensor

behavioral aspects of the architecture. However, if necessary, the architect may adjust the generated topology and introduce new types or interfaces as needed.

Behavioral definition in SosADL encompasses three elements: (i) **behavior**, that describes how a system or mediator behaves; (ii) **assume**, specifying the assumptions a gate will make about the environment; and (iii) **guarantee**, that specifies a set of properties that the system provides.

Behavioral declaration specifies how a system or mediator behaves. It consists in a set of *behavioral statements*, that might be: (i) setting/changing the value of a variable; (ii) using external interaction constructs to specify sending or requesting some information; (iii) sending or receiving an information; (iv) conditional statements (if-then-else); (v) choosing one behavior depending on a given information (choose/switch); and (vi) loops. A behavior can also be *unobservable*, to express situations in which the architect does not have access to the behavior of a given system or mediator.

Fig. 36 shows an of a example behavior definition of the constituent system *Sensor*. This behavior specifies that the sensor will always transmit either the data it sensed or another value that was transmitted to it. The definition of the behavior of each constituent and mediator is fundamental to the further steps of M2Arch: verification and validation.

Defining the assumptions allows the architect to abstract some constraints of the environment, simplifying the behavioral definition. The asserts (assumptions and guarantees) consists in the definition of a set of properties that will be fulfilled by the environment. These properties are defined using a set of statements similar to those used in behavior. Asserts can be empty, using the construct *anyaction* to express that any state of the environment/system (environment for assumptions, system for guarantees) will be

```
mediator Gateway() is {
  …
  duty transmit is {
  …
  } assume {
     anyaction
  } guarantee {
  protocol transmit is {
    repeat {
      via transmit::fromSensors receive measure
      via transmit::toMonitor send measure
    }
  }
}
```

Figure 37: Assert Definition of Mediator *Gateway*

accepted.

Fig. 37 depicts the definition of a pair assumption-guarantee for the mediator *Gateway*, this mediator assumes nothing and guarantees that the data transmitted is the same as received.

Asserts can be used for verification purposes, although currently M2Arch does not supports it.

4.3 Verification

The second step of M2Arch consists on checking the domain-related properties, defined in Activity 1.1: definition of the mission model. This step is almost fully automated, requiring some configuration and sometimes the implementation of external controllers, introduced in Sub-subsection 5.3.3.

Definition step produces an abstract architecture and a mission model as outputs. However, both verification and validation must be performed over concrete architectures. A concrete architecture is a **runtime architecture**, realized by the available resources in a given environment. Such concrete architecture is fundamental to **simulation**, which is used by the model checker to check the given properties.

SosADL simulation is further discussed in Section 4.3.1, in which we present the mechanism to generate concrete architectures and how these are used on model simulation. Section 4.3.2 presents the verification mechanism that uses the statistical model checker *PlasmaLab* to verify domain-specific properties.

4.3.1 SosADL Model Simulation

SosADL simulation is the base for all automatic verification and validation processes proposed by M2Arch. It allows the architect to observe the architecture in a controlled runtime environment. This process is supported by the SosADL Execution Engine, presented in Section 3.3.4.

Since the execution engine requires a concrete architecture to be able to perform the simulation, before starting the simulation process, it is fundamental to produce a concrete architecture. Such production is also automatized and requires the set of available constituent systems in the environment to be simulated. This process is further discussed in Sub-subsection 4.3.1.1.

Finally, after generated the concrete architecture, the model checker is able to verify the model for a given set of domain-related constraints. This step is completely automated and presented in Sub-subsection 4.3.2.

4.3.1.1 Generation of Concrete Architectures

Given the clear distinction between abstract and concrete architectures, presented in Section 3.3, it is possible to establish a mechanism to automatically generate concrete architectures based on an abstract architecture definition and the specification of the desired environment. This was done by Guessi et al (GUESSI; OQUENDO; NAKAGAWA, 2016) as a mechanism to verify feasibility of SoS architectures.

Guessi proposes the use of an exhaustive generator of concrete architectures to verify the feasibility of an abstract architecture. The approach reduces the problem to the Boolean Satisfiability Problem (SAT) and evaluates the environment in terms of available constituent systems to generate all possible architectures that comply with the given abstract architecture. If no solution is found, then an counterexample architecture is generated for each violation of the abstract architecture.

The approach uses *Alloy*, a SAT solver engine, to combine the provided environment (i.e. a set of available constituent systems) and the abstract architecture to generates all possible architectures that, combining the available systems, realizes the architecture. For doing so, a metamodel for SosADL was build using Alloy constructs. This metamodel (presented in Figure 38[1]) enables the use of SosADL abstract models as inputs for the solver.

[1]Extracted from (GUESSI; OQUENDO; NAKAGAWA, 2016)

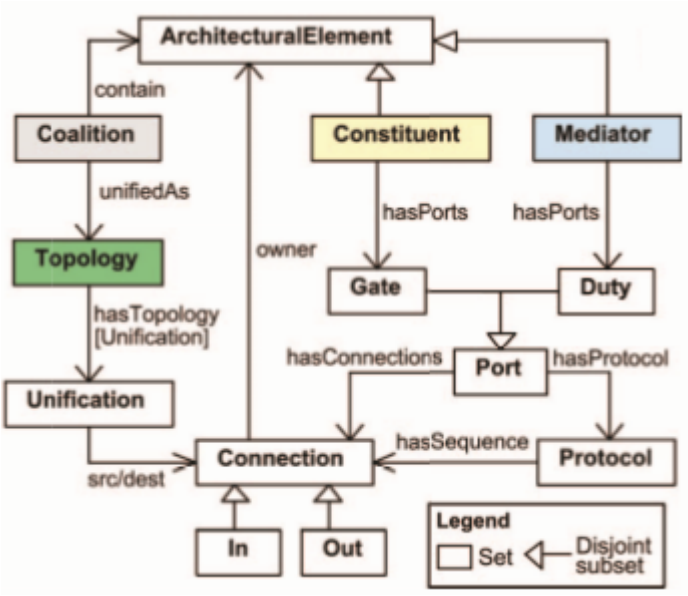


Figure 38: Alloy metamodel for SosADL

Since the solution uses this exhaustive approach, the execution might have a high cost in time. The authors are currently working in improvements to lower that computational cost. It is worth mentioning that, since Alloy is also formally grounded, it is able to maintain all constraints defined in SosADL during the derivation process.

4.3.2 Verifying Domain-Specific Properties

Given a concrete architecture and a mission model, M2Arch supports automatic verification of the domain-specific properties, specified as *Constraints* in mKAOS models.

Although the verification relies on the simulation, it is not necessary to configure the simulator at this stage. However, it is essential to configure the stimuli generators, either in the simulation configuration or using external controllers. We strongly suggest the use of the external controllers for this purpose, since they allow a wider control over the runtime model.

Each external controller must implement the *ExternalController* interface and be associated to a constituent in the concrete architecture. These activities were described in Sub-subsection 5.3.3. Overall, the architect selects the concrete architecture to be verified and invokes the verification.

The automatic verification process is divided in three activities, illustrated by Fig. 39: (i) *setup*, in which the involved tools are instantiated and configured; (ii) *initialization*, that starts the services; and (iii) *simulation*, that runs the simulations to perform the

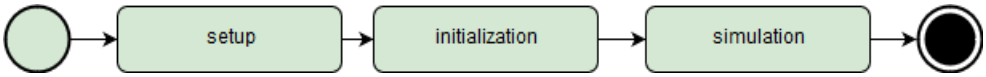


Figure 39: Activities of Automatic Verification

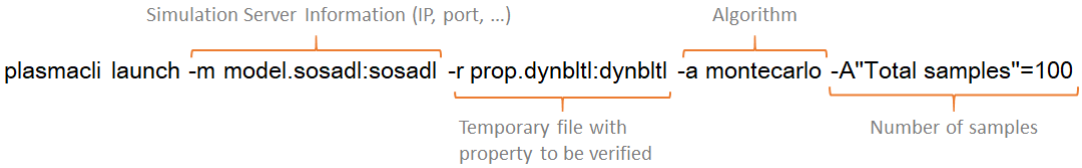


Figure 40: Initialization of PlasmaLab

checking.

First, in the **setup** activity, the verification tool creates an instance of a SosADL Simulator and reads the mission model, extracting all DynBLTL rules within the constraints. Each constraint is registered into a temporary file that will be used by PlasmaLab.

During **initialization**, a SosADL Simulation Server is initialized and its access data is saved into a temporary file, then the process initializes PlasmaLab and connects it to this simulator, using a set of parameters specified in the **Model Checking Configuration**, that can be automatically generated with default parameters. This configuration determines the number of simulation samples, algorithm and other optional parameters depending on the algorithm selected. The default configuration uses 100 samples and *MonteCarlo* algorithm. Fig. 40 presents the request that initializes PlasmaLab with the default parameters.

Finally, PlasmaLab takes over control the simulation and perform the verification of the properties. The properties will be verified individually, therefore, whenever there is a constraint violation, the tool is capable to report exactly which constraint was violated and the circumstances in which that happened. The SosADL Simulator will keep registry of all operations and allows the architect to track the whole execution to the original violation, using the event report. Each simulation sample produces a report with all activities executed in each simulation, as presented in Fig. 41.

Furthermore, at the end of the verification process, a **verification report** is generated, this report specifically contains the domain-specific constraints. Fig. 42 presents a simulation report that checks for two constraints: *heur1* and *inv1*. The report is generated by PlasmaLab, although some filtering is applied to avoid excessive data. Altogether, the simulation and verification report helps the architect to identify the faulty points in the architecture.

```
# Simulation Report for fmsos_solution1.sos

Setting up Execution Engine...
Initializing...
Done.
Step 1.
Step 2.
Step 3.
Step 4.
Step 5.
ExternalController SensorStimuliGenerator injected value 8 into sensor1.g1.opt
Step 6.
Value 8 from sensor1.g1.opt transmitted to gateway.g1.d1 via gatewayMediator
Step 7.
ExternalController SensorStimuliGenerator injected value 12 into sensor1.g1.opt
Value 8 from gateway.g1.d1 transmitted to rms.sensors.s1 via sensorsNetwork
```

Figure 41: Simulation Report

```
# Verification report for fmsos_solution1.sos
# Using mission model: fmsos.mkaos

inv1 violation on simulation 26
inv1 violation on simulation 43
+----------+---------------+-----------------------+--------+
| Name     | # Simulations | # Positive Simulation | Result |
+----------+---------------+-----------------------+--------+
| heur1    | 100           | 100                   | 1.0    |
+----------+---------------+-----------------------+--------+
| inv1     | 100           | 98                    | 0.98   |
+----------+---------------+-----------------------+--------+
```

Figure 42: Verification Report

4.4 Validation

Architectural validation is the activity that is responsible for identifying whether the system implements what it is intended for. In the context of SoS, this activity is directly related with the mission models. Since the objectives of a SoS are expressed in terms of missions, analyzing the commitment of the system with these missions is probably the most notorious task on a validation process.

However, there is another aspect of validation that does not concern on the mission model, but on the system conception itself. Mission modeling is an activity similar to requirements engineering, therefore it relies on the communication between stakeholders and the capability of those to express the needs in the model. These aspects might lead the stakeholders to produce a mission model that does not reflect their actual expectations or needs. Hence, we propose a process that not only validates the architecture within the mission model, but also supports validation of both mission model and architecture within the expectations and needs of the stakeholders.

We propose a two-step validation process: (i) automatic validation of missions and emergent behaviors; and (ii) manual validation through simulation. These steps are complementary and focus on different aspects of validation. Altogether, these activities support the validation of the architecture and mission model, regarding the stakeholders' needs.

It is important to highlight that our **automatic validation** relies on a **verification process**. In fact, although the automatic validation validates the architecture within the mission model, the stakeholders need to validate its results and perform the manual validation in order to validate the architecture within their needs.

In this Section we present our solution, in Section 4.4.1 we present the automatic validation that is responsible for checking the compliance of the architecture with the mission model. In Section 4.4.2 we introduce a method to manually validate both the mission model and the architecture.

4.4.1 Automatic Validation of Missions and Emergent Behaviors

The automatic validation will detect whether the SoS architecture comply with its mission model, using a similar process to the verification. Indeed, for the final user the only difference is the *Model Checking Configuration*. Although the automatic validation

```
# Model Checking Configuration file
# This is a comment line

# general properties
missionModel=./fmsos.mkaos
type=validation
constraints=all
algorithm=montecarlo
plasmaParam=-A"Total Samples"=100 -progress
```

Figure 43: Model Checking Configuration for Model Validation

might use the default parameters of PlasmaLab, the configuration must explicitly specify that the process is a validation, as displayed in Fig. 43.

In M2Arch, the automatic validation is divided in four steps: (i) setup, tools initialized and instantiation of involved objects; (ii) initialization, service start; (iii) simulation, that will perform the simulation-checking of the missions and emergent behaviors; and (iv) analysis, in which the output of the model checker will be analyzed to infer additional information.

The automatic validation uses the same tools of verification and the same process. However, instead of checking constraints, the model checker will analyze the formal definition of missions and emergent behaviors. In the analysis step, M2Arch will define a priority for mission achievement, based in *Mission Priority*: higher priority missions should be achieved more often than lower priority missions. We called the frequency of achievement of a mission as *achievement rate*, which is calculated by PlasmaLab during the property checking process. Such achievement rate is essential to the analysis step.

The **analysis step** consists of analyzing the priorities of every mission and comparing it with the achievement rate that is obtained at the end of the automatic validation. An additional warning is produced whenever a lower priority mission is achieved more often than a higher priority. It is important to highlight that this behavior does not indicate the model is invalid, since a lower-priority mission might be easier to achieve and that would justify such behavior.

Analysis is also responsible for triggering critical faults. These occur when a mission achievement rate is sufficiently close to zero or zero. The architect must define in the *Model Checking Configuration* the default threshold to be used by the tools. By default, M2Arch considers a threshold of 0.5, which means that every mission must be achieved in at least 50% of scenarios. The achievement rate is calculated based on PlasmaLab responses, therefore it has the same confidence level as the model checker.

```
# Mission analysis
GlobalMission
(DetectFloodWithMaximumConfidence)
Rate: 97%
Failures: sim34 (step72), sim81 (step8)

GlobalMission (AlertCitizenInRiskyArea)
Rate: 98%
Failures: sim34 (step82), sim66 (step11)

IndividualMission
(ProvideWeatherBulletins)
Rate: 99%
Failures: sim34(step72)
```

Figure 44: Validation Report

As a result, analysis produces a detailed report that encompasses not only the result of the property checking, but also the over mentioned analysis regarding mission achievement. Fig. 44 depicts a partial validation report, in which global mission *DetectFloodWithMaximumConfidence* displays an achievement rate of 97%, failing in simulations *sim34* and *sim81*. The report also includes the step in which the violation of the rule was first detected.

4.4.2 Validating Concrete Architectures and Mission Models

The stakeholders must be able to foresee the overall behavior of the SoS, allowing them to identify unexpected emergent behaviors and potential mistakes in the mission model. We propose a simulation-oriented validation, that consists in executing the architecture in a controlled environment, with a step-by-step feedback that allows the stakeholders to track all activity of the system. For doing so, our simulator implements some types of report that allow it to build reports that contains the various aspects of the system. The stakeholders might choose to focus on the data operations, such as production or consumption, communications between constituent systems or even a combination of these two.

We propose the use of a combined set of reports to observe the overall behavior of the architecture. As illustrated by Fig. 45, the first activity is to execute the simulator that reports the data operations, that will allow the stakeholders to observe how each constituent system is behaving, independently. This activity through the specification of *reportType* on the simulation configuration. If any constituent system presents a misbehavior, it is necessary to test its behavior definition in the architecture, if there is any. Constituent systems with unobservable behaviors are operationalized by *ExternalControllers*, hence,

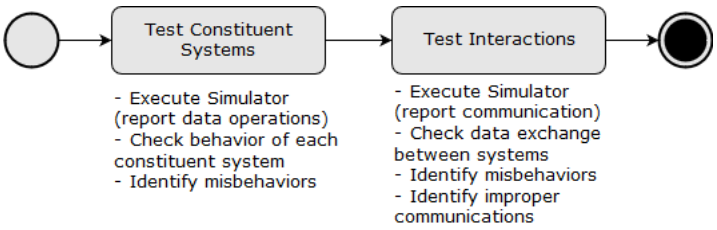


Figure 45: Activities of Manual Validation

there might be an issue with its *ExternalController* or the use of that specific constituent system is not adequate for the context of this SoS.

The second activity is to test the cooperation between the constituent systems, that can be obtained from the simulator through the selection of *reportType=communication*. This report type will concern on the mediators, that will be initialized and will operate when necessary. Through the reviewing of the communication event, that stakeholders are capable to observe faulty communication or cooperation between systems that should not. Any issue on the communication might be caused by the mediators, therefore it is fundamental to review the behavioral definition of the mediators in this context.

Altogether, these activities allow the stakeholder to identify adjustments to the mission model or architectural description of the SoS. Additional emergent behaviors might be found and we encourage their description in the mission model, even when they are not necessary to achieve of the SoS missions.

At the end of the validation activity, the stakeholders will have a validated mission model and architecture that should be maintained and evolve together. M2Arch should be restarted on every change in those models. Thanks to the associated toolkit, the method produces most of the artifacts automatically.

5 M2Arch Toolkit

Since M2Arch process provides an extensive, semi-automated methodology to produce SosADL architectures from mKAOS models, to provide a set of tools is key to assist the process. In this context, we introduce the M2Arch Toolkit, an Eclipse environment to support the whole proposed modeling process.

The toolkit encompasses four tools, as illustrated in the package diagram in Fig. 46: (i) the **modeling environment**, which includes modeling tools for both SosADL and mKAOS, visual and textual editors; (ii) the **mapping mechanism** that requires the modeling tools; (iii) the **simulation environment**, and (iv) the **V&V module**.

The outline of this chapter is organized as Section 5.1 describes the modeling environment; Section 5.2 presents an automated mapping mechanism, capable of partially generating SosADL models. Section 5.3 concerns on the implementation of the SosADL simulator. Finally, Section 5.4 regards on the V&V Module. Altogether, they compose the M2Arch Toolkit that supports every step of M2Arch.

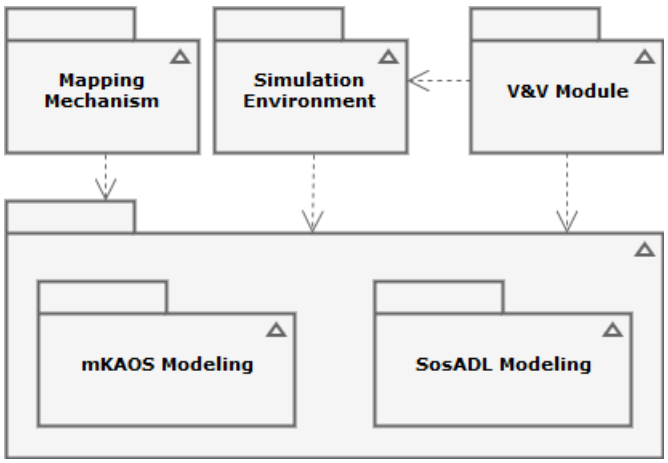


Figure 46: M2Arch Toolkit Overview

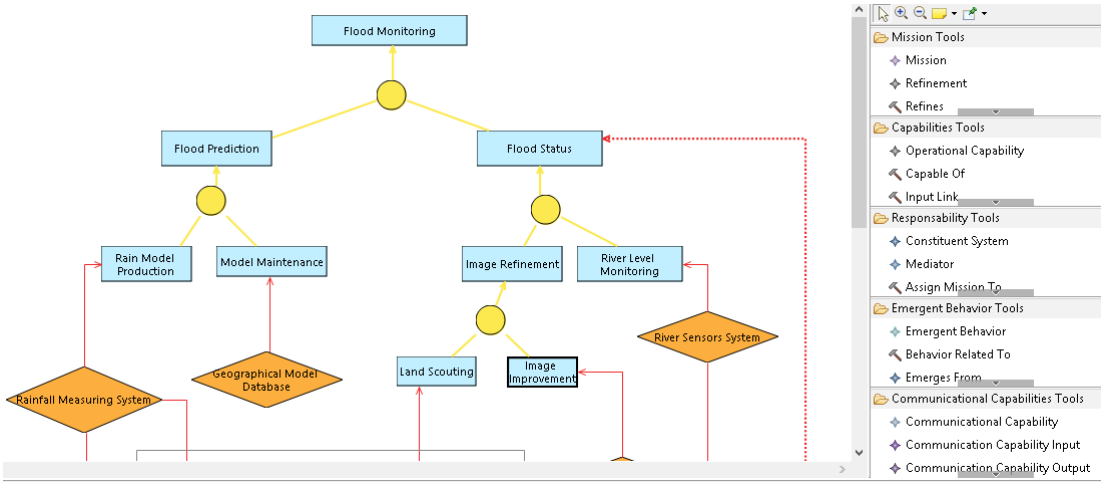


Figure 47: mKAOS Modeling Environment

5.1 Modeling Environment

The modeling environment of M2Arch Toolkit encompasses two main modeling tools. The first is the mKAOS tool, that was extended with the formalism and deployed as an Eclipse plug-in. mKAOS Modeling Environment encompasses the original graphical view of the language and the textual support of the formal version of the language, presented in Section 3.1. Figure 47 presents the mKAOS graphical modeling environment, introduced by (SILVA; BATISTA; CAVALCANTE, 2015).

On the other hand, SosADL modeling was enhanced to allow a new graphical viewpoint for the ADL. The modeling environment is based on the original SosADL tool, developed by the ArchWare team. It also encompasses the graphical tools for the language, described in Section 3.2.

Figure 48 despites the main screen of the modeling environment of SosADL, it presents an architectural diagram and the associated textual description. In this figure, it is possible to identify the correspondence between the textual (left) and graphical (right) descriptions, maintained by EMF. The tool also provides an outline view of the model for quick navigation.

Altogether, these modeling environments provide the necessary tools for modeling, visualization and edition of models in all languages involved in the process. Furthermore, they provide the interface necessary for the implementation of the mapping mechanism.

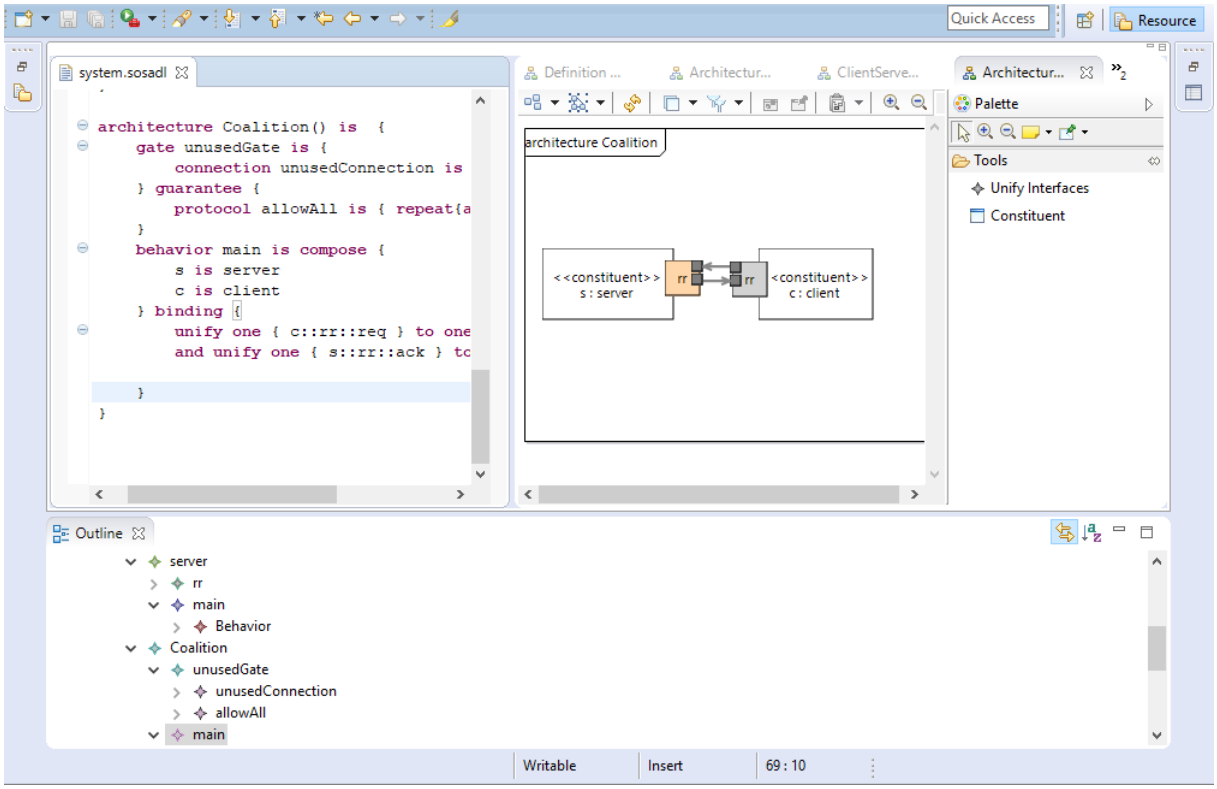


Figure 48: SosADL Modeling Environment

5.2 Mapping Mechanism

The mapping is implemented to be automatic, programmatically executed using a M2M transformation. This ensures the traceability of the missions and simplify the architecture design process: the architect is concerned only with describing behavior and detailing further elements not related to the mission model. Although the transformation does not encompass all mKAOS elements neither the SosADL elements, it still can be realized in both directions. However, it is important to mention that both mission and architectural models are complimentary to each other and they must be independently maintained. In the proposed mapping process, we have chosen a constructive approach in which the refinement will produce a single architecture capable of achieving the required missions and emerge the desired behaviors. An alternative is to build a set of possible architectures and verify the conformance of each one with the mission model, but this approach is computationally too expensive.

To implement the mapping process using EMF, we rely on the existing metamodels for mKAOS and SosADL. The implementation was developed using the ATL Transformation language [1], which was chosen due to two main reasons. First, the tools developed to

[1] http://www.eclipse.org/atl

```
rule ProduceSos {
  from
    missions : MKAOS!mKAOS
  to
    eblock : SOSADL!EntityBlock(
       datatypes <- missions.entities(),
       systems <-
      missions.constituent(),
       mediators <-
      missions.capabilities(),
       architectures <- arch
    ),
    sos : SOSADL!SoS (
       name <- 'GeneratedSoS',
       decls <- eblock
    ), ...
```

Figure 49: Main ATL transformation rule from mKAOS to SosADL

support both mKAOS and SosADL languages are based on the Eclipse environment, thereby easing their integration along with the ATL transformations. Second, ATL is often used in the community for model transformation approaches, thus having consolidated tools and detailed documentation available. It is important to acknowledge that there are other options for implementing the transformation with equivalent relevance, such as QVT[2]. In this case our choice rely on personal experience.

In ATL, the transformation is based on a set of rules that are executed whenever necessary, conducted by a main rule that leads the transformation. The main rule for the transformation from mKAOS to SosADL is presented in Fig. 49. The *ProduceSos* rule is responsible for controlling the transformation process as a whole, calling all other transformation rules. This rule transforms a mKAOS model into an SoS architectural model, generating datatypes from entities (step 1), systems from constituent systems (step 2), and mediators from communicational capabilities (step 4).

Fig. 50 presents a part of an ATL rule that implements the third step (operational capabilities to gates). This rule iterates over all possible inputs and outputs for each capability, producing a connection for each input or output relation. The produced connection is identified as an input or output connection and then the information is stored as the connection mode. Finally, the produced connections are stored in a gate generated from an operational capability.

