



HAL
open science

ArchSoS : un langage de description d'architectures avancées des systèmes de systèmes

Akram Seghiri

► **To cite this version:**

Akram Seghiri. ArchSoS : un langage de description d'architectures avancées des systèmes de systèmes. Arithmétique des ordinateurs. Université de Pau et des Pays de l'Adour; Université Constantine 2 - Abdelhamid Mehri, 2022. Français. NNT : 2022PAUU3031 . tel-04117671

HAL Id: tel-04117671

<https://theses.hal.science/tel-04117671v1>

Submitted on 5 Jun 2023

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.



UNIVERSITÉ
CONSTANTINE 2
ABDELHAMID MEHRI



Université Constantine 2 Abdelhamid Mehri - Faculté des Nouvelles Technologies
de l'Information et de la Communication - Département des Technologies
Logicielles et Systèmes d'Information
Université de Pau et des Pays de l'Adour - École doctorale des Sciences Exactes
et leurs Applications (ED SEA 211)

ArchSoS : Un langage de description d'architectures avancées des systèmes de systèmes

THÈSE

en vue de l'obtention du grade de

Docteur en informatique

préparée dans le cadre d'une cotutelle entre

Université Constantine 2 - Abdelhamid Mehri

et

Université de Pau et des Pays de l'Adour

présentée et soutenue par

Akram SEGHIRI

le 11/Juin/2022 à l'Université de Constantine 2-Abdelhamid Mehri

Composition de jury :

Pr. Meriem <i>BELGUIDOUM</i>	<i>Université Constantine 2 - Abdelhamid Mehri, Algérie</i>	Présidente
Dr. Ouassila <i>HIOUAL</i>	<i>Université de Khenchela, Algérie</i>	Examinatrice
Pr. Ladjel <i>BELLATRECHE</i>	<i>Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, France</i>	Rapporteur
Dr. Ismael Rodriguez <i>BOUASSIDA</i>	<i>Université de Sfax, Tunisie</i>	Rapporteur
Pr. Faiza <i>BELALA</i>	<i>Université Constantine 2 - Abdelhamid Mehri, Algérie</i>	Directrice
Dr. Nabil <i>HAMEURLAIN</i>	<i>Université de Pau et des Pays de l'Adour, France</i>	Diecteur

Laboratoire d'Informatique RÉpartie (LIRE)
Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour
(LIUPPA EA 3000)

*“The only place where success comes
before work is in the dictionary.”*

Vidal Sassoon

Remerciements

Je n'aurais jamais pu finaliser ce travail sans le soutien d'un grand nombre de personnes qui m'ont permis de progresser dans cette phase aussi bien instructive que délicate pour mener cette thèse à son terme. C'est pourquoi il me sera très difficile de remercier tout le monde.

Je voudrais exprimer tout d'abord ma profonde gratitude à mes deux directeurs de thèse : Madame Faiza Belala, Professeur à l'Université Constantine 2 – Abdelhamid Mehri, pour sa gentillesse, sa disponibilité, son soutien sans faille, sa patience, ses riches conseils et ses exigences de rigueur depuis mes premiers pas jusqu'à l'intégration totale dans la recherche, et Monsieur Nabil Hameurlain, maître de conférences habilité à diriger des recherches à l'université de Pau et des Pays de l'Adour (UPPA) qui a toujours su trouver les mots justes pour me motiver et me donner l'envie d'avancer dans ma réflexion. Qu'il sache aussi combien j'ai été touchée par la confiance qu'il m'a toujours accordée, et ce, depuis qu'il a accepté de m'accueillir dignement au sein de l'équipe GL du laboratoire LIUPPA. Ce travail ne pourrait être réalisé sans leurs aides, appuis et soutiens.

J'adresse tous mes remerciements à Monsieur Ladjel Bellatreche, Professeur à l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique (ENSMA), France, ainsi qu'à Monsieur Ismael Rodriguez Bouassida, maître de conférences habilité à diriger des recherches à l'université de Sfax, Tunisie, de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse, et pour le temps qu'ils ont consacré à mon travail.

Je tiens également à remercier Madame Meriem Belguidoum, professeur à l'Université Constantine 2 - Abdelhamid Mehri, pour m'avoir fait l'honneur d'accepter de présider le jury de ma thèse. Ainsi que Madame Ouassila Hioual, maître de conférences habilitée à diriger des recherches, d'avoir acceptée d'être dans mon jury de thèse en tant qu'examinatrice. Je les remercie pour l'honneur qu'elles me font, pour l'intérêt qu'elles ont bien voulu porter à mes travaux. Qu'elles trouvent ici l'expression de ma profonde reconnaissance.

Un grand merci aux membres du laboratoire LIRE et du département TLSI, ainsi que les membres du LIUPPA du site de Pau et le personnel de l'école doctorale SEA (ED 211) de l'UPPA pour leur support, tout au long de la réalisation de mes travaux.

Au terme de ce parcours, je remercie celles et ceux qui me sont chers et que j'ai quelque peu délaissés ces derniers mois pour achever cette thèse. Leurs attentions et encouragements m'ont accompagné tout au long de ces années. Merci à mes parents, mon frère et ma sœur, ma chère femme, mes amis et mes collègues pour leur soutien moral et leur précieuse aide. Et enfin, je veux remercier tous ceux qui m'ont aidé de près ou de loin dans l'élaboration et la finalisation de ce travail.

Résumé

Les systèmes de systèmes (ou SoS pour "Systems of Systems") sont une classe de systèmes dont les constituants sont des systèmes autonomes, qui collaborent dans une hiérarchie spécifique pour aboutir à un but commun (appelé mission) qu'aucun d'eux ne peut accomplir à lui seul. Ces constituants sont dotés de la particularité d'avoir une opérabilité et une évolution indépendante, ce qui résulte à des missions imprévisibles. Ils peuvent aussi être hétérogènes, ce qui incrémente le degré de complexité du SoS qui les contient. Fournir un plan d'action précis pour gérer et capturer ses missions n'est pas une tâche triviale.

Les travaux de cette thèse visent à pallier ce manque, en proposant une solution modulaire, réutilisable, générique et basée sur un cadre formel pour décrire les SoS et leurs constituants. Elle consiste à la description et l'analyse des architectures logicielles des SoS en définissant un nouveau langage de description d'architectures "ArchSoS", dédié aux SoS, offrant ainsi un niveau d'abstraction qui permet de réduire la complexité de ce type de systèmes.

Dans un premier temps, nous définissons une architecture de référence pour les SoS, inspirée de la norme *ISO/IEC/IEEE 42010*. Cette architecture sert à l'association d'une syntaxe concrète, qui peut être graphique ou textuelle, à ArchSoS. Ensuite, nous adoptons les *Systèmes Réactifs Bigraphiques* (BRS) pour définir une syntaxe abstraite et formelle de ce langage. L'intérêt d'utiliser ce formalisme réside dans le fait que les deux aspects structurel et comportemental sont considérés respectivement à l'aide des bigraphes et des règles de réactions bigraphiques. D'autre part, cette syntaxe formelle d'ArchSoS peut aussi avoir deux vues : une vue graphique visuelle et une vue algébrique textuelle.

Nous complétons la définition d'ArchSoS en lui associant une sémantique opérationnelle à base de la logique de réécriture. Tous les aspects intégrés dans les descriptions des SoS à base d'ArchSoS (*hiérarchie, contraintes fonctionnelles, événements, missions, etc.*) trouvent leur interprétation sémantique dans le langage de stratégies Maude, qui est une extension du langage Maude. L'utilisation de cette extension a permis de combler le manque qui existait dans la version basique de Maude, où la réécriture des états d'un SoS peut être guidée afin d'accomplir telle ou telle mission en séquentiel et même en parallèle. Il suffit de définir des stratégies pour guider l'évolution du comportement d'un SoS.

La validation de nos contributions théoriques est faite sur une étude de cas pratique et d'une grandeur naturelle, il s'agit d'appliquer toutes les étapes de notre approche (spécification, exécution du modèle et son analyse) pour concevoir et prototyper l'architecture dynamique d'un SoS de réponses aux crises ou CRSoS ("Crisis Response SoS").

Mots Clés : Architecture Logicielle, Systèmes de systèmes, Langage de description architecturale, Méthodes formelles, Systèmes Réactifs Bigraphiques, Logique de réécriture, Maude, Langage de stratégies Maude, Reconfiguration dynamique.

Abstract

Systems of Systems (SoS) are a class of systems whose constituents are autonomous systems. They collaborate in a specific hierarchy to achieve a common goal (called mission) that none of them can accomplish alone. These constituents have the particularity of owning independent operability and evolution, which results in unpredictable missions. They can also be heterogeneous increasing the complexity of the SoS. So, providing a precise plan of action to manage and capture these missions is not a trivial task.

