



HAL
open science

Formal Modeling and Analysis of Time-Resource Aware Systems-of-Systems.

Charaf Dridi

► **To cite this version:**

Charaf Dridi. Formal Modeling and Analysis of Time-Resource Aware Systems-of-Systems.. Computation and Language [cs.CL]. Université de Pau et des Pays de l'Adour; Université Abdelhamid Mehri - Constantine 2 (Constantine, Algérie), 2024. English. NNT : 2024PAUU3071 . tel-04739443

HAL Id: tel-04739443

<https://theses.hal.science/tel-04739443v1>

Submitted on 16 Oct 2024

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

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



University of Constantine 2 Abdelhamid Mehri - Faculty of New Technologies of
Computing and Communication - Department of Software Technologies and
Information Systems

University of Pau and Adour Countries - Doctoral School of Exact Sciences and their
Applications (ED SEA 211)

Formal Modeling and Analysis of Time-Resource Aware Systems-of-Systems

THESIS

submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Prepared within a joint supervision between

UNIVERSITY OF PAU AND ADOUR COUNTRIES, FRANCE
and

UNIVERSITY OF CONSTANTINE 2 ABDELHAMID MEHRI, ALGERIA

Presented by

Charaf Eddine DRIDI

Defended on July 3, 2024, 02:00 p.m. at UC2 in front of the jury

President:	Pr. Mahmoud BOUFAIDA	University of Constantine 2 Abdelhamid Mehri, Algeria
Reviewers:	Pr. Ahmed HADJ KACEM	University of Sfax, Tunis
	Pr. Khalil DRIRA	University of Toulouse, France
Examiner:	Dr. Sofia KOUAH	University of Larbi Ben M'Hidi, Algeria
Supervisors:	Dr. Nabil HAMEURLAIN	University of Pau and Adour Countries, France
	Pr. Faiza BELALA	University of Constantine 2 Abdelhamid Mehri, Algeria

Thesis prepared at [Laboratoire Informatique de l'Université de Pau et des Pays de l'Adour \(LIUPPA\)](#) and [Laboratoire d'Informatique REpartie \(LIRE\)](#)

Acknowledgements

I want to express my deepest gratitude to my supervisors at two distinguished universities. I thank my supervisor at University Constantine2, Mrs. Faiza Belala, for proposing that I work on such an interesting research topic. Our meetings were always fruitful and beneficial, contributing significantly to my understanding and progress. I would also like to thank her for her guidance, for encouraging me to try new scientific alternatives, and for her professionalism and exemplary behavior throughout my thesis.

Additionally, I am deeply grateful to my supervisor at UPPA, Mr. Nabil Hameurlain for his guidance. His valuable advice and the relevance of his remarks, as well as his human qualities and scientific knowledge, have greatly helped me to progress in my research work. I would also like to thank him for introducing me to his professional network of contacts and for putting me in touch with specialists in the field covering the broad research/academic areas, and for facilitating my integration into the LIUPPA laboratory.

I would like to thank Mr. Ahmed Hadj Kacem, professor at the University of Sfax, and Mr. Khalil DRIRA, professor at the University of Toulouse, for the honor they gave me by agreeing to be reviewers of this thesis, and for the time they dedicated to my work.

I would like to thank Mr. Mahmoud Boufaida, professor at the University of Constantine 2 for agreeing to preside over my thesis jury, as well as Mrs. Sofia Kouah, associate professor at the University of Oum Bouaghi for agreeing to be on my thesis jury as examiner. I thank them for the honor they give me, for their scientific participation, and for the time they have dedicated to my research.

I thank Mr. Eric Cariou, Mr. Abderrahim Aitwakrime, Mr. Chihab Hanachi and, and Mr. Ahmed Hadj Kacem for their participation and availability during the various thesis follow-up committees. They took the time to listen to me and discuss with me. Their comments allowed me to advance my project and to consider my work from another angle.

A big thank you to the members of the Computer Science Department at the STEE College at UPPA, the members of LIUPPA at the Pau site, as well as the staff of the Doctoral School SEA (ED 211) at UPPA, for their warm welcome and support throughout the completion of my work. I would particularly like to thank those who shared my daily life during these years of thesis. I think of the friendships woven and all those people who were there in all circumstances.

My final thanks go to my family, and particularly to my parents, who did everything to help me, who supported and endured me in everything I undertook.

Abstract

Systems of Systems (SoSs) are predominantly large-scale, complex, and sometimes critical software. An SoS is the result of integrating a set of autonomous, heterogeneous, and evolving Constituent Systems (CSs) that interact with each other to fulfill their operational purpose (mission) and to offer new functionalities that exceed those of the individual CSs.

Currently, the modeling, simulation, and analysis of SoS present a significant challenge from a software engineering perspective, due to the complexity associated with the specific characteristics of such systems: temporal constraints linked to missions, the production and consumption of resources by each mission, and finally the emergence of desired or undesired behavior within a CS or the SoS. The work of this thesis aims to propose a generic, model-based approach to describe the architecture of an SoS (and its CSs), the functional behavior of each mission, and the validation through simulation of the global behavior of an SoS. We propose a modular and intuitive multi-view architectural framework based on the ISO/IEC/IEEE 42010 standard. This framework is associated with a set of SoS engineering processes and a UML profile dedicated to SoSs, this framework facilitates and improves the design of SoS architectures. Regarding the behavioral aspect of the SoS, our approach relies on the Maude language as a rewriting logic-based language, to describe, analyze, and validate through modeling the functional behavior of an SoS, integrating the resources produced and consumed by each mission, as well as the temporal constraints of each mission within a CS.

In this thesis, we propose a set of generic and reusable meta-models to describe the various entities involved in resource allocation and mission execution. We use the Maude language to define the operational semantics of various concepts, entities, and properties presented in the meta-models. We adopt a centralized control for the management and allocation of resources and their coordination, specifying missions, roles, and resources, as well as their quantitative properties. Finally, we define a set of strategies for managing the execution of missions. These strategies aim to improve the operational functioning of the SoS by avoiding undesirable behaviors and promoting desirable and optimal mission paths taking into account temporal constraints related to resources and missions, resource availability, etc. In this thesis, we utilize the RT-Maude (Real Time Maude) and Maude Strategy extensions of the Maude language, to describe these strategies and validate the behavior of the SoS using the model-checking verification techniques offered by the Maude language. All these contributions have been validated through a comprehensive case study approach focusing on crisis management SoS.

Keywords: SoS, CS, Missions, Centralized Control, Management Strategies, MAPE-K Loop Control, Runtime, Meta-Modeling, MDA, Formal Modeling and Verification.

Résumé

Les Systèmes de Systèmes (SdS) sont des logiciels prépondérants de grande échelle, complexes et parfois même critiques. Un SdS est le résultat de l'intégration d'un ensemble de Systèmes Constituants (SCs) autonomes, hétérogènes et évolutifs, qui interagissent les uns avec les autres, pour remplir leur propre but (mission) opérationnel et de faire émerger de nouvelles fonctionnalités qui dépassent celles des SCs individuels.

Actuellement, la modélisation, la simulation et l'analyse des SdS posent désormais un grand défi du point de vue génie logiciel, en raison de la complexité liée aux caractéristiques spécifiques de tels systèmes : contraintes temporelles liées aux missions, la production et la consommation des ressources par chaque mission, et enfin l'émergence d'un comportement désiré ou non désiré au sein d'un SC ou du SdS. Les travaux de cette thèse visent à proposer une approche générique, à base de modèles, pour décrire l'architecture d'un SdS (et ses CSs), le comportement fonctionnel de chaque mission ainsi que la validation par simulation du comportement global d'un SdS. Nous proposons un cadre architectural multi-vues modulaire et intuitif qui s'appuie sur la norme ISO/IEC/IEEE 42010. Ce cadre est associé à un ensemble de processus d'ingénierie de SdS et à un profil UML dédié aux SdS, afin de faciliter et d'améliorer la conception des architectures de SdS. Concernant l'aspect comportemental des SdS, notre approche s'appuie sur le langage Maude, un langage à base de la logique de réécriture, pour décrire, analyser et valider par modèle le comportement fonctionnel d'un SdS, intégrant les ressources, produites et consommées par chaque mission, ainsi que les contraintes temporelles de chaque mission au sein d'un CS.

Dans cette thèse, nous proposons un ensemble de méta-modèles génériques et réutilisables pour décrire les différentes entités impliquées dans l'allocation des ressources et l'exécution des missions. Nous utilisons le langage Maude pour définir la sémantique opérationnelle de divers concepts, entités et propriétés présentés dans les méta-modèles. Nous adoptons un contrôle centralisé pour la gestion et l'allocation des ressources et leur coordination, en spécifiant les missions, les rôles et les ressources, ainsi que leurs propriétés quantitatives. Enfin, nous définissons un ensemble de stratégies de gestion de l'exécution des missions. Ces stratégies visent à améliorer le fonctionnement opérationnel du SdS en évitant les comportements indésirables et en favorisant les chemins de mission désirables et optimaux vis à vis des contraintes temporelles liées aux ressources et aux missions, la disponibilité des ressources, etc. Dans cette thèse, nous exploitons les extensions RT-Maude (Real Time Maude) et Maude Strategy du langage Maude, pour décrire ces différentes stratégies et valider le comportement des SdS en utilisant les techniques de vérification par model-checking offertes par le langage Maude. Toutes ces contributions ont été validées par une approche d'étude de cas complète axée sur les SdS de gestion de crise.

Mots clés: SdS, SCs, Missions, Contrôle Centralisé, Stratégies de gestion, Boucle de contrôle MAPE-K, Runtime, Meta-Modélisation, MDA, Modélisation et vérification formelles.

Contents

I	Preamble	3
I.1	Introduction	3
I.1.1	Problem Statement	4
I.1.2	Research Objectives and Contributions	5
I.2	Thesis Chapters	7
I.3	Publications	8
II	Basics and Prerequisites	10
II.1	Introduction	10
II.2	Model-Driven Engineering	11
II.2.1	Model-Driven Architecture	11
II.2.2	IEEE-ISO 42010 Standard	15
II.3	Rewriting Logic and Maude Language	16
II.3.1	Rewriting theories	17
II.3.2	Maude Language: modules and extensions	21
II.3.3	Formal Analysis and Execution	26
II.4	Conclusion	29
III	Key Concepts, Definitions and State of the Art	31
III.1	Introduction	31
III.2	Systems-of-Systems	32
III.2.1	Definitions	32
III.2.2	Dimensions	34
III.2.3	Categories	35
III.2.4	Application domains	39
III.3	Current research on SoSs modeling	42
III.3.1	Semi-formal methods	42
III.3.2	Formal methods	49
III.3.3	Synthesis	54
III.4	Conclusion	56
IV	Methodology and General Principle	57
IV.1	Introduction	57

IV.2	A process for SoS Engineering	58
IV.2.1	Domain Engineering	60
IV.2.2	Application Engineering	62
IV.3	Solution principle	64
IV.3.1	Basic elements description	64
IV.3.2	MDA-based SoS Framework design	65
IV.3.3	Formal semantics of centralized control	65
IV.3.4	Formal specification of management strategies	66
IV.3.5	Autonomic execution and verification	67
IV.4	Conclusion	68
V	Model-Based SoS Framework Domain	69
V.1	Introduction	69
V.2	SoSs Commonalities	70
V.2.1	Abstract assets: Application and Framework Domains	71
V.2.2	A Unified Architecture Framework: Overview	77
V.2.3	Case study: Aircraft Emergency Response SoS	79
V.3	A Multi Viewpoints-based Architecture Framework	81
V.3.1	Architectural concepts	82
V.3.2	Associations	86
V.4	UML Extensions for Modeling the unified architecture	88
V.4.1	SoS_Knowledge_Package	89
V.4.2	CSs_Selection_Package	91
V.4.3	Conceptual_Design_Package	93
V.4.4	Architectural_Design_Package	94
V.4.5	Interaction_Package	95
V.4.6	Integration_Deployment_Package	97
V.5	Conclusion	99
VI	Formalization of Centralized Control in SoSs	101
VI.1	Introduction	101
VI.2	Time-Resource Aware SoSs	102
VI.2.1	Temporal constraints of Missions	102
VI.2.2	Understanding resource categorization	104
VI.2.3	Abstract assets: Variability Domain	106
VI.2.4	Rewriting-based approach for resources allocation control	107
VI.3	Formal semantics of structural entities	108
VI.3.1	Missions and temporal constraints	109
VI.3.2	Resources categorization	110
VI.3.3	Roles encoding	112

VI.3.4	Resource Allocation Controller: RAC	113
VI.4	Formal semantics of dynamic aspects	113
VI.4.1	Missions' lifecycle	114
VI.4.2	Resource' lifecycle	117
VI.4.3	Roles' lifecycle	119
VI.4.4	Resources Allocation Control lifecycle	122
VI.5	Conclusion	126
VII	Control-based Formalization of Management Strategies	127
VII.1	Introduction	127
VII.2	Abstract assets: Variability Domain for Management	128
VII.3	Strategic Management of Behavior	129
VII.3.1	Managing Workflows in SoSs	131
VII.3.2	Mission execution and resource management	133
VII.3.3	Management Strategies	142
VII.4	Real-Time regulating mechanism using MAPE-K loop	146
VII.4.1	Knowledge: Data Foundation	148
VII.4.2	Monitor: Processing	148
VII.4.3	Analysis: Workflow	149
VII.4.4	Plan: Strategic Control and Management	149
VII.4.5	Execution: Rewriting Management System	150
VII.5	Conclusion	153
VIII	Simulation and Formal Verification	154
VIII.1	Introduction	154
VIII.2	Case study	155
VIII.3	Managing FESoS through a MAPE-K loop	156
VIII.3.1	Knowledge: Foundation	158
VIII.3.2	Monitor: Processing	158
VIII.3.3	Analysis: Workflow Analysis	159
VIII.3.4	Plan: Strategic Control and Management	159
VIII.3.5	Execution: Rewriting Management System	160
VIII.4	Design time: workflow and initial configuration	160
VIII.5	Simulation and execution	163
VIII.5.1	Resource Allocation Control	165
VIII.5.2	Managing FESoS Workflow with strategies	167
VIII.5.3	Executing Functional Chains	169
VIII.6	Formal verification	171
VIII.6.1	Maude-based verification for management strategies	172
VIII.6.2	Model-checking SoSs proprieties	174

VIII.7 Conclusion	176
IX General Conclusion	177
IX.1 Conclusion	177
IX.2 Perspectives	180

List of Figures

II.1	<i>Principle of the MDA approach.</i>	12
II.2	<i>The four meta-layers of MDA architecture.</i>	13
II.3	<i>ISO/IEC/IEEE 42010:2011 Conceptual Model [38].</i>	16
II.4	<i>Inference Rules of a Rewrite Theory.</i>	20
III.1	<i>SoS and CSs Relationships in a Directed SoS.</i>	36
III.2	<i>SoS and CSs Relationships in an Acknowledged SoS.</i>	37
III.3	<i>SoS and CSs Relationships in a Collaborative SoS.</i>	38
III.4	<i>SoS and CSs Relationships in a Virtual SoS.</i>	39
III.5	<i>SoSSec MetaModel Block Diagram. [27].</i>	43
III.6	<i>Mission conceptual model [8].</i>	44
III.7	<i>SoSE core elements [47].</i>	46
III.8	<i>Simplified version of M2SoS[5].</i>	47
III.9	<i>Description of the design process [43].</i>	49
III.10	<i>ArchSoS definition process[79].</i>	50
III.11	<i>Missions and CSs in FMSoS. [80]</i>	51
III.12	<i>Meta-rules for adding publish/subscribe components[30].</i>	52
III.13	<i>BiGMTE architecture [29].</i>	53
IV.1	<i>Evolutionary process for dynamic SoSE.</i>	59
IV.2	<i>General overview of the approach.</i>	60
V.1	<i>Four-layer Meta-Modelling infrastructure of OMG.</i>	71
V.2	<i>General overview of MeMSoS.</i>	72
V.3	<i>An extended conceptual model of the SoS-AF.</i>	74
V.4	<i>Overview of the updated MeMSoS.</i>	75
V.5	<i>Multi-level approach-based UML Profile.</i>	77
V.6	<i>Multi-Viewpoints Framework for SoSs' Architectures.</i>	78
V.7	<i>Aircraft Emergency Response(AERSoS).</i>	79
V.8	<i>SoS development processes.</i>	84
V.9	<i>SoS-UML Profile's diagrams.</i>	86
V.10	<i>Overview of SoS-AF.</i>	87
V.11	<i>SoS-UML profile packages.</i>	89

V.12	<i>The structure of the SoS_Knowledge_Package.</i>	90
V.13	<i>Requirements diagram for AERSoS.</i>	91
V.14	<i>The structure of the CSs_Selection_Package.</i>	91
V.15	<i>Goals Diagram for AERSoS.</i>	92
V.16	<i>Capabilities diagram for AERSoS.</i>	93
V.17	<i>The structure of the Conceptual_Design_Package.</i>	93
V.18	<i>Domain Model for the PowerUnitsSoS.</i>	95
V.19	<i>The structure of the Architectural_Design_Package.</i>	95
V.20	<i>Constituent diagram for EvacuationCS.</i>	96
V.21	<i>The structure of the Interaction_Package.</i>	97
V.22	<i>Roles Interactions diagram for SafeLandingCS' Roles.</i>	97
V.23	<i>The structure of the Integration and deployment package.</i>	98
V.24	<i>Roles Interfaces diagram of PowerUnitsSoS.</i>	99
VI.1	<i>Resource Allocation Controller Meta-Model, RAC-MM.</i>	106
VI.2	<i>SoSs resource management modeling.</i>	108
VI.3	<i>Mission's transition system.</i>	116
VI.4	<i>Resource's transition system.</i>	119
VI.5	<i>Role's transition system.</i>	121
VI.6	<i>RAC's transition system.</i>	124
VII.1	<i>Management Strategies Meta-Model, MS-MM.</i>	129
VII.2	<i>MS-MM operational semantics in Maude Strategy Langage.</i>	131
VII.3	<i>The MAPE-K architecture.</i>	147
VII.4	<i>Overall structure of the modules.</i>	147
VIII.1	<i>The MAPE-K architecture for SoS Control and Management.</i>	157
VIII.2	<i>Overview of the Logical Architecture model of FESoS.</i>	162
VIII.3	<i>Overview of the Logical Architecture model of FESoS.</i>	164
VIII.4	<i>FESoS initial configuration.</i>	165
VIII.5	<i>Time-lapse during design of the three missions.</i>	170
VIII.6	<i>Time-lapse during Runtime of the three missions.</i>	170
VIII.7	<i>Runtime execution and control of the three missions.</i>	171
VIII.8	<i>Search results of Transporting mission.</i>	176

Chapter I

Preamble

Contents

I.1	Introduction	3
I.1.1	Problem Statement	4
I.1.2	Research Objectives and Contributions	5
I.2	Thesis Chapters	7
I.3	Publications	8

I.1 Introduction

System Engineering (SE) always accompanied by its methods and techniques allows the design of innovative systems that meet the public expectations and the various Stakeholders' needs. This continuous cycle justifies the increasing complexity of current systems and is evident across multiple domains. In this context, Systems-of-Systems (SoSs) have experienced significant attention from the computer science community due to their evolutionary progress and ability to integrate multiple Constituent Systems (CSs) from diverse domains, including Healthcare, Military Defense, Smart City, Smart Energy Grids, Emergency Management and Response. SoSs distinguish themselves by assembling various CSs from different subfields, presenting a robust and cohesive approach to constructing large-scale systems. The latter are characterized by their new functionalities that individual CSs cannot provide but emerge from their combination and interaction. Unlike monolithic systems, SoSs transcend their composite nature, dynamic control mechanisms, evolving environments, extensive networks of CSs, and various stakeholders. Given that CSs are inherently designed to operate independently, often utilizing heterogeneous technologies and catering to different platforms, SoSs do not follow a conventional top-down design approach. Instead, they are complex configurations of collaborating CSs, aiming to achieve common goals while maintaining their operational independence

and geographical dispersion.

Furthermore, the inherent properties of SoSs can be classified into two main categories: (1) component characteristics, such as operational and managerial independence, geographical distribution, autonomy, heterogeneity, and CS ownership, and (2) global characteristics, including emergent behavior, evolutionary development, and diversity of functionalities. These systems also navigate complexities related to heterogeneous technologies tailored for various platforms, dynamic control and reconfiguration, and continuous evolution. This means that the joint dynamics of SoSs and CSs should be studied together in an organizational and technical environment to realize the specified common goal. The latter is composed of several missions or sub-goals that are executed together, to offer one global mission for the SoS. These missions may need resources for execution such as humans, machines, services, etc.

These are the main reason that make the SoS Engineering (SoSE) process quite different from that of traditional SE, in which software systems can only be implemented from scratch. SoSE not only requires focus on system specification and verification but also requires additional consideration for the overall SoS mission, individual characteristics of components, issues solving and integration process. Therefore, the term SoSE will refer to the application of engineering principles to SoS development, and will consists of activities for managing the creation of SoS, including identifying and executing CSs missions based on stakeholder requirements analysis, specifying and designing conceptual architectures, integrating and assembling the selected CSs, and controlling the SoS as CSs execute various missions over time.

I.1.1 Problem Statement

The field of SoSs comes up against constraints during the SoSE process. The difficulty of modeling SoSs lies in the complexity resulting from the interaction, cooperation and collaboration of their heterogeneous CSs, which each have specific missions to accomplish, different roles to play, and they are not easily interoperable. Independent evolution and dynamic changes can cause these CSs to behave differently. These changes can affect their interactions and communications within the SoS and consequently, it can derail the overall mission of the SoS. Analyzing and specifying SoSs help to understand how they work, as well as to master its complexity well before its implementation. This offers considerable advantages to the designers of such systems. As well, SoS modeling can be seen as the separation of different functional and non-functional needs which are related to SoSs characteristics, SoSs quality/quantity attributes, management and control, SoSs architecture, design and Implementation.

In this new perspective, SoSs development does not follow the normal system development process. As SoSs' capabilities are based on the contributions of the individual CSs, their interdependencies make a document-centric development impractical as an exorbitant effort. The development process refers to activities that can guide an SoS' lifecycles from the system requirements level down to the software implementation level, and naturally, by coordinating the various processes for the development of a project. From a SoSEE perspective, design decisions made at the architectural level

have a direct impact on the fulfillment of functional and requirements of SoSs development. At this stage, the SoS' Stakeholders identify functional and non-functional characteristics through the use of their own theoretical backgrounds, notations and environments. In addition, the SoSs' architectures are still created without the support of a systematic process and traditional design approaches do not adequately support the creation of these types of systems due to composed nature, large-scale, decentralized control mechanism, evolving environments, and large number of stakeholders. To get a handle on this complexity, it is necessary to maintain consistency and coherence between the different viewpoints of different stakeholders as well as the ability to reconcile and include all their viewpoints before proceeding to the various process in the SoSE lifecycle.

On the other hand, SoSs can take four different types, Directed (with a central managed purpose and central ownership of all CSs), Virtual (lack of central purpose and central alignment), Collaborative (with voluntary interactions of CSs to achieve a goal), and Acknowledged (independent ownership of the CSs). Subsequently, the definition of SoSs is introduced based on the key consideration that SoSs are more than simply a set of connected CSs sharing data and offering missions, but, it defines the logical structure and behavior of qualitative/quantitative features involved in SoSs, whose CSs can have their operational/managerial independence but their emergent behavior is aligned with specific missions' execution.

In complex SoSs, two foundational elements dictate the efficacy and success of missions: temporal constraints and priorities, and resource management. The first element necessitates that the emergent behavior of SoSs not only aligns with mission objectives but also adheres to specific time frames and order of execution priorities. And the second one highlights the importance of optimally utilizing and coordinating the resources distributed among the different CSs.

Nowadays' SoSs, especially in critical and emergency applications, require the missions of various CSs to be accomplished in real-time and within specific amounts of time. This property significantly influences the launch, execution and completion of missions, where a violated constraint can result in a disaster. Therefore, the challenge here is that the time constraints applied on missions, and various time-related factors like duration and deadlines significantly impact the planning and execution of missions and can also be explicitly affected by the environmental features of different CSs.

At the same time, these features are strongly related to resource features. The resources that SoS requires at run-time, are sometimes limited, unlimited, renewable or not renewable. Therefore, resource management is another key element for execution, performance, and success. Resources can range widely, from computational power, physical units, data storage, and specialized hardware to human operators and communication channels. Therefore, understanding of these resources is vital for achieving functional missions, maintaining situational awareness, and ensuring system success.

I.1.2 Research Objectives and Contributions

The literature review shows various methods that have been modified from conventional systems development to better fit the needs of SoS design. However, each of these methods favors some aspects of SoS over others. More specifically, diverse approaches based on semi-formal methods have

been proposed to overcome some of major issue using suitable modeling languages like UML and SysML which provide intuitive and flexible frameworks for capturing the architectural design and facilitating communication among stakeholder. In contrast, formal methods such as ADL and BRS offer rigorous semantics, making it unsuitable for formal verification and analysis of system design. As proposed solution, we aim to combine the strength of both formalism's and tools to create robust, controllable, and executable models that can aid in the design, simulation, and verification of these complex SoSs.

For this end, four principal research objects (RO) are identified as follows:

- **RO1:** Overcome the lack of standards for the basic elements to describe the structure and behaviors of SoSs.
- **RO2:** Define a multi-viewpoints-based Architecture Framework to support SoSs' development.
- **RO3:** Define behaviors related to quantitative aspects such as time and resources of SoSs and their Control.
- **RO4:** Ensure strategic management of the execution of these different missions in SoSs.
- **RO5:** Ensure the autonomous executability of the desired behaviors within SoSs and verify their correctness.

In other words, the general objective of this thesis tends to propose a solution with semi-formal and formal foundations to (1) to develop a systematic architecture framework for SoSs and (2) to formally describe, analyze, and verify SoSs behavior effectively. This manuscript presents three parts of contributions as follows:

- **MDA-Based Approach:**

High abstract Assets: to provide Meta-Models providing the foundational basis necessary for specifying structural, behavioral, and functional aspects of SoSs. We focus on defining the logical structure and behavior of quantitative features involved in the SoS' definition and characteristics, which allows for describing all the SoS features at the same high level of abstraction. Hence, to create these generic models, we have to consider a set of concepts, aspects, and features, i.e. hierarchical composition of CSs, missions' organization, stakeholders, concerns, temporal variations, resource allocation, etc. These Meta-Models enable the designers to easily understand the SoSs' architectures, including reasoning about their features their structures and instantiating them by proposing a semantics mapping between their concepts and other formal constructs to deduce various executable models.

A multi-viewpoint Architecture Framework: called SoS-AF which is understandable and easily manipulated by different stakeholders. This framework aims to offer the SoSs' Stakeholders the tools to facilitate the task of developing a multi-viewpoint architecture that is managed by systematic SoSE processes and documented through the SoS-UML profile's models. Besides, this approach conforms to a widespread standard in the software architectures community "IEEE

42010” which was designed to standardize the definition of Systems and Software Architecture description. Specifically, SoS-AF inherits the definitions of the main elements from the MeMSoS Meta-Model which is a part of this standard, and extends them by the two elements “SoSE process” and “SoS-UML Profile”. It is based around the construction of multi-viewpoint SoSs’ architectures, through the definition of several viewpoints for a given SoS architecture.

- Formal approach to defining operational semantics of:

Resource Control: the approach addresses the challenges of resource consumption and production in SoSs. It consists of designing concepts like mission, role, and resource, specifying their properties, etc. Due to its expressivity, RT-Maude language can execute and validate all the relevant concepts defined in a specific Meta-Model designed to propose a centralized control of resources without losing information and features. Its executable semantics support designing the lifecycle of each entity, and offer more accurate modeling of SoSs whose dynamic behavior depends on missions’ quantitative priorities, resource allocation, and their data types.

Management Strategies the approach employs Maude Strategy Language to introduce self-management in SoSs, emphasizing the integration of dynamic strategies to manage both desired and undesired behaviors. More specifically, we address the complexities of workflow and functional chain management, mission execution, and conflict resolution. The proposed method aims to improve system functionality and resource efficiency in complex SoS environments by avoiding undesired behaviors and selecting optimal mission paths based on criteria such as arrival time, duration, and resource availability.

- Runtime, autonomic execution, and formal verification:

We specifically illustrate how the various modules can be organized into an autonomic operational framework through the application of the MAPE-K loop phases—Monitoring, Analysis, Planning, and Execution—, this loop adopts a systematic iteration between the design phase and the runtime execution, allowing for continuous refinement and enhancement of both the design and functional aspects of SoS. Moreover, the loop demonstrates the formal executable semantics of Maude to ensure that the behaviors adhere to the specifications, with verification mechanisms in place to validate the correctness of the operational logic.

I.2 Thesis Chapters

The chapters of the thesis are outlined as follows:

- **Chapter II.** introduces the essential prerequisites for analyzing, modeling, and implementing software systems, focusing on techniques from MDE, rewriting logic and the Maude language, detailing their theoretical foundations and practical uses.
- **Chapter III.** explores SoSs, their characteristics, their applications, and their challenges, etc. It also examines the state of research in SoS Engineering by evaluating semi-formal and formal

methodologies, focusing on the dynamic/behavioral challenges and the necessity for robust modeling techniques.

- **Chapter IV.** presents an overview of the proposed approach that integrates domain and application engineering within an MDA framework. The approach uses comprehensive meta-models and formal semantics through the Maude language to address SoSs' dynamic and behavioral complexities.
- **Chapter V.** introduces the SoS Architecture Framework (SoS-AF) which supports SoSs commonalities. The framework outlines stakeholder engagement, SoS concerns, and detailed SoSE processes, while also exploring SoS architecture viewpoints and model types through a developed UML profile.
- **Chapter VI.** explores the process of modeling and analyzing the variabilites of Time-Resource Aware SoSs using Maude language. Specifically, the chapter introduces the implementation of the Resource Allocation Controller (RAC) which provides an autonomous coordination for resource allocation and management.
- **Chapter VII.** introduces an approach for managing both desired and unwanted behaviors in missioned SoSs taking into account the temporal constraints and resources. The method involves specifying self-management strategies, encoding behaviors, and applying rewriting rules to effectively govern mission execution and resource utilization.
- **Chapter VIII.** discusses the execution, simulation and verification of SoSs, using the French Emergency SoS (FESoS) as a case study. The aim is to describe the proposed contributions and validate the behavior of the SoS using the model-checking verification techniques offered by the Maude language.

I.3 Publications

Peer-reviewed scientific journal

- Charaf Eddine Dridi, Zakaria Benzadri, Faiza Belala. (2023). A Unified Architecture Framework Supporting SoS's Development: Case of the Aircraft Emergency Response System-of-Systems. *International Journal of Organizational and Collective Intelligence (IJOICI)*, 13(1), 1-30.

Peer-reviewed international workshop

- Charaf Eddine Dridi, Nabil Hameurlain, Faiza Belala. (2022). A Maude-Based Rewriting Approach to Model and Control System-of-Systems' Resources Allocation. In *International Conference on Model and Data Engineering* (pp. 207-221). Cham: Springer Nature Switzerland.

Peer-reviewed international Conferences

- Charaf Eddine Dridi, Nabil Hameurlain, Faiza Belala. (2023). A Maude-Based Formal Approach to Control and Analyze Time-Resource Aware Missioned Systems-of-Systems. In 2023 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE) (pp. 1-6). IEEE.
- Charaf Eddine Dridi, Zakaria Benzadri, Faiza Belala. (2022). Towards a Multi-Viewpoints Approach for the SoS Engineering. In 2022 International Conference on Advanced Aspects of Software Engineering (ICAASE) (pp. 1-6). IEEE.
- Charaf Eddine Dridi, Zakaria Benzadri, Faiza Belala. (2020). System of Systems Modelling: Recent work Review and a Path Forward. In 2020 International Conference on Advanced Aspects of Software Engineering (ICAASE) (pp. 1-8). IEEE.
- Charaf Eddine Dridi, Zakaria Benzadri, Faiza Belala. (2020). System of Systems Engineering: Meta-Modelling Perspective. In 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE) (pp. 000135-000144). IEEE.

Chapter II

Basics and Prerequisites

Contents

II.1	Introduction	10
II.2	Model-Driven Engineering	11
II.2.1	Model-Driven Architecture	11
II.2.2	IEEE-ISO 42010 Standard	15
II.3	Rewriting Logic and Maude Language	16
II.3.1	Rewriting theories	17
II.3.2	Maude Language: modules and extensions	21
II.3.3	Formal Analysis and Execution	26
II.4	Conclusion	29

II.1 Introduction

Computer systems have automatically increased in complexity with the evolution of systems and information technologies. Their engineering, design, and analysis have become increasingly difficult to master and costly to ensure. To reduce this complexity, and to offer a high abstraction that makes it possible to describe, represent, and unambiguously analyze these systems, one solution is to use Modeling and analysis to design complex systems. The former allows describing the system, while the second makes it possible to evaluate the system properties. Moreover, the field of Software and/or Systems Engineering have treated modeling and analysis techniques as two distinct areas. Model-Driven Engineering (MDE) for instance uses domain-specific models as primary assets to develop a system. Conversely, formal methods are mathematically rigorous techniques for the specification, development, analysis, and verification of diverse properties of these software/systems. i.e. MDE principles often described by a Model-Driven Architecture (MDA). However, in order to have the

power of expressiveness which gives it the possibility of describing complex systems, it must be reinforced by mathematical foundations that allow better modeling and analysis of systems.

In this chapter, we introduce the semiformal and formal tools adopted in this manuscript, they provide fundamental concepts to readers who are not familiar with these models, and facilitate the understanding of the contents and contributions of this thesis. These models allow for a precise description of properties related to the operation of the designed systems. Formal methods particularly provide a means to verify and guarantee the satisfaction of these properties systematically.

II.2 Model-Driven Engineering

MDE is increasingly recognized as a vital approach in complex software projects across diverse fields such as embedded systems, automatic and telecommunications. This approach places a strong emphasis on modeling as a core concept, offering a means to abstract and simplify system design and comprehension. Moreover, MDE is not just about employing Unified Modeling Language (UML) models [28], [76] in the initial stages of software development, but rather it extends to leveraging these models beyond simple initial specifications, integrating them throughout the analysis and implementation phases.

II.2.1 Model-Driven Architecture

In MDE, a model serves as an abstraction of some aspect of a system, potentially one that does not yet exist, and is crafted to fulfill specific purposes, such as providing a human-understandable description or enabling comprehensible analysis [49], [81]. The effectiveness of these models depends heavily on the expressiveness of the languages used to create and interpret them. Therefore, we distinguish between General-Purpose Languages (GPLs), which offer broad expressiveness for modeling various system aspects, and Domain-Specific Modeling Languages (DSMLs), which provide tailored abstractions for specific areas of expertise [49], [78]. DSMLs are characterized by both syntax and semantics. The syntax includes both an abstract syntax, defining the concepts and their relationships, and a concrete syntax, offering a human-readable format for these concepts. The semantics, on the other hand, comprise a semantic domain and a mapping from the abstract syntax concepts to this domain.

In this section, we explore the key principles of MDA and the various aspects relevant to the definition and application of Models, particularly in the context of software development and design.

II.2.1.1 MDA Different models

Launched by the OMG in 2000 [70], [81], MDA is a key aspect of MDE, and utilizes abstract UML models to streamline software development. It primarily involves three model types as shown in Figure II.1 [7]:

- **Computation Independent Models (CIMs)**: These models outline the application's requirements at a high level, focusing on what the application should do without specifying how

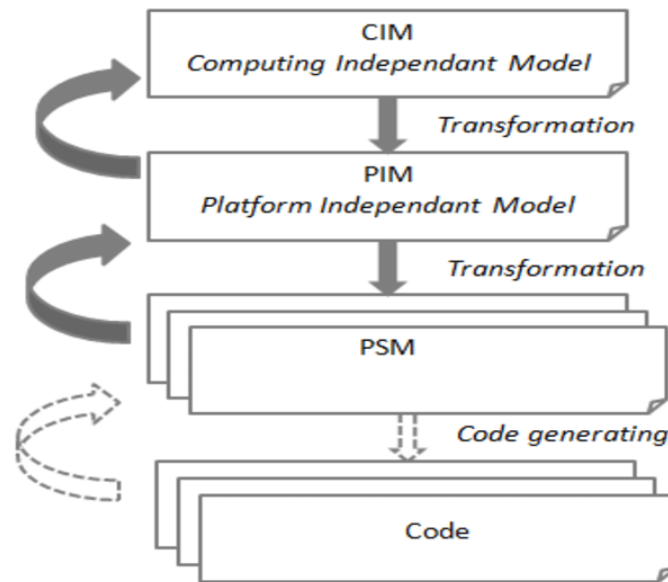


Figure II.1: *Principle of the MDA approach.*

it should be done.

- **Platform Independent Models (PIMs):** PIMs are abstract, they focus on the design and analysis of the application, remaining abstract to specific technologies and act as intermediaries, translating CIMs' requirements into detailed designs.
- **Platform Specific Models (PSMs):** Tied closely to specific technologies, PSMs provide detailed implementation guidelines, adapting PIMs to specific platforms like .NET or Java EE. For example, an Enterprise JavaBeans (EJB) PSM would include specific constructs such as entity beans and session beans.

MDA places these models, particularly the PSMs, as significant means of facilitating efficient code generation, treating application code as a structured set of instructions derived from these models.

II.2.1.2 Four-layers architecture

The concept of Meta-Modeling in MDE is structured into a four-layer architecture, as detailed in [7], [45], [70] and further elaborated by the OMG [81]. This structure, known as the "Four-layers Meta-Modeling pyramid", arranges the concepts of models, Meta-Models, and meta-metamodels across distinct levels, with each level building upon the one below it as illustrated in Figure II.2. The layers are as follows:

- **M3 Layer:** This is the topmost layer containing meta-metamodels. These are described in terms of themselves due to their reflective property. It represents the language used for defining Meta-Models.

- **M2 Layer:** This layer contains the Meta-models to define the models like UML elements, or definitions of Domain-specific Languages or Generic-purpose Languages. they define the different structures and semantics of the models in M1 of the next layer.
- **M1 Layer:** it contains the models. i.e. this level includes, for example, a UML class diagram. they represent abstractions of systems and conform to the Meta-Models defined in M2.
- **M0 Layer:** The lowest level consists of real-world instances or objects, such as a specific instance of a sensor in an application.

It's possible to have multiple Meta-Models for a domain, but a model typically adheres to one Meta-Model. The four-layer architecture is significant, especially in the context of model transformations, as it clearly defines the hierarchical relationship between different modeling concepts.

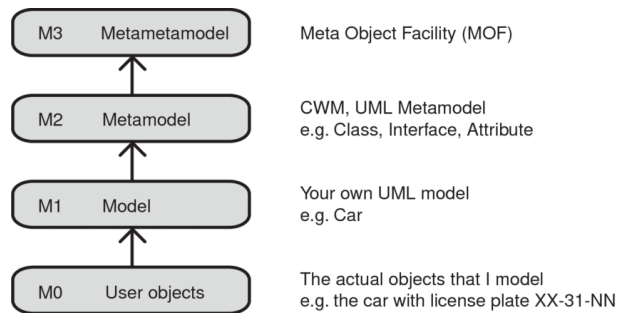


Figure II.2: *The four meta-layers of MDA architecture.*

II.2.1.3 Modeling languages

Modeling languages are broadly categorized into Domain-Specific Modeling Languages (DSMLs) and General-Purpose Modeling Languages (GPMLs), such as UML. While DSMLs are tailored to the specific requirements of a particular domain, GPMLs like UML offer flexibility across various domains and design challenges.

- Using UML in development process

Using UML in the development process often results in a longer timeframe compared to DSMLs, due to its foundation on more abstract, less domain-specific models. However, UML models tend to be more universally accessible, unlike DSMLs which typically require in-depth domain expertise. UML is a versatile language for software systems modeling, which supports the depiction of diverse system views, drawing from object-oriented language concepts. Its broad applicability allows for extensions to suit specific domains. These extensions are realized through UML's Meta-Model mechanisms, resulting in the creation of UML profiles [56], [82]. The latter is an assembly of extensions that includes specific terminology, notation, and syntax tailored for domain-specific elements. The OMG outlines two primary approaches for creating domain-specific languages:

1. **Creating a new language:** This approach involves developing an entirely new language, for which OMG advocates the use of the Meta-Object Facility (MOF) language. MOF has been pivotal in shaping the syntax and semantics of UML for the creation of new object-oriented languages. This method ensures a direct alignment between the language and the application domain concepts but may encounter compatibility challenges with existing object-oriented languages and tools that predominantly rely on UML.
2. **Extending UML notation:** The alternative approach involves augmenting UML with new elements that specialize its pre-existing components, such as Class, Association, and Package. The extension mechanisms employed include stereotypes, primitive values, and constraints, grouped into a domain-specific UML Profile. This approach benefits from UML's established modeling elements to represent basic objects and concepts, with stereotypes detailing domain-specific aspects of the system. Constraints within the profile provide a means to express and validate specific semantics in the application domain models.

Compared to creating a new language, the utilization of UML profiles tends to be more facilitative, aligning seamlessly with standard UML and integrating easily into existing object-oriented development frameworks.

- Using UML profiles for Domain-Specific Customization

UML Profiles play a pivotal role in MDA by enabling the specialization of Meta-Models for specific application domains. These profiles adapt the standard UML Meta-Model to accommodate different platforms (like J2EE or .NET) or domains (such as real-time or business process modeling), as per the OMG MOF standards [56], [82]. Therefore, a UML profile contains various elements that extend the standard UML Meta-Model without altering its core structure, often referred to as 'Lightweight extensions'. These include:

1. **Marked Values, Stereotypes, and Constraints:** These elements adapt the semantics of the UML Meta-Model. Stereotypes, for instance, are defined as extensions of a UML metaclass and can have specific properties or operations. Marked values act as attributes of a metaclass, allowing the arbitrary attachment of information to an instance.
2. **Scope and Structure:** A profile is essentially a stereotyped package that may import external sources. It is a collection of stereotypes specific to a domain, thereby specializing in one or more standard Meta-Models, referred to as reference Meta-Models for that domain.
3. **Technical Aspects:** Technically, a profile can include additional standard elements, a subset of existing Meta-Model elements, specific semantic concepts in natural language, and custom graphical notations.
4. **Constraints and Rules:** Constraints can be defined at the level of a specific metaclass or stereotype, further specializing the semantics of reference Meta-Model elements used in the profile. These constraints are usually expressed in the Object Constraint Language

(OCL) but can also be informally specified in natural language. They apply to modeling elements to specify the use of their instances and are often associated with stereotypes.

The primary function of a UML profile is to define the semantics of usage rules and constraints of the UML Meta-Model. For example, a profile for EJB modeling can be depicted as a class diagram showing stereotypes and marked value definitions related to EJB and their connection to UML Meta-Model elements. This form of specialization allows the profile to be dedicated to a specific domain within the reference Meta-Models. Furthermore, UML profiles can define transformation rules to express how a model can be transformed for specific modeling or implementation goals, validation rules to ensure the model possesses the correct properties of the profile's domain, and presentation rules to determine which modeling elements appear in specific diagram types.

II.2.2 IEEE-ISO 42010 Standard

The ISO/IEC/IEEE 42010 standard provides a structured approach to software architecture description, recognizing the complexity of systems and the need for a comprehensive framework to address this. The standard's conceptual model defines the relationships between various architecture description elements, highlighting the importance of understanding how systems interact with their environment and stakeholders' concerns [38]. The recent revision of the ISO/IEC/IEEE 42010 standard brings clear definitions to enhance the reuse and interoperability of technical architectures through three key mechanisms[32], [77]:

- **Architecture Reflections:** Common methods for expressing and resolving a set of known architectural concerns, which can be reused across projects.
- **Architecture Description Languages (ADLs):** Specialized languages designed to articulate specific system concerns through one or more modeling resources.
- **Architecture Frameworks:** a prefabricated structure that stakeholders can use to organize an architectural description into complementary views.

These architectural viewpoints not only organize the structure and the architecture description by specifying different views of the total architecture, each of them is created using specific conventions, notations, and modeling tools, but also by representing a form of reusable architectural knowledge, utilized to address architecture description challenges.

Key components of this model include:

- **Stakeholders and Concerns:** The standard highlights the identification of stakeholders and their concerns, ensuring that the architecture adequately addresses various interests.
- **Architecture Viewpoints and Views:** This entity highlights the use of architecture Viewpoints VPs to frame the different specific stakeholder concerns and establish conventions for creating views. Moreover, Views represent the system's architecture from the perspective of

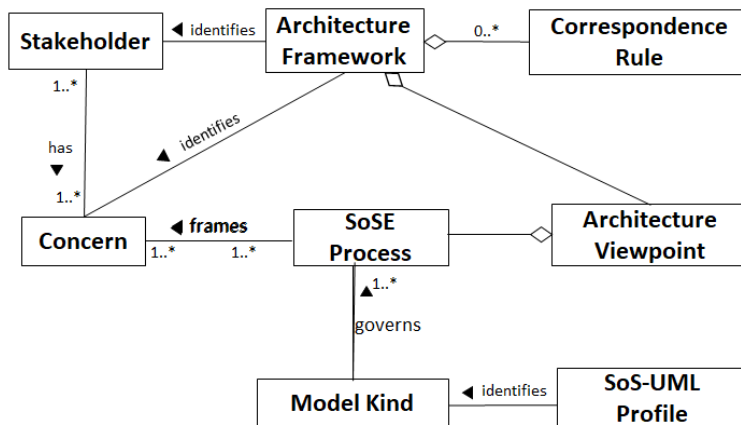


Figure II.3: ISO/IEC/IEEE 42010:2011 Conceptual Model [38].

these stakeholder’s concerns, employing various modeling languages, notations, and design rules specified by the VPs.

- **Model Kinds in Viewpoints:** Different viewpoints may include various model kinds, each architecture view comprises multiple models which adhere to the conventions defined by their respective model kinds, aiding in the construction and interpretation of the architectural framework.
- **Correspondences and Rules:** The architecture description includes correspondence rules. These elements define and enforce relationships between architecture description elements, ensuring consistency across models, views, and other components of the architecture.

The standard ISO/IEC/IEEE 42010 is designed for reusability with various system and software engineering standards, enhancing their capabilities in architectural aspects. the standard is not only applicable across the entire life cycle of systems, providing comprehensive guidance for the specification, documentation, and ongoing maintenance of architecture descriptions, but it is also adaptable for the incorporation of additional elements as much as needed.

II.3 Rewriting Logic and Maude Language

Rewriting logic is a versatile paradigm that was introduced by José Meseguer [13], [57] as a result of his extensive work on general logic, specifically to describe concurrent systems. This computational logic serves as an expressive semantic framework for various computational tasks involving concurrency, parallelism, communication, and interaction. In this logic, rewriting a term means replacing it with an equivalent term, adhering to the laws of term algebra [12], [58]. This allows for the calculation of a rewritability relationship between algebraic terms. The logic permits syntactic transformations and offers a framework for logical inferences, making the rewriting semantics highly adaptable and fruitful [57]. By examining the basic concepts, semantics, and multiple dimensions of rewriting logic, alongside the practical utility of languages like Maude, this section aims to offer

a well-rounded understanding of this formal method and its tools, starting with the basic concepts of rewriting logic, followed by introductions to the Maude system, Maude Strategy Language and RT-Maude. For more details, see [53] and [11].

II.3.1 Rewriting theories

The utility of rewriting logic has been widely substantiated, especially in its capability to provide robust reasoning about the behavior of concurrent systems.

II.3.1.1 Rewriting logic

Rewriting logic finds its application in multiple computational logic. For example, in equational logic, calculations are interpreted as equations between terms, whereas, in constraint satisfaction logic, a rewriting rule can be seen either as a syntactic transformation or as a logical inference of a new formula [12], [44]. In the context of practical applications, Maude represents a formal language that is particularly effective in the verification of concurrent systems. Developed as an algebraic specification and declarative programming language, Maude is simple, yet expressive and efficient.

- Theories of Rewriting Logic

In rewriting logic, we find equational theories dedicated to specifying the static aspects of systems, rewriting theories that describe the possible evolutions of the states of a concurrent system, and also real-time rewriting theories for systems with temporal constraints. Adding a set of rewriting rules extends equational theories that specify the possible transitions between states in concurrent and distributed systems. Using them enables the study of the dynamics of these systems and allow for the formal specification and analysis of system behavior, the rewriting rules are employed as rewriting axioms, taking the form $t \rightarrow t'$, where t and t' are algebraic terms. Two interpretations can be associated with rewriting rules [53], [57]:

1. Computational: In this view, the rewriting rule $t \rightarrow t'$ is interpreted as a local transition in the system. That is, t and t' model parts of the system's distributed state, and the rule describes a change from a part of the global state instantiated according to t to the state corresponding to the model t' .
2. Logical: In this perspective, the rewriting rule $t \rightarrow t'$ is seen as an inference rule. That is, a formula of the form t' can be inferred from a formula of the form t .

According to different points of view for understanding and applying rewriting theory, these interpretations could be interpreted as a computer model or as a logical framework. In the following subsections, we present the different formal notions necessary for understanding the logic of rewriting

Definition II.1 (Rewriting Theory). *Formally, a rewriting theory is defined by a quadruplet $\mathfrak{R} = (\Sigma, E, \phi, R)$*

[64], where :

- (Σ, E) is the membership equational theory modulo which the rewriting is operated;
- ϕ is a function that assigns to each function symbol $X = \left\{ f : x_1 : k_1, \dots, x_n : k_n \right\}$ of Σ a set $\phi(f) \subseteq \left\{ 1, \dots, n \right\}$ of frozen arguments;
- R is a set (universally quantified) of rewriting rules of the form $(\forall X)t \rightarrow t'$, with $X = \left\{ x_1 : k_1, \dots, x_n : k_n \right\}$ a set of typed variables and $t, t' \in T_{\Sigma}(X)_K$.

Given a rewriting theory, the set of formulas implied by this theory is defined by the deduction rules of rewriting logic. These rules formalize the notion of rewriting terms by defining how theorem proofs are formed. The static structure of a system is described by the algebraic specification (Σ, E) while the dynamic structure is described by the rewriting rules R .

- Deduction in Rewriting Logic

Computation in a concurrent system is a sequence of transitions (rewriting rules) executed from a given initial state. It corresponds to a proof or deduction in rewriting logic. This deduction is inherently concurrent and allows for accurate reasoning about the system's evolution from one state to another [12], [57].

Definition II.2 (Deduction Principle). *Given a rewriting theory $\mathfrak{R} = (\Sigma, E, \phi, R)$, we say that the sequence $[t] \rightarrow [t']$, is provable in \mathfrak{R} and we write $\mathfrak{R} : [T] \mapsto [T']$ if and only if $[t] \rightarrow [t']$ is obtained by a finite application of the following deduction rules:*

- Reflexivity: For each term $[T] \in T_{\Sigma, E}(X)$,

$$\overline{[T] \mapsto [T']}$$

where $T_{\Sigma, E}(X)$ is the set of Σ -terms with variables built on the signature Σ and the equations E ;

- Congruence: for each function $f \in \Sigma_n, n \in N$,

$$\frac{[t_1] \rightarrow [t'_1] \dots [t_n] \rightarrow [t'_n]}{[f(t_1, \dots, t_n)] \rightarrow [f'(t'_1, \dots, t'_n)]}$$

- The congruence: for each rule $[t(x_1, \dots, x_n)] \rightarrow [t'(x_1, \dots, x_n)]$ in R :

$$\frac{[w_1] \rightarrow [w'_1] \dots [w_n] \rightarrow [w'_n]}{[t(\bar{w}/\bar{x})] \rightarrow [t'(\bar{w}'/\bar{x}')]}$$

Knowing that $t(\bar{w}/\bar{x})$ describes how to perform a substitution operation; Essentially, it means that you replace each variable x_i in the term t with a corresponding value w_i , where \bar{x} is the vector x_1, \dots, x_n of variables;

- Transitivity:

$$\frac{[t_1] \rightarrow [t_2][t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Generally, deduction in rewrite logic is an iteration of the following steps:

1. The replacement rule identifies all rewrite rules whose left-hand side matches a sub-term of the current global state. Since rewriting logic is a logic of change, the reflexivity rule, when applied to unidentified sub-terms, transforms them into themselves.
2. The rewrite rules, identified by the replacement rule along with the reflexivity rule, are executed concurrently and independently of each other. The congruence rule combines the effects, or right-hand members, of these rules to construct the new global term.
3. Steps 1. and 2. are repeated until no more rules are applicable.
4. Finally, the transitivity rule constructs the sequence of rewrites made from the initial term to the final term. The sequence thus constructed corresponds to a possible computation in the concurrent system.

The rewrite sequences for a system describe all the concurrent transitions of the system as axiomatized by \mathfrak{R} . Logically, they describe all the possible deductions of one formula from another formula, also axiomatized by \mathfrak{R} . The previously given inference rules can be schematized as shown in Figure II.4. In this context, the rewrite sequences serve as a comprehensive guide to understanding how the system evolves over time, capturing all possible state changes logically. This provides a powerful way to analyze and understand the behavior of concurrent systems formally.

II.3.1.2 Real-time Rewriting Theory

The Real-time Rewriting Theory is a formal framework for modeling and analyzing real-time and hybrid systems. In this context, special rewriting rules called *tick rules* are used to represent the passage of time or the duration of an event in the system. These *tick rules* work in parallel with traditional rewriting rules that describe the instantaneous state changes in the system. The equational theory underlying such a rewriting theory must also include a representation or axiomatization of the concept of time. Additionally, a specific operator is often defined to encapsulate the global state of the system. The purpose of this operator is to ensure a uniform flow of time across all components of the system, in accordance with the execution of the tick rules[10]. In this framework, one can accurately model how time and the states of the system evolve, which is crucial for understanding and verifying the properties of real-time and hybrid systems:

$$l : \{t\} \xrightarrow{\tau_l} \{t'\} \text{ if } Cond$$

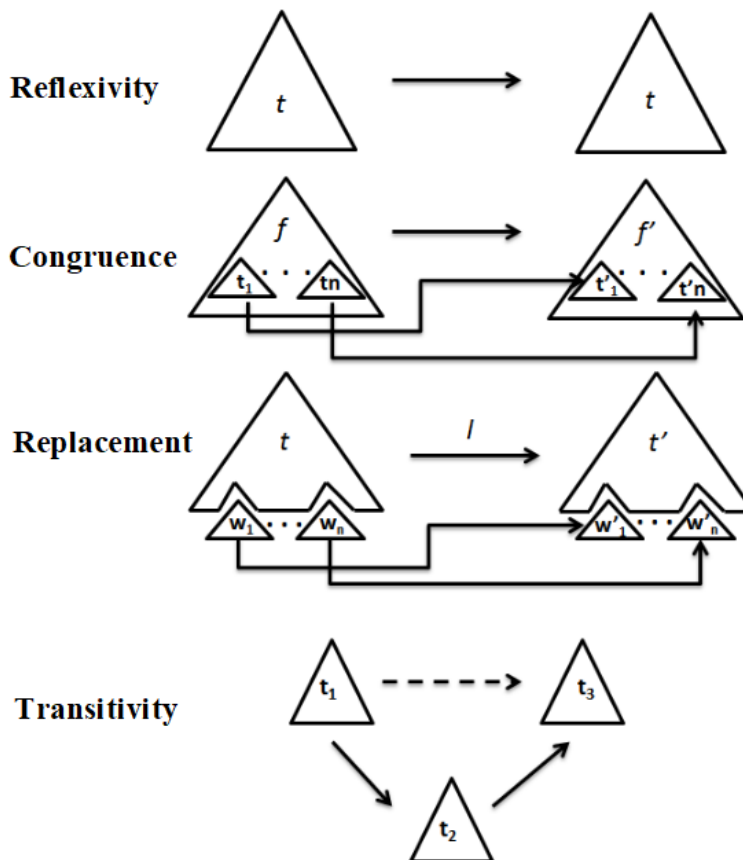


Figure II.4: Inference Rules of a Rewrite Theory.

Where, l refers to the label of the rewriting rule, and the term τ_l , of the sort Time, represents the duration of the rewriting step. This labeling and timing information allows for a precise understanding of the time-related aspects of system behavior, adding another layer of detail to the model. Therefore, Real-time rewrite theories serve as a natural model for the behavior of real-time systems. The expressiveness of these theories provides the models of these systems with formal semantics, as well as advanced forms of communication and data types [62]–[64].

Definition II.3 (Real-Time Rewrite Theory). : A real-time rewrite theory is a quadruplet $\mathfrak{R} = (\Sigma, E \cup A, \phi, R)$, where:

1. $(\Sigma, E \cup A)$ is the underlying equational theory that describes the states of the system: Σ is its signature, E is the set of equations, and A is the set of equational attributes. The theory $(\Sigma, E \cup A)$ includes another underlying equational theory $(\Sigma_{Time}, E_{Time})$ that formalizes the concept of "time".
2. ϕ is a function that associates to each operation the "frozen arguments" (arguments that cannot be rewritten).
3. R is a set of rewrite rules, composed of two types of rules: instantaneous rules I_R of the form $l : t \rightarrow t' \text{ if } Cond$, and time advancement rules T_R which are of the form $l : t \xrightarrow{\tau} t' \text{ if } Cond$ (τ

is the duration of time advancement).

II.3.2 Maude Language: modules and extensions

Maude is a high-performance, declarative language that faithfully implements all theoretical concepts of rewriting logic. Developed by Jose Meseguar and his team at SRI International's computer science lab, it excels in both executable specification and declarative programming for a wide range of applications, including equational and rewriting logic[11]. In Maude, a program essentially represents a rewriting theory consisting of a signature and a set of rewriting rules. Computations in this language are essentially deductions in rewriting logic based on the axioms specified in these theories or programs. The language is not only expressive but also efficient, capable of millions of rewrites per second, making it competitive with high-level languages in terms of efficiency. It also supports network programming through the use of sockets. In terms of data types, they are algebraically defined by equations, while the dynamic behavior of the system is defined by rewriting rules. Maude also supports object-oriented programming, including features like multiple inheritance and asynchronous message passing.

II.3.2.1 Maude Languages modules

To specify a system, Maude employs three types of modules: functional modules for implementing equational theories, system modules for implementing rewriting theories that define a system's dynamic behavior, and object-oriented modules for implementing object-oriented rewriting theories. These object-oriented modules can essentially be reduced to system modules. Overall, Maude system provides a robust, efficient platform for algebraic specification and modeling of concurrent systems, making it one of the leading languages in its domain.