Fig. 51 depicts an example of the mapping by showing the capability model in mKAOS (Fig. 51a) and a corresponding architecture in SosADL (Fig. 51b). In Fig. 51a, *Meteo-*

[2] `http://www.eclipse.org/qvt`

```
connections <-
 let allConnections :
   Sequence(SOSADL!Connection) =
  mkaos_operationalCapability
    .output()
    .union(mkaos_operationalCapability
       .input()) in
         allConnections->
           iterate(i; p : SOSADL!Connection =
              allConnections->first() |

-- check if input or output and set the value
                if (mkaos_operationalCapability.input()->includes(i))
       then
                 i.refSetValue('mode',
            'ModeTypeIn')
                else
                  i.refSetValue('mode',
                  'ModeTypeOut')
                endif
            )
```

Figure 50: ATL transformation rule for producing connections in gates from operational capabilities

rological and *Social Network* are constituent systems mapped to system elements with the same name, realizing the second step of the mapping process. In the third step, the *ProvideLocation* and *FindCitizen* operational capabilities are mapped to gates in the corresponding constituent systems. Each of the produced gates (PL and FC) will have a single connection since the capabilities handle only one parameter. In the fourth step, the *FormatData* communicational capability is mapped to a mediator, whose duties are defined encompassing the *Information* connections. In the final step of the model transformation, bindings are established based on the inputs and outputs of the capabilities and the mediator.

The tool provides a simple mechanism to run the transformation, that consists on simply selecting the mKAOS file and invoking the transformation. Figure 52 shows how this mechanism is provided to the user of the modeling environment: in a context menu for mKAOS files.

5.3 SosADL Simulator

SosADL execution plug-in was build upon the existing tools without any change in the original plugins. Hence, the tool can be integrated with those plugins and will be able to execute every existing model unchanged. This section details the structure and

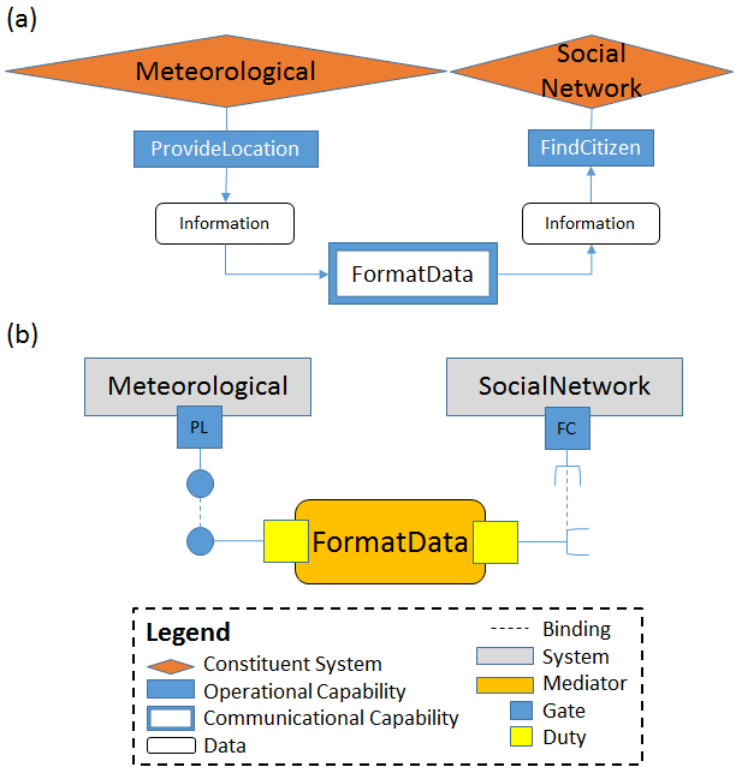


Figure 51: Example of refinement of a Capability Model in mKAOS to an architecture in SosADL

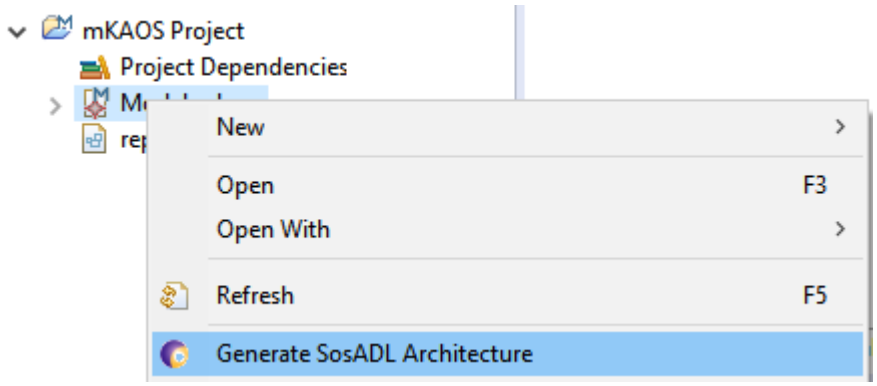


Figure 52: Transforming Mechanism Invocation

function of such execution plugin, which is responsible for the simulation of SosADL models. Section 3.3.4.2 introduces the architecture of the plugin. Section 5.3.2 details how each simulation is performed within the model execution plugin. Section 5.3.3 introduces the simulation configuration, a mechanism for the architect to control the simulation. Finally, the Simulation Server that implements a connector to PlasmaLab, an SMC tool, is presented in Section 5.3.4.

5.3.1 Context Manager

Probably the most important component of the SosADL Simulator, the context manager is responsible for creating and managing a structure we called *Context*. A context can be seen as an extension of a *scope*, including not only the variables, but also the data that is manipulated by the environment, its events and the status of every constituent and mediator.

The *Context Manager* controls the values that are used by the system, updating the *contexts* and creating new *contexts* when an adaptation process demands it. The *Context Manager* provides methods to verify the current value of any variable of the execution, but also to check the current state of a given constituent, mediator or external controller.

This component is also responsible for monitoring the values, triggering new data events whenever necessary. Context Manager's interface is presented by Fig. 53.

5.3.2 Simulating SosADL Architectures

The simulation is performed by the third and fourth layer. The third layer is responsible for the interpretation of the expression, execution of statements and verification of asserts, but also for synchronization and control of the environment. The fourth layer controls the execution and call those functionalities on demand.

Based on the **execution workflow** defined in Section 3.3.2, the SosADL simulator implements a derived workflow to execute SosADL models. The execution is divided in three steps: (i) *setup*, in which the *Simulation Configuration Manager* and the *Execution Engine* read the configuration file, define the model to be executed and the external controllers that will be loaded; (ii) *initialization*, which creates and initialize contexts, and loads the model and the external controllers; (iii) *step*, that will be iterated until the end of the execution, performing a single *execution step*.

```java
public interface ContextManager {
    /**
     * Sets event manager
     * @param e the event managet
     */
    public void setEventMananger(EventManager e);
    /**
     * Initialize a context
     * @param baseElement the architectural element
     * @return the new context
     */
    public Context initContext(EObject baseElement);
    /**
     * Gets a context for a given element
     * @param element the architectural element
     * @return the context
     */
    public Context getContext(EObject element);
    /**
     * Gets the status of a constituent system or mediator
     * @param s the constituent system or mediator
     * @return the status (normal, unavailable, unknown)
     */
    public Status getStatus(Constituent s);
}
```

Figure 53: Context Manager Interface

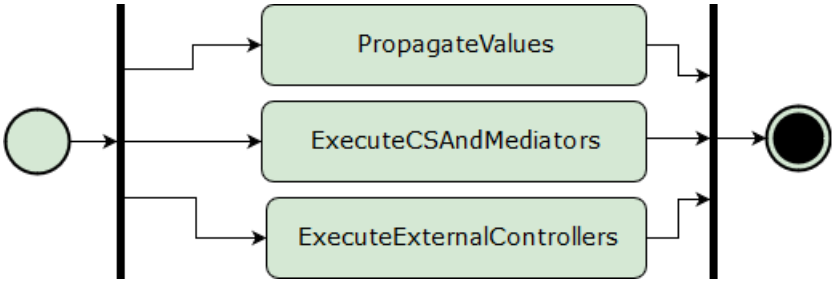


Figure 54: Activities of Execution Step

The execution step, on the other hand, is also divided in 3 steps, which are executed simultaneously, as illustrated by Fig. 54. A step **propagates the values**, based on the *unify* relations of the model. If a value is consumed or produced in a *connection*, this value (or the value *empty*) will be propagated to all *connections* that are unified to it. The steps also **execute the constituent systems and mediators**, which will be executed if the asserts are fulfilled and the necessary data is available. A third activity is the **execution of external controllers**, that will execute in the same circumstances as the constituent systems and mediators. In this activity, the default external controller will also introduce the data predefined in the configuration. This later activity is better discussed in Section 5.3.3.

Every activity may produce events that are used to follow up the execution. The

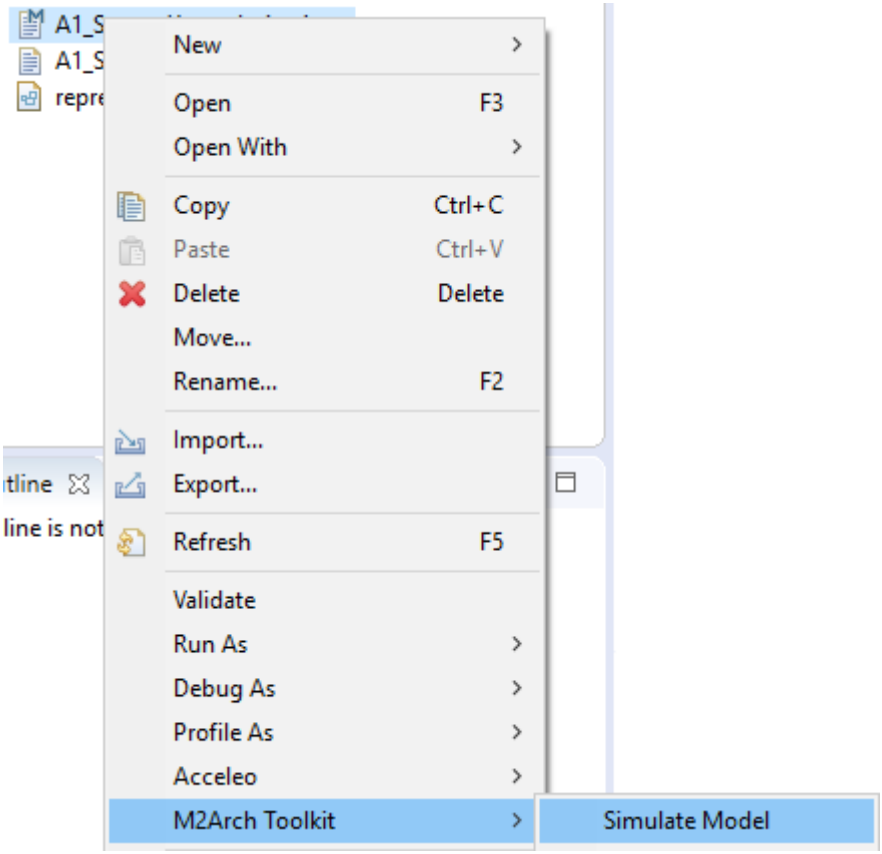


Figure 55: Starting SosADL Simulator

events are stored and managed by the *Event Manager*, that will add timestamps to the events, allowing the events to be chronologically ordered.

To perform a simulation, the user uses the interface provided by the *Simulation Environment*, simply selecting the file with the concrete architecture to execute and starts the simulator, as illustrated by Fig. 55.

5.3.3 Simulation Configuration

The *Simulation Configuration* is a special abstraction that stores information regarding the simulation itself. For doing so, we use a file with the extension *.sosconf*. If the configuration file has the same name as the model file, the *Simulation Configuration Manager* loads it automatically.

A configuration file is divided in three sections: (i) *simulation control*, in which the user defines the maximum number of steps and selects the report mechanism; (ii) *external controllers definition*, in which the user specifies which controllers will be used and their corresponding classpath; and (iii) *predefined stimuli*, in which the user may specify a value

```
# This is a comment line
# First section is Simulation Control
[control]
iterations=100
reportType=all
```

Figure 56: Simulation Control on Configuration File

```
public abstract boolean canExecute(Context context);
public abstract void execute(Context context);
```

Figure 57: External Controller Interface

to be introduced in the model in a predefined step of the execution.

The simulation control section contains two fields: (i) *iterations*, with the max number of iterations of the simulation; and (ii) *reportType*, that selects the detail level to be reported by the *Event Manager*. The report type might be "*all*", in which the event manager reports every event in the textual output; "*data*", in which only data production and consumption will be reported, and "*communication*" that will report only data propagation. Additional report types might be added in the future. Fig. 56 shows an example simulation control section, that specifies a simulation with a maximum of 100 iterations that reports every event.

External controllers use a plug-in architecture to interact with the system. For doing so, every controller must implement an interface, presented by Fig. 57. This interface contains only two methods: (i) *canExecute* that returns *true* if the controller can execute, and *false* otherwise; (ii) *execute*, in which the controller executes, manipulating the context as needed.

The user must specify the plug-ins folder in which the external controllers artifacts will be placed, the *External Controllers Manager* can only find controllers in this folder. Each controller classpath will then be associated to an architectural element through its *qualified name*, as illustrated in Fig. 58.

Finally, the predefined stimuli section contains associations between step numbers and expressions, in SosADL. These stimuli are loaded to the default external controller, that

```
# External Controllers Section
[controllers]
pluginsDir=\_plugins
server20=org.archware.constituents.ExperimentalController
```

Figure 58: External Controllers in Configuration File

```
# Predefined data injections
[data]
29 clients0.rr2.req1 12
110 clients0.rr2.req2 null
```

Figure 59: Predefined stimuli on Configuration File

will evaluate and inject the value in the specified field. In Fig. 59, two predefined values will be injected: (i) the value *12* in connection *clients0.rr2.req1*, at step 29; and (ii) the value *null* in the connection *clients0.rr2.req2*, at step 110.

Simulation Configurations are used during *setup* and *initialize* phases of the *Execution Engine*.

5.3.4 Simulation Server – PlasmaLab Connector

The last piece of the SosADL Simulator is the *Simulation Server*. Our simulator was built aiming at integration with Statistical Model Checking tools, specifically PlasmaLab[3] (LEGAY; SEDWARDS; TRAONOUEZ, 2016). Since PlasmaLab integration interface relies on TCP connections, we needed to implement a server able to handle some requests and translate the events to PlasmaLab format.

TAMIS team[4], responsible for the development and maintenance of PlasmaLab, provided a major support in this contribution, providing a set of common Java classes that PlasmaLab is able to handle and detailed instructions on how to build this server. Hence, in this context our contribution consists essentially on event translators.

SosADL Simulator was planned based on this interface, therefore, the *Execution Engine* has one method for each of these requests. The results of each call, that are events, are then translated and sent to PlasmaLab.

5.3.4.1 Interpreting SosADL Behavior

One of the major contributions of SosADL is the formal behavioral description provided by the language. SosADL allows architects to describe the behavior of a constituent system, mediator, gate, etc using constructs formally grounded in π-calculus. To enable simulation of SosADL models, it is fundamental to develop a tool capable of use behavioral description in SosADL to generate or manipulate data. In fact, an interpreter for the

[3] https://project.inria.fr/plasma-lab/
[4] `https://www.irisa.fr/en/teams/tamis`

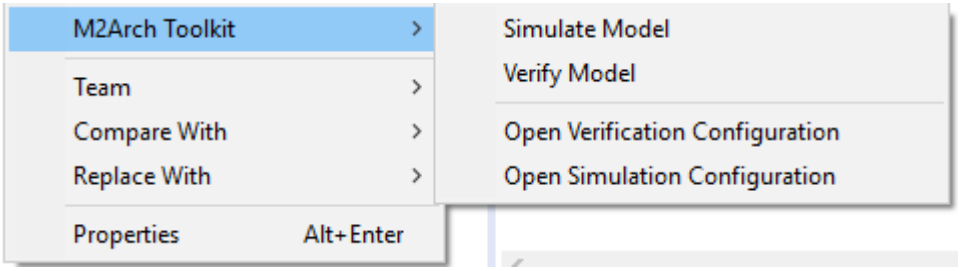


Figure 60: M2Arch Popup Menu

language is the main module for the simulation mechanism.

In this work, we are not concerned with this interpreter. However, the SosADL execution engine was structured to allow this interpreter to be implemented by future students. Currently, a small subset of the statements are capable of being interpreted, such as: (i) performing simple arithmetical operation; (ii) storing values in variables; and (iii) checking boolean values on variables. Most of the statements requires some features of the SosADL *typechecker* that are not currently available.

5.4 Verification and Validation Tools

M2Arch toolkit also encompasses a module that automatically configures and starts the automatic verification and partial validation. Both processes are done by PlasmaLab, and invoked by the user through a context menu. This context menu shown in Fig. 60 is available for all SosADL files.

5.4.1 V&V Module Overview

Since M2Arch proposes an extensive methodology that encompasses verification and validation, its associated toolkit supports the automation (partial or whole) of such activities. For doing so, the so-called V&V module uses the SosADL Simulator and the statistical model checker PlasmaLab (LEGAY; SEDWARDS; TRAONOUEZ, 2016).

The structure of the V&V module is simple, consisting essentially in a coordinator that is responsible for setting up the two involved tools and preparing the inputs for their initialization. This coordinator decides whether the operation is a verification or validation, based on a **configuration file**, and creates a set of temporary **property files** that will be provided as input to PlasmaLab. Fig. 61 presents an overview of the activities performed by the coordinator.

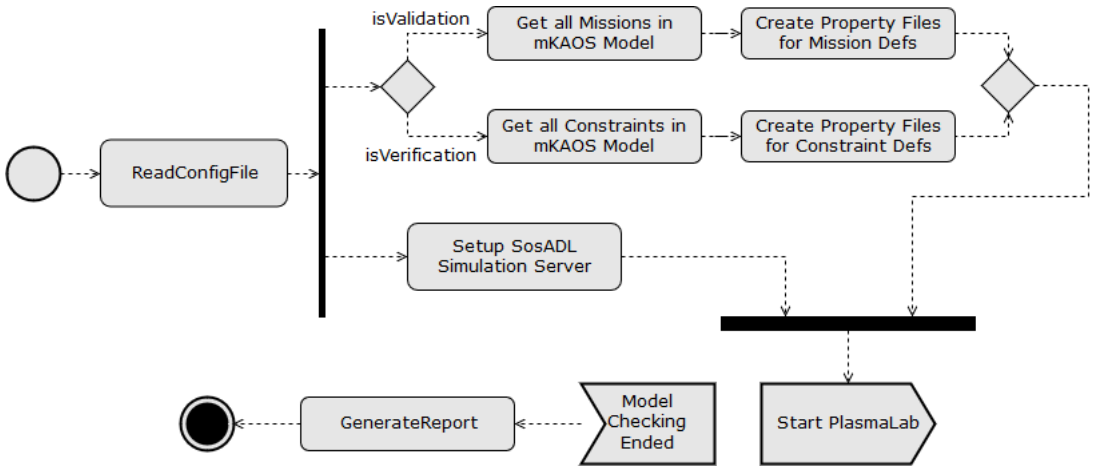


Figure 61: Model Checker Coordinator Activities

Initially, the coordinator will read the configuration files. Based on these files, the SosADL Simulation Server will be set up. Also based on the configuration files, the coordinator decides if the process is a verification or validation, depending on which process is to be executed, the temporary files will contain mission formal definitions or constraints definitions. The set of these property files and the informations concerning on the SosADL Simulation Server are used to build the initialization parameters for PlasmaLab.

After starting PlasmaLab, the coordinator is put on hold until the execution is finished. Finally, it will use the PlasmaLab-generated report and create an M2Arch report, depending on the type of the process (verification or validation).

The whole process is automatic: the user selects the SosADL file and accesses the context menu after selecting *Verify Model*, the tool will setup the V&V module and start the verification, as illustrated by Fig. 62. The tool will initialize the required parameters and start the verification. Such verification might be used by the verification process or the automatic validation.

The output of the verification/validation process is a report file, by default, although the user might select between a report file or the default textual output within M2Arch environment.

5.4.1.1 Verification Configuration

To allow the user to have more control over the operations supported by M2Arch toolkit, there is a set of configuration files that are used as input to the SosADL Simulator and the V&V Module.

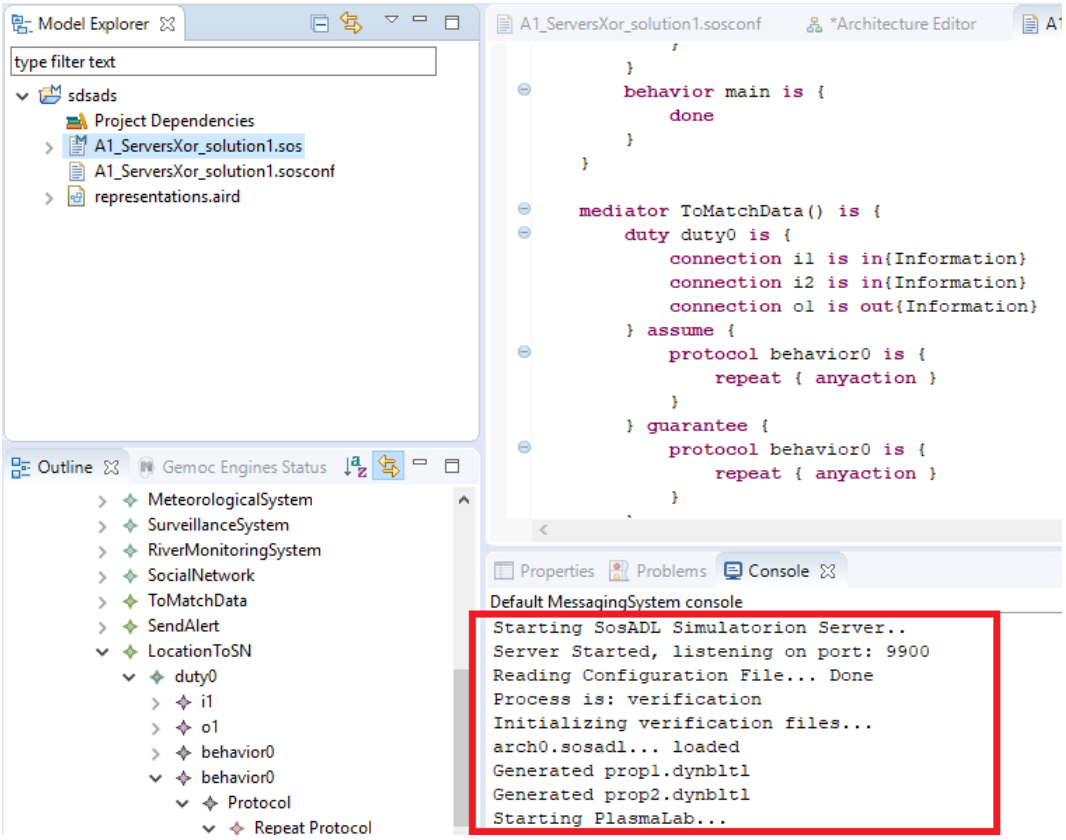


Figure 62: Starting Verification

It is important to highlight that both verification and automatic validation are performed by PlasmaLab. The simulation that is used along the process is hence controlled by the statistical model checker. Therefore, the V&V module might override some parameters of the simulation configuration on-the-fly. For instance, the number of steps of each simulation can be altered depending on the needs of PlasmaLab.

In this context, there are two simulation files that are somehow related: (i) simulation configuration and (ii) model checking configuration. The first is responsible for the parameters of the simulation and was described in Section 5.3.3.

The model checking configuration, on the other hand, determines how the V&V module will setup the checking and perform its activities. Unlike the simulation configuration, the tool is not capable of generating the model checking configuration file. Although some parameters have a default value, some of them must be defined by the user. The parameters of this configuration file are listed in Table 6, in which only the *missionModel* cannot be generated by the tool.

Additionally, it is possible to force the overriding of any parameter on the simulator. This allows the stakeholders to change the simulator for a given process of verification or

Param	Values	Description
missionModel	-	Path to the mission model to be used in the process
type	validation, verification	Determines the type of the checking
algorithm	any supported by PlasmaLab	Specify the algorithm to use on PlasmaLab
plasmaParam	-	parameters to be passed to PlasmaLab

Table 6: Parameters of the V&V Configuration File

```
# override simulator
-simulator
[control]
iterations=50

[controllers]
server20=org.archware.constituents.ServerAlternative
```

Figure 63: Overriding Simulation Configuration parameters

validation, without changing the existing configuration for the simulator. Fig. 63 shows a configuration file that overrides the simulator configuration, changing the *ExternalController* for the constituent *server20* and the number of steps.

The configuration file is parsed by the V&V module itself, any syntactical misuse of parameters will halt the checking process.

5.4.1.2 Reports

Key to the verification and validation steps, M2Arch reports provide detailed information about the processes that are automated by the tool. Specifically, there are two natures of report: (i) simulation report and (ii) model checking report.

The **Simulation Report** is built by the Simulation Environment, it describes events that occurred during every simulation. Using these reports, the user might follow up the whole architectural execution.

The simulation reports support four kinds of events: (i) Data; (ii) Communication; (iii) Execution; (iv) Structure. Fig. 64 presents a Class Diagram that specifies the Events involved in a *SimulationReport*. Every event has a *timestamp*, that relates the moment in which the event was triggered. This timestamp is automatically generated by the class constructor.

Execution and **structure** events are present in all reports. Execution events concerns on constituents or external controllers, signalizing their execution. Structure events, on

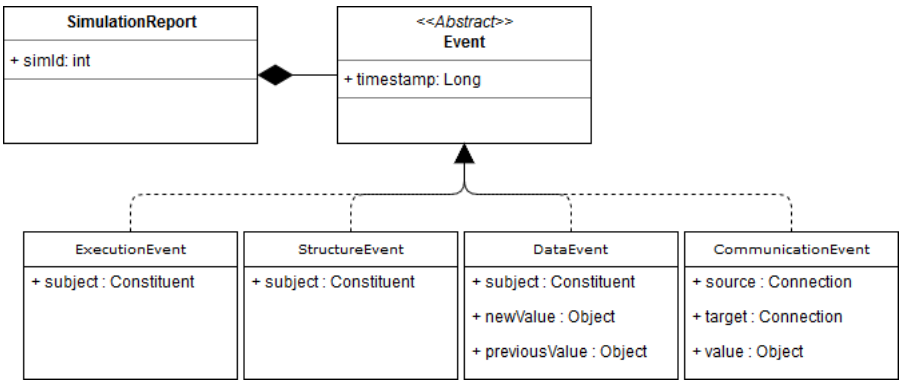


Figure 64: Events Classes that compose a Simulation Report

the other hand, report any change in the architectural structure, such as the leave or discovery of a constituent. Execution and structure events have a single attribute: *subject*, that refers to the element that was executed or changed.

Data events and **communication events** are present whenever the simulation configuration specify so, as mentioned in Section 5.3.3.

Data events report a consumption or generation of a new value, which occur on the *connection* according to the constituent behavior or a *ExternalController* intervention. Data events encompass three attributes: (i) *subject*, that refers to the element responsible for changing the value on a *connection*; (ii) *new value*, the new value of that *connection*; (iii) *previous value*, that is not included in the report but is stored and may be monitored for debugging.