The work of this thesis aims to overcome this shortcoming, by proposing a modular, reusable, generic and formal solution to describe SoS and their constituents. It consists in the definition of a new Architecture Description Language (ADL) "ArchSoS", dedicated to SoS, and thus offering a high level of abstraction allowing to reduce the complexity of this type of systems.

First, we define a reference architecture for SoS, inspired from the *ISO/IEC/IEEE 42010* standard. This architecture is used to associate a concrete syntax, which can be graphical or textual, to ArchSoS. Then, we adopt *Bigraphic Reactive Systems* (BRS) to define the abstract and formal syntax of this language. The interest of using this formalism lies in the fact that the two structural and behavioral aspects are considered respectively. On the other hand, this formal syntax of ArchSoS can also be represented with a graphical visual view and an algebraic textual view.

We complete the definition of ArchSoS by associating an operational semantic, based on rewriting logic, to this language. All aspects embedded in ArchSoS-based SoS descriptions (hierarchy, functional constraints, events, missions, etc.) find their semantic interpretation in the Maude strategy language, which is an extension of the Maude language. The use of this extension has filled the gap that existed in the basic version of Maude, where the rewriting of an SoS states can be guided in order to accomplish any pair of missions sequentially and even in parallel.

The validation of our theoretical contributions is done on a practical and concrete case study, it consists of applying all the phases of our approach (specification, execution of the model and its analysis) to conceive and prototype the dynamic architecture of a Crisis Response SoS (CRSoS).

Key Words : Software Architecture, Systems of Systems, Architectural Description Language, Formal Methods, Bigraphical Reactive Systems, Rewriting Logic, Maude, Maude Strategy Language, Dynamic Reconfiguration.

Table des matières

Table des figures	xvi
Liste des tableaux	xix
Liste des acronymes	xxi
1 Introduction générale	1
1.1 Contexte et problématique	1
1.2 Objectifs et contributions	3
1.3 Organisation du manuscrit	5
1.4 Diffusion scientifique	5
2 Systèmes de systèmes (SoS) : état de l'art	8
2.1 Introduction	8
2.2 Définitions et caractéristiques	8
2.3 Types d'un SoS et domaines d'application	11
2.4 Ingénierie d'un SoS (SoSE)	13
2.5 Architecture d'un SoS	16
2.6 Conclusion	17
3 Fondements Préliminaires	19
3.1 Introduction	19
3.2 Langages de description d'architecture	20
3.3 Systèmes Réactifs Bigraphiques (BRS)	21
3.3.1 Anatomie et définition des Bigraphes	21
3.3.2 Langage à termes algébriques	24
3.3.3 Dynamique des Bigraphes	26
3.3.4 Extension des Bigraphes	27
3.3.5 Outils pratiques	28
3.4 Langage Maude	29
3.4.1 Syntaxe et notations	30
3.4.2 Langage de Stratégies Maude	32
3.4.3 Vérification formelle et model-checking	37
3.5 Conclusion	40
4 Modélisation et Analyse d'un SoS	42
4.1 Introduction	42
4.2 Travaux existants	43
4.2.1 Approches semi-formelles	43
4.2.2 Approches formelles	44
4.2.3 Synthèse	49
4.3 Principe de la solution basée ArchSoS	51
4.3.1 Syntaxe concrète d'ArchSoS	51

4.3.2	Syntaxe abstraite d'ArchSoS	52
4.3.3	Implémentation et vérification	52
4.4	Conclusion	53
5	Syntaxes d'ArchSoS	55
5.1	Introduction	55
5.2	Concepts de base	56
5.3	Syntaxe concrète	59
5.4	Syntaxe abstraite basée BRS	64
5.4.1	Aspect structurel	64
5.4.2	Aspect comportemental	67
5.5	Conclusion	71
6	Sémantique Opérationnelle d'ArchSoS basée Maude	74
6.1	Introduction	74
6.2	Intégration d'une spécification ArchSoS dans Maude	74
6.2.1	Principe	75
6.2.2	Aspects structurels	76
6.2.3	Aspects dynamiques	78
6.3	Réécriture et exécution d'une spécification ArchSoS	81
6.4	Analyse formelle d'un SoS	83
6.5	Conclusion	86
7	Aspects d'implémentation	88
7.1	Introduction	88
7.2	Etude de cas (Crisis Response SoS)	89
7.2.1	Syntaxe concrète de CRSoS	90
7.2.2	Syntaxe abstraite de CRSoS	93
7.2.3	Sémantique opérationnelle basée Maude de CRSoS	95
7.3	Évaluation	99
7.3.1	Simulation et reconfiguration dynamique de CRSoS	99
7.3.2	Vérification qualitative des comportements de CRSoS	102
7.4	Vers un framework pour l'implémentation d'ArchSoS	104
7.4.1	Phase de spécification	106
7.4.2	Phase de simulation :	110
7.4.3	Phase d'analyse	111
7.5	Conclusion	112
8	Conclusion générale	114
	Bibliographie	118

Table des figures

2.1	Illustration des systèmes constituant d'un SoS de commerce électronique [Nielsen et al., 2015]	14
2.2	Principales activités dans la SoSE	15
3.1	Anatomie des bigraphes	21
3.2	Exemple d'un graphe de place	22
3.3	Exemple d'un graphe de liens	23
3.4	Bigraphe avec des ports explicites	26
3.5	Exemple d'une règle de réaction	27
3.6	Exemple d'un bigraphe dirigé	28
3.7	Module Système BLACKBOARD [Eker et al., 2007]	32
3.8	Module fonctionnel EXT-BLACKBOARD [Eker et al., 2007]	32
3.9	Exemple d'un module de stratégies	33
3.10	Exemple d'une stratégie	34
3.11	Module de stratégies BLACKBOARD-STRAT [Eker et al., 2007]	36
3.12	Résultats d'exécution des stratégies du BLACKBOARD [Eker et al., 2007]	37
3.13	Modules de l'exemple des philosophes	39
3.14	Vérification de l'exemple des philosophes avec le model-Checker de Maude	40
3.15	Vérification des philosophes avec le model-Checker du langage des stratégies Maude	40
4.1	Modèle SoS à base de contrats [Bryans et al., 2014]	43
4.2	Classe système qui définit un constituant d'un SoS [Nielsen and Larsen, 2012]	44
4.3	Composition Horizontale dans [Stary and Wachholder, 2016]	45
4.4	Composition Verticale dans [Stary and Wachholder, 2016]	45
4.5	Architecture de l'outil BiGMTE [Gassara et al., 2019]	46
4.6	Modèle CPN pour l'analyse d'un SoS de surveillance des inondations [Akhtar and Khan, 2019].	47
4.7	Syntaxe abstraite de SoSADL. [Oquendo, 2016a]	48
4.8	Mapping entre SoSADL et DEVS [Neto, 2016]	48
4.9	Vue globale de notre approche	51
5.1	Architecture MDA [Blanc and Salvatori, 2011]	56
5.2	Syntaxe du méta-modèle d'un diagramme de classe [Blanc and Salvatori, 2011]	57
5.3	Méta-modèle issu de la norme ISO/IEC/IEEE 42010 [ISO/IEC/IEEE, 2011]	58
5.4	Modèle conceptuel des éléments AD et des correspondances [ISO/IEC/IEEE, 2011]	59
5.5	Syntaxe concrète d'ArchSoS	60
5.6	Méta-modèle d'ArchSoS	61
5.7	Méta-modèle normalisé d'ArchSoS	63
5.8	Description graphique d'un SoS dans ArchSoS	66
5.9	Action d'acquisition de rôle dans ArchSoS	70
6.1	Du modèle BRS vers les modules Maude	76