1. Functional modules

In Maude, a functional module is specified using the keywords:

```
fmod MODULNAME is
  <Module Body>
endfm
```

Where the module body defines a theory $(\Sigma, E \cup A, \phi)$ in membership equational logic, the signature Σ includes sorts (indicated by the keyword `sort`), subsorts (specified by the keyword `subsort`), and operators (introduced with the keyword `op`). Operator syntax is user-defined, indicating the position of arguments with the symbol $(_)$. Certain arguments can be marked as `frozen` using the keyword `frozen(PositionArgument)`.

The E is a set of equations and membership tests and A is a set of equational axioms introduced as attributes of some operators in σ , as illustrated in Table. II.1 they include associativity (keyword

`assoc`), commutativity (keyword `comm`), and identity (keyword `id`). Equations are expressed with the keyword `eq` or `ceq` for conditional equations, and membership tests are introduced with the keywords `mb` or `cmb` for conditional tests. Variables can be declared in the modules using the keywords `var` or `vars`, or directly introduced in the equations and membership tests in the form of an expression `var: sort`.

Table II.1: Description of Sets E and A and Their Associated Keywords.

Symbol	Description
E	Set of equations and membership tests (conditional included)
<i>eq</i>	Specifies an equation
<i>ceq</i>	Specifies a conditional equation
<i>mb</i>	Specifies a membership test
<i>cmb</i>	Specifies a conditional membership test
A	Set of equational axioms as operator attributes
<i>assoc</i>	Specifies associativity of operators
<i>comm</i>	Specifies commutativity of operators
<i>id</i>	Specifies identity element for operators
<i>var</i>	Declares a single variable
<i>vars</i>	Declares multiple variables
<i>var: sort</i>	Introduces a variable in the form of an expression

In summary, functional modules in Maude define a theory in membership equational logic, specifying data types, operations, variables, equations, and axioms. The module body consists of declarations such as sorts, operators, variables, and comments. These modules form the basis for simplification rules, enabling the evaluation and reduction of each expression to its canonical form, which is unique and independent of the equation application order.

2. System modules

In Maude, system modules are used to define the dynamic behavior of concurrent systems by extending functional modules with a set of rewrite rules. These modules are introduced using the keywords:

```

mod MODULNAME is
    <Module Body>
endm

```

The module body outlines a rewrite theory $\mathfrak{R} = (\Sigma, E \cup A, \phi, R)$, where $(\Sigma, E \cup A, \phi)$ is the underlying equational theory. Rewrite rules R are introduced with the keywords `r1` or `cr1`. They

are specified in Maude with the syntax `cr1 [l] : t => t' if cond`. If the rule is unconditional, the keyword `cr1` is replaced with `r1`, and the `if cond` clause is omitted.

This system module specifies a rewrite theory that includes sorts, operations, variables, equations, membership axioms (both conditional and unconditional), as well as conditional and unconditional rewrite rules. A rewrite rule is activated when its left-hand side matches a portion of the system's global state and any specified conditions are met.

In summary, system modules in Maude extend functional modules to include rewrite rules for expressing concurrency in systems. These rewrite rules are specified using the keywords `r1` for unconditional rules and `cr1` for conditional rules. They become active based on matching conditions in the system's global state.

3. Object-oriented modules

Full Maude extends Core Maude to provide powerful and extensible algebraic modules that support object-oriented programming. Object-oriented modules are introduced with the keywords:

```
(omod MODULNAME is
  <Module Body>
Endom)
```

These modules encompass a rewrite theory $\mathfrak{R} = (\Sigma, E \cup A, \phi, R)$ and support the specification and manipulation of objects, messages, classes, and inheritance.

In this object-oriented system, a concurrent OO-system is modeled as a multiset of juxtaposed objects and messages. Interactions between objects are governed by rewrite rules. An object is represented as `< O : C | a1 : v1, ..., an : vn >`, where `O` is the name of the object, an instance of class `C`, and `ai` are the names of the object's **attribute identifiers**, and the `vi`'s are the corresponding values, for $i = 1 \dots n$.

Class declarations follow the syntax `class < C | a1 : s1, ..., an : sn`, where `C` is the class name and `si` are sorts for attribute `ai`. Subclasses can also be declared, making use of inheritance. Messages are declared using the keyword `msg`.

In Full Maude, the general form of a rewrite rule in object-oriented syntax is:

```
cr1 [r] : M1 ... Mn < O1 : F1 | a1 > ... < Om : Fm | am >
  =>
  < O1i1 : F'i1 | a'i1 > ... < Oik : F'ik | a'ik > M1' ... Mp'
  if Cond .
```

Here `r` is the rule label, $M_s, s \in 1..n$ and $M_u, u \in 1..p$ are messages, $O_i, i \in 1..m$ and $O_{il}, l \in 1..k$ are objects, and `Cond` is the condition for the rule. If the rule is unconditional, `cr1` is replaced by `r1`, and the `if Cond` clause is omitted.

Full Maude enhances Core Maude by offering rich support for OO programming, i.e. it enables the representation and manipulation of objects, messages, and classes and the use of inheritance. It also provides a more user-friendly syntax for defining rewrite rules in this context.

Moreover, Full Maude provides a predefined module named `CONFIGURATION` that declares sorts representing essential concepts like objects, classes, messages, and configurations. It can also use system and functional modules. Rewrite rules for objects utilize the associative and commutative nature of multisets, making the configuration more flexible, and allowing objects and messages to interact in a concurrent transaction.

4. Predefined modules

The predefined modules in Maude are stored in a specific library and can be imported by other user-defined modules. These modules are introduced in Maude's source files `prelude.maude` and `model-checker.maude`. For example, the modules `BOOL`, `STRING`, and `NAT` are predefined. They declare sorts and operations for manipulating boolean values, strings, and natural numbers, respectively. The file `model-checker.maude` contains predefined modules that interpret the necessary tools for using Maude's LTL (Linear Temporal Logic) Model Checker. This tool allows for model-checking tasks, where one can verify the properties of a system modeled in Maude against temporal logic formulas.

II.3.2.2 Maude Language extensions

Maude comes equipped with other notations and extensions that allow for different expressions of the structure and dynamics of specified systems. It mainly relies on the object-oriented paradigm [64], [74] to define the two following extensions:

1. Maude Strategy Language

This language defines an extension dedicated to strategies [9] [55] [54] aimed at the modular separation between rewriting rules and their controls (execution) using strategies. In the context of our work, we use this Maude extension to define the behaviors of complex and distributed systems; this language allows for a mechanism to guide the rewriting of their behaviors.

The behavior of a system in Maude depends solely on its rewrite rules and their applications. However, in Maude's strategy language, this is no longer the case. Thanks to the separation provided by this language, we can have multiple strategy modules for the same Maude system module, where each module offers a definition of a different execution path, thus offering more flexibility in various executed scenarios. A behavior will depend on the strategies that control the rewrite rules. We define it below:

Definition II.4 (Maude Strategy Module). : *A strategy module declared by the Maude strategy language can be in the form: $\mathfrak{R} = (\Sigma, E \cup A, S(R, SM))$, where:*

where:

Table II.2: Key Elements in a Strategy Module

Instruction	Description
<code>smod</code>	Declaration of a Maude strategy module
<code>Module1</code>	The module whose rewrites we want to control
<code>ModuleStrategieX (Optional)</code>	Importing another Maude strategy module to use its strategies in this module
<code>strat</code>	Declare a new strategy
<code>S1</code>	The name (identifier) of the strategy
<code>@ MX</code>	The declared strategy will be applied to terms of a kind <code>MX</code>
<code>sd</code>	Definition of a strategy (<code>sd</code> : strategy definition) followed by the identifier and expression of this strategy
<code>Expression</code>	A term to describe the strategy

- Σ, E , and A represent the equational theory of the evolving system;
- S is a semantics describing the behavior of a system, built from a module R containing rewrite rules, and a set of strategies SM that will guide the rewriting of these rules using strategies.

The syntax of the strategy modules is as follows:

```
smod <module name> is
    <declarations and expressions>
endsd
```

Where `smod` and `endsd` are the keywords indicating the beginning and end of the strategy module, while the declarations and expressions represent variables, imports from other modules, and strategy declarations [75]. Generally, a Maude strategy module has the generic syntax presented in Table II.2.

To execute a strategy in Maude's strategy language, the `srew` command is used with the following syntax: `srew Term using Expression`, where `Term` is the initial state (or the state at a given moment) of the system, `Expression` is the strategy to apply to this term.

2. Real-Time Maude

The Real Time Maude system (RT-Maude) [62], [63] is a language and a tool allowing the formal specification and analysis of time-sensitive and hybrid systems. Its specification formalism, based on rewrite logic, is particularly suitable for describing real-time object-oriented systems. The tool is implemented in Maude as an extension of Full Maude. The RT-Maude specification language allows modeling real-time systems in terms of real-time rewrite theories. It allows implementing these rewrite theories via timed modules or timed object-oriented

modules whose body is encapsulated between the keywords `tmod` and `endtm` (specifying the beginning and the end of a timed module):

```
tmod <module name> is
    <declarations and expressions>
endtm
```

Note that each timed module must essentially include:

- An abstraction of the time domain (of sort `Time`). The time domain managed by timed modules can be discrete or dense. For example, to handle discrete time in a timed module, the user must import the predefined `NAT-TIME-DOMAIN-WITH-INF` module from RT-Maude, which defines the time domain as natural numbers by adding the constant `INF` of supersort `TimeInf`.
- The `GlobalSystem` sort and a free constructor `_`, encompassing the entire state of the modeled system.
- Ordinary rewrite rules modeling instantaneous changes in a system. These rules have the same syntax as those of the system modules in Maude.
- Tick rules to model the passage of time in a system, these rules have the following syntax:

```
cr1 [label] : {t} => {t\'} in time D if cond .
```

Where t and t' are terms of sort `System` denoting the state of the system, *cond* is the condition of the rule, and D is a term, which can contain variables, of sort `Time` indicating the duration of the rule.

An initial state of a system modeled in RT-Maude must be a term reducible to terms of the form t by the application of equations in the specification. The form of the *tick* rules thus ensures a uniform passage of time throughout all parts of the system. In this case, it is necessary to determine the time advancement strategy to guide the application of *tick* rules. The choice of such a strategy is made by the following RT-Maude command: `(set tick def r.)` where r is a term of sort `Time`, indicating the step of advancement in time defined by the user and attempted by RT-Maude with each application of a *tick* rule.

II.3.3 Formal Analysis and Execution

In Maude, a rewriting theory is specified as a system module, offering an executable mathematical model. This allows Maude specifications to simulate various behaviors of systems or perform different

types of formal analyses. When appropriate conditions are met, these mathematical models can be verified to ensure they meet defined properties or provide counterexamples that show a violation of a given property. Maude is equipped with numerous formal analysis techniques and tools, such as invariant verification, theorem proving, LTL model checking, termination analysis, and consistency analysis. This section focuses on semantic execution, formal analysis and invariant verification, as they are relevant to the thesis at hand. For further details, the reader can consult references [9].

II.3.3.1 Semantic execution under Maude

In a module written in Maude, rewriting rules are the foundational unit of execution, interpreting the local actions of the modeled system. These rules can be executed in constant time and concurrently. Maude enables simulation of such rewrites, either through rewriting rules or equational rewrites, in a module M through two main commands: `reduce` and `rewrite`[9], [11].

By following the syntax: `reduce {in module:} term .`, and the syntax: `rewrite {in module:} term .`, we use a set of commands to facilitate the rewriting of an initial term using rules, equations, and membership axioms in the specified module. The command (reduce: `red`) allows an initial term to be reduced by applying equations and membership axioms in a given module, and both commands (rewrite : `rew`) and (fair rewriting : `frew`) execute a single rewrite sequence from a given initial term, respectively.

Complementing this, RT-Maude expands on these functionalities with a set of commands for simulation, formal analysis, and verification of LTL properties through model checking. The execution and analysis of a timed module in RT-Maude are contingent upon the chosen strategy for applying 'tick' rules of the module. Notably, RT-Maude's time advancement techniques involve sampling the state space of a timed system. Rather than covering the entire time domain, only selected moments are considered, enabling the verification of a system's behavior in a subset of states accessible at specific times. This approach is particularly effective provided the system does not display Zeno behavior, where an infinite number of timed steps occur in a bounded duration.

To simulate the possible behavior of a system, RT-Maude has two modes of timed rewriting, implemented respectively by the standard `timed rewrite` command (or `trew`) and the `timed fair rewrite` command (or `tfrew`). Each of these commands simulates a possible behavior of the system, according to its own strategy, up to a given duration of time.

II.3.3.2 Invariant Verification

Formal verification of models on systems specified in Maude is carried out using tools available around the Maude system, among which the search command plays a pivotal role. This command is not only used for accessibility analysis but also for invariant verification, a straightforward yet highly useful technique. Invariant verification is commonly employed for verifying safety properties in diverse computer systems, and can also be utilized for liveness properties. Given a transition system t and an initial state s_0 , an invariant I is a predicate that defines a subset of states containing s_0 and all states accessible from s_0 through a finite number of transitions[9], [10].

In this context, safety is confirmed if the invariant holds, signifying that no undesirable events occur. Similarly, liveness properties are verified by considering the opposite scenario, i.e. desirable or target states. The `search` command in Maude is instrumental in this process, with its syntax being: `search [n, m] in <module-name>: <Term-1> <search-arrow> <Term-2> <Condition>`. Here, `n` and `m` are optional arguments setting the limits on the number of solutions and the maximum depth of the search, respectively. The `<module-name>` specifies the module where the search occurs, `<Term-1>` and `<Term-2>` represent the initial and target patterns, respectively, and `<search-arrow>` dictates the type of rewriting proof.

For accessibility analysis in Real-Time systems, RT-Maude also offers a timed search command (`tsearch`) to verify whether a state `t` in a system is accessible from an initial state within a given time limit. This command defaults to a breadth-first search strategy in the computation tree (accessibility tree).

Furthermore, the model checking verification of properties expressed in linear temporal logic (LTL) is another critical aspect of system analysis in Maude. This involves an exhaustive exploration of all states accessible from an initial state, using a Kripke structure K and a temporal logic property Φ , to determine if $K \models \Phi$. This method is particularly valuable as it can return an execution trace of the system violating the property when it is deemed invalid [9].

II.3.3.3 Example: River Crossing Problem in Maude

The example involves a classic problem known as the "River Crossing Puzzle" [86]. In the Maude implementation, the scenario involves a shepherd who needs to transport a wolf, a goat, and a cabbage across a river. The challenge is that the boat can only carry the shepherd and one other item at a time, and if left unattended, the wolf would eat the goat, and the goat would eat the cabbage.

```
1 mod RIVER-CROSSING is
2   --- Define sides of the river
3   sort Side . ops left right : -> Side .
4   op change : Side -> Side .
5   eq change(left) = right .
6   eq change(right) = left .
7
8   --- Define items that can be taken across the river
9   sort Item . ops wolf goat cabbage none : -> Item .
10
11  --- Define the state of the system
12  sort State . op <_,_,_,_> : Side Item Item Item -> State .
13
14  --- Variables for sides and items
15  vars S W G C : Side .
16
17  --- Rules for crossing the river with or without items
18  rl [crossWolf] : < S, wolf, G, C > => < change(S), none, G, C > .
19  rl [crossGoat] : < S, W, goat, C > => < change(S), W, none, C > .
20  rl [crossCabbage] : < S, W, G, cabbage > => < change(S), W, G, none > .
21  rl [crossAlone] : < S, W, G, C > => < change(S), W, G, C > .
22
```

```

23 --- Rules for handling unwanted interactions
24 crl [wolfEatsGoat] : < S, wolf, G, C > => < S, none, G, C > if S = G /\ S /= change(C) .
25 crl [goatEatsCabbage] : < S, W, G, cabbage > => < S, W, G, none > if S = G /\ S /= change(W) .
26 endm

```

The RIVER-CROSSING module defines the fundamental logic of the problem, including state definitions and rules for changing states based on actions taken.

- **Sorts and Operations:** Define entities like Side, Item, and State to represent the sides of the river, the items (wolf, goat, cabbage), and the current state of the game, respectively.
- **Change Function:** Switches the side from left to right and vice versa.
- **Crossing Rules:** Define how each entity can cross the river, either with the shepherd or alone.
- **Conflict Rules:** Prevent scenarios where the goat can be eaten by the wolf or the cabbage by the goat when the shepherd is not present.

In this example, we use Maude Strategy Language to specify which actions (crossing actions) should be prioritized and under what conditions. This is crucial for such problems, where not all actions are viable due to the constraints (the wolf eating the goat, etc.). The strategy is integrated with the existing River Crossing model RIVER-CROSSING by including its module. This integration allows the strategy to directly govern and guide the application of rewrite rules defined in the original module.

```

1 mod RIVER-CROSSING-STRATEGY is
2   --- Import the core River Crossing module
3   including RIVER-CROSSING .
4
5   --- Define the strategy for action execution
6   op strategy : -> Strategy .
7   eq strategy = (crossWolf < crossGoat < crossCabbage < crossAlone) ; not(wolfEatsGoat);
8   not(goatEatsCabbage); repeat .
9 endm

```

In this module, "strategy" specifies the order of operations using a priority-based strategy (crossWolf < crossGoat < crossCabbage < crossAlone), and avoids the conflict states (wolfEatsGoat and goatEatsCabbage) attempting to apply these rules in a specific sequence and repeating this sequence to explore possible solutions.

II.4 Conclusion

In this chapter, we introduced the foundations adopted in our work. Initially, we presented MDE principles, we have described MDE method, MDA's principles, and the three-layer model to guide development from abstract requirements to specific implementations. Subsequently, we have explained how these models and Meta-Models can be categorized by general-purpose languages like

UML and domain-specific languages for tailored designs. Moreover, we have presented the IEEE-ISO 42010 standard to support structured software architecture, promoting clear understanding and stakeholder communication throughout a system's lifecycle. We also introduced various concepts related to the Maude language and its different extensions. More specifically, we introduced the syntax and notations of the Maude (including Real-Time and Maude Strategy) Languages and formal analysis techniques in the Maude system.

Chapter III

Key Concepts, Definitions and State of the Art

Contents

III.1	Introduction	31
III.2	Systems-of-Systems	32
III.2.1	Definitions	32
III.2.2	Dimensions	34
III.2.3	Categories	35
III.2.4	Application domains	39
III.3	Current research on SoSs modeling	42
III.3.1	Semi-formal methods	42
III.3.2	Formal methods	49
III.3.3	Synthesis	54
III.4	Conclusion	56

III.1 Introduction

Systems-of-Systems (SoSs) exist in many sectors such as Health, Emergency, Aerospace, and Military sectors. They often appear as a collaboration of Constituents systems (CSs) designed to perform specific goals and missions that none of the CSs can accomplish. This collaborative nature makes the design and development of such systems differ from that of traditional software applications because they have characteristics and needs that particularly distinguish them from traditional systems. Moreover, SoSs are subject to strict technical constraints, functional performance, temporal properties, resource consumption, and emergent behavior. These constraints imposed by the system's needs must be considered from the early phases of the development cycle.

In this chapter, we firstly present a state-of-the-art on SoSs, containing several definitions, characteristics and dimensions, as well as the different types of SoS and their application domains. Secondly, we focus on the design approaches for SoS which we can position our research work. The approaches considered in our review of the existing aim at designing SoSs in an abstract and modular way to reduce its complexity.

III.2 Systems-of-Systems

In an era characterized by technological complexity and systemic interdependence, distributed systems frequently operate as constituents of more complex meta-systems, commonly designated as SoS. The conceptual understanding of SoS is imperative for the effective design, analytical evaluation, and operational management of complex assemblies across a broad array of disciplines, including but not limited to engineering and healthcare. This section is structured to explore the complicated nature of SoSs, aiming to provide a rigorous examination of its constituent definitions, operative dimensions, and typological categorizations.

III.2.1 Definitions

The understanding of the term "System of Systems" lacks a widely accepted definition. The understanding of the term "System of Systems" lacks a widely accepted definition, despite its frequent usage. The use of a separate term implies a taxonomic grouping, suggesting the existence of distinct classes within systems. According to IEEE 1471[52], [72], a system is defined as:

Definition III.1 (System). *a collection of components organized to accomplish a specific function or set of functions.*

Under this definition, a personal computer can be considered a system, and its components such as the disk drive, video monitor, processor, and others can be regarded as individual systems within it. However, the term SoS may not have distinctive power in a formal sense. Nevertheless, the widespread use of the term indicates the value researchers have found in distinguishing highly complex and distributed systems from less complex and compact ones. The common understanding suggests the integration of significantly complex components into a larger system.

According to [23], [51], various discussions by researchers and experts from different viewpoints reflect the complex nature of SoS, they also argue that although until now there is no consensus on a specific definition of SoS [39], [51], [61], but there is some convergence in the literature on their main definitions and characteristics, this includes these examples:

- The author [46] has defined an SoS as large-scale, concurrent, distributed systems composed of complex subsystems. These SoS are represented by “communicating structures”, which focus on the systematic modeling of the SoS, emphasizing aspects such as communication, data traffic and data placement. In these structures, components of subsystems are represented as nodes connected by edges in networks. Nodes have a memory that can hold items, and those

items can move from one node to another along connecting links. Nodes and networks can have a hierarchical organization.

- INCOSE [34] defines a SoS as an assembly of independent CSs that come together to form a more extensive system, providing capabilities that are beyond what any individual system could achieve alone. These independent systems collaborate to generate collective behaviors that they could not produce independently.
- The authors of [41] consider a SoS as an advanced form of composite systems, with a particular focus on the challenges of integrating autonomous and independent systems within large-scale projects. Moreover, SoSs are characterized as complex multi-systems that establish a global objective and comprise interdependent CSs that work together to achieve that objective.
- In [51], the authors define the term SoS as a taxonomic grouping, indicating the presence of distinct classes within systems. This classification can be beneficial in engineering when it reflects specific demands in design, development, or operation. Generally, a SoS is understood as a collection of components that collectively produce behavior or functionality beyond what any individual component can achieve.
- Other definitions refer to a SoS as assemblies of useful, independent sub-systems that collaborate to form a large-scale system. This collaboration enables capabilities that individual CSs could not achieve on their own. The definition and focus of SoSs are generally influenced by their specific application domains. Often, the term SoS is used to denote a package of interoperable systems integrated to function as a single entity for achieving a specific mission capability [6], [15], [18]

In the context of this thesis, we define a System-of-Systems (SoS) as a collection of distributed and complex CSs that interact within a network structure. These CSs, which are physically and functionally heterogeneous, collaborate to achieve a unified capability or system function that surpasses the capabilities of any individual CS. The SoS is driven by specific missions, which give purpose and direction to the coordinated efforts of the CSs. These missions are aimed at achieving objectives or goals that single CSs could not accomplish independently. Within this context, the missions lead to the emergence of new properties and behaviors through the interaction of CSs, underscoring the transformative and dynamic nature of SoS. The missions emphasize the complexity and interdependence among CSs, involving multiple heterogeneous systems working in unison to achieve a global mission. To support these missions, the SoS relies on the allocation and employment of necessary elements such as Resources, Control Mechanisms and Capabilities.

The coordination and integration of these missions and necessary elements enable the SoS to achieve its desired outcomes with enhanced efficiency, effectiveness, and scalability. The interactions, interdependencies, and effective management of missions and these elements are fundamental for the successful functioning and performance of the SoS.

III.2.2 Dimensions

Based on the various definitions presented in the previous section, it's evident that there is no singular, formal definition of what constitutes a SoS. Nevertheless, numerous researchers have attempted to define and characterize SoS through their publications. These academic efforts share common characteristics that describe the domain of SoS in multi-dimensional terms, drawing from shared concepts and aspects. In this section, we aim to provide an overview of these dimensions, presenting the commonly shared characteristics across different works. This serves to highlight the range of perspectives while also emphasizing the converging understanding of what constitutes an SoS [15], [23], [51], [61]:

- **Autonomy of CSs:** Autonomy in SoSs refers to the degree to which CSs operate based on their own rules rather than external governance. This concept is linked to individual ownership and managerial independence, allowing systems to fulfill their own objectives while also contributing to the SoS. The notion of autonomy is pivotal in SoS engineering and implies that constituents should have the capacity for purposeful behavior and goal-setting.
- **Independence:** Independence in SoS signifies the capability of individual systems to operate autonomously when disconnected from the larger structure. This trait is foundational in both the design and operation of SoS, allowing CSs to display varied behaviors, some of which may be opaque to SoS engineers. This highlights the need for model-based techniques that can manage hidden information and dependencies.
- **Distribution:** Distribution in SoS encompasses the spatial or network-based dispersal of CSs that are connected for communication or information sharing. This can refer to geographical distribution across wide areas or to a network-based distribution involving concurrent processes. In such frameworks, it's critical to accurately model the allocation of system processes to computational infrastructures and manage communication mechanisms, including potential failures.
- **Evolution:** Evolution is a defining characteristic, manifesting in changes to functionality, quality, and the composition of CSs. This evolution can be deliberate, often occurring through adaptive or preservative interventions like system upgrades or responses to a changing environment. Such long-lasting, evolving systems do not have a permanent state. Given this dynamic nature, model-based approaches for SoS engineering must provide mechanisms for verifying the ongoing conformance of CSs' interfaces during evolutionary steps. This ensures that specified properties are preserved as the system evolves, requiring periodic re-verification for compliance.
- **Dynamic Reconfiguration:** Dynamic reconfiguration refers to the ability of an SoS to autonomously change its structure and composition, typically in real-time and without planned intervention. This functionality is vital to the resilience of an SoS, allowing it to adapt to

fails and faults. Unlike evolution, which involves planned, small-scale changes, dynamic re-configuration concerns the technical ability of the system to change its composition during its operation. In order to support such dynamic capabilities, SoS models must offer abstractions to dynamically change architectures and interfaces and must provide the means to reason about these changing structures.

- **Emergence of Behavior:** Emergence in the context of SoS refers to the novel behaviors that manifest from the collaborative interactions of CSs. These emergent behaviors provide a level of functionality that individual systems could not achieve independently. Emphasis on emergence places specific requirements on modeling and analytical methods for SoS. Specifically, these methods must be able to capture and verify global properties at the SoS level, as emergent behaviors often cannot be fully understood or articulated at the level of individual CSs. Tools for modeling and analysis should allow the specification and verification of emergent properties, as well as the identification of undesired emergent behaviors, such as feature interactions.
- **Interdependence:** Interdependence in SoS denotes the mutual reliance among CSs to achieve a common overarching goal. Even though CSs maintain a degree of operational independence, their integration into the SoS often necessitates a level of interdependence, potentially requiring trade-offs in individual behaviors to meet the collective objectives. This concept encompasses various related terms such as interrelationships, interdependencies, and belonging. Importantly, the coexistence of "Independence" and "Interdependence" in an SoS does not present a contradiction; rather, it highlights the need for a delicate balance between the two, facilitating both autonomous functionality and collaborative synergy. To effectively model and analyze an SoS, methods should facilitate the explicit mapping of interdependence, enable the tracking of mutual dependencies, and provide the means to evaluate the ramifications of changes in individual CSs on the larger system configuration.
- **Interoperability:** Interoperability in SoS involves the integration of various CSs with different interfaces, protocols and standards. This facilitates seamless interaction between existing and new systems, ensuring consistent functionality. We address this aspect through various terms such as simultaneous operation, capability integration and heterogeneity.

The Table. III.1 identifies a set of key dimensions that are crucial for understanding and exploring the SoS domain [51], [61].

III.2.3 Categories

In SoS Engineering, recognizing and classifying the unique traits of CS is paramount for their effective management and evolution. Unrecognized SoS often miss out on the advantages that come with rigorous Systems Engineering (SE) practices. Understanding the key characteristics such as objectives, authority, and interrelationships between the CS within the SoS can bridge this gap. The U.S. Department of Defense and academic research provide a categorization that divides SoS

Table III.1: Key Dimensions in the SoS Domain.

Dimension	Description
Evolution	Reflects the ability of the SoS to adapt and evolve over time, highlighting the dynamic nature of systems interactions and the long-term development of SoS capabilities.
Emergence of Behavior	Describes new or unexpected behaviors that emerge from the interaction of SoS components, not predictable from the properties of individual components.
Dynamic Reconfiguration	Addresses the SoS’s ability to change its configuration or functionality in response to changing requirements or environments.
Distributed Systems	Focuses on the challenges and characteristics of systems that are geographically distributed, emphasizing the importance of communication and coordination among components.

into four primary types i.e. Directed, Acknowledged, Collaborative, and Virtual. These categories hinge on elements like managerial control, adaptability, and common goals, as well as the degree of independence among CSs. While this framework offers a useful guideline, it’s essential to remember that it’s not strictly exclusive; a single SoS can display attributes from multiple categories. This flexibility enriches the framework’s utility, allowing it to describe various types of SoS based on their varying objectives and levels of authority among CSs, thereby providing a more comprehensive understanding of SoS and their dynamics [17], [23], [61]:

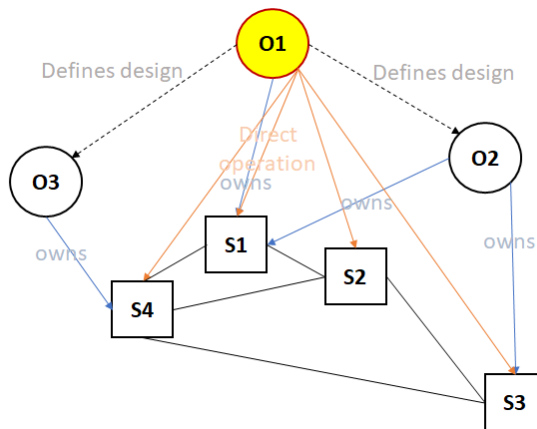


Figure III.1: SoS and CSs Relationships in a Directed SoS.

In a Directed SoS: operators O2 and O3 accept direction from O1 in terms of the specification and operation of the systems they own (O2 owns systems S2 and S3; O3 owns S4) This type of SoS is highly controlled by the central managing entity (O1).

- **Directed SoS:** These systems are built for specific purposes and display a form of planned emergence. CSs (CS) can operate independently but are centrally managed to meet specific objectives. For example, healthcare systems or military command and control structures would fall under this category. This type involves strong specification of a central decision-making

authority. Examples include: Health Care SoSs[3], [87], Mars Science Laboratory (MSL), Military Command and Control [17], NexGen – US Air Traffic Management, Army’s Future Combat systems in the US DoD, etc. This type has a central decision-making authority that strongly specifies the objectives and operations. As the Figure III.1 shows.

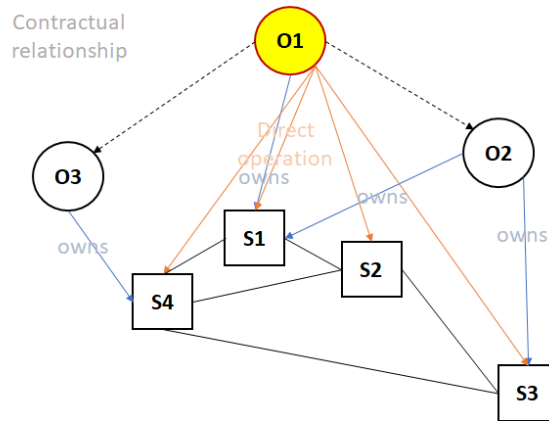


Figure III.2: SoS and CSs Relationships in an Acknowledged SoS.

In an Acknowledged SoS: O1 directs the choice of systems and operation; O2 and O3 have a contractual relationship (e.g. Service Level Agreement) with O1. In this case, the central managing entity (O1) has less control over the systems owned by O2 and O3 (S2, S3, S4) and must rely more on influence.

- **Acknowledged SoS:** Recognized by the DoD and also in academic literature, this category acknowledges a shared purpose among the CS while retaining their independent management. They focus on collaborative management at the SoS level but maintain technical independence at the CS level. Examples include Smart Cities[3], NATO Alliance. Autonomy and ownership are sustained, and changes are decided collaboratively based on common objectives. Examples include: Smart Cities, NATO Alliance, SESAR-Single European Sky (EU), etc. Here, autonomy and ownership are sustained, and decisions are made collaboratively based on shared objectives. Figure III.2 shows the relationships between the SoS and constituent systems in the Acknowledged SoS.

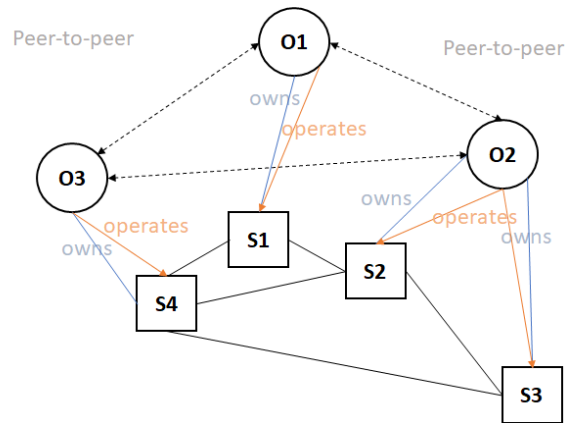


Figure III.3: SoS and CSs Relationships in a Collaborative SoS.

In a Collaborative SoS: there is mutual agreement to collaborate; usually covered by agreements of some form, but there is no overall managing entity; systems owners (O1, O2, O3) operate their systems and collaborate with others to realize some shared benefit.

- **Collaborative SoS:** In this setup, the CS voluntarily collaborate for mutual benefits without the compulsion of a centralized management. It often has limited or no power to enforce decisions. Domains like regional crisis response and public transport employ such systems. The concept of centralized management exists, but it has limited or no enforceable powers. Examples include: Regional Area Crisis Response System [85], Public Transport Information [3], [19], [40], Global Financial System Intelligent Transport Systems, Internet Engineering Task Force. While a concept of centralized management exists, it has limited or no enforceable powers. Figure III.3 shows the relationships between the systems and the SoS in Collaborative SoS.
- **Virtual SoS:** Lacking any managerial control or common purpose, these systems are highly emergent, making it challenging to discern their exact functionality. Examples include the Internet and automated high-speed trading systems. Because they lack managerial control, their behavior and fulfilled objectives are highly emergent. Examples include: Internet, Automated High-Speed Algorithmic Trading Systems, National Economies, etc. Due to the lack of managerial oversight, their behavior and objectives are often unpredictable. Figure III.4 shows the relationships between systems and the SoS in Virtual SoS.

The core characteristics of CSs, namely objectives, authority, and inter-relationships—are what principally differentiate SoS from other types of systems. Any system lacking these key traits cannot be accurately classified as an SoS. Within this framework, it's crucial to recognize that SoS categorization is neither mutually exclusive nor comprehensive. Specifically, it's common to find a single SoS composed of local subsystems that belong to different categories. These mixed categorizations arise because of variations in control and ownership levels among the CSs. Given this complexity, stakeholders may hold conflicting views regarding the appropriate levels of control within the SoS. This presents a challenge in developing models that accurately reflect these multi-faceted

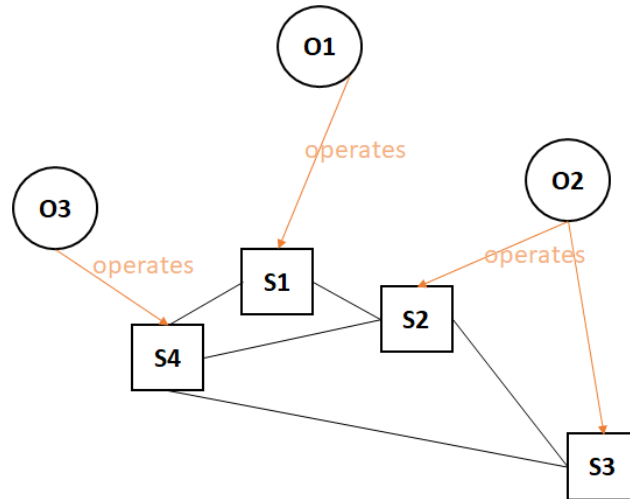


Figure III.4: *SoS and CSs Relationships in a Virtual SoS.*

In a Virtual SoS: Owners (O1, O2, O3) access other systems through their own systems in order to realize individually sought benefits, though high level emergent behaviour may still occur. There is no overall goal, no central management and interoperation is achieved by recognized protocols, or standards, not through individual agreements between pairs of systems.

control dynamics. The issue is particularly acute for SoS types that are Directed and Collaborative, where different degrees of managerial control and autonomy come into play. Although the four types of SoS —Directed, Acknowledged, Collaborative, and Virtual— are not the only ways to categorize SoS, they provide a practical framework. The utility of this classification system lies in its ability to describe a broad range of SoS configurations based on their objectives, authority levels, and the nature of inter-relationships among the CSs.

III.2.4 Application domains

Today's SoSs are ubiquitous and their applications span across an increasingly diverse array of domains. Both scholarly research and practical studies have delved deeply into these domains. In this section, we provide a synthesis of existing research efforts and literature on SoS to illustrate the wide-ranging applications that define the SoS landscape. From healthcare to military systems, and from intelligent transportation to global financial systems, SoS has a transformative impact across sectors, underlining the importance of understanding its complexity and adaptability.

- **Transportation:** the transportation of goods and passengers is an essential activity for daily lives. People use railways, traffic and road networks, vehicles, airplanes, ships, trains, sensors, airports, road-side infrastructures, traffic management centers, satellites, and other transportation system components by adopting different variations of wireless communication technologies and standards. Each one of these components relies upon multiple modes of transportation (air, water, and land transport, pipelines, cable transport, and space transport) and has its own complexity that makes building a modern intelligent transport SoS largely trying to bypass problems related to sensor technology, distributed control communications, accounts, and

control mechanisms to improve safety, coordination, and services in traffic management while sharing information in real time.

The authors of [19], have relied on the distinguishing characteristics of SoS and other related traits (Networks, Heterogeneity and Trans-domain) to explain how the National Transportation System (NTS) can be viewed as a SoS. Likewise, despite transportation varies, communities' needs, structures changes, and various complexities, the world today places before us many realistic and well-known transportation SoS such as Intelligent Transport Systems (ITS), Air Transport System (ATS) and National Airspace System (NAS), National Transportation System (NTS), Air Traffic Management SoS (ATM), Maritime Transport SoS (MTSoS), Next Generation Air Transport (often What is referred to as NexGen) and unmanned aerial vehicle (UAV) [33], [40], [61].

- **Healthcare:** A HSoS can be defined as an arrangement of independent large scale complex, dispersed CSs. HSoSs also exhibit several key characteristics of a general SoS. First, they exhibit operational and managerial independence. For example, hospitals and organizations such as HMOs, though they work together, work independently of each other. Similarly, government funding agencies such as Medicare/Medicaid and physicians, hospitals work independently of each other. Second, they are geographically dispersed with no central management. Third, healthcare systems exhibit evolutionary development in the sense that they change continuously in response to government regulation needs and new threats [87].

In fact, the integration of different computational services and medical devices of MSoS' CSs can perform various operations in real-time, and the different medical devices operate on/or near the human body. Since these applications are critical and impact patients' life, there is a strong need to ensure efficiency, safety, reliability, and correctness. It is also important to maintain the right timing of the behavior of such components [33].

- **Military Defense:** Most military missions depend on sets of Government defence organizations to work together effectively as a SoS to provide the needed user capability whether those missions are implemented by a single nation or by a coalition. National defence is one of the leading SoSs application domains because of various domains where SoS occur in military applications like missions, platforms and information technology. Example of the type of mission systems includes command and control, communications, and IT-based SoS.i.e. systems interoperate in this environment as missions exchange critical operations to support different objectives in a constantly changing, sometimes adversarial, environment. Most military missions depend on sets of systems to work together effectively as a SoS to provide the needed user capability whether those missions are implemented by a single nation or by a coalition [17], [48].
- **Smart Houses:** "Smart home" technology for autonomous service delivery requires a constant focus on the needs and interests of inhabitants. SHSoS has become a widely accepted term to describe internal and external surveillance and supervision systems in houses. The nature

of these systems is inherently complex due to the different independent and interoperated complex CSs that interact with each other within the SHSoS. Besides, the interests of key inhabitants and their customs and manners are translated into a myriad of needs that lead to heterogeneity in the delivery and management of SHSoS services. The major components involved in the SHSoS are security protocols, detection sensors, energy usage and control information exchange[33].

- **Smart Energy Grids:** The smart grid is a well-known SoS application and it is considered one of the most complex systems ever. It has made it possible to improve living standards for humans as all aspects of life are directly related to central/industrial power plants, energy storage and transmission facilities, renewable energy resources [33]. A smart grid can be considered as SoS based on the characteristics that define what constitutes the SoSs [1]:
 - A single component can operate effectively even if it is isolated from the overall network.
 - All elements in the smart grid operate independently, although they together influence the result of the whole network.
 - Components can be isolated, added, integrated or retired at any time without causing an impact to other parts of the electrical system.
 - The integration of different CSs with various capabilities to serve stochastic energy loads leads to emergent properties that fulfill the major purpose of the smart grid.
 - The components of the smart grid are geographically dispersed. They are linked only through information exchange channels.
- **Emergency Management and Response:** EMRSoS are varied and expensive. In these type of systems, various operationally independent CSs intent to deliver the desired emergent phenomenon of improving the efficiency and the reliability of fast response to handle the threats, and manage emergencies by means of information technologies, social media, collaboration among professionals, local and national authorities and the community. Not all disasters require the equivalent CSs assets to manage and resolve them. Therefore, various components and technologies (e.g. unmanned aerial, ground vehicles, sensors, etc.) can be integrated and deployed into the infrastructures to provide efficient search and rescue efforts to manage emergency response and disaster recovery in the future [33].
- **E-commerce:** E-commerce serves as a real-world example of a SoS. It involves various independent systems—like virtual marketplaces, payment handling, inventory management, and shipping—that must interact seamlessly to achieve the common goal of facilitating online sales and deliveries. Companies like Amazon employ decentralized architectures that integrate hundreds of systems to support millions of customers. While e-commerce companies often own many of these systems, giving them more control and flexibility, they also rely on systems owned by external stakeholders, like payment processors or shipping services. This blend

of ownership creates challenges in development strategies and necessitates effective communication between system owners, making e-commerce a complex but illustrative example of SoS[61].

III.3 Current research on SoSs modeling

The complexity of the dynamic nature of components in an SoS presents a significant challenge for modeling and analysis. Addressing this challenge requires robust approaches and formalisms, ranging from semi-formal to formal, each with its strengths and limitations[36], [88]. Semi-formal methods like UML and SysML provide intuitive and flexible frameworks for capturing the architectural design and facilitating communication among stakeholders. In contrast, formal methods such as ADL and BRS offer rigorous, mathematically grounded tools for ensuring the correctness and reliability of SoS, particularly in safety-critical domains. This means that the use of both of them will leverage the strength of both formalisms and tools to create robust, adaptable, and executable models that can aid in the design, simulation, and verification of complex SoS architectures. In this section, we examine recent developments in SoS modeling methodologies, structured into two sub-sections: Semi-Formal and Formal methods.

III.3.1 Semi-formal methods

The papers in this category focus on using SysML, UML profiles and other modeling languages for the architectural design and analysis of SoS. These approaches, while structured, offer a degree of flexibility and are tailored to facilitate understanding, communication, and practical application in complex system environments[36], [88].

The authors in [59] have focused on the modeling of SoS using a SysML profile. The paper is a part of the AMADEOS project and delves into the complexities of SoS, and it lies in developing a conceptual model for SoS, providing a robust vocabulary to capture the complex interrelationships within these systems. The model not only aids in understanding SoS but also serves as the foundation for the SysML profile designed specifically for SoS modeling. The study demonstrates the practical application of this SysML profile through a Smart Grid scenario. This application highlights the profile's utility in modeling the high-level design of SoS architectures, supporting various types of analyses, and its integration into an MDE tool. This integration facilitates rapid modeling, validation, code generation, and simulation of SoS. The research adopts a viewpoint-based approach to SoS modeling, considering diverse aspects like structure, dynamicity, evolution, dependability, security, time, multicriticality, and emergence. Each viewpoint is tailored to address specific challenges and requirements in the design and operation of SoS, ensuring a comprehensive analysis and management of these complex systems. Additionally, the paper discusses enhancements to the SysML profile, including the addition of new viewpoints and improvements in user-friendliness for easier tool integration. These enhancements are crucial in simplifying the design, validation, and simulation processes, making it more accessible for SoS engineers. The study significantly contributes to the field by addressing the cognitive complexities involved in SoSE. It offers a structured methodology

to model, analyze, and manage the dynamic and often complex interactions within an SoS, thus facilitating a deeper understanding and effective management of these systems. The research, particularly illustrated through the Smart Grid scenario, stands as a significant advancement in SoS modeling, providing a detailed conceptual framework and practical tools for designing and analyzing complex systems across various domains.

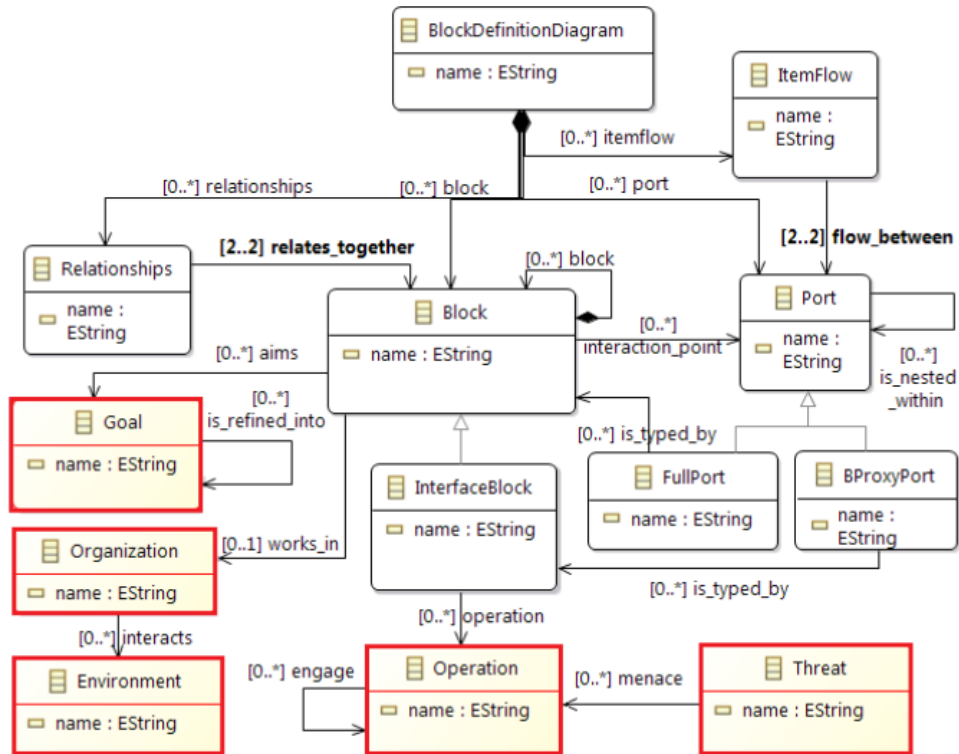


Figure III.5: SoSsec MetaModel Block Diagram. [27].

In a similar approach, a novel approach to addressing security challenges in SoS architectures has been presented in [27]. The research focuses on the problem of cascading attacks within SoS, where vulnerabilities in CSs can combine to produce significant security breaches. To tackle this, the authors propose a DSML called SoSsec, which extends the SysML to incorporate security aspects specific to SoS. The paper highlights the complexity of modeling SoS security due to their inherent characteristics such as operational independence, absence of central authority, and emergent behavior. The authors emphasize the need for a modeling language that can represent SoS security architecture effectively. The proposed SoSsec DSML is designed to enable the discovery, analysis, and resolution of cascading attacks in the architecture phase, thereby preventing costly and time-consuming revisions in later stages of development. An integral part of the research is the development of a graphical editor for the DSML, which facilitates modeling SoS security architectures. The authors illustrate the utility of SoSsec through a Smart Campus case study, demonstrating how it can model and identify potential cascading security threats in an SoS environment. By employing MDE principles as shown in Figure III.5, SoSsec provides a comprehensive approach to capture the

security architecture of SoS, ensuring early detection of vulnerabilities that could lead to cascading attacks.

In the same context, the authors of [8] have presented a comprehensive approach to SoSE, focusing on the critical transition from mission definition to architecture description. The main point of the paper is to address the challenges in SoS development, particularly the lack of coherent and structured methodologies that can effectively bridge the gap between mission definition and architectural realization. The authors have proposed a model-based process that strengthens the link between the SoS analysis stage and the architecture stage in the SoS lifecycle. This process is centered around the concept that the mission and role definitions for SoS should be abstract enough to accommodate environmental variability. These definitions are then translated into an abstract architecture that guides system architects during the design and evolution stages as shown in Figure III.6. The paper makes use of SysML for the proposed process, adapting it to meet the specific needs of SoSE. A key feature of the approach is its emphasis on mission-oriented analysis, which is considered vital for effective SoS design. The process is designed to be sufficiently formal to guide and control the system architect’s decisions during the design and evolution stages. A case study on crowd management SoS is included to illustrate the practical application of the proposed process. This case study demonstrates how the approach can be used to model and identify potential security and operational challenges in an SoS environment, highlighting the relevance and applicability of the model-based process in real-world scenarios.

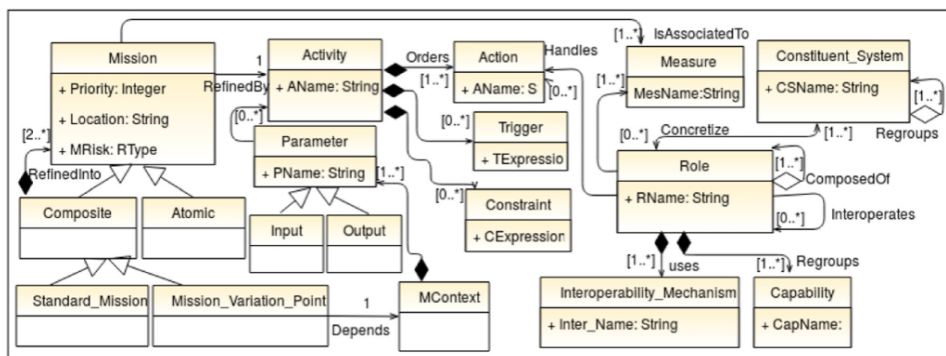


FIGURE 2 Mission conceptual model

Figure III.6: Mission conceptual model [8].

Another interesting contribution of [71] has presented a detailed approach to modeling and simulating net-centric SoS. The research focus on the challenges involved in designing and comprehending large networked systems, such as the Global Earth Observation System of Systems (GEOSS). This paper demonstrates how SysML and Colored Petri-nets (CP-nets) can be effectively combined to model and simulate complex SoS. The authors highlight the importance of understanding the interfaces and interactions among the various systems, subsystems, and components within an SoS. SysML is used to model these aspects, demonstrating an object-oriented approach to model development. The paper also discusses the issues related to architecture description, development, presentation, and integration for the chosen domain of GEOSS, an evolving complex network-centric

system. They have used Colored Petri-nets to synthesize an executable model from the static views developed using SysML. This executable model is used to validate the architecture against the static model. The authors define a methodology to model and simulate complex network-centric SoS, aiming to understand and simulate their behavior using a scenario-based approach. This approach involves a step-by-step process of transforming SysML diagrams into an executable model, followed by simulation and validation. The paper also highlights some of the key differences between SysML, used to model a wide range of systems, and UML, which is primarily used to model information systems. The integration of SysML and CP-nets offers a comprehensive approach to capture both the structural and dynamic aspects of SoS. This integration allows for the exploration of various scenarios, contributing to a deeper understanding of the behavior and interactions within the SoS.

A three-tier framework has also been introduced to analyze requirements and architecture design for SoS [89]. The framework focuses on the complexities of large-scale complex SoS, where traditional design principles are challenged by rapidly growing complexity and continuously changing requirements. The authors have proposed a service-oriented modeling method for SoS requirements analysis. This method is based on a three-tier framework with multi-ontologies, aiming to provide a reusable and flexible solution for SoS requirements modeling. The approach enables the high-layer requirements description of SoS to be directly mapped to a service-oriented system design architecture. The authors emphasize that service-oriented computing offers a flexible integration architecture to meet the dynamic requirements of SoS, making it a suitable approach for analyzing large-scale complex systems. To this end, they have introduced a method that can map strategic level requirements down to IT implementation by considering service as the basic granularity. It also provides rigorous modeling semantics through defining multiple ontologies, which is essential for domain knowledge reuse. Moreover, a case study is provided to demonstrate the applicability of the method. This practical example underlines the potential of the proposed service-oriented modeling method in practical scenarios.

To address the dynamic requirements and high complexity of SoSs, the authors [35] have offered a comprehensive MDA approach for architecting, modeling, and simulating SoS to integrate various modeling languages and simulation techniques, it is based on the Service Oriented Architecture Modeling Language (SoaML), used in conjunction with Model Driven Service Engineering (MDSE). This combination enables a flexible integration approach, combining complex system analysis methods with service-oriented methodologies. The authors have used SysML to manage the complexity of SoS requirements and SoaML to improve IT implementation, and DEVS simulation to validate the architecture design. One of the key aspects of this approach is its multi-level modeling capability. It allows for transformations between different levels of models and generates service implementation artifacts. This facilitates aligning complex business requirements with IT systems, addressing a common challenge in SoSE. The paper also has presented an example covering some of the development phases of SoSE.

From safety-critical contexts, the authors [31] have focused on enhancing the modeling and management of SoSs to adapt SoS during runtime while ensuring safe operation. The paper proposed an MDA that defines interactions and goals as fundamental elements of design. This approach em-

phasizes the adaptability of SoS to changes, including context changes, while maintaining safety. The authors have introduced a well-defined modeling approach based on components as structural elements, the contract paradigm for designing interactions, and graph transformations to address the adaptiveness of SoS. They enrich the component model with explicit goals that support evaluation functions to determine the level of target achievement. This method allows for the analysis of interaction protocols, role definitions, and the transformation rules that guide the adaptation process in response to changing contexts. A key aspect of this work is its focus on self-adaptiveness, a major characteristic of SoS, which encompasses the reconfiguration of the SoS in response to changes within its context. The paper provides detailed explanations and examples to illustrate these concepts, using scenarios like a classical firefighting setting to demonstrate timing analysis and the application of their approach.

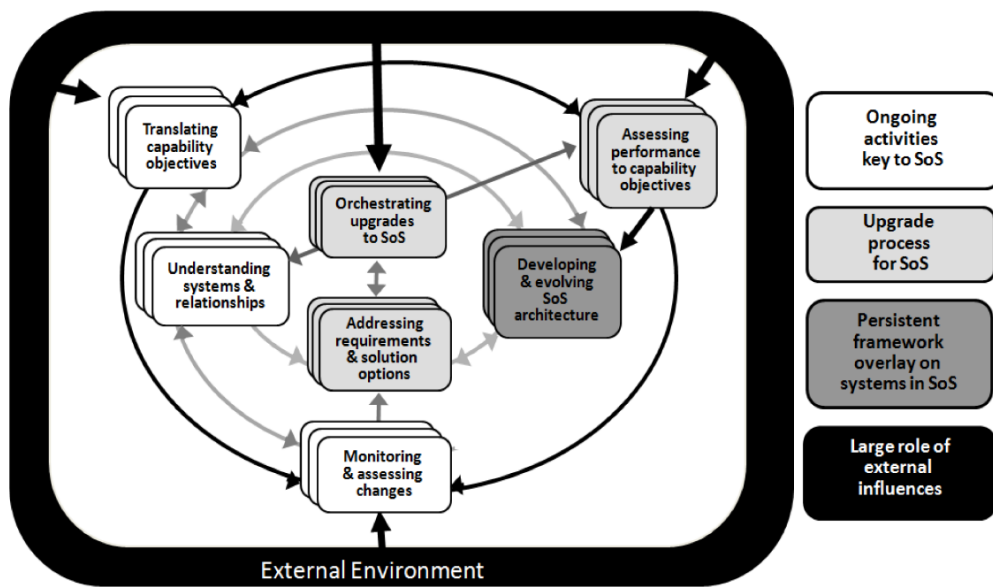


Figure III.7: SoSE core elements [47].

Another contribution of SysML, where the authors of [47] have explored the application of SysML in understanding and evolving SoS. The research is grounded in the context of the Department of Defense (DoD) systems engineering guide, which highlights the evolution of traditional systems engineering activities to support SoS engineering (Figure III.7). The paper outlines the challenges in managing SoS, particularly in the context of software-intensive systems, and proposes the use of SysML as a tool to model and simulate SoS for better decision-making and evolution. The authors emphasize the importance of modeling in understanding the relationships and interactions within SoS, a crucial aspect given the complex and emergent behaviors of these systems. A significant part of the paper is dedicated to detailing how SysML can be employed to characterize SoS architectures and capabilities. The authors have provided a thorough explanation of SysML’s application in various aspects of SoS, including translating capability objectives, understanding systems and their relationships, developing and evolving SoS architecture, and addressing requirements and solution options. The paper also includes practical examples and illustrations to demonstrate the utility of

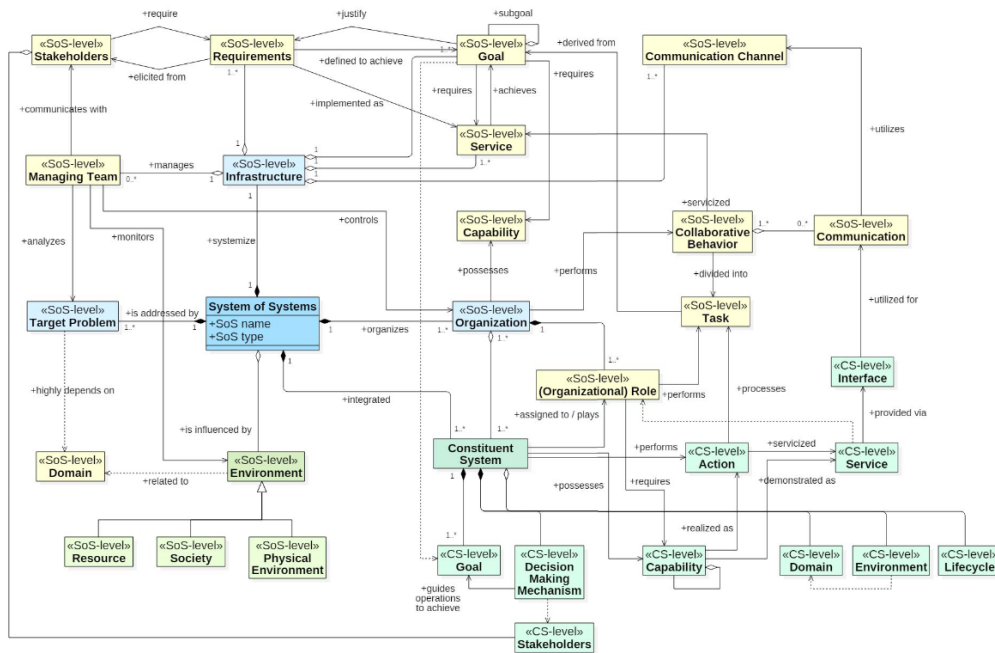


Figure III.8: Simplified version of M2SoS[5].