Communication events regard in the data exchange between constituents. They are triggered whenever a data is transmitted from one *connection* to another. Notice that these events do not concern on the *mediators*, but on the *unifications*. In SosADL, a mediator is a also constituent in the coalition context, hence, the execution of mediators are also execution events. Communication events have three attributes: (i) *source*, that refers to the *connection* from which the value was previously stored; (ii) *target*, referring to the *connection* that will receive the value; and (iii) *value*, the value that was transmitted. An example of a simulation report is available in Chapter 4, Fig. 41.

The second type of report is the **Model Checking Report**. This report is generated by the V&V module, based on PlasmaLab output. A Model Checking Report depends on the type of the process: validation or verification.

It is important to highlight that PlasmaLab reports depends on the algorithm used in the process, currently, our tool only supports *montecarlo* reports for building detailed

reports. However, we consider this a minor limitation, since the V&V module will report PlasmaLab results either way.

Verification reports focus on properties and constraints. They are simpler than Validation reports, reporting only the set of constraints, with their respective number of simulations and positive results. These reports will also notify constraint violations, indicating in which simulation a given property was violated. Verification reports were previously introduced, by Fig. 42 in Chapter 4.

Validation reports are more complex. They provide a more detailed analysis on the results of evaluation of formally described missions, combining the results with the mission model. Validation reports were previously introduced by Fig. 44 in Chapter 4.

6 Case Study: Proof of Concept

6.1 Foreword

To evaluate M2Arch we ran a case study with the FMSoS, introduced in Section 2.8. Applying the whole process to the SoS, we generated a concrete architecture that was verified and validated throughout the techniques hereby proposed. The resulting architecture shown is very similar to the one previously modeled by the ArchWare team, although some relevant differences were noticed.

It is important to highlight that, for didactic purposes, some examples presented in this Chapter may be simplified. The full version of our case study is available at `http://consiste.dimap.ufrn.br/projects/m2arch`.

6.2 Application: FMSoS

The Flood Monitoring System-of-Systems (FMSoS) is an acknowledged SoS introduced in Section 2.8. This section details the application of M2Arch to produce an architecture for FMSoS, detailing all steps and presenting the involved models.

The outline of this Section follows the overall steps of the methodology, Section 6.2.1 regards on the Definition activity, that encompasses mission modeling and architectural modeling, including the automatic mapping between these. Section 6.2.2 concerns on automatic verification of domain-related properties. Section 6.2.3 presents the validation process, including the automatic validation of missions and behaviors and the manual analysis of the simulation; finally, Section 6.2.4 presents our conclusions about the methodology usage, by comparing the resulting models with previously defined models.

Constituent system	Individual mission
Meteorological System	Provide Weather Bulletin Monitor Weather
River Monitoring System	Monitor River Levels
Surveillance System	Monitor City Areas Calculate Risky Area
Social Network	Identify Citizens

Table 7: Individual Missions of the Flood Monitoring SoS

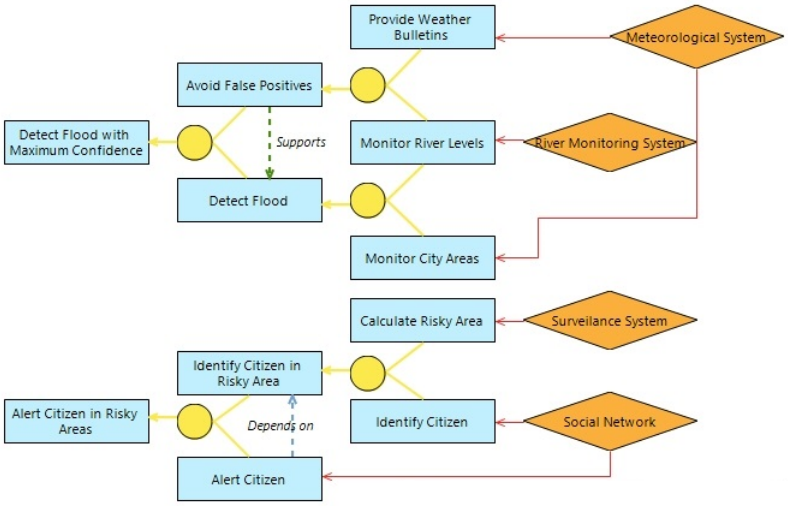


Figure 65: Mission Model and Responsibility Model for FMSoS

6.2.1 Definition

6.2.1.1 Mission Modeling

The first activity of the definition step is probably the most important activity in M2Arch: **the definition of the mission model**. As output, it produces a mKAOS mission model that describes the SoS as a whole, from its global missions to the capabilities and the data objects exchanged by the involved parts.

The FMSoS was introduced in Section 2.8, as shown in Fig. 5, such an SoS has two global missions, namely *Detect Flood with Maximum Confidence* and *Alert Citizen in Risky Areas*. These missions are refined into six individual missions assigned to four constituent systems, as described in Table 7.

To model these missions in mKAOS, we started by the global missions, using the top-down approach. These missions are refined into individual missions. Later, using the **Responsibility Model** we define the constituent systems and assign responsibilities over the individual missions, using the information of Table 7.

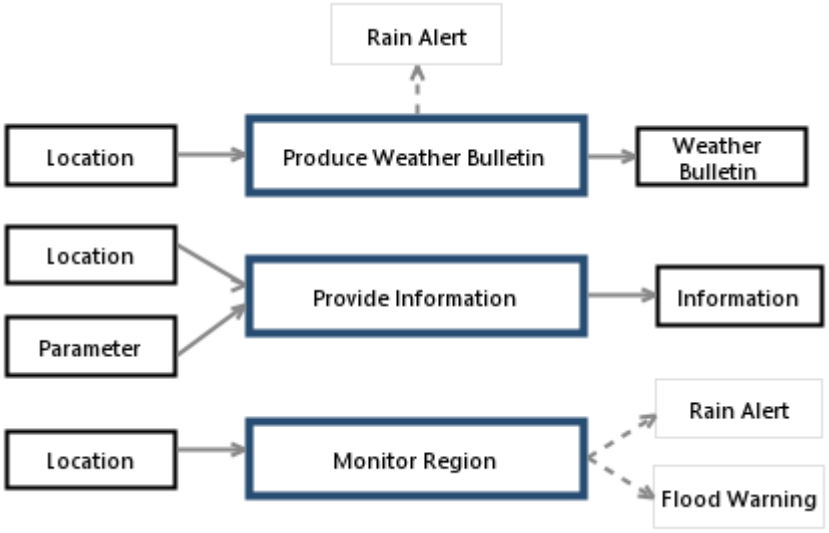


Figure 66: Capabilities of the meteorological system

Now, it is necessary to identify the capabilities of the constituent systems, that make them capable of achieving their individual missions by its own. This is done through the **Operational Capability Model**.

The first constituent system, the *meteorological system*, is capable of gathering data regarding weather, such as temperature, humidity, wind speed, wind direction,and rain amount. This information is collected by sensors and radars and provided in form of bulletins. As the data depend on the geographical location, the system receives as input the location and provides the data as soon as they become available. Fig. 66 shows the operational capabilities of the system as designed in mKAOS. The *Produce Weather Bulletin* capability receives a *Location* as input and produces a *Weather Bulletin*. It can also trigger a *Rain Alert* event, which can be provided before the bulletin completion.The *Provide Information* capability is responsible for providing a specific information (such as temperature, wind speed, etc.) given a *Location* and a *Parameter* (type of desired information). Finally, the *Monitor Region* capability receives a *Location* and keeps monitoring this region, triggering the *Rain Alert* and the *Flood Warning* events.

A second constituent system, the *surveillance system*, is capable of taking aerial images (using balloons, airplanes, satellites, etc.) of a given area. Fig. 67 shows the operational capability model for the surveillance system. Its only capability: *Provide Images*, receives a *Location* as input, providing an *Image* as output. The surveillance system is also responsible for calculating a **risky area**, that is represented by a list of locations. For doing so, it uses the capability *Calculate Risky Area*, taking a center Location and a range (represented as Integer) as input.

Figure 67: Capabilities of the surveillance system

Figure 68: Capabilities of the river monitoring system

The *River Monitoring System* is a constituent system that is also a SoS. It is composed of a group of sensors and gateways and operate together to monitor river levels in different spots in a riverbed. It is not necessary, however, to model another SoS in this context. Instead, we see it as a single constituent system, capable of providing the current water level of the river. Fig. 68 depicts the operational capability model of the River Monitoring System. This diagram presents a new concept: a capability that **does not require** any input, *Provide Water Level*.

Finally, the **Social Network** is one of the most complexes constituent systems in FMSoS. Some of the relevant capabilities are presented in Fig. 69. *Receive Message* is a capability that triggers an event: *Message Received*, representing a message the user received. The capability *Send Message* sends a *Message* to a given *Participant*. The *Social Networks* also allows to search participants, based on a *String* (name of participant) or *Location*. These two capabilities provides as output a list of *Participants*, that contains all the participants that met the search criteria.

Regarding communicational capabilities, the flood monitoring SoS has one important communicational capability. The *To Match Data* capability is responsible for providing a single accurate information based on two provided information. It represents the mechanism of fault tolerance of the system and it is also responsible for the implementation of the Detect False Positive emergent behavior. Both communicational capability and emergent behavior are defined in mKAOS as illustrated in Fig. 70. The *To Match Data* communicational capability receives two *Information* objects and provides a single *Information*. The information used by these communicational capabilities is provided by the *Meteorological System* and *River Monitoring System* constituent systems through the operational capabilities *Provide Information* and *Provide River Level Information*. The

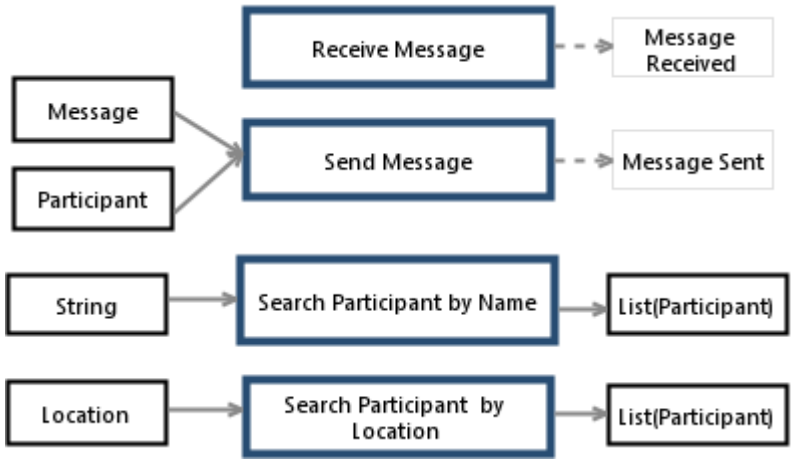


Figure 69: Capabilities of the social network

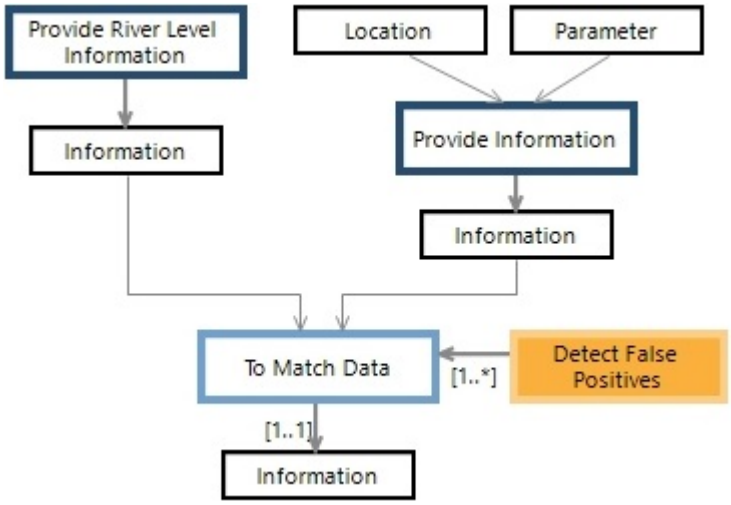


Figure 70: Communicational capability To Match Data

refinement process for this communicational capability produces a mediator with a duty and three connections, two input connections with the *Information* data type and one output connection with the *Information* data type, as shown in Fig. 77.

An important communication capability is the one responsible for using an *Alert*, produced by the *Meteorological System* and a *Participant*, provided by the *Social Network*, to build up a message to be sent, containing an alert to the participants. Fig. 71 presents this communicational capability: *Send Alert*.

Other communicational capabilities are present, regarding data exchange between different constituent systems, such as the capability *Location to SN*, described in Fig 72. This capability uses a list of *locations* given by the *Surveillance System* to provide the information to the *Social Network*, allowing the identification of the *participants* in the risky area.

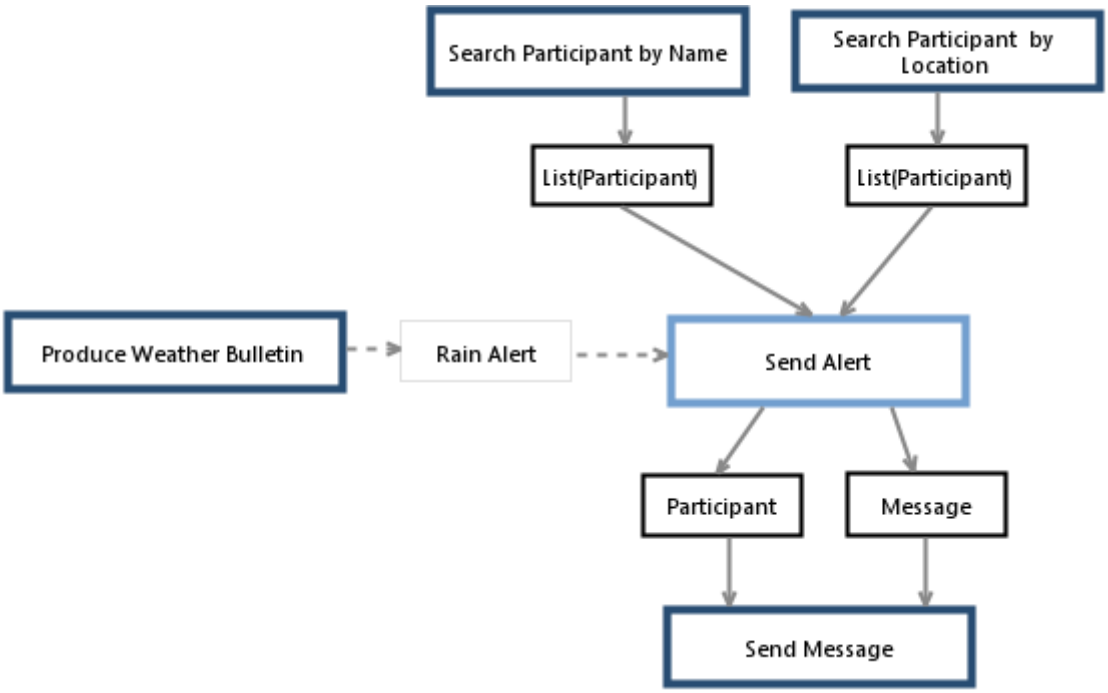


Figure 71: Communicational capability Send Alert

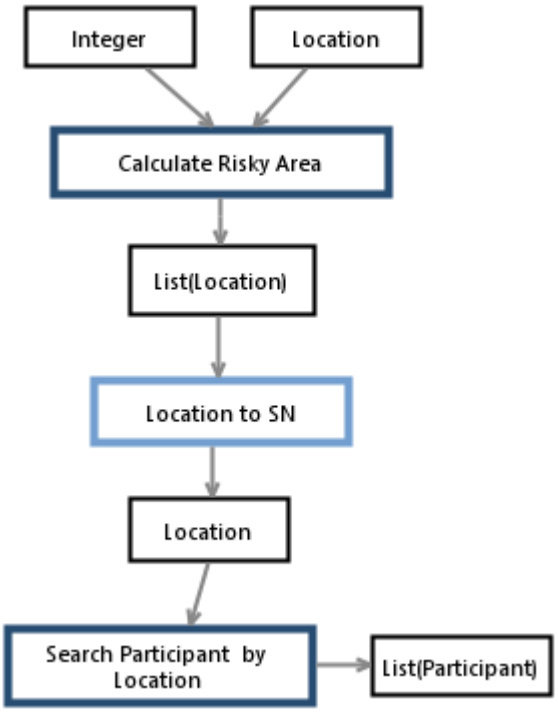


Figure 72: Communicational capability Location to SN

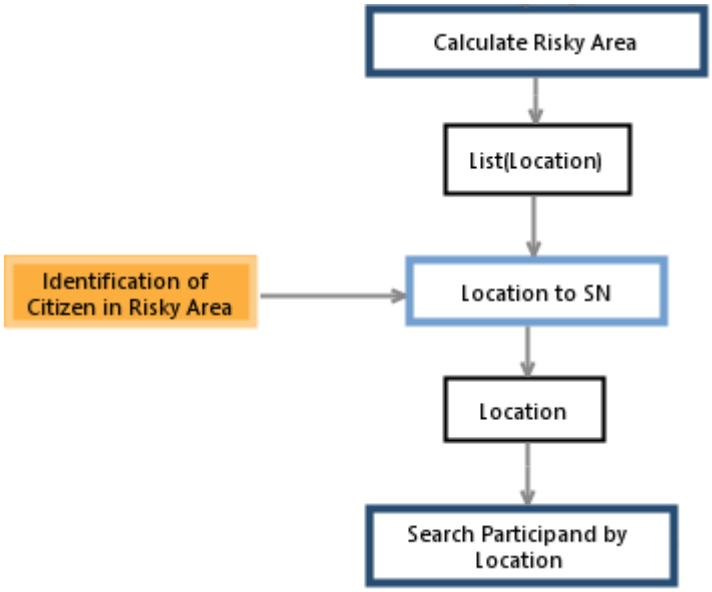


Figure 73: Emergent Behavior Identification of Citizen in Risky Area

Figure 74: Emergent Behavior Send Alert

The communicational capabilities enables emergent behaviors, that are defined in *Emergent Behavior Model*. *Emergent Behaviors* have a major influence on the achievement of a global mission. Therefore this diagram is fundamental for validation purposes.

One of the desired emergent behaviors is *Identification of Citizen in Risky Area*, that emerges from the interaction between the surveillance system and the social network, through the communicational capability *Location to SN*. One or more interactions of this kind emerges this behavior, as shown the mKAOS diagram in Fig 73. This emergent behavior influences the mission *Identify Citizen in Risky Area*.

Another emergent behavior is homonymous to the communicational capabilities it emerges from. Fig. 74 shows the emergent behavior Send Alert, that emerges from the communicational capability *Send Alert* and influences the achievement of the global mission *Alert Citizen in Risky Area*.

Finally, the FMSoS has a single constraint: the triggering of an *Alert* event by the *Meteorological System* must, eventually, trigger a *Message Sent* event on *Social Network*. This ensures that every time there is an Alert, someone will receive this alert. Fig. 75 shows the description of this constraint as a **Domain Invariant**, in formal mKAOS.

```
invariant AlertAlwaysSent {
   formalDef = always during 100 time units
      forall e:allOfType(Alert) {
         exists m:MessageSent m.message = e.message
      }
}
```

Figure 75: Domain Invariant for FMSoS

```
system MeteorologicalSystem ( ) is {
    gate ProvideInformation is {
        connection i1 is in {Information}
        connection e1 is out {Information}
        connection o1 is out {Information}
    }
    gate ProduceWeatherBulletin is {
        connection i1 is in {Information}
        connection i2 is in {Information}
        connection o1 is out {Information}
    }
    gate MonitorRegion is {
        connection i1 is in {Information}
        connection e1 is out {Information}
        connection e2 is out {Information}
    }
```

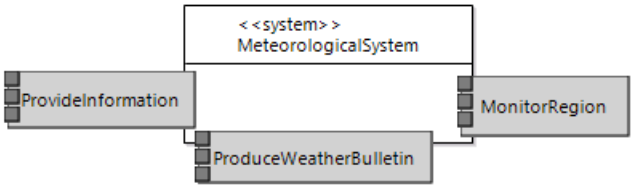


Figure 76: Partial definition of the *MeteorologicalSystem* in SosADL resulted from the mapping process from mKAOS

6.2.1.2 Automatic Mapping

Based on the mission model, the automatic mapping is capable of generating a SosADL architectural model that encompasses the element definitions and a coalition with the structure of the architecture.

An example of element definition that is generated is presented in Fig. 76. This partial description describe the *Meteorological System*, as well as the required data for the required connections. This construction includes a set of type definitions and a system with four gates, each one related to an operational capability of the system. For instance, the *ProduceWeatherbulletin* gate has three connections that represent the inputs and outputs for this operational capability.

```
mediator ToMatchData() is {
  duty mediate is {
    connection i1 is in{Information}
    connection i2 is in{Information}
    connection o1 is out{Information}
  }
}
```

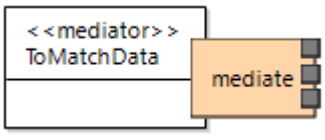


Figure 77: Mediator in SosADL generated from the To Match Data communicational capability described in mKAOS

```
architecture coalition () is {
    behavior coalition is compose {
        meteo is MeteorologicalSystem
        river is RiverMonitoringSystem
        med is ToMatchData
    }
    binding {
        unify one { meteo::sys::provideInfo } to one { med::mediate::i1 } and
        unify one { river::sys::provideInfo } to one { med::mediate::i2 }
    }
}
```

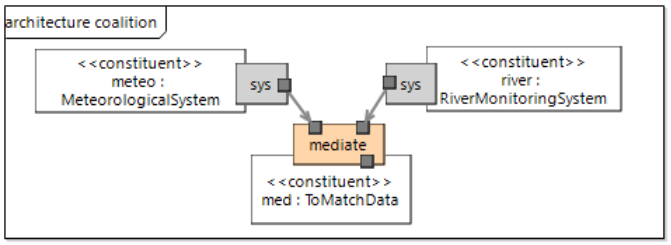


Figure 78: Coalition in SosADL representing the architecture of the flood monitoring SoS

The coalition representing the architecture of the flood monitoring SoS is built using the produced constituent systems and mediator. The bindings are based on the input/output links in mKAOS, in which the systems will interact through the parameters of the communicational capabilities. Additionally, the inputs and outputs of communicational capabilities not used by any individual constituent systems are bound to the SoS gates, through the relay instruction. Fig. 78 shows the produced architecture for the flood monitoring SoS based on the mKAOS mission models. In this partial description, two constituent systems (*MeteorologicalSystem* and *RiverMonitoringSystem*) and one mediator (*ToMatchData*) are defined, the latter handling the interaction between the former. The mediator takes data from both systems and produces an *Information* object that is used by the SoS.

6.2.1.3 Architectural Modeling

Although the overall structure is generated by the automatic mapping, it might be necessary to do some adjustments. In this case, specifically, no major change was required. However, it is still necessary to describe the behavior of the constituent systems and mediators, that cannot be automatically generated since mKAOS does not concern on system's behavior.

Except for the *River Monitoring System*, the internal behavior of the constituent systems is unknown. We choose, however, to treat all constituents as if they have unknown behavior, for simplification purposes.

Even constituent systems with unknown behavior can be expressed in SosADL and

hence supported by M2Arch. It is important, however, to be able to simulate their behavior, based on observation over these systems or their overall definitions. To simulate their behavior it is possible to use the *ExternalControllers*, described in Section 3.3.4. The automatic processes of verification and validation will not be able to execute if there is an unknown constituent system or mediator with no associated *ExternalController*.

We implemented a set of four *ExternalControllers*: *MeteorologicalController*, *RiverMonitoringController*, *SurveillanceController* and *SocialNetworkController*. These controllers are be responsible for implementing the interfaces of the constituent systems, reading the inputs and producing the outputs when requested.

Further, we know that sometimes a constituent system may be unable to respond. At this stage of the modeling process, we are not sure about the causes and this anomalous behavior do not happen often. Although the controllers are implemented to simulate the constituent systems, we also implemented a failing mechanism that defines a response rate of 99.9%, which means that the controller will respond properly to 99.9% of requests. In the 0.1% left, the controller consumes the input but does not generate any result. This allows us to simulate situations in which there is a network unavailability or any structural issue, but also some misfunction in the constituent system.

Some of the controllers, as the one responsible for the *River Monitoring System*, uses a stochastic process to produce the values for river levels. The produced values are in a normal distribution, with a low probability of providing a water level that represents a flood.

Also, to allow a more accurate simulation, we implemented some data exchange between the controllers. The *RiverMonitoringController* interacts with the *MeteorologicalController*, to allow their data to be cohesive, since their associated constituent systems make measurements in a common physical environment. If the *MeteorologicalController* produces a rain alert, the *RiverMonitoringController* will provide higher river level measures. The opposite occurs if no rain alert was produced for some time: the *RiverMonitoringController* will provide lower river level measures.

On the other hand, the mediators are part of the constituent system and therefore we can define their guarantees, although an *ExternalController* could also be built in this case. Since all mediators perform simple operations, we choose to describe their behavior using SosADL.

One of the most important mediators in the FMSoS is the *SendAlert* mediator, au-

```
mediator SendAlert(alert:RainAlert,
  p1:Sequence{Participant} , p2:Sequence{Participant}) is {
  duty mediate is {
    connection i1 is in{Sequence{Participant}}
    connection i2 is in{Sequence{Participant}}
    connection e1 is in{RainAlert}
    connection o1 is out{Participant}
    connection o2 is out{Message}
  }
  assume {
    anyaction
  }
  guarantee {
    value message is Message = e1.message
    protocol behavior is {
      repeat {
        choose {
          via mediate::i1 receive participant {
        } or {
          via mediate::i2 receive participant
        }
        via mediate::o1 send participant
        via mediate::o2 send message
      }
    }
  }
}
```

Figure 79: Behavior of mediator SendAlert

tomatically generated based on the *Send Alert* communicational capability, previously described in Fig. 71. Fig. 79 depicts the behavior of the *SendAlert* mediator, that takes a *Rain Alert* and a set of *Participants*, generating a *Message* that will be sent to each participant, containing is the message in the *Rain Alert*.

The mediator *SendAlert* receives a *Participant*, through either *p1* or *p2*, and sends a message that contains the *RainAlert* message and send it to this *Participant*.

6.2.2 Verification

M2Arch V&V module, responsible for verification and validation, relies on the SosADL Simulator and PlasmaLab. Therefore, it requires some configuration to be able to perform the automatic routines.

First, as the SosADL Simulator is able to simulate only concrete SosADL architectures, it is necessary to generate these concrete architectures before starting. Currently, we faced some issues to execute Guessi's solution (GUESSI; OQUENDO; NAKAGAWA, 2016) to perform this generation. We succeeded after a few attempts and generated a concrete architecture identical to the initial, meaning that the abstract architecture was also a concrete architecture for the given environment. In this context, our generator environment

```
# Simulator Configuration File
# General properties
[control]
iterations=100
reportType=all

# External controllers
[controllers]
pluginsDir=\_plugins
rms=org.archware.fmsos.controllers.RiverMonitoringController
meteorological=org.archware.fmsos.controllers.MeteorologicalController
network=org.archware.fmsos.controllers.SocialNetworkController
surveillance= org.archware.fmsos.controllers.SurveillanceController

# Predefined data injections
[data]
```

Figure 80: Simulator Configuration for FMSoS

```
# Model Checking Configuration file
# This is a comment line

# general properties
missionModel=./fmsos.mkaos
type=verification
constraints=all
algorithm=montecarlo
plasmaParam=-A"Total Samples"=100 -progress
```

Figure 81: Verification Configuration for FMSoS

encompassed only a single possible constituent system of each kind, what explains the production of a single concrete architecture.