7.1	Vue d'ensemble des constituants de CRSoS	89
7.2	Déclaration de CRSoS dans ArchSoS	91
7.3	CRSoS selon la syntaxe abstraite d'ArchSoS : vue graphique	93
7.4	CRSoS selon la syntaxe abstraite d'ArchSoS : vue algébrique	94
7.5	CRSoS selon la syntaxe abstraite d'ArchSoS : l'état final du scénario 1	95
7.6	Opérations d'identifications des éléments de CRSoS	96
7.7	État initial de CRSoS avec tous les événements des deux scénarios	97
7.8	Exemple de règle de réécriture conditionnée de CRSoS : la règle <i>LinkHurricaneAppearance</i>	98
7.9	Stratégies d'évolution de CRSoS	99
7.10	Simulation de CRSoS (mission <i>FireDistinguish</i> avant <i>HurricaneEvacuation</i> dans la séquence d'exécution)	100
7.11	Simulation de CRSoS (mission <i>HurricaneEvacuation</i> avant <i>FireDistinguish</i> dans la séquence d'exécution)	100
7.12	Simulation de CRSoS avec les stratégies Maude	101
7.13	Vérification des propriétés de <i>sureté</i> et de <i>vivacité</i>	102
7.14	Prédicat pour la propriété <i>consistency</i>	103
7.15	Vérification de la mission <i>FireDistinguish</i> dans Maude	103
7.16	Vérification de la consistance de la mission <i>FireDistinguish</i> avec les stratégies Maude	104
7.17	Vérification de la consistance de la mission <i>HurricaneEvacuation</i> avec les stratégies Maude	104
7.18	Vue globale d'ArchSoSTool	105
7.19	Spécification de CRSoS dans ArchSoSTool	106
7.20	Exemple de modélisation dans ArchSoSTool	107
7.21	Création de noeuds de CRSoS dans ArchSoSTool	107
7.22	Modèle Bigraphique (état initial) de CRSoS avec ArchSoSTool	108
7.23	Saisie d'une règle de réaction " <i>LinkFireAppearance</i> " de CRSoS sous ArchSoSTool	109
7.24	Simulation de CRSoS avec la stratégie <i>CrisisDetectionS</i> dans ArchSoSTool	110
7.25	Vérification de la propriété de vivacité de CRSoS dans ArchSoSTool	111

Liste des tableaux

2.1 Définitions d'un SoS	9
2.2 Différences entre un SoS et un système individuel [INCOSE, 2018]	12
2.3 Domaines d'applications des SoS	12
2.4 Exemple de types des SoS [Kinder et al., 2015]	13
2.5 Principales Différences entre SE et SoSE	15
3.1 Langage à termes algébriques	24
3.2 Outils autour des BRS	29
3.3 Les éléments clés dans un module de stratégies	33
3.4 Opérateurs de stratégies	35
3.5 Tableau des symboles et opérateurs LTL	37
4.1 Travaux existants pour la modélisation des SoS	50
5.1 Conditions de règles de formation pour les bigraphes des SoS	66
5.2 Description graphique et algébrique de la syntaxe d'ArchSoS	68
5.3 Description des actions d'évolution dans ArchSoS	69
5.4 Prédicat de contrôle d'évolution des SoS dans ArchSoS	71
6.1 Correspondance table between ArchSoS aspects and Maude.	77
6.2 Prédicats de contrôles définis sous Maude	79
6.3 Implementation des actions d'évolution	82
7.1 Contraintes comportementales de CRSoS	90
7.2 Scénarios d'exécution de CRSoS	92

Liste des acronymes

ADL	Architecture Description Language
BP	Behaviours Protocoles
BPL	Bigraphical Programming Language
BRS	Bigraphic Reactive Systems
BSL	Bigraph Specification Language
BTL	Bounded Temporal Logic
BiGMTE	Bigraph Matching and Transformation Engine
BigMC	Bigraphical Model Checker
BigraphER	Bigraph Evaluator and Rewriting
CCP	Concurrent Constraint Paradigm
CML	COMPASS modelling language
CPN	Colored Petri Nets
CRSoS	Crisis Response System of Systems
CSP	Constraint Satisfaction Problem
CWM	Common Warehouse Meta model
DEVS	Discrete Event System Specification
GL	Génie Logiciel
GLSD	Génie Logiciel et Systèmes Distribués
GMTE	Graph Matching and Transformation Engine
GQM	Goal-Question-Metric
IDE	Integrated Development Environment
IIoT	Industrial Internet of Things
LTL	Logique Temporelle Linéaire
LTS	Labelled Transition System
MDA	Model Driven Architecture
MOF	Meta Object Facility
OMG	Object Management Group
OPM	Object Process Methodology
SE	Systems Engineering
SOA	Service-Oriented Architecture
SoS	System of Systems
SoSE	Systems of Systems Engineering
TIC	Technologies de l'Information et de la Communi- cation
UML	Unified Modeling Language
VDM-RT	Vienna Development Method Real-Time
VeMo	Vehicle Monitoring
XML	eXtensible Markup Language

Introduction générale

Sommaire

1.1	Contexte et problématique	1
1.2	Objectifs et contributions	3
1.3	Organisation du manuscrit	5
1.4	Diffusion scientifique	5

1.1 Contexte et problématique

De nos jours, les systèmes d'ingénierie sont de plus en plus complexes, par leur nature multi-physique que par leur opérabilité dans des environnements variables et d'autant plus complexes. L'observation de Beer [Stafford, 1979] sur les systèmes complexes tient toujours : Ces systèmes sont un puzzle de milliers de diamètre et nous essayons de voir comment l'assembler. Depuis, une croissance explosive sans précédent a eu lieu dans la technologie et les systèmes produits par cette technologie, l'accès à l'information a augmenté de façon exponentielle. Cependant, nous semblons non plus capables de résoudre le puzzle de Beer. En fait, nous avons maintenant un problème centré sur l'intégration de plusieurs puzzles qui changent au fur et à mesure que nous essayons d'assembler les pièces [Keating et al., 2003].

L'augmentation de la complexité des systèmes logiciels est suite à l'évolution des exigences et des besoins de notre société mondiale actuelle, qui nécessite une richesse plus grande en termes de fonctionnalités, et ceci dans plusieurs domaines d'un large éventail, tels que la défense, le transport dans toutes ses formes, la santé, le commerce, la gestion et la réponse aux crises, les réseaux énergétiques, etc. Cette complexité a engendré une extension des réseaux de communications, ainsi qu'une hétérogénéité entre les systèmes existants, d'un point de vue matériel et logiciel. Ces systèmes sont devenus des entités totalement indépendantes à grande échelle, qui opèrent individuellement les unes des autres. Avec la croissance de nouvelles exigences dans le marché des technologies de l'information et de la communication, le besoin d'interaction et de coopération entre ces systèmes est apparu. Par conséquent, de nouvelles fonctionnalités plus complexes ont émergé pour poser des défis supplémentaires. Ainsi, le concept de système de systèmes (ou SoS pour "Systems of systems") s'est imposé durant ces dernières décennies pour remédier à ces défis et maîtriser ces fonctionnalités.

En fait, les systèmes de systèmes viennent à l'origine des secteurs spatial et militaire mais ils sont de plus en plus utilisés et applicables à d'autres secteurs comme le transport, les réseaux de télécommunications, les machines spéciales, la santé, l'énergie, etc. La maîtrise des systèmes complexes devient critique à mesure que leur application s'élargit au sein du secteur industriel.

Un SoS est un ensemble de systèmes autonomes interconnectés et coordonnés pour satisfaire une capacité où des fonctions spécifiques que les systèmes indépendamment ne pourraient réaliser. Un SoS peut avoir plusieurs niveaux hiérarchiques, car les systèmes qui le constituent peuvent eux même être des SoS.

L'un des principaux défis à relever de la complexité des SoS vient de l'imprévisibilité du comportement des constituants d'un SoS ; Ils coopèrent pour achever des missions suite à des événements environnementaux qui affectent le SoS qui les contient. Cette imprévisibilité résulte de l'absence de comportement figé au sein d'un SoS, qui est plutôt obtenu dynamiquement et émerge au moment de la coopération entre les constituants. Les comportements d'un SoS doivent être cohérents, par exemple il faut préserver la sécurité de certaines fonctionnalités des systèmes constituants qui ne doivent pas être combinés ou exposés, ou bien il ne faut pas avoir des liens particuliers entre certains systèmes, pour des raisons d'autorités ou de législation. Des contraintes doivent être établies et respectées afin de garantir cette cohérence. En plus, les comportements des SoS ne doivent pas influencer l'un et l'autre, s'ils occurrent en parallèle. Ils doivent être dynamiquement gérés indépendamment dans le cas où plusieurs événements les affectant simultanément.

- L'ingénierie de systèmes englobe tout le cycle de vie du système et il faut donc que le logiciel soit aussi capable d'apporter cette vision d'ensemble.
- Suivi des exigences fonctionnelles pendant tout le cycle de vie.
- Processus d'intégration, de qualification, de vérification et de validation.
- Gestion de données techniques associée à une gestion de configuration.
- Processus de modifications pour répondre à l'évolution des systèmes.

Les théoriciens et les praticiens de l'ingénierie des SoS sont conscients du fait que l'utilisation des modèles représente toujours le moyen idéal pour comprendre, analyser et appliquer les critères ci-dessus. Nous adhérons à cette idée dans le contexte de ce travail de thèse et nous nous intéresserons à l'élaboration de modèles pour décrire les SoS de manière abstraite afin de réduire leur complexité. Nous nous acharnons à la spécification rigoureuse de leurs aspects architecturaux et d'évolution. En effet, tout logiciel, au-delà d'un niveau minimal de complexité, est un édifice qui mérite une phase de réflexion initiale pour l'imaginer dans ses grandes lignes. Cette phase correspond à l'activité d'architecture qui consiste notamment à identifier les différents éléments qui vont composer ce logiciel et à organiser les interactions entre ces éléments.