SysML in real-world scenarios. The authors describe a hypothetical Regional Area Crisis Response SoS (RACRS) to illustrate how SysML models can be used to model the “as-is” state of an SoS and explore alternatives for new capabilities and performance enhancements.

The authors of [5] have explored the development of a conceptual meta-model, called M2SoS (Meta-model for SoSs), for representing SoS ontologies (see Figure III.8). This research addresses the critical need for a holistic understanding of SoS, especially for various stakeholders and engineers involved in SoSE. The focus is on systematically integrating independent CSs to achieve higher-level common goals of an SoS. The paper begins by emphasizing the complexity and independence of CSs within an SoS and the challenges in their integration for achieving overarching objectives. The authors propose the development of a meta-model that provides a common knowledge base to facilitate a comprehensive understanding of the SoS as a whole. This model is developed by investigating several documents related to Mass Casualty Incident (MCI) response systems. The investigation led to the identification of essential objects and features required for SoS descriptions, which were then generalized into SoS entities. A key contribution of the paper is the design of the M2SoS meta-model, which borrows organizational concepts from meta-models for multi-agent systems. The entities and relationships in M2SoS are redefined to specify SoS concepts, providing a structured way to represent high-level ontologies for SoS. The meta-model is analyzed with respect to SoS characteristics, evaluating its ability to represent ontologies for two distinct SoS case scenarios. The paper outlines the development process of M2SoS, starting with the selection of an MCI response system as a case study. The authors analyzed documents related to this system to identify objects comprising the SoS and generalized these objects into SoS-oriented entities. The development process also involved defining requirements for the SoS meta-model based on the investigation.

Employing an Industry 4.0 domain, the paper [4] offers a comprehensive exploration into the application of I4.0 principles in the context of SoSs, particularly focusing on the construction domain. The research is significant in understanding how the concepts of I4.0, primarily developed for manufacturing, can be adapted and applied to other domains, notably in construction SoS. The paper begins by outlining the challenges inherent in integrating constituent systems (CS) into an SoS, especially those preexisting systems that require adaptation to fit within the SoS context. The authors delve into the nuances of applying I4.0 standards to enhance the flexibility and adaptability of CS, evaluating the potential of these standards beyond manufacturing. The study is grounded in a case from the construction domain, where the authors develop a generic SoS architecture and suggest several extensions and adaptations of I4.0 standards. A notable aspect of the paper is its focus on the continuous interplay between the design decisions at the SoS level and those at the level of its CSs. This approach acknowledges the operational and managerial independence of CSs and the need to adapt existing systems to fit within a future SoS context. The case study in the construction domain serves as a practical demonstration of the paper's concepts. It illustrates how I4.0 standards can be applied to make CSs more flexible and adaptive. The study explores the challenges of applying these standards in a domain characterized by less repetitive work than manufacturing and with looser managerial control.

The authors of [43] have offered an innovative solution to the complexities involved in modeling SoSs architectures. This research is particularly relevant in the complicated nature of software architectures often posing significant challenges in design, verification, and validation. Central to this paper is the development of a multi-scale description for software architectures. The authors introduce an iterative modeling approach that transitions from a coarse-grain to a fine-grain description. This method allows for the validation of software architectures at various levels of detail, effectively managing the complexity inherent in SoS architectures. To facilitate this process, the approach integrates visual notations that extend the graphical UML. These notations are used to represent both the structural and behavioral features of software architectures, ensuring a comprehensive and coherent modeling process. A key aspect of the methodology is its focus on iterative refinement. Starting with an abstract architecture, the model is progressively detailed, allowing for validation at each step. This step-wise refinement is crucial in managing the complexity of SoS architectures, ensuring that each layer of detail contributes to a more accurate and functional model. The use of UML diagrams in this process is particularly notable. UML, a widely recognized and used language in software engineering, facilitates the presentation of both structural and behavioral properties of the architectures. By integrating UML notations, the approach aligns with established software architecture practices, making it more accessible and applicable in various engineering contexts. The practical applicability of this approach is demonstrated through a case study focusing on the Emergency Response and Crisis Management System (ERCMS). This case study underscores the effectiveness of the proposed multi-scale modeling approach in a real-world SoS. It highlights how the approach can handle the complexities and dynamic nature of software architectures in practical scenarios.

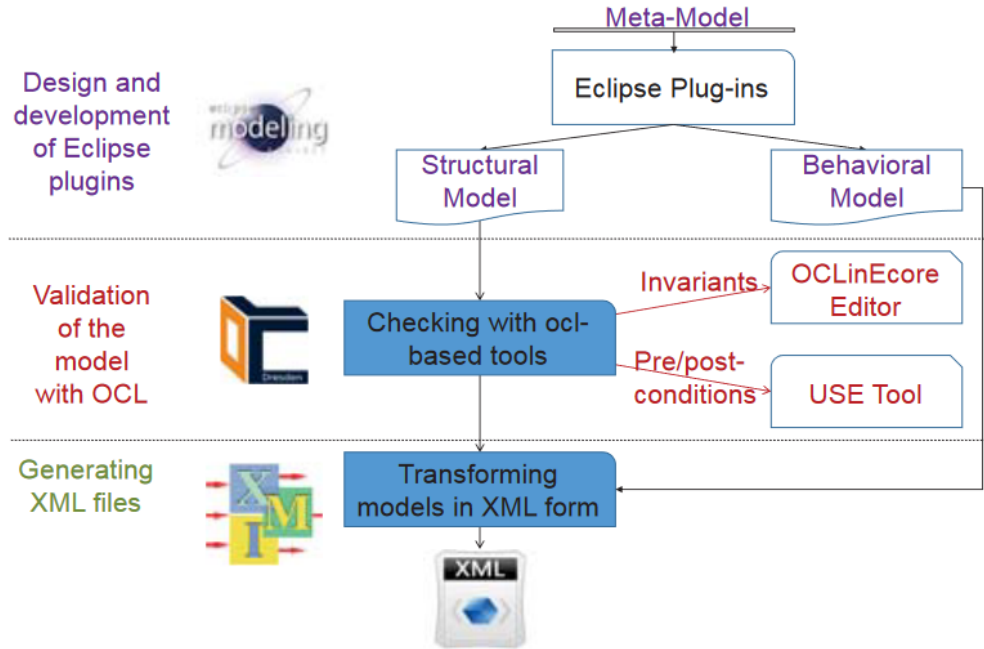


Figure III.9: Description of the design process [43].

III.3.2 Formal methods

The research in this category employs formal languages and methodologies like ArchSoS, mKAOS, and Bigraphical Reactive Systems (BRS). These approaches are more rigorous and rely on formal logic and operational semantics, providing a precise framework for modeling, verifying, and analyzing SoS architectures especially in behavioral and dynamic contexts [26], [69].

From a modeling and verification perspective, the paper [79] has introduced a formal approach for modeling and verifying SoS architectures. This research aims to provide a structured and formalized way to design and analyze SoS, particularly addressing the challenges of the dynamic nature and hierarchical structures inherent in SoS. The paper's central contribution is the development of ArchSoS, a formal ADL, which integrates concepts from BRS and Maude language (Figure. VII.2). ArchSoS is designed to model the hierarchical structures of SoS and their dynamic reconfigurations. It also enables managing potential cooperation between CSs, offering a graphical and formal syntax to facilitate understanding and analysis. The paper details the operational semantics associated with ArchSoS, using rewrite theories to define an SoS semantic. This includes implementing ArchSoS specifications in the Maude language, which provides capabilities for executing and simulating these specifications and analyzing their properties. A case study, the Crisis Response System of System (CRSoS), is used to illustrate the application of ArchSoS. This example demonstrates how the language can model and identify potential challenges in an SoS environment, specifically focusing on missions like the Fire-Distinguish Mission. Maude's rewriting engine and LTL model-checking engine enables the verification of SoS behaviors and the consistency of SoS missions through qualitative analysis.

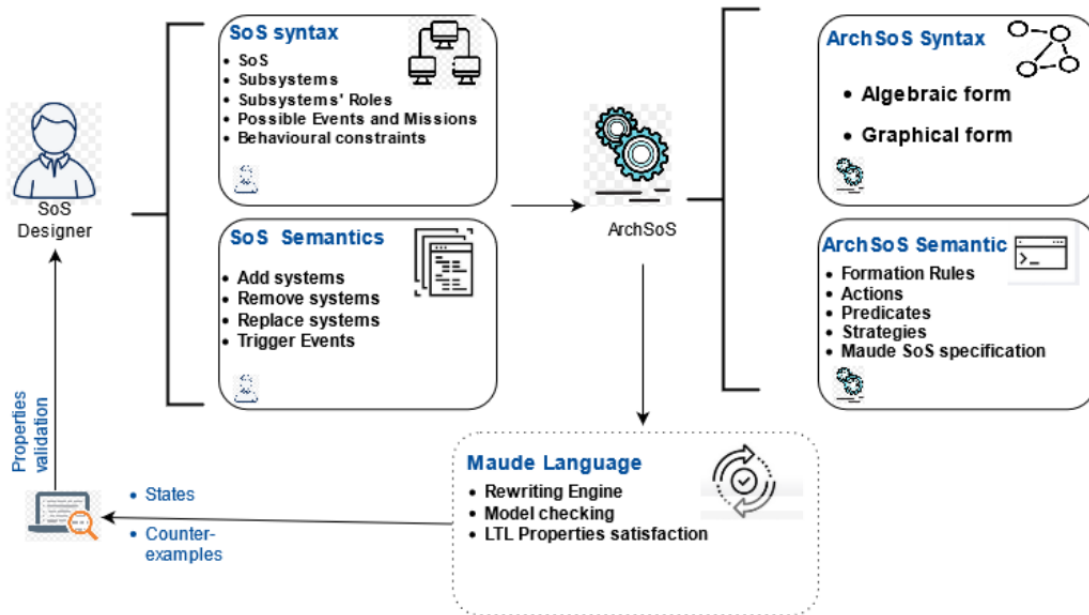


Figure III.10: *ArchSoS* definition process [79].

In the same context, the authors of [80] have proposed a methodology for formally verifying mission-related properties in architectural models of SoSs, with a particular emphasis on addressing emergent behaviors and mission accomplishment. Leveraging mKAOS and DynBLTL language, they formally described properties, missions, and emergent behaviors. They pursued verification of properties across three distinct levels, which are automatically derived from the mKAOS model representing the SoS. Verification is executed through statistical model checking. The proposed tool is constructed to be extensible, enabling compatibility with various languages by the verification engine through a universal interface for language simulation. Moreover, the tool manages communication with the statistical model checker, PlasmaLab, while also extracting pertinent properties from an mKAOS model and managing the connection between an architectural model simulator and PlasmaLab. Furthermore, the tool is capable of conducting a post-evaluation process, using PlasmaLab outputs to generate detailed reports of the verification process, with indicators of whether a constraint was violated and identifying when and where the violation occurred. Their solution is not tied to a specific modeling language, allowing stakeholders to check conformance with a mKAOS mission model across various model types. A critical part of this study is the application of this methodology in a real-world scenario, demonstrated through a case study of a Flood Monitoring SoS. This example is instrumental in showcasing the practicality and effectiveness of the proposed method. The Flood Monitoring SoS, with its dynamic environment and mission-critical requirements, serves as an ideal testbed for the verification approach.

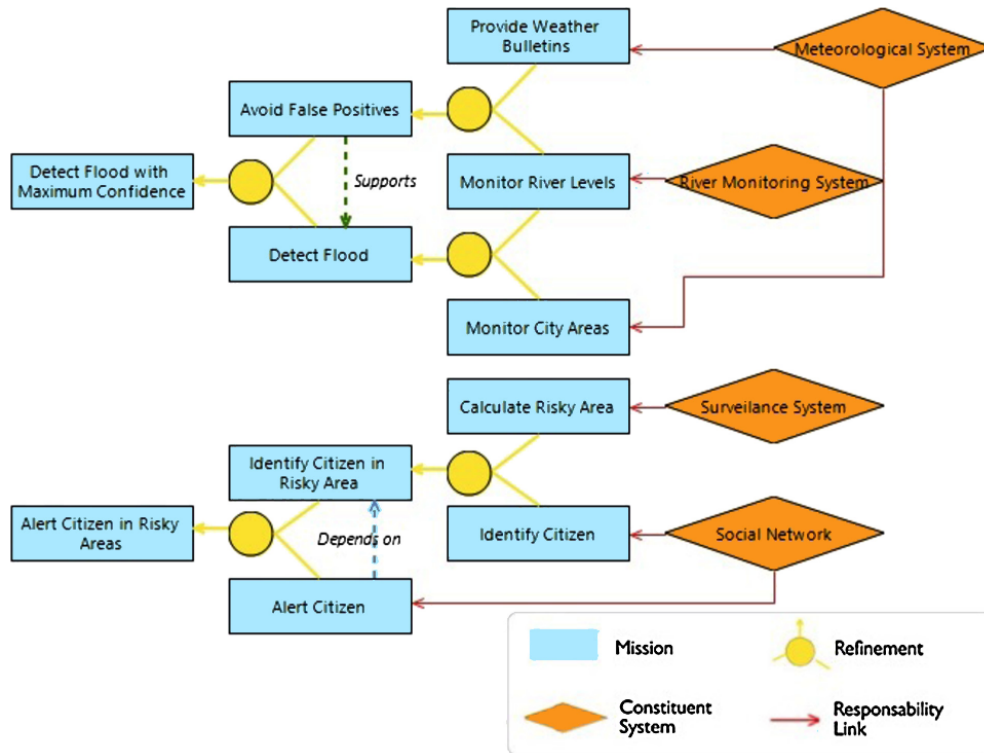


Figure III.11: Missions and CSs in FMSoS. [80]

The collective outcomes of studies by the authors of [65]–[67] have presented a SosADL, an ADL based on pi-Calculus with Concurrent Constraints, articulating its capabilities and adaptations. The original version of SosADL, presented in [67], offers a specialized framework for describing SoS architectures, by embodying pertinent architectural concepts while concurrently providing a formal language that supports automated analysis. This foundational work establishes a methodical approach to capturing the complexity and nature of SoS architectures, contributing significantly to practical models around SoS design and analysis. Building upon this, [66] introduces an extended version, which enriches the approach by enabling the description of evolutionary architectures that sustain emergent behaviours and support dynamic reconfigurations for ongoing SoS missions. The work defines SosADL from a behavioural viewpoint, facilitating the specification of independent CSs, mediators among them, coalitions of mediated CSs, and the architectural conditions that necessitate the emergence of specific SoS behaviours. Meanwhile, [65] shifts the focus towards supporting automated verification and establish correctness properties of SoS architectures, thereby ensuring that the structured and analytical approach to SoS architecture is not only descriptive but also verifiable. In another research focus, [60] delves deep into the functionalities of the ArchWare ADL, emphasizing its role in active architectures. The paper underscores the challenges of a cohesive system wherein model specification and enactment coalesce as integral facets of a singular, dynamic execution state. The authors maintain that the model specification, at any given moment, renders an accurate description of the model execution, thus reinforcing the precision and reliability of active architecture specifications within the ArchWare.

Using bigraphical reactive systems, the authors of [30] have presented a novel multi-scale modeling methodology for SoS using bigraphical reactive systems. This methodology, named B3MS, offers a "correct by design" approach, ensuring the correctness of SoS architectures through a refinement process. The paper emphasizes the need for rigorous modeling methods in SoS due to their complexity and dynamic nature, highlighting the challenges in ensuring the correctness of their architectures. B3MS methodology is rooted in the formal technique of BRS, combined with an approach inspired by multi-scale modeling. This approach begins with a coarse-grained scale defined by the designer, which is then automatically refined by adding lower scale details. This refinement process respects the system constraints to ensure the correctness of the obtained scale architectures. Additionally, the paper addresses the dynamic aspect of SoS by providing model-based rules for reconfiguration actions, essential for managing changes in SoS. A key feature of this methodology is its ability to handle both the static and dynamic aspects of SoS architectures. The static aspect is modeled through a multi-scale approach, where each scale represents different levels of detail within the SoS. The dynamic aspect, on the other hand, is managed by defining reconfiguration meta-rules, which guide the evolution of the SoS over time. To demonstrate the applicability and effectiveness of the B3MS methodology, the authors present a case study on smart buildings. This case study showcases how the methodology can be used to model and manage the complexities of smart building systems, illustrating its potential in practical scenarios. The case study effectively demonstrates the methodology's capability in handling both the static architecture and dynamic reconfiguration of SoS.

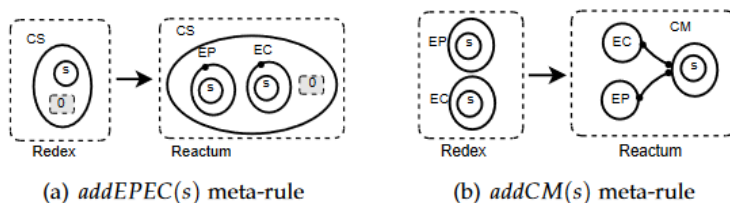


Figure III.12: *Meta-rules for adding publish/subscribe components*[30].

In another formal approach, the authors of [83] have presented a unique perspective on enhancing interoperability and emergent behavior in SoSs through the use of bigraphs. The focus of the paper is on addressing the challenges inherent in designing highly interactive distributed systems, such as e-learning environments, where dynamic adaptation to user needs and operational conditions is crucial. The authors have proposed the concept of bigraphs as a means to represent and manage the dynamic interactions within SoS. Bigraphs offer a way to capture both the physical (space) and logical (link) relationships within systems. This dual representation is particularly useful in managing the complexity of SoS, where multiple, independent CSs interact in a federated manner to achieve overarching objectives. The use of bigraphs allows for a detailed representation of these interactions and supports the re-specification of systems through behavior adaptations, a key aspect in achieving emergent behavior in SoS. The paper explores the challenges of achieving interoperability

in SoS, which is essential for the CSs to work together seamlessly and adapt dynamically to changing environments. The authors highlight the importance of preserving interoperability among systems from both structural and behavioral perspectives, which is where bigraphs play a crucial role. This approach ensures that systems within an SoS can maintain their operational independence while still being able to interact and cooperate effectively. To demonstrate the practical application of their approach, the authors use learning environments as a case study. They show how bigraphs can be effectively employed to orchestrate two independent and distributed CSs, enabling them to respond directly to changes in the federated system’s context. This case study illustrates how the bigraph approach can lead to the emergence of new functionalities and behaviors in an SoS that are more than the sum of its individual parts.

In the context, the authors of [29] have introduced BiGMTE, a tool designed for bigraph matching and transformation. This tool is significant in the field of SoSs for its ability to model and simulate SoS architectures, addressing the challenges of their dynamic nature and complex interactions. The core of this paper revolves around the application of BRS and graph transformation techniques to SoS. The authors propose a method that encodes bigraphs into graphs and reaction rules into graph rules, allowing for the simulation of BRS using a Graph Transformation System (GTS). This approach leverages existing research and tools in graph transformations, applying them innovatively to the context of SoS. BiGMTE, the tool presented in the paper, is based on this methodology and incorporates the Big Red graphical editor for creating and editing bigraphs and reaction rules, and the GMTE tool for graph matching and transformation. The process outlined in the paper involves five steps: creating BRS using Big Red, encoding the BRS into graphs, graph matching, graph encoding, and displaying the resulting bigraphs. To demonstrate the applicability and effectiveness of BiGMTE, the authors present an example of modeling an SoS using the tool. This example underlines the potential of BiGMTE in practical scenarios, showcasing its ability to model and manage the complexities of SoS through a formal and graphical approach.

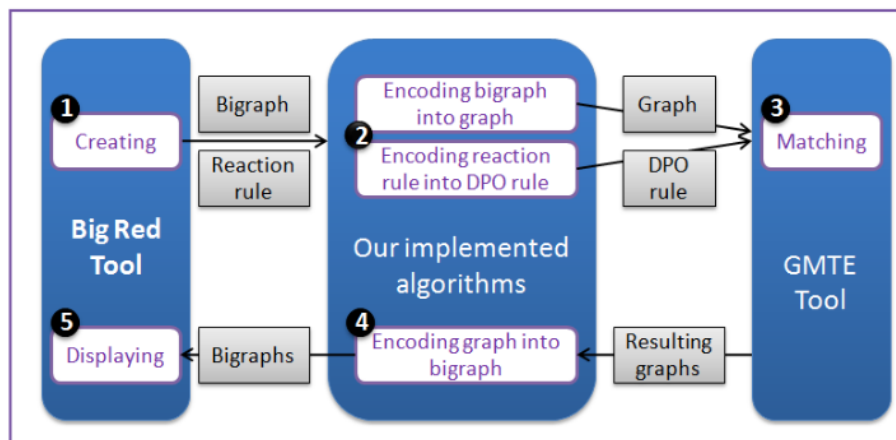


Figure III.13: *BiGMTE architecture* [29].

III.3.3 Synthesis

The collection of studies offers a variety of solutions for designing and analyzing SoS architectures, yet none fully encapsulate the entire SoSE process or the critical conceptual analysis phase. This phase is essential for comprehending high-level SoS requirements, understanding the relationships and interdependencies among CSs, and ensuring effective mission capability. More specifically, the research ranges from detailing the SoS as a whole, focusing on structural and organizational dynamics [59] [79], to tackling specific complexities such as security, safety, interactions, and adaptability [27] [31]. Other studies also address interoperability and emergent behaviors within SoS [83] and consider the evolution within software-intensive CSs [47].

Some papers explore the challenges of integrating various heterogeneous systems and their independent operations within an SoS [71] [5] [4], while others focus on expressing SoS at different abstraction levels to cover a range of SoS aspects [8] [43]. Additional techniques spotlight the behavior of adaptation and evolutionary architectures [67] [66] [65] and the dynamic execution of systems [60]. Some research is directed at mission-critical properties with an emphasis on simulation and verification through statistical model checking [80] [35]. Moreover, approaches like multiscale modeling ensure the correctness of SoS architectures [29], and frameworks with multi-ontologies offer flexible integration architectures [89] [35].

Table III.2: CHARACTERISTIC TABLE OF THE STUDIED APPROACHES.

		Semi-Formal			Formal			
		UML	SySML	DSML	ADL	BRS	π - Calculus	Maude
		[43][31]	[59][47][71][8][35]	[27][89] [5][4]	[60][66][65]	[30][83][29]	[67][80]	[79]
Framework	Processes	+/-	+/-	+/-	-	-	-	-
	Models/diag	+	+	+	+	+	+/-	+
	Viewpoints	-	+	-	-	-	-	-
	Stakeholder	-	+/-	-	-	-	-	-
	Standard	-	+/-	-	-	-	-	-
Composition	Expressivity	+	+	+	+/-	+/-	+/-	+
	Organization	+/-	+/-	+/-	+/-	+/-	+/-	+
Qtt prop	Time	-	-	-	-	-	-	-
	Resources	-	-	-	-	-	-	-
Behavior	RT-Control	-	-	-	-	-	-	-
	Desired/Undesired	-	-	-	-	-	-	-
	Design	+/-	+	+	+/-	+/-	+/-	+/-
	Runtime	-	+/-	-	+/-	+	-	+/-
Techniques	Executability	-	-	-	+/-	+	+	+
	Simulation	+/-	+/-	+/-	+/-	+/-	+/-	+/-
	Verification	-	-	-	+/-	+	+/-	+

+ : aspect taken into account; +/- : relatively considered aspect; - : aspect not taken into account;

To synthesize and discuss the contributions and shortcomings of the various cited works, we identify several criteria and characteristics that served as a basis for the critical study of attempts to address the issue of specification and design of SoSs. These include the formalism (semiformal formal and hybrid) adopted, the modeling of the structural framework of SoSs (development process, Models, Viewpoints of stakeholders, standards, etc.) used, the expressed entities (CS, Roles, Missions,

Capabilities, Links), the organizational composition, the methods of RT-Control and management provided, and the consideration of resources allocation mechanisms and temporal constraints during Design/Runtime. From the perspective of simulation, verification, and validating the introduced approaches, we focus on the formal verification technique used, as well as the approach followed for quantitative execution and validation. Table III.2 summarizes the analysis of these criteria in the context of the studied works.

Semiformal approaches:

From an architecture framework and composition perspective, we note that the examined methodologies present diverse yet complementary solutions. For the comprehensive frameworks in SoSs and their architectural viewpoints, the papers by [59] and [4] address SoS architecture, covering architecture, evolution, viewpoints, and processes. Other studies like [8], [31] adopt an MDA-based approach, presenting their solutions at varying levels of abstraction. However, there's less emphasis on specific standards that could ensure a systematic, reusable approach. These standards provide stakeholders with frameworks for managing viewpoints and lifecycle processes of systems, including detailed guidelines for all stages of a system's lifecycle, encompassing planning, execution, and monitoring. Concerning the static structural aspects, the papers by [5] [8] [31] describe entities like CSs, roles, missions, and goals in a detailed and expressive manner, focusing on organizational composition. These are typically demonstrated through interactions, interfaces, or ports to enable hierarchical entity composition. Nevertheless, no paper provides a unified specification aligning with a framework for the lifecycle processes of systems, which includes aspects like conception, design, development, production, operation, maintenance, and disposal. From simulation and validation perspective, the reviewed semiformal papers lack comprehensive executable support and verification tools to bring their theoretical abstract models to practical application. Moreover, there's no clear tool for monitoring the operational flow of the executing CSs, missions or tasks in an SoS. Neither to simulate their dynamic behavior. These works' shortfall extends to the control of execution, where there's no clear strategy for verifying the correctness of the proprieties.

Formal approaches:

Formal methodologies offer rigorous and mathematical solutions for addressing the complexity inherent in SoS architectures and behaviors. Notably, papers employing ArchSoS, mKAOS, Bi-graphical Reactive Systems (BRS), pi-Calculus, and Maude introduce considerable precision and analytical studies, facilitating the modeling, verification, and analysis of SoS with an emphasis on safety-critical and dynamic aspects. These methodologies distinguish themselves by their capacity to formalize and verify complex interactions and emergent behaviors within SoS, offering a structured approach to understanding and managing these systems. For instance, approaches like [79] and mKAOS [80] provide frameworks for capturing the hierarchical and dynamic nature of SoS, including the ability to model and verify architectural integrity and mission compliance across various system configurations. The use of formal languages and operational semantics enables the explicit representation of system behaviors and interactions [29], [59], [83], ensuring a robust foundation for analyzing system properties and behaviors. This is particularly evident in the application of [65]–[67], which emphasize executable models and formal verification, supporting the analysis of system

dynamics and the validation of system designs against specified requirements.

However, despite their strengths, formal approaches also exhibit limitations in fully addressing the practical aspects of SoS engineering. Similar to the semiformal approaches, there is a relative consideration of executability, simulation, and verification. While formal methods provide theoretical foundations for system specification and property verification, the integration of these methods into practical engineering workflows, including runtime monitoring and adaptive control and management mechanisms, remains a challenge. Furthermore, the aspects of time resources, real-time control, and differentiation between desired and undesired behaviors during runtime are not comprehensively covered.

The analysis underscores a gap in the current state of formal methodologies regarding the holistic management of SoS across their lifecycle, from design and development through to operation and evolution. There is a need for advancing formal approaches to incorporate more practical tools and techniques for simulation, runtime adaptability, and the management of temporal and resource constraints. Bridging these gaps would enhance the ability of formal methodologies to not only model and verify SoS architectures and behaviors rigorously but also to support the dynamic and evolving nature of SoS operations in real-world contexts.

III.4 Conclusion

In the first part of this chapter, we have presented the context in which the manuscript topic is inscribed. Firstly, we have presented the SoSs and the major definitions, dimensions, categories, application domains, etc. Then, we have presented various works related to the engineering and the modeling in SoSs found in the literature. Initially, we studied some semiformal models mainly designed for static and structural aspects of SoSs. Next, we examined some formal approaches and models that have proposed solutions for the specification and verification of behavioral aspects in the SoSs. We analyzed all the presented models and approaches based on a set of criteria and priorities that we deemed relevant, and we have highlighted their contributions and shortcomings and to prepare the reader to properly contextualize our contributions.

Chapter IV

Methodology and General Principle

Contents

IV.1	Introduction	57
IV.2	A process for SoS Engineering	58
IV.2.1	Domain Engineering	60
IV.2.2	Application Engineering	62
IV.3	Solution principle	64
IV.3.1	Basic elements description	64
IV.3.2	MDA-based SoS Framework design	65
IV.3.3	Formal semantics of centralized control	65
IV.3.4	Formal specification of management strategies	66
IV.3.5	Autonomic execution and verification	67
IV.4	Conclusion	68

IV.1 Introduction

In SoSE, the necessity for methodologies and frameworks that can dynamically tailor their structural and behavioral aspects is essential. This need arises from the inherent complexity and changing nature of SoS environments, where execution, environmental changes, and dynamic mission priorities continually redefine various requirements. Therefore, the complexity of these aspects makes designing SoSs particularly challenging. From the perspective of specification, verification, and validation, we consider four research objectives:

- **RO1**: Overcome the lack of standards for the basic elements to describe the structure and behaviors of SoSs.

- **RO2:** Define a multi-viewpoints-based Architecture Framework to support SoSs' development.
- **RO3:** Define behaviors related to quantitative aspects such as time and resources of SoSs and their Control.
- **RO4:** Ensure strategic management of the execution of these different missions in SoSs.
- **RO5:** Ensure the autonomous executability of the desired behaviors within SoSs and verify their correctness.

In this chapter, we explore how SoSE principles can be applied to Domain Engineering and Application Engineering (DE and AE, respectively) to overcome the defined challenges. On the one hand, the DE focuses on specifying and creating reusable, controllable and manageable assets tailored for SoSs. We define these assets by SoSs commonalities and variabilities to form the structural core of the SoS architecture and allow for dynamic control and management at various development processes. consequently, AE deals with the practical initiation and customization of the concrete assets into specific SoS configurations. This process involves the strategic use of the assets developed in the DE phase to create SoSs that are not only fit for their initial purpose but also capable of evolving in response to changing functional needs, conditions and requirements.

Moreover, adopting a complementary set of assets into SoSE effectively addresses several critical challenges, facilitating the development, control and management of SoSs at both design time and runtime. The proposed approach ensures that SoSs are not only built with design time requirements but are also capable of in-situ control, reflecting the dynamic nature of their operational environments. Thus, by combining the strategic reuse and variability management inherent in various SoSs categories with the complexity and controllability demands of SoS.

IV.2 A process for SoS Engineering

The bellow Figure. [IV.1](#) illustrates the proposed approach in SoSE that inspires from the Software Product Line principles to enhance the development lifecycle of SoS architectures. The approach starts with identifying various SoS requirements, which then transition into the phase of Dynamic SoS Engineering. The latter is employed to develop a consistent set of SoS assets that will serve as a basis for both the initial design and the runtime management of the SoS [[14](#)], [[68](#)], [[84](#)].

- **Dynamic SoS Engineering:** This process identifies the requirements based on the both DE and AE principles of reuse, variability, and commonalities, it ensures that the engineering process.
- **Creation of SoS Assets:** From the previous step, we develop SoS assets which assets represent the tangible output from applying DE and AE principles.
- **Application Requirements:** These requirements are taken into consideration within SoSs. This is where the SoS assets are tailored to meet the particular needs of a given application and inform the Design and Runtime Control phase to result in a functional SoS.

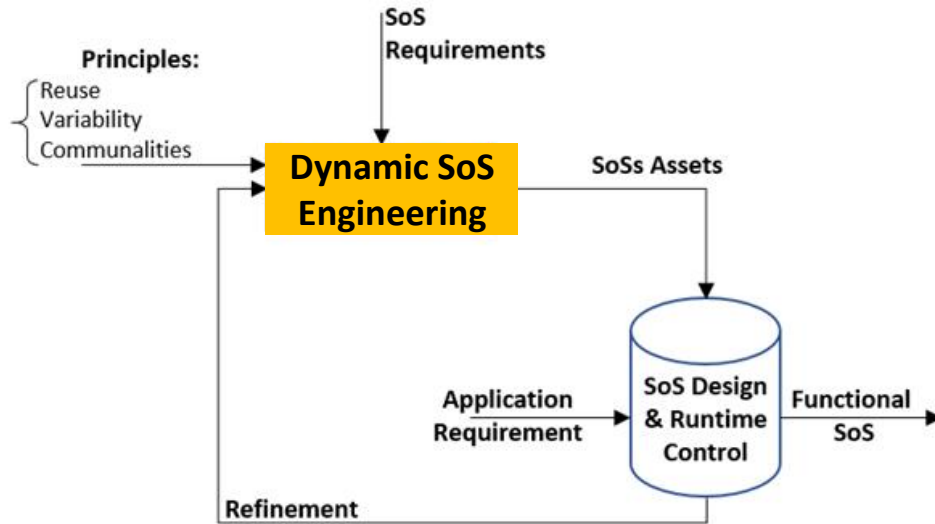


Figure IV.1: Evolutionary process for dynamic SoSE.

- **Refinement Loop:** This represents SoSs Design and Runtime Control phases as a cyclical progression where the functional SoS is continually refined based on performance quantitative features, evolving requirements, and stakeholder feedback.
- **Functional SoS:** The result would be a functional SoS managed and controlled based on DE and AE principles and application-specific features. This process allows for dynamic evolution as it encounters new requirements and any unpredictable/undesired behavior.

As shown in Figure. IV.1, the approach is supported by an iterative process that combines design-time decisions with runtime execution. This iterative nature allows for continuous refinement and enhancement of both the design and functional aspects of SoS. Feedback from runtime experiences is fed back into the design process, enabling stakeholders to make informed adjustments that are aligned with concrete models and real-world conditions

In the rest of this section, we present a general overview of the approach's architecture and practical implementation, tackling the inherent challenges of SoSE. Figure. IV.2 structures the approach according to the two domains DE and AE. On the one hand, the DE dimension forms the foundational layer, focusing on the development of generic abstract assets which are composed of various Meta-Models and their models that interact through different extensions and inheritances. The Meta-Models define standardized assets across four domains: Framework, Variability and Platform to encapsulate different commonalities and variabilities within the core, aspects, and features of an SoS.

The AE dimension on the other hand leverages the iterative nature of the approach between the Design and Runtime phases, i.e. the Design Time phase focuses on setting up the initial SoS configuration, tailored to specific characteristics and stakeholder viewpoints, and building initial configurations for functional integration. The Runtime phase defines the dynamic behavior of the SoS for effective self-management and control during operation. It includes runtime behavioral

modeling, capturing system responses under various conditions. This phase is supported by an autonomic MAPE-K loop control, enabling continuous refinement and improvement of both design and runtime execution.

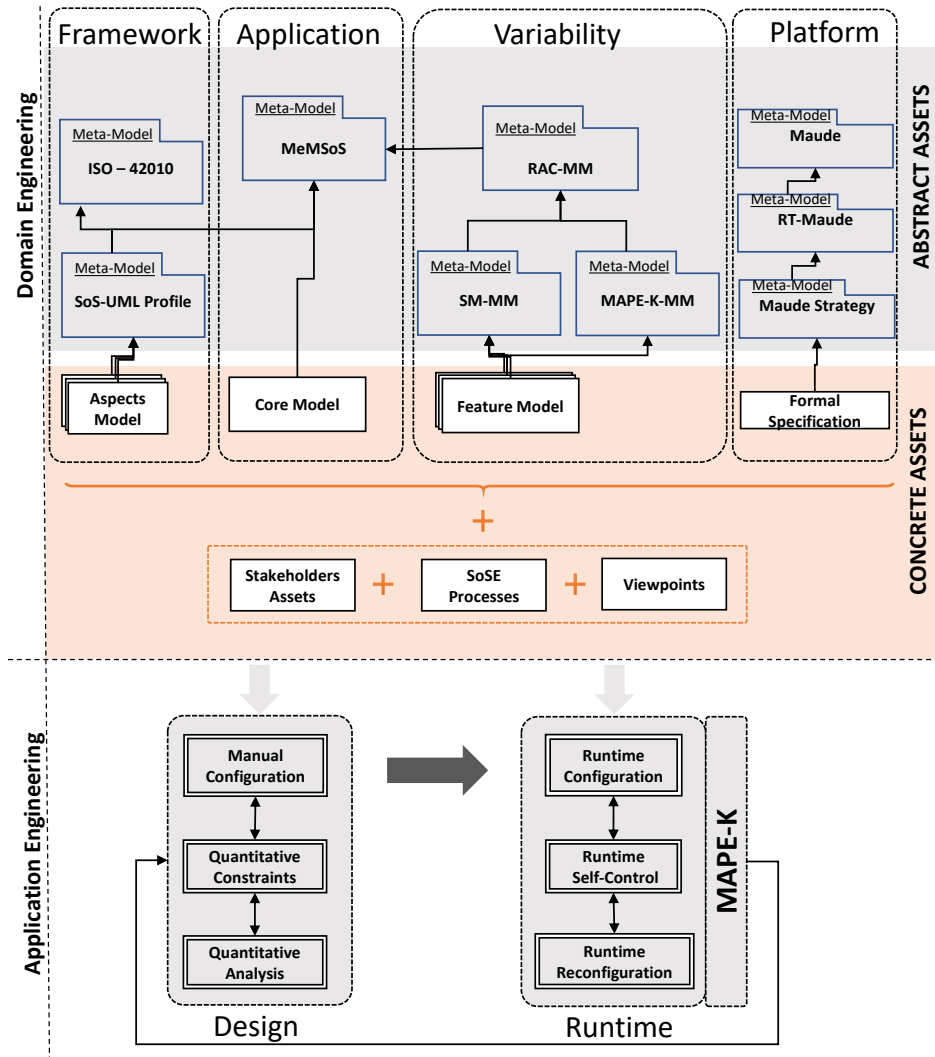


Figure IV.2: General overview of the approach.

IV.2.1 Domain Engineering

Adopting an MDA framework, the upper part of Figure IV.2 outlines the DE process for SoS products, focusing on the creation of assets that are essential in building SoS. The DE process involves generating two types of assets: abstract and concrete. The abstract assets, primarily comprising Meta-Models, provide the foundational basis necessary for developing concrete assets. The latter includes elements such as the architecture framework, control and management models, stakeholders' viewpoints, and processes. This relationship among these assets plays a crucial role in the SoS instantiation phase, where concrete assets are applied in the construction of SoS products.

Following the proposed approach, we define four different categories of domains, which are:

- **SoS Application domain:** The abstract assets in this domain present the core architecture which is defined by a Meta-Model called as Met-Model for SoSs (MeMSoS), it incorporates the commonalities that are fundamental to the design of the four multifaceted SoS categories (knowledge, directed, virtual, and collaborative). These commonalities ensure consistent characteristics across all CSs within SoSs, forming the standardized base from which the architecture is developed. Moreover, MeMSoS provides an architecture that helps stakeholders understand SoS challenges, offering a framework for potential solutions with a highly abstract architectural design. It is specifically adequate at outlining the interactions between different architectural entities, capturing the complexity inherent in such a system. Furthermore, the core domain leverages MDA to facilitate the enhancement and recognition of new entities within its structure, such as those in the Framework and Variabilities domains i.e. it contributes to the extension and refinement of the core for other purposes and objectives.
- **Framework domain:** The SoS framework domain is modeled by integrating an extended version of both MeMSoS and ISO-42010 Meta-Models to establish a comprehensive architecture framework. The latter is managed using a UML profile specifically designed to address the common aspects of the four types of SoS—knowledge, directed, virtual, and collaborative—without delving into their specific variabilities. It encompasses the essential aspects of MeMSoS to facilitate the identification and design of various architectural viewpoints that resonate with the diverse concerns of stakeholders. Utilizing well-defined software development processes, the concrete assets effectively manage diagrams from a developed SoS-UML profile to ensure the coherence of the architecture. Moreover, the framework establishes a set of Model Kinds within to support the modeling of SoS entities, covering both static, dynamic and requirement aspects. These are specifically tailored to address particular concerns of different stakeholders, reflecting the common characteristics shared across the SoS types.
- **Variability domain:** This domain focuses on the variabilities of a specific type of SoSs which are Real-Time SoSs (RT-SoSs). The latter is particularly crucial in Directed SoSs where centralized control and management are key. This is specified in the Resources Allocation Control Meta-Model RAC-MM and its specializations Meta-Model Management Strategies Meta-Model (MS-MM). More specifically, these Meta-Models are designed by feature assets, outlining the main concepts that capture the essential aspects addressing the quantitative features-oriented control and management. This is vital for ensuring optimal system performance in dynamic and evolving environments. Moreover, The strength of the Features in the variabilities domain lies in its ability to seamlessly integrate into the systematic architectural framework for SoSs at Design time/Runtime.

Additionally, it maintains a clear separation between the SoS Application and Framework domains. This distinction, particularly between the architectural Model Kind and the logic of when and how to control, allows for the reuse of the control techniques in a straightforward

manner. This integration on managing quantitative aspects directly addresses the variabilities inherent in Direct SoSs, demonstrating the Meta-Model’s adaptability to different RT-SoS while considering the main commonalities of the other multifaced SoSs.

- **Platform domain:** We chose the Maude language and its extensions as our formal tool environment due to its expressiveness, flexibility, and suitability for our requirements. Maude’s foundation ensures that Application, Framework, and variability Meta-Models can be mapped into executable specifications without information loss. This allows us to leverage the graphical benefits of MDA and Maude’s high-performance tool set, especially its conditional rewriting rules, strategies and model-checker engine.

We use Maude Language as the target platform. The latter alongside its extensions (RT-Maude and Maude Strategy Language) propose a formal framework to define an operational semantic for the high abstract Meta-Models, and focusing at the same time on SoSs control and management with a focus on different quantitative proprieties and constraints. It highlights how the different Meta-Models semantics are implemented using the Maude language to simulate and analyze SoSs behaviors.

The motivation behind this choice relates to the capabilities of the platform to match the dynamic processes between DE and AE, this integration outlines a framework for representing Maude specifications within MDA. The latter includes constraints and operations that align with the Maude language, providing a structured approach to formalize models and Meta-Models. The constraints ensure adherence to the language specifics (seen in Chapter II), such as no rewrite conditions for membership axioms, equations, and functional modules without rules [73].

As seen in the Maude Meta-Models, they facilitate the definition of the operational semantics of the Application, Frameworks and Variability Meta-Models by implementing and formalizing the infrastructure necessary to integrate all of them into a dynamic Domain Engineering approach. For a detailed examination of the Maude Meta-Models, its constraints, etc. readers are referred to [11], [57], [73].

For a complete understanding of the architectural model and the languages used to define core models, section II.3 explores the specifics of these abstract and concrete assets, explaining their roles and functionalities in SoS architecture. This section also offers elaborate insights into the MeMSoS, its extensions, ISO-42010 Meta-Model, RAC-MM, and platforms Meta-Models, all of which are essential for defining applications and establishing a harmonized architecture.

IV.2.2 Application Engineering

Concrete assets act as a vital link between the two domains, encompassing models and instantiations that leverage Meta-Models to encapsulate the commonalities and variabilities of SoSs products. This includes the Framework Aspects Model, which details different SoSs’ Aspects, and the Application Model, which outlines the commonalities and the core of SoSs products. When combined

with the Stakeholder framework, SoS Engineering Processes, and Viewpoints, concrete assets support a holistic and layered approach to SoS Engineering. On the other hand, the Directed RT-SoS Meta-Models are articulated through the variability model, which are specified using Maude, and the Framework and Application Meta-Models are realized as concrete assets through graphical tools and result from model instantiating applied to application models and the SoS's unified architectural profile.

IV.2.2.1 Design Phase

As we transition to the AE phase, the abstract assets are instantiated to create initial configurations. The latter turn concrete models into specific, detailed models that capture the precise operational parameters, performance metrics, and behavioral attributes of the SoS. Stakeholders are thus equipped to convert abstract specifications into functional, operational models that fulfill the SoS's intended design and functionality. These instantiated models from the architecture framework and the features models represent the concrete assets and serve as the design time model of the SoS prepared for the configuration phase.

Here, the initial architecture and system parameters are established, based on a sequence of critical steps:

- **Model Instantiating** In the first step, concrete assets provided by the architecture framework are transformed using the SoS-UML profile into detailed models. These models are then initialized with the specific operational parameters and quantitative values that will define the SoS's architecture, setting the stage for the next phase of execution.
- **Collaborative Configuration** This step involves stakeholders' collaboration to align the instantiated models with the strategic objectives of the SoS. This teamwork ensures the system's configuration not only holds the commonalities across different four types of SoSs, but also meets unique application-specific requirements.
- **Validation and Refinement** The final step confirms that the configured models meet all system requirements. Through iterative refinement, informed by stakeholder feedback and testing, the SoS architecture is fine-tuned for deployment. The UML profile and the feature models thus act as a crucial intermediary between the concrete assets and the design phase. It ensures a smooth progression from DE to AE, culminating in a detailed and functional SoS architecture.

IV.2.2.2 Runtime Phase

Using the feature Meta-Models, this phase introduces a structured approach that encapsulates the variability domain in the approach. To execute missions consuming resources, simulate their behavior, and model-check their logical properties, we employ the Maude language to offer a framework for modeling and analyzing SoSs configuration and execution which has three main objectives:

- Firstly, we explore a detailed approach of modeling and analyzing local and global resources within the SoSs. It explores the formal semantics of the RAC-MM Meta-Model to emphasize the integration of quantitative priorities and operational semantics to enhance centralized management and efficiency in real-time Directed SoS.
- Secondly, we focus on proposing a strategic management approach, addressing both desired and undesired behaviors in Directed SoS. It introduces a comprehensive management framework that encompasses workflow descriptions and functional chain execution. The approach manages mission execution and resource allocation through strategic planning, highlighting the importance of the defined auxiliary and main strategies.
- Thirdly, we show the application of RT-Self-Regulating mechanisms via a MAPE-K loop, emphasizing the significance of knowledge phases, monitoring, analysis, and strategic management in a running SoS.

IV.3 Solution principle

SoSs evolve in a particularly dynamic and variable environment. This distinguishes them from other computing models due to a significant complexity in understanding and controlling their structures and behaviors. More specifically, the variability characteristics of a SoSs, in terms of centralized control, strategic management, and execution depend on the combination of many factors such as the amount of available resources, temporal constraints, the logic governing the desired behavior of the real-time SoSs, as well as various high-level policies and properties to be satisfied.

Our work aims to provide a combined use of semi-formal and formal modeling approaches to design and analysis SoSs commonalities and their variabilities, covering all development process phases, from specification to the verification of behaviors. In the current section, we offer an overview of our proposed solution to overcome the different research questions defined earlier, presenting its main phases, objectives, and contributions. This discussion aligns with the structure of the manuscript, which systematically distributes contributions across the chapters that follow.

IV.3.1 Basic elements description

It is worth to mentioning that the comprehensive set of Meta-Models developed in this approach directly addresses the gaps in standardized definitions for the foundational elements of SoSs. By defining the structure and behaviors of SoSs, these Meta-Models provide a unified framework that not only designs the architectural and operational structures of SoSs but also standardizes the representation of their complex interactions and functionalities. This attempt facilitates a clearer understanding and management of SoS complexities and lays the groundwork for establishing consensus-driven solutions for management and control. The Meta-Models defined in the following chapters address the absence of standardized frameworks for SoSs, and this directly targets the first research objective in this manuscript (**RO1**).

IV.3.2 MDA-based SoS Framework design

This phase aims to get a handle on the complexity of consistency and coherence between the different proprieties of SoSs commonalities i.e. viewpoints of different stakeholders, as well as the ability to reconcile and include all their viewpoints before proceeding to the various processes in the SoSE lifecycle. In this context, the phase leverages the Framework Domain assets to introduce a Multi-viewpoints-based Architecture Framework, that is associated with a set of SoSE development processes and a UML profile dedicated to the SoSs that can facilitate and improve the design of SoSs' architectures. The purpose of this phase is to present an Arichetctural Framework (AF) that intends to foster the systematic development of SoSs' architectures. This is achieved in three main steps as follows:

- The proposal of an AF: based on the concepts proposed by the standard “ISO/IEC/IEEE 42010:2011 Systems and Software Engineering-Architecture description”, we specialize and extend it according to SoSs commonalities. This is achieved by specifying a set of SoSE processes, concerns and viewpoints necessary to have a semantic consistency between the different parts of the SoS' architecture specified in the different viewpoints.
- Improving the AF with systematic processes: this consists of integrating systematic processes of development SoSs' architectures which can have a positive impact on the overall quality of the framework. The design of these processes is based on a consolidated Model-Based SoSE in which the involved processes can be seen as a sequence of connected and dependent activities.
- Defining an SoS-UML Profile: this phase consists of providing a large number of models to separately capture, describe and organize each of the processes of different viewpoints. The proposed profile is a package of new stereotypes' notations that are defined in the Meta-Model MeMSoS. These new notations will complete the list of UML notations by modeling explicitly and appropriately all structural and behavioral aspects of SoSs.

By establishing these steps, we create a multi-viewpoint Architecture Framework that is both comprehensible and easily manipulated by different stakeholders. This approach is designed to support the SoS stakeholders with the necessary tools to simplify the development of a multi-viewpoint, governed by innovative SoSE processes and documented via models in the SoS-UML profile. Consequently, this phase is directly aiming at achieving the second research objective (**RO2**).

IV.3.3 Formal semantics of centralized control

The phase aims at providing a formal description of a centralized control which encompasses the variabilities and abstract assets within the proposed methodology, particularly focusing on the Resource Allocation Controller (RAC) Meta-Model formalization. The primary aim is to address and formalize the complexities associated with directed SoSs resources through a structured and formal approach using the Maude language. This is achieved through three main steps:

- RAC variabilities' abstract assets: This step highlights a comprehensive structure with classes like State, Resource, and Role, each contributing to the dynamic behavior and management of resources within the SoS. The RAC is depicted as key to managing these variabilities, ensuring resources are allocated efficiently and missions are executed within their specific conditions.
- Formal semantics of static entities: This step defines the formal semantics and mappings between the RAC Meta-Model's static concepts and Maude constructs. It describes the fundamental static structures of Missions, Resources, Roles, etc. along with their quantitative properties. The focus is on defining these entities' structural aspects and integrating them within the Maude language to facilitate dynamic and autonomous control of resources.
- Entities lifecycle dynamics: This step defines the operational semantics of the dynamic behaviors and lifecycle of missions, resources, roles, etc. within the SoS. It describes how each entity transitions through various states, governed by predicates and actions encoded in Maude. The lifecycle analysis covers the stages from creation, activation, execution, to completion or renewal, outlining the formal specifications for managing these transitions effectively.
- RAC's lifecycle: The step defines the operational semantics of RAC's lifecycle, illustrating its pivotal role in coordinating the states of missions, roles, and resources. It presents a sequence of states that the RAC transitions through, from initial trigger to processing requests and managing consumption or production. The RAC operates as an intermediary manager, employing predicates and actions to ensure efficient resource allocation and system balance.

By leveraging the Maude language which offers a robust framework for centralized control, addressing the challenges of resource allocation, mission execution, and dynamic behavior management in SoSs, this phase specifically targets the third research objective (**RO3**).

IV.3.4 Formal specification of management strategies

Another objective of the variability Domain in the proposed methodology is to introduce a comprehensive formal approach for enhancing management and planification of the executing missions in Directed RT-SoS. The approach highlights the integration of dynamic strategies to manage both desired and undesired behaviors utilizing the Maude Strategy language to define operational semantics across functional system and strategy modules. This method addresses the complexities of workflow management, mission execution, resource allocation, and conflict resolution, aiming to improve system functionality and resource efficiency in complex SoS environments. For this end, the approach is accomplished via three principal stages:

- Management strategies Meta-Model (MS-MM) design: Based on the previous RAC specification, this phase creates the new MS-MM for strategies. It structures the complex interactions between various CS, focusing on managing workflows within SoSs and designing behavior rules that describe different desired and unwanted behaviors within the system.

- Strategies formalization: This phase defines the formalization of self-management strategies using Maude, dividing the framework into functional modules, system modules, and strategy modules. These modules formalize the semantics of MS-MMSoS, including the specification of mathematical logic, rewriting rules for system behavior evolution, and strategies for guiding the application of these rules. The integration of these modules introduces five complementary modules that collectively cover the semantics of MS-MMSoS, focusing on workflow and functional chains, time-based and resource-based execution, and strategic management. Practically, the phase explores managing workflows in SoSs, and highlights the complexity for advanced coordination and resource management. It describes compositional nature of functional chains, the strategic mission execution, and resource management required to navigate SoS complexities. For this end, a set of auxiliary strategies are integrated into a set of main strategies to manage system operations, ensuring timely mission completion with efficient resource utilization.
- Real-Time regulating mechanism: Using the MAPE-K Loop control, this step aims at introducing an integrated approach that employs a MAPE-K control loop for the effective management of temporal aspects and optimization of system behavior. It describes the knowledge phase, monitoring, analysis, workflow analysis, and planning and execution phases, focusing on the real-time adjustment and synchronization of missions based on dependencies and operational semantics.

Integrating these steps, the formal methodology facilitates the management of both desired and undesired behaviors in Directed SoSs, highlighting the importance of dynamic strategies and the MAPE-K loop in securing the efficient and effective functioning of complex SoS environments. Consequently, this phase is explicitly designed to address the challenges associated with the fourth research objective (**RO4**).

IV.3.5 Autonomic execution and verification

The purpose of this phase is demonstrating the practical application and validation of the proposed formal approaches through runtime simulation, analysis, and verification. It defines the specified behaviors of the components, the RAC, Management strategies and Real-Time Self-Regulating mechanism. This means verifying that autonomously executed behaviors and actions are well-defined, correct, and describe a desirable behavior at design time and runtime stages. This involves defining a number of quantitative properties that the system must satisfy and guarantee throughout its runtime.

Specifically, the behavior of the RAC controller must allow: (1) the dynamic allocation of resources in real-time to meet the changing demands of the CSs and their missions, and (2) managing the life cycle of each resource, and ensuring optimal utilization to avoid bottlenecks and resource conflicts. On the other hand, the management strategies must :(1) prioritize resource allocation based on the criticality of missions, availability of resources, and predefined temporal constraints,

(2) govern the desired behavior and outcomes of CSs during mission execution, and (3) avoid any any unwanted behavior related to time and resource constraints of the SoS.

To this end, the phase addresses the formal verification of SoSs' behavior using Maude's model-checking capabilities. It verifies that the system meets its strategic objectives and operates as expected under various emergency scenarios. This involves checking invariants and using the search command in Maude to explore possible system behaviors and ensure compliance with defined properties. Therefore, by validating the effectiveness of the proposed system design, strategic planning, and control mechanisms in optimizing emergency response operations, this phase addresses the fourth research objective (**OR5**).

IV.4 Conclusion

In this chapter, we have briefly introduced our combined methodology which is based on two main assets of domain and application engineering leading to a robust model for SoSE, and improving the life cycle of SoS architectures. The methodology firstly is based on the adoption of an MDA framework that introduces a suite of comprehensive Meta-Models that are essential to the design, control, and management of SoSs. Secondly, the proposed methodology encompasses formal semantics for these Meta-Models using the Maude language and its extensions, which allows for a formal specification and verification of behaviors, ensuring that the system can dynamically evolve and adapt to new requirements and unpredictable behaviors. As we move forward in this manuscript, the different steps of the methodology are introduced in the next chapters, which will explore the specifics of the approach's architecture and practical implementation.

Chapter V

Model-Based SoS Framework Domain

Contents

V.1	Introduction	69
V.2	SoSs Commonalities	70
V.2.1	Abstract assets: Application and Framework Domains	71
V.2.2	A Unified Architecture Framework: Overview	77
V.2.3	Case study: Aircraft Emergency Response SoS	79
V.3	A Multi Viewpoints-based Architecture Framework	81
V.3.1	Architectural concepts	82
V.3.2	Associations	86
V.4	UML Extensions for Modeling the unified architecture	88
V.4.1	SoS_Knowledge_Package	89
V.4.2	CSs_Selection_Package	91
V.4.3	Conceptual_Design_Package	93
V.4.4	Architectural_Design_Package	94
V.4.5	Interaction_Package	95
V.4.6	Integration_Deployment_Package	97
V.5	Conclusion	99

V.1 Introduction

The early-stage design decisions in SoSE, underlined by the necessity of Domain Engineering for defining reusable assets, seamlessly align with the challenges addressed by the Domain Engineering Framework Domain. By employing Architectural Frameworks (AF) that prioritize Architectural Viewpoints, we provide a comprehensive yet modular approach to SoS design. This strategy, essential

for managing complex interrelations in SoS, mirrors our aim to address SoS development challenges through a structured, systematic framework. The proposed AF not only simplifies the development process but also ensures a holistic understanding of SoS, effectively managing its inherent complexity and diverse stakeholder needs.

V.2 SoSs Commonalities

As seen in Chapter IV, DE focuses on the specification and realization of reusable assets, which include architectures, requirements, etc. By employing Framework Domain techniques, we can specify the commonalities across the SoSs domain, thereby easing the development process of these complex SoSs. Moreover, AFs are an emerging discipline in Software Engineering (SE) that regards Architectural Viewpoints as first-class entities. These viewpoints have opened new avenues for system representation, offering a comprehensive yet modular way to design complex systems like SoSs. In the context of SoSs, we argue that leveraging the Framework Domain provided by DE and AE approach will not only simplify the development process but also yield a system of higher understanding. It's worth noting that a single comprehensive viewpoint of an SoS's architecture is often too complicated to be fully understood or communicated, encapsulating numerous relationships between various business, structural, and behavioral aspects. Therefore, we aim to represent SoS architectures through multiple architectural viewpoints, which collectively form a unified AF tailored for SoS, informed and enriched by Domain Engineering practices.