With the concrete architecture ready, aiming to improve quality, M2Arch proposes the use of automated verification to check the architecture for domain properties and constraints. For doing so, M2Arch requires a configuration file for the model checker and simulator. The simulator configuration file for the FMSoS is presented in Fig. 80. This file defines the *ExternalControllers* for the constituent systems and the parameters of the simulator, like, for instance, the number of iterations and report type.

The verification configuration is presented in Fig. 81. It specifies that our model checker will use the *Morte Carlo* algorithm to perform 100 simulations, verifying all constraints.

Finally, we started the procedure to automatically verify the property within the architecture. Table 8 presents our results. We did three experimentations, varying the

Constraint	Success Rate	Samples	Time
AlertAlwaysSent	1.0	100	14281ms
AlertAlwaysSent	0.99	1000	182310ms
AlertAlwaysSent	0.9878	10000	2165701ms

Table 8: Results of automatic verification

```
# Model Checking Configuration file
# This is a comment line

# general properties
missionModel=./fmsos.mkaos
type=validation
algorithm=montecarlo
plasmaParam=-A"Total Samples"=100 -progress
```

Figure 82: Verification Configuration for Validation of FMSoS

number of samples used by PlasmaLab, using a machine with a Intel i7 processor, 8gb RAM, Windows 10 system. We ascribe the performance of this experiment mainly to the SosADL Simulator, since PlasmaLab appears to request the new states faster than the simulator is capable of processing it.

Based on Table 8, we found that the invariant *AlertAlwaysSent* is maintained in around 99% of the simulations. The failing rate is associated with the implementations of the *ExternalControllers*, that intentionally fail, eventually. We decided that this failing rate acceptable in the context of this system.

6.2.3 Validation

Validation architectures, in M2Arch encompasses an automatic validation of formally described missions and a manual validation based on the simulation.

Similarly to the verification purpose, the automatic validation requires a configuration file with some parameters for PlasmaLab. The configuration for FMSoS is presented in Fig. 82. In this configuration, we specify that the V&V module will perform an automatic validation, focusing on formally described missions, but it is also possible to focus on formally described emergent behaviors.

However, we found some limitation on formal mKAOS regarding constraint definition. Since the language does not contain any architecture-related information, it is necessary to improve some of the constraints for the concrete architecture, detailing the connection that will interact with the property, whenever it applies. Fig. 83 shows an example of

```
emegent behavior DetectFalsePositives {
   formalDef = always during 100 time units
      forall m:allOfType(ToMatchData) {
          ToMatchData.informationOut !=null implies
             (ToMatchData.infIn1 - ToMatchData.infIn2) < 1 and
                 (ToMatchData.infOut = ToMatchData.infIn1
                 or ToMatchData.infOut = ToMatchData.infIn2)
      }
}
```

(a)

```
emegent behavior DetectFalsePositives {
   formalDef = always during 100 time units
      forall m:allOfType(ToMatchData) {
          ToMatchData.mediatie.o1 !=null implies
             (ToMatchData.mediate.i1 - ToMatchData.mediate.i2) < 1 and
                 (ToMatchData.mediate.o1 = ToMatchData.mediate.i1
                 or ToMatchData.mediate.o1 = ToMatchData.mediate.i2)
      }
}
```

(b)

Figure 83: Improvement of formal mKAOS Formal Definition

formal description of the emergent behavior *DetectFalsePositives*. Fig. 83a, shows the original formal definition, which is based on the mission model alone. Fig. 83b shows the updated definition of this behavior, including information generated by the automatic mapping.

Once updated the formal definitions of missions and emergent behaviors, the V&V module is capable of isolating these formulas and invoking PlasmaLab to check the existence of the emergent behaviors and the achievement rate of the missions. After updating the formal definition of the individual missions and some intermediary missions of FMSoS, we obtained the results presented by Table 9. In this study, the global missions are simply a combination of its sub-missions.

The V&V module took 11m22s to evaluate the model to produce these results, in a Core i7, 8gb RAM, Windows 10 system, with 100 samples. We expect longer times for more precise validations, using a higher number of samples.

After these automated processes, the manual validation consists on identifying misleads in the simulation itself, checking if the architecture is behaving as planned. For this case study, we found no misdirection in the planned execution path and no constituent behave differently of its plans.

Mission	Rate
Detect Floods with Maximum Confidence	95%
Alert Citizen in Risky Areas	99%
Avoid False Positives	98%
Detect Flood	97%
Identify Citizen in Risky Area	100%
Alert Citizen	99%
Identify Citizen	100%
Calculate Risky Area	100%
Monitor City Areas	99%
Monitor River Levels	98%
Provide Weather Bulletins	100%

Table 9: Mission achievement rate for FMSoS

```
Step 71.
Step 72.
Executing RiverMonitoringSystem...
Constituent Failed, input was consumed but no output will be produced
Done executing RiverMonitoringSystem.
Step 73.
```

Figure 84: Constituent System Fail in Simulation Report

For performing the manual validation, though, we use the simulation reports generated by the automatic validation procedure. Since the validation report includes data regarding in which circumstances the missions failed, it is possible to identify what caused the failure. For this study, all the failures were caused by the intentional failing mechanism introduced in the *ExternalControllers*. For instance, Fig. 84 shows a simulation report in which the constituent *RiverMonitoringSystem* intentionally failed. Fig. 85 displays the source code of the *ExternalController* that provoked this failure.

6.2.4 Discussion

The overall structure of the produced architecture, is very similar to the existing architecture, they differ on coupling and some gates presented a different definition. Along this Section, the previously existing architecture will be referred as *Arch1*, and the model produced through M2Arch will be called *Arch-M2Arch*.

Arch1 was produced previously to the definition of M2Arch, using no specific methodology. This architecture was based on the textual descriptions and available documentation of the FMSoS (HUGHES et al., 2011; DEGROSSI; AMARAL; VASCONCELOS, 2013). This architecture was used to analyze the needs of M2Arch, in terms of mechanisms and tech-

```
@Override
public void execute(Context context) {
    IValue input = context.getValue("provideImagems.i1");
    if (!fail(input,context) && input.getValue()!=null) {
        // consume input
        input.setValue(this, null);
        // set output based on a predefined table
        context.getValue("provideImages.o1").setValue(this, images.get(input));
    }
}

private boolean fail(IValue value, Context context) {
    Random i = new Random(); // initialize randomizer
    int r = i.nextInt() % 1000; // 0.1% failure chance
    if (r==0) {
        // execution fails, consume the value but do not produce anything
        value.setValue(this, null);
        System.out.println("Constituent Failed, input was consumed but no output will be produced");
        return true;
    }
    return false;
}
```

Figure 85: External Controller for *RiverMonitoringSystem*

niques.

Fig. 86 presents an overview of Arch1, showing the structure of the architecture in the SosADL view tool. This architecture encompasses four constituent systems, two mediators and a set of 20 connections. The whole SosADL description of Arch1 architecture is available at Appendix C.

On the other hand, the architecture *Arch-M2Arch*, produced through M2Arch have its structure presented by Fig. 87. This architecture also encompasses four constituent systems, however, it has six mediators and 17 connections. The whole SosADL description of Arch-M2Arch is available at Appendix D.

The main difference between the architectures is the number of mediators. Since mediators in Arch-M2Arch were generated based on the possible interactions, they have a simpler interface and perform fewer operations. The mediators of Arch1 are very overloaded, sometimes performing more complex operations. This severely impacts the resilience of the architecture. Since Arch-M2Arch has simpler mediators, it has fewer failure points and is easier to maintain.

Furthermore, we detected some additional relations between constituent systems in Arch-M2Arch, that were caused by a transformation rule. The process to establish a *unify* considers only the data types in mKAOS to produce a *unify* in SosADL. This may lead to the creation of unifications that were not predicted. We minimized this behavior by improving the mapping, double checking the data types and generated connections to minimize its occurrence. It is important to highlight that, due to the dynamic nature of

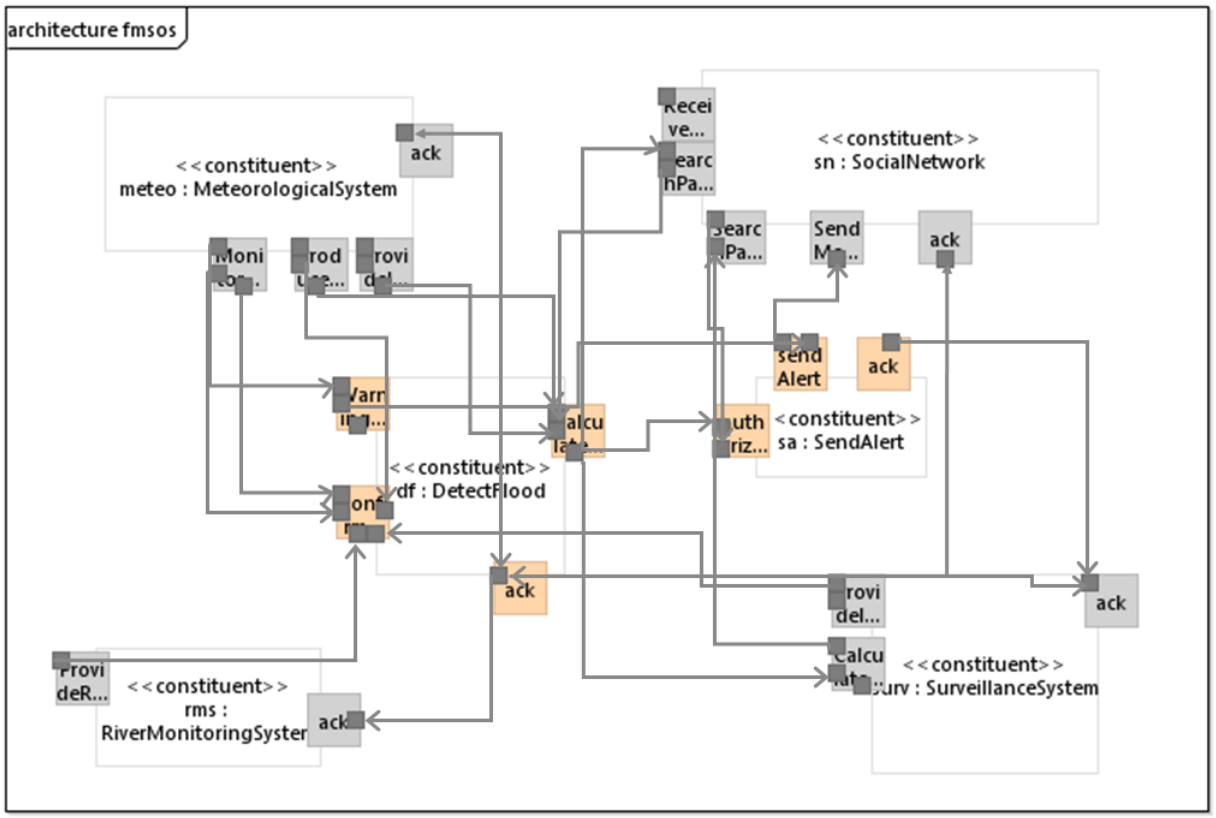


Figure 86: Overview of *Arch1*

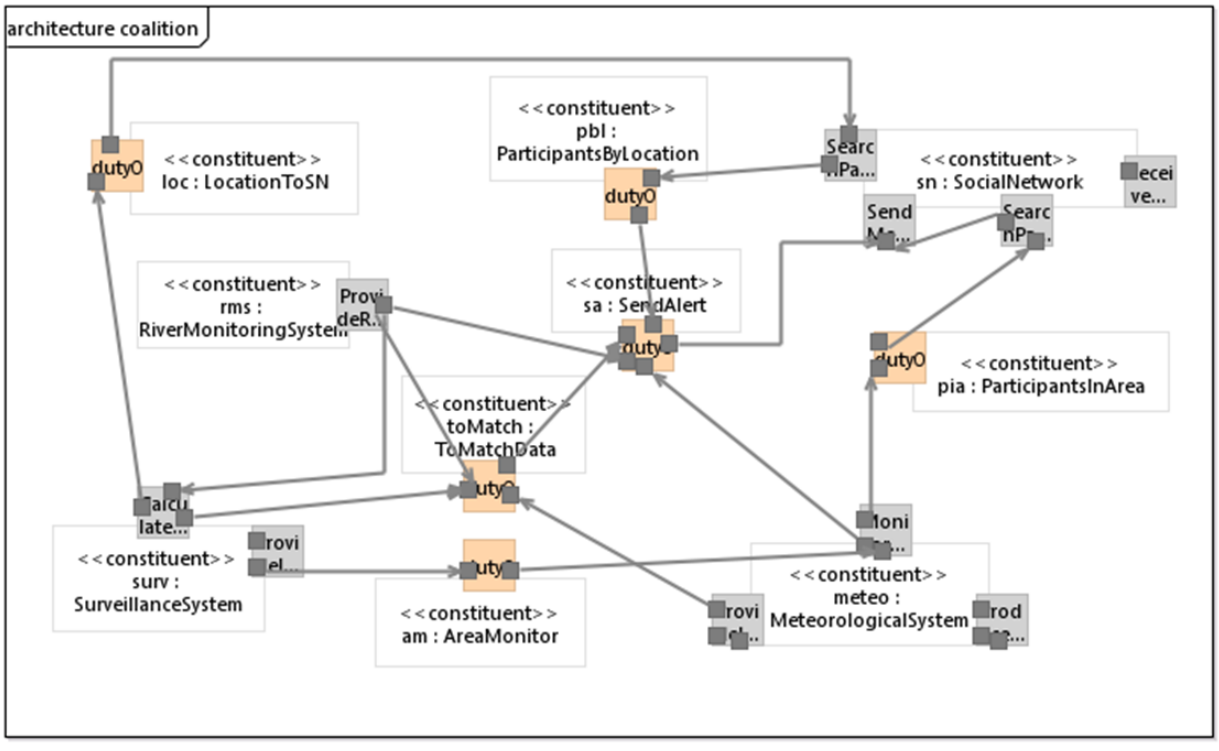


Figure 87: Overview of *Arch-M2Arch*

System	Arch1		Arch-M2Arch	
	IC	**OC**	**IC**	**OC**
River Monitoring System	1	3	0	3
Meteorological System	1	6	0	3
Social Network	4	3	3	2
Surveillance System	3	2	1	3

Table 10: Connections of Constituent Systems of FMSoS

Mediator	Arch1		Arch-M2Arch	
	AC	**EC**	**AC**	**EC**
Detect Flood	7	4	-	-
Send Alert	3	3	-	-
Area Monitor	-	-	1	1
LocationToSN	-	-	1	1
ParticipantsByLocation	-	-	1	1
SendAlert	-	-	3	1
ToMatchData	-	-	3	1
ParticipantsInArea	-	-	1	1

Table 11: Connections of Mediators of FMSoS

mediators that will perform a mediation only when the constituent systems require, we observed no impact of this issue on simulation: the architecture behaves exactly in the same way when we removed the "extra" relations.

To evaluate the architectures with an objective view, we performed an **interactions** analysis. We evaluate how many interactions the constituent systems do with other constituent systems, based on the number of connections that are being used by any relation.

We organized these connections as input connections (IC) and output connections (OC), that are summarized by Table 10. Furthermore, we also evaluated the number of connections of mediators, summarized by Table 11.

The number of connections were different and, more specifically, larger in *Arch1*. *Arch-M2Arch* uses a greater number of mediators, simplifying the communication between the constituent systems. *Arch1* has fewer mediators, but these are overloaded with several connections. The increased number of connections hampers the evolution process of the SoS, since a change in an overloaded mediator or constituent has impacts on several interactions.

Finally, we evaluated the degree of commitment of the architecture within the mission model. For doing so, we compared the achievement rate of both architectures using the automatic validation process with the same mission model. For a better accuracy of Plas-

Mission	Achievement Rate	
	Arch1	**Arch-M2Arch**
Detect Floods with Maximum Confidence	74.87%	95.20%
Alert Citizen in Risky Areas	91.66%	98.91 %
Avoid False Positives	87.64%	98.22%
Detect Flood	85.44%	96.93%
Identify Citizen in Risky Area	91,78%	99,71%
Alert Citizen	99.3%	99.2%
Identify Citizen	99.88%	99.9%
Calculate Risky Area	99.91%	99.81%
Monitor City Areas	98.52%	98.6%
Monitor River Levels	98.91%	98.31%
Provide Weather Bulletins	98.72%	99.91%

Table 12: Mission Achievement Rates of Architectures for FMSoS

maLab on the automatic validation, the number of samples was increased to 10.000, that increases the checking process time. Both architectures used the same simulation configuration, external controllers and mission model. Therefore, any difference relies exclusively on the architecture itself.

Table 12 presents the mission achievement rate of both configurations. Architecture Arch1 presents a higher failure rate, we associate this to the overloaded mediator: whenever it fails, the architecture fails in multiple missions at once.

It is worth highlighting that *Arch1* executes three times faster than *Arch-M2Arch*. We assign this difference to the increased number of mediators in *Arch-M2Arch*, allowing faster data exchange due to the parallelism that the simulator implements for the mediators.

Although we cannot associate the improved performance to M2Arch, this evaluation allows us to make a few conclusions about the methodology. Since M2Arch generated the topology of the system, with few or no changes to be made, the lower effort to develop using the methodology, the efficacy and the efficiency of the produced architecture allows us to suggest M2Arch accomplishes what it intends to, as a pioneer mission-based methodology to develop SoS architectures.

7 Related Work

This chapter discusses related and complimentary work found until September 2018. We were looking for works that deals with a refinement, methodology or process that bridge missions and software architecture of SoS.

Although we found no study directly addressed to this topic, we looked for studies that might somehow help answering the research questions presented in Section 1.2. We divided these works in three categories: (i) **alternative ADLs**, that may provide a better solution than SosADL; (ii) **mission languages**, that might present a different representation and an underlying formalism; and (iii) **refinement methodologies**, that would provide valuable knowledge for our work. During the production of this work, we found no relevant works on (i) and (ii). However, regarding (iii), a bibliographic review found interesting works, presented in Section 7.1. Finally, Section 7.3 presents a brief discussion about the current state of art, emphasizing the perspectives for the domain.

In addiction, we are aware that missions are closely related to requirements, thus, we choose a couple of works apart from SoS context, to illustrate the relationship between requirements and architecture, presented in Section 7.2. These works are chosen specifically since they use KAOS at some point of the modeling process or, at least, the goal-oriented approach used by KAOS. Since mKAOS is an extension of KAOS, these works potentially present some relevant topics to this study.

7.1 Systems-of-Systems Approaches

In this section we present the approaches for SoS, however, there is lack of works that uses missions as starting points. A remarkable study in SoS domain is COMPASS, that proposes a complete framework for developing SoS using a conventional requirements approach, presented in Section 7.1.1. Another interesting work is more theoretical, Haley and Nuseigeh (HALEY; NUSEIBEH, 2008) uses i* (YU, 2009) and Tropos (GIUNCHIGLIA;

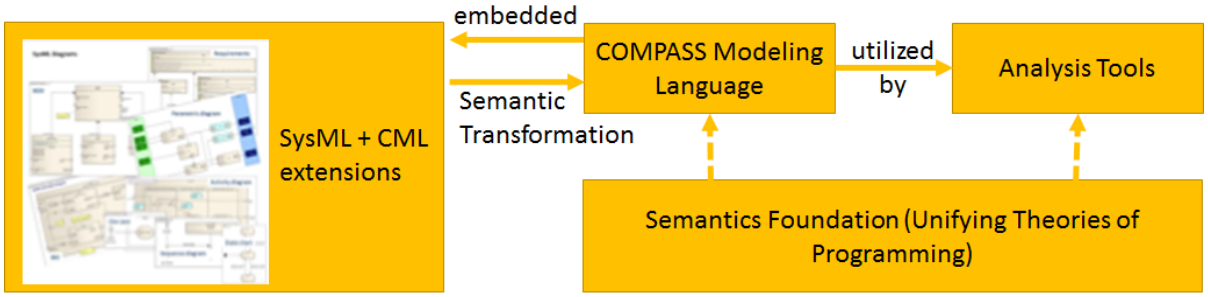


Figure 88: Overview of COMPASS approach

MYLOPOULOS; PERINI, 2003) as starting point for developing a methodology requirements engineering process to develop SoS architectures.

7.1.1 COMPASS

Comprehensive Modeling for Advanced Systems of Systems (COMPASS) (FITZGERALD; BRYANS; PAYNE, 2012; FITZGERALD; LARSEN; WOODCOCK, 2014) is a framework and methodology for building and maintaining systems-of-systems. It encompasses a set of tools, methods and formalisms for modeling and analyzing SoS with an underlying formal notation.

COMPASS is the most advanced and complete work we found in the literature, it concerns in SoS modeling from requirements to architecture. The approach encompasses all development steps, from requirements to architecture, formal verification and validation.

It uses SysML for the whole modeling process, from requirements engineering to architectural description, although it presents an extension for the language to provide a formal support for architectural descriptions. The architectural models are fully refined into CML (COMPASS Modeling Language), a formal, executable language that allows model simulation and analysis. CML is theoretically based on the Unifying Theories of Programming (UTP), (HOARE; JIFENG, 1998) a model semantics framework. Fig. 88 [1] summarizes the framework structure.

COMPASS relies on **competency viewpoints** to define roles and activities to establish an specific development process for each domain of SoS. The competency viewpoints are four: (i) *Competency Framework Definition*, that essentially defines the ontology to be used for the domain, (ii), *Competency Level Definition*, that defines the roles for the implementation process, (iii) *Competency Scope Definition*, that defines responsibilities over each activity of the development process, (iv) *Competency Profile Definition* assigns roles

[1] Based on `http://www.compass-research.eu/`

```
actions
   act_mainEntryPointState = act_generic_transtionsLoop /_\ ch_init->Skip
   act_generic_transtionsLoop =
   mu X@
       (
        (dcl s:StreamingLayerReplyEvent @ ch_genericEvent?e->( s :=
convertEventToState(e) );
         if isLegalTransition(s)
          then (Stramingtransition(s);ch_streamingReply!(currentState)->X)
         else (ch_streamingLayerError.<STATE_ERROR>->X)
        )
       )

   @ch_init ->initObject();act_mainEntryPointState
```

Figure 89: Example of CML code

to stakeholders. Based on these viewpoints, the roles and activities are defined to produce a multi-view architecture. The combination of these four views produces a well-defined process for SoS development.

In terms of requirements modeling, the approach uses traditional SysML requirements model to define the over-cited development process. The validation of the process is manual and consists of checking whether this process is complying to the specified requirements.

In terms of architectural modeling, COMPASS enhances SysML with CML code. CML is a formal language that defines semantics logic for the actions and activities defined in SysML. The embodiment of CML code within SysML allows the architecture to be simulated and verified. The process to produce architecture is based on a set of guidelines using the competency viewpoints to refine the requirements to the architectural level, thus supporting traceability between those requirements and elements in the architecture.

COMPASS also concerns in verification, hence, CML includes mechanisms for definition of constraints and a state logic. To ensure the set of required properties of a given communication, these contracts can be established in CML and are verified at simulation time. COMPASS suggests the use of contracts on communication processes, which can be verified using a formal simulator. Fig. 89 shows an example of CML specification of contract of a streaming service SoS: (i) “A valid interface implementation must always reply on a request”, which is checked by most of the code; and (ii) “if a state transition fails, a valid interface implementation stays in the current state”, which is verified through the first line, that skips the transition process if the state fails.

COMPASS approach is an extensive, well-defined process to architectural definition

of SoS, so far, it is the most advanced methodology that exists. However, it uses the usual concept of requirements instead of missions. Mission is a concept more adequate to the SoS context, since it naturally handles the dynamic nature of this kind of system. Since COMPASS project was developed before the arising of mission description languages, the approach uses requirements as starting point for the modeling process. We consider this decision outdated, since now we are capable of accurately describing missions and its specificities.

Another important point is the mechanism used to produce and represent architectures. COMPASS presents a set of guidelines to produce and validate architectures from requirement diagrams, however, the process is mostly manual. The description process is partially supported by descriptive tools, but instead of defining specific DSLs, the proposal enhances existing ones. Specifically, in the architectural description, COMPASS enhances SysML, a widely accepted ADL. However, SysML has some limitations regarding dynamism, since it was designed to define static systems.

CML extension adds formalism to the language, but it does not handle the dynamism of SoS. Since SoS are systems which configuration can change at runtime, constituent systems can come and go. We believe the use of contracts on communications is a successful decision, due to the potential heterogeneity and behavioral uncertainty of constituent systems. These characteristics requires the architecture to be able to handle different systems and protocols, the use of contracts upholds this process. However, it does not support dynamic reconfiguration, we consider this as a major limitation of COMPASS.

7.1.2 Haley and Nuseibeh's Work

The approach proposed by Haley and Nuseibeh (HALEY; NUSEIBEH, 2008) proposes a multidisciplinary process, using Software Engineering and Philosophy concepts together to produce an enhancement to the i*/Tropos approaches to develop SoS requirements, bridging the enhanced models with the software architecture through analysis.

Structured as a four-step process, the proposal aims to enhance requirements models in order to obtain a more detailed, refined model. The main objective is to allow a better understanding of the requirements, that will be used to describe software architecture. The process iterates over both architectural and requirements models, which helps to understand the impact of the requirements on the architecture as long as it is being build.

The process is not sequential, and the analyst can start by any step. It is necessary

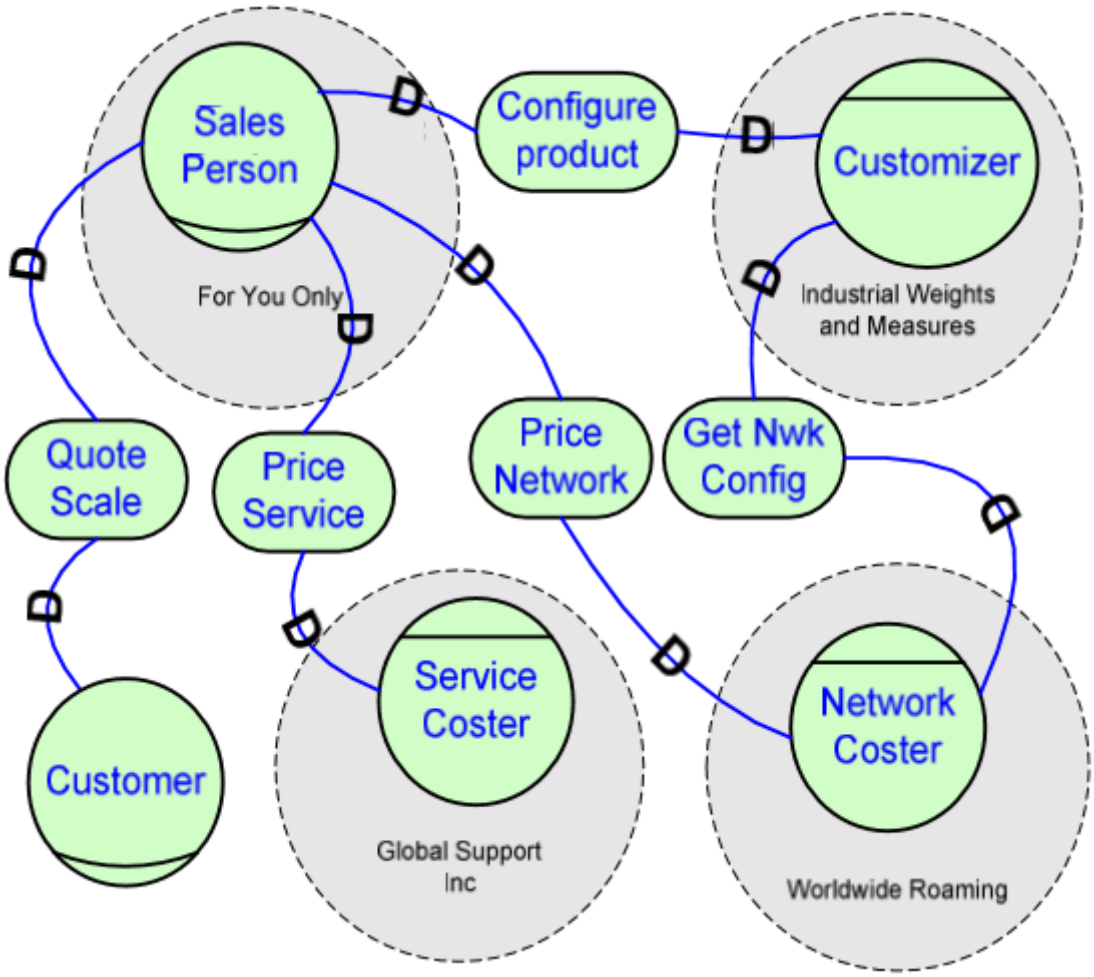


Figure 90: Example i^* diagram

to: (i) define the existing systems' behavior with i^* (YU, 2009)/Tropos (BRESCIANI et al., 2004; GIUNCHIGLIA; MYLOPOULOS; PERINI, 2003); (ii) describe the existing systems' architecture, using problem diagrams (JACKSON, 2001); (iii) describe the future, post-integration, SoS architecture; and (iv) describe the post-integration SoS behavior. After these steps, the approach proposes an analysis mechanism for correctness.