Dans le cas des SoS l'activité d'architecture est un travail de longue haleine au cours duquel, on effectue les grands choix les structurant : langages, technologies, méthodologies mises en œuvre, etc. Nous pouvons donc décrire l'architecture d'un SoS, comme tout logiciel, selon différents points de vue. Entre autres, une vue logique qui mettra l'accent sur les fonctionnalités et les responsabilités de chaque composant du SoS. Une vue physique pour présenter les processus, les machines et les liens réseau nécessaires à la coopération des composants d'un SoS ainsi que leur localisation.

Dans la littérature, plusieurs travaux de recherche se sont intéressés à la modélisation des SoS en passant par leurs architectures logicielles. En particulier, les langages de description d'architecture (ADL : Architecture Description Languages) peuvent jouer un rôle primordial dans la spécification formelle des composants interagissant dans un SoS. Une telle description doit permettre d'analyser, de raisonner et de valider a priori un SoS.

Un ADL défini pour les SoS doit pouvoir décrire leur architecture logicielle sans se limiter uniquement à la syntaxe des SoS, mais aussi à leur aspect sémantique et évolutif. L'établissement des liens ainsi que la combinaison de fonctionnalités des composants d'un SoS pour réaliser leurs missions doivent se faire d'une façon dynamique. Ainsi, un ADL basé sur des normes et

des standards communs doit bénéficier de moyens formels pour prendre en charge les deux aspects dans les SoS. En particulier, il doit répondre aux défis structureaux ou comportementaux suivants :

- Hiérarchie : Un SoS peut contenir plusieurs systèmes-constituants, qui peuvent être eux même des SoS, donc il y'a un besoin de décrire une structure cohérente qui permet de représenter, d'une façon hiérarchique, ces systèmes.
- Liens : Un SoS est considéré comme un ensemble de systèmes en interactions, il est vital de pouvoir spécifier l'aspect relationnel entre chaque système, en termes de liens dynamiques et de coopération entre eux.
- Évolution : Un SoS regroupe un ensemble de systèmes indépendants qui peuvent évoluer, communiquer entre eux et acquérir de nouvelles fonctionnalités, donc une spécification flexible doit être représentée et capturée pour pouvoir répondre aux besoins évolutifs d'un SoS.
- Missions : Le principe d'un SoS, c'est d'avoir de nouvelles fonctionnalités qui représentent son but global, donc il est nécessaire de trouver un moyen de faire émerger ces fonctionnalités à partir des systèmes constituants pour satisfaire ce but.
- Contraintes fonctionnelles : Comme les systèmes constituants sont en évolution et en interactions constantes, il faut adopter un mécanisme qui permet d'avoir un contrôle sur cette évolution et ces interactions afin d'éviter les comportements parfois indésirables des SoS.
- Exécution et Vérification : Une approche qui vise à modéliser et décrire les SoS doit aboutir par un outil ou un framework qui permet de mettre en marche les spécifications définies par cette approche. Cela servira à simuler les comportements des SoS afin de vérifier leur cohérence.
- Reconfiguration Dynamique : Étant donné que la mission d'un SoS est imprévisible, elle change constamment et dépend toujours d'un évènement externe qui provoque l'évolution des constituants d'un SoS. Afin de prendre en considération ce changement qui affecte le SoS, ainsi que ses constituants, le formalisme proposé doit offrir un moyen pour décrire la reconfiguration de sa structure et de son comportement dynamique.

1.2 Objectifs et contributions

Afin de faire face à cette complexité des SoS, plusieurs ADL ont été proposés pour décrire leurs architectures et fournir des modèles d'abstraction appropriés. Cependant, même avec cette intention, les résultats restent insuffisants et ne peuvent pas couvrir tous les aspects structureaux et comportementaux. Nos travaux de recherche se situent dans ce contexte et traitent les architectures logicielles pour les SoS. Nous nous concentrons, dans le cadre de notre thèse à la définition d'un ADL dédié à la modélisation et l'analyse des SoS.

Nous traitons essentiellement le problème de l'évolution des exigences dû à un changement imprévisible dans les fonctionnalités d'un SoS modélisé. Nous associons à un SoS des descriptions architecturales dynamiques répondant à cette situation. Ainsi, nous proposons des solutions de modélisation permettant de considérer tous les éléments architecturaux des SoS, de capturer leur nature évolutive et imprévisible, et s'assurer du bon fonctionnement de leurs comportements en évitant des situations indésirables.

L'ADL que nous définissons dans ce manuscrit, baptisé ArchSoS, couvre plusieurs maillons de la chaîne de description architecturale. Il offre trois approches permettant de décrire la structure hiérarchique d'un SoS, l'évolution de son comportement en respectant un ensemble de

contraintes fonctionnelles et les stratégies de reconfiguration pour adapter ses missions. De manière plus détaillée, nous mettons en évidence dans cette section les contributions scientifiques autour de la conception d'ArchSoS et l'exécution et l'analyse formelles des descriptions des SoS basées ArchSoS.

- **Contribution 1 : Syntaxe Concrète pour ArchSoS** : Nous nous inspirons d'une architecture de référence pour les SoS afin d'élaborer un modèle conceptuel pour ArchSoS. Le méta-modèle décrivant l'architecture d'un SoS est confronté par la suite à la norme ISO/IEC/IEEE 42010 [ISO/IEC/IEEE, 2011] pour indiquer la nécessité de prévoir l'ensemble des éléments importants dans la description d'un tel système. Ainsi, nous dégagons dans cette phase deux vues pour la syntaxe concrète d'ArchSoS, une vue textuelle inspirée de la syntaxe des ADL et une autre graphique qui hérite ses formes des modèles de l'approche MDA.
- **Contribution 2 : Syntaxe Abstraite d'ArchSoS** : La syntaxe abstraite d'ArchSoS se base sur le formalisme des Systèmes Bigraphiques Réactifs (BRS). Dans un premier temps, nous associons à une description d'architecture d'un SoS un bigraphe, permettant de modéliser les aspects structurels des SoS, tels que leurs constituants, leurs fonctionnalités, les liens entre eux et les événements pouvant les affecter pour accomplir des missions particulières. Les notions de localité et de mobilité des bigraphes offrent une modélisation adéquate des SoS. Ainsi, nous héritons du formalisme des BRS pour décrire les modèles SoS résultants selon deux vues : une vue graphique visuelle et une vue algébrique textuelle. Dans un deuxième temps, les architectures SoS définies sous forme de bigraphes peuvent évoluer dynamiquement grâce à un ensemble de règles de réaction. Nous adaptons alors ce type de règles pour décrire la sémantique d'évolution des SoS dans ArchSoS.
- **Contribution 3 : Implémentation, Reconfiguration et Vérification des SoS** : Dans cette contribution, nous proposons des solutions d'implémentation et d'exécution pour les modèles bigraphiques des SoS élaborés. Nous procédons à l'intégration de tous les éléments structurels et comportementaux d'ArchSoS dans le langage Maude. Nous associons ainsi une sémantique opérationnelle à cet ADL. Cette contribution est effectuée selon trois étapes essentielles :
 - Encodage et implémentation d'ArchSoS sous Maude : une description dans ArchSoS, modélisée sous forme de bigraphes est transcrite dans Maude : Un module fonctionnel spécifie la structure d'un SoS donné, tandis qu'un module système représente son évolution en décrivant ses actions comme des règles de réécritures, accompagnées de leurs prédicats de contrôle. Cette spécification permet d'exécuter le comportement d'un SoS sous Maude.
 - Reconfiguration dynamique des comportements des SoS dans ArchSoS : Bien que Maude permet de spécifier des règles de réécritures qui décrivent l'évolution du comportement des SoS, l'application de ses règles reste indéterministe, il n'existe par un mécanisme qui permet de contrôler la réécriture des états correspondants, et d'indiquer quelle est la règle à appliquer et à quel moment, surtout dans le cas où des règles concurrentes peuvent s'appliquer et l'application de l'une affecte l'autre. Pour remédier à cette limite, nous utilisons une extension de Maude, qui est le langage de stratégies Maude. Les stratégies définies permettent de contrôler et guider l'évolution des comportements des composants d'un SoS et leur reconfiguration pour réaliser les missions envisageables tout en respectant un certain nombre de contraintes fonctionnelles.
 - Vérification formelle du comportement des SoS : Enfin, cette partie consiste à vérifier l'évolution souhaitée du comportement d'un SoS. Un ensemble de propriétés génériques (vivacité, sûreté et consistance) est exprimé en logique linéaire temporelle

(LTL) et vérifié en utilisant l'outil model-checker de Maude. D'autres propriétés relatives au domaine des SoS modélisés sont aussi étudiées dans cette phase de notre étude.