This leads us to a focused examination of how the AF specifically addresses the following main key points:

- **Understanding and Mastering Complexity:** The Framework Domain aids in comprehending and managing the complexity of SoS architecture. This understanding is vital for architects to master SoS complexity, offering significant advantages in the design phase.
- **Static and behavioral Aspects:** The Framework Domain effectively distinguishes and addresses various static/structural and dynamic/behavioral aspects within SoS. This is crucial for managing SoS characteristics, quality attributes, and overseeing the architecture, design, and implementation.
- **Development Process Adaptation:** The AF adapts to the unique development process of SoS, which differs from standard systems due to the interdependencies of CSs. This adaptation guides the SoS lifecycle from requirements to software implementation, coordinating various development processes.
- **Architectural Design Decisions:** Architectural decisions significantly impact the fulfillment of SoS functional and quality requirements. The Framework Domain guides these decisions, helping stakeholders to identify and address specific functional and non-functional characteristics.

V.2.1 Abstract assets: Application and Framework Domains

The domain explores the primary Meta-Models that contribute to the establishment of a unified architecture framework for SoSs. It begins with the extension of IEEE ISO-42010 Meta-Model, which offers a standardized description framework for SoSs architecture. Building on this standard, the section then introduces another extended version of the MeMSoS (introduced in Chapter IV), which incorporates specific adaptations for the different aspects of SoS. This extended model allows for a more refined approach to SoSE. Together, these models highlight the commonalities of the SoSE approach, emphasizing reusable and adaptable design paradigms essential for SoSE.

In SoSE, MDA is critical in facilitating the creation of a robust framework for SoS development. MDA methods guide DE to construct a layered hierarchy of models, streamlining stakeholder concerns within the SoS architecture. These MDA principles, standardized by the Object Management Group (OMG), outline four universal abstraction layers applicable to domain applications [21]:

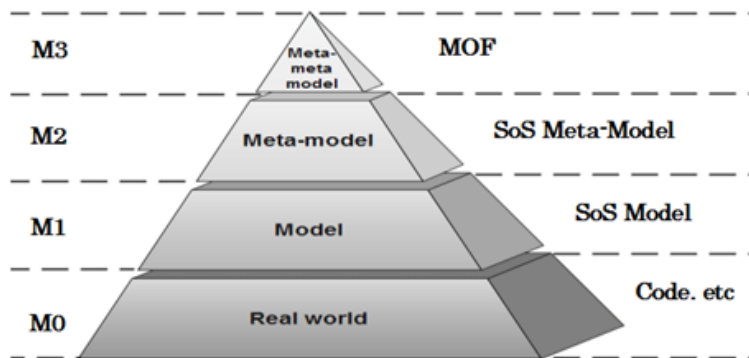


Figure V.1: *Four-layer Meta-Modelling infrastructure of OMG.*

- **M3 Layer (Meta-Meta-Model Layer):** This layer houses the MOF, providing the foundational language for Meta-Meta-Modeling that defines the structure for all subsequent layers.
- **M2 Layer (Meta-Model Layer):** Here, we introduce a Meta-Model that aligns with MOF standards, specifically tailored for SoS framework.
- **M1 Layer (Model Layer):** The Model conforms to the proposed Meta-Model and represents this various SoS model, i.e. this model articulates the SoS's architecture and serves as an instance for SoS design.
- **M0 Layer (Implementation Layer):** The operational layer represents the lower level where the modeled SoS is executed, transforming high-level abstractions into practical and executable code.

The MeMSoS, as described in the UML diagram below FIGURE V.2, serves as a comprehensive framework for the architectural characterization and design of diverse SoS, including knowledge, directed, virtual, and collaborative types. MeMSoS is designed to encapsulate the shared characteristics across all CSs, ensuring a standardized, evolutionary approach to SoS development. Central

to this approach is the hierarchical composition of SoS, where each CS, equipped with distinct missions, capabilities, and roles, collaborates to fulfill both local and overarching SoS goals, facilitated by effective communication channels [21].

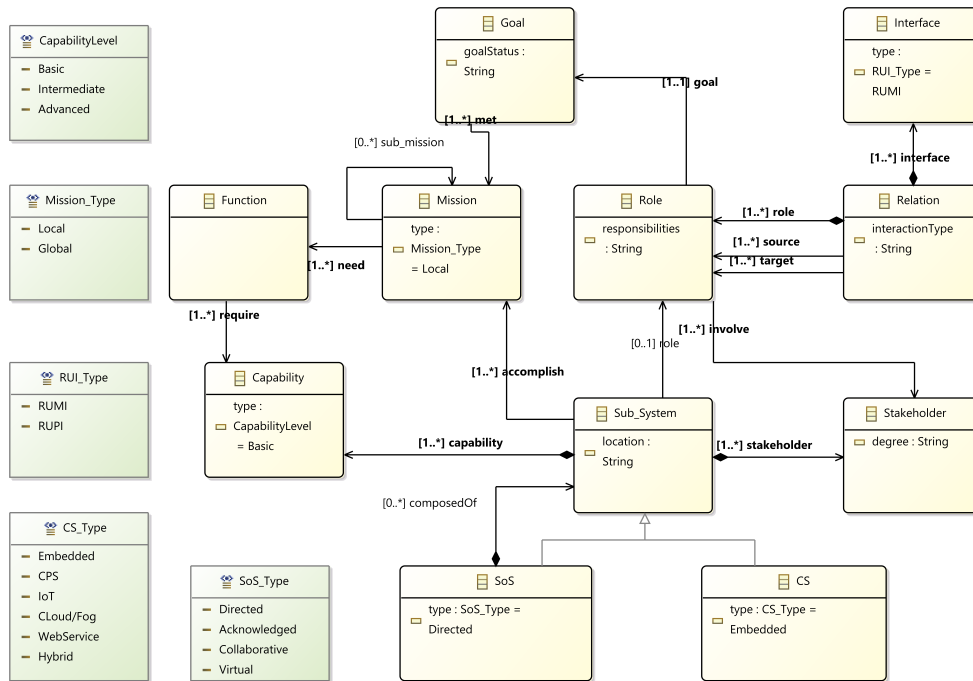


Figure V.2: General overview of MeMSoS.

Definition V.1 (SoS). : Defined within the scope of MDA, an SoS is formalized as a tuple $SoS = (CS, G, R, M, C, I, S, L)$, where:

- $CS = \{cs_1, cs_2, \dots, cs_n\}$: A finite set of CSs, each an autonomous system characterized by specific missions, capabilities, and roles, capable of operating as an independent SoS or as an atomic unit within a larger SoS.
- $G = \{g_1, g_2, \dots, g_k\}$: A finite set of goals representing the local objectives of individual CSs and the collective aims of the SoS.
- $R = \{r_1, r_2, \dots, r_p\}$: A finite set of roles defining the expected behaviors of CSs, each role aggregating necessary capabilities to achieve the SoS's global goals.
- $M = \{m_1, m_2, \dots, m_o\}$: The missions assigned to each CS or the SoS, comprising specific tasks or objectives to be accomplished.
- $C = \{c_1, c_2, \dots, c_i\}$: A finite set of capabilities representing the functions each CS can perform within their designated role.
- I : a finite set of interfaces facilitating communication, including both Message Interfaces and Physical Interfaces (RUMI and RUPI, respectively) associated with each role.

- $S = \{s_1, s_2, \dots, s_q\}$: A finite set of interdisciplinary stakeholders, individuals or groups with interests in the CS's capabilities.
- $L = \{l_1, l_2, \dots, l_u\}$: one or more dynamic links representing the potential interactions among different CSs, roles, goals, and capabilities, etc.

We highlight in this model the importance of different entities that we can find in an SoS such as roles, capabilities, and goals, structured to enable efficient sharing and reuse of system elements. This would reduce the effort and enhance adaptability to achieve collective SoS goals. The UML diagram illustrates these relationships and establishes a visual grammar that represents the system's hierarchical organization, goals, roles' interactions, and interoperability, ensuring system coherence and functional adequacy.

Moreover, MeMSoS recognizes new elements within the SoS structure, particularly in the Framework and Variabilities domains, allowing for the continuous extension and refinement of SoS architecture. Adhering to the M1 and M2 levels of the OMG Meta-Modelling infrastructure, MeMSoS offers a high-level abstraction for SoS features, positioning itself as a strategic plan for SoS engineering. This facilitates the mapping and transformation processes essential for a model derivation that respects the nuanced Meta-Model definition. The result is a flexible yet systematic framework that guides stakeholders from abstract SoS concepts to tangible system implementations, embodying a harmonious and mission-oriented architecture.

The subsequent sections will explore the Meta-Model layers, exploring their respective roles in the DE process. This exploration will include a detailed look at how Meta-Models such as IEEE ISO-42010, RAC-MM, and MS-MM capture the commonalities and variability of SoS architectures and the formal specification of these assets across various platforms, with a specific focus on Maude.

V.2.1.1 IEEE ISO-42010 Meta-Model adaptation

In the Framework Domain, we extend the Meta-Model of IEEE ISO-42010 introduced in Chapter II, Section II.2.2 to suit the specific needs of SoSE. This extension is a critical evolution of the standard, allowing for a more systematic approach to SoSE processes and models. Moreover, it enhances the Architecture Frameworks (AF) with the versatility to address the complexities of architecting large-scale, interoperable systems.

Therefore, AFs, serving as foundational mechanisms, adopt a uniform approach to the creation, interpretation, analysis, and application of architecture descriptions within a particular domain. The extensions to the Framework Domain facilitate this by ensuring coherence and traceability across diverse architectural viewpoints and the various concerns they encompass. Such frameworks are crucial not only for generating architecture descriptions but also for formulating modeling tools, methods, and processes that enhance communication and cooperation across multiple projects and organizations.

More specifically, the enhanced standard is structured around two pivotal classes—'SoSE Process' and 'SoS-UML Profile'—as indicated in red on the diagram. The 'SoSE Process' class is crucial

for capturing the diverse concerns identified by stakeholders and is defined by the 'Model Kind' it employs. This class orchestrates the various SoSE activities, ensuring systems are robust and adaptable. The 'SoS-UML Profile' class extends the standard UML profile to fit the SoS domain, providing stereotypes and notation rules essential for accurately modeling SoS architectures. This profile aligns architectural viewpoints with SoSE best practices, as dictated by the 'Model Kind' [20], [22].

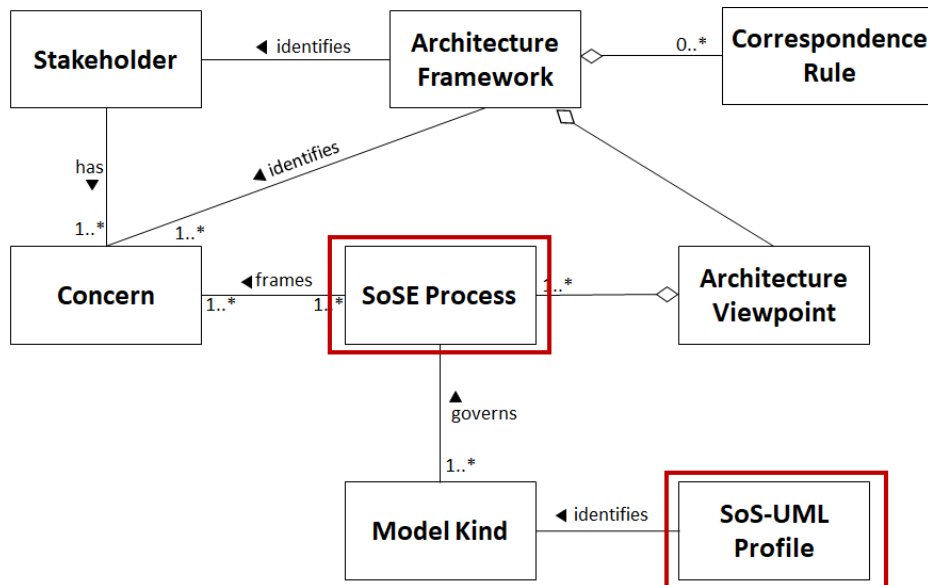


Figure V.3: An extended conceptual model of the SoS-AF.

As shown in Figure V.3, the extensions to the Framework Domain, particularly through the addition of tailored SoSE Processes and a specialized SoS-UML Profile, reinforce the prefabricated structure of knowledge inherent in architecture frameworks. This allows stakeholders to effectively organize architecture descriptions into a suite of complementary views, as advocated by the "ISO/IEC/IEEE 42010:2011, Systems and software engineering-Architecture Description" standard. This standard outlines a conceptual model that underpins the terms and concepts related to systems and architecture descriptions, thereby providing a scaffold for our expanded architecture framework within the SoSE landscape.

In Chapter V, we define an AF description for the SoSs called SoS-AF, which conforms to this international standard ISO/IEC/IEEE 42010:2011. The SoS-AF description is motivated by concerns commonly shared by SoSs' stakeholders across various development processes. Therefore, to form a collection of architecture VPs that constitutes the body of our SoS-AF description, we provide a comprehensive UML profile-based modeling basis for the notion of SoSs that can guide the development of SoSE processes pertained to shared concerns of each involved Stakeholder.

V.2.1.2 MeMSoS extension

Building on the initial MeMSoS model, we’ve developed an enhanced Meta-Model tailored for the complex AFs of SoS. The extended MeMSoS strengthens the Framework Domain by providing a more refined structure for SoSE, enabling the capture of diverse Viewpoints and Stakeholder Concerns. It enriches the domain with new constructs, assembly rules, and a detailed semantic mapping to automatically generate a comprehensive UML profile. This profile, within the SoS-UML Profile class, aids in managing cross-cutting concerns derived from stakeholders’ viewpoints. The extended version of MeMSoS explores the hierarchical structure of CSs and highlights the pivotal roles of its characteristics, such as Roles, Capabilities, Goals, and Relationships, in the overarching SoS architecture. To tailor the UML for the SoS framework domain, we expanded MeMSoS, introducing new constructs and assembly rules specifically adapted for SoS complexities. This evolution of MeMSoS not only defines the syntax of the SoSML modeling language but also specifies the elemental building blocks and their potential configurations.

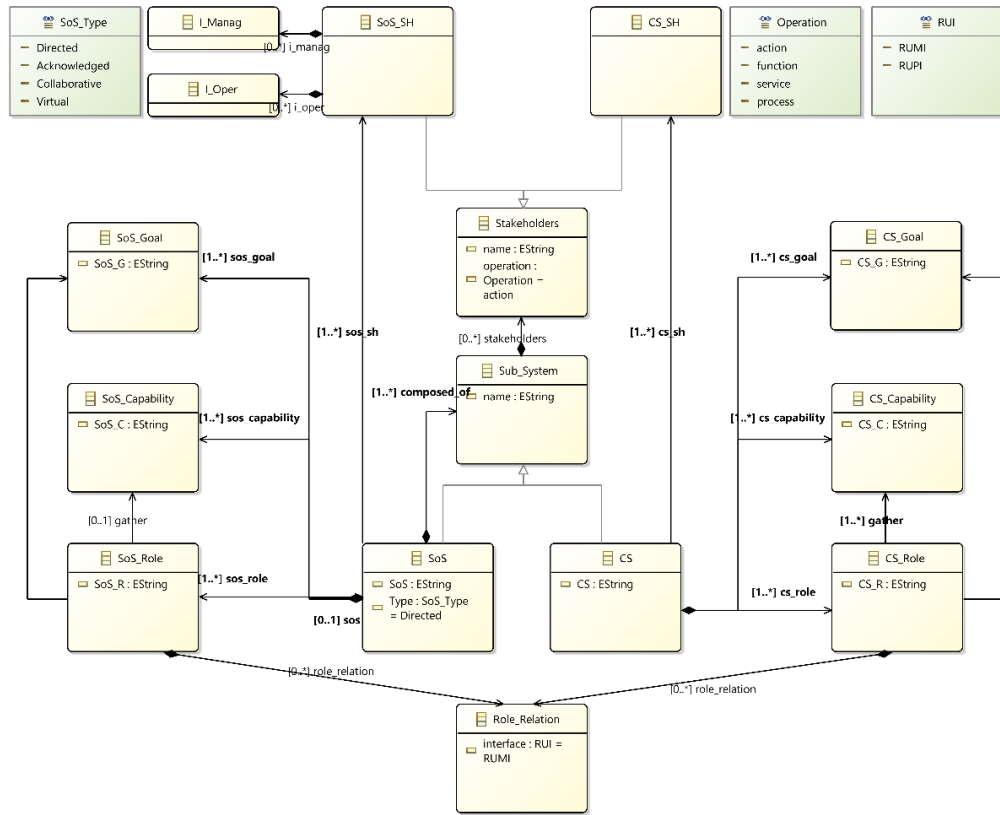


Figure V.4: Overview of the updated MeMSoS.

The enhanced version of MeMSoS aims to support for diverse Viewpoint modeling, encapsulate a wider range of Stakeholders and their Concerns, and integrate novel features. It provides comprehensive mapping details necessary to automatically generate a UML profile, which embodies the full semantic depth and precision required for SoS modeling. Leveraging these advanced abstract and concrete syntaxes within the SoS-UML Profile, we achieve a high-level specification of aspects and

effectively manage cross-cutting concerns derived from Stakeholders' Viewpoints [20].

As depicted in the Figure V.4, MeMSoS conceptualizes an SoS as a network of Sub-Systems, classified into two types: SoSs and CSs. These Sub-Systems engage in specific Capabilities and adopt necessary Roles to fulfill the SoS's Goals. At the heart of MeMSoS, the SoS class represents the system as an ensemble of Sub-Systems, each contributing to the collective mission. In the refined MeMSoS, we distinguish two sets of Capabilities, Goals, and Roles pertinent to each system—both CSs and SoSs. We identify distinct Capabilities for CSs (CS_Capabilities) and SoSs (SoS_Capabilities), with SoS_Capabilities comprising various CS_Capabilities. These are executed by CS Roles, which gather essential Capabilities allowing CSs to perform their roles toward realizing the SoS's overarching Goal.

Further, the Sub-System is a generalization that encompasses both SoS and CS, each associated with three aggregations that express their respective Capabilities, Roles, and Goals. The CS_Role and SoS_Role entities reference the 'gather' and 'accomplish' functions connecting them to corresponding Capabilities and Goals and are linked to the Relation class. This class introduces a novel attribute encapsulating the Interfaces through which CS_Roles and SoS-Roles interact, each Interface characterized by its Relied Upon Interface (RUI) type, which includes both Message (RUMI) and Physical (RUPI) Interfaces. Moreover, MeMSoS distinguishes two types of stakeholders for each system—one for CSs (CS_SH) and another for SoSs (SoS_SH)—with the latter further defined by associations with managerial and operational entities reflective of an SoS's independent properties.

In this updated MeMSoS iteration, we have enriched the Meta-Model by specializing and generalizing new classes that extend the Meta-Model's existing entities. These specialized classes fulfill the extension requirements for UML, aligning with the needs derived from the requirements Meta-Model and facilitating the prediction of significant requirements. To develop an AF in the Framework Domain, we have utilized the extended MeMSoS and the expanded ISO 42010 standard, along with UML Profile techniques. This domain leverages modeling to abstract SoS architectures, distilling commonalities that are independent of specific implementation technologies and execution platforms. An AF for SoSs is thus essential for automating SoS development and enabling abstract modeling that can later be transformed into a PSM and then into a specific platform implementation.

The proposed SoS-AF is grounded on the MeMSoS, which facilitates the PIM design level and allows modelers to refine the PIM concerning SoS aspects progressively. SoS-AF constitutes a set of UML Profiles for different viewpoints. The UML profile mechanism is advantageous as it is a lightweight method that does not alter the underlying MeMSoS. As illustrated in Figure V.5, we adopt an incremental, MOF-based, multi-level approach guided by the MDA. The topmost layer, M3, defines a language and framework for specifying, constructing, and managing Meta-Models, which underpin the definition of MeMSoS, the ISO standard, and our SoS-UML profile. The M2 layer employs the extended MeMSoS, defined using MOF as a Meta-Model, to establish mappings to the SoS-UML profile. This layer is crucial for specifying SoS commonalities, delineating various aspects among their CSs, and tailoring these to SoS modeling requirements. It also integrates the framework provided by the IEEE ISO 42010 standard. Additionally, the SoS-UML profile introduces new SoS modeling concepts by extending fundamental UML concepts with stereotypes.

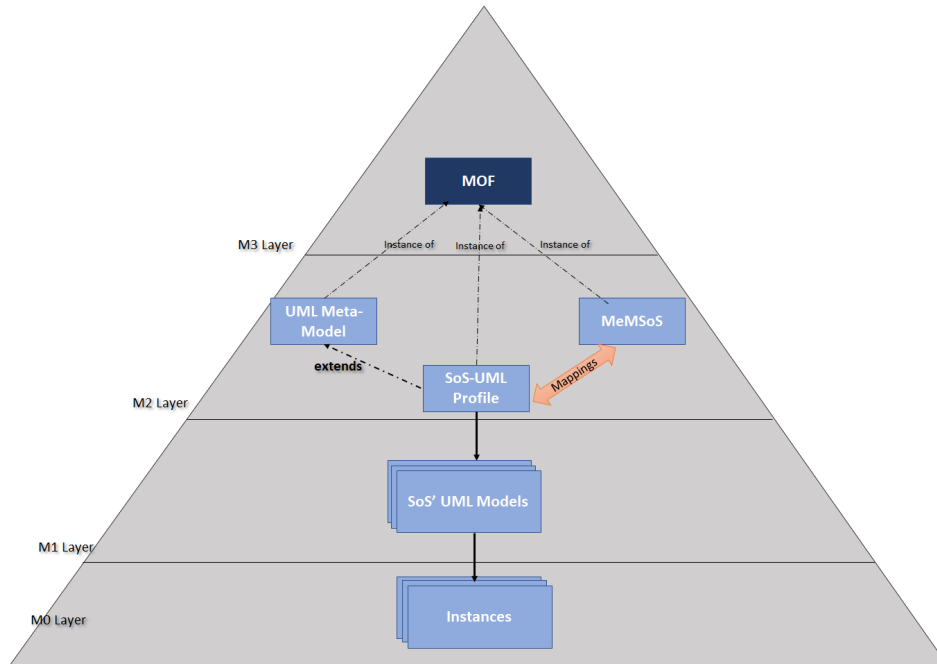


Figure V.5: Multi-level approach-based UML Profile.

The concrete assets for our domain, defined within the SoS-UML profile, are introduced at the M1 layer. The final M0 layer represents the operational SoS implementations, such as executed code. The M2 layer is particularly significant in our framework as it defines the concepts of the MeMSoS Meta-Model, serving as a descriptive tool for the AF and specifying the vocabulary that will be mapped to the SoS-UML profile, which will be presented in the subsequent section.

V.2.2 A Unified Architecture Framework: Overview

The approach described in Figure V.6 addresses the complexities of SoS architectures by introducing an SoS-AF within the Framework Domain. This framework integrates the multi-viewpoint approach of ISO/IEC/IEEE 42010:2011 with specialized SoSE processes for effective complexity management. The SoS-AF, essential to our systematic framework, incorporates various SoSE processes into a sequence of interconnected activities, thereby enhancing the framework's quality and aiding stakeholders in managing SoS architecture complexities.

A key component of the SoS-AF is the SoS-UML Profile, which introduces new stereotypes and notations derived from UML2.0 elements. This profile enables precise modeling of all structural and behavioral aspects of SoSs, facilitating the representation and organization of processes from diverse viewpoints. Addressing the lack of unified SoSE processes and consolidated Architecture Frameworks for SoSs identified in the literature, our approach utilizes a Model-Based SoSE methodology. The SoS-AF, a product of this methodology, supports SoS development and resolves common SoSE challenges. It presents a comprehensive structure encompassing four main phases, each vital to the development and application of the SoS-AF in various SoS contexts. This AF supports the development of SoSs and addresses common SoSE issues by providing a comprehensive framework

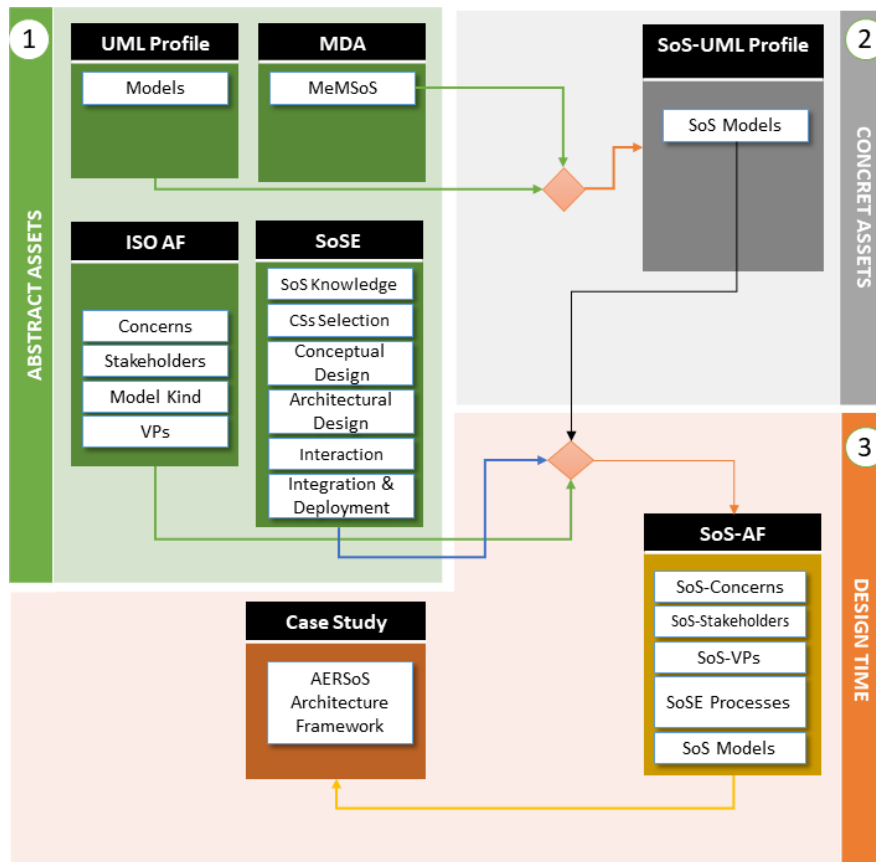


Figure V.6: *Multi-Viewpoints Framework for SoSs' Architectures.*

depicted in three distinct phases:

- **Abstract assets:** describes the initiation methods and the architectural principles required to create the framework domain; namely, ISO Standard 42010, MDA, UML profile and SoSE process model. This combination covers and explores the importance of commonalities across all CSs within the four types of SoSs in a high abstraction level framework. As well as it contains both the tools and the methods for constructing and managing SoSs architectures.
- **Concrete assets:** permits to leverage the MDA-based techniques to develop deduce the concrete assets of the framework domain. In particular, it proposes a UML Profile-based modeling tool for SoSs called SoS-UML Profile. The latter provides a generic extension mechanism for building UML models in SoSs domain, and offer a way that reflects and refines the specifications of the new framework.
- **Design time:** permits obtaining global SoS-AF by incorporating the first two phases. On the one hand, it is essentially based on the adaptation of “ISO/IEC/IEEE42010 standard: Systems and Software Engineering-Architecture Description” to make it suitable to a Model-Based SoS Engineering context. And on the other hand, it highlights the adoption of SoSE processes and demonstrates how SoS-UML Profile can be leveraged to lead the production of

SoSs architectures and to govern the involved stakeholders. Moreover, it can involve a set of examples to validate the framework and its related properties. It demonstrates that it can offer a consistent AF for any SoS case studies (e.g. AERSoS in Section V.2.3).

The proposed SoS-AF will enable various stakeholders to design each process from different viewpoints separately. i.e. it gives a generic methodology to ensure that the resulting SoSs architecture models will also yield the desired expectations. Moreover, we argue that this methodology and the SoS-UML profile-based model kind it provides, will offer a map to guide stakeholders toward achieving a unified SoS' AF.

V.2.3 Case study: Aircraft Emergency Response SoS

SoSs are an emerging vision for the next-generation systems that are built by interconnecting existing legacy systems. The field of SoSs comes up against constraints during the engineering process. SoS can take four different types. These types are primarily based on the governance, management complexity and the relationships among the CSs in the SoS. Air traffic has become widespread since the 1970s and it continues to grow; the International Air Transport Association (IATA) predicts 8.2 billion air travelers in 2037 [21]. Aviation accidents involve not only passengers and flight crews, but also a vast territory, around the airport with a large number of residents who, in some cases, have paid a heavy price in terms of their lives.

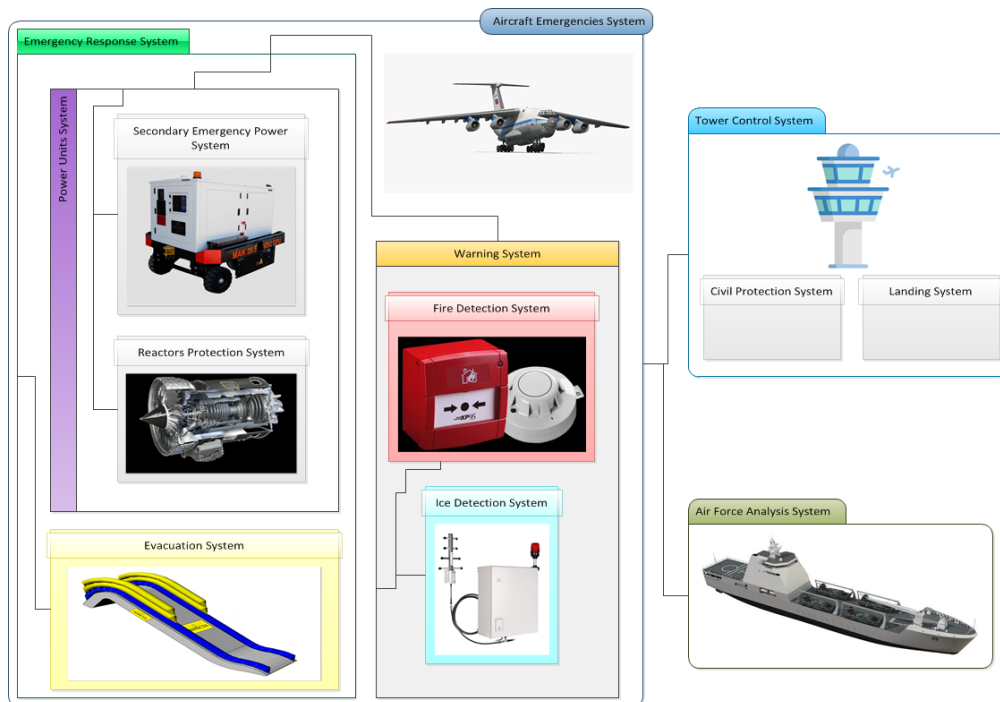


Figure V.7: Aircraft Emergency Response(AERSoS).

From an IT point of view, frequent problems causing aviation accidents are mainly due to a misunderstanding of the interaction and communication between aviation CSs. Stakeholders must

pay particular attention to the idea that aviation is a SoS [20] and the impact of such an approach at the conceptual design stage [27]; since the aviation system is bound by a set of distinct CSs provided by different companies (aircraftSubsystems, air traffic management/control subsystems, etc.), each of these CSs has managerial and operational independence. On this basis, we employ in this section an Aircraft Emergency SoS to illustrate through this case study the global mission of “maximizing the aircraft safety by minimizing critical situations”; this is done by providing the required emergency response and reacting to the aircraft critical problems. The AERSoS interacts with many independent CSs that are deployed in the aircraft or even those that are dispersed. AERSoS CSs are composed through communication links as described in Figure V.7.

At the aircraft level, hundreds of sensors are installed on all parts of the aircraft CSs (turbines, oil, lubricant, etc.) that communicate with each other and collect data on the aircraft’s performance and flight conditions. These CSs on the one hand are responsible for collecting data on high and unusual temperatures in turbines, oil, lubricant, etc., and on the other hand, they detect the gases emitted by potential fires. Moreover, warning SoS signals a problem that may occur with the aircraft at a given position. This system is constituted of a set of CSs that interact directly with the installed sensors; In the case that these sensors detect something wrong (fire, flame, gases...etc.), another autonomous CS takes on the mission of verifying the validity of the indications and evaluating the risks have:

- Signals a state of danger on the plane and informs the flight crew.
- Provokes the initial response systems which perform functions that are intended to mitigate the emergency:
 1. CSs designed to isolate damaged engines and suppress fires;
 2. CSs that generate secondary power(ex: hydraulic and electrical power, liquid oxygen, etc.);
 3. Passenger rescue CSs (provide O2, lifejackets, etc.).
- Interacts with the control tower to try to clear the runway and airway for an emergency landing at the nearest airport using GPS, as well as airport firefighters to ensure first aid.
- Interacts with the Air Force Analysis System (AFAS) for the analysis of the technical report containing the details of the missing parts to be sent to maintenance for quick repair upon landing. As we explained, this SoS like others requires special attention to describe its complex behaviour through the arrangement of its subcomponents, from several points of view.

In this chapter, the AERSoS case study will be considered to clarify and concretize the basic elements and the contributions made by SoS-AF. We will adopt it in the next Section V.4 to encapsulate all the necessary notions in the SoS-AF. To achieve the global goal, AERSoS must be designed in such a way that the CSs can interact and perform a unique capability that cannot be provided

by any of the CSs. Examples of the AERSoS' CSs include an AircraftEmergenciesSoS, EmergencyResponseSoS, WarningSoS, EvacuationCS, TowerControlCS, EnginesProtectionCS, LandingCS, etc. see Figure V.7.

Each CS of the AERSoS is a system that is specified by a set of entities, which are divided into three types. The first set represents the Roles describing the ideal behavior of CSs through the gathering of the required Capabilities to accomplish the AERSoS global-Goals, the second represents the Capabilities describing the functions provided by each CS in specific Roles to the wider needs of the AERSoS, and finally, the Goals, describing instances of the AERSoS' Roles, that represents sub-Goals of each CS and Global-Goal of AERSoS.

V.3 A Multi Viewpoints-based Architecture Framework

Architecting an SoS helps to understand how it works, as well as to master its complexity before its implementation. This offers considerable advantages to the designers of such systems. Besides, it can be seen as the separation of different static and dynamic aspects as well as the functional and non-functional needs which are related to SoSs characteristics, SoSs quality attributes, management and oversight, SoSs architecture, design and Implementation.

In this new perspective, SoSs development does not follow the normal system development process. As SoSs' capabilities are based on the contributions of the individual CSs, their interdependencies make a document-centric development impractical as an exorbitant effort. The development processes refer to activities that can guide an SoS' lifecycles from the system requirements level down to the software implementation level, and naturally, by coordinating the various processes for the development of a new system [20], [22].

From a SE perspective, design decisions made at the architectural level have a direct impact on the fulfillment of functional and quality requirements of SoSs development. At this stage, the SoS' Stakeholders identify functional and non-functional characteristics through the use of their theoretical backgrounds, notations and environments. In addition, the SoSs' architectures are still created without the support of systematic processes and traditional design approaches do not adequately support the creation of these types of systems due to their composed nature, their large scale, their decentralized control mechanism, their evolving environments, and their stakeholders [20], [22].

To get a handle on this complexity, it is necessary to maintain consistency and coherence between the different viewpoints of different stakeholders, as well as the ability to reconcile and include all their viewpoints before proceeding to the various processes in the SoSE lifecycle. In an SoSs Engineering, the next section introduces an SoS-AF based on a Framework Domain, that can facilitate and improve the design of SoSs' architectures.

Architecture frameworks are mechanisms widely used in architecture. They establish a common practice for creating, interpreting, analyzing and using architecture descriptions within a particular domain of application or stakeholder community. As a result, their uses include, but are not limited to [20]:

- Creating architecture descriptions.

- Developing architecture modeling tools and architecting methods.
- Establishing processes to facilitate communication, commitments, and interoperation across multiple projects and/or organizations.

The idea is that an AF is a knowledge-prefabricated structure that stakeholders can use to organize an architecture description into complementary views, [22]. The specification of an AF is one area of standardization in ISO/IEC/IEEE42010:2011. This standard proposes a conceptual model to describe the terms and concepts of systems and architecture description. This standard specifies an AF as a composition of multiple Viewpoints (VPs), each VP can be used to address specific concerns of different Stakeholders [37].

Our proposal in this section is to refine and tailor our SoS-AF to meet these architectural requirements more effectively. The SoS-AF proposes a multi-viewpoint approach to suit the specific dynamics of SoSE, simplifying its complexity and enhancing its practicality. This adoption of ISO42010 standard and its adaptation into our context ensure a more coherent integration of various architectural components and viewpoints, making the standard more user-friendly and relevant for SoS design.

Additionally, the framework advances by integrating systematic and model-based SoSE processes. This integration fills a critical gap in the existing literature, offering much-needed support for the design of SoSE processes in alignment with the IEEE 42010 standard. Moreover, the development of an SoS-UML Profile introduces a comprehensive range of models to accurately represent SoS architectures. With new stereotypes and notations extending from UML2.0, it enables a precise depiction of both structural and behavioral aspects of SoSs, facilitating a deeper understanding and more effective architectural modeling.

V.3.1 Architectural concepts

As seen in Section V.2.1.1, SoS-AF aligns with the IEEE 42010:2011 standard, it is designed to reflect the concerns prevalent among SoS stakeholders throughout different development processes. This framework integrates a collection of architecture VPs, each formed through the aggregation of specific SoSE processes and governed by distinct Model Kinds defined by the SoS UML Profile. These Model Kinds are pivotal in shaping the SoSE processes, outlining their relationships and dependencies. The subsequent sections will explore the SoS-AF in detail, focusing on its structure, the integration of Stakeholders' concerns, and the interaction between SoSE Processes and VPs.

V.3.1.1 SoS Stakeholders

In SoS-AF, we identify the various stakeholders of SoSs categorizing them based on their roles and responsibilities. This identification is detailed in Table V.2, where we introduce the diverse functions and tasks of each stakeholder. The SoS-AF leverages SE principles, enabling every stakeholder to contribute effectively to maintaining the SoS, whether their tasks are technical, functional, etc.

Table V.1: DIFFERENT ROLES OF STAKEHOLDERS.

Stakeholders	Analysts	Users	Owners	Developers	Maintainers
SoS Experts		X	X		X
Architects and Designers	X			X	X
Collaboration Specialists	X			X	X
Interactions Engineers				X	X

Stakeholders are inherently heterogeneous due to multiple users, their VPs, engineering processes, platforms, environments. . . etc. they could be individuals, groups, or organizations holding Concerns for an SoS. They use SoS-AF Description to understand, analyze and compare SoS’ architectures. Each SoSs’ concern could be managed by one or more stakeholders; four main stakeholders have been identified in this framework:

- **SoS Experts:** a group of persons responsible for the Business VP establishment, with strong theoretical knowledge in an SoS application domain and they have the ability to understand the practical implications of the existing CSs that can participate in SoS and can translate SoS Capability Objectives into High-Level SoS Requirements.
- **Architects and Designers:** their role is vital to the success of both Analysis and Design VPS, they translate the requirements into a demand for CSs Capabilities. i.e. they look at business plans and requirements provided by SoS Experts, analyze the Goals and the Capabilities of the selected CSs, and propose recommendations on the right selection of CSs to achieve the SoS’ Global Goal.
- **Collaboration Specialists:** they are also Analysis and Design experts; they are responsible for understanding CSs and their Capabilities’ collaboration. They also look at integrations with existing CSs, interfaces with people and other systems.
- **Interactions Engineers:** persons responsible for the Deployment VP and they are responsible for specifying communication and interactions between different Roles. They oversee the CSs’ Capabilities and their Roles’ interactions to facilitate the interaction modeling within an SoS application.

V.3.1.2 SoS Concerns

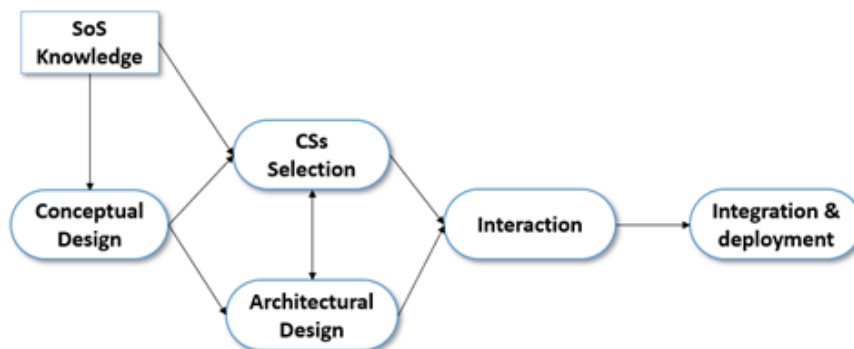
Concerns in SoSs emerge across all stages of development, from knowledge acquisition about CSs to design and implementation. These concerns can manifest in various forms, including stakeholder relationships, global missions of SoSs, capabilities, requirements, modeling constraints, interdependencies among CSs, quality attributes, and design decisions. Our work in [21]–[23] has enabled us to identify these concerns by thoroughly examining different aspects of SoSE. These concerns are organized according to five key SoSE processes, and to effectively address them, a series of critical modeling and implementation questions have been developed, as outlined in Table. V.2.

Table V.2: ARCHITECTURE FRAMEWORK-RELATED CONCERNS.

Concerns	Description
C.1	What requirements should the SoS meet?
C.2	How does the SoS ensure knowledge?
C.3	What are the design decisions related to any influences on the SoS in its environment?
C.4	How does the SoS support the stakeholder relationships?
C.5	What are the modeling constraints and CSs' interdependencies that the SoS should have?
C.6	What are the local and global goals of the SoSs?
C.7	How does the SoS achieve sustainability, flexibility, and interoperability?
C.8	How does the SoS carry out the self-organization and adaption requirements?
C.9	How does the SoS respect a good representation of the requirements?
C.10	How does SoS aim to reduce execution time and optimize resource consumption?
C.11	How does the SoS dynamically reconfigure the heterogenous set of CSs?
C.12	How do the SoS' constituents interact with each other at runtime?
C.13	Can new CSs be integrated into the SoS at any time?
C.14	How do CSs adapt to each other and the environment?
C.15	What are the new capabilities that can be added after the integration of the different CS?

V.3.1.3 SoSE processes

SoSE processes have a major stake in the SoS-AF and a strong influence on its implementation. Our contribution consists of inspiration from the SoSE processes provided by [2] to guide the SoSs lifecycle processes from CSs knowledge, to design and implementation. We propose to take inspiration from this work by modifying some elements to adapt it to our previous works. SoSE processes express the activities engaged under SoSE from the perspective of one or more Stakeholders to frame specific Concerns, using the conventions established by its models. Each SoSE process will be identified by one or more governing Model Kind that adhere to the conventions of SoS UML Profile.

**Figure V.8:** *SoS development processes.*

As shown in Figure V.8 the main SoS development processes involved in our SoS-AF are:

- **SoS Knowledge:** addresses high-Level SoS requirements and investigates existing CSs that can participate in the SoS.
- **CSs Selection:** this process consists of choosing a set of CSs and distinguishing their relevant Capabilities and Goals.
- **Conceptual Design:** the design involves creating a global vision of an SoS, defining the essential relationships and identifying mission capability assessment.
- **Architectural Design:** represents a global architecture for the SoS' constituents and their possible Roles. It could be developed in parallel with the CSs Selection process.
- **Interaction:** the different CSs involved in an SoS usually have different Capabilities. Therefore, a large part of the software engineering effort in the SoSE is to design interactions so that the CSs can interoperate.
- **Integration & deployment:** this process implies that the different CSs involved in the SoS work together and interact through the assigned Roles. Deployment of the system consists of setting up the CSs interactions in the organizations concerned and making it operational.

V.3.1.4 SoS Architecture Viewpoints

A viewpoint in SoS-AF is a selection of relevant aspects of the SoSE processes (and their Stakeholders' concerns); and the representation of that part of an architecture that is expressed in different Model Kind. It is claimed that the SoSE processes form a necessary and sufficient set to meet the needs of SoS-AF. Four main VPs are identified in our proposed SoS-AF:

- **Business Viewpoint:** This stage focuses on acquiring knowledge and identifying the requirements of Constituent Systems (CSs). The SoS experts draft initial SoS requirements, pinpointing potential capabilities derived from existing CSs in the application domain. Key challenges addressed include architecture challenges C.1 and C.2. The process is guided by the Knowledge process.
- **Analysis Viewpoint:** Led by architects, designers, and collaboration specialists, this viewpoint involves the selection of appropriate CSs to provide necessary SoS capabilities. It also encompasses building a conceptual design model to understand the relationships and interdependencies of CSs within the SoS. Challenges C.6, C.9, C.7, C.10, and C.11 are addressed in this process.
- **Design Viewpoint:** This viewpoint assists architects and designers in the SoS design process, from initial sketches to detailed design. It deals with the core processes of SoS development and the architecture of CSs, roles, and collaborations. It comprises two processes: Architectural and Interaction, focusing on detailed modeling for design decisions (C.3, C.4, C.8, C.15) and managing fundamental interdependencies among CSs, roles, and their collaborations.

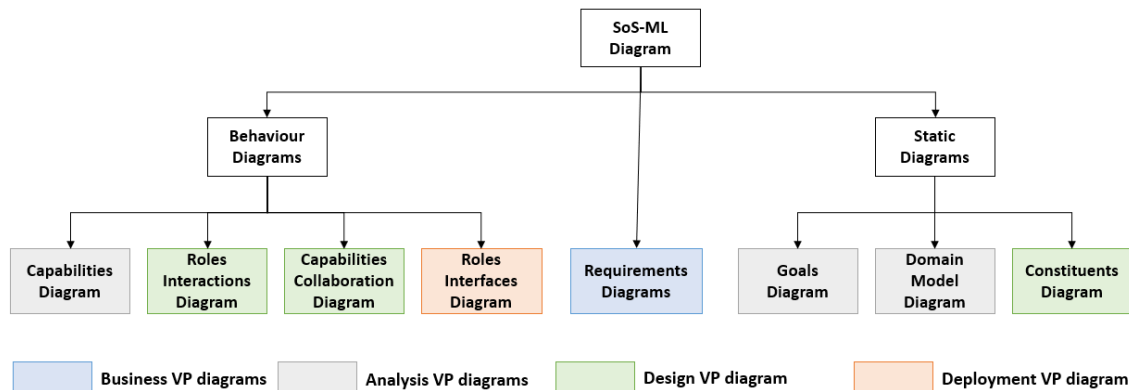


Figure V.9: SoS-UML Profile's diagrams.

Table V.3: ASSOCIATIONS OF STAKEHOLDERS, PROCESSES AND CONCERNS.

Stakeholder Process	SoS Experts	Architects Designers	Collaboration	Inter. Engineers
SoS Knowledge	C.1 C.2			
CSs Selection		C.6	C.9	
Conceptual Design		C.7	C.10 C.11	
Architectural Design	C.3	C.4 C.15	C.8	
Interaction			C.5	C.14
Integ. & deployment			C.12	C.13

- **Deployment Viewpoint:** Focused on the integration and deployment of CSs, this viewpoint is managed by collaboration and interaction experts. It involves merging diverse roles to define, control, and monitor complex interactions (C.12 and C.13) across SoS and CS boundaries.

V.3.1.5 Model kinds and UML-Profile

The SoS SoS-AF is designed to facilitate modeling all concepts and relationships within SoS entities, capturing both their static and dynamic aspects. To achieve this, we introduce the “SoS-UML Profile”, which provides graphical constructs for various diagrams associated with each VP, as summarized in Figure 6. This profile enables the representation of requirements diagrams and their integration with other Static and Behavior diagrams. Static diagrams encompass Goals, Domain model, and CSs diagrams, while Behavior diagrams include Capabilities, Roles Interaction, Capabilities Collaboration, and Roles Interfaces diagrams. This SoS-UML profile, its diagrams and all their associated concepts will be detailed in the next Section V.4.

V.3.2 Associations

Table V.3 illustrates the relationships between concerns, stakeholders, and SoSE processes. It categorizes concerns by stakeholder groups (columns) and aligns them with relevant SoSE processes (rows). This structured approach offers clarity on the roles and responsibilities of stakeholders for specific concerns and ensures effective management and execution across the SoSE lifecycle.

Table V.4 below maps out which development processes are associated with each viewpoint in SoS development. The 'X' marks indicate the involvement of a specific process within a particular viewpoint. For example, the SoS Knowledge process is associated with the Business viewpoint, while CS selection and Conceptual Design are part of the Analysis viewpoint, and so on. This format delineates the roles and responsibilities across different stages of SoS development.

Table V.4: ASSOCIATION OF VIEWPOINTS AND SoS DEVELOPMENT PROCESSES.

Viewpoint Process	Business	Analysis	Design	Deployment
SoS Knowledge	X			
CSs Selection		X		
Conceptual Design		X		
Architectural Design			X	
Interaction			X	
Integratio & deployment				X

In other words, Figure V.10 below illustrates the AF that describes the proposed methodology. It can depict four different and complementary parts:

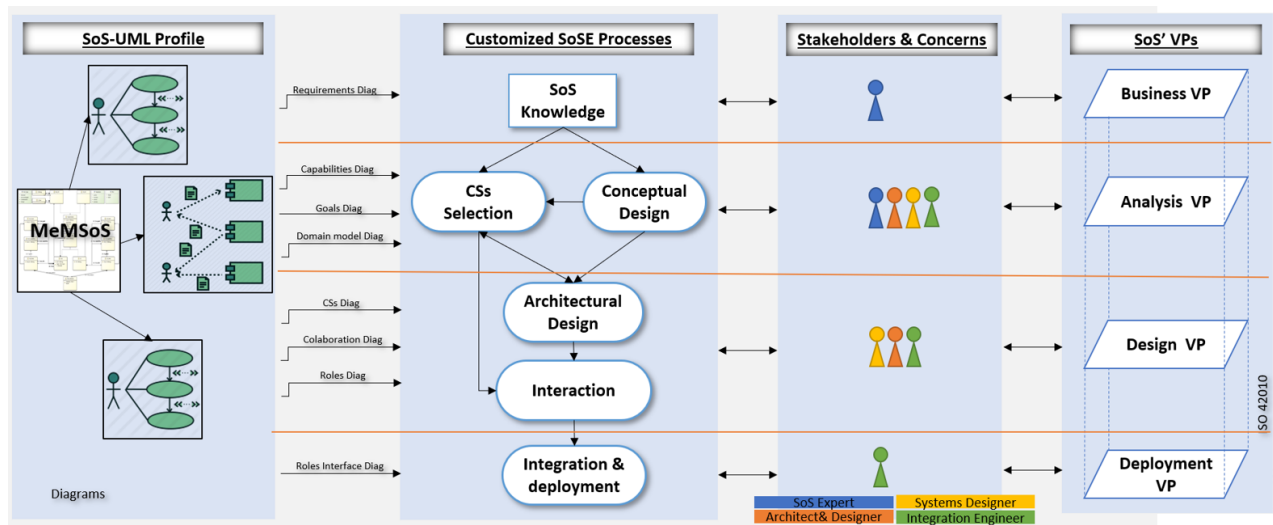


Figure V.10: Overview of SoS-AF.

- SoS-UML Profile, the proposed tool takes advantage of the MeMSoS model to frame the concepts, the characteristics and the structural and dynamic aspects of an SoS. The idea is that we consider the MeMSoS as our reference of concepts to map them to the UML Profile for SoSs, and implement the AF to introduce a set of model kind.
- To facilitate the task of designing and managing these diagrams, we used a customized set of SoSE development processes to support the specific parts of the architecture. the objective is to separately capture, describe and organize each diagram.

- The framework take advantage again of the customized SoSE processes model to enable the stakeholders to explicitly manage their concerns that they want and express them using an SoS-UML profile's diagram.
- The deliverable of this AF processes is a set of SoSs specifications that provide a set of guidelines for structuring the specifications expressed as models corresponding to different VPs. The design of these VP is based on a consolidated Model-Based SoSE in which the AF can be seen as a sequence of connected and dependent VP. i.e. each VP is created through an aggregation of one or more SoSE process. Each one of these latter is governed by a set of diagrams that are appropriate to specific concerns.

V.4 UML Extensions for Modeling the unified architecture

SoS-AF is our proposed architecture framework to support the entire development of SoSs' life-cycles, including its various processes from the requirements' specification process, design, to the implementation process. Hence, the need for a tool allowing the definition of the overall architecture of SoSs, and also the definition of specific models for each involved process (such as UML models) is essential. For this reason, we present a UML2.0 extension tool, denoted SoS-UML profile, for the management of SoS' architectures following the approach proposed in SoS-AF.

A key engineering challenge then is to construct an Architecture Framework of SoSs based on the proposed SoS-AF, and this requires to:

- Create different VPs matching the VPs of SoS-AF at high-level abstraction considering different stakeholders' concerns,
- Take advantage of SoSE processes that can be used to manage the AERSoS,
- Use the SoS-UML Profile to abstract their analysis and design from implementation technologies, increase the automation of the development of AERSoS and allow VPs modeling.

To be able to specialize the UML for the SoSs domain, we needed to extend the MeMSoS to integrate new entities and constructions adapted to treat SoSs. Consequently, MeMSoS' will represent the definition of SoS-UML Profile below and defines the available building elements and how they can be assembled.

The new version of MeMSoS offers better support for different VPs modeling, frames Stakeholders' concerns, and introduces new features. This new version provides all the mapping information required to automatically generate the SoS-UML profile with all the semantic expressiveness and precision. With the new extended abstract and corresponding concrete syntaxes of SoS-UML Profile, we can successfully reach high-level specifications of aspects and address the cross-cutting concerns that depend on the Stakeholders' VPs.

The proposed SoS-UML profile is a Meta-Model extension mechanism that allows stakeholders to add new elements of the MeMSoS Meta-Model, better suited to model particular systems as SoSs.

Every existing element will be specialized by a stereotype and semantically equivalent to a new class of the MeMSoS which will bear the same name as the stereotype. In the following, we introduce new elements enriching UML 2.0 diagrams. These elements will make it possible to frame the main processes, and aspects characterizing the notion of SoSs. As well as we give a brief representation of the abstract syntax for the proposed profile.

The main purpose of this section is to present some UML extensions that define the Meta-Modeling aspects of our SoS-UML Profile. We have used the “IBM Rational Software Architect 9.0” tool for the realization of this profile. As well as, we work on an instance of Eclipse which loads the plugins that we generated previously, to be able to design a set of examples of our models. This section presents the abstract/concrete syntaxes of the SoS-UML profile, which is structured into packages labeled by the SoSE development processes’ names to make groupings of different aspects, and thus better manage the complexity of each process. Figure V.11.(a). shows a screenshot of the different packages in IBM RSA tool that make up the Profile’s Meta-Model, it involves six packages namely: SoS_Knowledge_Package, CS_Selection_Package, Conceptual_Design_Package, Architectural_Design_Package, Interaction_Package and Integration_Deployment_Package. Additionally, Figure V.11.(b). shows their relevant inter-dependencies as mentioned in the SoSE processes model. In the following, we will demonstrate how we use each package of them as a model kind to design the main concepts of each process in the SoSE development.

To sum up, Figure V.11 below illustrates the AF that describes the proposed methodology. It can depict four different and complementary parts:

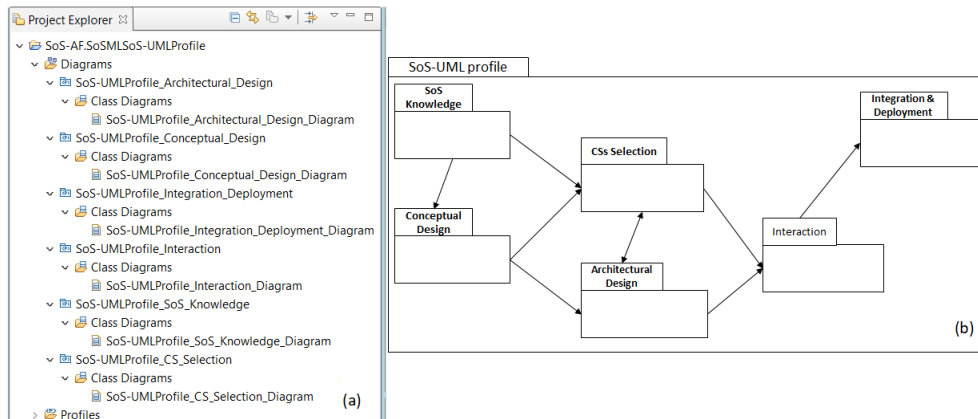


Figure V.11: SoS-UML profile packages.

V.4.1 SoS_Knowledge_Package

The SoS_Knowledge_Package describes the basic elements needed to describe the SoS Knowledge. The deliverable contained in this package will guide the description of the SoS’ Goals and support the identification of its Capabilities during the next process. Particularly, the two stereotypes Requirement and System are the central concepts in this model, and they represent a unit SoS Knowledge process. The latter starts with understanding the desired Requirement and suggesting a set of Systems as various options for achieving that Requirement. It is used for the representation

of the Requirement Diagram’s Model. The package contents are shown in the next Figure V.12.

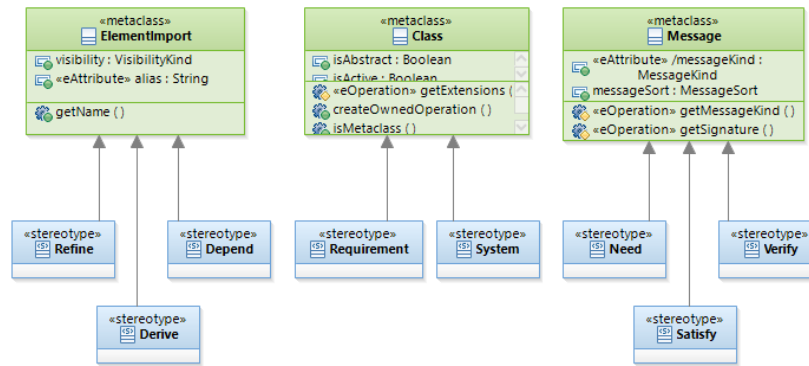


Figure V.12: The structure of the SoS_Knowledge_Package.

The Requirements Model represents the functionalities or the conditions that an SoS must fulfill based on the contributions of the collaborative CSs. As shown in the package figure, it contains different stereotypes for describing the knowledge and requirement of an SoS, and how they can be related to the necessary entities to gather, organize, analyze and decompose the different existing systems that can participate in an SoS. A part of this diagram, describing the most important stereotypes and the extended meta-classes is shown in Table V.5.

Table V.5: Stereotypes of the SoS_Knowledge_Package.