To model the requirements, the proposal uses i^*. The focuses of the requirements model are the agents and intention points of view. i^* shows delegation between agents and responsibilities, allowing variation along levels of detail. Using this approach, agents may be computer systems, humans or organizations. Fig. 90 [2] shows an example i^* diagram. In this example, the intentional model is shown for a sales system. Circles represents *actors*, ovals are *goals* the one actor delegate to another.

Architectural models are built using a variation of Jackson's problem diagrams (JACK-

[2]Extracted from (HALEY; NUSEIBEH, 2008)

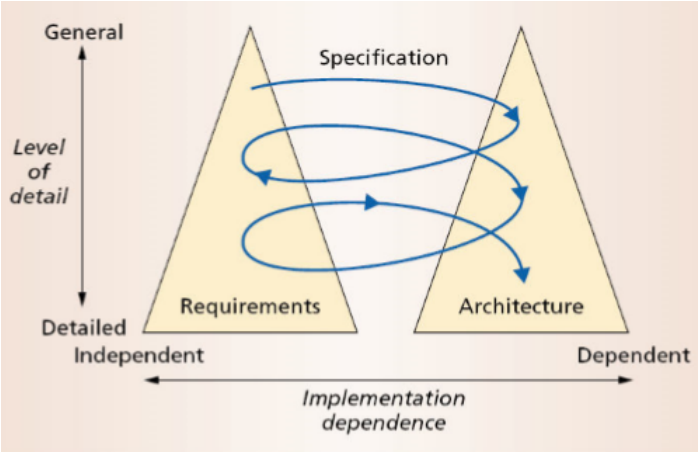


Figure 91: Twin Peaks Model

SON, 2001). In these diagrams, the systems are described in terms of physical domains and connections between them. It is important to highlight that this approach is very unusual, especially since it does not detail the interfaces of the systems in terms of data.

To produce the architecture, the proposal suggests the use of the Twin Peaks model (NUSEIBEH, 2001), presented in Fig. 91 [3]. Twin Peaks model consists of building the architecture a requirement per time, in a cyclic approach. This allows the architect to foresee the impact of a requirement in the architecture and favors traceability. During the architectural design, the architect must identify the capabilities of the constituent systems and the required capabilities. To provide the required capabilities, the architecture must fulfill a set of assumptions, that are verified in a final step of the architectural modeling.

Such process to define the architecture lacks on specific guidelines or rules. The architecture will be built without a well-defined framework, technique or methodology, in a very subjective manner. Furthermore, the language used for architectural modeling is not an ADL, therefore the concepts of software architecture are not present.

To validate the final architecture, the proposal simply verifies each assumption. The architecture is considered valid if every assumption is satisfiable. However, such process is completely manual without tool support. This work does not provide a clear mechanism for verification of architectural properties, although i*/Tropos are able to express some constraints.

Another important limitation of this approach is the lack of concern in the dynamism inner to SoS. Such as COMPASS, this approach does not give special attention to the dynamism of SoS and lacks representations of dynamic structures. Also, the study does

[3]Extracted from (HALEY; NUSEIBEH, 2008)

not concern on emergent behaviors and many aspects of SoS, such the heterogeneity and the behavioral uncertainty on the constituent systems.

7.2 Requirements Engineering Approaches

The relation between requirements and architecture exists since the conception of both domains. There are many tools, approaches and methods to derive and validate requirements and architectures. In this Section, we will cite a few approaches to illustrate the state-of-the-art.

The first approach we will discuss is KAOS (LAMSWEERDE, 2009). The methodology, homonymous to the language we extended to produce mKAOS, is briefly presented in Section 7.2.1. We also present two additional approaches, gathered by Avgeriou et al (AVGERIOU JOHN GRUNDY, 2011), that relates architecture and requirements. We present these approaches in Sections 7.2.2 and 7.2.3. Among the existing studies, we choose those two since they rely specifically on goal-oriented solutions, which appears to be closer to mission modeling, as discussed in (SILVA; BATISTA; OQUENDO, 2015).

7.2.1 KAOS

van Lamswerde (LAMSWEERDE, 2003) proposed a goal-oriented approach for architectural design based on KAOS. It defines a mechanism for deriving architectures from KAOS models, inspiring the solution proposed by this thesis.

KAOS' approach is based on the goal models, that must be defined following four steps: (i) **goal modeling**: defining the tree-like structure for goals; (ii) **object modeling**: entities, events, attributes derived from the goals; (iii) **agent modeling**: identification of agents and elicitation of its capabilities based on the goal models; (iv) **operationalization**: definition of operations in terms of capabilities that the agents are capable of performing.

For quality evaluation, the goals are formalized using temporal logic, aiming to prescribe intended behavior. This severely impacts in the process, guiding the architects and enabling generation of behavioral descriptions. In this context, however, the author superficially describes how it could be done.

The approach is very straightforward, extending the operationalization step to the architecture level. It consists on refining agents, entities, and events to an architectural

description language. Furthermore, it uses pattern analysis to select architectural styles that may achieve non-functional requirements. An abstract architecture is produced from this approach, which is refined using domain-specific constraints to produce a concrete architecture.

Validation of software architectures, using KAOS' approach, is essentially manual and relies on the notable traceability promoted by the methodology. Due to KAOS' (the language) structure, it is simple for the architect to identify how each requirement is implemented. Regarding validation, this approach concerns only on the non-functional requirements, that are expressed using the underlying formalism in LTL and can be verified using some tools, such as **Objectiver** [4]. It is worth mentioning that Objectiver, the main tool that implements the KAOS' methodology, is commercial with no free versions, although a trial is possible.

7.2.2 Goal-Oriented Software Architecting

Goal-Oriented Software Architecting (GOSA) (CHUNG et al., 2011) is a high-level, three-step process to derive architectures from goal models. Fig 92 [5] shows an overview of the development process, in which the first step: *Goal-Oriented Requirements Analysis* is divided in three stages: (i) *Domain Model*; (ii) *Hardgoals*; (iii) *Softgoals*. The second step is the *Logical Architecture Derivation*, followed by the *Concrete Architecture Derivation*.

During the first step, the requirements analyst must define a goal model, using any existing goal-modeling language, such as KAOS. Then, it is necessary to define hardgoals and softgoals. **Hardgoals** are goals that must be achieved. For this approach they are essentially Functional Requirements (FR) that must be achieved by the system at the design point. Given the importance and the impact of a hardgoal, the proposal includes exploring alternative tasks to achieve each hardgoal, in order to select the most adequate ones. The set of selected tasks are then assigned to agents, that will be responsible for implementing it. **Softgoals** are goals that the system may be unable to achieve at some point at runtime, although those goals are desirable. For this approach, they are essentially Non-Functional Requirements (NFRs) , since they have less clear-cut definition and achievement criteria. These softgoals are used to analyze the architecture, identifying the decision that impacts on each softgoal and selecting the most adequate one.

The second step is the *Logical Architecture Derivation*. It involves establishing a

[4] `http://http://www.objectiver.com`
[5] Extracted from (CHUNG et al., 2011)

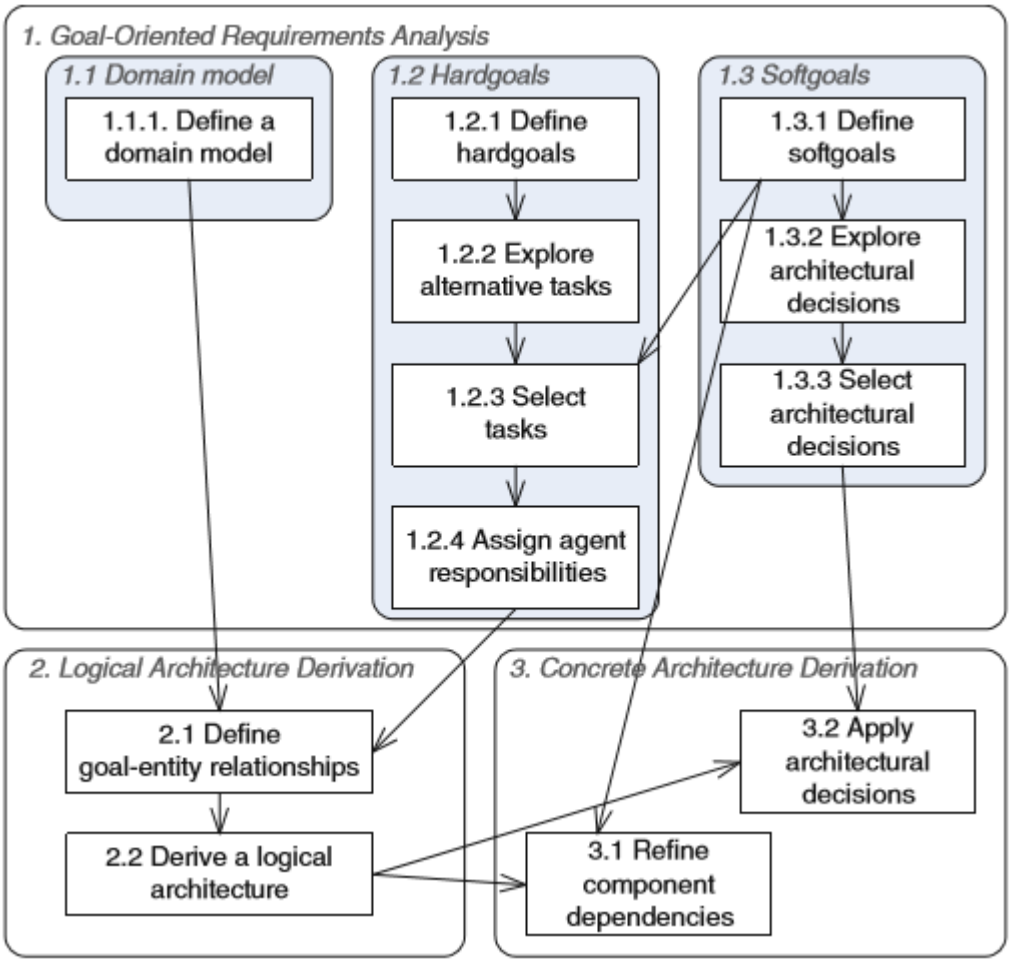


Figure 92: The goal-oriented software architecting process

hardgoal-entity relationship, in the sense of identifying how the hardgoals affect the entities of the goal model. After this first step, the architect can use the goal model and the goal-entity to define the logical architecture.

To establish the logical architecture, the architect starts by defining the process components. A process component is defined based on the relationships of entities and goals. Each entity that is related to goals as both consumer and producer will produce a process component. After the definition of the process components, the interface components are defined based on the agents: each agent that implements a task will produce an interface component, and this task will be assigned to this component. Then, it is necessary to derive the dependencies between process components. This dependency defines whether a process component A consumes a data produced by process component B. Finally, the process components are associated to interface components, based on the goal model. An interface component is associated to a process component if a task of the producer goal or the consumer goal related to the process component is assigned to an external agent being communicated via the interface component. The completion of this process produces the structural view of the system's abstract architecture.

Given the abstract architecture, the final step of the proposal is the *Concrete Architecture Derivation*. For doing so, it is necessary an analysis of the architecture and choice of the architectural style that better tackles the system's needs. The selection is based on the evaluation of each alternative style, analyzing the impact of the choice within the softgoals. The selected style is then applied to the abstract architecture, producing a concrete architecture.

It is important to highlight that all the steps proposed by GOSA are manual and abstract, in the sense that there is no tool that implements it and the steps are not bound to any language.

7.2.3 Adaptation Goals for Adaptive Service-Oriented Architectures

Baresi and Pasquale (BARESI; PASQUALE, 2011) propose an adaptation mechanism to support the dynamism of adaptive service-oriented architectures in goal models. The proposal relies on extending the goal-oriented mechanisms to support dynamism at both design time and runtime.

The proposal adopts KAOS and RELAX (WHITTLE et al., 2009) for representing goal

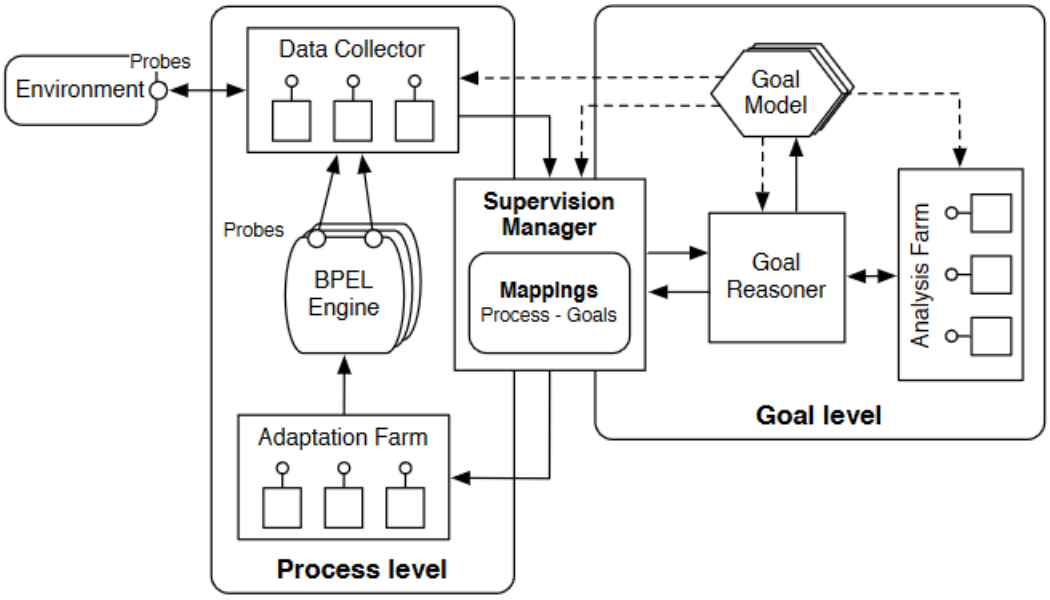


Figure 93: Runtime infrastructure

models in a complimentary way: RELAX is used to describe fuzzy goals, for instance, goals that can be partially satisfied.

Extending KAOS through adaptation capabilities, the proposal relies on the specification of adaptations to the goal model. For doing so, the adaptation capability is defined. An adaptation capability is the ability of the system to modify its goal model, impacting on both structure and operation of the system.

Each adaptation capability has its own trigger and set of conditions, similar to missions, and is operationalized by an action that can involve adding, removing or modifying goals or other adaptation goals, operation or entities. Furthermore, an action can also perform an operation, a goal, or substitute an agent.

Differently from traditional goals, missions are evaluated at runtime and can affect each other, which is similar to the effect of adaptation capabilities over goals. The proposal is very interesting to this thesis, since it proposes an infrastructure to runtime support in this similar context. The proposed infrastructure works at two levels: the process level and goal level, as illustrated by Fig. 93 [6].

The process level involves an Business Process Execution Language (BPEL) (OASIS, 2007) engine capable of executing the tasks of the system. This engine collects data and updates values for entities, detects events, and evaluates the satisfaction of goals. A data collector is responsible for gathering data, using probes to gather information from the

[6]Extracted from (BARESI; PASQUALE, 2011)

environment.

The goal level maintains a live goal model and updates it according to the information gathered by the engine, reconfiguring the system as needed. The goal level also evaluates the triggers and conditions for executing the adaptations. The relations between the processes and the goals are maintained by a supervisor, that can affect both levels. The proposal also uses the engine to realize service compositions, in order to satisfy a recently adapted goal model.

Self-adaptive service-oriented systems can provide many solutions for the specific case of SoS that uses service-oriented constituent systems. The Baresi and Pasquale proposal might contribute to the development of the simulation mechanism that is planned for this work. The simulation mechanism may be very similar to the infrastructure proposed, although it might need some additional information since the constituent systems can change depending only on the environment.

This proposal focuses on service-oriented architectures, which is one possible architectural style for SoS. The approach uses a live goal model at runtime. This model guides the reconfiguration process for the architecture. However, this solution focuses on runtime solutions and our focus is on the architectural process.

7.3 Discussion

As SoS is a recent concept (it first appeared in 1998 (MAIER, 1998)), thus it is not a surprise that there are many gaps in the proposals for this domain. Since the industry is showing some interest in the domain, many studies are being conducted in this context. However, in a sandy domain as such the ideas evolve slowly. The concept of mission was first modeled by a study of the group involved in this work (SILVA et al., 2016), therefore, it was expectable that no methodology, process or framework considered this concept within its definition.

Although some studies presented a notable contribution in the domain, of which COMPASS is worth highlighting, they rely on traditional requirements and techniques, lacking on specific support for dynamism, emergent behaviors and missions, that are essential concerns on the SoS domain.

The state-of-the-art shows a growing concern with verification and validation, and the studies tends to use some formalism to both support traceability and improve quality.

Most of the studies presented involves some level of formalism. Furthermore, simulation is also within the methodologies as the one we propose, as a support for the validation process.

Also, we detected a lack of tools to support the architectural modeling process. Some solutions present tools that partially support the process, but most of the approaches are essentially manual. We acknowledge the importance of CASE applications, therefore the development of such tools are a major work perspective in this context.

M2Arch differs from existing approaches for proposing a novel, tool-supported, mission-based method to produce software architectures for software-intensive systems-of-systems, that supports modeling, verification and validation whilst giving a special attention to emergent behavior.

8 Final Remarks

This study permeates among several domains of software engineering for systems-of-systems. We produced results in domains of: (i) mission modeling, (ii) architectural modeling, (iii) architectural verification, (iv) architectural validation, (v) modeling processes, (vi) architecture simulation and (vii) computer-aided software engineering.

Our main contribution is a pioneer methodology to produce software architectures for SoS, based on formally described mission models.We use many existing tools, languages and initiatives in the most various contexts. At the same time, we propose a process that is theoretically grounded, allowing then all involved tools and languages to be replaced with reduced effort.

M2Arch is a methodology that uses mission models as starting point for architectural modeling, using the language mKAOS (SILVA; BATISTA; OQUENDO, 2015; SILVA; BATISTA; CAVALCANTE, 2015) that was defined based on a goal-oriented language and a systematic review (SILVA et al., 2014) that identified how missions are defined in SoS context. The language was later enhanced, by adding a formalism coherent with the original one.

On the other hand, we produce architectures in SosADL (OQUENDO, 2016a), a pioneer ADL directed for SoS that is formally grounded in π-calculus (OQUENDO, 2016b). To establish a connection between the mission model and the architecture model, we identified a set of common concepts and developed a model-to-model transformation that generates a basic architectural structure.

We went further, defining a verification mechanism that uses Statistical Model Checking to automatically verify the constraints defined in the mission model. This same mechanism is also used to partially automatize a validation mechanism, automatically testing the achievement of formally described missions. The manual aspects of validation are also covered in M2Arch, with a simulation environment that allows the architect to foresee the actual behavior of the architecture.

Such wide study, however, is full of limitations. First of all, for proposing a pioneer methodology based on mission models, it was not possible to properly compare it to any existing study. Although we have plans to perform further studies to enhance M2Arch, incorporating positive aspects of other methodologies, it was not possible to do this yet due to time limitations. We performed a case study to evaluate the methodology, comparing the final result to the existing architecture of the system, as a result, we identified a small improvement in architectural quality.

The remainder of this chapter is structured as follows: Section 8.1 revisits our contributions, discussing the research questions and implementation. Section 8.2 presents some useful links, that can be consulted for additional information and details. Finally, Section 8.3 discusses our future works and evolution of M2Arch.

8.1 Revisiting the Contributions

8.1.1 Answering the Research Questions

We based this work on six research questions, presented in Section 1.2. We answered these questions as follows:

- **RQ1: What are the common concepts that permeate between the mission model's elements and the architectural model?**
  Some concepts permeate between both models. Specifically, *capabilities* are present in both mission model and architectural model. In mKAOS, they are explicit, represented as a first order element and divided into two kinds: *communicational* and *operational*. In SosADL, on the other hand, this concept is implicit and can be related to *interfaces*. A *operational capability* in SosADL can be defined through the set of connections of a given constituent system, forming a gate. *Gate* encompasses the inputs and output *connections* that defines an interface of a *constituent system* that implements a capability. Regarding *communicational capabilities*, they can be mapped to *mediators*, since they specify an interaction between two or more *constituent systems*. Based on this finding, we could define the M2Arch automatic mapping, that was implemented using ATL and allows automatic generation of partial architectural models.

- **RQ2: How can we relate mission model elements with architectural ele-**

ments? The concept of capability, that permeates between both architectural and mission model allowed us to draw an automatic mapping. Such automatic mapping promotes the traceability as it defines a relation between the elements of different models. Specifically, we can associate a *capability* in mKAOS to a *gate* or *duty* in SosADL.

- **RQ3: How to verify mission-related architectural properties in the SoS context?**
  Before verifying mission-related properties it is fundamental to **express** such properties. For doing so, we formalized mKAOS to introduce an extension of Linear Temporal Logic, allowing therefore the definition of formal constraints. Then, we adopted a strategy based on Statistical Model Checking and architectural simulation to allow the verification of such constraints. This solution handles the dynamism and behavioral uncertainty that are present in SoS architectural models.

- RQ4: **How to validate an architectural model within a mission model?**
  Based on the method we propose to verification, we defined an automatic validation for architectural models. This automatic verification is, in a broader perspective, a verification that checks the compliance of the architecture with some properties. However, in this case, the properties are formally described as **missions**. Hence, we can automatically validate an architecture within a mission model, detecting whether this architecture achieves the specified missions.

- RQ5: **How to validate an architecture produced through a mission-based process?**
  Validating an architecture is an essentially manual process, that consists in identifying whether an architecture meets stakeholders' needs. In case of SoS, this can be done through simulation. Based on the reports of a simulation process, the stakeholders are able to track, step-by-step, the execution of the architecture, hence identifying if the architecture meets their needs and the emergent behaviors are emerging as expected.

- RQ6: **Which kind of architectural validation can be done regarding emergent behaviors?**
  Validation of emergent behavior is a difficult and key activity on validation of architectures of SoS. We developed a method to automatically detect the occurrence of formally-described expected emergent behaviors, based on statistical model checking and simulation. Using this method, the stakeholders are able to identify whether

an architecture is emerging the expected behaviors and the frequency each behavior manifests.

8.1.2 Tool Implementations

M2Arch is an extensive methodology for producing software architectures for SoS. Due to its extension, it is fundamental to have a toolset that supports the application of the methodology. Therefore, we also implemented a set of tools that integrate existing tools into the so-called **M2Arch toolkit**.

Some features of M2Arch toolkit are worth highlight:

1. Textual and graphical description of **mKAOS models**
2. Textual and graphical description of **SosADL models**
3. **Automatic mapping**: mKAOS to SosADL
4. Automatic **verification of mission-based constraints** using PlasmaLab
5. Automatic detection of **formally described emergent behaviors**
6. Automatic calculation of **mission achievement rate**, based on architectural simulation
7. **Simulation of SosADL models**
8. Generation of detailed **simulation reports**

The SosADL simulator, the main contribution of M2Arch toolkit, was designed to be extensible, providing an **event manager** that can be extended or integrated on future tools for simulation.

mKAOS and SosADL tools are in constant evolution. However, since M2Arch toolkit was designed to operate over the existing tools, we expect the toolkit to continue to function with future versions of the overmentioned tools.

8.2 Relevant Links

Besides the contents of this document, additional information, source codes and models can be found on the following links:

1. `http://github.com/eduardoafs/mkaos`: The official GIT repository for mKAOS
2. `http://github.com/eduardoafs/m2arch`: A public GIT repository for M2Arch Toolkit
3. `http://eduardoafs.github.io/m2arch`: The official page of M2Arch

8.3 Future Work

M2Arch is a pioneer mission-based methodology for producing SoS architectures. Although it uses two specific languages for modeling, the whole methodology relies on the concepts that permeate between different constructs and elements. Therefore, we expect that the evolution of M2Arch also rely on these concepts, identifying additional concepts or alternative representations to allow evolution of all subsequent methods.

For replacing mKAOS for another mission description language, for instance, it is necessary to identify the representation of capabilities in this language, which must support detailing the interfaces. Then, it is necessary to adapt the formalism of the desired mission description language to be compatible with PlasmaLab. Implementing the automatic transformation to SosADL and a new module for producing PlasmaLab-compatible constraints should be enough for completely replacing mKAOS without losing cohesion with the rest of M2Arch.

Another important aspect that may be part of M2Arch evolution is the graphical animation of SosADL models during simulation. Since SosADL simulator was implemented as a layer-based architecture, it is possible to build additional layers to provide further information to the user. The animation can be implemented as an additional layer, using the *event manager* and Sirius animators [1].

A key future work, however, is the validation of the methodology within the industry. Initially, it was part of the planning for this work to perform controlled experiments to validate M2Arch. It was not possible due to time limitation and the lack of interaction with the specialized industry. In this context, it is also important to run a scalability test on the approach, to observe how it behaves when applied to large scale SoS.

Also, it is key to check expressiveness of DynBLTL in SoS context. Although the language was designed for dynamic systems, when it comes to SoS the new characteristics of this kind of system may required additional constructs, operations or functions.

[1] `https://github.com/SiriusLab/ModelDebugging`

As a long-term future work, each step of M2Arch can be refined. These steps can be detailed providing a set of guidelines and further instructions to allow stakeholders to be involved during all steps of architectural design. Also is important to give further attention to formal definitions, specially on missions, that can be automatically verified by M2Arch toolkit.

References

42010:2011, I. *ISO/IEC/IEEE Systems and software engineering – Architecture description*. Dec 2011. 1-46 p.

ALKHABBAS, F.; SPALAZZESE, R.; DAVIDSSON, P. Architecting emergent configurations in the internet of things. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017. p. 221–224.

ATL – A model transformation technology. Available at: <http:// https://eclipse.org/atl/>.