1.3 Organisation du manuscrit

En plus d'une introduction générale et d'une conclusion, ce document est structuré en six autres chapitres :

- **Le chapitre 2** (*Systèmes de Systèmes (SoS) : état de l'art*), consacré à la présentation du contexte de notre sujet de thèse, à travers des aspects importants et en s'aidant des définitions les plus adoptées des SoS. Il détaille aussi les différentes phases de l'ingénierie des SoS et de leur architecture.
- **Le chapitre 3** (*Fondements Préliminaires*), rappelle les différents concepts et fondements nécessaires à la bonne compréhension des contributions proposées dans la suite de la thèse. Il présente les langages de description d'architecture, les Systèmes Bigraphiques Réactifs et le langage Maude à travers son extension pour la définition des stratégies.
- **Le chapitre 4** (*Modélisation et Analyse d'un SoS*), décrit d'une façon détaillée plusieurs travaux de la littérature liés à la modélisation des SoS, afin de les comparer et de tirer une synthèse qui sert à mieux situer nos travaux de recherche. Il introduit ensuite le principe de notre approche et ses différentes phases.
- **Le chapitre 5** (*Syntaxes d'ArchSoS*), introduit les deux premières contributions de cette thèse : (1) Une syntaxe concrète pour exprimer les architectures logicielles d'un SoS dans ArchSoS, basée sur une architecture de référence pour les SoS et la norme ISO/IEC/IEEE 42010. (2) Une syntaxe abstraite à base des BRS pour le langage ArchSoS ayant deux vues (graphique visuelle et algébrique textuelle).
- **Le chapitre 6** (*Sémantique Opérationnelle basée Maude de ArchSoS*), décrit l'approche empruntée pour associer une sémantique opérationnelle à ArchSoS afin de pouvoir exécuter et analyser les modèles des SoS établis. La structure et les comportements dynamiques et émergents des SoS sont modélisés à l'aide de modules Maude. Nous prévoyons des modules fonctionnels, systèmes et de stratégies pour prendre en charge tous les aspects spécifiés dans ArchSoS.
- **Le chapitre 7** (*Aspects d'implémentation*), présente une validation de notre approche selon deux aspects : Le premier permet d'illustrer toutes les étapes de la solution proposée pour la conception architecturale des SoS, à travers une étude de cas réel et de grande importance, Il s'agit d'un SoS de réponses aux crises (CRSoS). Deux scénarios d'évolution de ce SoS sont étudiés, simulés et vérifiés. Le deuxième consiste à l'élaboration d'un prototype du Framework d'implémentation d'ArchSoS.

1.4 Diffusion scientifique

Nous listons ci-dessous les publications dont les travaux de ce manuscrit ont fait objet :

Revue scientifique avec comité de lecture

- Seghiri, A., Belala, F. & Hameurlain, N. (2022). A Formal Language for Modelling and Verifying Systems-of-Systems Software Architectures . *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, Volume 12 Issue 1.

Conférences internationales avec comité de lecture

- Seghiri, A., Belala, F., & Teniou, A. A. (2018, April). Towards software architectural description for systems of systems : Case of a maritime transport SoS. *In Proceedings of the 2018 International Conference on Internet and e-Business* (pp. 183-187).
- Seghiri, A., Belala, F., Benzadri, Z., & Hameurlain, N. (2018, June). A maude based specification for sos architecture. *In 2018 13th Annual Conference on System of Systems Engineering (SoSE)* (pp. 45-52). IEEE.

Systemes de systemes (SoS) : etat de l'art

Sommaire

2.1	Introduction	8
2.2	Définitions et caractéristiques	8
2.3	Types d'un SoS et domaines d'application	11
2.4	Ingénierie d'un SoS (SoSE)	13
2.5	Architecture d'un SoS	16
2.6	Conclusion	17

2.1 Introduction

Les systèmes informatiques ne cessent d'évoluer, en termes de taille, de capacités, de complexité et de nombre de communications entre eux. Les applications qui émergent récemment sont de plus en plus orientées vers la distribution, bien qu'elles peuvent être développées selon des spécifications différentes, et être complètement hétérogènes en termes de matériels et logiciels.

Cette évolution progressive a donné une nécessité d'interconnecter ces systèmes, afin de les faire collaborer pour réaliser des tâches communes. Elle engendre un besoin d'améliorer la vision classique de l'ingénierie de systèmes complexes vers un paradigme basé « System of systems » (ou SoS).

Les SoS apportent de nouvelles caractéristiques qui permettent une communication simple et flexible entre plusieurs systèmes existants. Bien que ces systèmes puissent être différents et fonctionnent de façon autonome, leurs interactions présentent et fournissent généralement une nature évolutive que les développeurs de tels systèmes doivent reconnaître, comprendre et analyser.

Dans ce chapitre, nous présentons un état de l'art sur les SoS, contenant un certain nombre de définitions et les caractéristiques qui les diffèrent des autres systèmes dans la section, ainsi que les différents types des SoS tout en évoquant leurs domaines d'applications. Nous donnons un intérêt particulier à la présentation de l'ingénierie des SoS tout en notant la différence par rapport à l'ingénierie des systèmes monolithique. Enfin, nous décrivons la notion d'architecture des SoS avant de conclure le chapitre.

2.2 Définitions et caractéristiques

Un SoS est un concept qui a émergé dans les dernières décennies, décrivant un ensemble de systèmes coopérant pour réaliser une tâche commune. Il existe plusieurs définitions décrivant

la terminologie des SoS, néanmoins il est à noter qu'il y a une grosse divergence concernant ces définitions, et il n'existe pas un consensus sur une définition universellement reconnue et unifiée qui décrit ces systèmes. Nous listons dans le tableau 2.1, de façon chronologique, un ensemble de définitions qui regroupent les points de vues concernant un SoS :

TABLE 2.1: Définitions d'un SoS

Auteurs	Définition Originale	Définition en Français
[Maier, 1998]	SoS is a class of systems which are built from components that are large scale systems in their own right.	SoS est une classe de systèmes qui sont construits à partir de composants qui sont des systèmes à grande échelle à part entière.
[Popper et al., 2004]	An SoS is a set of dedicated, or task-oriented systems that pool their resources and capabilities to form a new, more complex "meta-system" that offers more functionality and performance than the mere sum of the constituent systems.	Un SoS est un ensemble de systèmes dédiés, ou orientés tâches qui mettent en commun leurs ressources et capacités pour obtenir un nouveau "méta-système" plus complexe qui offre plus de fonctionnalités et de performances que la simple somme des systèmes constituants.
[Boardman and Sauser, 2006]	An SoS consists of parts, relationships and a whole that is greater than the sum of the parts, but with the distinction coming from the manner in which parts and relationships are gathered together and therefore in the nature of the emergent whole.	Un SoS se compose de parties, de relations et un tout supérieur à la somme des parties, avec la distinction venant de la manière dont les parties et les relations sont rassemblées et la nature de l'ensemble émergent.
[Jamshidi and Sage, 2009]	SoS are large-scale integrated systems which are heterogeneous and independently operable on their own, but are networked together for a common goal.	Les SoS sont des systèmes intégrés à grande échelle qui sont hétérogènes et opèrent indépendamment l'un de l'autre, mais en même temps ils sont interconnectés pour un objectif commun.
[DAU, 2010]	System of interest whose elements are operationally and managerially independent. This set of interoperable systems and / or produce results unachievable by individual systems.	Système d'intérêt dont les éléments sont indépendants sur le plan opérationnel et managérial. Cet ensemble de systèmes interoperables et / ou produisent des résultats irréalisables par les systèmes individuels.

[Shortell, 2015]	System-of-interest whose elements are managerially and/or operationally independent systems. These interoperating and/or integrated collections of systems produce results unachievable by the individual systems alone.	Système d'intérêt dont les éléments sont gérés indépendamment et opèrent d'une façon indépendante. Ces collections interoperables et/ou intégrées de systèmes produisent des résultats irréalisables par les seuls systèmes individuels.
[ISO, 2015]	SoS brings together a set of systems for a task that none of the systems can accomplish on its own. Each Constituent system keeps its own management, goals, and resources while coordinating within the SoS and adapting to meet SoS goals.	Un SoS rassemble un ensemble de systèmes pour une tâche qu'aucun des systèmes qui le constitue peut accomplir à lui seul. Chaque constituant garde sa propre gestion, ses objectifs et ses ressources tout en coordonnant au sein du SoS et en s'adaptant pour atteindre les objectifs du SoS.
[Brook, 2016]	An SoS is a system that results from the coupling of a number of systems at a given point in their life cycle.	Un SoS est un système qui résulte du couplage d'un certain nombre de systèmes à un moment donné de leur cycle de vie.
[ISO, 2019]	Set of systems or system elements that interact to provide a unique capability that none of the constituent systems can accomplish on its own.	Ensemble de systèmes ou d'éléments de système qui interagissent pour fournir une capacité unique qu'aucun des systèmes constituants ne peut accomplir seul.