Process	Model Kind	UML Diagram	Stereotypes	Description	Meta-class
SoS knowledge	Requirement Diagram	Class Diagram	Requirement	Capability or Goal that must (or should) be performed	Class
			System	Systems, SoS, CS. . . etc	
			Refine	Clarifies the requirement’s meaning or context	ElementImport
			Depend	A requirement uses or depends on other ones	
			Derive	Impose additional sub-requirements	
			Need	Express required systems for a requirement	Message
			Verify	Relate a requirement with a system that verifies it	
			Satisfy	Relate a requirement with a system that satisfies it	

Take the AERSoS case study, requirements can be organized as an ordered tree hierarchical structure. A typical structure may include a top-level requirement for all sub-requirements. By using different relationships, each requirement within the top-level one may be associated with different systems (SoSs or CS, component, etc.) to describe its scope; for example (see Figure V.13), the «Requirement»HandleAbnormalSituation wich is derived from «Requirement» EnsureAircraftSafety

can be satisfied by «CS» *AircraftEmergencySystem* using the two relationships «Derive» and «Satisfy» respectively.

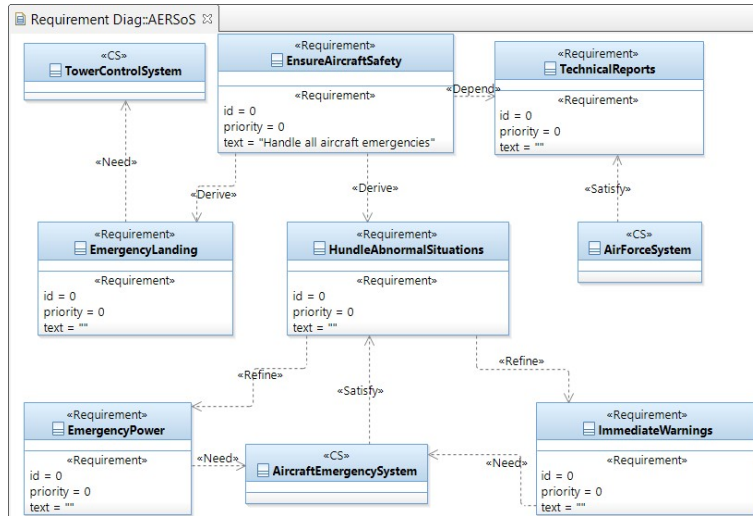


Figure V.13: Requirements diagram for AERSoS.

V.4.2 CSs_Selection_Package

The Selection of CSs requires the characterization of the Capabilities in which the CSs will perform to fulfill their Goal, and thus, the SoS' global Goal. Therefore, the main Systems that can participate in the SoS must be defined, described, and documented using the CSs_Selection_Package. As shown in the structure of this package that is depicted in Figure V.14, the extensions proposed here comprise stereotypes that reflect the entities that constitute the basis for the specification of Systems' Goals (CS_Goal and SoS_Goal) and the identification of their relevant Capabilities (CS_Capability and SoS_Capability).

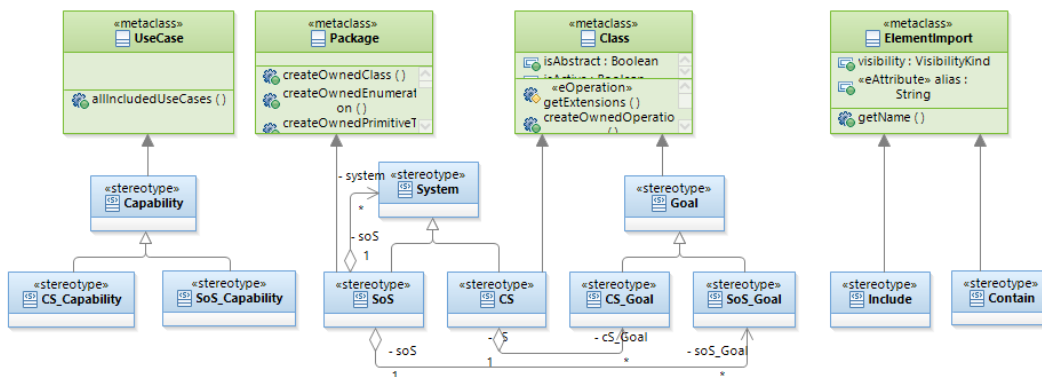


Figure V.14: The structure of the CSs_Selection_Package.

The CS_Selection_Package uses the output artifacts of the SoS Knowledge process (Requirements) to describe the SoS in terms of Goals and Capabilities. Thus, the notions of CS_Goal and

SoS_Goal are the central concepts in the Goals Model, representing a unit of the local goals of CSs as well as the SoS global goal in which high-level Goals may be realized through the combination of lower-level Goals. As well as, the Capabilities Model contains the concepts enabling the description of Capabilities which the selected CSs should perform to achieve the predefined Goals. The concepts in this package are divided into two diagrams, Goals Diagram and Capabilities Diagram (listed in Table V.6).

Table V.6: Stereotypes of the CS_Selection_Package.

Process	Model Kind	UML Diagram	Stereotypes	Description	Meta-class
CSs Selection	Goals Diagram	Class Diagram	Goal	Represents objectives of a system	Class
			CS_Goal	Represents objectives of a CS	
			SoS_Goal	Represents objectives of an SoS	
			Include	Split a goal into several sub ones	ElementImport
			Contain	Expresses the capability of having sub-goals	
	Capabilities Diagram	Use Case Diagram	System	Could be any type of System providing a Capability	Package
			SoS	SoS providing the Capability	Class
			CS	CS providing the Capability	
			Capability	Refers to functions provided by any system	Use Case
			CS_Capability	Refers to functions provided by any system	
SoS_Capability	Refers to functions provided by an SoS				

The first diagram in this package is Goals Diagram where different Goals can be organized as a tree structure in which a high level Goal that represents «SoS_Goal» may be realized through the combination of lower level Goals «CS_Goal» of CSs. In addition, relations between them denote sharing of the same common Goals; for example, Figure V.15, the «SoS_Goal» *Aircraft Safety* has three sub-goals, «CS_Goal» *Safe Landing*, «CS_Goal» *Safe Flight* and «CS_Goal» *Accident Report*.

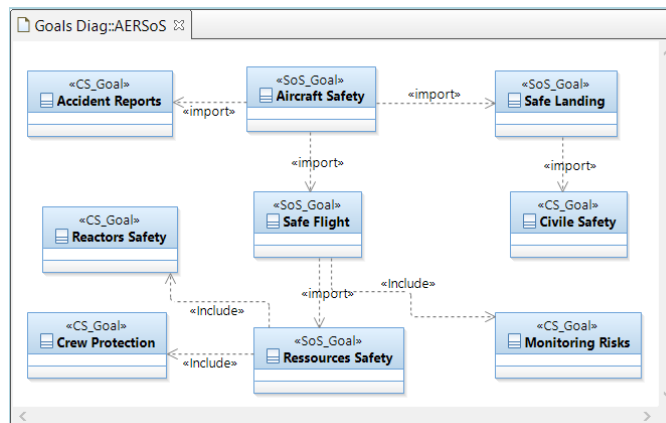


Figure V.15: Goals Diagram for AERSoS.

For the second diagram (Capabilities Diagram), it can be viewed as a mechanism to capture the SoS Capabilities in terms of the Capabilities of the pre-selected CSs which specify the expected behavior (what), and not the exact method of making it behave (how) of an SoS and thus, it represents a black-box view of the SoS; it is therefore well suited to serve later in Interactions and Architectural Design Diagrams. Figure V.16. represents the AERSoS capabilities, where a set of sub-capabilities (e.g. «CS_Capability» *GeneratingPower*, «SoS_Capability» *InsultingReactors*) for different CSs are required to perform the global-Capability «SoS_Capability» *ControllingSituation*.

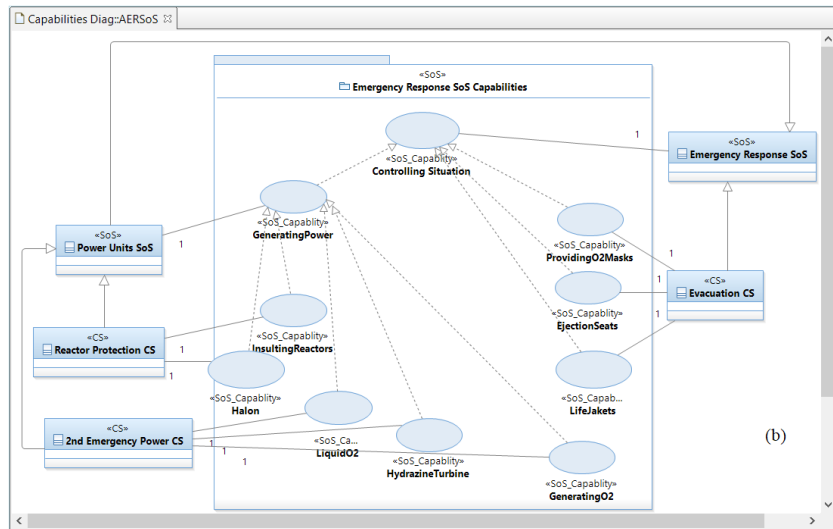


Figure V.16: Capabilities diagram for AERSoS.

V.4.3 Conceptual_Design_Package

This package captures the main blocks for designing the Conceptual Design process, the design allows creating a global vision of an SoS, defining the essential relationships and identifying mission capabilities. The main concepts in this model are Systems, Roles, Capabilities and different relationships types that can offer a global understanding of CSs, their Roles and their interdependencies as part of an SoS. A general structure of the package is depicted in Figure V.17.

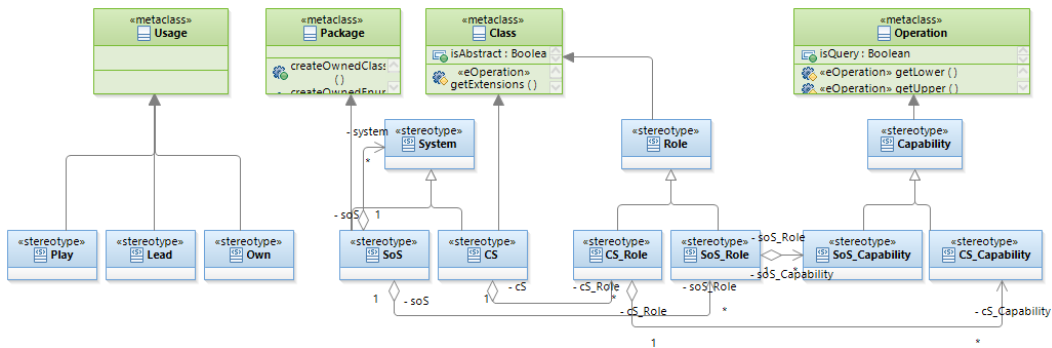


Figure V.17: The structure of the Conceptual_Design_Package.

Table V.7: Stereotypes of the Conceptual_Design_Package.

Process	Model Kind	UML Diagram	Stereotypes	Description	Meta-class
Conceptual Design	Domain Model	Class Diagram	System	Represents the involved System	Class
			SoS	Represents the involved SoS	
			CS	Represents the involved CS	
			Role	Ideal behavior of any type of System	
			CS_Role	Ideal behavior of a CS	
			SoS_Role	Ideal behavior of an SoS	
			Capability	Represents Capabilities of a System in specific Role	Operation
			CS_Capability	Represents Capabilities of a CS in specific Role	
			SoS_Capability	Represents Capabilities of an SoS in specific Role	
			Own	Expresses authority of one system to another	Usage
			Lead	Expresses control or guidance of one system to another	
			Play	Associates systems with the required Roles	

The stereotyped concepts in this model can be used to provide a global structure of an SoS to enhance the interaction of its CSs. Consequently, they can be used to describe the CSs, the Capabilities they have to accomplish Goals and the Roles they play within an SoS. In Addition, the Conceptual_Design_Package defines to which a Sub-System has access to and which Role it can play to solve missions. The concepts of this package introduce one single diagram called Domain Model, as showed in Table V.7.

At this stage, the Domain Model Diagram is used by stakeholders to design the SoS' characteristics in terms of its structural CSs, behavioral Roles, the internal Capabilities and relationships between the CSs. An example of PowerUnitsSoS as it is depicted in Figure V.18, the stakeholders can display various kinds of CSs and SoSs that constitute the top-level entities, e.g. «SoS» *PowerUnitsSoS*, «CS» *ReactorsProtectionCS*, etc. their corresponding Roles, e.g. «SoS_Role» *PowerGenerator* «CS_Role» *OriginProvider*, etc. and relationships among them «Play» «Lead», etc.

V.4.4 Architectural_Design_Package

This package which is depicted in Figure V.19 presents the concepts to support the Architectural Design decision in every CS' architectures. This is required to propagate the CS' architectural

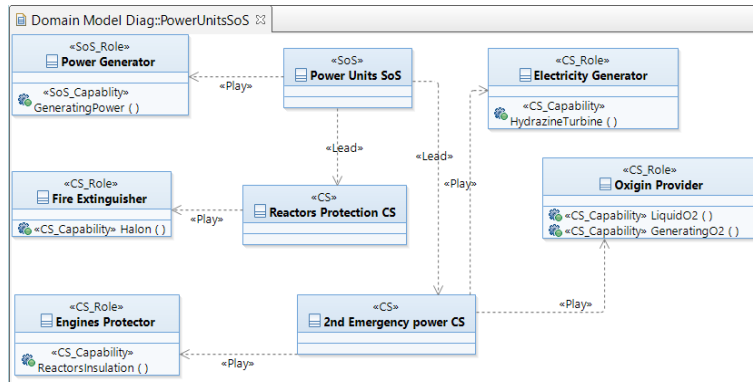


Figure V.18: Domain Model for the PowerUnitsSoS.

characteristics in the next processes in the lifecycle of an SoS. They manifest the structure of every CS by characterizing which Roles are part and which functions are used by the different stakeholders.

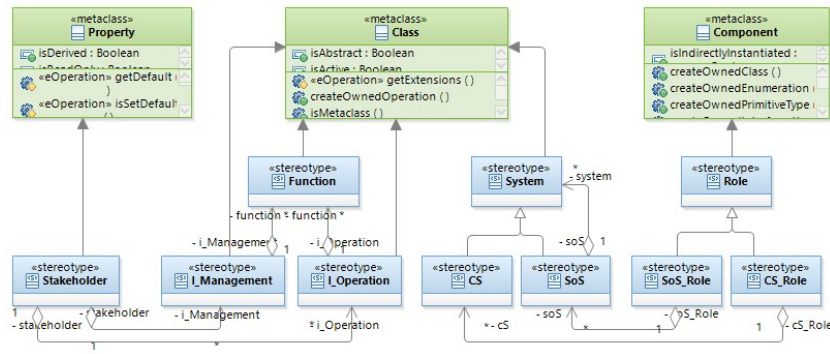


Figure V.19: The structure of the Architectural_Design_Package.

The Constituent Model has the ability to describe the internal structure of every autonomous entity cooperating within the SoS and how the operational and managerial independence can be defined. Additionally, the Architectural_Design_Package defines which CSs' stakeholders have access to and which functions they can perform. Detailed stereotypes are summarized in the Table V.8.

Figure V.20 shows an example of a Constituent Diagram used to model the decomposition of the EvacuationCS and its internal entities such as functions (e.g. «Function» *extinguish*, «Function» *manage*) which are associated to one of the main independence classes: the management class with the stereotype «I_Management» *CSManagement* and the operation class with stereotype «I_Operation» *CSOperation*, as well as, each function involves the corresponding stakeholders as attribute, e.g. «Stakeholder» *Manager* and «Stakeholder» *FireFighter*, respectively.

V.4.5 Interaction_Package

The Interaction_Package contains the concepts to describe how flexible collaboration and cooperation take place between different CSs in an SoS. This package's model supports the Interactions

Table V.8: Stereotypes of the Architectural_Design_Package.

Process	Model Kind	UML Diagram	Stereotypes	Description	Meta-class
Architectural Design	Constituent Diagram	Component Diagram	System	Refers to system's class	Class
			CS	Refers to a CS' class	
			SoS	Refers to an SoS' class	
			I_Operation	Independent operations	
			I_Management	Independent management	
			Function	Represents a service	
			Role	Roles of a system	Component
			CS_Role	Roles of a CS	
			SoS_Role	Roles of an SoS	
			Stakeholder	Intervening persons	Property

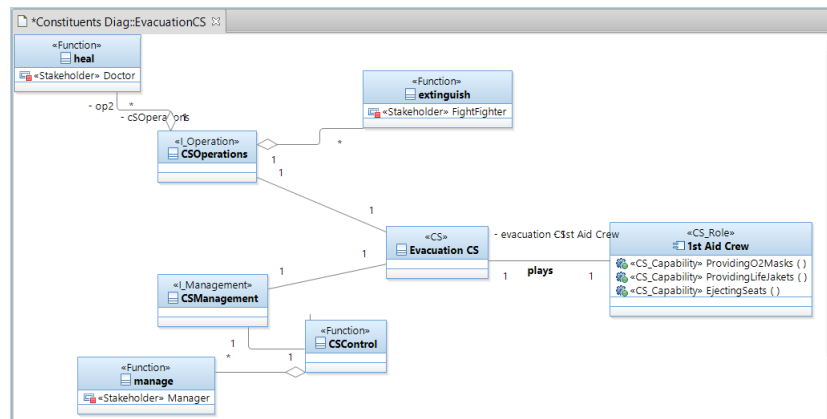


Figure V.20: Constituent diagram for EvacuationCS.

process by focusing on the quality of the interaction architecture and, as a consequence. We define two major Models: The Capabilities Collaboration Model and the Roles Interactions Model. Both determining the types of Collaborations among Collaborative Capabilities and the interactive Roles. The package's stereotyped concepts are represented in Figure V.21.

In the Capabilities Collaboration diagram, we define the stereotypes to describe the internal behavior of different CSs used for fulfilling predefined Goals. The global-Goals of an SoS can be achieved in terms of combining simple Capabilities of its participating CSs. Additionally, the Roles Interactions diagram covers the abstract representations of the collaborative Capabilities of different CSs within an SoS. Moreover, the Role package provides the different relationships that can be used to connect CSs among each other. The identified stereotypes are summarized in Table V.9.

The Figure ?? shows an example of Capabilities Collaboration diagram which describes “AssessingRisks” Capability of «SoS» WarningSoS collaborating with other CSs’ Capabilities («CS_Capability» CO_sensing and «CS_Capability» IR_radiation of «CS» FireDetectionCS) and («CS_Capability» Transducer_probe of «CS» IceDetectionCS). This diagram particularly offers a good method to express the flow of capabilities of the «SoS» WarningSoS and how its CSs can collaborate.

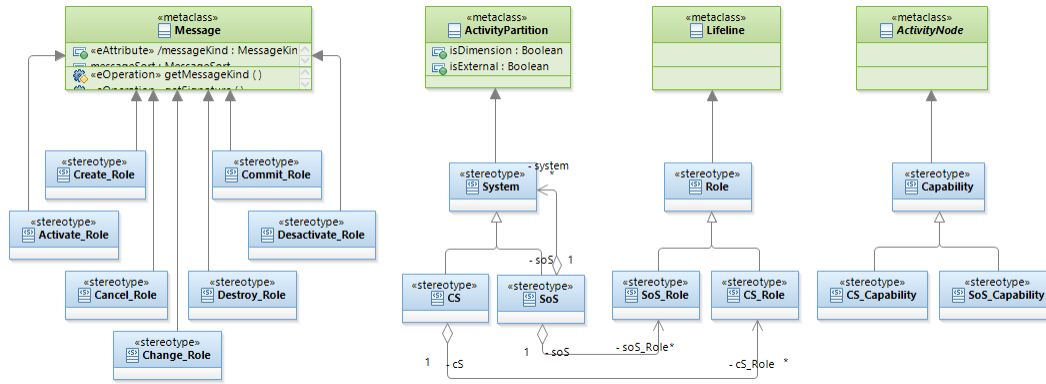


Figure V.21: The structure of the Interaction_Package.

We can use the Roles Interactions diagram Figure ??, to show how the different Roles interact within the *«SoS»AERSoS* when fulfilling the global goal *«SoS_Goal»SafeLanding* in case of critical situations in the aircraft. This diagram depicts a collection of interactions between external Roles of different CSs *«SoS_Role»LandingManager*, *«CS_Role»EmergencyLandingController*, etc. In this case, the Roles represent the specification of a sequence of Capabilities (*«CS_Capability»first_aid*, *«SoS_Capability»landing...etc.*), that an SoS (or CS) can perform. In addition, the roles represent a path or flows of a sequence of interactions (e.g. *«Change_role»Unplanned_Landing*, *«Create_role»pilot...etc.*) that occurs during the execution to accomplish the SafeLanding goal.

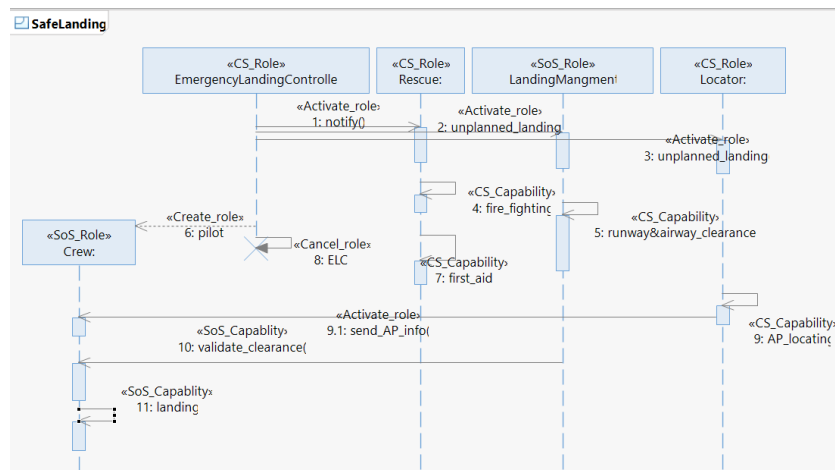


Figure V.22: Roles Interactions diagram for SafeLandingCS' Roles.

V.4.6 Integration_Deployment_Package

The Integration_Deployment_Package (Figure V.23) contains the concepts to describe the process of Deployment, including instances of Roles and the corresponding Interfaces. This model contains the stereotypes of CS' interfaces defined by their Roles and the necessary provided or/and

Process	Model Kind	UML Diagram	Stereotypes	Description	Meta-class
Interactions	Capabilities Collaboration	Activity Diagram	System	System performing Capabilities	ActivityPartition
			CS	CS performing Capabilities	
			SoS	SoS performing Capabilities	
			Capability	functions provided by a System	
			CS_Capability	functions provided by a CS	
	Roles Interactions	Sequence Diagram	SoS_Capability	functions provided by an SoS	ActivityNode
			Role	Interactive Role of a System	Lifeline
			CS_Role	Interactive Role of a CS	
			SoS_Role	Interactive Role of an SoS	
			Capability	Refers to a Capability of a System	Activation
			CS_Capability	Refers to a Capability of a CS	
			SoS_Capability	Refers to a Capability of an SoS	
			Create_role	Initiating a new Role	Message
			Destroy_role	Finishing a Role	
			Activate_role	Starting a Role	
			Deactivate_role	Interrupting a Role	
			Cancel_role	Omitting a Role	
			Change_role	Replacing a Role	
			Commit_role	Performing a Role	

Table V.9: Stereotypes of the Interaction_Package.

required Interfaces (RUIs for Relied Upon Interfaces).

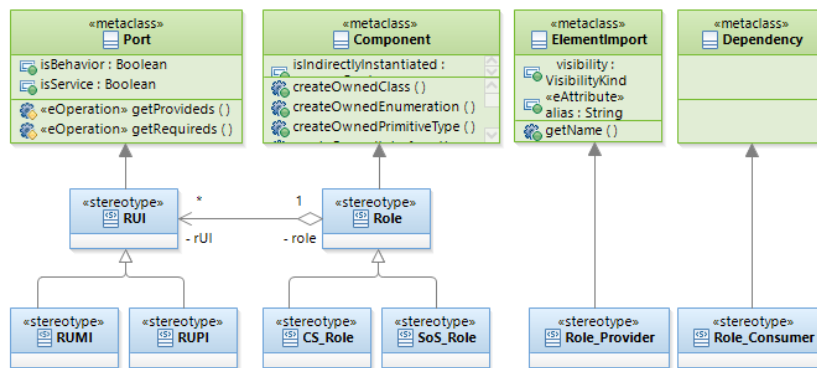


Figure V.23: The structure of the Integration and deployment package.

The different stereotypes in Table V.10 included in this diagram describe the aspect of Integration of an SoS itself. In this case, the Integration and deployment diagram describes the physical deployment of different interfaces required or provided by CSs. The interfaces extend the meta-class port to specify the interaction points among CSs supporting the integration of behavior and structure.

The Figure V.24 shows an example of this diagram of PowerUnitsSoS. the stakeholders assemble a set of Relied Upon Message or/ and Physical Interfaces (e.g. «RUMI»I_OP and «RUPI»I_RP1) and their associations (e.g. «Role_Provider» control_Engines and «Role_Consumer»provide_O2) that constitute the basic elements to define how the CSs of one SoS can collaborate among each other to realize the integration of structure and/or behavior of the SoS.

Process	Model Kind	UML Diagram	Stereotypes	Description	Meta-class
Integration and Deployment	Roles Interfaces	Component Diagram	System	Integrated System	Component
			CS	Integrated CS	
			SoS	Integrated SoS	
			RUI	Relied Upon Interface	Port
			RUMI	Relied Upon Message Interface	
			RUPI	Relied Upon Physical Interface	
			Role_Provider	Role providing a RUI	ElementImport
			Role_Consumer	Role consuming a RUI	Dependency

Table V.10: Stereotypes of the Integration_Deployment_Package.

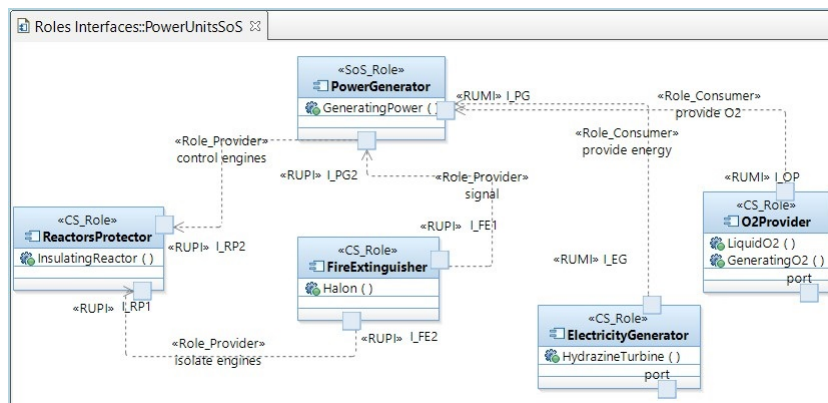


Figure V.24: Roles Interfaces diagram of PowerUnitsSoS.

V.5 Conclusion

In this chapter, we have presented a multi-viewpoint Architecture Framework called SoS-AF which is understandable and easily manipulated by different stakeholders. This methodology aims to offer the audience of SoSs’ Stakeholders the tools to facilitate the task of developing a multi-viewpoint architecture that is managed by a new SoSE process and documented through the SoS-UML profile’s models. Besides, this approach conforms to a very widespread standard in the software architectures community “IEEE 42010” which was designed to standardize the definition of Systems and Software Engineering-Architecture description. Specifically, SoS-AF inherits the definitions of the main elements which are part of this standard, and extends them by the two elements “SoSE process” and “SoS-UML Profile”. We have mainly focused on the construction of multi-viewpoint SoSs’ architectures, through the definition of several viewpoints for a given SoS architecture, which is in our example we used an Aircraft Emergency Response System-of-Systems (AERSoS) as a case study.

The first extension of the standard is to integrate the notions contained in the SoSE process. This adoption allows the SoS-AF to pass through several processes. At the end of each process, each one of the involved stakeholders must raise the level of concretization of his architecture by creating

a set of models that better meet his essential concerns. The stakeholders can benefit from the SoS-UML Profile to support the design of all the concepts and relationships of an SoS' CSs. This tool defines structural diagrams: Goals diagram, Domain model diagram and Constituents diagrams. In addition, to behavioral diagrams which aim to represent the dynamic aspects of an SoS: Capabilities diagrams, Roles Interaction diagrams, Capabilities Collaboration diagrams and Roles Interfaces diagrams. The visual syntax allows using diagrams to manage the SoSE development processes by its audience of stakeholders within the SoS-AF's Viewpoints.

Chapter VI

Formalization of Centralized Control in SoSs

Contents

VI.1	Introduction	101
VI.2	Time-Resource Aware SoSs	102
VI.2.1	Temporal constraints of Missions	102
VI.2.2	Understanding resource categorization	104
VI.2.3	Abstract assets: Variability Domain	106
VI.2.4	Rewriting-based approach for resources allocation control	107
VI.3	Formal semantics of structural entities	108
VI.3.1	Missions and temporal constraints	109
VI.3.2	Resources categorization	110
VI.3.3	Roles encoding	112
VI.3.4	Resource Allocation Controller: RAC	113
VI.4	Formal semantics of dynamic aspects	113
VI.4.1	Missions' lifecycle	114
VI.4.2	Resource' lifecycle	117
VI.4.3	Roles' lifecycle	119
VI.4.4	Resources Allocation Control lifecycle	122
VI.5	Conclusion	126

VI.1 Introduction

Designing well-tuned SoS to deal with a variety of control issues such as resources management and temporal constraints violations while providing high-level assurance about their specified behavior is very challenging. This chapter proposes a Time-Resource aware model for SoSs to address this

complexity. Therefore, we adopt a formal approach to model and verify a centralized controller to manage the specific characteristics of such systems' entities i.e. temporal constraints linked to missions and the production/consumption of resources by each mission using a rewriting-based method. Firstly, we provide a comprehensive description of Time-Resource aware model for SoSs essential for quantifying entities and components with local and global aspects' considerations. Secondly, in order to describe the variability domain of SoSs, we propose a generic Meta-Model named Resource Allocation Centralized Controller (RAC-MM) to design the Time-Resource aware SoSs. The RAC-MM is key in describing the interactions and control mechanisms across various system components such as missions, roles, and resources. Finally, we utilize the Maude language to establish the formal definitions and semantics of RAC-MM's structural and dynamic elements, i.e. by using the Maude language, we integrate their quantitative priorities with their operational semantics, offering a detailed description of their states from the perspective of centralized management, and analyze their configurations through monitoring predicates to enhance the controllability and responsiveness of the SoS.

VI.2 Time-Resource Aware SoSs

In directed SoSs, three key aspects can dictate the efficacy and success of missions: temporal constraints, resource allocation, and centralized control [24], [25]. The first aspect treats the complexities of time management, exploring how various time-related properties(e.g. mission duration, deadlines, etc.) affect the planning and execution of missions. The second aspect highlights the management, allocation, and control of both global and local resources. And the third aspect examines how management and/or control serve to coordinate all these elements. This section introduces a Time-Resource aware model for missions in SoSs, aimed at addressing these frequent temporal challenges. The model acts as an analytical tool, exploring the expressiveness of mission modeling approaches and providing actionable insights to navigate temporal concerns in mission management, thus aspiring to bridge the identified gaps in systematic temporal specification and management in Time-Resource Control Aware SoSs.

VI.2.1 Temporal constraints of Missions

Current methodologies for managing temporal aspects in SoS modeling, reveal critical gaps in addressing mission-critical and time-aware missions. Whereas domains like Cyber-Physical Systems (CPS) and Cloud computing utilize advanced planning tools to manage time constraints proficiently across varied tasks, SoS modeling conspicuously lacks equivalent operational support [25]. In scenarios where SoS manages large-scale emergency responses, like natural disasters, mission success hinges not merely on completion but on rigorous adherence to predefined temporal constraints. For example, even a minor delay in missions, such as deploying search and rescue teams or delivering vital supplies, can yield severe repercussions, obstructing relief efforts and jeopardizing the entire emergency response mechanism of the SoS. In the context of SoS missions, temporal proprieties

dictate a wide array of time-related attributes that are essential for mission planning and execution (Table VI.1).

Table VI.1: Temporal Attributes in SoS Missions.

Temporal Constraint	Description
Clock	The universal time reference used for synchronizing mission execution.
Duration	The expected time window within which a mission is intended to be completed.
Arrival Time	Indicates when a mission becomes active or is slated to commence.
Deadline	The last acceptable moment for the completion of a mission.
Delay	The extent to which a mission can tolerate delays without adverse effects.
WCET	Worst-Case Execution Time; the maximum time a mission might take under the worst-case scenario.
Quit Time	The actual time at which a mission or task is completed.

The management of SoS requires a rigorous understanding of various temporal properties that govern individual missions, as well as their interrelationships. This is crucial for the effective and efficient orchestration of complex missions that often involve multiple CSs with varying degrees of autonomy, control, and interdependence. The temporal properties can be broadly categorized into those related to individual missions, inter-mission relationships, and intra-mission activities.

1. Individual missions Temporal constraints

- Must Start At (MSA): This property requires a mission to initiate at a precisely defined timestamp. Example: A surveillance operation must start exactly at midnight to ensure maximum secrecy.
- Must Finish At (MFA): The MFA property dictates that a mission should conclude at a specified timestamp. Example: A network vulnerability scan must finish by 2 a.m. to minimize disruption to users.
- Must Start In (MSI): Unlike MSA, the MSI property provides a time window during which a mission must begin. Example: These missions must begin within a defined period before the rainy season starts to ensure that responders are well-prepared to manage potential flood events.
- Must Finish In (MFI): This indicates a bounded time interval within which the mission must conclude. Example: Medical evacuation mission has a strict four-hour window to evacuate injured soldiers from the battlefield to a hospital for urgent medical care.
- Duration: Represents the time span an activity within a mission is expected to take. Example: The refueling of an autonomous underwater vehicle must not exceed 20 minutes.
- Recurrent Missions: These are missions that must take place periodically or sporadically. Example: Weather data collection missions can be recurring and take place every six hours.

2. Inter-Mission Temporal constraints

- Cycle: This property denotes the recurring sequence of a set of missions, constrained by time intervals or maximum iterations. Example: Routine system health checks may occur in cycles every 24 hours.
- Temporal Dependency: Represents the time-bound relationship between the start, finish, or intermediate milestones of different missions. Example: The success of a subsequent data analysis mission depends on the timely completion of a data collection mission.

3. Intra-Mission Temporal constraints

- Start-to-Start: Two missions must start simultaneously. Example: Two sensor arrays must begin data collection at the same time.
- Start-to-Finish: The start of one mission signals the end of another one, for example, the start of data analysis should cause the end of data collection.
- Finish-to-Finish: Both missions must conclude at the same time. Example: Calibration and testing must both finish before final checks can commence.
- Finish-to-Start: One mission must finish before another can start. Example: Battery charging must complete before a drone can commence its flight.

VI.2.2 Understanding resource categorization

Efficient resource management lies at the heart of any high-performing and reliable SoS. The nature of an SoS—comprising diverse, interconnected, and often independent systems—makes the task of resource management especially challenging and critical. This section not only underscores the importance of a well-categorized resource management system but also elaborates on the definitions and classifications that constitute the ecosystem of resources within an SoS. In the complex context of SoS, resource management is a key element for performance, reliability, and success. Resources can range widely, from computational power, data storage, and specialized hardware to human operators and communication channels. Effective understanding of these resources is vital for achieving operational missions, maintaining situational awareness, and ensuring system adaptability. This section emphasizes the importance of resource categorization in an SoS environment and discusses their main characteristics.

Global resources are shared among CSs and present challenges due to their limited capabilities. Mismanagement of these resources can lead to serious consequences, including missions failures that compromise the overall missions of the SoS. On the other hand, local resources are specific to individual systems and are crucial for their local missions. Any inefficiency or mismanagement in this level can create bottlenecks that negatively impact the entire SoS. Consequently, the challenges raise from the interaction between both global and local resources. These complications could result to mission delays or even complete failures. To address these challenges, a well-defined categorization system is crucial to facilitate predictive analytics, proactive conflict resolution, and real-time control management.

The categorization of resources acts as the cornerstone for effective management in SoS. Given the complex interplay between global and local resources, a well-defined classification system is essential for optimizing performance, reliability, and success[50]:

VI.2.2.1 global Resources

Global resources are shared across all constituent systems in the SoS, and their effective management is crucial for achieving collective objectives. Failure to properly manage these resources has ramifications that extend throughout the SoS.

- Limited (L): resources whose availability is limited and which, once exhausted, must be replenished or replaced. For example, satellite time could be a limited resource shared between different navigation and communications CSs.
- Limited but Renewable (LR): resources that can reach their limits but can be regenerated or replenished over time. For example, solar energy systems could be a renewable resource used in multiple CSs.
- Unlimited but Shareable (US): resources that have no inherent limits, but nevertheless require coordination for optimal use. One example could be a cloud-based data storage system that multiple constituent systems can access.

VI.2.2.2 Local Resources

On the flip side, local resources are confined to individual systems. While they are not shared among systems, poor management of local resources can create bottlenecks that affect the entire SoS.

- Limited (L): resources in the context of a single system and, once depleted, must be replenished. For example, medical supplies in an emergency vehicle could be a limited local resource.
- Limited but Renewable (LR): These are resources within a single system that can be regenerated or recycled. An example might be a team of first responders who can attend to multiple emergencies but need periods of rest.
- Unlimited but Shareable (US): abundant resources within single CSs units that still require effective management. For example, a database of patient medical records may be unlimited but requires proper management for optimal system performance.

The challenge in this chapter lies in balancing global and local resources to avoid conflicts and ensure availability, and taking into account the temporal constraints of each mission to prevent systemic failure. Therefore, understanding these categories allows for robust predictive analytics, proactive conflict resolution strategies, and adaptive real-time management.

VI.2.3 Abstract assets: Variability Domain

As seen in Chapter IV, DE focuses on the specification and realization of reusable assets, which include architectures, requirements, etc. By employing Variability Domain techniques, we can specify the features and specific characteristics of directed SoSs with central control and ownership of CSs, thereby easing the specification and design of these Time-Resource Models of SoSs through the description provided by the RAC-MM Meta-Models.

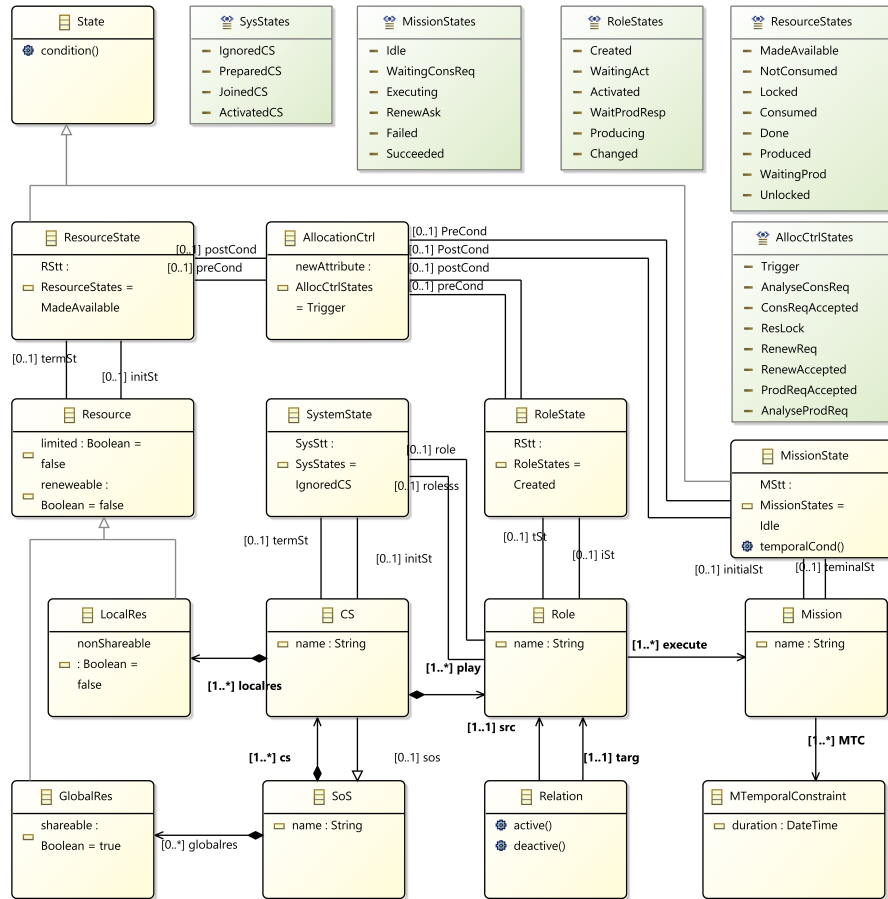


Figure VI.1: Resource Allocation Controller Meta-Model, RAC-MM.

The RAC-MM Meta-Model depicted in Figure VI.1 serves as a comprehensive framework for the architectural characterization and design of diverse temporal missions and entities, including Time constraints, Resource categories, and centralized control mechanisms. As mentioned in the approach section, RAC-MM is designed to encapsulate the variabilities of this specific type of SoSs, where centralized control and management are key. This is specified in the RAC-MM. The latter represents a comprehensive structure of RT-missions' entities, consisting of interconnected states and transitions of these elements. At the core, it outlines a State class, which includes various conditions and serves as a parent to more specialized states such as ResourceState, which tracks the availability and conditions of resources. Resources themselves are characterized by their limitations

and the possibility of renewal, with distinctions made between local resources, which may or may not be shareable, and global resources, which are inherently shareable across the system [25].

On the one hand, the CSs are associated with both local and global resources, indicating a layered structure of resource allocation and usage. Each CS can play multiple roles within the SoS, as depicted by the Role class, which is in a created state by default. These roles are then executed within Missions, encapsulating the functional aspect of the SoS where missions have specific temporal conditions. On the other hand, the SoS itself encompasses global resources. Relationships between different entities are managed by the Relation class, which includes operations to activate or deactivate these relationships, thereby providing a dynamic aspect to the interconnections within the SoS.

The system's dynamic behavior is further detailed by various states like MissionStates, RoleStates, and SystemStates, each encompassing different stages such as idle, executing, and created, respectively. These stages are pivotal for tracking the progress and flow of operations within the SoS. The AllocationCtrl class oversees the resource allocation process, guided by pre- and post-conditions to ensure a smooth transition between states and the efficient use of resources. i.e. the AllocationCtrl serves as a mechanism for managing and controlling the allocation of resources within the SoS and its CSs. It defines the logic and conditions under which resources are assigned and utilized by various components of the system, such as missions or roles. The AllocationCtrl contains attributes for new allocation states and is responsible for triggering state changes. It operates based on predefined conditions (PreCond and PostCond), which ensure that resources are allocated correctly and efficiently. The AllocCtrlStates further specifies the various states of the allocation control process, ranging from the initial request for resources (ConsReqAccepted) to the renewal and production requests (RenewReq, ProdReqAccepted). This suggests a dynamic and responsive allocation system that adapts to the operational demands of the system, ensuring that resources are allocated based on current needs and availability.

VI.2.4 Rewriting-based approach for resources allocation control

The proposed methodology for resource management employs Maude's rewriting logic to dynamically and autonomously control resources. It is supported by Variability Domain that extends its Application Domain integrating static structures with dynamic behaviors expressed with the different quantitative properties, thus enabling an adaptable and autonomous SoS. This Meta-Model allows for the representation of specific Directed SoS architectures and facilitates the simulation and verification of system behaviors within Maude. The approach progresses from defining static entities such as Missions, Resources, and Roles to modeling their behaviors and interactions Figure.VI.2, encompassing a systematic method for managing the behaviors of resources.

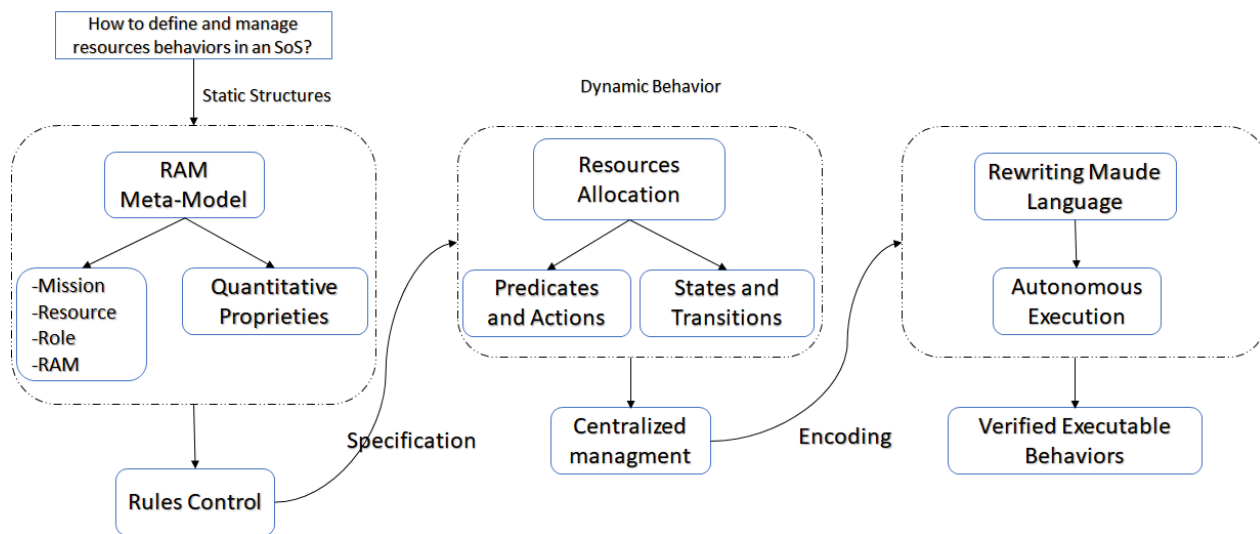


Figure VI.2: *SoSs resource management modeling.*

- **Static Structures:** describe the fundamental entities within the SoS—Missions, Resources, Roles, and the RAC itself—each with specific quantitative properties. This static configuration forms the key concepts upon which the dynamic behaviors are contextualized and managed. It is where the missions and their lifecycle, resource types with associated properties, and roles with their functional capacities are defined.
- **Dynamic Behavior:** this is the stage where operational semantics are applied, encoding the behaviors into Maude’s syntax through the use of predicates and actions that facilitate the transitions among states of each entity. The resource allocation process is thus executed, reflecting a centralized management system that is autonomously governed by the encoded rewrite rules.
- **Autonomous execution and verification:** the final step of this approach is the autonomous execution of behaviors within the Maude framework, enabling the SoS to self-control and adapt to changes dynamically. The formal executable semantics of Maude ensure that the behaviors adhere to the specifications, with verification mechanisms in place to validate the correctness of the operational logic.

VI.3 Formal semantics of structural entities

RAC-MM is a generic Meta-Model to address the challenges of resource consumption and production in modeling SoSs. It consists of concepts (mission, role, resource, etc. that provide straightforward modeling of SoSs applications. In this section, we propose a semantics mapping between RAC-MM static concepts and Maude constructs to describe all RAC-MM components and their structural aspects.

In this section, the RAC Meta-Model static semantics are translated into a set of Object-Oriented

classes. The latter encompass objects with attributes, typing semantics (sorts), subsort relations, and algebraic operations used to define the syntactic structure of various elements in SoSs like Mission, Roles, Resources, and their proprieties.

VI.3.1 Missions and temporal constraints

Missions present the applications executed in different environments in SoSs. We consider missions as the central entities in our modeling, they represent the entry point of roles, and they need access to different resources types that are deployed in different environments. In Maude specification, the Missions are specified as objects having sets of attributes that describe the states and their transitions.

The specification of each mission includes the necessary information to describe the mission's structure, checking predicates, temporal constraints, and their violation signals, all of which are specified in the RT-Maude object-oriented module SoS_CLASSES_MOD. The latter contains all the necessary declarations defining the class Mission, which is characterized by its timeline and operational parameters. This class is defined by several attributes that describe the mission's state and progress, alongside its temporal constraints and resource requirements. This class is defined by the following Maude declaration:

```

1  class Mission |
2      localClock    : Time,      --- Tracks the mission local timeline.
3      duration      : Time,      --- Total allocated time for the mission.
4      arrivalTime   : Time,      --- Scheduled start time of the mission.
5      quitTime      : Time,      --- end time of the mission.
6      delay         : Time,      --- Incurred delays.
7      deadline      : Time,      --- Ultimate completion deadline.
8      missionState  : MissionState, --- Current state of the mission.
9      resType       : ResType,   --- Type of required resources.
10     RAUT          : Time,      --- Remaining Available Use Time.
11     sg            : Sig,       --- Signals for temporal constraints.
12     rs            : Rs .       --- Specification of required resources.

```

The localClock attributes in this class serves as a real-time chronometer, essential for tracking the mission's progress and ensuring adherence to schedules. The duration outlines the total operational timeframe, while arrivalTime and quitTime mark the starting and ending, framing the mission's execution state. Any deviations from this timeline are captured by the delay attribute, providing a realistic view of the mission's execution against planned schedules. The deadline enforces a strict time limit for mission completion, ensuring time discipline. The missionState dynamically reflects the mission's current phase. Resource management, a pivotal aspect, is detailed through resType and RAUT (Remaining Available Use Time), indicating resource types and usage efficiency. The rs specifies the exact resources required for the mission's success. Together, these attributes create a robust structure for effective mission management, encompassing time, resources, and operational status.

VI.3.2 Resources categorization

Resources in SoSs are fundamental, serving as key support elements for missions and CSs. In Maude’s specifications, these resources are represented as objects with detailed attributes. These attributes define their states, transitions, availability, capacity, and interactions with missions. Moreover, resources are categorized based on properties such as being limited, unlimited, renewable, or non-renewable, which is crucial for a dependencies-based approach in system execution. This approach aids in managing interactions, resolving resource conflicts, and recommending corrective actions. The specification of these resources is critical for both local and global missions since they ensure alignment with the system’s functional requirements at both global and local CSs.

In the Maude specification, Resources are defined as objects with attribute sets that describe their status for use by CSs, distinguishing between Local Non-Shareable and Global Shareable Resources. These resources undergo various states and transitions, comprising the resource consumption cycle. The OO_module SoS_CLASSES_MOD encapsulates all necessary declarations for the Resource class, detailing attributes related to structure, transitions, and properties. This class includes attributes that track time and specify the resource’s type and state. Resource types are divided into local or global categories, each with distinct states indicating characteristics like capacity, shareability, and renewability.

```

1 class Resource |
2   localClock : Time,      --- Time relevant to the resource s usage.
3   res        : ResourceType, --- Specifies whether the resource is local/ global.
4   resP       : ResProp,   --- Properties of the resource.
5   resSt      : ResState,  --- Current state of the resource.
6   resCapacity : Time .   --- Total time capacity of the resource.

```

The Resource class in Maude is defined with several attributes, each serving a specific purpose:

- **localClock**: This attribute is responsible for keeping track of the time that is relevant to the resource, primarily used for monitoring its usage or availability over time.
- **res (ResourceType)**: This attribute categorizes the resource as either local or global. A local resource is confined to individual CSs, while a global resource is accessible across the entire SoS.
- **resP (ResProp)**: Encompasses the properties of the resource, such as its limitations, shareability, and renewability.
- **resSt (ResState)**: This attribute reflects the current state of the resource, indicating whether it is in use, available, or in some other state.
- **resCapacity**: It denotes the total available time capacity of the resource, crucial for understanding how long the resource can be utilized before it needs replenishment or becomes unavailable.

Moreover, using sort ResourceType, the class categorizes resources as either local localRes or global globalRes.

```

1 sort localRes globalRes ResourceType .
2 subsort localRes globalRes < ResourceType .
3 ops localRes globalRes : -> ResourceType [ctor] .
```

- **localRes**: Represents resources that are specific and confined to individual systems. These resources are not shared across the SoS and are typically used for local operations within a single CS.
- **globalRes**: Refers to resources that are shared and accessible across the entire SoS. These resources are available for use by multiple constituent systems and play a crucial role in collaborative operations within the SoS.

Each type of the previous resources has specific properties `localResProp` or `globalResProp` associated with them, i.e. `ResProp` may include attributes such as `limited`, `nonShareable`, `limitedButRenewable` for local resources, and `unlimited`, `shareable`, `renewable` for global resources, each influencing how resources are allocated and managed within missions.

```

1 sort ResProp ResState .
2 subsort localResProp globalResProp < ResProp .
3 ops localResProp globalResProp : -> ResProp [ctor] .
4 ops limited nonShareable limitedButRenewable: -> localResProp [ctor] .
5 ops unlimited shareable renewable: -> globalResProp [ctor] .
```

The operations `limited`, `nonShareable`, and `limitedButRenewable` are constructors (`ctor`) for the `localResProp` sort in the Maude specification. These operations define the specific properties that a local resource can have:

- **limited**: Indicates that the resource has a fixed capacity or availability that cannot be exceeded.
- **nonShareable**: Specifies that the resource cannot be used by multiple missions or entities concurrently.
- **limitedButRenewable**: Denotes that the resource, while having a limited capacity, can be replenished or its availability extended.

Similarly, the operations `unlimited`, `shareable`, and `renewable` are constructors for the `globalResProp` sort. These are used to define the properties of global resources:

- **unlimited**: This indicates that the resource does not have a fixed capacity and can be used extensively without depletion.
- **shareable**: Specifies that the resource can be used by multiple missions or entities at the same time.
- **renewable**: Indicates that the resource can be renewed or replenished after consumption or after its initial allocation period has ended.

These properties are essential for the modeling of resources as they control how resources can be allocated, consumed, and managed within the system. They form the basis for the rules and logic that manage the resource lifecycle in the Maude specification, influencing how the system responds to various resource-related events and conditions.

VI.3.3 Roles encoding

Roles within SoSs encapsulate the missions performed by various CSs, directly influencing the execution of missions by producing the necessary resources. Moreover, Roles are dynamic, they not only respond to activation or deactivation requests but also adapt to the environment by transitioning between various states.

In the Maude specification, roles are depicted as objects with attributes that outline their states and transitions. The specification details the role's structure, activation conditions, resource production capabilities, and process analysis. Role specification is encoded in the Maude OO-module SoS_CLASSES_MOD, which includes necessary declarations for the class Role. Each role in this class is characterized by its state and operational parameters, with attributes defining these aspects.

```
1
2 class Role |
3   localClock : Time,      --- Monitors the role s relevant timeline.
4   roleStt    : RoleStt,   --- Current state of the role.
5   resType    : ResType,   --- Type of resources the role is capable of producing.
6   roleSg     : RoleSig,   --- Signals for role state changes.
7   prodCap    : Capacity . --- Capacity for resource production.
```

- **localClock**: Functions as a timekeeper, vital for managing the timeline of the role's activities.
- **roleStt**(Role State): Indicates the current operational state of the role.
- **resType**: Identifies the type of resources that the role is designed to produce.
- **roleSg**: Emits signals that correspond to state changes or events within the role's lifecycle.
- **prodCap**: Defines the production capacity of the role, determining the volume of resources it can generate.

The actions within the Role class are designed to depict the interactions or processes that a role can engage in:

```
1 sort Role .
2 subsort actReq actRep < RoleAction .
3 ops actReq actRep : Role Role -> RoleAction [ctor] .
```

- **actReq** (Activation Request): This action symbolizes a request to initiate the role's functions or missions within the system.
- **actRep** (Activation Response): It represents a response to an activation request, signaling the start of the role's active participation in the system.

VI.3.4 Resource Allocation Controller: RAC

The RAC in a Directed SoS effectively balances the dual aspects of consumption and production. Its primary function is to synchronize the states of Missions and Roles with available Resources. This synchronization includes managing consumption requests from Missions and overseeing the production contributions from Roles. To this end, the RAC acts as a comprehensive controller to ensure a strict operation by maintaining a state-based correspondence between Missions, Roles, and Resources. It employs a series of their predicates and actions that guide decisions on whether to continue with or modify the current states of these entities, thereby facilitating efficient resource allocation and utilization.

The RAC extends the three previous classes Mission, Resource, and Role to capture and execute the decision-making logic required for efficient resource management. Specified within the SoS_CLASS_MOD module, the RAC embodies allocation and control semantics, effectively integrating and executing resource management strategies within the system as follows:

```

1  class ResourceAllocationManager |
2      mission      : Oid,          --- Identifier for the Mission entity.
3      role         : Oid,          --- Identifier for the Role entity involved in production.
4      resource     : Oid,          --- Identifier for the Resource entity being managed.
5      racState    : RacState,     --- The current state of the RAC in the allocation cycle.

```

- **mission**: This attribute stores the identifier of the Mission that is requesting resources or is currently utilizing them.
- **role**: This attribute holds the identifier of the Role that is involved in producing or replenishing resources.
- **resource**: This attribute identifies the specific Resource that is the subject of allocation or production.
- **racState**: Reflects the dynamic state of the RAC, which determines the subsequent actions in the resource management process.
- **resMangA**: Represents a set of allocation actions that the RAC can execute based on the evaluated predicates.

VI.4 Formal semantics of dynamic aspects

In this section, we present a key aspect of our approach, we introduce a full description of the states of the previous static entities and concepts from the perspective of centralized control in SoSs. We analyze their configurations through monitoring predicates and outline an approach to the formal specification of autonomous execution of dynamic behavior modeling using Maude's rewriting logic.

VI.4.1 Missions' lifecycle

This section initially defines the mission attributes. Subsequently, it provides various actions and predicates to capture their states, followed by modeling their dynamic behavior.

VI.4.1.1 States, predicates, and actions encoding

From a state perspective (i.e. `missionState` attribute in class `Mission`), each mission instance during its lifetime goes through different states. After its `Idle` state and if its TCs are respected, it moves to `waitConsResp` and waits for an answer from the RAC controller after sending an allocation message. At this moment, a resource may be allocated to this mission instance, and it moves to the state `executing`, then to `succeeded` after finishing the execution. In case of unavailability of resources (or absence), the `Mission` will receive a `reject` message from the controller and return to the `Idle` state. If the `Mission` encounters any unexpected problem, it can ask the controller to extend the availability time of the resource by sending the message `isAskRenewing` to ask for a renewal of the allocation of resources. And then, the controller can reply by sending a `renewAskOk` message, If the renewal request is accepted, or by sending `isFailed` message if the request is denied and the mission reaches the `Idle` state, and at the end, the mission instance will send an exit message to the controller and it will move to `Idle` state.

In order to specify the different missions' states and the possible transitions among each other, The `MissionState` sort defines possible states that a mission can be in throughout its lifecycle:

```
1  sort MissionState .
2  ops idle:    --- The mission is not yet active.
3  failed:    --- The mission has ended unsuccessfully.
4  waitConsResp: --- Awaiting a response for resource consumption.
5  executing:  --- The mission is currently in execution.
6  succeeded:  --- The mission has been completed successfully.
7  rnwAsk:    --- A request to renew resource consumption has been made.
8  : -> MissionState [ctor] .
9
10 ops isConsReqSent isConsReqAccepted isConsReqRejected : M R -> Bool .
```

Since each state is critical as it determines the next steps in the mission's execution, we identify a set of predicates that enable or disable the different transitions in a given mission instance, as shown in Table [VI.2](#).