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. *Computer Networks*, v. 54, n. 15, p. 2787 – 2805, 2010. ISSN 1389-1286. Available at: <http://www.sciencedirect.com/science/article/pii/S1389128610001568>.

AVGERIOU JOHN GRUNDY, J. G. H. P. L. I. M. P. (Ed.). *Relating Software Requirements and Architectures*. Springer Berlin Heidelberg, 2011.

BARESI, L.; PASQUALE, L. Adaptation goals for adaptive service-oriented architectures. In: *Relating Software Requirements and Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 161–181. ISBN 978-3-642-21001-3.

BATISTA, T. Challenges for SoS architecture description. In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems (SESoS 2013)*. New York, NY, USA: ACM, 2013. p. 35–37. ISBN 978-1-4503-2048-1.

BOARDMAN, J.; SAUSER, B. System of systems – The meaning of *of*. In: *Proceedings of the 2006 IEEE/SMC International Conference on Systems of Systems Engineering*. USA: IEEE Computer Society, 2006.

BOEHM, B.; LANE, J. A. 21st century processes for acquiring 21st century software-intensive systems of systems. *The Journal of Defense Software Engineering*, v. 19, p. 4–9, 2006.

BRESCIANI, P. et al. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, v. 8, n. 3, p. 203–236, 2004. ISSN 1573-7454. Available at: <http://dx.doi.org/10.1023/B:AGNT.0000018806.20944.ef>.

CALINESCU, R.; KWIATKOWSKA, M. Software Engineering techniques for the development of systems of systems. In: CHOPPY, C.; SOKOLSKY, O. (Ed.). *Proceedings of the 15th Monterey Workshop Foundations of Computer Software: Future trends and techniques for development*. Germany: Springer-Verlag Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6028). p. 59–82. ISBN 978-3-642-12565-2.

CAVALCANTE, E. *A Formally Founded Framework for Dynamic Software Architectures*. Tese (Doutorado), 2016.

CHUNG, L. et al. Goal-oriented software architecting. In: *Relating Software Requirements and Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 91–109. ISBN 978-3-642-21001-3.

CLARKE JR., E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8.

COMBEMALE, B.; BARAIS, O.; WORTMANN, A. Language engineering with the gemoc studio. In: *IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017. p. 189–191.

COURETAS, J. M.; ZEIGLER, B. P.; PATEL, U. Automatic generation of system entity structure alternatives: Application to initial manufacturing facility design. *Trans. Soc. Comput. Simul. Int.*, Society for Computer Simulation International, San Diego, CA, USA, v. 16, n. 4, p. 173–185, dez. 1999. ISSN 0740-6797. Available at: <http://dl.acm.org/citation.cfm?id=340538.340554>.

COUTO, L. D.; FOSTER, S.; PAYNE, R. Towards verification of constituent systems through automated proof. In: *Workshop on Engineering Dependable Systems of Systems. ACM CoRR.* 2014.

DAMS, D. et al. Model checking using adaptive state and data abstraction. In: . 1994. p. 455–467.

DARDENNE, A.; LAMSWEERDE, A. van; FICKAS, S. Goal-directed requirements acquisition. *Science of Computer Programming*, v. 20, n. 1, p. 3 – 50, 1993. ISSN 0167-6423.

DARIMONT, R.; LAMSWEERDE, A. van. Formal refinement patterns for goal-driven requirements elaboration. In: *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 1996.

DASHOFY, E. M.; HOEK, A. V. d.; TAYLOR, R. N. A highly-extensible, xml-based architecture description language. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2001. (WICSA '01), p. 103–. ISBN 0-7695-1360-3.

DEGROSSI, L. C.; AMARAL, G. G. do; VASCONCELOS, E. S. M. de. Using wireless sensor networks in the sensor web for flood monitoring in brazil. In: *Proceedings of the 10th International ISCRAM Conference*. Baden-Baden, Germany: [s.n.], 2013.

DEMRI, S.; SANGNIER, A. When model-checking freeze ltl over counter machines becomes decidable. In: ONG, L. (Ed.). *Foundations of Software Science and Computational Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 176–190. ISBN 978-3-642-12032-9.

ECLIPSE IDE. Available at: <http://eclipse.org/>.

ECLIPSE Modeling Framework. Available at: <http://eclipse.org/modeling/emf/>.

EMERSON, E. A. Handbook of theoretical computer science (vol. b). In: LEEUWEN, J. van (Ed.). Cambridge, MA, USA: MIT Press, 1990. cap. Temporal and Modal Logic, p. 995–1072. ISBN 0-444-88074-7. Available at: <http://dl.acm.org/citation.cfm?id=114891.114907>.

ENOIU, E. P. et al. Vital: A verification tool for east-adl models using uppaal port. In: *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. 2012. p. 328–337.

FITZGERALD, J.; BRYANS, J.; PAYNE, R. A formal model-based approach to engineering systems-of-systems. In: CAMARINHA-MATOS, L. M.; XU, L.; AFSARMANESH, H. (Ed.). *Collaborative Networks in the Internet of Services: 13th IFIP WG 5.5 Working Conference on Virtual Enterprises, PRO-VE 2012, Bournemouth, UK, October 1-3, 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 53–62. ISBN 978-3-642-32775-9.

FITZGERALD, J.; LARSEN, P. G.; WOODCOCK, J. Foundations for model-based engineering of systems of systems. In: *Complex Systems Design & Management: Proceedings of the Fourth International Conference on Complex Systems Design & Management CSD&M 2013*. Cham: Springer International Publishing, 2014. p. 1–19. ISBN 978-3-319-02812-5.

GARLAN, D.; MONROE, R.; WILE, D. Acme: An architecture description interchange language. In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1997. (CASCON '97), p. 7–.

GARLAN, D.; SHAW, M. *An Introduction to Software Architecture*. Pittsburgh, PA, USA, 1994.

GIESECKE, S.; HASSELBRING, W.; RIEBISCH, M. Classifying architectural constraints as a basis for software quality assessment. *Advanced Engineering Informatics*, v. 21, n. 2, p. 169 – 179, 2007. ISSN 1474-0346. Ontology of Systems and Software Engineering; Techniques to Support Collaborative Engineering Environments. Available at: <http://www.sciencedirect.com/science/article/pii/S1474034606000681>.

GIUNCHIGLIA, F.; MYLOPOULOS, J.; PERINI, A. The tropos software development methodology: Processes, models and diagrams. In: *Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002 Bologna, Italy, July 15, 2002 Revised Papers and Invited Contributions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 162–173. ISBN 978-3-540-36540-2.

GOLDSTEIN, M.; SEGALL, I. Automatic and continuous software architecture validation. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 59–68. Available at: <http://dl.acm.org/citation.cfm?id=2819009.2819021>.

GONCALVES, M. et al. Towards a conceptual model for software-intensive system-of-systems. In: *Proceeding of the 2nd International Workshop on Software Engineering for Systems-of-Systems*. 2014.

GUESSI, M. et al. A systematic literature review on the description of software architectures for systems of systems. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015.

GUESSI, M.; OQUENDO, F.; NAKAGAWA, E. Y. Checking the architectural feasibility of systems-of-systems using formal descriptions. In: *2016 11th System of Systems Engineering Conference (SoSE)*. 2016. p. 1–6.

HALEY, C. B.; NUSEIBEH, B. Bridging requirements and architecture for systems of systems. In: *2008 International Symposium on Information Technology*. 2008. v. 4, p. 1–8. ISSN 2155-8973.

HOARE, C. A. R.; JIFENG, H. *Unifying Theories of Programming*. [S.l.]: Prentice Hall College Division, 1998.

HOLZMANN, G. J. The logic of bugs. In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2002. (SIGSOFT '02/FSE-10), p. 81–87. ISBN 1-58113-514-9. Available at: <http://doi.acm.org/10.1145/587051.587064>.

HUGHES, D. et al. A middleware platform to support river monitoring using wireless sensor networks. *Journal of the Brazilian Computer Society*, v. 17, n. 2, p. 85–102, Jun 2011. ISSN 1678-4804.

IEC61508-3. *Functional Safety*. 2010.

IEEE Standard for Software Verification and Validation. *IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998)*, p. 1–110, June 2005.

ISO/IEC9126. *Quality Attributes*. 1995.

ISSARNY, V.; BENNACEUR, A. Composing distributed systems: Overcoming the interoperability challenge. In: *Proceedings of the 11th International Symposium on Formal Methods for Components and Objects*. 2013.

JACKSON, M. *Problem Frames*. Addison Wesley, 2001.

KAMIDE, N. Bounded linear-time temporal logic: A proof-theoretic investigation. *Annals of Pure and Applied Logic*, v. 163, n. 4, p. 439 – 466, 2012. ISSN 0168-0072. Available at: <http://www.sciencedirect.com/science/article/pii/S0168007211001758>.

KUMAR, N. Software architecture validation methods, tools support and case studies. In: SHETTY, N. R.; PRASAD, N.; NALINI, N. (Ed.). *Emerging Research in Computing, Information, Communication and Applications*. New Delhi: Springer India, 2016. p. 335–345. ISBN 978-81-322-2553-9.

LAMSWEERDE, A. van. Goal-Oriented Requirements Engineering: A guided tour. In: *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*. Washington, DC, USA: IEEE Computer Society, 2001. p. 249–262.

LAMSWEERDE, A. van. From system goals to software architecture. In: *Formal Methods for Software Architectures: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003. Advanced Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 25–43. ISBN 978-3-540-39800-4.

LAMSWEERDE, A. van. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

LAMSWEERDE, A. van; LETIER, E. From object orientation to goal orientation: A paradigm shift for requirements engineering. In: WIRSING, M.; KNAPP, A.; BALSAMO, S. (Ed.). *Radical Innovations of Software and Systems Engineering in the Future*. Springer Berlin Heidelberg, 2004, (Lecture Notes in Computer Science, v. 2941). p. 325–340. ISBN 978-3-540-21179-2.

LEEM, C. S.; KIM, B. G. Taxonomy of ubiquitous computing service for city development. *Personal and Ubiquitous Computing*, v. 17, n. 7, p. 1475–1483, Oct 2013. ISSN 1617-4917.

LEGAY, A.; DELAHAYE, B.; BENSALEM, S. Statistical model checking: An overview. In: *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 122–135. ISBN 978-3-642-16612-9.

LEGAY, A.; SEDWARDS, S. On statistical model checking with plasma. In: *2014 Theoretical Aspects of Software Engineering Conference*. [S.l.: s.n.], 2014. p. 139–145.

LEGAY, A.; SEDWARDS, S.; TRAONOUEZ, L.-M. Plasma lab: A modular statistical model checking platform. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part I*. Cham: Springer International Publishing, 2016. p. 77–93. ISBN 978-3-319-47166-2.

LEITE, J.; OQUENDO, F.; BATISTA, T. Sysadl: A sysml profile for software architecture description. In: DRIRA, K. (Ed.). *Software Architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 106–113. ISBN 978-3-642-39031-9.

LICHTNER, K.; ALENCAR, P.; COWAN, D. A framework for software architecture verification. In: *Proceedings 2000 Australian Software Engineering Conference*. 2000. p. 149–157.

LUCKHAM, D. C. *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events*. Stanford, CA, USA, 1996.

MAGEE, J.; KRAMER, J.; GIANNAKOPOULOU, D. Behaviour analysis of software architectures. In: *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1) 22–24 February 1999, San Antonio, Texas, USA*. Boston, MA: Springer US, 1999. p. 35–49. ISBN 978-0-387-35563-4.

MAIER, M. Architecting principles for systems-of-systems. *Systems Engineering*, John Wiley & Sons, Inc., v. 1, n. 4, p. 267–284, 1998. ISSN 1520-6858.

MANNA, Z.; PNUELI, A. *The Temporal Logic of Reactive and Concurrent Systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992. ISBN 0-387-97664-7.

MICHAEL, J. B.; RIEHLE, R.; SHING, M. T. The verification and validation of software architecture for systems of systems. In: *System of Systems Engineering, 2009. SoSE 2009. IEEE International Conference on*. [S.l.: s.n.], 2009. p. 1–6.

MIKIC-RAKIC, M.; MEDVIDOVIC, N. Architecture-level support for software component deployment in resource constrained environments. In: *Proceedings of the IFIP/ACM Working Conference on Component Deployment*. London, UK, UK: Springer-Verlag, 2002. (CD '02), p. 31–50. ISBN 3-540-43847-5. Available at: <http://dl.acm.org/citation.cfm?id=647479.727942>.

MS4 Systems. Available at: <http://www.ms4systems.com/pages/main.php>.

NAKAGAWA M. GONCALVES, M. G. L. B. R. O. E. Y.; OQUENDO, F. The state of the art and future perpsectives in systems-of-systems software architectures. In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. 2013.

NAQVI, S. A. et al. Cross-document dependency analysis for system-of-system integration. In: CHOPPY, C.; SOKOLSKY, O. (Ed.). *15th Monterey Workshop Foundations of Computer Software, Future Trends and Techniques for Development*. Germany: Springer-Verlag Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6028). p. 201–226. ISBN 978-3-642-12565-2.

NETO, V. Validating emergent behaviors in systems-of-systems through model transformations. In: *Proceedings of the ACM PhD Student Research Competition at MODELS 2016 co-located with the 19th International Conference on Model Driven Engineering Languages and Systems*. St. Malo, France: [s.n.], 2016. (MODELS 2016).

NETO, V. V. G. et al. Asas: An approach to support simulation of smart systems. In: *Turning Smart: Challenges and Experiences in Smart Application Development*. 2018.

NETO, V. V. G. et al. Stimuli-sos: a model-based approach to derive stimuli generators for simulations of systems-of-systems software architectures. *Journal of the Brazilian Computer Society*, v. 23, n. 1, p. 13, Oct 2017. ISSN 1678-4804.

NUSEIBEH, B. Weaving together requirements and architectures. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 34, n. 3, p. 115–117, mar. 2001. ISSN 0018-9162. Available at: <http://dx.doi.org/10.1109/2.910904>.

OASIS. *Web services business process execution language*. 2007. Available at: <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.

OQUENDO, F. π-adl: An architecture description language based on the higher-order typed-calculus for specifying dynamic and mobile software architectures. In: *ACM SIGSOFT Software Engineering Notes*. [S.l.: s.n.], 2004.

OQUENDO, F. Formally describing the software architecture of systemsof-systems with sosadl. In: *Proceedings of the 11th IEEE International Conference on System-of-Systems Engineering*. 2016.

OQUENDO, F. π-calculus for sos: A foundation for formally describing software-intensive systems-of-systems. In: *Proceedings of the 11th IEEE International Conference on System-of-Systems Engineering*. [S.l.: s.n.], 2016.

OQUENDO, F. On the emergent behavior oxymoron of system-of-systems architecture description. In: *2018 13th Annual Conference on System of Systems Engineering (SoSE)*. 2018. p. 417–424.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 17, n. 4, p. 40–52, out. 1992. ISSN 0163-5948.

PNUELI, A. The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977. p. 46–57. ISSN 0272-5428.

QUILBEUF, J. et al. A Logic for the Statistical Model Checking of Dynamic Software Architectures. In: *ISoLA*. Corfou, Greece: Springer, 2016. (Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques, v. 9952), p. 806 – 820.

QVT Operational. Available at: <https://projects.eclipse.org/projects/modeling.mmt.qvt-oml>.

SENDALL, S.; KOZACZYNSKI, W. Model transformation: The heart and soul of model-driven software development. In: *IEEE Software*. [S.l.: s.n.], 2003.

SILVA, E.; BATISTA, T.; CAVALCANTE, E. A mission-oriented tool for systemof-systems modeling. In: *Proceedings of the 3rd IEEE/ACM International Workshop on Software Engineering for Systems-of-System (SESoS'15)*. 2015.

SILVA, E.; BATISTA, T.; OQUENDO, F. A mission-oriented approach for designing system-of-systems. In: *10th System of Systems Engineering Conference (SoSE'15)*. 2015. p. 346–351.

SILVA, E.; CAVALCANTE, E.; BATISTA, T. Refining missions to architectures in software-intensive systems-of-systems. In: *Proceedings of the Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems*. Piscataway, NJ, USA: IEEE Press, 2017. (JSOS '17), p. 2–8. ISBN 978-1-5386-2799-0.

SILVA, E. et al. On the characterization of missions of systems-of-systems. In: *Proceeding of the 2nd International Workshop on Software Engineering for Systems-of-Systems*. 2014.

SILVA, E. et al. Bridging missions and architecture in software-intensive systems-of-systems. In: *21st International Conference on Engineering of Complex Computer Systems (ICECCS'16)*. 2016. p. 201–206.

TSAI, J. J.; XU, K. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, v. 10, n. 1, p. 207–223, Nov 2000. ISSN 1573-7489. Available at: <https://doi.org/10.1023/A:1018960305057>.

VASARHELYI, G. et al. Optimized flocking of autonomous drones in confined environments. *Science Robotics*, v. 3, p. eaat3536, 07 2018.

VöLTER T. STAHL, J. B. A. H. M.; HELSEN., S. *Model-Driven Software Development: Technology, engineering, management*. John Wiley & Sons, Inc, 2006.

WHITTLE, J. et al. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In: *2009 17th IEEE International Requirements Engineering Conference*. 2009. p. 79–88. ISSN 1090-705X.

YU, E. S. Social modeling and i*. In: *Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 99–121. ISBN 978-3-642-02463-4.

ZHANG, J. et al. Model checking software architecture design. In: *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*. 2012. p. 193–200. ISSN 1530-2059.

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. ISSN 0164-1212. Available at: <http://www.sciencedirect.com/science/article/pii/S0164121209003070>.

APPENDIX A – Publications

Our publications were achieved along the duration of the PhD, sharing our findings with the community. Table 13 summarizes the publications. Fig. 94 relates these publications with the chapters of this thesis. As shown by Fig. 94, all contributions of this work are grouped in Chapters 3, 4 and 5.

Id	Title	Authors	Mean
ICECSS'16	*Bridging Missions and Architecture in Software-Intensive Systems-of-Systems*	**Eduardo Silva**, Everton Cavalcante, Thais Batista, Flavio Oquendo	ICECSS'16
ECSA'17	*Taming Missions and Architecture in Software Intensive Systems-of-Systems*	**Eduardo Silva**	ECSA'17 Doctoral Symposium
SESoS'17	*Refining Missions to Architectures in Software-Intensive Systems-of-Systems*	**Eduardo Silva**, Everton Cavalcante, Thais Batista	SESoS'17
SAC'18	*Formal Modeling Systems-of-Systems Missions with mKAOS*	**Eduardo Silva**, Thais Batista	ACM SAC'18
SCP'18	*Expressing and Checking Mission-Related Properties on Systems-of-systems Design*	**Eduardo Silva**, Thais Batista, Flavio Oquendo	Science of Computer Programming, *to appear*
-	*Simulating SosADL Concrete Architectures*	**Eduardo Silva**, Thais Batista, Flavio Oquendo	Under development

Table 13: Publications derived from this work

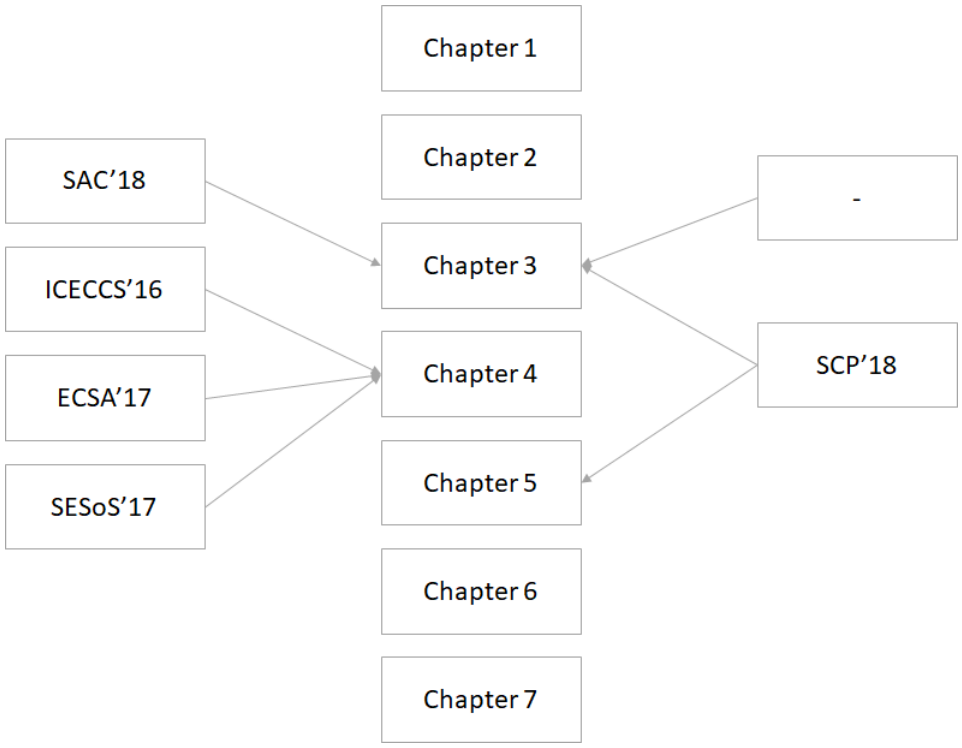


Figure 94: Relation between Publications and Chapters

APPENDIX B – ATL Rules for mKAOS to SosADL transformation

Fully available at: http://www.github.com/eduardoafs/mkaos

```
-- @path MKAOS=/Kaos/model/mkaos.ecore
-- @path SOSADL=/org.archware.sosadl.SosADL/model/generated/SosADL.ecore


module mkaos2sosadl;
create OUT: SOSADL from IN: MKAOS;

--  quick way to identify all the outputs of a capability
helper context MKAOS!OperationalCapability def: output(): Sequence(MKAOS
    !Object) =
self.output.union(self.produces);

helper context MKAOS!CommunicationalCapability def: output(): Sequence(
    MKAOS!Object) =
self.output.union(self.produces);

-- quick way to identify all the inputs from a capability
helper context MKAOS!OperationalCapability def: input(): Sequence(MKAOS!
    Object) =
self.refImmediateComposite().oclAsType(MKAOS!mKAOS).consistsOf() ->
    select(p | p.
oclIsKindOf(MKAOS!Entity) and p.oclAsType(MKAOS!Entity).inputs.
contains(self));

helper context MKAOS!CommunicationalCapability def: input(): Sequence(
    MKAOS!Object) =
self.refImmediateComposite().oclAsType(MKAOS!mKAOS).consistsOf() ->
    select(p | p.
```

```
oclIsKindOf(MKAOS!Entity) and p.oclAsType(MKAOS!Entity).inputs.
contains(self));

-- all inputs or outputs of a capability
helper context MKAOS!CommunicationalCapability def : interface() :
   Sequence(MKAOS!Object) =
self.output()->union(self.input);

-- functions for isolate constituent systems, entities and capabilities
helper context MKAOS!mKAOS def: entities(): Sequence(MKAOS!Entity) =
self.consistsOf -> select(p | p.oclIsTypeOf(MKAOS!Entity));

helper context MKAOS!mKAOS def: constituent(): Sequence(MKAOS!
   ConstituentSystem) =
self.consistsOf -> select(p | p.oclIsTypeOf(MKAOS!ConstituentSystem));

helper context MKAOS!mKAOS def: capabilities(): Sequence(MKAOS!
   CommunicationalCapability)
=
self.consistsOf -> select(p | p.oclIsTypeOf(MKAOS!
   CommunicationalCapability));

-- production rules for empty behaviors
helper def: emptyProtocol(): SosADL!ProtocolDecl =
let prot : SosADL!ProtocolDecl = SosADL!ProtocolDecl.newInstance(name =
   'behavior',
behavior = SosADL!Protocol.newInstance(statements = SosADL!
   RepeatProtocol.
newInstance(repeated = SosADL!AnyAction.newInstance())))
in prot;

helper def: emptyBehavior(): SosADL!BehaviorDecl =
let behavior : SosADL!BehaviorDecl = SosADL!BehaviorDecl.newInstance(
   name =
'behavior', body = SosADL!Behavior.newInstance(statements =
SosADL!RepeatProtocol.newInstance(repeated = SosADL!Unobservable.
newInstance())))
in behavior;

helper def: emptyAssertion(): SosADL!AssertionDecl =
let assertion : SosADL!AssertionDecl = SosADL!AssertionDecl.newInstance(
   name =
'behavior', body = self.emptyProtocol()) in assertion;
```

```
helper def: emptyFunction(): SosADL!FunctionDecl =
let fun : SosADL!FunctionDecl = SosADL!FunctionDecl.newInstance(name = '
    f1', type =
self, return = SosADL!Any.newInstance()) in fun;

rule ProduceSos {
from
missions: MKAOS!mKAOS
to
eblock: SOSADL!EntityBlock (
datatypes <- missions.entities(),
systems <- missions.constituent(),
mediators <- missions.capabilities(),
architectures <- arch
),
sos: SOSADL!SoS (
name <- 'GeneratedSoS',
decls <- eblock
),
arch: SOSADL!ArchitectureDecl (
behavior <- archb
),
archb: SOSADL!ArchBehaviorDecl (
constituents <- missions.constituent(),
bindings <- let bin : SosADL!Binding = self.buildBindings() in bin
)
}

rule DataTypesFromEntities {
from
entity: MKAOS!Entity
to
dtype: SOSADL!DataTypeDecl (
name <- entity.name
)
}

rule ProduceConstituentSystem {
from
mkaos_cs: MKAOS!ConstituentSystem
to
sos_cs: SOSADL!SystemDecl (
```

```
name <- mkaos_cs.name,
-- gates will be produced from operational capabilities
gates <- mkaos_cs.capableOf
)
}

rule ProduceGateFromCapability {
from
mkaos_operationalCapability: MKAOS!OperationalCapability
to
output_gate: SOSADL!GateDecl (
name <- mkaos_operationalCapability.name,
protocols <-
let prot : SosADL!ProtocolDecl = self.emptyProtocol() in prot,
)
}

rule ProduceInputConnectionFromEntity {
from
mkaos_entity : MKAOS!Entity
to
sos_connection : SOSADL!Connection (
valueType <- mkaos_entity,
mode <- 'ModeTypeIn',
name <- 'i0'
)
}

rule ProduceMediator {
from
mkaos_cs: MKAOS!CommunicationalCapability
to
sos_cs: SOSADL!MediatorDecl (
name <- mkaos_cs.name,
-- gates will be produced from operational capabilities
duties <- mkaos_cs.interface()
)
}

rule ProduceDutyFromCapability {
from
mkaos_com: MKAOS!CommunicationalCapability
to
```

```
output_gate: SOSADL!GateDecl (
name <- mkaos_com.name,
assume <-
let g : SosADL!Assertion = self.emptyAssertion() in g,
garantees <-
let g : SosADL!Assertion = self.emptyAssertion() in g
)
}
```