Toutes ces définitions indiquent que le but d'un SoS est d'accomplir des missions ou des buts globaux, qui sont le résultat d'une collaboration entre ses systèmes constituants. La particularité de ces systèmes réside dans le fait qu'ils sont capables d'opérer sur deux niveaux, le premier est indépendamment l'un de l'autre, et le deuxième au sein du SoS qui les contient.

Le manque de consensus dans la définition d'un SoS a incité les chercheurs à orienter leurs définitions vers les caractéristiques des SoS, qui ont été largement reconnues et établies initialement par [Maier, 1998]. Nous notons cinq principales caractéristiques :

- (a) **Indépendance opérationnelle** : Chaque système constituant un SoS est une entité indépendante en soi ; s'il est détaché du SoS, il doit pouvoir continuer à être opérable tout seul.
- (b) **Indépendance managériale** : Chaque système dans un SoS peut évoluer de manière indépendante sans tenir compte des autres constituants qui font partie du SoS.
- (c) **Distribution géographique** : Les systèmes constituant un SoS sont dispersés sur une zone géographique relativement large en fonction de la taille du SoS, limitant les interactions en échange d'information seulement.
- (d) **Comportement émergent** : Les systèmes constituant un SoS coopèrent efficacement pour réaliser des tâches, qu'aucun d'entre eux ne peut accomplir à lui seul. En d'autres termes, la mission globale de SoS émerge uniquement avec le comportement coopératif de ses constituants.
- (e) **Développement évolutif** : Un SoS est toujours en évolution, il n'est jamais complètement fini ou formé, les fonctionnalités des systèmes constituants changent en permanence,

et de nouveaux constituants peuvent être rajoutés ou supprimés.

Parmi ces caractéristiques, l'indépendance opérationnelle et l'indépendance managériale sont primordiaux et distinguent le comportement d'un SoS d'un système classique. Un système qui ne contient pas ces deux caractéristiques n'est pas considéré comme un SoS. Cela a été expliqué par [Rebovich et al., 2009] comme l'aspect fondamental pour un SoS : « du point de vue des systèmes individuels, leur participation aux capacités du SoS représente des obligations, des contraintes et des complexités supplémentaires. La participation à un SoS est rarement perçue comme un gain net du point de vue des acteurs du système individuel ».

Prenons le cas d'un système d'un véhicule routier, où chaque partie, comme le moteur à titre d'exemple, peut être considéré comme un système qui co-existe avec d'autres systèmes, formant un regroupement plus étendu afin d'achever un but commun, qui est le transport de manière générale. Un cas similaire est celui d'un ordinateur, qui est composé d'un processeur, d'une unité centrale, d'un écran, etc. Où chaque périphérique est considéré comme un système. Ces composants collaborent pour former un ordinateur qui aura des fonctionnalités plus grandes que les fonctionnalités de chaque composant individuel.

Cependant, dans les deux cas de figure, si on détache les composants de l'ordinateur ou du véhicule, et on les prends individuellement, ils ne peuvent pas opérer ou gérer leurs ressources en dehors de leurs regroupements. Donc, même s'ils collaborent pour atteindre un but global, ils ne peuvent pas être considéré comme des SoS, car ils ne possèdent pas les deux caractéristiques principales des SoS, qui sont l'indépendance opérationnelle et l'indépendance managériale.

D'autres chercheurs dont [Boardman and Sauser, 2006] ont regroupé et décrit les caractéristiques des SoS qui les distinguent des autres systèmes différemment, en portant une attention particulière à la refonte de nouveaux constituants d'un SoS, avec ses propres systèmes existants pour former le SoS. Ils identifient cinq caractéristiques principales connues par l'acronyme "ABCDE" des SoS :

- (A) **Autonomie (Autonomy)** : Chaque système est libre, indépendant, et il a son propre objectif de fonctionnement.
- (B) **Appartenance (Belonging)** : Les systèmes fonctionnent en collaboration pour atteindre un but commun qu'aucun d'eux peut accomplir.
- (C) **Connectivité (Connectivity)** : La coopération est rendue possible par un réseau distribué de constituants dynamiques.
- (D) **Diversité (Diversity)** : Les constituants sont des systèmes hétérogènes auto-suffisants capables d'être améliorés par l'évolution et l'adaptation.
- (E) **Émergence (Emerging)** : Les actions et interactions entre les constituants d'un SoS donnent lieu à de nouveaux comportements attribués au SoS dans son ensemble.

En plus des caractéristiques citées ci-dessus, les SoS possèdent un nombre de différences avec des systèmes individuels ou atomiques. Le tableau 2.2 montre les principales différences qui séparent un SoS des autres systèmes classiques.

2.3 Types d'un SoS et domaines d'application

Les domaines d'application des SoS sont larges et s'étendent à presque tous les domaines de la vie courante. A l'origine, les SoS ont été identifiés dans l'environnement de la défense, ils ont ensuite atteint pratiquement tous les secteurs tels qu'ils sont résumés dans le tableau 2.3. Les premiers travaux dans le secteur de la défense ont fourni la base initiale des SoS, y compris ses fondements intellectuels, ses approches techniques et son expérience pratique. Un peut plus

TABLE 2.2: Différences entre un SoS et un système individuel [INCOSE, 2018]

Les systèmes ont tendance à...	Les SoS ont tendance à...
Avoir un ensemble clair de parties prenantes.	Avoir plusieurs niveaux de parties prenantes avec des intérêts mixtes et éventuellement concurrents.
Avoir des objectifs et un but clair.	Avoir des objectifs multiples, voire contradictoires.
Avoir des priorités opérationnelles claires.	Avoir des priorités opérationnelles multiples, et parfois différentes.
Avoir un seul cycle de vie.	Avoir plusieurs cycles de vie avec des éléments implémentés de manière asynchrone.
Avoir un propriétaire claire avec la possibilité de déplacer les ressources entre les éléments appartenant à ce propriétaire.	Avoir plusieurs propriétaires prenant des décisions de ressources indépendantes.

tard, les concepts et principes des SoS s'appliquent à d'autres domaines gouvernementaux, civils et commerciaux.

TABLE 2.3: Domaines d'applications des SoS

Domaine	Exemple
Santé	Gestion de la santé personnelle, Réseau des hôpitaux, Services d'urgences
Energie	Réseau électrique intelligent, Maisons intelligentes et villes intelligentes
Environnement	Gestion d'eau, Gestion de ressources forestières et récréatives
Transport	Gestion de la circulation aérienne, Transport terrestre intégré, Gestion spatiale
Militaire	Défense antimissile, Capteurs en réseau
Réponses aux crises	SoS de réponses aux catastrophes, y compris les incendies de forêt, les inondations et les attaques terroristes

Les SoS sont munis d'une taxonomie décrivant leurs types [Maier, 1998] [Dahmann and Baldwin, 2008]. Elle est basée sur le niveau d'indépendance des systèmes constituants et offre un cadre de compréhension du SoS basée sur la nature de ses objectifs. Nous notons qu'un SoS reflètera une combinaison de types la plupart du temps, qui peut changer avec le temps suite aux évolutions de ses constituants. Les quatre principaux types des SoS sont cités ci-dessous.

Dirigé : Le SoS est créé et géré pour accomplir des objectifs spécifiques et les systèmes constituants sont subordonnés au SoS. Ces constituants maintiennent la capacité de fonctionner de manière indépendante. Leur gestion est centralisée afin de permettre au SoS de continuer à réaliser ses objectifs au long terme.

Reconnu : Dans ce type, les SoS ont un objectif reconnu et un gestionnaire bien connu ; cependant les systèmes constituants gardent leur nature indépendante, managerielle et évolutive. Les changements dans les systèmes sont basés sur des accords de coopération entre le SoS et le système.