Table VI.2: *Mission Predicates in Maude.*

Maude encoding		Description
(a) P_{Mis}	isConsReqSent(M,R)	Verifies if a resource consumption request has been sent for M and R.
(b) P_{Mis}	isTCR(M)	Checks if M is adhering to its defined temporal constraints.
(c) P_{Mis}	isConsReqAccepted(M,R)	Confirms the approval of a resource consumption request for M and R.
(d) P_{Mis}	isConsReqrejected(M,R)	Indicates the denial of a resource consumption request for M and R.
(e) P_{Mis}	isFailed	Used to indicate that a mission has not achieved its objectives or has encountered significant problems.
(f) P_{Mis}	isMnV(M)	Triggers if the M's duration is shorter than the specified minimum.
(g) P_{Mis}	isMxV(M)	Signals if M exceeds its maximum allowed duration.
(h) P_{Mis}	isAskRenewing(M,R)	Indicates a request for extending or renewing R for M.
(i) P_{Mis}	isFinished(M)	Confirms the completion of M.
(g) P_{Mis}	isRenewOk(M)	Signifies the approval of a renewal request for M.
(k) P_{Mis}	isRenewRejected(M)	Indicates the rejection of a renewal request for M.

Moreover, a set of specific allocation actions is employed which are instrumental in transitioning missions through different states based on the previous predicates and conditions. Below is a Table VI.3 introducing these actions, providing a clear understanding of their roles:

Table VI.3: *Mission Actions in Maude.*

Maude encoding		Description
(1) A_{Mis}	sendConsReq(M,R)	Initiates a resource consumption request for M and R, marking the start of the allocation process.
(2) A_{Mis}	returnIdle(M)	Resets M back to the Idle state, used for restarting or temporarily stopping the mission.
(3) A_{Mis}	execute(M)	Transitions M into an active state, starting its operational activities.
(4) A_{Mis}	Exit(M)	Marks the completion of M, concluding its current operational cycle.
(5) A_{Mis}	renew(M)	Extends the duration of resources for M, facilitating the continuation of its activities.
(6) A_{Mis}	cancel(M)	Aborts or halts M, used in situations where the mission needs to be stopped immediately.

Lastly, we use the Sig sort to represent the signal events associated with a mission's temporal constraints.

```

1  sort Sig .
2  ops isTCR
3  isMnV
4  isMxV : -> Sig [ctor] .
    
```

- These signals are used to indicate whether a mission is adhering to its time-based requirements or if any violations have occurred:
- isTCR: Indicates that the mission's temporal constraints are respected.
- isMnV: Signifies that the mission's minimum duration constraint has been violated.
- isMxV: Shows that the mission's maximum duration constraint has been violated.

VI.4.1.2 Mission Behavior modeling

To effectively manage and respond to various operational scenarios in a Maude system. The latter combines a set of monitoring predicates which provides insights into the current state of a mission instance. the Figure VI.3 is a visual representation that maps out the possible states of a

mission and the transitions between them, based on the evaluation of predicates and the execution of actions. It provides a blueprint for the mission's behavior in response to various predicates and conditions.

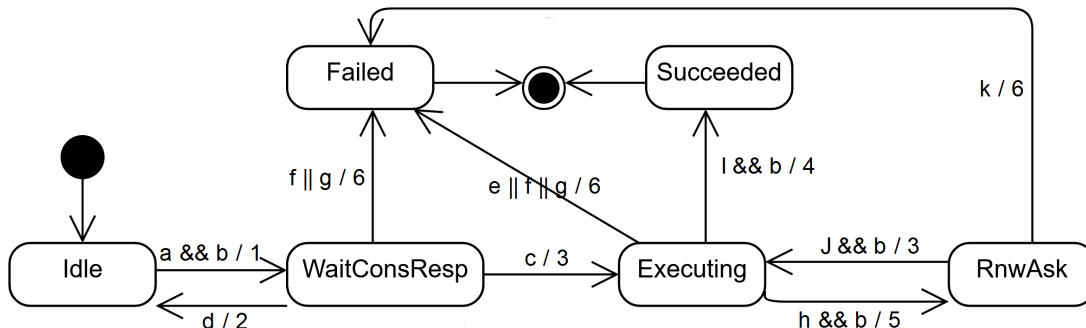


Figure VI.3: *Mission's transition system.*

In the Maude system, the behavior of mission instances is orchestrated through a structured set of predicates and actions illustrated in both Table VI.2 and VI.3, respectively. These are encoded as $P_{Mis}()$ for predicates and $A_{Mis}()$ for actions¹. Predicates act as the necessary conditions or 'guards' that need to be satisfied for state transitions to occur. They are intricately designed to monitor the mission's adherence to its temporal constraints and resource usage.

The RAC manager relies on these predicates to make decisions on whether to enable or disable certain actions for a mission instance. For example, a predicate might check if a mission is currently in a state where it should be waiting for resource allocation (waitConsResp), and if so, the corresponding action to send a resource request (sendConsReq(M,R)) is enabled.

Maude's powerful feature of conditioned rewrite rules comes into play to dynamically manage the state transitions of the missions. These rules are conditionally applied based on the evaluation of the $P_{Mis}()$ predicates. A typical rewrite rule in Maude has the following syntax:

```
1 crl [rule-name] : < Mid : Mission | state : S, other-attributes... > => < Mid : Mission | state : S,
  other-attributes... > if PMis() .
```

Here, [rule-name] is a unique identifier for the rule, S and S' represent the current and next states of the mission, respectively, and PMis() is the predicate that must be true for the transition from state S to state S' to take place. These rewrite rules are inherently reactive—they automatically respond to changes in the system's state and adjust the mission's state accordingly. Through the interplay of these predicates and actions, encapsulated in the rewrite rules, Maude effectively simulates the complex behavior of missions in response to a variety of scenarios, ensuring that each mission progresses through its lifecycle as intended, with its state transitions and resource consumption.

¹For convenience and simplicity, note that in this figure and the upcoming statecharts, all transitions have a set of Predicates that are denoted by letters before the “/”, and Actions that are denoted by numbers after it.

VI.4.2 Resource' lifecycle

Similarly to the previous section, this one initially defines the Resource attributes. Subsequently, it provides various actions and predicates to capture their states, followed by modeling their dynamic behavior.

VI.4.2.1 States, proprieties, predicates and actions encoding

Similar to Missions, Resources within SoSs transition through a variety of states as they are produced, allocated, consumed, and eventually released. The lifecycle of a resource is governed by the ResState sort, which defines the possible states that a resource can inhabit throughout its existence. The ResState sort is declared with the following operations, which act as constructors (ctor) for the states:

```
ops madeAvailable notConsumed locked consumed done produced waitingProd unlocked : -> ResState [ctor] .
```

- **Produced:** The resource has been created and is ready to be made available.
- **WaitingProd:** The resource is pending production before it can be made available.
- **MadeAvailable:** The resource is now available for allocation and use.
- **Locked:** The resource has been allocated to a mission and is currently not available to others.
- **Consumed:** The resource is actively being used by a mission.
- **Unlocked:** The resource has been released from its locked state and is again available.
- **Done:** The resource has served its purpose and is now inactive.
- **NotConsumed:** The resource was available but not used before moving to the Done state.

For each state, unique predicates and actions are identified to facilitate transitions (See TableVI.4 and TableVI.5). These elements are vital for guiding the next states in both a resource's lifecycle and missions.. Based on the provided structure and descriptions, below are the two tables representing them within the Maude system.

Table VI.4: Resource’s Predicates in Maude.

Maude encoding		Description
(a) P_{Res}	isAvailable(R)	Checks if R is currently available for allocation.
(b) P_{Res}	isConsApproved(R)	Confirms the approval of a consumption request for R.
(c) P_{Res}	isGlobalRes(R)	Indicates if R is a global resource.
(d) P_{Res}	isLocalRes(R)	Determines if R is a local resource.
(e) P_{Res}	isConFinished(R)	Verifies if the consumption process of R is complete.
(f) P_{Res}	isUnbounded(R)	Denotes that R has an unlimited capacity.
(g) P_{Res}	isNotAvailable(R)	Signifies that R is not available for consumption.
(h) P_{Res}	isLimited(R)	Indicates that R has a limited capacity.
(i) P_{Res}	isUnlimited(R)	Specifies that R has unlimited capacity.
(j) P_{Res}	isRenewable(R)	Signifies that R’s availability can be renewed.
(k) P_{Res}	isStopped(R)	Denotes that R is no longer active.
(l) P_{Res}	isLogical(R)	Classifies R as a logical resource.
(m) P_{Res}	isPhysical(R)	Classifies R as a physical resource.
(n) P_{Res}	isProductionApplied(R)	Indicates R is in the production phase.
(o) P_{Res}	isProductionFinished(R)	Confirms that the production phase for R is complete.
(p) P_{Res}	isNotRenewable(R)	Specifies that R cannot have its duration extended.

Table VI.5: Resource’s Actions in Maude

Maude encoding		Description
(1) A_{Res}	prepareCons(R)	Prepares R for the consumption process.
(2) A_{Res}	consume(R)	Initiates the consumption process for R.
(3) A_{Mis}	lock(R)	Locks R, preventing it from being consumed or allocated.
(4) A_{Res}	unlock(R)	Unlocks R, making it available for consumption.
(5) A_{Res}	update(R)	Updates the status or properties of R.
(6) A_{Res}	withdraw(R)	Withdraws R from availability.
(7) A_{Res}	waitProdReq(R)	Puts R on hold while awaiting a production request.
(8) A_{Res}	produceF(R)	Initiates the production process for R.
(9) A_{Res}	makeAvailable(R)	Marks R as available for allocation and consumption.

VI.4.2.2 Resource Behavior Modeling Using Rewrite Rules

In a Maude system, resources are managed and monitored through the above set of predicates and actions. These elements provide insights into the current state of a resource instance and guide the transitions between various states. The provided figure (Figure VI.4) serves as a visual representation that outlines the possible states of a resource and the pathways between them. This mapping is based on the evaluation of predicates and the execution of corresponding actions, providing a blueprint for the system’s response to different events and operational scenarios.

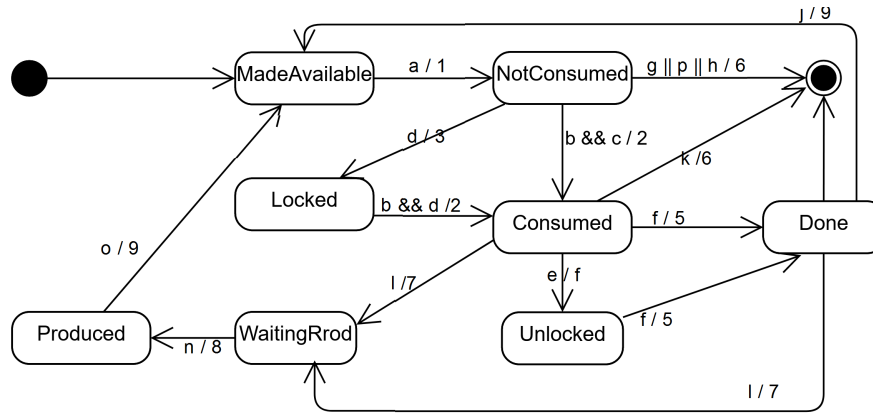


Figure VI.4: Resource's transition system.

In the formal Maude specification, the behavior of resources is directed by a well-defined set of predicates $\text{PRes}()$ and actions $\text{asARes}()$. These predicates function as necessary conditions or 'guards' that must be met for state transitions to take place. They are employed to monitor the resource's compliance with its designated properties and constraints.

The Resource Controller utilizes these predicates to inform decisions about enabling or disabling certain actions for a resource instance. For instance, a predicate could assess whether a resource is in a state to be made available (madeAvailable), and if so, the corresponding action to lock the resource for use ($\text{lock}(R)$) could be triggered. For this end, Maude leverages its feature of conditioned rewrite rules to dynamically manage the transitions of resource states. These rewrite rules are applied conditionally, contingent upon the evaluation of $\text{PRes}()$ predicates. A typical resource rewrite rule in Maude is expressed with the following syntax:

```

1  crl [rule-name] : < Rid : Resource | state : S, other-attributes... > => < Rid : Resource | state : S,
    other-attributes... > if PRes() .
    
```

In this structure, rule-name is a unique identifier for the rule. S and S' denote the current and subsequent states of the resource. $\text{PRes}()$ signifies the predicate condition that must be validated for the state transition from S to S' to be executed. These rules are adaptive and automatically responding to changes in the system's state, thus updating the resource's status as necessary. This ensures that each resource undergoes its lifecycle as planned, with transitions, consumption, and production.

VI.4.3 Roles' lifecycle

Similarly to the previous section, this one initially defines the Roles attributes. Subsequently, it provides various actions and predicates to capture their states, followed by modeling their dynamic behavior.

VI.4.3.1 States, Predicates, and Actions Encoding for Roles

For roles in the Maude system, as in Missions and Resources, their lifecycle is characterized by various states and transitions to fulfillment of their roles. The states encapsulate the role’s activation from being created, awaiting activation, becoming active, and eventually producing resources or changing state based on the system’s needs. The encoding of roles within the SoS_CLASSES_MOD module in Maude defines the states and transitions, facilitated by a set of predicates and actions that determine the progression of roles through their lifecycle. The RoleStt sort defines these states, and operations within this sort serve as constructors for the states:

```

1 sort RoleStt .
2 ops created waitingActivation activated changed waitProdResp producing : -> RoleStt [ctor] .

```

Each state corresponds to a stage in the lifecycle of a role:

- created: The role has been instantiated but is not yet activated.
- waitingActivation: The role is ready and waiting for a signal to activate.
- activated: The role is currently active, performing its designated tasks.
- changed: The role has experienced a change, possibly requiring a reassessment of its activities or status.
- waitProdResp: The role is in a standby state, awaiting a response to a production-related request.
- producing: The role is engaged in the production of resources.

To manage the transitions between these states, a set of predicates (Table VI.6) evaluate specific conditions of a role, and actions (Table VI.7) are taken to move the role to the next appropriate state.

Table VI.6: Role’s Predicates in Maude.

Predicate ID	Maude encoding	Description
(a) P_{Rol}	isActivationReqSent(RL)	Checks if an activation request for role RL has been issued.
(b) P_{Rol}	isActivationReqRejected(RL)	Indicates that RL’s activation request has been rejected.
(c) P_{Rol}	isActivationReqAccepted(RL)	Confirms that RL’s activation request has been accepted.
(d) P_{Rol}	isProductionReqSent(RL)	Checks if a production request has been sent for RL.
(e) P_{Rol}	isProductionReqRejected(RL)	Indicates rejection of a production request for RL.
(f) P_{Rol}	isProductionReqOk(RL)	Confirms that the production request for RL is approved.
(g) P_{Rol}	isRoleChanged(RL)	Indicates that RL has undergone a change.
(h) P_{Rol}	isInactive(RL)	Checks if RL is currently inactive.
(i) P_{Rol}	isProductionFinished(RL)	Verifies if RL has completed its production.

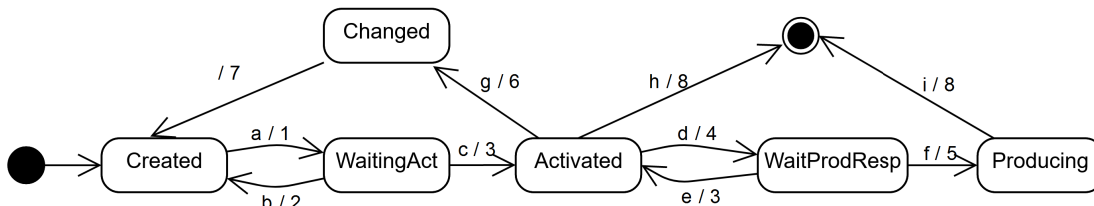
Table VI.7: Role's Actions in Maude.

Action ID	Maude encoding	Description
(1) A_{Rol}	waitActivation(RL)	Puts RL in a waiting state for activation to be processed.
(2) A_{Rol}	rejectActivation(RL)	Handles the rejection of RL's activation.
(3) A_{Rol}	active(RL)	Transitions RL to an active state for performing its functions.
(4) A_{Rol}	waitProdReq(RL)	Puts RL in a waiting state for a production request to be processed.
(5) A_{Rol}	produce(RL)	Initiates the production process by RL.
(6) A_{Rol}	changeR(RL)	Reflects a change in RL's state, possibly requiring different actions.
(7) A_{Rol}	create(RL)	Creates a new instance of RL in the system.
(8) A_{Rol}	exit(RL)	Handles the RL's exit from the active system following production completion.

These predicates and actions are designed to correspond directly with the states outlined in the RoleStt sort. When a predicate is evaluated to be true, it triggers an action that results in a state transition for the role.

VI.4.3.2 Role Behavior Modeling Using Rewrite Rules

To manage and respond effectively to different operational scenarios within a Maude system, it is essential to have a structured approach to monitoring and controlling the state transitions of roles. Similar to missions, roles are subjected to various states reflecting their activities and responsibilities within the SoS. The state chart for roles, as illustrated in the provided Figure VI.5, offers a visual mapping of the potential states and transitions based on the evaluation of predicates and the execution of actions. This chart is pivotal as it outlines a blueprint for the system's behavior in reaction to a variety of events and conditions.


Figure VI.5: Role's transition system.

In the Maude formal system, the behavior of roles is governed by a set of predicates and actions. These are encoded as $PRol()$ for predicates and $ARol()$ for actions. Predicates function as the necessary conditions or 'guards' that facilitate state transitions, ensuring that roles progress according to the system's operational demands and constraints. The RAC, which oversees role activities, uses these predicates to determine whether to enable or disable specific actions for a role instance. For instance, a predicate might assess if a role is in a state where it should be activated (waitingActivation), and if so, the corresponding action to initiate activation ($active(RL)$) would be permitted.

Conditioned rewrite rules within Maude play a crucial role in dynamically managing the state transitions of roles. These rules are conditionally applied based on the predicates' evaluation. The syntax for a rewrite rule in Maude is as follows:

```
1  crl [rule-name] : < RId : Role | state : S, other-attributes... > => < RId : Role | state : S,  
   other-attributes... > if PRol() .
```

In this syntax, [rule-name] is a unique identifier for the rule. S and S' represent the current and subsequent states of the role, while PRol() is the predicate that must be satisfied for the transition to occur. The rewrite rules are reactive by nature, automatically responding to system state changes and adjusting the role's state as required. Through the interplay of predicates and actions within these rewrite rules, Maude simulates the complex behavior of roles, ensuring each role fulfills its lifecycle as planned.

VI.4.4 Resources Allocation Control lifecycle

Incorporating the dual aspects of consumption and production, the RAC serves as an intermediate manager within an SoS. It is designed to synchronize the states of Missions and Roles with the available Resources, effectively managing both the consumption requests by Missions and the production contributions by Roles.

VI.4.4.1 RAC States, Predicates, and Actions Encoding

From a state perspective, each instance of the RAC during its operation goes through a sequence of states. Starting from the trigger state, the RAC may transition to analyseConsReq when it begins to process a resource consumption request. Once the conditions for resource allocation are met, it may move to waitConsAccp to await confirmation of the request. Upon acceptance, the RAC transitions to consReqAccepted, and it may then proceed to prodReqAccepted when a production request is confirmed. The RAC can transition to renewReq and upon acceptance, to renewAccepted. In the event of encountering issues or upon the completion of the allocation cycle, the RAC may return to the trigger state, ready to process new requests. The RacState sort in Maude reflects these possible states:

```
1  sort RacState .  
2  ops trigger analyseConsReq waitConsAccp consReqAccepted renewReq renewAccepted prodReqAccepted  
   analyseProdReq : -> RacState [ctor] .
```

To guide the decision-making process in resource management, the operational logic of the RAC in Maude is centered around a transition system controlled by predicates (PRC) and actions (ARC). The RAC's functionality is driven by rewrite rules, which are conditionally activated based on these predicates. These predicates and actions are derived from the existing series of Missions, Roles, and Resources. They play a critical role in determining whether to maintain or modify the current states of these entities. Here's a Table VI.6 that outlines the predicates used to determine the transitions of the RAC:

Table VI.8: RAC Predicates in Maude.

Maude encoding	Description	Based on Mission, Role, and Resource Predicates
(a) P_{RAC}	Verifies if a resource consumption request is sent	(a) P_{Mis}
(b) P_{RAC}	Checks if temporal constraints are respected	(a) P_{Mis} , (b) P_{Mis} , (a) P_{Res} , (b) P_{Res} , (c) P_{Res}
(c) P_{RAC}	Confirms consumption request approval	(a) P_{Mis} , (b) P_{Mis} , (a) P_{Res} , (d) P_{Res}
(d) P_{RAC}	Indicates denial of a consumption request	(a) P_{Mis} , (b) P_{Mis} , (b) P_{Res} , (d) P_{Res}
(e) P_{RAC}	Indicates a problem or failure in processing	(h) P_{Mis}
(f) P_{RAC}	Indicates a renewal request is sent	(h) P_{Mis} , (j) P_{Res}
(g) P_{RAC}	Indicates a production request is sent	(p) P_{Res}
(h) P_{RAC}	Indicates a role-related condition	(d) P_{Rol}
(i) P_{RAC}	Indicates a complex condition involving roles	(f) P_{Rol} , (n) P_{Rol}

The RAC employs a specific set of allocation actions (Table VI.7) that are instrumental in transitioning through its different states:

Table VI.9: RAC Actions in Maude.

Maude encoding	Description	Based on Mission, Role, and Resource Actions
(1) A_{RAC}	Initiates a resource consumption request	(1) A_{Mis}
(2) A_{RAC}	Processes a resource allocation	(3) A_{Mis} , (2) A_{Res}
(3) A_{RAC}	Updates resource status	(2) A_{Res}
(4) A_{RAC}	Accepts a resource consumption request	(3) A_{Mis} , (2) A_{Res}
(5) A_{RAC}	Processes internal RAC activity	(3) A_{Mis}
(6) A_{RAC}	Processes a resource renewal request	(3) A_{Mis} , (9) A_{Res}
(7) A_{RAC}	Updates resource status based on renewal	(6) A_{Res}
(8) A_{RAC}	Processes a role's production request	(4) A_{Rol} , (7) A_{Res}
(9) A_{RAC}	Executes a production-related role action	(8) A_{Rol}

These predicates and actions are encoded within the Maude system as part of the RAC's class to facilitate the dynamic management of resource allocation and consumption. They ensure that the RAC can handle various scenarios effectively, from normal operation to exception handling and recovery.

VI.4.4.2 RAC behavior modeling Using Rewrite Rules

In the Maude system, the RAC is pivotal for managing interactions among missions, roles, and resources. It handles resource requests from missions and orchestrates resource production by roles, ensuring efficient allocation. The behavior of the RAC is encoded in Maude's language, using objects, classes, and rewrite rules. These rules, governed by specific conditions or predicates, direct state transitions, enabling the RAC to dynamically manage resource allocation and maintain system balance.

The RAC operates by transitioning through various states, the predicates (PRC) and actions (ARC) described are specific to the RAC and are based on the predicates and actions related to missions (PMis, AMis), roles (PRol, ARol), and resources (PRes, ARes). The RAC's behavior is

visualized through statecharts in Figure VI.6, showing potential states and their transitions, which are controlled by these predicates and actions. These elements are formalized as conditional rewrite rules in Maude, allowing the RAC to adaptively respond to system changes and effectively manage resources. The RAC’s role encompasses:

- **State Management:** Monitoring and adjusting its state and those of missions, roles, and resources.
- **Predicate Evaluation:** Assessing situations, like resource requests or compliance with temporal constraints.
- **Action Execution:** Implementing actions based on predicate outcomes to transition the system’s state.
- **Resource Management:** Balancing resource needs of missions with roles’ production capacities for efficient allocation.
- **Rewrite Rules:** Implementing rules in Maude for real-time adaptation and resource management.

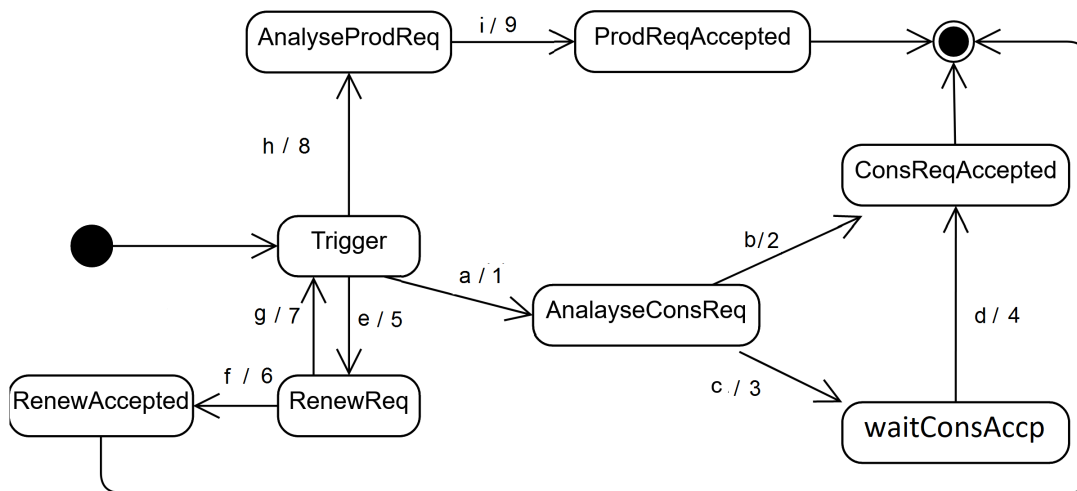


Figure VI.6: RAC’s transition system.

In the provided context, the Resource Allocation Manager (RAC) in Maude leverages a set of rewrite rules governed by predicates (PRC) and actions (ARC). These rules integrate the fundamental predicates and actions associated with Missions (PMis, AMis), Roles (PRol, ARol), and Resources (PRes, ARes). Each rule is structured to evaluate conditions and execute actions that influence the system’s state transitions and resource management processes. The predicates and actions specific to the RAC (PRC, ARC) are directly linked to the core elements of missions, roles, and resources, ensuring a cohesive and adaptive resource management strategy. This setup enables

the RAC to effectively handle the dynamic requirements of missions, roles, and resources within the Maude system. The conditional rewrite rules in this case will be as follow:

Rewrite rule for allocating a local non-shareable resource: The RAC checks that the global resource `Rid` is available `PRes(isAvailable(Rid))`, is a global resource `PRes(isGlobal(Rid))`, and is marked as shareable `PRes(isShareable(Rid))`.

```

1  crl [allocate-global-shareable-resource] :
2    < RCid : ResourceAllocationManager | mission : Mid, resource : Rid, racState : trigger >
3    < Rid : Resource | resSt : Available, resP : globalResProp(shareable) >
4    < Mid : Mission | state : waitConsResp >
5    =>
6    < RCid : ResourceAllocationManager | mission : Mid, resource : Rid, racState : consReqAccepted >
7    < Rid : Resource | resSt : Shared >
8    < Mid : Mission | state : Executing >
9    if PRes(isAvailable(Rid)) and PRes(isGlobal(Rid)) and PRes(isShareable(Rid)) .
    
```

In this rule:

- If the resource is available and shareable, its state transitions to `Shared`, allowing multiple missions from different CSs in the SoS to access it concurrently.
- The mission `Mid` can now execute as it has access to the required resource.

Local Non-Shareable Resource Allocation: for allocating a local non-shareable resource, the RAC must ensure the resource is not already allocated before locking it for a mission.

```

1  crl [allocate-local-nonshareable-resource] :
2    < RCid : ResourceAllocationManager | mission : Mid, resource : Rid, racState : trigger >
3    < Mid : Mission | missionState : waitConsResp, resType : localRes >
4    < Rid : Resource | resState : available, resP : localResProp(limitedButRenewable), resType : localRes >
5    =>
6    < RCid : ResourceAllocationManager | mission : Mid, resource : Rid, racState : consReqAccepted >
7    < Mid : Mission | missionState : executing >
8    < Rid : Resource | resState : locked >
9    if PRes(isAvailable(Rid)) and PRes(isNonShareable(Rid)) and not PRes(isLocked(Rid)) .
    
```

In this example: the rule checks if the local resource is available, non-shareable, and not currently locked. If all conditions are met, it allocates the resource by changing its state to `locked` and the missions state to `executing`.

Resource production: Example of rewrite rule for the RAC managing resource production

```

1  crl [produce-resource] :
2    < RCid : ResourceAllocationManager | role : Rlid, resource : Rid, racState : prodReqAccepted >
3    < Rlid : Role | roleStt : waitingActivation >
4    < Rid : Resource | resSt : NotConsumed >
5    =>
6    < RCid : ResourceAllocationManager | role : Rlid, resource : Rid, racState : renewAccepted >
7    < Rlid : Role | roleStt : activated >
8    < Rid : Resource | resSt : Produced >
9    if PRol(isActivationReqAccepted(Rlid)) and PRes('isNotRenewable(Rid)) .
    
```

These rewrite rules are based on the predicates `PRes` which are functions that check the state of resources `resState`, properties `resP`, and the type `resType` to determine the availability and shareability as per the conditions given in the RAC operation.

VI.5 Conclusion

In this chapter, we have presented the formalization of centralized control in SoSs, we have seen how the Meta-Model RAC-MM provides a comprehensive framework for addressing the complicated dynamics of resource management in real-time SoS environments, integrating at the same time the temporal constraints, resource categorization, and centralized management mechanisms, which are essential for the effective allocation in SoSs. More specifically, we have explained the importance of state management, dynamic behavior modeling, and the coordination of RAC. The formal semantics defined by the Maude language enable precise specification and verification of behaviors, offering a systematic approach to face the challenges posed by resources' complexities. In the following chapter, we explore the implementation and practicalities of the RAC within SoSs. We will explore how the theoretical and formal models presented here can be translated into strategic management to govern different executions in SoSs' workflows.

Chapter VII

Control-based Formalization of Management Strategies

Contents

VII.1	Introduction	127
VII.2	Abstract assets: Variability Domain for Management	128
VII.3	Strategic Management of Behavior	129
VII.3.1	Managing Workflows in SoSs	131
VII.3.2	Mission execution and resource management	133
VII.3.3	Management Strategies	142
VII.4	Real-Time regulating mechanism using MAPE-K loop	146
VII.4.1	Knowledge: Data Foundation	148
VII.4.2	Monitor: Processing	148
VII.4.3	Analysis: Workflow	149
VII.4.4	Plan: Strategic Control and Management	149
VII.4.5	Execution: Rewriting Management System	150
VII.5	Conclusion	153

VII.1 Introduction

Desired behaviors refer to the intended actions and missions that align with the system's objectives, facilitating effective operation and achievement of goals. Conversely, unwanted behaviors represent actions or outcomes that deviate from planned intentions, potentially leading to inefficiencies or conflicts within the SoS. In this chapter, we introduce a strategic management approach

to handle the desired and unwanted behaviors within SoS through strategic planning. The chapter starts with an examination of the workflow and functional chains in SoSs, which are crucial for the systematic execution of missions. Workflows describe the sequence and organization of interconnected tasks aimed at completing specific missions, while functional chains represent a more specific sequence of missions that ensure that these workflows are executed. We use these two elements to maintain the coherence and efficiency of SoS operations, providing a strategical framework for mission execution.

Moreover, we incorporate the MAPE-K (Monitoring, Analysis, Planning, and Execution and Knowledge) autonomic loop to facilitate the real-time management of SoS, enabling the system to self-monitor, analyze ongoing operations, plan strategically in response to changes, and execute necessary adjustments, all while continuously updating its knowledge base to reflect current system states and external conditions.

We employ the Maude language to offer a formal method to define operational semantics across functional, system, and strategy modules. It addresses the complexities of workflow management, mission execution, resource allocation, and conflict resolution. The proposed method aims to improve system functionality and resource efficiency in complex SoS environments, showcasing its applicability through detailed examples and theoretical foundations.

VII.2 Abstract assets: Variability Domain for Management

Building on the fundamental Meta-Model presented in the previous chapter, we have refined and enriched the RAC-MM to handle the different features related to the variabilities of directed SoSs. This enhancement supports the robustness of the variability domain, offering a more comprehensive structure for SoSE. The new version of the Meta-Model is illustrated in Figure VII.1 and it mainly serves as a structured reference for workflow management within a SoS. It is designed to orchestrate complex interactions between various CSs. At the core of the Meta-Model is the SoS class, which oversees the execution of workflows. The latter is designed and governed by a series of behavior rules that describe the different desired and unwanted behaviors within the system.

These Behaviors are classified into two main types: desired and unwanted, each associated with specific rules that are time-based or resource-based. These rules are encapsulated within their respective enumerations and are leveraged by strategies to enforce or prevent certain behaviors. The Strategy class forms the foundation for two derived classes: MainStrat and AuxiliaryStrat. The MainStrat class represents the primary strategies that are essential for workflow execution, while AuxiliaryStrat provides additional support to govern and respond to dynamic functional conditions.

The main strategies within the MainStrat class, such as mission-based and resource-based functional chain strategies, determine the execution path of the workflow, guiding it toward completing its goals. These strategies are robust and incorporate the auxiliary rules to enhance the system's management and control and ensure that the workflow remains resilient in the face of changing internal and external conditions.

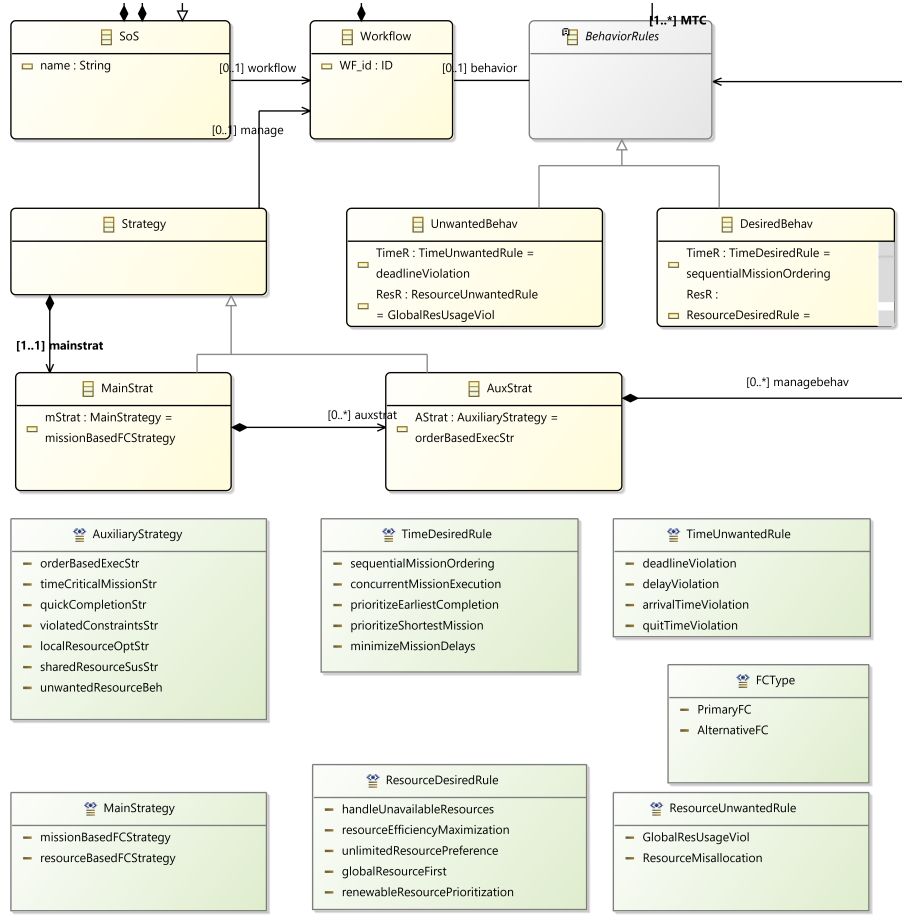


Figure VII.1: Management Strategies Meta-Model, MS-MM.

VII.3 Strategic Management of Behavior

In this section, we summarize the specification of the proposed self-management approach, giving the operational semantics of different elements introduced MS-MM using Maude. In its basic form, the latter is equipped with two types of modules: functional modules and system modules. By using the extension of Strategy Language for Maude [57], we add the strategies' module to formalize the semantics of MS-MM, the three types of modules are summarized below:

- Functional Modules ($(\Sigma_{MS-MM}, E_{MS-MM} \cup A_{MS-MM})$): They specify Maude equational theory, which is based on mathematical logic of belonging, where Σ_{MS-MM} is the signature that specifies the typed structure of each element of a system (sorts, subtypes, operators, etc.), E_{MS-MM} is the collection of equations (possibly conditional) declared acting on the structure, A_{MS-MM} is the collection of equational attributes declared for different operators, like the attribute of commutativity or associativity between elements.
- System Modules ($(\Sigma_{MS-MM}, E_{MS-MM} \cup A_{MS-MM}, R_{MS-MM})$): Describe a Maude rewriting theory where $(\Sigma_{MS-MM}, E_{MS-MM} \cup A_{MS-MM})$ represents the equational theory of a

system, R_{MS-MM} is the set of rewriting rules (possibly conditional) that are applied to change the behavior of a system, and thus evolve its equational structure.

- Strategy Modules ($\Sigma_{MS-MM}, E_{MS-MM} \cup A_{MS-MM}, S_{MS-MM}$ (R_{MS-MM}, SM_{MS-MM})): They define a set of strategies that control and guide the application of rewriting rules in Maude. $(\Sigma_{MS-MM}, E_{MS-MM} \cup A_{MS-MM})$ represent the equational theory of the system that is supposed to evolve. S_{MS-MM} is a semantics describing the behavior of a system, it is constructed from a set R_{MS-MM} containing rewriting rules, and a strategy module SM_{MS-MM} that will guide the rewriting and use of these rules using strategies.

In this chapter, we accomplish two main objectives. Firstly, we formalize self-management strategies and their associated rewriting rules, aimed at autonomously managing the various structural and behavioral elements outlined in the MS-MM meta-model (Figure. VII.1). Secondly, we ensure that these strategies adhere to the operational semantics relevant to the different architectural entities discussed in Chapter IV. Therefore, the self-management strategies of MS-MM are encoded and seamlessly integrated within Maude's established functional and system modules. This integration results in the introduction of five complementary modules that collectively englobe the semantics of MS-MM, summarized below:

- Functional Module WORKFLOW-DATA: its role is to define the structural syntax of the WF of Directed SoSs by implementing the functional chain elements (i.e. different executing missions) in the form of sorts and operations. This structure allows for the construction of a complete structured process that begins with individual missions and finishes with one or more global missions, resulting in a structured missions workflow enabled by the systematic organization of temporal dependencies and resources.
- The System Modules, TIME-BASED-EXEC and RESOURCE-BASED-EXEC: both of them include the WORKFLOW-DATA module to implement a set of rules and conditions for the MS-MM, this enables the execution of missions and the utilization of resources. The former, TIME-BASED-EXEC, is primarily focused on planning and executing missions based on temporal attributes and conditions, while the latter, RESOURCE-BASED-EXEC, is tailored to optimize resource allocation and utilization. These modules encapsulate the semantics provided by a suite of rewriting rules, facilitating the execution of a suitable functional chain by selecting of the optimal choices. For this, conditional rewrite logic (CRL) in Maude is incorporated, enabling them to dynamically adapt to the execution of missions and the allocation of resources based on evolving conditions and needs.
- The FUNC-CHAIN-STRAT module integrates TIME-BASED-EXEC and RESOURCE-BASED-EXEC modules to formulate strategies that direct and plan SoS execution. Its primary function is to enforce specific CRLs to self-regulate behaviors, ensuring missions align with objectives and resources. It effectively avoids undesired behaviors and selects optimal mission paths based on criteria such as arrival time, duration, and resource availability.

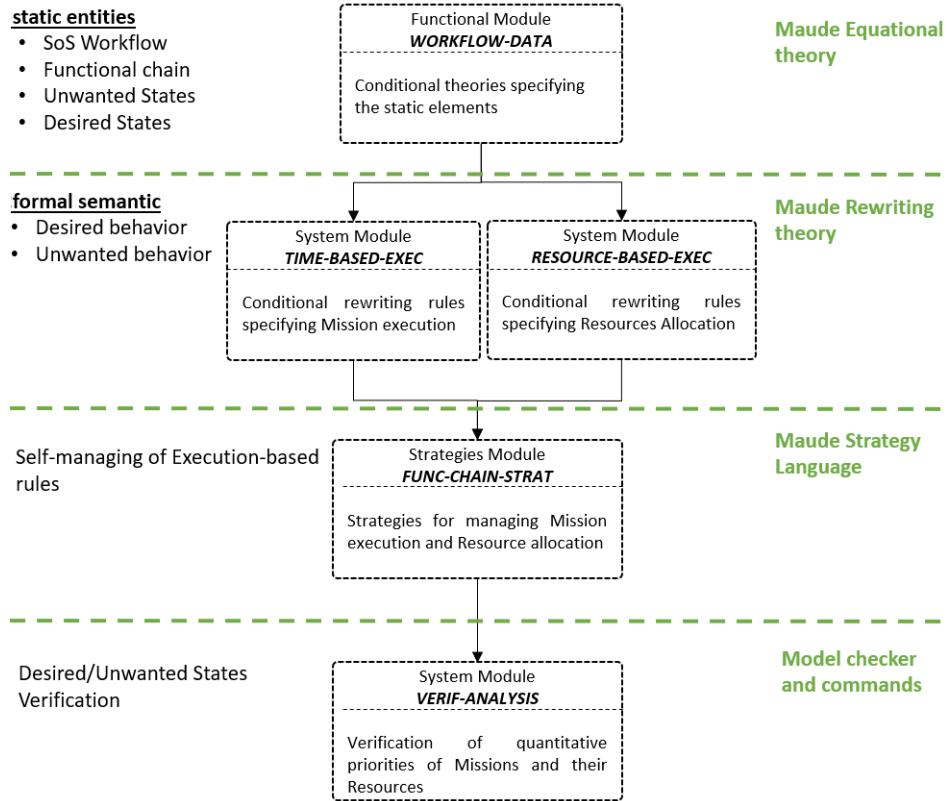


Figure VII.2: MS-MM operational semantics in Maude Strategy Language.

- System VERIF-ANALYSIS Module: it describes a set of LTL properties, introduced to verify the desired/unwanted behavior related to time and resource constraints of the SoS. These properties are analyzed using the Maude model-checker tool and more precisely search commands, which will perform a verification of a property on a simulation from an initial state of a SoS to a final state, applying the set of CRLs.

Implementing specifications in Maude language using rewrite logic provides significant flexibility, extensibility, and reuse. This is achieved through a modular approach, in which Maude-based modules can be integrated with others, allowing easy extension or independent editing of their specifications. More specifically, by leveraging conditional rewrite logic, the system maintains the operational flexibility and resilience essential to managing the complexities of large-scale integrated systems.

VII.3.1 Managing Workflows in SoSs

Within the SoS Workflow (WF), missions are not limited to a single CS for execution. Instead, multiple CSs, each with unique capabilities and resources, can participate in different ways for the same mission. This approach introduces a rich model of parallelism, where similar missions are executed concurrently but under varying constraints and resource conditions. This diversity in

resource availability and operational constraints among the CSs adds layers of complexity to the Workflow, necessitating advanced coordination and resource management.

VII.3.1.1 Workflow and Functional Chains description

In SoSs' context, we define the concept of WF as a comprehensive model that structures a set of FCs of missions to accomplish specific goals. The WF model includes FCs which are specific sequences or combinations of missions designed to execute a part of the overall WF. These elements are characterized by their:

- Dynamics: unlike traditional linear task sequences, FCs are flexible and diverse, they allow for various execution modes such as exclusive, concurrent, or sequential.
- Collaborative CSs: FCs depend largely on the collaborative efforts of the CSs which are working collaboratively execute and manage the chains.
- Strategic Mission and Resource Management: the ability of CSs to adapt to changing scenarios (e.g., resource constraints, shifting priorities, or external environmental changes) ensures that the WF remains resilient and aligned with global SoS goals.

The WORKFLOW-DATA module is designed to establish structured WFs and their associated FCs. This module is primarily centered around (1) linking various FCs and integrating them into the overarching WF structure, and (2) classifying these chains into two distinct categories: primary and alternative, allowing for defining the operational behavior and managing missions' priorities within the WF. We use WORKFLOW-DATA module to:

- Construct FCs and prepare them for execution.
- Classify missions as 'Primary' or 'Alternative'.
- Identify and address desired behavior and avoid or correct unwanted one.
- Integrate and enable the two system modules TIME-BASED-EXEC and RESOURCE-BASED-EXEC to develop strategies for executing a series of missions.

To this end, the WORKFLOW-DATA module introduces a systematic classification of elements and types that are fundamental to manage SoS' WFs and their FCs. This classification is accomplished through the definition of sorts such as Mission, SecOrPri, FuncChain, and Workflow, etc. which form the structure of these elements (Listing below). Each sort serves a distinct purpose: Mission encapsulates individual missions or operations, while SecOrPri represents a secondary or primary categorization of these missions. The FuncChain sort particularly acts as a generic structure under which both missions and their classifications (secondary or primary) are nested.

```
1 fmod WORKFLOW-DATA is
2   sorts Mission SecOrPri FuncChain Workflow .
3   subsorts Mission SecOrPri < FuncChain .
```

```

4
5   var M1 : Mission .
6   op DelayViolated DeadlineViolated isWCETViolated isArrivalTimeViolated: Time Time -> Bool .
7   op isResourceUnavailable isGlobalResourcePreferred isResourceNonShareable isResourceLimited : Resource ->
      Bool .
8   op isTCViolated : Mission -> Bool .
9
10
11  ops AlternativeM PrimaryM : -> SecOrPri [ctor] .
12  op __ : FuncChain FuncChain -> FuncChain [ctor assoc comm] .
13  op _|_ : FuncChain FuncChain -> Workflow [ctor comm prec] .
14
15  vars FC1 FC2 : FuncChain .
16  op initial : -> Workflow .
17  eq initial = AlternativeM M1 | PrimaryM .
18
19  --- Temporal and Resource Constraints
20  ceq DelayViolated(Clock, ExpectedDelay) = true if Clock > ExpectedDelay .
21
22  ...
23  endfm

```

Depending on scenarios where resource constraints and time-sensitive missions are involved, the classification of missions into primary (PrimaryM) and alternative (AlternativeM) allows for the prioritization of certain flows of execution over others:

- PrimaryM: represents a subset/subsequence of missions that are given priority in terms of short duration and/or resource constraints during the runtime to achieve the final global Mission.
- AlternativeM: represents a subset/subsequence of missions that are not given priority for the current execution, these missions could still exist in the system and could be relevant in future executions.

Moreover, in order to enable real-time responses to changes in WF, the module employs a set of conditional equations to analyze the complexities (e.g. time and resource management). It defines equations such as DeadlineViolated, isResourceUnavailable, and DelayViolated, which work on monitoring the system's compliance with its operational parameters. These conditions act as triggers for rewriting rules in other system modules, allowing the SoS to adapt its behavior in response to environmental changes. The table VII.1 introduces the different conditional equations for applying constraints and managing the WF in an SoS. These equations are used to evaluate and respond to various scenarios, and to ensure that missions meet defined time constraints and resource availability.

VII.3.2 Mission execution and resource management

The two system modules TIME-BASED-EXEC and RESOURCE-BASED-EXEC, collaboratively prioritize missions to execute the optimal FC, aligning with desired behavior, preferences, functional objectives and unpredictable scenarios. The former module determines 'what' missions to execute

Table VII.1: Function Descriptions and Maude's Code

Function Name	Description	Maude's code
isWCETViolated	Checks if the Worst Case Execution Time (WCET) constraint is violated.	<code>ceq isWCETViolated(Clock, WCET)= true if Clock > WCET</code>
isArrivalTimeViolated	Determines whether the arrival time constraint of a mission is violated.	<code>ceq isArrivalTimeViolated(Clock, ExpectedArrivalTime)= true if Clock < ExpectedArrivalTime</code>
isResourceUnavailable	Evaluates if a specified resource is currently unavailable.	<code>ceq isResourceUnavailable(R)= true if R's state indicates unavailability</code>
isGlobalResourcePreferred	Checks if a global resource is preferred for a given mission based on certain conditions.	<code>ceq isGlobalResourcePreferred(Mid)= true if /* condition to check global resource preference for mission Mid */</code>
isResourceNonShareable	Determines if a specified resource is non-shareable.	<code>ceq isResourceNonShareable(Rid)= true if /*condition to check if resource Rid is non-shareable */</code>
isResourceLimited	Checks whether a resource is limited in terms of its availability or capacity.	<code>ceq isResourceLimited(Rid)= true if /*condition to check if resource Rid is limited */</code>
isDeadlineViolated	Assesses if a mission has exceeded its deadline.	<code>ceq isDeadlineViolated(Clock, Deadline)= true if Clock > Deadline</code>
isDelayViolated	Checks if there's a delay beyond the expected duration for a mission.	<code>ceq isDelayViolated(Delay, ExpectedDuration)= true if Delay > ExpectedDuration</code>
isQuitTimeViolated	Evaluates if a mission has exceeded its quit time.	<code>ceq isQuitTimeViolated(Clock, QuitTime)= true if Clock > QuitTime</code>
isMissionUnscheduled	Determines if a mission is executing without being scheduled properly.	<code>ceq isMissionUnscheduled(M)= true if < M : Mission</code>
isResourceImproperlyAllocated	Checks if a mission's required resource has been allocated improperly (e.g, the resource is unavailable).	<code>ceq isResourceImproperlyAllocated(M)= true if < M : Mission</code>

and 'when', focusing on sequencing and timing to optimize mission execution. The second module complements this by managing 'how' these missions are resourced and 'with what', ensuring efficient resource allocation. Together, they facilitate the execution of the most advantageous functional chain or the most adapted to the specific needs and priorities of WF.

VII.3.2.1 Temporal condition-based prioritization of Missions: TIME-BASED-EXEC module

Including the WORKFLOW-DATA functional module, the TIME-BASED-EXEC system module prioritizes and manages missions according to their time constraints, guaranteeing efficient and rapid execution. It manages the sequential and simultaneous processing of missions, meeting the dynamic requirements of the system. It incorporates rules to address violations of time constraints and resource limitations representing undesirable behavior, therefore, it also demonstrates its ability to simulate and manage missions in an environment where time is a critical and unpredictable factor. Integrating the functionalities of the TIME-BASED-EXEC module with those of conditional equations defined in the WORKFLOW-DATA module creates a dynamic system within Maude. This integration facilitates a more responsive approach to scheduling, prioritizing, and managing missions based on their temporal constraints of operational parameters. This includes:

- The system prioritizes missions based on urgency and potential impact, and aligns them with strategic goals.
- The system sets criteria for mission completion, duration, delays etc, triggering appropriate subsequent actions, thus contributing to the SoS's progression and evolution.
- Rules manage sequential and simultaneous mission executions.
- Using conditional equations from WORKFLOW-DATA, the rules are also used to manage situations where missions deviate from set temporal constraints or when operational conditions change unexpectedly.
- The conditional equations in WORKFLOW-DATA can adapt and reconfigure its operations to maintain efficiency and continuity.

To fully understand this module, it is essential to recognize how it classifies missions as primary and alternative. The module's rules are designed to build a functional chain, focusing on behaviors such as shortest and earliest execution etc, while dynamically adjusting mission status between primary and alternate as needed. These rules allow the system to manage missions efficiently, adhering to predefined conditions, as follows:

- Ordered Mission Sequencing

The rule *sequentialMissionOrdering* is used for scenarios where missions are in sequence order, it ensures that each mission follows one another in a predetermined order. This is especially

important for missions that depend on the completion of a previous mission or when a certain sequence is necessary for successful execution.

```

1  r1 [sequentialMissionOrdering] : M1 M2 AlternativeM | PrimaryM =>
    AlternativeM | PrimaryM M1 M2 .

```

In this rule, missions M1 and M2 are arranged in a sequential order, both M1 and M2 need to be classified or added as primary missions. The rule ensures that if they are currently in a mixed arrangement with AlternativeM or PrimaryM, they will be reordered to maintain this sequential execution, respecting the primary status.

- Parallel Mission Execution

The concurrent mission execution rule *concurrentMissionExecution*, allows parallel execution of missions. It is employed when missions are independent of each other and can be executed simultaneously. This rule is beneficial for optimizing the use of time, as it allows multiple operations to take place simultaneously without any impact on each other.

```

1  r1 [concurrentMissionExecution] : M2 M3 AlternativeM | PrimaryM M1 =>
    AlternativeM | PrimaryM M1 M2 M3 .

```

This rule facilitates the concurrent execution of M2 and M3 after finishing M1, by moving them to PrimaryM FC for concurrent execution.

- Start Time Prioritization

The rule for *prioritizeEarliestStart* ensures that missions with earlier start times are given precedence over others. This prioritization is critical in environments where timing is of the essence, and starting earlier can lead to more efficient overall mission management.

```

1  cr1 [prioritizeEarliestStart] :
2  M2 M3 AlternativeM | PrimaryM M1 => if (getArrivalTime(M2) <
    getArrivalTime(M3)) then AlternativeM M3 | PrimaryM M2 M1
3  else AlternativeM M2 | PrimaryM M3 M1
4  fi .

```

This conditional rule evaluates the arrival times of M2 and M3. The mission with the earlier start time is prioritized as a primary mission, while the other is reclassified as alternative, thus restructuring the functional chain's construction and execution order.

- **Completion Time Prioritization** The *priorEarliestCompletion* rule classifies missions based on their expected completion times, it ensures that missions scheduled to end earlier are prioritized in the execution schedule. This rule is particularly useful when it is important to complete shorter or urgent missions first.

```

1  cr1 [prioritizeEarliestCompletion] : M2 M3 AlternativeM | PrimaryM M1 =>
    if (getQuitTime(M2) <= getQuitTime(M3)) then AlternativeM M3 |

```

```

    PrimaryM M2 M1
2  else AlternativeM M2 | PrimaryM M3 M1
3  fi .

```

This rule focuses on quit times, rearranging M2 and M3 based on which mission is expected to be completed earlier. The mission with the earlier completion time is set as primary, and the other as alternative.

- **Mission Duration Prioritization** The *priorShortestMission* rule is employed to prioritize missions of shorter duration, it is advantageous in scenarios where quickly executing smaller missions is beneficial allowing the system to achieve certain goals faster.

```

1  crl [prioritizeShortestMission] : M2 M3 AlternativeM | PrimaryM M1 => if
    (getDuration(M2) <= getDuration(M3)) then AlternativeM M3 |
    PrimaryM M2 M1
2  else AlternativeM M2 | PrimaryM M3 M1
3  fi .

```

This rule compares the durations of M2 and M3. The shorter mission is classified as primary for earlier execution, while the longer one is set as alternative.

- **Mission Delay Minimization** The *minimizeMissionDelays* rule is used to reduce delays in mission execution and to minimize overall system delay, ensuring that missions are completed on time.

```

1  crl [minimizeMissionDelays] : M2 M3 AlternativeM | PrimaryM M1 => if (
    getDelay(M2) <= getDelay(M3)) then AlternativeM M3 | PrimaryM M2 M1
2  else AlternativeM M2 | PrimaryM M3 M1
3  fi .

```

This rule assesses the delays of M2 and M3. The mission with lesser delay is prioritized as primary, while the other is reclassified as alternative, it compares the delays and gives priority to those with the least delay.

VII.3.2.2 Resource allocation and Optimization: RESOURCE-BASED-EXEC module

Similar to the previous module, the RESOURCE-BASED-EXEC system module includes the functional module WORKFLOW-DATA to structure the execution of missions within complex WFs, with a particular emphasis on resource-sensitive missions. i.e. this module prioritizes and manages missions based on their resource requirements and constraints. It handles the allocation and utilization of global and local resources for missions, catering to the dynamic needs of the system. By incorporating rules that address both resource constraints and the optimization of resource usage, the rewriting-based rules handle and manage missions in environments where resource allocation is a

challenging factor. The core functionality of the module is based on its rules-based rewrite approach to resource allocation, prioritization, and management for missions. This achieved by:

- Prioritizing of missions based on their resource availability, resource type, importance, etc.
- Managing both the allocation of resources to local missions and the sharing of resources among global missions.
- Handling situations where resource allocation may deviate from the desired behavior or when operational conditions require a re-evaluation.
- Allowing the system to select and reconfigure its missions for efficient and continuous resource management.
- Adjusting resource allocations in response to changes in resource availability or mission requirements.

Similar to the previous module, this one also classifies missions as primary and alternative, it provides rules designed to build a functional chain, focusing on behaviors such as resource types, sharing, and limitation. These rules govern the system in the efficient management of resources, as follows:

- Resource Allocation Efficiency

The rule *handleUnavailableResources* manages the execution of missions based on the availability of required resources, i.e. it classifies missions to an alternative status if their required resources are unavailable, while missions with available resources are given a primary status. This action ensures that the system does not attempt to execute missions without the necessary resources.

```
1  cr1 [handleUnavailableResources] :
2    < M2id : Mission | resType : Res2, ... >
3    < M3id : Mission | resType : Res3, ... >
4    AlternativeM | PrimaryM M1 =>
5    if (isResourceUnavailable(Res2))
6    then AlternativeM M2id | PrimaryM M3id M1
7    else if (isResourceUnavailable(Res3))
8      then AlternativeM M3id | PrimaryM M2id M1
9      else PrimaryM M2id M1 | AlternativeM M3id
10   fi
11  fi .
```

- Resource Efficiency Maximization

The rule *resourceEfficiencyMaximization* prioritizes missions based on their resource utilization time (RAUT), it promotes the most efficient use of resources, especially in environments where

resources are scarce or must be conserved. Particularly, it favors missions with a lower or equal RAUT in the “PrimaryM”.

```

1  crl [resourceEfficiencyMaximization] :
2    < M1id : Mission | RAUT : RAUT1, missionState : waiting, ... >
3    < M2id : Mission | RAUT : RAUT2, missionState : waiting, ... >
4    AlternativeM | PrimaryM =>
5    if RAUT1 <= RAUT2
6    then PrimaryM M1id | AlternativeM M2id
7    else PrimaryM M2id | AlternativeM M1id
8    fi .