APPENDIX C – mKAOS Grammar

Also available at: http://www.github.com/eduardoafs/mkaos

```
// Made using Xtext
grammar mkaos.Language with org.eclipse.xtext.common.Terminals

import "platform:/resource/Kaos/model/mkaos.ecore"
import "platform:/resource/Kaos/model/kaos.ecore" as KAOSModel
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

mKAOS returns mKAOS:
'Model' name=EString
(linkedBy+=Link
| consistsOf+=Nodes)*
;

Link returns KAOSModel::Link:
AssignmentLink | ConflictLink | ObstructionLink | OutputLink | InputLink
    | Refinement_Impl | AndRefinement | OrRefinement |
   OperacionalizationLink | ResolutionLink | ResponsabilityLink;

Nodes returns KAOSModel::Nodes:
EmergentBehavior | Mission | Operation | OperationNode_Impl | Event |
   Entity | Associations | SoftwareAgent | EnvironmentAgent | Obstacle |
    Goal_Impl | Expectation | DomainProperty_Impl | Requirement |
   DomainHypothesis | DomainInvariant;

EmergentBehavior:
'EmergentBehavior' name=EString '{'
(('informalDef' '=' informal=EString)?
& ('formalDef' '=' formal=expr)?
& 'emergesFrom' emerge+=EmergeLink (',' emerge+=EmergeLink)*)
'}'
```

```
;

EmergeLink returns EmergeLink:
capability=[CommunicationalCapability|ID] '[' cardinality=EString ']'
;

Agent returns KAOSModel::Agent:
SoftwareAgent | EnvironmentAgent;

Mission returns Mission:
'Mission' name=EString '{'
(links+=MissionLink (',' links+=MissionLink)*)?
& ('resolves' resolve+=[KAOSModel::Obstacle|EString]
| 'conflicts' conflicts+=[KAOSModel::Goal|EString]
| 'concerns' concerns+=[KAOSModel::Object|EString])*
& ('assigned' 'to' assignedTo=[ConstituentSystem|EString])?
& ('priority' '=' priority=INT
& 'informalDef' '=' description=STRING
& 'trigger' '=' trigger=expr
& ('formalDef' '=' rule=expr)?)
(refinement=MissionRefinement)?
'}'
;

RefinableNode returns KAOSModel::RefinableNode:
Mission | Obstacle | Goal_Impl | Expectation | DomainProperty_Impl |
    Requirement | DomainHypothesis | DomainInvariant;

MissionLink returns MissionLink:
DisruptLink | SupportLink | BlockLink
;

DisruptLink returns DisruptLink:
'disrupt' target=[Mission|EString]
;

SupportLink returns SuportLink:
'support' target=[Mission|EString]
;

BlockLink returns BlockLink:
'block' target=[Mission|EString]
;
```

```
MissionRefinement returns MissionRefinement:
'refinement' '[' (kind=MissionRefinementKind | custom=expr ) ']'
'{'
submissions+=Mission*
'}'
;

enum MissionRefinementKind:
all='all' | atLeastOne='atLeastOne' | alternative='alternative' | custom
    ='custom'
;

Refinement returns KAOSModel::Refinement:
Refinement_Impl | AndRefinement | OrRefinement | MissionRefinement;

Goal returns KAOSModel::Goal:
Goal_Impl | Expectation | DomainProperty_Impl | Requirement |
    DomainHypothesis | DomainInvariant;

Object returns KAOSModel::Object:
Entity | Associations | SoftwareAgent | EnvironmentAgent;


EString returns ecore::EString:
STRING | ID;

AssignmentLink returns KAOSModel::AssignmentLink:
'assignment' assignsGoalTo+=[KAOSModel::Agent|EString] (',',
    assignsGoalTo+=[KAOSModel::Agent|EString])*;
//name=EString
//'{'
//  'assignsGoalTo' '(' assignsGoalTo+=[KAOSModel::Agent|EString] ( ","
    assignsGoalTo+=[KAOSModel::Agent|EString])* ')'
//'}';

ConflictLink returns KAOSModel::ConflictLink:
{KAOSModel::ConflictLink}
'ConflictLink'
name=EString;

ObstructionLink returns KAOSModel::ObstructionLink:
'ObstructionLink'
```

```
name=EString
'{'
'relateKGoalTo' '(' relateKGoalTo+=[KAOSModel::Obstacle|EString] ( ","
    relateKGoalTo+=[KAOSModel::Obstacle|EString])* ')'
'}';

OutputLink returns KAOSModel::OutputLink:
{KAOSModel::OutputLink}
'OutputLink'
name=EString;

InputLink returns KAOSModel::InputLink:
'InputLink'
name=EString
'{'
'objectInputOn' '(' objectInputOn+=[KAOSModel::Operation|EString] ( ","
    objectInputOn+=[KAOSModel::Operation|EString])* ')'
'}';

Refinement_Impl returns KAOSModel::Refinement:
'refinement'
//name=EString
'{'
'refines' refines=[KAOSModel::RefinableNode|EString]
'}';

AndRefinement returns KAOSModel::AndRefinement:
'AndRefinement'
name=EString
'{'
'refines' refines=[KAOSModel::RefinableNode|EString]
'}';

OrRefinement returns KAOSModel::OrRefinement:
'OrRefinement'
name=EString
'{'
'refines' refines=[KAOSModel::RefinableNode|EString]
'}';

OperacionalizationLink returns KAOSModel::OperacionalizationLink:
'OperacionalizationLink'
name=EString
```

```
'{'
'relateOperationTo' '(' relateOperationTo+=[KAOSModel::Requirement|
    EString] ( "," relateOperationTo+=[KAOSModel::Requirement|EString])*
    ')'
'}';

ResolutionLink returns KAOSModel::ResolutionLink:
'ResolutionLink'
name=EString
'{'
'assignObstacleTo' '(' assignObstacleTo+=[KAOSModel::Requirement|EString
    ] ( "," assignObstacleTo+=[KAOSModel::Requirement|EString])* ')'
'}';

ResponsabilityLink returns KAOSModel::ResponsabilityLink:
'ResponsabilityLink'
name=EString
'{'
'assignAgentTo' '(' assignAgentTo+=[KAOSModel::Requirement|EString] ( ",
    " assignAgentTo+=[KAOSModel::Requirement|EString])* ')'
'}';

Operation returns KAOSModel::Operation:
'Operation' name=EString
'{'
'produces' '(' produces+=[KAOSModel::Event|EString] ( "," produces+=[
    KAOSModel::Event|EString])* ')'
'output' '(' output+=[KAOSModel::Entity|EString] ( "," output+=[
    KAOSModel::Entity|EString])* ')'
'operationalize' '(' operationalize+=[KAOSModel::Requirement|EString] (
    "," operationalize+=[KAOSModel::Requirement|EString])* ')'
'}';

SoftwareAgent returns KAOSModel::SoftwareAgent:
'SoftwareAgent'
name=EString
'{'
'performs' '(' performs+=[KAOSModel::Operation|EString] ( "," performs
    +=[KAOSModel::Operation|EString])* ')'
('composition' '(' composition+=[KAOSModel::Agent|EString] ( ","
    composition+=[KAOSModel::Agent|EString])* ')' )?
'responsibleFor' '(' responsibleFor+=[KAOSModel::Requirement|EString] (
    "," responsibleFor+=[KAOSModel::Requirement|EString])* ')'
```

```
'}';

EnvironmentAgent returns KAOSModel::EnvironmentAgent:
'EnvironmentAgent'
name=EString
'{'
('performs' performs+=[KAOSModel::Operation|EString] ( "," performs+=[
    KAOSModel::Operation|EString])*)?
('composition' composition+=[KAOSModel::Agent|EString] ( "," composition
    +=[KAOSModel::Agent|EString])*)?
'}';

Event returns KAOSModel::Event:
{KAOSModel::Event}
'Event' name=EString;

Entity returns KAOSModel::Entity:
'Entity' name=EString
('{'
'composition' '=' composition+=[KAOSModel::Entity|EString] ( ","
    composition+=[KAOSModel::Entity|EString])*
'}')?;

Requirement returns KAOSModel::Requirement:
'Requirement'
name=EString
'{'
'refinedBy' '(' refinedBy+=[KAOSModel::Refinement|EString] ( ","
    refinedBy+=[KAOSModel::Refinement|EString])* ')'
'resolve' '(' resolve+=[KAOSModel::Obstacle|EString] ( "," resolve+=[
    KAOSModel::Obstacle|EString])* ')'
'conflicts' '(' conflicts+=[KAOSModel::Goal|EString] ( "," conflicts+=[
    KAOSModel::Goal|EString])* ')'
'concerns' '(' concerns+=[KAOSModel::Object|EString] ( "," concerns+=[
    KAOSModel::Object|EString])* ')'
'}';

Obstacle returns KAOSModel::Obstacle:
'Obstacle'
name=EString
'{'
'refinedBy' '(' refinedBy+=[KAOSModel::Refinement|EString] ( ","
    refinedBy+=[KAOSModel::Refinement|EString])* ')'
```

```
'obstruct' '(' obstruct+=[KAOSModel::Goal|EString] ( "," obstruct+=[
    KAOSModel::Goal|EString])* ')'
'}';

Goal_Impl returns KAOSModel::Goal:
'Goal'
name=EString
'{'
'refinedBy' '(' refinedBy+=[KAOSModel::Refinement|EString] ( ","
    refinedBy+=[KAOSModel::Refinement|EString])* ')'
'resolve' '(' resolve+=[KAOSModel::Obstacle|EString] ( "," resolve+=[
    KAOSModel::Obstacle|EString])* ')'
'conflicts' '(' conflicts+=[KAOSModel::Goal|EString] ( "," conflicts+=[
    KAOSModel::Goal|EString])* ')'
'concerns' '(' concerns+=[KAOSModel::Object|EString] ( "," concerns+=[
    KAOSModel::Object|EString])* ')'
'}';

Expectation returns KAOSModel::Expectation:
'Expectation'
name=EString
'{'
'refinedBy' '(' refinedBy+=[KAOSModel::Refinement|EString] ( ","
    refinedBy+=[KAOSModel::Refinement|EString])* ')'
'resolve' '(' resolve+=[KAOSModel::Obstacle|EString] ( "," resolve+=[
    KAOSModel::Obstacle|EString])* ')'
'conflicts' '(' conflicts+=[KAOSModel::Goal|EString] ( "," conflicts+=[
    KAOSModel::Goal|EString])* ')'
'concerns' '(' concerns+=[KAOSModel::Object|EString] ( "," concerns+=[
    KAOSModel::Object|EString])* ')'
'assignedTo' '(' assignedTo+=[KAOSModel::EnvironmentAgent|EString] ( ","
     assignedTo+=[KAOSModel::EnvironmentAgent|EString])* ')'
'}';

DomainProperty_Impl returns KAOSModel::DomainProperty:
'DomainProperty'
name=EString
'{'
'refinedBy' '(' refinedBy+=[KAOSModel::Refinement|EString] ( ","
    refinedBy+=[KAOSModel::Refinement|EString])* ')'
'resolve' '(' resolve+=[KAOSModel::Obstacle|EString] ( "," resolve+=[
    KAOSModel::Obstacle|EString])* ')'
'conflicts' '(' conflicts+=[KAOSModel::Goal|EString] ( "," conflicts+=[
```

```
    KAOSModel::Goal|EString])* ')'
'concerns' '(' concerns+=[KAOSModel::Object|EString] ( "," concerns+=[
    KAOSModel::Object|EString])* ')'
'usedIn' '(' usedIn+=[KAOSModel::Refinement|EString] ( "," usedIn+=[
    KAOSModel::Refinement|EString])* ')'
'}';

DomainHypothesis returns KAOSModel::DomainHypothesis:
'DomainHypothesis'
name=EString
'{'
'refinedBy' '(' refinedBy+=[KAOSModel::Refinement|EString] ( ","
    refinedBy+=[KAOSModel::Refinement|EString])* ')'
'resolve' '(' resolve+=[KAOSModel::Obstacle|EString] ( "," resolve+=[
    KAOSModel::Obstacle|EString])* ')'
'conflicts' '(' conflicts+=[KAOSModel::Goal|EString] ( "," conflicts+=[
    KAOSModel::Goal|EString])* ')'
'concerns' '(' concerns+=[KAOSModel::Object|EString] ( "," concerns+=[
    KAOSModel::Object|EString])* ')'
'usedIn' '(' usedIn+=[KAOSModel::Refinement|EString] ( "," usedIn+=[
    KAOSModel::Refinement|EString])* ')'
'}';

DomainInvariant returns KAOSModel::DomainInvariant:
'DomainInvariant'
name=EString
'{'
'refinedBy' '(' refinedBy+=[KAOSModel::Refinement|EString] ( ","
    refinedBy+=[KAOSModel::Refinement|EString])* ')'
'resolve' '(' resolve+=[KAOSModel::Obstacle|EString] ( "," resolve+=[
    KAOSModel::Obstacle|EString])* ')'
'conflicts' '(' conflicts+=[KAOSModel::Goal|EString] ( "," conflicts+=[
    KAOSModel::Goal|EString])* ')'
'concerns' '(' concerns+=[KAOSModel::Object|EString] ( "," concerns+=[
    KAOSModel::Object|EString])* ')'
'usedIn' '(' usedIn+=[KAOSModel::Refinement|EString] ( "," usedIn+=[
    KAOSModel::Refinement|EString])* ')'
'}';

Associations returns KAOSModel::Associations:
{KAOSModel::Associations}
'Associations'
name=EString;
```

```
OperationNode_Impl returns KAOSModel::OperationNode:
{KAOSModel::OperationNode}
'OperationNode'
name=EString;

ConstituentSystem returns ConstituentSystem:
'System' name=EString
'capableOf' capableOf+=[OperationalCapability|EString] (',' capableOf+=[
    OperationalCapability|EString])*
;

OperationalCapability returns OperationalCapability:
'OperationalCapability' name=EString '{'
'in' input+=[KAOSModel::Entity|EString]
'out' output+=[KAOSModel::Entity|EString]
('description' '=' desc=STRING)?
'}'
;

CommunicationalCapability returns CommunicationalCapability:
'CommunicationalCapability' name=EString '{'
'in' input+=[KAOSModel::Entity|EString]
'out' output+=[KAOSModel::Entity|EString]
('description' '=' desc=STRING)?
'}'
;


// All DynBLTL constructs, we don't need to store properly, just sintax
    checking

expr returns DynBLTL: // returns [Expr val]:
q=RuleQuantifier val=ID COL c=function t=temporal    //{ val = new
    QuantExpr($q.q,new Var($ID.text),$c.val,$e.val); }
| temporal                                          //{ val = $temporal.val ;
    }
;

enum RuleQuantifier: // returns [UnOp q]:
EXISTS='exists'  //{ q = UnOp.Exists ; }
| FORALL='forall'  //{ q = UnOp.Forall ; }
| COUNT='count'  //{ q = UnOp.Count ; }
```

```
;

temporal returns RuleTemporal:// returns [Expr val]:
val1=implication                    //{val = $implication.val ; }
( o=RuleTempBinOp  b=bound e=expr   //{ val =  new TemporalBinOpExpr(val
    ,$tempbinop.o,$expr.val,$bound.b); }
)?
| o1=RuleTempUnOp  b=bound e=expr     //{ val = new TemporalUnOpExpr(
    $tempunop.o,$expr.val,$bound.b); }
;


bound returns RuleBound:// ; returns [Bound b]// @init{ Expr boundVal =
    new UndefValue();}:
( integerlit=INT                   //{ boundVal = new IntValue(
    $integerlit.val) ;  }
| floatlit=FLOAT                    //{ boundVal = new FloatValue(
    $floatlit.val) ;  }
| LP e=expr RP                    //{ boundVal = $expr.val; }
) ( STEPS                        //{ b = new Bound(boundVal,false); }
| T_UNITS                        //{ b = new Bound(boundVal,true) ; }
)
;


enum RuleTempBinOp: // returns [BinOp o]:
UNTIL='until'              //{ o = BinOp.Until ; }
| WEAK='weak until'              //{ o = BinOp.Weak ; } // FIXME remove _
;


enum RuleTempUnOp: // returns [UnOp o]:
FATALLY='eventually before'     //{ o = UnOp.Fatally ; }
| GLOBALLY='always during'   //{ o = UnOp.Globally ; }
| NEXT='in'        //{ o = UnOp.Next ; }
;

implication returns RuleImplication: // returns [Expr val]       @init{
    UnOp undefOp = null; }
(undefop                                 //{ undefOp = $undefop.val ; }
)? l=disjunction                         //{ val = $l.val; }
(IMP r+=disjunction         //{ val = new BinOpExpr(val, BinOp.Imp, $r.
    val);  }
```

```
)*      //{ if(undefOp != null) { val = new UnOpExpr(undefOp,val) ; } }
;

undefop: // returns[UnOp val]:
ISTRUE                  //{ val = UnOp.IsTrue; }
| ISNFLS                 //{ val = UnOp.IsNotFalse ; }
;

disjunction returns RuleDisjunction: // returns [Expr val]
l=conjunction                          // { val = $l.val; }
(OR r+=conjunction        // { val = new BinOpExpr(val, BinOp.Or, $r.val
    ); }
)*
;

conjunction returns RuleConjunction: // returns [Expr val]:
l=equality                             //{ val = $l.val; }
( AND r+=equality             //{ val = new BinOpExpr(val, BinOp.And, $r
    .val); }
)*
;

equality returns RuleEquality: //returns [Expr val]          @init{
    boolean neg = false; }
(neg?=NOT                                  //{ neg = true ; }
)? l=relExp                          //{ val = $l.val; }
(eop r=relExp           //{ val = new BinOpExpr($l.val, $eop.val, $r.val
    ); }
)?        //{ if(neg) {val = new UnOpExpr(UnOp.Not, val); } }
;

eop: //returns [BinOp val]
EQ                                     //{ val = BinOp.Eq; }
| NEQ                                    //{ val = BinOp.Neq; }
;

relExp returns RuleRelExp: // returns [Expr val]:
l=numExp                       //{ val = $l.val; }
(rop r=numExp       //{ val = new BinOpExpr(val, $rop.val, $r.val);  }
)?
;

rop //returns [BinOp val]
```

```
: GT                          //{ val = BinOp.Gt; }
| LT                          //{ val = BinOp.Lt; }
| GE                          //{ val = BinOp.Ge; }
| LE                          //{ val = BinOp.Le; }
;

numExp returns RuleNumExp: //returns [Expr val]:
l=term                          //{ val = $l.val; }
(addop r+=term          //{ val = new BinOpExpr(val, $addop.val, $r.val);
     }
)*
;

addop: //returns [BinOp val]:
ADD                            //{ val = BinOp.Add;  }
| MIN                           //{ val = BinOp.Min;  }
;

term returns RuleTerm: //returns [Expr val]:
l=factor                        //{ val = $l.val; }
(mulop r+=factor      //{ val = new BinOpExpr(val, $mulop.val, $r.val); }
)*
;

mulop: // returns [BinOp val]:
MUL                         //{ val = BinOp.Mul; }
| DIV                         //{ val = BinOp.Div; }
;

factor returns RuleFactor: // returns [Expr val]  @init{ boolean neg =
   false; }
MIN?
( vallit=literal               // { val = $literal.val ;}
| valvar=var                    //{ val = $var.val     ;}
| valfun=function               //{ val = $function.val;}
| LP par=expr RP          //{ val = $par.val; }
| LC curl=expr RC         //{ val = $curl.val; }
)                         //{ if(neg) { val = new UnOpExpr(UnOp.Neg,val);
   }  }
;

var returns RuleVar: // returns [Var val]:
val+=ID                         //{ val = new Var($ID.text); }
```

```
| val+=ID DOT val+=ID            //{ ArrayList<Expr> index = new
    ArrayList<Expr>(); }
('[' i+=numExp ']'       //{ index.add(i);}
)*                      //{ val = new ConnectionVar($c.text, new Var($n.
    text),index);  }
;

function returns RuleFunction: // returns [FuncExpr val]:
ID LP l=params RP // { val = FuncExpr.createFunction($ID.text,$params.l)
    ;  }
;

params returns RuleParams: //returns [List<Expr> l]:
l+=var        //{ l= new ArrayList<Expr>() ; l.add($v1.val); }
( COMMA l+=var //{ l.add($v2.val) ; }
)*
;


literal returns RuleLiteral:// returns [Value val]:
floatlit                    //{ val = new FloatValue(f);   }
| integerlit                  //{ val = new IntValue(i);       }
| stringlit                   //{ val = new StringValue(s); }
| booleanlit                  //{ val = new BooleanValue(b);  }
| tuplelit                    //{ val = t; }
| seqlit                      //{ val = l; }
| nodelit                     //{ val = new NodeValue(n); }
;

integerlit returns RuleIntegerLit: //returns [int val]:
val=INT             //{ val = Integer.parseInt($DIGITS.text); }
;

floatlit returns RuleFloatLit:
val=FLOAT
;

terminal FLOAT returns ecore::EFloat:
('-')? INT '.' INT
;

stringlit: // returns [String val]:
STRING                        //{ val=$STRING.text ; }
```

```
;

nodelit: // returns [String val]:
'node<' ID '>'        //{ val = $name.text ; }
;

booleanlit: // returns [boolean val]:
TRUE                        //{ val = true; }
| FALSE                       //{ val = false; }
;

tuplelit returns RuleTupleLit: //returns [TupleValue val]  @init{
    ArrayList<Value> vals = new ArrayList<Value>();}
'tuple<' vals+=literal                      //{ vals.add(m);}
( ',' vals+=literal                    //{ vals.add(m);}
)+
'>'                             //{ val = new TupleValue(vals); }
;

seqlit returns RuleSeqLit: //returns [SequenceValue val]  @init{
    ArrayList<Value> vals = new ArrayList<Value>();}
'seq<' vals+=literal                        //{ vals.add(m);}
( ',' vals+=literal                     //{ vals.add(m);}
)+
'>'                                  //{ val = new SequenceValue(vals);
    }
;

//MODALITIES
STEPS: 'steps' ;
T_UNITS: 'time' 'units' ;

// ATOMS
FALSE : 'false';
TRUE : 'true';

// ARITH
ADD: '+';
MIN: '-';
MUL: '*';
DIV: '/';

// BOOL
```

```
NOT: 'not';
AND: 'and';
OR: 'or';
IMP: 'implies';

// 3 valued to 2 valued logic
ISTRUE: 'isTrue';
ISNFLS: 'isNotFalse';

// COMPARISONS
EQ: '=';
NEQ: '!=';
LT: '<';
LE: '<=';
GE: '>=';
GT: '>';


// OTHER SYMBOLS
LP: '(';
RP: ')';
LB: '[';
RB: ']';
LC: '{';
RC: '}';
//SH: '#';
COL:':';
SEMI:';';
COMMA:',';
DQ: '"';
COLEQ: ':=';
DOT: '.';
```

APPENDIX D – Arch1 SosADL Description

```
with A1_ServersXor_library

sos ServersXor0 is {

// fmsos
datatype Location is integer{0..0} {
function (self:Location)::f():Location is {
return self
}
}
datatype String is integer{0..0} {
function (self:String)::f():String is {
return self
}
}
datatype Participant is integer{0..0} {
function (self:Participant)::f():Participant is {
return self
}
}

datatype RainAlert is integer{0..0} {
function (self:RainAlert)::f():RainAlert is {
return self
}
}
datatype FloodWarning is integer{0..0} {
function (self:FloodWarning)::f():FloodWarning is {
return self
}
}
datatype Parameter is integer{0..0} {
```

```
function (self:Parameter)::f():Parameter is {
return self
}
}
datatype Message is integer{0..0} {
function (self:Message)::f():Message is {
return self
}
}
datatype WaterLevel is integer{0..0} {
function (self:WaterLevel)::f():WaterLevel is {
return self
}
}
datatype Image is integer{0..0} {
function (self:Image)::f():Image is {
return self
}
}

datatype Information is integer{0..0} {
function (self:Information)::f():Information is {
return self
}
}
datatype WeatherBulletin is integer{0..0} {
function (self:WeatherBulletin)::f():WeatherBulletin is {
return self
}
}

// constituents
system MeteorologicalSystem() is {
gate ack is {
connection ack is in{Information}
}
guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
```

```
gate ProduceWeatherBulletin is {
connection i1 is in{Location}
connection o1 is out{WeatherBulletin}
connection e1 is out{RainAlert}
}
guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
gate ProvideInformation is {
connection i1 is in{Location}
connection i2 is in{Parameter}
connection o1 is out{Information}
} guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
gate MonitorRegion is {
connection i1 is in{Location}
connection e1 is out{RainAlert}
connection e2 is out{FloodWarning}
} guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}

behavior main is {
done
}
}

system SurveillanceSystem() is {
gate ack is {
connection ack is in{Information}
```

```
}
guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
gate ProvideImages is {
connection i1 is in{Location}
connection o1 is out{Image}
} guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
gate CalculateRiskyArea is {
connection i1 is in{Location}
connection i2 is in{integer{0..0}}
connection o1 is out{sequence{Location}}
} guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
behavior main is {
done
}
}

system RiverMonitoringSystem() is {
gate ack is {
connection ack is in{Information}
}
guarantee {
protocol behavior0 is {
repeat {
anyaction
}
```

```
}
}
gate ProvideRiverLevel is {
connection o1 is out{Information}
} guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
behavior main is {
done
}
}

system SocialNetwork() is {
gate ack is {
connection ack is in{Information}
}
guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
gate ReceiveMessage is {
connection e1 is out{Message}
}guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
gate SendMessage is {
connection i1 is in{Message}
}guarantee {
protocol behavior0 is {
repeat {
anyaction
}
```

```
}
}
gate SearchParticipantsByName is {
connection i1 is in{String}
connection o1 is out{sequence{Participant}}
}guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
gate SearchParticipantsByLocation is {
connection i1 is in{Location}
connection o1 is out{sequence{Participant}}
}guarantee {
protocol behavior0 is {
repeat {
anyaction
}
}
}
behavior main is {
done
}
}

mediator DetectFlood() is {
duty WarningAndLocation is {
connection warning is in{FloodWarning}
connection location is in{Location}
connection locationOut is out{Location}
} assume {
protocol behavior0 is {
repeat { anyaction }
}
}
guarantee {
protocol behavior0 is {
repeat { anyaction }
}
}
duty ConfirmWarning is {
```

```
connection rain is in{RainAlert}
connection warning is in {FloodWarning}
connection riverLevel is in{Information}
connection whetherBulletin is in{Information}
connection trueWarning is out{FloodWarning}
} assume {
protocol behavior0 is {
repeat { anyaction }
}
}guarantee {
protocol behavior0 is {
repeat { anyaction }
}
}
duty CalculateRiskyArea is {
connection location is in{Location}
connection range is in{Parameter}
connection area is out{Parameter}
}assume {
protocol behavior0 is {
repeat { anyaction }
}
}guarantee {
protocol behavior0 is {
repeat { anyaction }
}
}duty ack is {
connection msg is out{Information}
}assume {
protocol behavior0 is {
repeat { anyaction }
}
}guarantee {
protocol behavior0 is {
repeat { anyaction }
}
}
behavior main is {
done
}
}

mediator SendAlert() is {
```

```
duty sendAlert is {
connection alert is in{FloodWarning}
connection msg is in{Message}
}assume {
protocol behavior0 is {
repeat { anyaction }
}
}guarantee {
protocol behavior0 is {
repeat { anyaction }
}
}
duty authorizeAlert is {
connection alert is out{FloodWarning}
connection authorization is in{Information}
}assume {
protocol behavior0 is {
repeat { anyaction }
}
}guarantee {
protocol behavior0 is {
repeat { anyaction }
}
}
duty ack is {
connection msg is out{Information}
}assume {
protocol behavior0 is {
repeat { anyaction }
}
}guarantee {
protocol behavior0 is {
repeat { anyaction }
}
}
behavior main is {
done
}
}


// architecture
architecture fmsos() is {
```

```
gate unusedGate0 is {
connection unusedConnection0 is in{RangeType0}
}
guarantee {
protocol allowAll is {
repeat {
anyaction
}
}
}
behavior coalition is compose {
meteo is MeteorologicalSystem
rms is RiverMonitoringSystem
sn is SocialNetwork
surv is SurveillanceSystem
df is DetectFlood
sa is SendAlert
} binding { unify one { surv :: ack :: ack } to one { df :: ack :: msg }
    and unify one { df :: ack :: msg } to one { rms :: ack :: ack } and
   unify one {df::WarningAndLocation::warning} to one{rms::MonitorRegion
   ::e2}
}

}
}
```