TABLE 2.4: Exemple de types des SoS [Kinder et al., 2015]

Exemple de SoS	Types de SoS
Programme des futurs systèmes de combat [Pernin et al., 2012]	<i>Dirigé</i> : c'était un contrôle centralisé sous forme de programme d'approvisionnement
Programme national pour l'informatique dans le service national de santé [Office, 2011]	<i>Dirigé</i> à l'origine mais s'est ensuite déplacé vers <i>Reconnu</i>
Programme de système intégré en eau profonde [Hutton et al., 2011]	<i>Dirigé</i> : c'était un contrôle centralisé sous forme de programme d'approvisionnement
Système national de gestion des délinquants (C-NOMIS) [Office, 2009]	<i>Dirigé</i> à l'origine mais s'est ensuite déplacé vers <i>Reconnu</i>
Secours aux sinistrés de l'ouragan Katrina [Comfort, 2007]	<i>Collaboratif</i> au départ mais évoluant vers <i>Reconnu</i> quand le le contrôle était établi. Cependant, au sein des agences individuelles, il était de type <i>Reconnu</i> tout au long des opérations d'aide et de récupérations
Grille d'énergie de l'Arizona et de la Californie du Sud [FERC, 2014]	<i>Reconnu</i> auprès des utilisateurs formant des éléments virtuels du SoS
E-commerce (Commerce électronique) [Nielsen et al., 2015]	<i>Virtuel</i> qui représente une place de marché virtuelle, où le traitement des opérations de paiement, la gestion des stocks et les chaînes d'approvisionnement sont effectués

Collaboratif : Les systèmes constituants interagissent et collaborent volontairement pour remplir des objectifs communs, ils décident collectivement comment fournir ou refuser des fonctionnalités centrales selon des standards bien reconnus.

Virtuel : Dans ce type, il n'existe pas d'autorité ou de gestion centralisée des systèmes constituants. Il n'y'a pas d'objectif convenu, donc des comportements à grande échelle, parfois inattendus, peuvent émerger.

Le tableau 2.4 identifie les types possibles des SoS pour un ensemble d'exemples concrets. A titre d'exemple, le SoS d'E-commerce est un SoS virtuel que de nombreuses personnes rencontrent presque quotidiennement, là où nous trouvons l'achat et la vente de biens et de services via des achats en ligne. Le commerce électronique implique un certain nombre de systèmes indépendants différents travaillant ensemble afin d'atteindre l'objectif global de vente et de livraison. La figure 2.1 illustre les constituants de cet SoS. Il contient des systèmes pour gérer les données, des systèmes qui agissent comme centres de distribution, ainsi que des systèmes réservés aux fonctionnalités de l'achat et de la vente, comme les systèmes de fournisseurs, d'expéditeurs et de transactions financières. Tous ces systèmes créent un marché virtuel où peut figurer des comportements à grande échelle.

2.4 Ingénierie d'un SoS (SoSE)

La littérature reconnaît clairement l'importance des SoS aux futurs développeurs de systèmes. Les SoS doivent être conçus de manière à fournir des conceptions, des déploiements, des

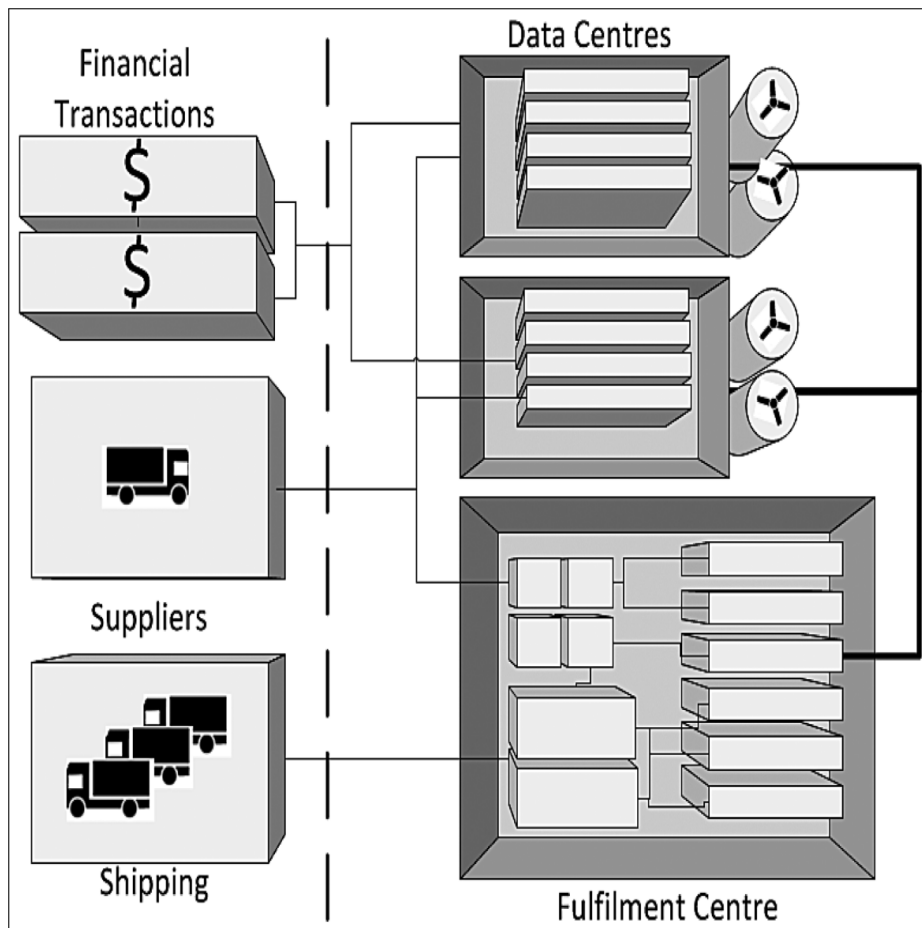


FIGURE 2.1: Illustration des systèmes constituant d'un SoS de commerce électronique [Nielsen et al., 2015]

exploitations et des évaluations rigoureuses.

Traditionnellement, l'ingénierie des systèmes (SE : Systems Engineering) s'est concentrée sur des problèmes de systèmes complexes isolés. Le processus de définition, d'analyse et de développement de ces systèmes vise à répondre à un problème ou à un besoin bien spécifique.

Cependant, l'ingénierie des SoS (SoSE : System of Systems Engineering) se concentre sur l'intégration de plusieurs systèmes complexes dans un seul méta-système. Cette intégration peut impliquer des systèmes existants, des systèmes nouvellement conçus ou une hybridation. L'originalité est que la formation du SoS, composé d'un ensemble de systèmes complexes, est engendrée par un besoin émergent ou une mission particulière. Cela représente une dérivation significative avec la tradition de l'ingénierie des systèmes qui se concentre sur un seul système.

La SoSE a été définie par [Keating et al., 2003] comme :

Définition 2.1 (SoSE). *"The design, deployment, operation, and transformation of independent systems that must function as an integrated complex SoS to produce desirable results. These constituent systems are themselves comprised of multiple autonomous embedded complex systems that can be diverse in technology, context, operation, geography, and conceptual frame."*

Afin de donner un aperçu sur les caractéristiques de la SoSE, nous commençons par la situer par rapport à la SE, en dressant un ensemble de différences entre les deux concepts. Pour distinguer entre l'ingénierie de systèmes SE, et SoSE, nous définissons dans le tableau 2.5 les principales différences entre elles selon plusieurs aspects structurels et comportementales. Bien que SoSE s'écarte de SE de plusieurs manières, Les techniques et les méthodes classiques au génie logiciel prennent toujours une base importante dans le domaine de l'ingénierie d'un SoS.

TABLE 2.5: Principales Différences entre SE et SoSE

Aspect	SE	SoSE
Hierarchie	Système complexe unique	Plusieurs systèmes complexes intégrés
Comportement	Fixe	Émergent
Approche	Processus	Méthodologie
Objectif	Solution a un problème existant	Réponse à plusieurs nouveaux problèmes
Constituants	Fixes	Mobiles
Gouvernance	Un seul propriétaire	Un SoS n'a pas d'autorité sur tous ses constituants

Parmi les atouts de SE qu'un SoS doit bénéficier : (1) Le lien avec la théorie des systèmes et les principes de conception, d'analyse et d'exécution (2) L'accent interdisciplinaire sur la résolution de problèmes et le développement de systèmes. Tirer parti de ces atouts servira à renforcer le développement de SoSE en tant qu'évolution de SE traditionnel.