```

- Unlimited Resource Preference

The rule *unlimitedResourcePreference* ensures that missions with access to unlimited resources take priority over those that rely on limited resources, which is essential to maintaining a balanced distribution of resources between different missions.

```

1  crl [unlimitedResourcePreference] :
2    < M1id : Mission | resType : R1, ... >
3    < M2id : Mission | resType : R2, ... >
4    AlternativeM | PrimaryM =>
5    if (isLimited(R1) and not isLimited(R2))
6    then PrimaryM M2id | AlternativeM M1id
7    else if (not isLimited(R1) and isLimited(R2))
8    then PrimaryM M1id | AlternativeM M2id
9    else PrimaryM M1id M2id | AlternativeM
10   fi
11  fi .

```

The rule evaluates whether R1 and R2, the resources for M1id and M2id, are limited or unlimited. It prioritizes missions utilizing unlimited resources, like R2 in the 'PrimaryM' category (first case), to ensure continued execution of missions without resource scarcity.

- Global Resource First

The rule *globalResourceFirstStrategy* is designed to optimize the distribution and utilization of resources by prioritizing missions that use global resources over those dependent on local resources. It helps in achieving a more balanced resource distribution at the global level of the entire SoS.

```

1  crl [globalResourceFirstStrategy] :
2    < M1id : Mission | resType : R1, ... >
3    < M2id : Mission | resType : R2, ... >

```

```

5   AlternativeM | PrimaryM =>
6   if (isLocal(R1) and isGlobal(R2))
7   then PrimaryM M2id | AlternativeM M1id
8   else if (isGlobal(R1) and isLocal(R2))
9       then PrimaryM M1id | AlternativeM M2id
10      else PrimaryM M1id M2id | AlternativeM
11      fi
12  fi .

```

When assessing the type of resources (R1 and R2) used by M1id and M2id, it assess the type of resources (global or local) used in missions and prioritizes those using global resources.

- Renewable Resource Prioritization

The rule *renewableResourcePrioritization* ensures that missions using renewable resources receive higher priority, particularly, it supports the use of resources that can be renewable.

```

1   crl [renewableResourcePrioritization] :
2   < M1id : Mission | resType : R1, ... >
3   < M2id : Mission | resType : R2, ... >
4   AlternativeM | PrimaryM =>
5   if (isRenewable(R1) and not isRenewable(R2))
6   then PrimaryM M1id | AlternativeM M2id
7   else if (not isRenewable(R1) and isRenewable(R2))
8       then PrimaryM M2id | AlternativeM M1id
9       else PrimaryM M1id M2id | AlternativeM
10      fi
11  fi .

```

This rule evaluates the type of resources (R1 and R2) for M1id and M2id missions and prioritizes Missions that rely on renewable resources (R1 or R2).

VII.3.2.3 Avoiding unwanted behavior and conflict resolution

Since SoSs often encounter obstacles and challenges due to the complexity of their CSs' interactions (e.g. the unpredictability of environmental conditions, or human error), the application of the previous rules does not always guarantee compliance or the desired results. Therefore, this challenge can have a significant impact on the effectiveness of rule implementation in dynamic and critical emergency scenarios. One example could be a set of CSs within an SoS that is responsible for coordinating the rescue efforts of multiple agencies during a major earthquake. Due to a communication error by an operator, important information regarding the location of the most affected locations is not being received correctly. This error may cause a significant delay in the deployment of emergency teams and resources to areas in need. Environmental unpredictability is

another factor that can disrupt rule adherence in CSs. During a large-scale natural disaster, an SoS responsible for coordinating emergency medical responses faces a critical challenge. i.e. the unavailability of transportation resources due to damaged infrastructure unexpectedly prevents the distribution of medical supplies. Despite the urgency, the system struggles to reallocate resources effectively, leading to delays in delivering essential healthcare to affected areas.

Both modules TIME-BASED-EXEC and RESOURCE-BASED-EXEC modules address unwanted and unpredictable behaviors. These behaviors are handled also through a set of rules, whose conditions are predefined in the WORKFLOW-DATA functional module. This section explores the nature of these rules and their operational importance within the SoS' WF. These rules allow the system to avoid and respond to various issues, ensuring missions continue to align with the strategic missions of the SoS. The rules consider the following concerns:

- Deadline Exceedance

The rule *deadlineViolation* checks if a mission exceeds its deadline. i.e. if the current time (Clock) is greater than the mission's deadline (Deadline), and the mission is not already marked as failed, the rule identifies it as a mission that cannot be completed within their assigned time frame and changes its state to failed.

```

1  crl [deadlineViolation] :
2    < Mid : Mission | deadline : Deadline, localClock : Clock, missionState
   : State, ... > =>
3    < Mid : Mission | missionState : failed, ... >
4    if isDeadlineViolated(Clock, Deadline) and State /= failed .

```

- Delay violation

The rule *delayViolation* starts when the delay of a mission exceeds the expected duration. i.e. if the delay (Delay) is greater than the expected duration (ExpectedDuration), and the mission's state is not already failed, the mission state is updated to failed. It's vital for monitoring missions that are taking longer than planned.

```

1  crl [delayViolation] :
2    < Mid : Mission | delay : Delay, expectedDuration : ExpectedDuration,
   missionState : State, ... > =>
3    < Mid : Mission | missionState : failed, ... >
4    if isDelayViolated(Delay, ExpectedDuration) and State /= failed .

```

- Arrival Time Violation

In order to ensure that missions do not start too early, the rule *arrivalTimeViolation* concerns situations in which a mission begins earlier than expected, i.e. if the clock time is less than ExpectedArrivalTime and the mission status is not already failed, the mission status is updated to failed.


```

1  crl [arrivalTimeViolation] :
2    < Mid : Mission | arrivalTime : ExpectedArrivalTime, localClock : Clock,
      missionState : State, ... > =>
3    < Mid : Mission | missionState : failed, ... >
4    if isArrivalTimeViolated(Clock, ExpectedArrivalTime) and State /=
      failed .
    
```

- Quit Time Violation

The rule *quitTimeViolation* checks if a mission exceeds its quit time. If the current time (Clock) surpasses the mission's quit time (QuitTime), and the mission's state is not already failed, the mission state is updated to failed. The rule ensures missions do not overrun their allocated execution window, impacting other operations.

```

1  crl [quitTimeViolation] :
2    < Mid : Mission | quitTime : QuitTime, localClock : Clock, missionState
      : State, ... > =>
3    < Mid : Mission | missionState : failed, ... >
4    if isQuitTimeViolated(Clock, QuitTime) and State /= failed .
    
```

- Resource Misallocation

In a case where missions rely on various resources, the *ResourceMisallocation* rule expresses the cases where missions don't have the proper necessary resources. If a mission is executed with a resource that turns out to be improperly allocated or unavailable, this rule intervenes to update the state of the mission accordingly.

```

1  crl [ResourceMisallocation] : < M : Mission | missionState : executing,
      resType : Res, ... > < R : Resource | resType : Res, resSt :
      unavailable, ... > => < M : Mission | missionState : failed, ... > if
      isResourceImproperlyAllocated(M) .
    
```

VII.3.3 Management Strategies

The rules introduced in Section VII.3.2.1 and Section VII.3.2.2 (modules TIME-BASED-EXEC and RESOURCE-BASED-EXEC, respectively) express the desired behavior of CSs during mission execution, they prioritize missions to execute the optimal FC and align them with desired behavior and functional objectives. Moreover, the rules introduced in Section VII.3.2.3, express how the unwanted behavior can occur and how missions can be violated, delayed, failed or overruns, etc. To guide and govern desired behaviors while preventing the execution of unwanted behaviors, we propose a series of strategies designed to avoid states characterized by violations, conflicts, and undesired actions. These strategies function by avoiding the execution of rules that lead to unwanted behaviors, i.e. this approach necessitates the preemptive application of these strategies before the re-execution

of any operational rules, ensuring that only desired behaviors (rules) are encouraged and maintained. The strategies help at:

- Autonomous decision-making for desired/unwanted behavior: This involves dynamically determining the execution of missions and allocation of resources provided by CSs within the SoS.
- Execution of the optimal FC: Tailoring the execution path to achieve the SoS's final goal, considering factors such as urgency and resource efficiency. Each strategy operates independently to provide focused optimization.
- Time-Critical Mission strategies: Prioritize missions with urgent or time-sensitive requirements. They schedules missions according to their time-criticality, addressing scenarios where timing is a crucial factor.
- Resource-Critical Mission strategies: Focus on reducing waste and ensuring an adequate supply of resources. These strategies is vital in scenarios where resource conservation and efficient utilization are key.

VII.3.3.1 Auxiliary Strategies

In this subsection, we define a set of auxiliary strategies based on time and resource constraints in the module FUNC-CHAIN-STRAT, the latter includes both MISSION-BASED-EXEC and RESOURCE-BASED-EXEC to provide seven auxiliary strategies as follows:

- Order Based-Execution

The strategy *timeCriticalMissionStr* switches between concurrent and sequential mission execution. It first attempts concurrent execution, governing parallel missions, and then goes back to sequential ordering if concurrent execution is not feasible.

```
1 sd orderBasedExecStr := SequentialMissionOrdering or-else
   ConcurrentMissionExecution .
```

- Time Critical Mission

The strategy *timeCriticalMissionStr* focuses on time-sensitive missions. It first prioritizes missions based on their start times and then shifts focus to completion times, ensuring that the most urgent missions are addressed promptly.

```
1 sd timeCriticalMissionStr := PrioritizeEarliestStart or-else
   PrioritizeEarliestCompletion .
```

- Quick Completion

The strategy *quickCompletionStr* emphasizes quick mission completion. It aims to prioritize missions with shorter durations and minimize delays in mission execution.

```
1 sd quickCompletionStr := PrioritizeShortestMission or-else
   MinimizeMissionDelays .
```

- Violated Constraints

The strategy *violatedConstraintsStr* handles missions that risk violating time constraints. It dynamically applies rules to manage and rectify such situations, preventing potential disruptions in mission execution.

```
1 sd violatedConstraintsStr := DeadlineViolation | DelayViolation |
   ArrivalTimeViolation | QuitTimeViolation .
```

- Local Resource Optimization

The strategy *localResourceOptStr* optimizes local resource allocation by handling unavailable resources by preferring missions that can utilize unlimited resources.

```
1 sd localResourceOptStr := HandleUnavailableResources or-else
   ResourceEfficiencyMaximization or-else UnlimitedResourcePreference .
```

- Shared Resource Sustainability

The strategy *sharedResourceSusStr* manages shared resources by prioritizing renewable and then global resources, promoting sustainable and strategic resource utilization across the SoS.

```
1 sd sharedResourceSusStr := GlobalResourceFirstStrategy or-else
   RenewableResourcePrioritization .
```

- Unwanted Resource Behavior

The strategy *unwantedResourceBeh* addresses unwanted behaviors related to resource allocation and mission scheduling, correcting issues like unscheduled mission execution and resource misallocation to prevent inefficiencies.

```
1 sd unwantedResourceBeh := UnscheduledMissionExecution |
   ResourceMisallocation .
```

In these strategies, the operators (or-else, |, and ;) play pivotal roles in strategizing mission execution and resource allocation within a SoS:

- The or-else operator is crucial for providing fallback options, ensuring that if one strategy is not applicable, an alternative can be immediately employed. This mechanism is illustrated in the *timeCriticalMisStr* strategy, where the system initially tries to apply the *prioritizeEarliestStart* strategy. If this execution is not possible, the system then switches to *prioritizeEarliestCompletion* as a backup plan.

- The | operator introduces a layer of non-determinism, it allows the system to choose any applicable strategy without a fixed order, as seen in the violatedConstraintsStr strategy, where the system can select any one of the violation handling rules like deadlineViolation or delayViolation, etc.
- The ; operator, on the other hand, ensures a controlled, sequential execution of strategies.

Collectively, these operators ensure that rules are executed in a specific sequence to guarantee a structured and orderly approach. Therefore, by leveraging the flexibility of or-else, the non-determinism of |, and the controlled execution of ;, the system will align with the dynamic and unpredictable nature of SoS environments, ensuring efficient and effective system operations.

VII.3.3.2 Main Strategies

The module also defines two main strategies missionBasedFCStrategy and resourceBasedFCStrategy which incorporate these auxiliary strategies:

- Mission-Based FC Strategy

The first strategy *missionBasedFCStrat* where auxiliary strategies like orderBasedExecStr, timeCriticalMissionStrategy, and quickCompletionStr are executed in a specific sequence. This sequential execution guarantees a structured and orderly approach to handling missions. Collectively, these strategies enable the system to adaptively manage various temporal scenarios in an SoS, prioritizing mission execution utilization while avoiding conflicts and undesirable states. The latter is reached by leveraging the flexibility of not(violatedConstraintsStr) which aligns with the dynamic and the unpredictable nature violated constraints in an SoS environments.

```

1  strat missionBasedFCStrat .
2  sd missionBasedFCStrat := (match AlternativeM | G) ? idle : (
3  orderBasedExecStr; timeCriticalMissionStr;
   quickCompletionStr; not(violatedConstraintsStr);
   selectedFuncCh) .

```

- Resource-Based FC Strategy

Similarly, the strategy *resourceBasedFCStrat* executes the two auxiliaries ones, i.e. localResourceOptStr, sharedResourceSusStr. This sequential execution ensures a structured and strategic approach to handling resources. Together, these strategies enable the system to autonomously manage various cases in an SoS, i.e. optimizing resources allocation while avoiding conflicts and undesirable states by executing the strategy not(unwantedResourceBeh) which aligns with the dynamic and the unpredictable nature consumed resources in an SoS environments.

```

1  strat resourceBasedFC .

```

```

2 sd resourceBasedFCStrat := (match AlternativeM | G) ? idle : (
    localResourceOptStr; sharedResourceSusStr; not(unwantedResourceBeh);
    selectedFuncCh) .

```

Both strategies `missionBasedFCStrategy` and `resourceBasedFCStrategy` are recursive strategies that repeat these steps forever or until the final mission is reached. They are restrictive and avoid executing conflicts or any unwanted states by discarding all missions where violatedConstraints and/or `missionBasedFCStrategy` are possible with $\text{not}(\alpha) \equiv \alpha ? \text{fail} : \text{idle}$. i.e. Executing violatedConstraints and/or `missionBasedFCStrategy` still requires visiting the conflicts and unwanted states defined in the module `WORKFLOW-DATA` and specified in the two execution modules as rules, but this execution path is discarded as if the state were never visited.

Central to these strategies are the operational semantics defined in `WORKFLOW-DATA`, including various ops and ceq conditions that determine the states of missions and resources. For instance, conditions like `DelayViolated`, `DeadlineViolated`, `isWCETViolated`, and `isArrivalTimeViolated` identify risky states based on time and resource constraints. These, along with the `unscheduledMissionExecution` and `resourceMisallocation` rules in `TIME-BASED-EXEC` and `RESOURCE-BASED-EXEC`, help identify and manage undesirable states.

By synchronizing these strategies and rules, the module effectively prevents management conflicts and contradictory decisions. It ensures that each mission is executed with the necessary resources while maintaining the overall balance within the SoS. The strategies are designed to be distinct yet complementary, enabling the system to autonomously decide the best course of action based on urgency, resource efficiency, and other critical factors.

VII.4 Real-Time regulating mechanism using MAPE-K loop

The MAPE-K loop which is fundamental to the concept of autonomic computing, consists of five key components: Monitor, Analyze, Plan, Execute, and Knowledge. Each of these components plays a crucial role in enabling a system to self-manage by autonomously monitoring its own operations, making decisions based on real-time data, and executing necessary adjustments. Here's a detailed explanation of each phase [42] [16], see Figure. VII.3 .

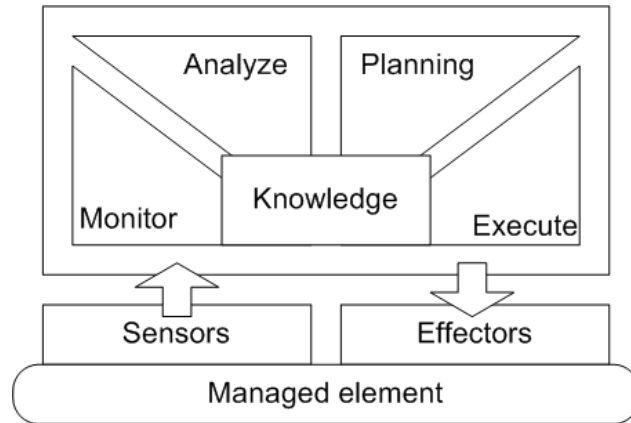


Figure VII.3: *The MAPE-K architecture.*

In this section, we specifically illustrate how the modules defined in this chapter and the previous one can be organized into an autonomic operational framework. Through the application of the MAPE-K loop phases, we demonstrate their integration and functionality within the context of the SoS. Moreover, the approach not only showcases the systematic organization of the specified modules but also introduces enhancements via new modules tailored with new modules designed to handle runtime regulating mechanisms and adjustments (see Figure VII.4).

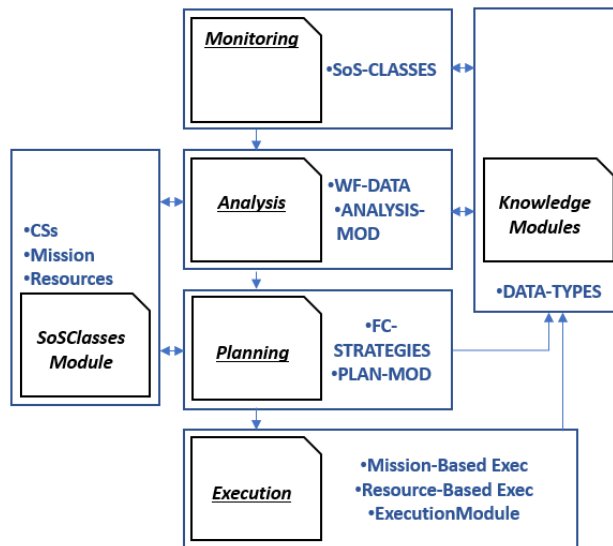


Figure VII.4: *Overall structure of the modules.*

These modules form the main components of our formal method. They work together to provide an executable approach for the specification, analysis, planning, and execution of the MAPE-K control loop within the SoS. In this context, we identify four types of temporal dependencies between various missions:

- Start-to-Start (S2S): This relationship signifies that the start of one Mission B is dependent on the start of another Mission A, i.e. Mission B cannot start until Mission A has begun.

- Start-to-Finish (S2F): This relation means that the completion of Mission B is dependent on the start of Mission A. It indicates that Mission B cannot finish until Mission A has started.
- Finish-to-Start (F2S): This relation implies that Mission B cannot start until Mission A has finished. i.e. the completion of one Mission triggers the beginning of another.
- Finish-to-Finish (F2F): In this relation, the finish of Mission B is dependent on the finish of Mission A. Mission B can only conclude once Mission A has been completed. Both Missions can run concurrently.

VII.4.1 Knowledge: Data Foundation

In the previous Chapter.VI, we have treated the CSs of a SoS as 'Black Boxes', the focus was primarily on the missions associated with each CS, rather than exploring the internal architecture, composition or specific details of the CSs themselves. In this Chapter, we enrich the approach with the temporal OO module DATA-TYPES which provides the operational semantic of the necessary data types such as CSs and SoSs. The CSIdSet data type represents a list of Oid's (Object Identifiers) that store the missions associated with each CS. It allows to effectively capture and organize mission-related data within the CS. The "none" operation creates an empty CSIdSet, while the ";" operation enables the concatenation of two CSIdSets, facilitating the management of multiple missions.

```
1  (tomod DATA-TYPES is
2  sort CSIdSet .
3  subsort Oid < CSIdSet .
4  op none : -> CSIdSet [ctor] .
5  op _;_ : CSIdSet CSIdSet -> CSIdSet [ctor assoc
6  ---
7  sorts Capacity ResourcePool . subsort Resource < ResourcePool
8  op resType:_resState:_quantity:_ : ResType State Time -> Resource [ctor] .
9  op emptyPool : -> ResourcePool [ctor] .
10 op __ : ResourcePool ResourcePool -> ResourcePool [ctor assoc id: emptyPool] .
11 sort Pool . subsort Pool < Configuration .
12 op [Pool: _ ] : ResourcePool -> ResPool [ctor] .
13 ...
```

VII.4.2 Monitor: Processing

For this step, we adopt the two modules SoS-CLASSES-MOD presented in Chapter.VI and DATA-TYPES for the initiation of the SoS configuration and providing valuable insights to running CSs, helping the designers make informed decisions based on real-time data by utilizing the equations provided by these two modules, the monitoring process evaluates the state of missions, their temporal constraints, and the associated resources of the initial running mission instances. This module in

dynamically initiating missions, resources, and CSs by instigating the OO-temporal classes as soon as events/changes are detected, thus ensuring that all components are promptly aligned with the latest operational requirements.

VII.4.3 Analysis: Workflow

This module analyzes the execution of missions within the SoS, considering resource availability and time constraints by employing the same operational semantic of WORKFLOW-DATA into the temporal OO module AnalyseModule, and enrich it with the necessary equation time information to assess the accessibility of required resources within the specified constraints. The module considers the current environmental context, including changes in mission or resource states, to determine if any runtime control is necessary for the subsequent Plan phase. The AnalyseModule relies on two sources of information. Firstly, it receives runtime states of ongoing missions and their consumed resources from the SoS-CLASSES-MOD, DATA-TYPES and WORKFLOW-DATA. This information help evaluate the progress and resource utilization of missions, and provide contextual data about the SoS initial configuration, enabling informed runtime decisions and accurate analyses.

```

1  (tomod AnalyseModule is
2  ops arrived accomplished pending failed : -> MissionState
3  eq calcState(DD, ED, DT) = if (DD > ED plus DT) then accomplished else pending fi
   . eq calcQuitTime(ED, DT) = ED plus DT .
4  eq calcQuitTime(ED, DT, ST) = ED plus DT plus ST .
5  eq calcArrivalTime(ST, D) = ST plus D .
6  eq calcRemainedResource(ST, D) = ST plus D .
7  eq calcLocalResource(ST, D) = ST plus D .
8

```

VII.4.4 Plan: Strategic Control and Management

In this phase, we define a new OO module PlanModule that leverages FUNC-CHAIN-STRAT module, the latter plans and selects missions to execute based on the analysis performed in the AnalyseModule and WORKFLOW-DATA modules. It interacts closely with the data types defined in the Knowledge Modules to retrieve relevant information, and update the knowledge accordingly. The primary objective of the PlanModule is to strategically plan and manage mission execution as seen in Chapter.VII. In this step, the system generates an execution strategy based on the FC-STRATEGY-MOD that selects the series of missions that would execute according to the SoS requirements and the current situation, considering factors such as WCET, delays, and precedence. The PlanModule proposes that we leverage RT-messages and operations to implement the MAPE-K control.

```

1  (tomod PlanModule is
2  including DATA-TYPES .

```



```

3  protecting NAT-TIME-DOMAIN . --- needed for the value 0
4  msg findRtt : Oid -> Msg .
5  msg initSoS : Oid -> Msg .
6  msg startAt : Oid Time -> Msg .
7  msg finishAt : Oid Time -> Msg .
8  --- ASAP: AS soon as possible, NLT: No Later than,
9  msg findRtt : Oid -> Msg .
10 msg start2Start : Oid -> Msg .
11 msg start2Finish : Oid Time -> Msg .
12 msg finish2Start : Oid Time -> Msg .
13 msg finish2Finish : Oid Time -> Msg .
14 msg ASAP : Oid Time -> Msg .
15 msg NLT : Oid Time -> Msg .
16

```

VII.4.5 Execution: Rewriting Management System

Upon receiving strategies which plans the execution of the functional chain with corresponding mission information from the planning module, the executor in the EXECUTEModule attempts to resolve temporal issues using corrective actions such as start-2-start, start-2-finish, etc. It self-regulates the selected rewriting rules behavior that address these temporal constraints to perform the corrective actions in response to changing conditions.

Table VII.3: Temporal control mechanisms for SoS Missions.

Control Mechanism	Conditions	Corrective Action
Early Arrival	Mission M1 arrives earlier than expected at time T ($T < T'$)	Start-to-Start (S2S) adjustment for M2, M2 starts at time T
Early Arrival	Mission M1 arrives earlier than expected at time T ($T < T'$)	Start-to-Finish (S2F) adjustment for M2, M2 finishes at time T
Late Completion	Mission M1 takes longer to complete than initially anticipated	Finish-to-Start (F2S) adjustment for M2, M2 starts at time T
Late Completion	Mission M1 takes longer to complete than initially anticipated	Finish-to-Finish (F2F) adjustment for M2, M2 finishes at time T
No Control Needed	No deviations or issues	No corrective action required

Table.VII.3. explores the temporal control mechanisms in RT-Maude, highlighting their relevance in addressing early arrival and late completion.

In the case of Early Arrival:

- Start-to-Start (S2S): This mechanism is utilized in scenarios where there is an early arrival of a mission, specifically when Mission M1 arrives earlier than its scheduled time. Under S2S control, if M1 arrives earlier than expected, the start time of Mission M2 is adjusted to coincide with the new start time of M1. This synchronization ensures that both missions commence simultaneously, aligning their execution times. For example, consider Missions M1 and M2, where M1 is scheduled to start at time T' , but it arrives and is ready to start at an earlier time T (where $T < T'$). In response to this early arrival, the system applies the S2S mechanism. Consequently, the start time of Mission M2, which might have been scheduled to commence after T' , is now revised to start at time T , aligning with the new start time of M1.
- Start-to-Finish (S2F): This mechanism is utilized when there is an early arrival of Mission M1. If M1 arrives sooner than expected (at a time T which is earlier than the scheduled time T'), the finishing time of Mission M2 is adjusted to align with the starting time of M1. This means that M2 will complete its tasks right as M1 begins, ensuring a seamless transition or overlap in mission execution. For example, if M1 was scheduled to start at time T' but arrives at time T (where $T < T'$), and M2 was initially scheduled to finish after T' , M2's finish time is now revised to T . This adjustment ensures that M2's operations are wrapped up by the time M1 commences.

In the case of Late Completion:

- Finish-to-Start (F2S): This adjustment is employed when Mission M1 finishes later than planned. If M1's completion time extends beyond its scheduled finish time, the start time of Mission M2 is revised to begin only after M1 has finished. This ensures that M2 does not start until M1 has fully completed its objectives. In this scenario, the delay in M1's completion directly influences the commencement of M2. For instance, if M1 was supposed to finish at time T but extends to time T' (where $T' > T$), M2, which was initially planned to start post-time T , now starts at time T' .
- Finish-to-Finish (F2F): This mechanism comes into play similarly to F2S, where Mission M1 finishes later than anticipated. However, in this case, instead of changing the start time of M2, the finish time of M2 is adjusted to match the delayed finish time of M1. For example, if M1 was expected to finish at time T but overruns to T' ($T' > T$), and M2 was initially set to finish at or around time T , the finish time of M2 is now altered to T' . This synchronization ensures that both M1 and M2 conclude their respective missions simultaneously, maintaining alignment in their operational timelines.

The table's entries illustrate the corrective actions for each control mechanism, specifying the adjustments made to the starting and finishing times of the respective missions. These control

mechanisms are related to the RT-Maude rules. i.g. the Start-to-Start control mechanism can be represented as:

cr1[Start-to-Start]: M1 starts at time T => M2 starts at time T in time u if (T < T')".

This rule signifies that if Mission M1 starts at time T and meets the condition of an early arrival (T < T'), then Mission M2 should also start at time T. The ExecutionModule is responsible for executing the control and orchestration strategies defined in the runtime Plan, which is generated by the PlanModule. During the Execute phase of the MAPE-K control loop, the ExecutionModule applies the tick rewriting rules to execute the Plan in the real-world.

Bellow is an example of Start-to-Start (S2S) rule:

```

1  cr1 [start2start] : start2start(M1, M2, ST)
2  < 0 : CS | missionSet : OS, clock : R >
3  < RES : Resource | resClock : R, resDuration : RD >
4  < M1 : Mission | loClock : D, duration : ED, arrivalTime : AT, quitTime : QT,
   delay : DT, deadline : DD, missionState : arrived, raut : RA >
5  < M2 : Mission | loClock : D', duration : ED', arrivalTime : AT', quitTime : QT',
   delay : DT', deadline : DD', missionState : idle, raut : RA' >
6  =>
7  < RES : Resource | resClock : R, resDuration : RD >
8  < M1 : Mission | loClock : ST, duration : ED, arrivalTime : ST, quitTime :
   calcQuitTime(ED, DT, ST), delay : DT, deadline : DD, missionState : executing
   , raut : RA >
9  < M2 : Mission | loClock : ST, duration : ED', arrivalTime : ST, quitTime :
   calcQuitTime(ED', DT', ST), delay : DT', deadline : DD', missionState :
   executing, raut : RA' > .

```

In this rule, both M1 and M2 start at the same time (ST) if M1 arrives early.

After receiving the generated plans and strategies from the Plan, the ExecutionModule uses the analysisMonitor to select each mission based on specific states and data. These criteria may include temporal constraints, resource availability, and the impact of MAPE-K loop on other running missions within the SoS environment. To handle the passage of time, we define the tick rule. This rule advances the system's configuration by a fixed time interval (1 in this case), as long as the elapsed time is within the system's maximum time limit (mte). The delta operation models the effect of time elapse on a mission's configuration. It updates the loClock attribute of the mission with the elapsed time. The mte operation determines the maximum time limit of the system based on the individual missions and timers. Additionally, the Monitor converts the Plan into a corresponding sequence of rewriting terms. This conversion facilitates the execution of the rules of both TIME-BASED-EXEC and RESOURCE-BASED-EXEC in the SoS environment. The rewriting rules define the actions to be performed based on the state of its CSs.

```

1  cr1 [tick] : {C:Configuration} => {delta(C:Configuration, R)} in time R if R <=

```

```

1   mte(C:Configuration) [nonexec] .
2   op delta : Configuration Time -> Configuration [frozen (1)] .
3   eq delta(none, R) = none .
4   eq delta(NeC:NEConfiguration NeC':NEConfiguration, R) =
5   delta(NeC:NEConfiguration, R) delta(NeC':NEConfiguration, R).
6   op mte : Configuration -> TimeInf [frozen (1)] .
7   eq mte(none) = INF .
8   eq mte(NeC:NEConfiguration NeC':NEConfiguration) =
9

```

VII.5 Conclusion

In this chapter, we have described how the SoS can autonomously decide on the execution path of workflows and the allocation of resources, considering both temporal and resource constraints. This ensures that missions are executed in alignment with strategic goals, even in the face of unpredictability and changing operational conditions. To this end, we have presented a comprehensive approach for formalizing management strategies within SoSs, we have emphasized the management of emergent behaviors, mission execution, resource management, etc. Formally, we have used the Maude Strategy Language to define operational semantics that enable the strategic management of SoS workflows. This is done by exploring the functional chains of SoSs, where both desired and undesired behaviors emerge due to the dynamic interactions between CSs. On the other hand, we have adopted the MAPE-K loop to introduce a real-time regulating mechanism that continually monitors, analyzes, plans, and executes management strategies, maintaining the SoS's adaptability and resilience. The MAPE-K components are integrated into the operational framework, facilitating the self-management of SoSs by autonomously responding to real-time constraints and executing necessary adjustments.

Chapter VIII

Simulation and Formal Verification

Contents

VIII.1 Introduction	154
VIII.2 Case study	155
VIII.3 Managing FESoS through a MAPE-K loop	156
VIII.3.1 Knowledge: Foundation	158
VIII.3.2 Monitor: Processing	158
VIII.3.3 Analysis: Workflow Analysis	159
VIII.3.4 Plan: Strategic Control and Management	159
VIII.3.5 Execution: Rewriting Management System	160
VIII.4 Design time: workflow and initial configuration	160
VIII.5 Simulation and execution	163
VIII.5.1 Resource Allocation Control	165
VIII.5.2 Managing FESoS Workflow with strategies	167
VIII.5.3 Executing Functional Chains	169
VIII.6 Formal verification	171
VIII.6.1 Maude-based verification for management strategies	172
VIII.6.2 Model-checking SoSs proprieties	174
VIII.7 Conclusion	176

VIII.1 Introduction

In this chapter, we cover the practical application of the proposed approaches through runtime simulation, analysis, and verification. We show the execution results through the implementation of

autonomic control and management within the French Emergency System of Systems (FESoS). We start by examining the case study that demonstrates the application of a MAPE-K loop—consisting of Monitoring, Analysis, Planning, Execution, and Knowledge phases—for managing emergency scenarios effectively. i.e. we explore the design time, where workflows and initial configurations are mapped out to support the system’s design phase. This is crucial for establishing an initial state from which the system’s performance can be evaluated. Next, we introduce various simulation scenarios and analyses to simulate the system’s centralized control behaviors and management strategies, ensuring they align with operational requirements, and make sure that the executed behaviors and actions are well-defined, promote the desirable behavior, and avoid the unwanted one at runtime stage. Finally, we use the formal verification techniques provided by Maude’s model-checking capabilities to verify that the system meets its strategic objectives and operates as expected under various emergency scenarios, this involves checking invariants and using the search command in Maude to explore possible system behaviors and ensure compliance with defined properties.

VIII.2 Case study

The case study explores a French Emergency System of Systems (FESoS) aims at protecting people and property. The FESoS consists of several interconnected CSs, and Missions. The MonitoringSoS, SAMU¹, Hospital CS, Civil Security, SDIS35², Search and Rescue Teams (SRT), and Fire and Rescue Services (FRS) are all part of the FESoS. In a possible situation, a major fire breaks out in a densely populated area, posing a significant threat to the safety of people and property. The FESoS is activated to respond to this emergency. The MonitoringSoS deploys Unmanned Aerial Vehicles (UAVs) for Aerial Surveillance to gather RT-information about the fire’s spread, intensity, and potential hazards. The Wireless Sensor Net CS (WSNCS) conducts Environmental Monitoring using a network of sensors to assess air quality, temperature, and other environmental factors. The CODIS³ controller oversees operations, analyzes data, coordinates resources, and prioritizes the protection of people and property. The SAMU is tasked with two missions: Patient Evacuation and Patient Transportation. They rapidly evacuate injured individuals from the affected area and provide immediate medical attention while transporting them to appropriate medical facilities. The HospitalCS activates its Emergency Reception and Triage to receive and assess the incoming patients, categorizing them based on the severity of their injuries. The Medical Treatment provides necessary medical interventions, and Continuous Patient Monitoring ensures ongoing observation and care. Civil Security takes charge of Emergency Response Coordination, managing communication and coordination between all involved entities. They ensure smooth information flow, resource sharing, and cooperation among different units and organizations. SDIS35 and SDIS56, the fire-fighting and emergency response organizations, implement their respective missions to contain and extinguish the fire. They deploy fire trucks, equipment, and skilled firefighters to battle the flames

¹Service d’Aide Médicale Urgente

²Service Départemental d’Incendie et de Secours

³Centre Opérationnel Départemental d’Incendie et de Secours

and mitigate risks. SRT conduct Rapid Assessment and Search Operations to locate and rescue individuals trapped or stranded due to the fire. They prioritize extraction and evacuation of survivors and provide initial medical triage and treatment on-site. FRS engage in Fire Suppression and Control, ensuring the fire does not spread further. They also handle Hazardous Material Handling and Containment and Structural Assessment and Collapse Rescue if needed.

The effective coordination of response efforts within the FESoS is a significant challenge that necessitates the presence of centralized controller and strategic workflow management. The RAC in Chapter.VI is vital for dynamically allocating resources like personnel and equipment, using real-time data to adaptively respond to evolving situations, this controller must possess a centralized management purpose and exercise ownership over all resources. Chapter.VII's emphasis on the strategic functional chain further enhances this coordination by ensuring that missions such as civilian evacuation, fire suppression, and medical response are prioritized and sequenced efficiently, maximizing their collective impact. The integration of resource allocation control with strategic mission sequencing allows FESoS to address the complexities of coordinating multiple CSs and units.

This comprehensive approach is key to successfully managing the multifaceted demands of emergency situations, ensuring that FESoS operates as a cohesive, efficient, and effective unit in protecting lives and property during crises. In this context, we propose the implementation of a MAPE-K controller, which assumes the responsibility of controlling resources and prioritizing missions within the FESoS. Functioning as the centralized authority. Furthermore, in the event of a major fire outbreak, where time assumes critical importance in mitigating risks to both individuals and assets, the RT-management assumes a vital role in overseeing the temporal aspects of missions. Through the coordination and execution of missions within designated timeframes, the it ensures effective emergency response, minimizes potential delays, and maximizes the utilization of pooled resources.

VIII.3 Managing FESoS through a MAPE-K loop

In this section, we show how the FESoS architecture can be defined by a collection of MAPE-K control loops, specified using the modules specified previously in the preceding chapters. This method employs a MAPE-K feedback loop, combining the Resource Allocation Controller (RAC)/Management Strategies, along with Maude's formal analysis and verification capabilities. The MAPE-K loop follows a systematic iteration between the design phase and the runtime execution. This iterative process focuses primarily on the effective management of temporal aspects and the optimization of the FESoS behavior.

At the design stage, two pathways are offered:

- Concrete instantiating of the UML profile from Chapter.V, tailored to the specificities being addressed. This involves detailing the FESoS architecture, interactions, and constraints.
- Direct application of the Meta-Models introduced in previous chapters.IV, which form the conceptual foundation and support the FESoS' key traits. The concrete model then becomes

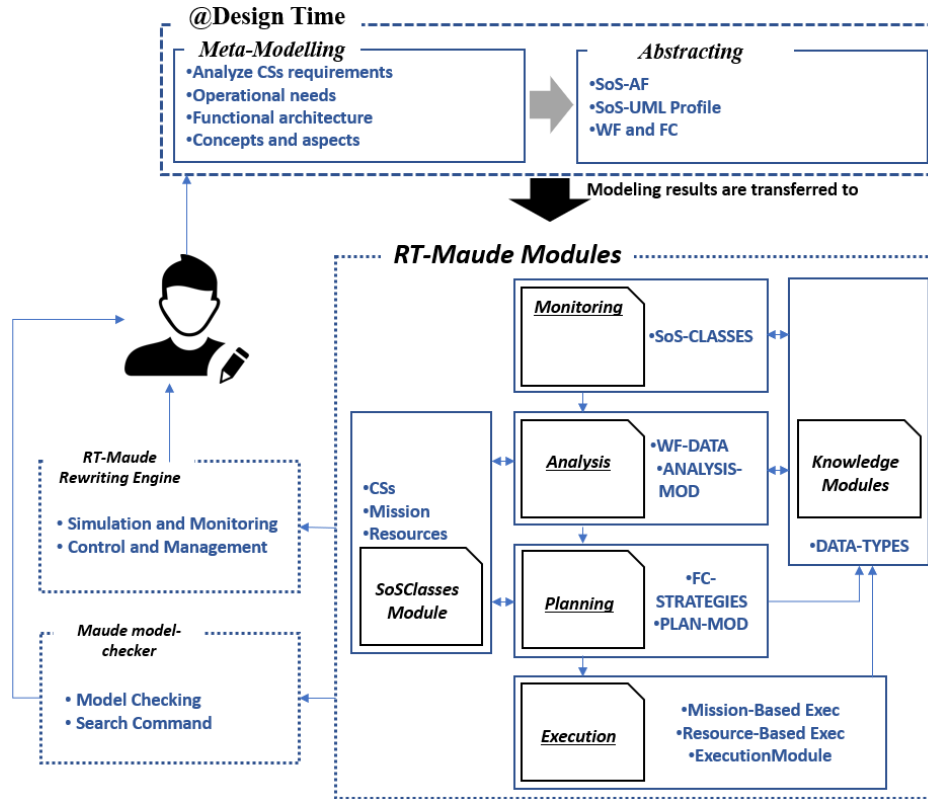


Figure VIII.1: The MAPE-K architecture for SoS Control and Management.

an instance of the system in practice. Designers can employ any capable modeling tool to implement these meta-models e.g. Capella/Arcadia, SysML, or BPMN, ensuring the tool can adequately encompass the depth of the meta-models and their conceptual framework.

The FESoS runtime phase is captured and handled using the different modules of Maude, the latter are managed using the phases of the MAPE-K loop in order to define a loop for each regulating concern. All together, they also provide a rich set of formal analysis methods and tools for reasoning about the specifications of the adopted Meta-Models. These methods include model checking, theorem proving, and reachability analysis. By applying these techniques, we can verify additional properties of the FESoS, detect potential issues, voids undesired behaviors and selects optimal mission paths, and ensure the correctness and effectiveness of our model representations.

Figure.VIII.1 summarizes the main components of our formal modules for the specification and execution of the FESoS. These modules work together to provide an executable approach for the specification, analysis, planning, and execution of the MAPE-K control loop within the FESoS. They enable the modeling of initial configurations, monitoring of changes, analysis of resource availability and mission execution times, planning of operational functions, and the execution of control actions.

On the other hand, the MAPE-K loop execution structure typically involves:

- Concurrent Mission Management: This includes the ability to initiate, run, and monitor multiple missions of FESoS' different CSs at the same time.

- Synchronization Mechanisms: These are crucial for ensuring missions that are dependent on each other's outputs are coordinated effectively. This could involve S2S, S2F, F2S and F2F, as previously mentioned.
- Real-Time Monitoring and Analysis: Continuous monitoring of all parallel missions is essential for ensuring everything is running as expected and for making real-time adjustments as needed.
- Knowledge-Based Decision-Making: The Knowledge component of MAPE-K is critical in parallel execution, as it provides the necessary information and insights for effective decision-making regarding task coordination, resource management, and adaptation strategies.

VIII.3.1 Knowledge: Foundation

In the context of FESoS, the Knowledge phase employs the temporal OO module DATA-TYPES, which is pivotal for modeling the essential data types that represent the system's operational state, capabilities, and ongoing activities. This includes the CSIdSet data type, which is a list of Object Identifiers (Oid) for storing missions associated with each CS. This data type keeps track of mission assignments and statuses across the different CSs within FESoS.

Furthermore, the Knowledge phase introduces the Capacity and ResourcePool data types, representing the available resources within FESoS. These data types detail the resources' types, states, quantities, and other attributes, allowing for the aggregation and management of resources critical for adaptive response to the crisis. In the emergency scenario of a major fire outbreak, this phase serves several essential functions, i.e it supports the dynamic allocation and management of resources, such as personnel, UAVs for aerial surveillance, and firefighting equipment, ensuring efficient deployment where most needed. By utilizing the CSIdSet, the system can track all ongoing missions and their associated CSs, and this is what will allow prioritizing missions, coordinate efforts between CSs like SAMU, Hospital CS, and SDIS35, and adjust strategies as the situation evolves.

The temporal attributes captured within the Knowledge base, including mission duration, start and end times, and delays, are critical for executing missions within designated timeframes, especially for time-sensitive operations like patient evacuation and emergency triage at hospitals. This phase also consolidates data on the fire's intensity, spread, and environmental conditions from CSs like MonitoringSoS and WSNCS, enabling FESoS to adapt its strategies in real-time, including re-prioritizing missions, reallocating resources, and deploying additional support where necessary.

VIII.3.2 Monitor: Processing

This phase continuously observes and detects changes within the FESoS environment, a major fire outbreak, for instance. It assesses the intensity and spread of the fire, the availability and state of resources, and the progress of ongoing missions. This phase of the MAPE-K loop leverages operational semantics defined in the SoS-CLASSES-MOD and DATA-TYPES modules. More specifically, it utilizes the equations provided by these two modules to evaluate, assess, and initiate the state of missions, their temporal constraints, and associated resources for initial configuration.

As changes are detected, whether in the form of evolving fire dynamics, resource availability, or mission requirements, the Monitoring phase promptly triggers adjustments across the FESoS components and CSs. This ensures that the UAVs for aerial surveillance are deployed efficiently, the Environmental Monitoring by WSNCS is accurate and timely, and the coordination among CSs like SAMU, Hospital CS, and SDIS35 is seamless and effective. This phase is also responsible for the initiation of FESoS and the provision of its initial configuration, setting the stage for a response strategy that aligns with the current situation and requirements.

VIII.3.3 Analysis: Workflow Analysis

This phase leverages the operational semantics introduced by the WORKFLOW-DATA module, integrating them within the temporal OO module and AnalysesModule. This enhancement with necessary equations and time information allows for a detailed assessment of resource accessibility against the backdrop of specified constraints. At the same time, it relies on two sources of information : the runtime states of ongoing missions, including resource consumption details sourced from SoS-CLASSES-MOD, DATA-TYPES, and WORKFLOW-DATA modules. AnalysesModule utilizes states like arrived, accomplished, pending, and failed to specify mission states, coupled with equations that compute mission state based on deadlines, execution duration, and delays. These computational tools (calcState, calcQuitTime, calcArrivalTime, calcRemainedResource, and calcLocalResource) are vital for navigating the temporal and resource-related complexities of mission execution within FESoS. They ensure that every mission is analyzed within its environmental context, including any shifts in mission requirements, to reply to the needs of runtime adjustments ahead of the Plan phase.

VIII.3.4 Plan: Strategic Control and Management

By leveraging the PlanModule, which incorporates strategies from the FUNC-CHAIN-STRAT module, FESoS is equipped to strategically select and prioritize missions based on the detailed analysis provided by the AnalyseModule and insights from the WORKFLOW-DATA modules.

The process begins as the PlanModule assesses the current situation, utilizing the dynamic information retrieved from the Knowledge phase to understand the evolving emergency. The execution strategy generated by the PlanModule, guided by the FC-STRATEGY-MOD, meticulously sequences the series of missions to be executed. This sequencing takes into account the SoS requirements and the unique demands of the current situation, factoring in the Worst Case Execution Time (WCET), potential delays, and mission precedence. For instance, it may prioritize UAV deployment for aerial surveillance to provide real-time data on the fire's spread, coordinate the evacuation of injured individuals by SAMU, and ensure the efficient distribution of medical and firefighting resources. Moreover, using RT-messages and operations, the phase dynamically adjusts mission parameters and resource allocations in real-time. This includes initiating missions like Environmental Monitoring by WSNCS for accurate and timely assessment of air quality and other environmental factors affecting the emergency response. It also ensures seamless coordination among CSs like SAMU, Hospital CS,

and SDIS35, adapting their roles and responsibilities as the situation evolves.

VIII.3.5 Execution: Rewriting Management System

Upon receiving execution strategies from the PlanModule, which are devised based on the base of FUNC-CHAIN-STRAT, AnalyseModule, and WORKFLOW-DATA, the EXECUTEModule transforms the strategic plans into real-world actions. Therefore, it addresses temporal issues and orchestrates the execution of the functional chain, applying corrective actions such as start-2-start and start-2-finish among others, to ensure temporal alignment of missions according to the evolving conditions of the emergency. For instance, in scenarios where a mission arrives earlier than anticipated, the EXECUTEModule utilizes the start-2-start mechanism to adjust subsequent missions, ensuring they commence with the early-starting mission. This coordination is crucial for optimizing the deployment and effectiveness of resources and missions, such as the swift deployment of UAVs for aerial surveillance or the timely initiation of SAMU's patient evacuation missions. Similarly, the module employs start-2-finish and finish-2-finish rules to ensure seamless transition and completion of missions, aligning their execution with the real-time emergency situation demands. This involves adjusting the start or finish times of missions in response to early arrivals or delays, ensuring that all actions are synchronized to address the emergency efficiently. The EXECUTEModule's strategic application of rewriting rules, informed by RT-Maude's temporal control mechanisms, demonstrates FESoS's capability to adapt dynamically to the challenges presented by the emergency. By leveraging the detailed planning and analysis phases.

VIII.4 Design time: workflow and initial configuration

The proposed approach follows a systematic iteration between the design phase and the runtime execution (Fig. VIII.1), emphasizing the collaboration with stakeholders to incorporate their needs and feedback into the design process, and leveraging the power of the MAPE-K control loop and the formal analysis and verification methods of Maude. This iterative process ensures the effective management of temporal aspects and the optimization of desired behavior in SoSs environments. During the design phase, the focus is on employing the Meta-Model and instantiating it into a concrete model that represents the specific SoS under consideration. This phase involves defining the structure, relationships, and constraints of the SoS, etc. representing the concrete model or the initial system configuration.

To show the reusability of the proposed Meta-Models, we adopt other Methodology/tool ARCADIA/Capella to create a visual representation of the initial concrete model of FESoS, showcasing interactions and dependencies among the CSs, Missions, and the MAPE-K Controller. The design phase, provide a comprehensive approach to analyze CSs requirements, operational needs, and stakeholder expectations. The concrete model establishes a Functional Architecture that captures the structure, behavior, and interactions of the FESoS' CSs. We ensure that the logical architecture design aligns with emergency response systems' standards, best practices, and regulatory require-

ments, considering the capabilities and limitations of missions, their resources, and the coordination mechanisms required for effective collaboration.

To design the different entities and units in FESoS, we employ the Logical Architecture model as a graphical model to present collections of interlinked missions that produce specific missions or capabilities. Moreover, this model focuses on the core features of the FESoS, particularly the constructs of WFs and their functional chains. The model provides graphical representations for CSs, missions, exchanges, and ports, all of which are interconnected through functional chains. Each mission in the model represents an atomic function, with incoming and outgoing flows denoting its dependencies and outputs. Events in the system are represented by function exchanging among different missions, which also capture the execution order between two missions, enforcing temporal constraints and sequencing within the model (Figure.VIII.2).

On the other hand, functional chains play a crucial role in the logical architecture model of the proposed FESoS. They provide a structured and systematic process to represent the sequence of missions and interactions necessary to accomplish the global of FESoS. They enhance the clarity and the decomposition ensuring that the system adequately supports the desired functionalities. In addition, functional chains with parallel paths allow for the representation of multiple alternative paths or simultaneous missions within the FESoS. For example, during an emergency response, parallel paths can be used to illustrate different response strategies or the execution of concurrent tasks. This flexibility enables the system to adapt to dynamic situations and optimize resource utilization. Furthermore, they provide parallel paths helping to identify the interdependencies between CSs, facilitating a better understanding of how different CSs rely on each other. By specifying the sequence and timing of operations, functional chains enable the coordination and synchronization of actions, ensuring that the FESoS functions as a cohesive system. The inclusion of parallel paths in functional chains also supports simultaneous execution of critical missions within. This enables efficient coordination and response during emergency situations, as multiple missions can be performed concurrently. In the event of disruptions or failures in one path, alternative paths can be activated to maintain the continuity of critical operations. This redundancy enhances the system's ability to withstand unexpected events and minimize disruptions in emergency response efforts.

To further strengthen functional chains within the Logical Architecture model of the FESoS, we propose the inclusion of control nodes (OR, AND, IT) which can represent the different sequences of missions and temporal dependency links establishing the order of execution between the functions, along with the addition of parameters to nodes and links. By introducing parameters to the nodes and links, the functional chain description becomes more detailed and quantitative, capturing important information such as durations, starting times, deadlines, WCET, resource types, and time usage amounts. The parameters represent the quantitative information associated with each mission, enabling a precise analysis of resource utilization, time constraints, and dependencies. For example, a parameter can represent the duration of a specific mission, indicating how long it takes to complete. Another parameter may denote the starting time or deadline for a mission, defining when it should begin or be completed. Resource parameters identify the types of resources required by a mission, such as personnel, equipment, or supplies, while the associated time usage amount

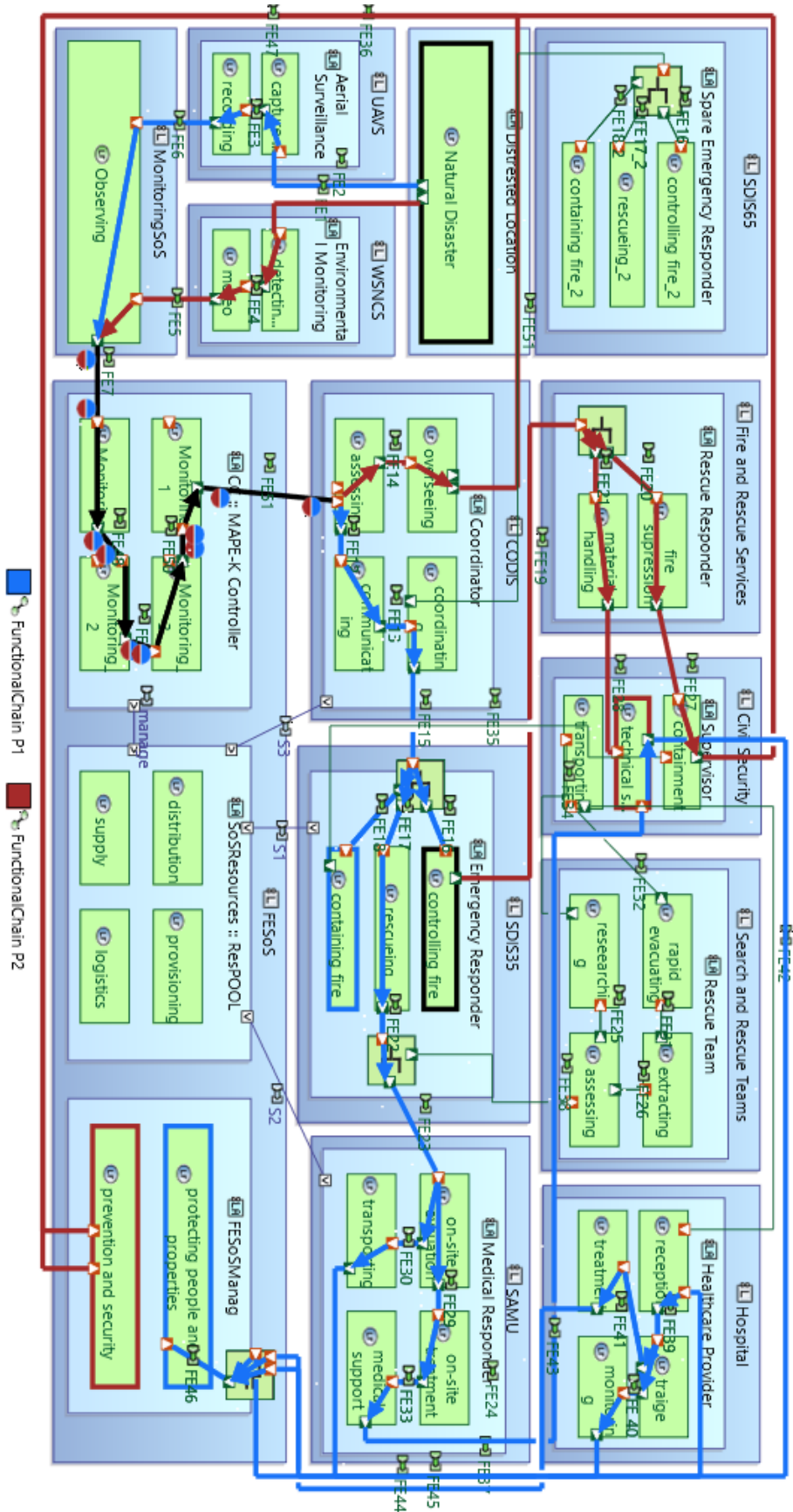


Figure VIII.2: Overview of the Logical Architecture model of FESoS.

parameter indicates the quantity or duration of resource utilization.

By incorporating these parameter values into the functional chains, the logical architecture model gains a higher level of precision and specificity. Designers can now explore and evaluate different design alternatives by varying the parameter values and adjusting the temporal dependency links. This allows for a thorough analysis of the system’s behavior and performance, considering factors such as resource allocation, time constraints, and dependencies. To overcome the space limitations of this paper, we have carefully selected a subset of key representative Figure.VIII.3 from the Functional Chain Description P1 with parameter values. Although not all parameters values can be included due to the page limit, the figure illustrates the overall concepts and functionality of the Functional Chain Description while effectively showcasing the impact of the parameter values on the CSs and their Roles, missions and quantitative attributes.

Despite the fact of that this design offers a thorough analysis that enables the different design alternatives through different alternative functional chains, there remains a gap in their full adoption during the runtime phase where the quantitative priorities of the SoS need to be explored analyzed and assessed, and this what will ensure the selected design option meets resource and time constraints. This design stand short of helping in a further step to identify issues and unwanted behavior from overlapping or conflicting mission plans and ensures coordinated and strategic execution. Moreover, even the integration of temporal interdependencies into the logical architecture model enhances the understanding of the SoSs’ quantitative behavior, would not allow for the design of an effective emergency response SoS that optimizes resource allocation, meets time constraints, and ensures the smooth execution of missions within the FESoS.

VIII.5 Simulation and execution

Once the design phase is completed, the runtime phase begins. The latter is guided by the MAPE-K control loop, specified using Maude and its extensions Maude Strategy and RT-Maude modules. At this stage, the Monitor observes the SoS’s state, while the Analyze component analyzes the data collected. Execution and Simulation in Maude system can be provided due to its high-performance rewrite engine. Maude provides a range of commands for system modules enabling exploration of the specified behavior from an initial configuration of the system, such as the rewrite, and frewrite, etc. For simplicity, the next section is dedicated to particularly simulating the design of the proposed behavior of the RAC and Strategic Manager using a MAPE-K loop.