APPENDIX E – Arch-M2Arch SosADL Description

```
with A1_ServersXor_library

sos ServersXor0 is {

  // fmsos
  datatype Location is integer{0..0} {
    function (self:Location)::f():Location is {
      return self
    }
  }
  datatype String is integer{0..0} {
    function (self:String)::f():String is {
      return self
    }
  }
  datatype Participant is integer{0..0} {
    function (self:Participant)::f():Participant is {
      return self
    }
  }

  datatype RainAlert is integer{0..0} {
    function (self:RainAlert)::f():RainAlert is {
      return self
    }
  }
  datatype FloodWarning is integer{0..0} {
    function (self:FloodWarning)::f():FloodWarning is {
      return self
    }
```

```
}
datatype Parameter is integer{0..0} {
  function (self:Parameter)::f():Parameter is {
    return self
  }
}
datatype Message is integer{0..0} {
  function (self:Message)::f():Message is {
    return self
  }
}
datatype WaterLevel is integer{0..0} {
  function (self:WaterLevel)::f():WaterLevel is {
    return self
  }
}
datatype Image is integer{0..0} {
  function (self:Image)::f():Image is {
    return self
  }
}

datatype Information is integer{0..0} {
  function (self:Information)::f():Information is {
    return self
  }
}
datatype WeatherBulletin is integer{0..0} {
  function (self:WeatherBulletin)::f():WeatherBulletin is {
    return self
  }
}

// constituents
system MeteorologicalSystem() is {
  gate ProduceWeatherBulletin is {
    connection i1 is in{Location}
    connection o1 is out{WeatherBulletin}
    connection e1 is out{RainAlert}
  }
  guarantee {
    protocol behavior0 is {
      repeat {
```

```
        anyaction
      }
    }
  }
  gate ProvideInformation is {
    connection i1 is in{Location}
    connection i2 is in{Parameter}
    connection o1 is out{Information}
  } guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
  gate MonitorRegion is {
    connection i1 is in{Location}
    connection e1 is out{RainAlert}
    connection e2 is out{FloodWarning}
  } guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }

  behavior main is {
    done
  }
}

system SurveillanceSystem() is {
  gate ProvideImages is {
    connection i1 is in{Location}
    connection o1 is out{Image}
  } guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
```

```
  gate CalculateRiskyArea is {
    connection i1 is in{Location}
    connection i2 is in{integer{0..0}}
    connection o1 is out{sequence{Location}}
  } guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
  behavior main is {
    done
  }
}

system RiverMonitoringSystem() is {
  gate ProvideRiverLevel is {
    connection o1 is out{Information}
  } guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
  behavior main is {
    done
  }
}

system SocialNetwork() is {
  gate ReceiveMessage is {
    connection e1 is out{Message}
  }guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
  gate SendMessage is {
    connection i1 is in{Message}
```

```
  }guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
  gate SearchParticipantsByName is {
    connection i1 is in{String}
    connection o1 is out{sequence{Participant}}
  }guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
  gate SearchParticipantsByLocation is {
    connection i1 is in{Location}
    connection o1 is out{sequence{Participant}}
  }guarantee {
    protocol behavior0 is {
      repeat {
        anyaction
      }
    }
  }
  behavior main is {
    done
  }
}

mediator ToMatchData() is {
  duty duty0 is {
    connection i1 is in{Information}
    connection i2 is in{Information}
    connection o1 is out{Information}
  } assume {
    protocol behavior0 is {
      repeat { anyaction }
    }
  } guarantee {
    protocol behavior0 is {
```

```
      repeat { anyaction }
    }
  }
  behavior main is {
    done
  }
}
mediator SendAlert() is {
  duty duty0 is {
    connection e1 is in{RainAlert}
    connection i1 is in{sequence{Participant}}
    connection i2 is in{sequence{Participant}}
    connection o1 is out{Participant}
    connection o2 is out{Message}
  } assume {
    protocol behavior0 is {
      repeat { anyaction }
    }
  } guarantee {
    protocol behavior0 is {
      repeat { anyaction }
    }
  }
  behavior main is {
    done
  }
}

mediator LocationToSN() is {
  duty duty0 is {
    connection i1 is in{sequence{Location}}
    connection o1 is out{Location}
  } assume {
    protocol behavior0 is {
      repeat { anyaction }
    }
  } guarantee {
    protocol behavior0 is {
      repeat { anyaction }
    }
  }
  behavior main is {
    done
```

```
  }
}

mediator AreaMonitor() is {
  duty duty0 is {
    connection i1 is in{sequence{Location}}
    connection o1 is out{Location}
  } assume {
    protocol behavior0 is {
      repeat { anyaction }
    }
  } guarantee {
    protocol behavior0 is {
      repeat { anyaction }
    }
  }
  behavior main is {
    done
  }
}

mediator ParticipantsByLocation() is {
  duty duty0 is {
    connection i1 is in{sequence{Location}}
    connection o1 is out{Location}
  } assume {
    protocol behavior0 is {
      repeat { anyaction }
    }
  } guarantee {
    protocol behavior0 is {
      repeat { anyaction }
    }
  }
  behavior main is {
    done
  }
}
mediator ParticipantsInArea() is {
  duty duty0 is {
    connection i1 is in{sequence{Location}}
    connection o1 is out{Location}
  } assume {
```

```
    protocol behavior0 is {
      repeat { anyaction }
    }
  } guarantee {
    protocol behavior0 is {
      repeat { anyaction }
    }
  }
  behavior main is {
    done
  }
}



// architecture
architecture coalition() is {
  gate unusedGate0 is {
    connection unusedConnection0 is in{RangeType0}
  }
  guarantee {
    protocol allowAll is {
      repeat {
        anyaction
      }
    }
  }
  behavior coalition is compose {
    meteo is MeteorologicalSystem
    rms is RiverMonitoringSystem
    sn is SocialNetwork
    surv is SurveillanceSystem
    toMatch is ToMatchData
    sa is SendAlert
    loc is LocationToSN
    am is AreaMonitor
    pbl is ParticipantsByLocation
    pia is ParticipantsInArea
  } binding { ( unify one { meteo :: ProvideInformation :: i1 } to one
      { toMatch :: duty0 :: i1 } and unify one { surv ::
     CalculateRiskyArea :: o1 } to one { loc :: duty0 :: o1 } and
     unify one { sa :: duty0 :: o1 } to one { sn :: SendMessage :: i1
     } and unify one { pbl :: duty0 :: o1 } to one { sa :: duty0 :: o2
```

```
        } and unify one { sn :: SearchParticipantsByLocation :: i1 } to
       one { pbl :: duty0 :: i1 } and unify one { loc :: duty0 :: i1 }
       to one { sn :: SearchParticipantsByLocation :: o1 } and unify one
        { toMatch :: duty0 :: i2 } to one { sa :: duty0 :: e1 } and
       unify one { meteo :: MonitorRegion :: e1 } to one { sa :: duty0
       :: i2 } and unify one { pbl :: duty0 :: i1 } to one { sn ::
       SearchParticipantsByLocation :: i1 } and unify one { surv ::
       CalculateRiskyArea :: i2 } to one { toMatch :: duty0 :: o1 } and
       unify one { rms :: ProvideRiverLevel :: o1 } to one { surv ::
       CalculateRiskyArea :: i1 } and unify one { rms ::
       ProvideRiverLevel :: o1 } to one { pbl :: duty0 :: o1 } and (
       unify one { rms :: ProvideRiverLevel :: o1 } to one { sa :: duty0
        :: i1 } and unify one { am :: duty0 :: o1 } to one { meteo ::
       MonitorRegion :: e2 } ) and unify one { surv :: ProvideImages ::
       o1 } to one { am :: duty0 :: i1 } and ( unify one { loc :: duty0
       :: i1 } to one { sn :: SendMessage :: i1 } and unify one { meteo
       :: MonitorRegion :: i1 } to one { loc :: duty0 :: o1 } ) and (
      unify one{meteo::ProvideInformation::o1} to one {toMatch::duty0::
         i1})
      )
    }

  }
}
```

APPENDIX F – K3 aspect file for SosADL

Made using Kermetta3.

```
package sosADL.aspects

import fr.inria.diverse.k3.al.annotationprocessor.Aspect
import fr.inria.diverse.k3.al.annotationprocessor.InitializeModel
import fr.inria.diverse.k3.al.annotationprocessor.Main
import fr.inria.diverse.k3.al.annotationprocessor.Step
import java.util.LinkedList
import java.util.List
import org.archware.sosadl.sosADL.ArchitectureDecl
import org.archware.sosadl.sosADL.BinaryExpression
import org.archware.sosadl.sosADL.Connection
import org.archware.sosadl.sosADL.Constituent
import org.archware.sosadl.sosADL.Expression
import org.archware.sosadl.sosADL.GateDecl
import org.archware.sosadl.sosADL.IdentExpression
import org.archware.sosadl.sosADL.MediatorDecl
import org.archware.sosadl.sosADL.SystemDecl
import org.archware.sosadl.sosADL.Unify
import org.eclipse.emf.common.util.EList
import org.eclipse.emf.ecore.EObject
import java.util.Random
import org.archware.sosadl.sosADL.DutyDecl
import org.archware.sosadl.utility.ModelUtils

//import org.eclipse.gemoc.executionframework.engine.annotations.
    EventHandler
//import org.eclipe.gemoc.executionframework.engine.annotations.
    EventEmitter
```

```
//import org.eclipse.gemoc.executionframework.engine.annotations.
    EventHandler

@Aspect(className=ArchitectureDecl)
class ArchitectureDeclAspect {
//private Map<String, String> context;
@Main
//@EventHandler // handles NEW_STATE and END
//@EventEmitter
def public void main() {
// propagate values from input gates
for (GateDecl g : _self.gates) {
for (Connection c : g.connections) {
ConnectionAspect.propagateValue(c)
}
}


// try to execute component's behavior
for (Constituent c : _self.behavior.constituents) {
val EObject o = ModelUtils.resolve(c.value as IdentExpression)
if (o instanceof SystemDecl) {
SystemDeclAspect.run(o)//, _self.context)
} else if (o instanceof MediatorDecl) {
MediatorDeclAspect.run(o)//, _self.context)
}
}
}

@InitializeModel
//@EventHandler // handles INIT and NEW_TRACE
def public void init(EList<String> args) {
for (GateDecl gate : _self.gates) {
// random values into connections
for (Connection c : gate.connections) {
ConnectionAspect.value(c, Values.empty) // initialize values with empty
}
}
// unify gates
ExpressionAspect.performAction(_self.behavior.bindings)
println("Started")
}
```

```
}

@Aspect(className=SystemDecl)
class SystemDeclAspect {
@Step
def public void run(){//Map<String, String> context) {
println("Running "+_self.name)
}
}

@Aspect(className=MediatorDecl)
class MediatorDeclAspect {
@Step
def public void run(){//Map<String, String> context) {
println("Running "+_self.name)
}
}

@Aspect(className=Unify)
class UnifyAspect extends ExpressionAspect {
public def void performAction() {
val Connection left = ModelUtils.resolve(_self.connLeft) as Connection
val Connection right = ModelUtils.resolve(_self.connRight) as Connection
if (ConnectionAspect.unifiedConnections(left) === null)
ConnectionAspect.unifiedConnections(left, new LinkedList<Connection>())
if (ConnectionAspect.unifiedConnections(right) === null)
ConnectionAspect.unifiedConnections(right, new LinkedList<Connection>())
ConnectionAspect.unifiedConnections(left).add(right)
ConnectionAspect.unifiedConnections(right).add(left)
}
}

@Aspect(className=BinaryExpression)
class BinaryExpressionAspect extends ExpressionAspect {
public def void performAction() {
_self.left.performAction // recursive call to unify
_self.right.performAction // recursive call to unify
}
}

@Aspect(className=Expression)
abstract class ExpressionAspect {
```

```
public def Object evaluate() {
println("Evaluating expression "+_self)
return null
}

abstract def void performAction()
}

@Aspect(className=Connection)
class ConnectionAspect {
protected String value
protected List<Connection> unifiedConnections

def public void propagateValue() {
for (Connection c : unifiedConnections(_self)) {
if (ConnectionAspect.value(c) != ConnectionAspect.value(_self)) {
// copy value
ConnectionAspect.value(c, ConnectionAspect.value(_self))
// propagate recursivelly
ConnectionAspect.propagateValue(c)
}
}
}
}

@Aspect(className=GateDecl)
class DateDeclAspect {
public Object value
}

@Aspect(className=DutyDecl)
class DutyDeclAspect {
public Object value
}

class StatementAspect {

}
```

ANEXO A – SosADL Grammar

Made using Xtext.

```
grammar org.archware.sosadl.SosADL with org.eclipse.xtext.common.
    Terminals

generate sosADL 'http://www-archware.irisa.fr/sosadl/SosADL'

SosADL: (imports+=Import)* content=(NewNamedLibrary | NewSoS)
;

Import: 'with' importedLibrary=Name
;

NewNamedLibrary returns Unit: {Library} 'library' name=Name 'is' '{'
    decls=EntityBlock '}'
;

NewSoS returns Unit: {SoS} 'sos' name=Name 'is' '{'
(decls=EntityBlock)
'}'
;

EntityBlock: {EntityBlock}
(datatypes+=DataTypeDecl)*
(functions+=FunctionDecl)*
(systems+=SystemDecl)*
(mediators+=MediatorDecl)*
(architectures+=ArchitectureDecl)*
;

SystemDecl: 'system' name=Name '(' (parameters+=FormalParameter (','
    parameters+=FormalParameter)*)? ')' 'is' '{'
```

```
(datatypes+=DataTypeDecl)*
(gates+=GateDecl)+
behavior=BehaviorDecl
'}' ('guarantee' '{' assertions+=AssertionDecl+ '}')?
;

ArchitectureDecl: 'architecture' name=Name '(' (parameters+=
    FormalParameter (',' parameters+=FormalParameter)*)? ')' 'is' '{'
(datatypes+=DataTypeDecl)*
(gates+=GateDecl)+
behavior=ArchBehaviorDecl
'}' ('guarantee' '{' assertions+=AssertionDecl+ '}')?
;

MediatorDecl: 'mediator' name=Name '(' (parameters+=FormalParameter (','
     parameters+=FormalParameter)*)? ')' 'is' '{'
(datatypes+=DataTypeDecl)*
(duties+=DutyDecl)+
behavior=BehaviorDecl
'}'
('assume' '{' assumptions+=AssertionDecl+ '}')?
('guarantee' '{' assertions+=AssertionDecl+ '}')?
;

GateDecl:
'gate' name=Name 'is' '{'
(connections+=Connection)+
'}' 'guarantee' '{' protocols+=ProtocolDecl+ '}'
;

DutyDecl:
'duty' name=Name 'is' '{'
(connections+=Connection)+
'}'
'assume' '{' assertions+=AssertionDecl+ '}'     // WAS: 'require' '{'
    assertion=AssertionDecl '}'
'guarantee' '{' protocols+=ProtocolDecl+ '}'    // WAS: 'assume' '{'
    protocol=ProtocolDecl '}'
;

Connection:
(environment?='environment')? 'connection' name=Name 'is' mode=ModeType
    '{' valueType=DataType '}'
```

```
;

AssertionDecl:
('property'|'protocol') name=Name 'is' body=Protocol
;

ProtocolDecl:
('property'|'protocol') name=Name 'is' body=Protocol
;

Protocol:
'{' (statements+=ProtocolStatement)+ '}'
;

ProtocolStatement:
{ValuingProtocol} valuing=Valuing
| {AssertProtocol} assertion=Assert
| ProtocolAction
| {AnyAction} 'anyaction'
| {RepeatProtocol} 'repeat' repeated=Protocol
| {IfThenElseProtocol} 'if' condition=Expression 'then' ifTrue=Protocol
   ('else' ifFalse=Protocol)?
| {ChooseProtocol} 'choose' branches+=Protocol  ('or' branches+=Protocol
   )+
| {ForEachProtocol} 'foreach' variable=Name 'in' setOfValues=Expression
   repeated=Protocol
| {DoExprProtocol} 'do' expression=Expression
| {DoneProtocol} 'done'
;

ProtocolAction:
'via' complexName=ComplexName suite=ProtocolActionSuite
;

ProtocolActionSuite:
({SendProtocolAction} 'send' expression=FinalExpression)
| ('receive' ({ReceiveAnyProtocolAction} 'any'
|{ReceiveProtocolAction} variable=Name))
;

BehaviorDecl:
'behavior' name=Name 'is' body=Behavior
// WAS: 'behavior' name=Name '(' (parameters+=FormalParameter (','
```

```
    parameters+=FormalParameter)*)? ')' 'is' body=Behavior
;

Behavior:
'{' (statements+=BehaviorStatement)+ '}'
;

BehaviorStatement:
{ValuingBehavior} valuing=Valuing
| {AssertBehavior} assertion=Assert
| Action
| {RepeatBehavior} 'repeat' repeated=Behavior
| {IfThenElseBehavior} 'if' condition=Expression 'then' ifTrue=Behavior
    ('else' ifFalse=Behavior)?
| {ChooseBehavior} 'choose' branches+=Behavior  ('or' branches+=Behavior
    )+
| {ForEachBehavior} 'foreach' variable=Name 'in' setOfValues=Expression
    repeated=Behavior
| {DoExprBehavior} 'do' expression=Expression
| {DoneBehavior} 'done'
| {RecursiveCall} 'behavior' '(' (parameters+=Expression (',' parameters
    +=Expression)*)? ')'
| {UnobservableBehavior} 'unobservable'
;

Assert:
{TellAssertion} 'tell' name=Name 'is' '{' expression=Expression '}'
| {UntellAssertion} 'untell' name=Name
| {AskAssertion} 'ask' name=Name 'is' '{' expression=Expression '}'
;

Action:
'via' complexName=ComplexName suite=ActionSuite
;

ActionSuite:
{SendAction} 'send' expression=FinalExpression
| {ReceiveAction} 'receive' variable=Name
;

ArchBehaviorDecl:
// WAS: 'behavior' name=Name '(' (parameters+=Expression (',' parameters
    +=Expression)*)? ')'
```

```
'behavior' name=Name
'is' 'compose' '{' (constituents+=Constituent)+ '}'
'binding' '{' bindings=Expression '}'
;

Constituent:
name=Name 'is' value=Expression
;

Binding returns Expression:
{Relay} 'relay' connLeft=ComplexName 'to' connRight=ComplexName
| {Unify} 'unify' multLeft=Multiplicity '{' connLeft=ComplexName '}' 'to
    ' multRight=Multiplicity '{' connRight=ComplexName '}'
| {Quantify} quantifier=Quantifier '{' elements+=ElementInConstituent ('
    ,' elements+=ElementInConstituent)* 'suchthat' bindings=Expression '}
    '
;

enum Quantifier:
QuantifierForall='forall' | QuantifierExists='exists'
;

ElementInConstituent:
variable=Name 'in' constituent=Name
;

enum Multiplicity:
MultiplicityOne='one'
| MultiplicityNone='none'
| MultiplicityLone='lone'
| MultiplicityAny='any'
| MultiplicitySome='some'
| MultiplicityAll='all'
;

DataTypeDecl: 'datatype' name=Name ('is' datatype=DataType)? ('{'
    functions+=FunctionDecl+ '}')?;

DataType:
BaseType
| ConstructedType
| {NamedType} name=Name  // name of another type
;
```

```
FunctionDecl:
'function' '(' data=FormalParameter ')' '::'
name=Name '(' (parameters+=FormalParameter (',' parameters+=
    FormalParameter)*)? ')' ':' type=DataType 'is' '{'
(valuing+=Valuing)*
'return' expression=Expression
'}'
;

FormalParameter:
name=Name ':' type=DataType
;

BaseType returns DataType:
{IntegerType} 'integer'
;

ConstructedType returns DataType:
{TupleType} 'tuple' '{' fields+=FieldDecl (',' fields+=FieldDecl)* '}'
| {SequenceType} 'sequence' '{' type=DataType '}'
| {RangeType} 'integer' '{' vmin=Expression '..' vmax=Expression '}'  //
    range of Integer
| {ConnectionType} mode=ModeType '{' type=DataType '}'
;

FieldDecl:
name=Name ':' type=DataType
;

enum ModeType:
ModeTypeIn='in' | ModeTypeOut='out' | ModeTypeInout='inout';

Name: ID ;

ComplexName:
name+=Name ('::' name+=Name)*
;

Valuing:
'value' name=Name (':' type=DataType)? '=' expression=Expression;

Value returns Expression:
```

```
BaseValue
| ConstructedValue
;

BaseValue returns Expression:
IntegerValue
| {Any} 'any'
;

// IntegerValue is a natural integer (>=0). Use a UnaryExpression to get
    a negative value.
IntegerValue:
absInt=INT // INT == ('0'..'9')+ rend une valeur ecore::EInt;
;

ConstructedValue returns Expression:
{Tuple} 'tuple' '{' elements+=TupleElement (',' elements+=TupleElement)*
    '}'
| {Sequence} 'sequence' '{' (elements+=Expression (',' elements+=
    Expression)*)? '}'
;

TupleElement:
label=Name '=' value=Expression
;

Expression:
BinaryExpression0
;

BinaryExpression0 returns Expression:
BinaryExpression1 ({BinaryExpression.left=current} op=BinaryOp0 right=
    BinaryExpression0)?
;

BinaryExpression1 returns Expression:
BinaryExpression2 ({BinaryExpression.left=current} op=BinaryOp1 right=
    BinaryExpression2)*
;

BinaryExpression2 returns Expression:
BinaryExpression3 ({BinaryExpression.left=current} op=BinaryOp2 right=
    BinaryExpression3)*
```

```
;

BinaryExpression3 returns Expression:
BinaryExpression4 ({BinaryExpression.left=current} op=BinaryOp3 right=
    BinaryExpression4)*
;

BinaryExpression4 returns Expression:
BinaryExpression5 ({BinaryExpression.left=current} op=BinaryOp4 right=
    BinaryExpression5)*
;

BinaryExpression5 returns Expression:
BinaryExpression6 ({BinaryExpression.left=current} op=BinaryOp5 right=
    BinaryExpression6)*
;

BinaryExpression6 returns Expression:
BinaryExpression7 ({BinaryExpression.left=current} op=BinaryOp6 right=
    BinaryExpression7)*
;

BinaryExpression7 returns Expression:
FinalExpression ({BinaryExpression.left=current} op=BinaryOp7 right=
    FinalExpression)*
;

FinalExpression returns Expression:
UnaryExpression
| CallExpression
| '(' Expression ')'
| Binding
;

UnaryExpression:
op=UnaryOp right=FinalExpression
;

CallExpression returns Expression:
(
{IdentExpression} ident=Name
| {CallExpression} function=Name '(' (parameters+=Expression (','
    parameters+=Expression)*)? ')'
```

```
| LitteralExpression
)

(’::’
(
{Field.object=current} field=Name
| {Select.object=current} ’select’ ’{’ variable=Name ’suchthat’
    condition=Expression ’}’
// WAS: {Map.object=current} ’map’ ’{’ variable=Name ’to’ expression=
    Expression ’}’
| {Map.object=current} ’collect’ ’{’ variable=Name ’suchthat’ expression
    =Expression ’}’
| {MethodCall.object=current} method=Name ’(’ (parameters+=Expression (’
    ,’ parameters+=Expression)*)? ’)’
)
)*

;

LitteralExpression returns Expression:
Value
;

BinaryOp0:  ’implies’ ;

BinaryOp1:  ’or’ ;

BinaryOp2:  ’xor’ ;

BinaryOp3:  ’and’ ;

BinaryOp4:  ’=’ | ’<>’ ;

BinaryOp5:  ’<’ | ’<=’ | ’>’ | ’>=’ ;

BinaryOp6:  ’+’ | ’-’ ;

BinaryOp7:  ’*’ | ’/’ | ’mod’ | ’div’ ;

UnaryOp:
BooleanUnaryOp
| ArithmeticUnaryOp
;
```

```
BooleanUnaryOp: 'not';

ArithmeticUnaryOp:  '+' | '-';

HiddenBooleanType returns DataType:
{BooleanType}
;

// the end.
```

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

Titre : Conception d'architecture de système-de-systèmes à logiciel prépondérant dirigée par les missions

Mots clés : Architecture logicielle, systèmes-de-systèmes à logiciel prépondérant, modélisation de missions, conception architecturale semi-automatisée

Résumé : La formulation des missions est le point de départ du développement de systèmes-de-systèmes, étant utilisée comme base pour la spécification, la vérification et la validation d'architectures de systèmes-de-systèmes.

Élaborer des modèles d'architecture pour systèmes-de-systèmes est une activité complexe, cette complexité reposant spécialement sur les comportements émergents, c'est-à-dire, des comportements issus des interactions entre les parties constituantes d'un système-de-systèmes qui ne peuvent pas être prédits même si on connaît tous les comportements de tous les systèmes constituants.

Cette thèse adresse le lien synergique entre mission et architecture dans le cadre des systèmes-de-systèmes à logiciel prépondérant, en accordant une attention particulière aux comportements émergents créés pour réaliser les missions formulées. Nous proposons ainsi une approche pour la conception d'architecture de systèmes-de-systèmes dirigée par le modèle de mission.

Dans notre approche, le modèle de mission sert à dériver et à valider les architectures de systèmes-de-systèmes. Dans un premier temps, nous générons la structure de l'architecture à l'aide de transformations de modèles. Ensuite, lors que l'architecte spécifie les aspects comportementaux, la description de l'architecture résultante est validée à l'aide d'une démarche conjointe qui comprend à la fois la vérification des propriétés spécifiées et la validation par simulation des comportements émergents. La formalisation en termes de logique temporelle et la vérification statistique de modèles sont les fondements formels de l'approche. Un outil mettant en œuvre l'ensemble de l'approche a été également développé et expérimenté.

Title: Mission-driven Software-intensive System-of-Systems Architecture Design

Keywords: Software Architecture, Software-intensive Systems-of-Systems, Mission Modeling, Semi-Automated Architecture Design

Abstract: The formulation of missions is the starting point to the development of Systems-of-Systems (SoS), being used as a basis for the specification, verification and validation of SoS architectures.

Specifying, verifying and validating architectural models for SoS are complex tasks compared to usual systems, the inner complexity of SoS relying specially on emergent behaviors, i.e. features that emerge from the interactions among constituent parts of the SoS which cannot be predicted even if all the behaviors of all parts are completely known.

This thesis addresses the synergetic relationship between missions and architectures of software-intensive SoS, giving a special attention to emergent behaviors which are created for achieving formulated missions. We propose a design approach for the architectural modeling of SoS driven by the mission models.

In our proposal, the mission model is used to both derive, verify and validate SoS architectures. As first step, we define a formalized mission model, then we generate the structure of the SoS architecture by applying model transformations. Later, when the architect specifies the behavioral aspects of the SoS, we generate concrete SoS architectures that will be verified and validated using simulation-based approaches, in particular regarding emergent behaviors. The verification uses statistical model checking to verify whether specified properties are satisfied, within a degree of confidence. The formalization in terms of a temporal logic and statistical model checking are the formal foundations of the developed approach. A toolset that implements the whole approach was also developed and experimented.