In the context of the FESoS, we exhibit the operational capabilities of the RAC by applying a sequence of rewriting rules within the different modules introduced in previous chapters. The initial configuration state, initState, is referenced in Figure. VIII.4, encapsulates a suite of missions that have been identified during the Design stage. These missions, depicted as ‘on-site evaluation’, ‘medical support’, ‘technical support’, ‘reception’, and ‘triage’, are systematically aligned with their respective CSs such as ‘SAMU’, ‘SRT’, and ‘CivilSecurity’. Corresponding resources such as ‘Ambulance’, ‘MedicalEquipment’, ‘HealthcareFacilities’, and ‘TechnicalPersonnel’ are also incorporated. This ensemble is anchored by the FESoSResPool, representing the accessible resources within the

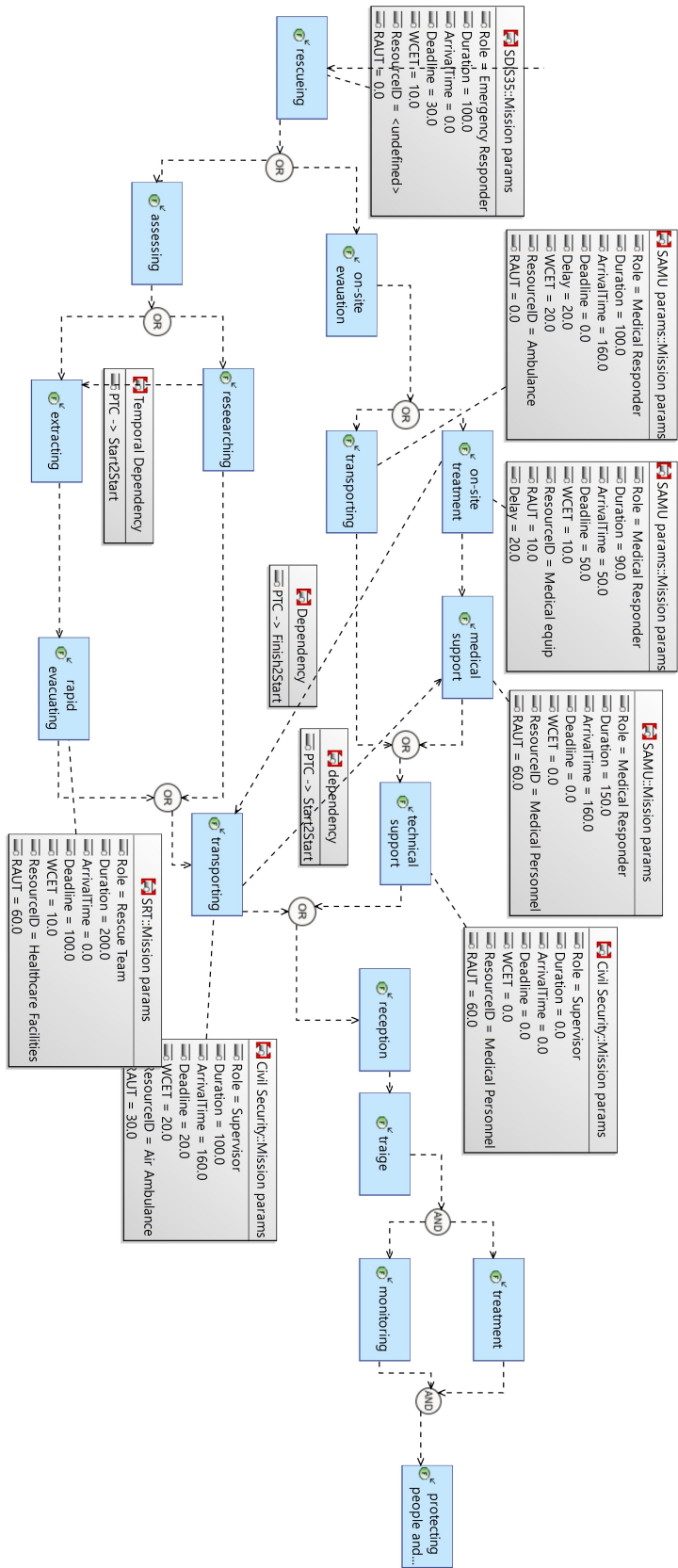


Figure VIII.3: Overview of the Logical Architecture model of FSSoS.

FESoS infrastructure. This defined initial state serves as the foundation from which the FESoS can dynamically orchestrate resource management and execute missions, thereby enabling the activation of the Monitoring phase within the MAPE-K feedback loop.

```

-STATES.maude  *SoS.maude x
op initState : -> GlobalSystem .
eq initState ={
  startAt(on-siteTreatment, 50)
  finish2start(on-siteTreatment, transporting, 50)
  start2start(transporting, medicalSupport, 0)
  < on-siteTreatment : Mission | loClock : 0, duration : 100, arrivalTime : 0 >
  < transporting : Mission | loClock : 0, duration : 100, arrivalTime : 0, q >
  < medicalSupport : Mission | loClock : 0, duration : 150, arrivalTime : 0, q >
  < SAMU : CS | clock : 0, missionSet : on-siteTreatment ; medicalSupport >
  < CivilSecurity : CS | clock : 0, missionSet : transporting >
  < Ambulance : Resource | resClock : 0, resDuration : 10 >
  < MedicalEquipments : Resource | resClock : 0, resDuration : 40 >
  < MedicalPersonnel : Resource | resClock : 0, resDuration : 30 >
  < FESoSResPool : ResPool | globalRes : 400 >
.
ndtom)
set tick def 1 .)
tfrew initState in time < 300 .)

```

Figure VIII.4: FESoS initial configuration.

VIII.5.1 Resource Allocation Control

In this section, we explore a series of scenarios to demonstrate the RAC applicability as a centralized controller addressing resource allocation. These simulations, ranging from UAV deployments for aerial surveillance to allocations of ambulances and fire engines, showcase the system's capacity to handle diverse and challenging emergency response situations effectively.

- Scenario 1: Deployment of UAVs for Aerial Surveillance:

In this scenario, the FESoS aims to deploy UAVs for real-time surveillance of a significant urban fire. The UAVs are global shareable resources, initially in an 'Available' state, ready for deployment. The Aerial_Surveillance mission awaits resource allocation, indicated by its 'waitConsResp' state. Upon executing the [allocate-global-shareable-resource] Maude rule, the UAVs transition to a 'InUse' state, actively participating in the mission. Consequently, the Aerial_Surveillance mission moves to an 'Executing' state, commencing comprehensive aerial monitoring.

```

1
2  crl [allocate-global-shareable-resource] :
3  < RAC : ResourceAllocationManager | mission : Aerial_Surveillance, resource
   : UAVs, racState : trigger >
4  < UAVs : Resource | resSt : Available, resP : globalResProp(shareable) >
5  < Aerial_Surveillance : Mission | state : waitConsResp >
6  =>

```



```

7 < RAC : ResourceAllocationManager | mission : Aerial_Surveillance, resource
  : UAVs, racState : consReqAccepted >
8 < UAVs : Resource | resSt : InUse>
9 < Aerial_Surveillance : Mission | state : executing >
10 if PRes(isAvailable(UAVs)) and PRes(isGlobal(UAVs)) and PRes(isShareable(
    UAVs)).

```

- Scenario 2: Allocating Ambulances for Medical Evacuation

The FESoS faces the urgent need to allocate ambulances for medical evacuation amid the urban fire. Ambulances, as local non-shareable resources, are in an 'available' state initially. The Medical_Evac mission is in a 'waitConsResp' state, signaling the need for resource allocation. The application of the [allocate-local-nonshareable-resource] Maude rule allows for the ambulance's state to transition to 'locked', indicating its commitment to the Medical_Evac mission. Simultaneously, the mission's state changes to 'executing', reflecting the initiation of medical evacuation operations.

```

1 crl [allocate-local-nonshareable-resource] :
2 < RAC : ResourceAllocationManager | mission : Medical_Evac, resource :
  Ambulance, racState : trigger >
3 < Medical_Evac : Mission | missionState : waitConsResp, resType : localRes >
4 < Ambulance : Resource | resState : available, resP : localResProp(
  limitedButRenewable), resType : localRes >
5 =>
6 < RAC : ResourceAllocationManager | mission : Medical_Evac, resource :
  Ambulance, racState : consReqAccepted >
7 < Medical_Evac : Mission | missionState : executing >
8 < Ambulance : Resource | resState : locked >
9 if PRes(isAvailable(Ambulance)) and PRes(isNonShareable(Ambulance)) and not
  PRes(isLocked(Ambulance)).

```

- Scenario 3: Locking a Limited Local Resource in FESoS for Firefighting

In this scenario, the FESoS faces the mission of locking a limited local resource, essential for combating an urban fire. This resource, for instance, a specialized fire engine, is initially in an 'available' state, indicating it is ready for deployment. The Fire_Suppression mission, reliant on this resource, is in a 'waitConsResp' state, waiting for resource allocation. The implementation of a Maude rule [lock-limited-local-resource] is required to transition the resource's state to 'locked', signifying its exclusive allocation to the Fire_Suppression mission. Concurrently, the mission's state changes to 'executing', indicating the commencement of active fire suppression efforts. This scenario highlights FESoS's ability to promptly allocate and lock vital local resources, ensuring their dedicated use in critical firefighting operations.

```

1  crl [lock-limited-local-resource] :
2  < RAC : ResourceAllocationManager | mission : Fire_Suppression, resource :
   FireEngine, racState : trigger >
3  < FireEngine : Resource | resState : available, resP : localResProp(limited)
   , resType : localRes >
4  < Fire_Suppression : Mission | missionState : waitConsResp >
5  =>
6  < RAC : ResourceAllocationManager | mission : Fire_Suppression, resource :
   FireEngine, racState : consReqAccepted >
7  < FireEngine : Resource | resState : locked >
8  < Fire_Suppression : Mission | missionState : executing >
9  if PRes(isAvailable(FireEngine)) and PRes(isLimited(FireEngine)) and not
   PRes(isLocked(FireEngine)).

```

- Scenario 4: Request for Locked FireEngine by Urban_Rescue Mission in FESoS

In the scenario where the Urban_Rescue mission requests the FireEngine, which is currently engaged in the Fire_Suppression mission, the system faces a resource contention issue. Upon the Urban_Rescue mission's request, the FESoS handles this by changing the mission's state to 'rnwAsk', indicating a request for renewed allocation of the FireEngine. The RAC now has the task of addressing this renewed request. It can choose to wait until the FireEngine completes its current task with the Fire_Suppression mission before reallocating it to Urban_Rescue, or it can seek an alternative resource to fulfill the Urban_Rescue mission's requirements.

```

1  crl [request-locked-resource] :
2  < RAC : ResourceAllocationManager | mission : Urban_Rescue, resource :
   FireEngine, racState : trigger >
3  < FireEngine : Resource | resState : locked >
4  < Urban_Rescue : Mission | missionState : waitConsResp >
5  =>
6  < RAC : ResourceAllocationManager | mission : Urban_Rescue, resource :
   FireEngine, racState : resourceWait >
7  < Urban_Rescue : Mission | missionState : rnwAsk >
8  if PRes(isLocked(FireEngine)) and PRes(currentMission(FireEngine) !=
   Urban_Rescue).

```

VIII.5.2 Managing FESoS Workflow with strategies

As depicted in Figure. VIII.2, two functional chains are presented, demonstrating their potential in achieving global mission objectives. These missions within the functional chains are designated as M1, M2, M3,...,Mn for simplicity and easy reference. They serve as key elements within these chains. The Maude Strategy Language is employed to execute these mission specifications, providing

a practical simulation and evaluation framework.

The strategy `missionBasedFCStrat` is central to this simulation. It is specifically designed to assess the viability of accomplishing a critical global mission, identified as M34, starting from the predefined initial state, i.e. all the missions in the configuration module. This strategic assessment utilizes a combination of rules and conditions from the `MISSION-BASED-EXE` module. These encompass strategies like `SequentialMissionOrdering` and `ConcurrentMissionExecution`, as well as prioritization tactics such as `PrioritizeEarliestStart` and `PrioritizeEarliestCompletion`.

The strategy's core functionality revolves around effectively categorizing missions into two distinct groups: 'PrimaryM' and 'AlternativeM' focusing on temporal factors, which distinguishes it from strategies that prioritize resource allocation. This temporal emphasis is crucial in scenarios where the timing of mission execution is critical. This is done by using the `srew` command, which allows for a systematic exploration of different state transitions under the strategy.

```
Maude> srew initial using missionBasedFCStrat.
```

```
Solution 1
```

```
rewrites: 124
```

```
result missionBasedFCStrat: AlternativeM M4 M5 M11 M13...M29 |
```

```
PrimaryM M0 M1 M2 M3 M6 M7 M8.....M30 M33 M34
```

```
No more solutions.
```

```
rewrites: 124
```

- PrimaryM Missions: These missions are selected to form the main functional chain, crucial for accomplishing the global mission M34. They are prioritized based on various factors like urgency, their start, their completion times and their critical role in achieving the global objective. The missions M0 through M30, M33, and M34, by being in 'PrimaryM', are given precedence in execution, signifying their importance in the successful completion of the global mission.
- AlternativeM Missions: Missions categorized under 'AlternativeM', such as M4, M5, M11, and M13 through M29, serve a supportive role. They are auxiliary to the primary missions, providing necessary support and resources to ensure the seamless execution of the primary functional chain.

The `srewrite` command in the Maude system, applied to this scenario, simulates the strategic allocation and execution of missions, executing 124 rewrite steps. The result of this simulation offers an insightful glimpse into the effective planning and management of missions, underscoring the strategic prowess of the `missionBasedFCStrat` in prioritizing complex operational missions.

VIII.5.3 Executing Functional Chains

Simulation and analysis in RT-Maude system can be provided due to its high-performance rewrite engine. Maude provides a range of commands for system modules enabling exploration of the specified behavior from an initial configuration of the system such as the `rewrite`, and `tfrewrite`, etc. In this section, we present the results of simulation experiments conducted to evaluate the performance of the MAPE-K loop within the FESoS and assess the impact of integrating such a centralized controller into the FESoS. These experiments were performed using the developed RT-Maude modules, which provides an executable specification of the FESoS. The objective of the evaluation is to gain insights into the effectiveness and efficiency of the controller in managing the behavior and to investigate the benefits of incorporating the centralized control architecture. By conducting these experiments, we aim to assess the system's performance under various scenarios and analyze the impact of different control strategies on mission execution, resource utilization, and overall system behavior.

This section discusses the evaluation of corrective actions and self-regulating Mechanisms in FESoS case study. The aim is to understand the runtime evolution and verify if the execution meets properties identified during the design stage of the FESoS. Therefore, A possible scenario showcases the execution of three interlinked missions in the FESoS, specifically focusing on the missions of `on-siteTreatment`, `transporting` and `medicalSupport` (see Figure. VIII.3). These missions are interdependent and involve sequential and parallel execution, with certain delays, dependencies and specific resources (e.g. `Ambulance`, `MedicalEquipments`, and `MedicalPersonnel`, respectively). The `on-siteTreatment` mission is designed to finish before the `transporting` and `medicalSupport` missions, with the latter two missions beginning simultaneously. During the design time phase, the designers carefully define the initial values for the parameters of each mission in the functional chain description, these parameters are represented by the red rectangles. For instance, mission `on-siteTreatment`, has the following parameter values: `duration`: 90, `arrivalTime`: 50, `delay`: 20, and `raut` (resource usage time): 10.

The designers' inputs, while based on the available information and assumptions, do not fully capture the dynamic nature of the FESoS and the potential runtime changes that can occur. e.g. even though the `on-siteTreatment` mission was accomplished on time, without any delays, the subsequent missions (`transporting` and `medicalSupport`) still have to wait for the predefined delay 20 units see Figure. VIII.5 and Figure. VIII.4 . This means that these two missions will essentially waste 20 units of time and start executing at (160) which can ultimately impact their ability to be completed within the expected time frame (300 time unit for instance). This situation highlights a limitation in the design phase, where the initial expectations and assumptions do not fully account for the dynamic nature of the system and the potential runtime changes that may occur.

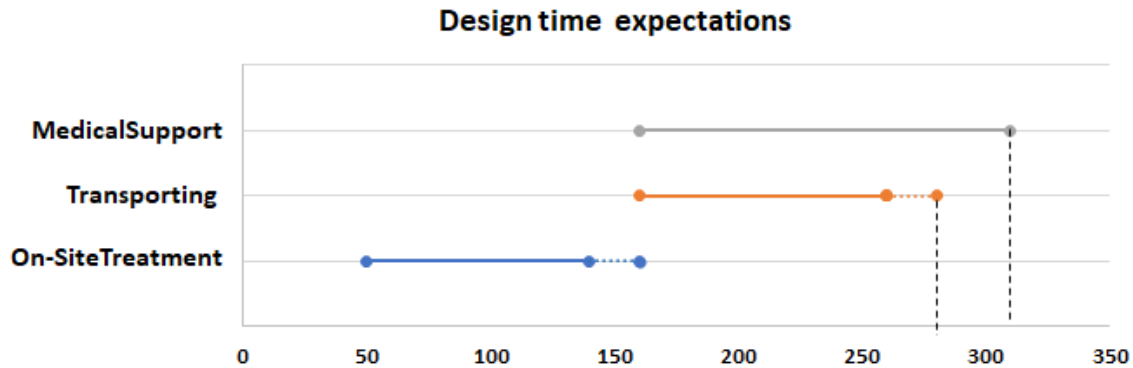


Figure VIII.5: *Time-lapse during design of the three missions.*

The gap between desired behavior and real-time environmental changes highlights a design flaw that requires a dynamic mechanism for any runtime shifts is vital. Using the MAPE-K loop to implement corrective actions to consistently Monitor unforeseen delays, Analyzing system operations and Planning strategies. This allows for parameter and resource reallocations in future Execution addressing issues identified during the design/runtime stage. These corrections ensure that the three missions can be successfully accomplished within the specified time frame 290 units, Figure. VIII.6 and Figure VIII.7 runtime.

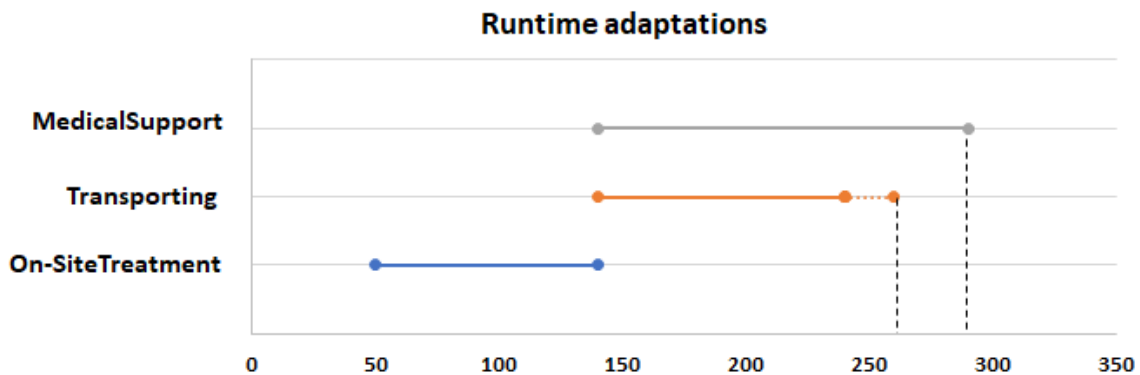
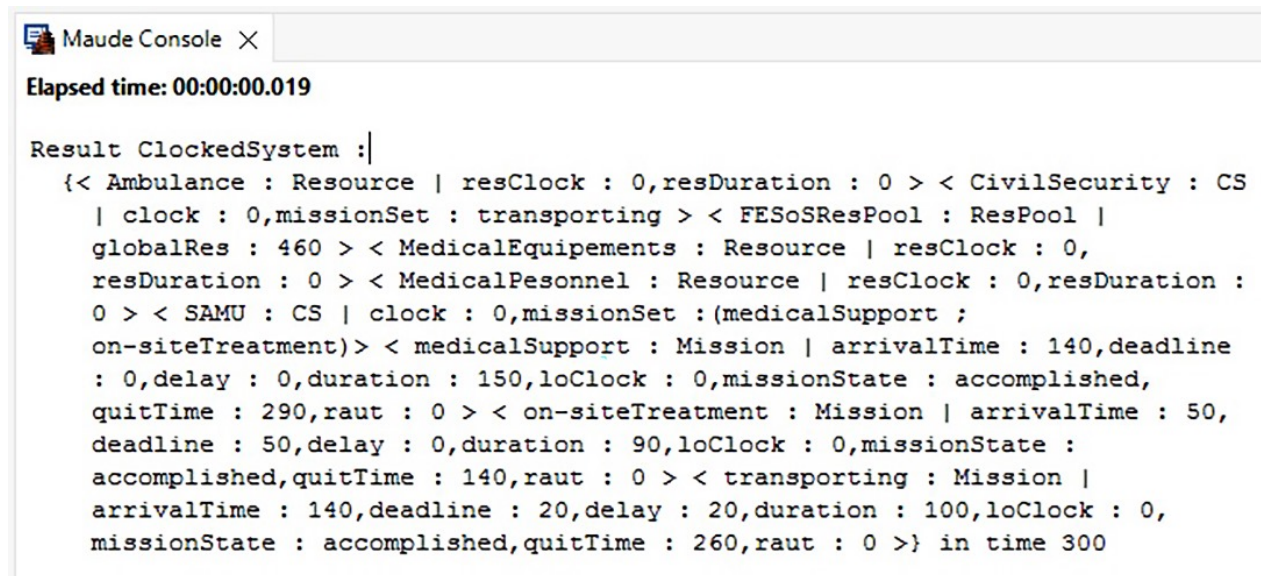


Figure VIII.6: *Time-lapse during Runtime of the three missions.*

We define an initial state `initState` in Figure. VIII.4, which consists of three mentioned missions and requisite resources. Additionally, the FESoS also integrates a resource pool `FESoSResPool`, which represents the available resources in the FESoS. In the runtime phase, the FESoS system can make several corrections to the scenario. One such modification is if the `on-siteTreatment` mission finishes without any delays, the system adjusts its delay from 20 units to 0 units. This adjustment allows the mission to complete its execution earlier than expected. As a result of these changes,

the system manages to self-regulate its behavior. i.e. the on-siteTreatment mission finishes at 140, consequently, both missions transporting and medicalSupport will start earlier than expected at 140 instead of 160. As a result, the former mission will finish at 240 instead of 260, and the latter at 290 instead of 310.

The resource pool reflects the transfer of spare resources, reaching a capacity of 430 units. On the other hand, the required resources for each mission were allocated accordingly: 10 units for on-siteTreatment, 30 units for transporting, and 60 units for medicalSupport. This guarantees adequate resource provision for each mission (see Figure. VIII.7). The remaining resources from the executed missions are transferred to the resource pool. Specifically, 10 units from the transporting mission and 30 units from the medicalSupport mission are added to the resource pool. This transfer of spare resources increases the capacity of the resource pool to 460 units.



```

Maude Console ×
Elapsed time: 00:00:00.019

Result ClockedSystem :|
  {< Ambulance : Resource | resClock : 0,resDuration : 0 > < CivilSecurity : CS
    | clock : 0,missionSet : transporting > < FESoSResPool : ResPool |
    globalRes : 460 > < MedicalEquipements : Resource | resClock : 0,
    resDuration : 0 > < MedicalPersonnel : Resource | resClock : 0,resDuration :
    0 > < SAMU : CS | clock : 0,missionSet :(medicalSupport ;
    on-siteTreatment)> < medicalSupport : Mission | arrivalTime : 140,deadline
    : 0,delay : 0,duration : 150,loClock : 0,missionState : accomplished,
    quitTime : 290,raut : 0 > < on-siteTreatment : Mission | arrivalTime : 50,
    deadline : 50,delay : 0,duration : 90,loClock : 0,missionState :
    accomplished,quitTime : 140,raut : 0 > < transporting : Mission |
    arrivalTime : 140,deadline : 20,delay : 20,duration : 100,loClock : 0,
    missionState : accomplished,quitTime : 260,raut : 0 >} in time 300

```

Figure VIII.7: Runtime execution and control of the three missions.

This successful execution demonstrates how the control enables a centralized control in the system, ensuring that the missions are accomplished within the specified time constraints. By dynamically adjusting parameters, allocating resources, and utilizing the resource pool, the system controls the changes and optimizes mission execution.

VIII.6 Formal verification

Simulation, analysis and verification in Maude system can be provided due to its high-performance rewrite engine. Maude provides a range of commands for system modules enabling exploration of the specified behavior from an initial configuration of the system such as the rewrite, and tfrewrite, search, tsearch etc. The rewrite and frewrite commands each explore just one possible behavior (sequence of rewrites) of a system described by a set of rewrite rules tracing a single execution path from an initial state. The search command in Maude is a versatile tool for exploring (following

a breadth-first strategy) the reachable state space in different ways of systems defined by rewrite rules. This command is particularly powerful for systems where multiple outcomes or states can result from a given initial condition, and it's essential for analyzing systems with complex behaviors or verifying properties across potentially vast numbers of states.

Unlike the other commands, search enables the examination of multiple pathways, thereby providing a more extensive analysis of possible system behaviors or verifying specific properties across a broad set of states. This exploration can be finely tuned through optional arguments that limit the number of solutions or the depth of the search, making it adaptable to the analysis of both finite and infinite-state systems.

```
search [ n, m ] in <ModId> : <Term-1> <SearchArrow> <Term-2>
such that <Condition>.
```

At its core, the search command's syntax incorporates several key components: an optional bound on the number of desired solutions (n), an optional maximum depth for the search (m), the module where the search is to be conducted (though this can be omitted if clear from the context), the initial term or state from which the search begins, the target pattern or condition that the resultant states must match, and the type of search indicated by a specific arrow notation. This notation includes arrows for exactly one step ($=>1$), one or more steps ($=>+$), none or more steps ($=>*$), and exclusively final states ($=>!$). Additionally, an optional condition can be specified, which must be satisfied by the reached states, akin to conditions used in conditional rewrite rules.

In this section, we present the results of simulation and verification experiments conducted to evaluate the performance of the controller and the strategies executed within the FESoS. The objective of the evaluation is to gain insights into the effectiveness and efficiency of managing the behavior. This verification process allows for effective analysis of the SoS' behavior and adherence to the defined proprieties. The search command becomes particularly valuable when verifying complex queries about the system's behavior, such as whether a priority state is reachable under predefined conditions or how various initial conditions can converge to identical or divergent outcomes. We focus on Maude's model-checking, especially checking invariants with the search command i.e. this command undertakes a breadth-first search of system states, verifying important SoSs properties. If certain outcomes, the inverse of the invariant, are unattainable and the invariant is deemed true. If "search init $=>*$ C:Configuration such that not I(C:Configuration)" yields no result, the invariant is valid.

VIII.6.1 Maude-based verification for management strategies

In the context of the FESoS case study, the search command could be used to simulate different scenarios of emergency responses and resource allocations, ensuring that the strategic objectives are met and that the system behaves as expected in various potential emergency situations. In Figure.VIII.2, FESoS case study illustrates two functional chains that have the potential to achieve global mission objectives. These missions within the functional chains are designated as M1, M2,

M3,...,Mn for simplicity and easy reference. Using Maude Strategy Language, these specifications can be immediately executed. The 'search' command in Maude is particularly helpful, enabling exploration of all potential functional chain rules and identifying terms that correspond with a set target.

```
Maude> search initial =>* AlternativeM | PrimaryM M0 M1 M2 M3 M4 M5 M6 M7
M8..... M33 M34 .
```

```
Solution 1 (state 74)
states: 75 rewrites: 123
empty substitution
```

```
no more solutions.
states 80
```

The Maude search command is used to find out whether an initial system state can transition into an expected configuration PrimaryM chain of missions are correctly represents the final goal. The search process involved 123 rewrite steps across 75 possible states and confirmed a single valid state that matches the expected mission sequence, with no variable substitutions required. For this, The search process involved hundreds of rewrite steps across tens of possible states and confirmed a valid path that match the expected mission sequence.

Despite the fact that the desired state can be reached, but there's conflicts regarding whether it adheres to the module's rules. Therefore, the path the search command followed to arrive at the goal position involves navigating through unwanted behavior states. This can be further validated by employing the 'show path' command and referencing the state number.

```
Maude> show path 40 .
states 0, WF: PrimaryM | AlternativeM M0 M1 M2 M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 1, WF: PrimaryM M0 | AlternativeM M1 M2 M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 2, WF: PrimaryM M0 M1 | AlternativeM M2 M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 3, WF: PrimaryM M0 M1 M2 | AlternativeM M3 M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[sequentialMissionOrdering] . ] ===>
state 4, WF: PrimaryM M0 M1 M2 M3 | AlternativeM M4 M5 M6 M7 M8..... M33 M34
===[ rl ...[prioritizeEarliestStart] . ] ===>
state 5, WF: PrimaryM M0 M1 M2 M3 M6 | AlternativeM M4 M5 M6 M7 M8..... M33 M34
.....
```



```
===[ r1 ...[arrivalTimeViolation] . ] ==>
state 40, WF: PrimaryM M0 M1 M2 M3 M6 M7 M8.....M30 | AlternativeM M4 M5 M6 M7
M8..... M33 M34
```

The provided result showing a sequence of states and transitions in a workflow (denoted as WF). This workflow is structured around managing a set of missions (M0, M1, M2, etc.) using two categories: PrimaryM and AlternativeM:

- Initial State (state 0): All missions (M0 through M34) are under AlternativeM. This represents a starting point where all missions are initially in an alternative or secondary queue, awaiting prioritization or scheduling.
- Transitions Using [sequentialMissionOrdering]: The sequence of transitions from state 1 to state 4 appears to follow a rule labeled [sequentialMissionOrdering]. This suggests a process where missions are moved one by one from AlternativeM to PrimaryM. For example, in state 1, M0 is moved to PrimaryM, and in state 2, M1 is also moved, and so on.
- Transition Using [prioritizeEarliestStart]: At state 4, the transition rule changes to [prioritizeEarliestStart], indicating a shift in the strategy for ordering missions. This means that instead of simply moving missions sequentially, the next mission to be moved to PrimaryM is selected based on the criterion of earliest start time. This is evident as M6 is moved to PrimaryM in state 5, skipping M4 and M5.
- Final State (state 40): By state 40, a set of missions (M0, M1, M2, M3, M6, M7, M8, ..., M30) are in PrimaryM, and the remaining ones (M4, M5, M31, M32, M33, M34) are in AlternativeM. The transition to this state is marked by [arrivalTimeViolation], suggesting that this transition might be due to a violation of expected arrival times of missions, impacting the scheduling or prioritization.

VIII.6.2 Model-checking SoSs proprieties

In this subsection, we utilize the runtime changing conditions presented in Table.VII.1 to represent temporal constraints and properties that characterize the requirements of Missions and Resources in the SoS. These constraints and properties are verified using RT-Maude's model-checking invariants tool. We formalize these properties specifically for SoSs as state properties expressed through logical formulas that combine the changing conditions with basic boolean operators. To verify these logical properties, we utilize RT-Maude's tsearch command, which operates on an initial state of the SoS. After each SoS reconfiguration carried out by a self-control action using RT-Maude's rewrite engine, the tsearch command verifies the logical properties. The properties are expected to be satisfied, and if an invariant property matches the current SoS configuration model, it is considered either true or false based on the context. These properties can define various temporal constraints such as arrival time, quit time, and worst-case execution time (WCET) on resource consumption, interactions

between CSs, availability of spare resources in the Resource Pool (ResPool), and more. In the following, we provide examples of such properties that express specific application proprieties:

- Missions' *DeadlineViolation?* Is a crucial for ensuring the timely execution of missions which encompass various aspects, including the time interval between mission arrival and completion, as well as specific requirements for mission start times. Consider a Transporting Mission in our FESoS' scenario, where the designer may require missions to start at specific times to align with operational schedules, resource availability, or other external factors. Additionally, the designer specifies that the mission should be completed within a duration that includes the expected mission duration (duration) plus the WCET. This constraint ensures that the mission is accomplished within a reasonable time frame, accounting for potential delays or unforeseen circumstances. To further refine this constraint, the designer may establish an upper limit on the total time allowed for the mission, known as the deadline. The deadline can be expressed as $(\text{duration} + \text{delay WCET}) < (\text{duration} + \text{delay} + \text{WCET} + \text{deadline})$, ensuring that the mission is completed within the specified timeframe. By incorporating these constraints, missions begin precisely when desired, enabling coordination with other activities or systems. This constraint provides control and synchronization within the FESoS, ensuring that missions are executed according to the overall operational plan.
- Ressources' *MutualExcluion?* SoS designer may impose constraints that govern the production/consumption of spare resources associated. One essential constraint focuses on maintaining confidentiality and preventing the co-consumption of the same resource by multiple missions. For example, consider two Transporting missions executed by two distinct CSs within the SoS. To enforce mutual exclusion for confidentiality, a number of conditions can be defined to formalize the constraint. It ensures that these missions do not co-consume the same resource i.e. specifically an ambulance. These conditions that allow missions to add Excess Resources to the Resource Pool at any time. However, accessing these resources would be subject to certain rules i.e. a mission can only gain access to a resource if it is not being used by any other mission or CS. This ensures exclusive utilization of the resource and prevents conflicts or data breaches. By enforcing this mutual exclusion constraint, the SoS designer ensures that the confidentiality of resources, such as the ambulance, is preserved. Each Transporting mission can operate independently, without interfering with the resource allocation of other missions executing in different CSs.

These properties representing SoSs designers' requirements are formalized in our RT-Maude using several invariants declared within the system module SoS Invariants. The different RT-Maude commands used to run the model-checking of invariants using the FESoS model as an initial configuration are given bellow. This model is formally analyzed and verified with respect to the various properties expressing the introduced formal requirements specific to SoSs architecture. For instance, the timed tsearch command shows that no state isDeadlineViolated (transporting) with $240 \leq t < 260$ can be reached, yielding No solution Figure. VIII.8 meaning that the specified property "is-

DeadlineViolated” is not satisfied, the self-control is managing runtime delays, and the mission can be executed within the defined deadline (without violations) as it was specified in the design phase.

```
(tsearch [1] {isDeadlineViolated(transporting)}  
=>* {isDeadlineViolated(transporting)}  
in time-interval between >= 240 and < 260 .)
```

```
(tsearch [1] {mutualExclusion(transportingSAMU, transportingCS, ambulance)}  
=>* {mutualExclusion(transportingSAMU, transportingCS, ambulance)}  
in time-interval between >= 240 and < 260 .)
```

```
rewrites: 8751 in 6374638180ms cpu (139ms real) (0 rewrites/second)  
  
Timed search [1] in TEST-MANY-RTTS  
  {isDeadlineViolated(transporting)} =>* {isDeadlineViolated(  
    transporting)}  
in time between >= 100 and < 200 and with mode default time increase 1 :  
  
No solution
```

Figure VIII.8: *Search results of Transporting mission.*

VIII.7 Conclusion

In this final chapter, we have conducted a comprehensive evaluation of the Resource Allocation Centralized Controller within the French Emergency SoS. We begin by detailing a series of simulations to assess the RAC’s impact on mission execution and resource allocation during critical scenarios. We then confirm the effectiveness of management strategies enabled by the RAC, demonstrating their capability to regulate SoS behavior in line with the centralized control architecture. We also analyze the integration of the MAPE-K loop, highlighting its role in adapting to dynamic changes and emergencies. Finally, we discuss the application of formal verification methods through the Maude system, which validate that the SoS meets its design and functional specifications, thereby ensuring the reliability of the RAC’s strategies in emergency management contexts.

Chapter IX

General Conclusion

Contents

IX.1	Conclusion	177
IX.2	Perspectives	180

IX.1 Conclusion

Nowadays, Systems of Systems (SoSs) represent a new paradigm promising substantial scientific value for industry and academia. They are characterized by their ability to offer new functionalities that individual CSs cannot provide, resulting from their combination. CSs are operationally independent, geographically distributed, developed with different technologies, and intended for different platforms. Moreover, they are increasingly present in numerous applications (healthcare, transportation, military, avionics, etc.). However, the complexity of the dynamic nature of a SoS's components presents a significant challenge for modeling and analysis. This complexity means that the SoS development process must follow a rigorous approach to meet the often critical requirements of these systems. All existing design methods and formalisms, such as semi-formal methods like UML and SysML, and formal methods like ADL and BRS, are specific to particular system examples and do not suggest generic and effective solutions.

Moreover, the design of SoSs requires a significant level of reliability, which can only be achieved through the use and combination of both methods, i.e., semi-formal methods provide an intuitive and flexible framework for capturing architectural design and facilitating communication among stakeholders, and formal methods offer rigorous and mathematically founded tools to ensure the accuracy and reliability of the SoS, especially in critical areas. Consequently, to leverage both, the manuscript aims to combine the strength of both formalisms to create robust, reusable, and executable models to facilitate the design, simulation, and verification of complex SoS. Moreover, it

is necessary to evaluate design choices and reason about the runtime behaviors of these systems very early during the development phases. This context is where our work is situated. Its main goal is to contribute to the development of a new approach for the architectural design and formal analysis of real-time SoSs, based on revised rewriting logic.

In the context of this work, the focus has been on proposing a methodology for SoS engineering, which provides a comprehensive model for the design and analysis of SoS. This model facilitates the development of a systematic multi-view architectural framework, addressing the complexity of SoS and allowing the integration of heterogeneous CS. The formal method ensures the consistency of quantitative properties and allows the description and modeling of the SoS's structure and behaviors. Moreover, the methodology proposes mechanisms to guarantee the autonomous excitability of behaviors and provides means to verify their accuracy. The integration of appropriate formal methods to support the dynamic and concurrent aspects of these systems as well as their temporal analysis proves very useful. It motivates the use of rewriting logic through its extended versions. In the following sections, we outline our main contributions and some future perspectives that could constitute the possible follow-ups for this work.

1. Classification of SoS modeling approaches: A state-of-the-art review concerning SoS engineering and modeling was established to motivate the need to adopt a new integrated approach based on MDA and formal method.

2. Definition of AF-SoS: We have introduced a Multi-viewpoints approach-based Architecture Framework, that is associated with a set of SoSE development processes and a UML profile dedicated to the SoSs that can facilitate and improve the design of SoSs' architectures. The purpose of this work is to present an AF that intends to foster the systematic development of SoSs' architectures. The key elements of this paper are:

- The proposal of an AF that encompasses the concepts proposed by “ISO/IEC/IEEE 42010:2011 Systems and Software Engineering-Architecture description” to manage the complexity of SoSs' architectures following a multi-viewpoint approach.
- Improving the potential of the AF by integrating systematic processes of developing SoSs' architectures which can have a positive impact on the overall quality of the framework. It should be noted that to our knowledge, there is no available SoSE process in the literature to support the design of IEEE 42010 standard.
- Defining an SoS-UML Profile to model a set of SoSs' processes. The advantage of this profile is to provide a large number of models to separately capture, describe and organize each of the processes of different viewpoints.
- Demonstration of the work by designing an SoSs' architecture of an “Aircraft Emergency Response System-of-Systems” (AERSoS). It is therefore the aim of the paper to show that a unified AF for SoSs' architectures abstractions are needed.

3. Centralized Control-based formal approach: We have proposed a formal description of a centralized Control which encompasses the variabilities and abstract assets within the proposed

methodology, particularly focusing on the Resource Allocation Controller (RAC). The latter is depicted as central mean to managing the different variabilities of directed SoSs, ensuring resources are allocated efficiently and missions are executed within their specific conditions. This is achieved through the definition of the: (1) Static structures to describe the fundamental entities within the SoS—Missions, Resources, Roles, and the RAC itself—each with specific quantitative properties. And (2) Dynamic behavior where operational semantics are applied, encoding the behaviors into Maude’s syntax through the use of predicates and actions that facilitate the transitions among states of each entity.

4. Formalization of Management Strategies: We have proposed an approach for enhancing self-management in SoSs, emphasizing the integration of dynamic strategies to manage both desired and undesired behaviors using Maude Strategy Language to offer a formal method to define operational semantics. Here, He have addressed the complexities of workflow management, mission execution taking into consideration resource allocation and conflict resolution. The proposed method aims to improve system functionality and resource efficiency in complex SoS environments, showcasing its applicability through detailed cases and theoretical foundations.

5. MAPE-K loop feedback for adjustments: Leveraging the power of the MAPE-K control loop and the formal analysis and verification methods of Maude, we have proposed an approach which follows a systematic iteration between the design phase and the runtime execution. We have illustrated how the modules specifying the various contributions can be organized into an autonomic operational framework. Through the application of the MAPE-K loop phases, we have demonstrated their integration and functionality within the context of the SoS. This approach not only showcases the systematic organization of the specified modules but also allows for refinement and continuous improvement of the design and runtime execution. Feedback from the runtime phase informs the design phase, leading to adjustments in the concrete model.

6. Real-Time regulating mechanisms: In order to enrich the MAPE-K loop control, we have proposed a set of real-time corrective actions to enable the dynamic adjustment of missions behavior based on changing runtime conditions and requirements. The approach captures the essential structural, behavioral, and quantitative aspects related to the SoS’s functionalities, and enables timely adjustments to ensure that mission objectives are met within the specified time frames. i.e. based on the analysis of mission execution scenarios, the system formulates appropriate corrective actions to address the identified issues. For scenarios where no control is needed, the system continues executing missions according to their planned sequence and timing. For early arrival control, control mechanisms are employed to synchronize subsequent missions and optimize system efficiency. For late completion control, the system adjusts the sequencing of sequel missions and accommodates delays.

7. Execution and formal verification:

Through a case study, we have demonstrated the practical application of the proposed approaches through runtime simulation, analysis, and verification. We have defined the specified behaviors of the components, the RAC, Management strategies and Real-Time Self-Regulating mechanism. This means verifying that autonomously executed behaviors and actions are well-defined, correct, and

describe a desirable behavior at design time and runtime stages. This involves defining several quantitative properties that the system must satisfy and guarantee throughout its runtime. To this end, we have used the formal verification techniques of SoSs' behavior using Maude's model-checking capabilities. It verifies that the system meets its strategic objectives and operates as expected under various emergency scenarios. This involves checking invariants and using the search command in Maude to explore possible system behaviors and ensure compliance with defined properties. Therefore, by validating the effectiveness of the proposed system design, strategic planning, and control mechanisms in optimizing emergency response operations, this phase addresses the fourth research objective.

IX.2 Perspectives

Here we identify two main perspectives in the short and long term to continue the scientific work presented in this manuscript. From a practical standpoint, the first perspective relates to the development of a fully assisted tool for designing and analyzing SoSs and their components. From a theoretical viewpoint, the second perspective involves studying the control and management strategies for more expansion and enhancement.

1. Development of an assisted tool for the specification, verification, and evaluation of SoSs and their CSs

Development of an assisted tool for the specification, verification, and evaluation of elasticity. In the short term, it would be interesting to develop a completely automated and assisted tool from the modeling phase to the analysis of the results obtained from the proprieties verification. Ideally, the tool should provide an intuitive graphical interface that allows a designer to easily model their SoSs and CSs using the proposed Meta-Models. From this modeling, the idea is to create a translator that would enable the generation of Maude specifications and configuration from the designed models. Finally, another component would allow for the execution and analysis of the described control and management behaviors to proceed with their quantitative verification according to criteria defined by the designers. Ultimately, this tool would also incorporate simulation and monitoring capabilities for the modeled SoSs and CSs to facilitate their execution.

2. Expanding the strategies

In the middle term, we aim to expand upon our initial exploration of centralized control and management strategies for SoSs. This involves advancing the experimental study and quantitative validation processes. Future efforts will focus on the formalization, the verification and the quantitative analysis of the main self-* properties (such as self-adaptation, self-awareness, self-stabilization, etc.) to support a wider range of behaviors, including both predictable and unpredictable workflows in SoSs. Additionally, we plan to refine our evaluation approach to better predict resource needs and missions performance pre-deployment, thereby offering SoSs a more robust tool for planning and optimizing their infrastructure. This direction will allow us to explore richer behaviors and their impact on high-level policies like performance, cost efficiency, and resource optimization.

Bibliography

- [1] T. M. Aljohani, “Analysis of the smart grid as a system of systems,” *arXiv preprint arXiv:1810.11453*, 2018.
- [2] M. E. Arass, K. Ouazzani-Touhami, and N. Souissi, “The system of systems paradigm to reduce the complexity of data lifecycle management. case of the security information and event management,” *International Journal of System of Systems Engineering*, vol. 9, no. 4, pp. 331–361, 2019.
- [3] M. A. Assaad, R. Talj, and A. Charara, “A view on systems of systems (sos),” in *20th World Congress of the International Federation of Automatic Control (IFAC WC 2017)-special session*, 2016.
- [4] J. Axelsson, J. Fröberg, and P. Eriksson, “Architecting systems-of-systems and their constituents: A case study applying industry 4.0 in the construction domain,” *Systems Engineering*, vol. 22, no. 6, pp. 455–470, 2019.
- [5] Y.-M. Baek, J. Song, Y.-J. Shin, S. Park, and D.-H. Bae, “A meta-model for representing system-of-systems ontologies,” in *Proceedings of the 6th International Workshop on Software Engineering for Systems-of-Systems*, 2018, pp. 1–7.
- [6] J. A. Bossard, C. P. Scarborough, Q. Wu, *et al.*, “Mitigating field enhancement in metasurfaces and metamaterials for high-power microwave applications,” *IEEE Transactions on Antennas and Propagation*, vol. 64, no. 12, pp. 5309–5319, 2016.
- [7] A. W. Brown, “Model driven architecture: Principles and practice,” *Software and systems modeling*, vol. 3, pp. 314–327, 2004.
- [8] I. Cherfa, N. Belloir, S. Sadou, R. Fleurquin, and D. Bennouar, “Systems of systems: From mission definition to architecture description,” *Systems Engineering*, vol. 22, no. 6, pp. 437–454, 2019.
- [9] M. Clavel, F. Durán, S. Eker, *et al.*, *All about maude-a high-performance logical framework: how to specify, program, and verify systems in rewriting logic*. Springer, 2007, vol. 4350.

-
- [10] M. Clavel, F. Durán, S. Eker, *et al.*, “Maude manual (version 3.1),” *SRI International University of Illinois at Urbana-Champaign* <http://maude.lcc.uma.es/maude31-manual-html/maude-manual.html>, 2020.
- [11] M. Clavel, F. Durán, S. Eker, *et al.*, “Maude: Specification and programming in rewriting logic,” *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.
- [12] M. Clavel, F. Durán, S. Eker, *et al.*, “The maude 2.0 system,” in *International Conference on Rewriting Techniques and Applications*, Springer, 2003, pp. 76–87.
- [13] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, “Principles of maude,” *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 65–89, 1996.
- [14] P. Clements and L. Northrop, *Software product lines*. Addison-Wesley Boston, 2002.
- [15] D. Cocks, “How should we use the term “system of systems” and why should we care?” In *INCOSE International Symposium*, Wiley Online Library, vol. 16, 2006, pp. 427–438.
- [16] A. Computing *et al.*, “An architectural blueprint for autonomic computing,” *IBM White Paper*, vol. 31, no. 2006, pp. 1–6, 2006.
- [17] J. S. Dahmann, “Systems of systems characterization and types,” *Systems of Systems Engineering for NATO Defence Applications (STO-EN-SCI-276)*, pp. 1–14, 2015.
- [18] F. DAU, “Defense acquisition guidebook,” *Defense Acquisition University, VA*, 2004.
- [19] D. DeLaurentis, “Understanding transportation as a system-of-systems design problem,” in *43rd AIAA aerospace sciences meeting and exhibit*, 2005, p. 123.
- [20] C. E. Dridi, Z. Benzadri, and F. Belala, “A unified architecture framework supporting sos’s development: Case of the aircraft emergency response system-of-systems,” *International Journal of Organizational and Collective Intelligence (IJOICI)*, vol. 13, no. 1, pp. 1–30, 2023.
- [21] C. E. Dridi, Z. Benzadri, and F. Belala, “System of systems engineering: Meta-modelling perspective,” in *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, IEEE, 2020, pp. 000 135–000 144.
- [22] C. E. Dridi, Z. Benzadri, and F. Belala, “Towards a multi-viewpoints approach for the sos engineering,” in *2022 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, IEEE, 2022, pp. 1–6.
- [23] C. E. Dridi, Z. Benzadri, and F. Belala, “System of systems modelling: Recent work review and a path forward,” in *2020 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, IEEE, 2020, pp. 1–8.
- [24] C. E. Dridi, N. Hameurlain, and F. Belala, “A maude-based formal approach to control and analyze time-resource aware missioned systemsof-systems,” in *31th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE-2023)*, IEEE, 2023, 6.p.

Bibliography

- [25] C. E. Dridi, N. Hameurlain, and F. Belala, “A maude-based rewriting approach to model and control system-of-systems’ resources allocation,” in *International Conference on Model and Data Engineering*, Springer, 2022, pp. 207–221.
- [26] A. Egyed and N. Medvidovic, “A formal approach to heterogeneous software modeling,” in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2000, pp. 178–192.
- [27] J. El Hachem, Z. Y. Pang, V. Chiprianov, A. Babar, and P. Aniorte, “Model driven software security architecture of systems-of-systems,” in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2016, pp. 89–96.
- [28] H.-E. Eriksson, M. Penker, B. Lyons, and D. Fado, *UML 2 toolkit*. John Wiley & Sons, 2003.
- [29] A. Gassara, I. Bouassida, and M. Jmaiel, “A tool for modeling sos architectures using bigraphs,” in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 1787–1792.
- [30] A. Gassara, I. B. Rodriguez, M. Jmaiel, and K. Drira, “A bigraphical multi-scale modeling methodology for system of systems,” *Computers & Electrical Engineering*, vol. 58, pp. 113–125, 2017.
- [31] T. Gezgin, C. Etzien, S. Henkler, and A. Rettberg, “Towards a rigorous modeling formalism for systems of systems,” in *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, IEEE, 2012, pp. 204–211.
- [32] M. B. Gonçalves, E. Cavalcante, T. Batista, F. Oquendo, and E. Y. Nakagawa, “Towards a conceptual model for software-intensive system-of-systems,” in *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2014, pp. 1605–1610.
- [33] V. Gunes, S. Peter, T. Givargis, and F. Vahid, “A survey on concepts, applications, and challenges in cyber-physical systems,” *KSII Trans. Internet Inf. Syst.*, vol. 8, no. 12, pp. 4242–4268, 2014.
- [34] M. C. Hause, “Sos for sos: A new paradigm for system of systems modeling,” in *2014 IEEE Aerospace Conference*, IEEE, 2014, pp. 1–12.
- [35] J. Hu, L. Huang, X. Chang, and B. Cao, “A model driven service engineering approach to system of systems,” in *2014 IEEE International Systems Conference Proceedings*, IEEE, 2014, pp. 136–145.
- [36] W. C. In and M. H. LinLee, “Informal, semi-formal, and formal approaches to the specification of software requirements,” Ph.D. dissertation, University of British Columbia, 1994.
- [37] I. ISO, “Ieee: Systems and software engineering—architecture description,” *International Organization for Standardization ISO/IEC/IEEE*, vol. 42010, 2011.
- [38] ISO/IEC/IEEE, “Systems and software engineering – architecture description,” *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1–46, Jan. 2011. DOI: [10.1109/IEEESTD.2011.6129467](https://doi.org/10.1109/IEEESTD.2011.6129467).

-
- [39] *Iso/iec/ieee international standard - systems and software engineering*. DOI: [10.1109/IEEESTD.2015.7106435](https://doi.org/10.1109/IEEESTD.2015.7106435). [Online]. Available: <https://ieeexplore.ieee.org/document/7106435>.
- [40] M. Jamshidi, "System of systems engineering-new challenges for the 21st century," *IEEE Aerospace and Electronic Systems Magazine*, vol. 23, no. 5, pp. 4–19, 2008.
- [41] N. Karcianas and A. G. Hessami, "Complexity and the notion of system of systems: Part (ii): Defining the notion of system of systems," in *2010 World Automation Congress*, IEEE, 2010, pp. 1–7.
- [42] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
- [43] I. Khlif, M. H. Kacem, C. Eichler, and A. H. Kacem, "A multi-scale modeling approach for systems of systems architectures," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 3, pp. 17–26, 2017.
- [44] C. Kirchner, H. Kirchner, and M. Vittek, "Designing constraint logic programming languages using computational systems," *Principles and Practice of Constraint Programming. The Newport Papers*, pp. 131–158, 1995.
- [45] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.
- [46] V. Kotov, *Systems of systems as communicating structures*. Hewlett Packard Laboratories, 1997, vol. 119.
- [47] J. A. Lane and T. Bohn, "Using sysml modeling to understand and evolve systems of systems," *Systems Engineering*, vol. 16, no. 1, pp. 87–98, 2013.
- [48] J. A. Lane and D. Epstein, "What is a system of systems and why should i care?" *University of Southern California*, 2013.
- [49] J. Lönngren and K. Van Poeck, "Wicked problems: A mapping review of the literature," *International Journal of Sustainable Development & World Ecology*, vol. 28, no. 6, pp. 481–502, 2021.
- [50] Z. Maamar, N. Faci, S. Sakr, M. Boukhebouze, and A. Barnawi, "Network-based social coordination of business processes," *Information Systems*, vol. 58, pp. 56–74, 2016.
- [51] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering: The Journal of the International Council on Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.
- [52] M. W. Maier, D. Emery, and R. Hilliard, "Ansi/ieee 1471 and systems engineering," *Systems engineering*, vol. 7, no. 3, pp. 257–270, 2004.
- [53] N. Martí-Oliet and J. Meseguer, "Rewriting logic as a logical and semantic framework," *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 190–225, 1996.
- [54] N. Martí-Oliet, J. Meseguer, and A. Verdejo, "A rewriting semantics for maude strategies," *Electronic Notes in Theoretical Computer Science*, vol. 238, no. 3, pp. 227–247, 2009.

Bibliography

- [55] N. Martí-Oliet, J. Meseguer, and A. Verdejo, “Towards a strategy language for maude,” *Electronic Notes in Theoretical Computer Science*, vol. 117, pp. 417–441, 2005.
- [56] S. J. Mellor, *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.
- [57] J. Meseguer, “Conditional rewriting logic as a unified model of concurrency,” *Theoretical computer science*, vol. 96, no. 1, pp. 73–155, 1992.
- [58] J. Meseguer and G. Roşu, “Rewriting logic semantics: From language specifications to formal analysis tools,” in *International Joint Conference on Automated Reasoning*, Springer, 2004, pp. 1–44.
- [59] M. Mori, A. Ceccarelli, P. Lollini, B. Frömel, F. Brancati, and A. Bondavalli, “Systems-of-systems modeling using a comprehensive viewpoint-based sysml profile,” *Journal of Software: Evolution and Process*, vol. 30, no. 3, e1878, 2018.
- [60] R. Morrison, G. Kirby, D. Balasubramaniam, *et al.*, “Support for evolving software architectures in the archware adl,” in *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, IEEE, 2004, pp. 69–78.
- [61] C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, “Systems of systems engineering: Basic concepts, model-based techniques, and research directions,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–41, 2015.
- [62] P. C. Ölveczky and J. Meseguer, “Abstraction and completeness for real-time maude,” *Electronic Notes in Theoretical Computer Science*, vol. 176, no. 4, pp. 5–27, 2007.
- [63] P. C. Ölveczky and J. Meseguer, “Recent advances in real-time maude,” *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 1, pp. 65–81, 2007.
- [64] P. C. Ölveczky and J. Meseguer, “Semantics and pragmatics of real-time maude,” *Higher-order and symbolic computation*, vol. 20, pp. 161–196, 2007.
- [65] F. Oquendo, “ π -calculus for sos: A foundation for formally describing software-intensive systems-of-systems,” in *2016 11th System of Systems Engineering Conference (SoSE)*, IEEE, 2016, pp. 1–6.
- [66] F. Oquendo, “Formally describing the architectural behavior of software-intensive systems-of-systems with sosadl,” in *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE, 2016, pp. 13–22.
- [67] F. Oquendo, “Formally describing the software architecture of systems-of-systems with sosadl,” in *2016 11th system of systems engineering conference (SoSE)*, IEEE, 2016, pp. 1–6.
- [68] D. L. Parnas, “On the design and development of program families,” *IEEE Transactions on software engineering*, no. 1, pp. 1–9, 1976.
- [69] H. A. Partsch, *Specification and transformation of programs: a formal approach to software development*. Springer Science & Business Media, 2012.

-
- [70] J. D. Poole, “Model-driven architecture: Vision, standards and emerging technologies,” in *Workshop on Metamodeling and Adaptive Object Models, ECOOP*, vol. 50, 2001.
- [71] M. Rao, S. Ramakrishnan, and C. Dagli, “Modeling and simulation of net centric system of systems using systems modeling language and colored petri-nets: A demonstration using the global earth observation system of systems,” *Systems Engineering*, vol. 11, no. 3, pp. 203–220, 2008.
- [72] A. Reichwein and C. Paredis, “Overview of architecture frameworks and modeling languages for model-based systems engineering,” in *Proc. ASME*, vol. 1275, 2011.
- [73] J. E. Rivera, F. Durán, and A. Vallecillo, “A metamodel for maude,” *Technical report*, 2008.
- [74] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo, “Parameterized strategies specification in maude,” in *Recent Trends in Algebraic Development Techniques: 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 2–5, 2018, Revised Selected Papers 24*, Springer, 2019, pp. 27–44.
- [75] R. R. Rubio Cuéllar, N. Martí Oliet, I. Pita Andreu, and J. A. Verdejo López, “The semantics of the maude strategy language,” 2021.
- [76] J. Rumbaugh, *The unified modeling language reference manual*. Pearson Education India, 2005.
- [77] V. M. Santos, S. Misra, and M. S. Soares, “Architecture conceptualization for health information systems using iso/iec/ieee 42020,” in *International Conference on Computational Science and Its Applications*, Springer, 2020, pp. 398–411.
- [78] D. C. Schmidt *et al.*, “Model-driven engineering,” *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.
- [79] A. Seghiri, F. Belala, and N. Hameurlain, “A formal language for modelling and verifying systems-of-systems software architectures,” *International journal of systems and service-oriented engineering (IJSSOE)*, vol. 12, no. 1, pp. 1–17, 2022.
- [80] E. Silva, T. Batista, and F. Oquendo, “On the verification of mission-related properties in software-intensive systems-of-systems architectural design,” *Science of Computer Programming*, vol. 192, p. 102 425, 2020.
- [81] R. Soley *et al.*, “Model driven architecture,” *OMG white paper*, vol. 308, no. 308, p. 5, 2000.
- [82] O. A. Specification, “Omg unified modeling language (omg uml), superstructure, v2. 1.2,” *Object Management Group*, vol. 70, 2007.
- [83] C. Stary and D. Wachholder, “System-of-systems support—a bigraph approach to interoperability and emergent behavior,” *Data & Knowledge Engineering*, vol. 105, pp. 155–172, 2016.
- [84] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, “A classification and survey of analysis strategies for software product lines,” *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 1–45, 2014.
- [85] N. S. W. C. D. D. VA, “Leading edge. volume 7, issue number 4, 2012,” 2012.

Bibliography

- [86] A. Verdejo and N. Martí-Oliet, “Basic completion strategies as another application of the maude strategy language,” *arXiv preprint arXiv:1204.5542*, 2012.
- [87] N. Wickramasinghe, S. Chalasani, R. V. Boppana, and A. M. Madni, “Healthcare system of systems,” in *2007 IEEE International Conference on System of Systems Engineering*, IEEE, 2007, pp. 1–6.
- [88] R. Wleringa and E. Dubois, “Integrating semi-formal and formal software specification techniques,” *Information Systems*, vol. 23, no. 3-4, pp. 159–178, 1998.
- [89] Y. Zhang, X. Liu, Z. Wang, and L. Chen, “A service-oriented method for system-of-systems requirements analysis and architecture design,” *J. Softw.*, vol. 7, no. 2, pp. 358–365, 2012.