Le guide du département de défense [DAU, 2010] a décrit comment les activités traditionnelles de la SE, qui traitent des systèmes individuels et atomiques, ont évolué vers la SoSE. Il est clair que les processus traditionnels de SE doivent être adaptés pour l'ingénierie et le développement des SoS. Nous nous inspirons de ce guide pour caractériser SoSE en identifiant six activités de base suivantes : (voir figure 2.2)

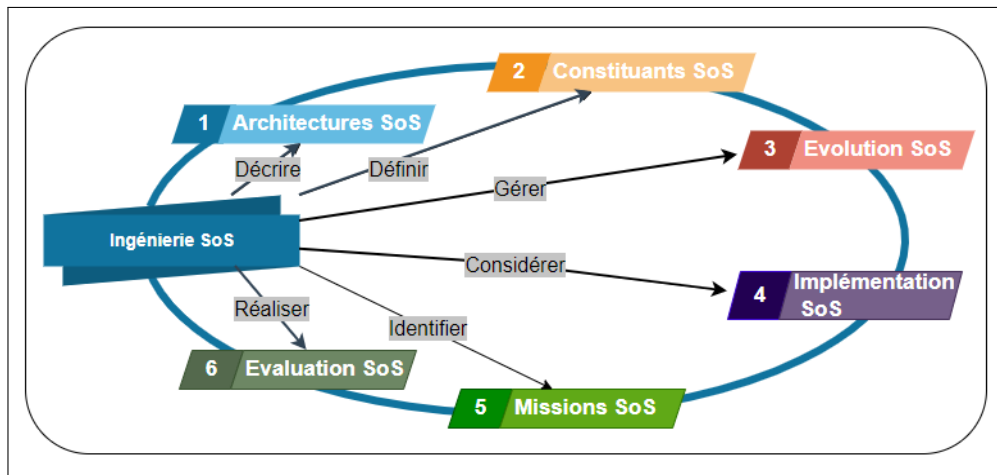


FIGURE 2.2: Principales activités dans la SoSE

- Décrire l'architecture d'un SoS : La description de l'architecture logicielle d'un SoS consiste à définir les éléments syntaxiques et sémantiques qui permettent de définir non seulement les propriétés structurelles telles que les hiérarchies et les fonctionnalités des constituants d'un SoS, mais aussi les propriétés comportementales qui peuvent être affectées par des décisions et des changements architecturaux au sein du SoS.
- Définir les constituants des SoS : La compréhension des systèmes constituants un SoS et leurs relations est un aspect très important. Un concepteur doit comprendre les fonctionnalités des constituants pour qu'il puisse identifier leurs rôles et les liens et coopérations possibles entre eux au sein d'un SoS.
- Gérer l'évolution des SoS : Pour accomplir cette tâche, il faut comprendre la nature et la dynamique d'un SoS pour bien anticiper les différents changements à travers son

évolution. L'évolution doit préserver les propriétés spécifiées de l'architecture d'un SoS, et doit aboutir à des structures cohérentes des constituants.

- Offrir une solution implémentable pour les architectures des SoS : Une architecture logicielle dédiée aux SoS doit être développée avec des outils et des frameworks adéquats, afin d'avoir un aspect qui permet de tester, en exécutant ou en simulant, les changements évolutifs d'un SoS et de constituants.
- Identifier les missions des SoS : Une ingénierie SoS doit pouvoir capturer les missions des SoS, c'est à dire définir des moyens et des mécanismes qui permettent d'identifier, avec précision, la possibilité qu'un SoS peut exhiber pour accomplir certaines missions suite aux différentes évolutions structurels et comportementales de ses constituants.
- Évaluer la cohérence des comportements des SoS : Un concepteur des SoS doit être conscient que les systèmes constituants évoluent indépendamment des SoS. En comprenant l'impact des changements causés par ces évolutions, il faut soit intervenir pour éviter les problèmes d'incohérence au niveau des missions à atteindre dans un SoS, soit développer des stratégies pour réduire l'incohérence dans l'évolution des constituants dans un SoS.

Les approches de conception et de réalisation des systèmes complexes dans le domaine du génie logiciel ont prouvé leur efficacité pour ces systèmes, mais s'avèrent incomplètes pour le cas des SoS. Évidemment, les activités évoquées ci-dessus doivent considérer les spécificités des SoS et leurs constituants. Nous suivons les travaux de recherche sur la SoSE qui ont confirmé le besoin d'appliquer la modélisation et l'analyse formelle pour soutenir la conception des SoS. Particulièrement, nous nous penchons vers les approches qui visent à :

- Définir des langages de modélisation pour décrire et évaluer conceptuellement les SoS.
- Introduire des méthodes et des outils pour décrire et analyser les interactions et les fonctionnalités des constituants d'un SoS.
- Proposer des mécanismes pour gérer les comportements imprévisibles résultant des interactions des constituants d'un SoS.
- Simuler, analyser et observer l'évolution des architectures des SoS.

2.5 Architecture d'un SoS

Dans le cadre de la SoSE, l'architecture d'un SoS est censée réduire sa complexité, en définissant comment les systèmes constituants coopèrent et communiquent ensemble pour réaliser les missions d'un SoS, tout en prenant en compte les détails structurels et comportementaux de ces systèmes et leur impact sur les performances ou les fonctionnalités du SoS.

Dans ce contexte, les architectures logicielles des SoS sont considérées comme un élément essentiel à la réussite du développement de tels systèmes. Elles peuvent apporter des solutions à la modélisation du comportement émergent et la reconfiguration dynamique des SoS.

Les descriptions d'architectures logicielles sont essentielles pour documenter, évaluer et fournir des informations précises et fiables sur des systèmes. Une architecture logicielle, telle définie par la norme ISO 42010 [ISO/IEC/IEEE, 2011], est une représentation des aspects structurels et comportementaux d'un système, y compris ses artefacts associés tels que les composants, les relations et l'environnement. Selon la norme IEEE [IEEE, 1990], elle est décrite comme la structure des composants, leurs relations et les principes et directives régissant leur évolution de conception dans le temps. En particulier, et dans le contexte des SoS, les apports de l'architecture logicielle sont assez importants et motivent son application. En effet, elle permet de :

- Relier les propriétés globales d'un SoS, y compris son comportement émergent, à ses objectifs et missions, ainsi qu'à ses avantages attendus par l'utilisation de modèles en général.
- Donner la possibilité de définir une stratégie d'intégration et de tests progressifs du SoS construit, tout en restant dans la phase de la conception par le biais d'une planification de tests et de simulations par rapport aux éléments architecturaux.
- Être en mesure de montrer l'apparence générale d'un SoS aux utilisateurs prévus, en rendant l'aspect extérieur de la solution physique visible via des vues visuels et graphiques.
- Évaluer les options de solutions alternatives avant de s'engager pleinement dans un processus de développement relativement plus coûteux.
- Offrir la possibilité de réutiliser les éléments d'architectures, en fournissant des supports pour la réutilisabilité.

Ainsi, il est important de définir de nouveaux concepts pour décrire les architectures SoS et de nommer des nouveaux termes alignés sur la terminologie SoS [Oquendo, 2016].

2.6 Conclusion

Les SoS, étant de systèmes complexes et difficiles à gérer, apporte une vision différente mais nécessaire pour faire face à l'évolution croissante des systèmes informatiques. Ils permettent à un regroupement de systèmes d'interagir, de collaborer, et d'aller au-delà de leurs capacités standards afin d'aboutir à de nouvelles fonctionnalités.

Cette première partie de l'état de l'art a mis en revue le contexte de la thèse. Nous avons présenté le concept de SoS et ses différentes définitions. Ensuite nous avons donné les cinq caractéristiques principales des SoS et leurs différents types afin de pouvoir mieux les classer selon leurs domaines d'applications.

Enfin, la dernière partie de ce chapitre introduit la principale différence entre la SoSE et la SE en donnant une importance particulière à l'étape de description d'architecture logicielle dans la SoSE.

Fondements Préliminaires

Sommaire

3.1	Introduction	19
3.2	Langages de description d'architecture	20
3.3	Systèmes Réactifs Bigraphiques (BRS)	21
3.3.1	Anatomie et définition des Bigraphes	21
3.3.2	Langage à termes algébriques	24
3.3.3	Dynamique des Bigraphes	26
3.3.4	Extension des Bigraphes	27
3.3.5	Outils pratiques	28
3.4	Langage Maude	29
3.4.1	Syntaxe et notations	30
3.4.2	Langage de Stratégies Maude	32
3.4.3	Vérification formelle et model-checking	37
3.5	Conclusion	40

3.1 Introduction

Avec la complexité incrémentale des systèmes informatiques, les méthodes de conception et de développement classiques s'avèrent insuffisantes pour garantir la fiabilité de l'analyse de ses systèmes, de leur validité et de leur maintenance.

Afin de réduire cette complexité, et afin d'offrir une haute abstraction qui permet de décrire, de représenter et d'analyser d'une façon non-ambiguë ces systèmes, une solution est de recourir à une architecture logicielle, souvent décrite par un langage de description architecturale (ADL : Architectural Description Language). Pour qu'un ADL ait la puissance d'expressivité qui lui donne la possibilité de décrire des systèmes complexes, il doit être renforcé par des fondements mathématiques qui permettent une meilleure modélisation et analyse des systèmes.

Ce chapitre a pour but de présenter les modèles formels adaptés dans ce manuscrit, et fournir les concepts fondamentaux aux lecteurs qui ne sont pas familiers avec ces modèles, afin de faciliter la compréhension des contenus et des contributions de cette thèse.

Dans un premier temps, la section 3.2 donne une vision globale des langages de descriptions architecturales. Ensuite, la section 3.3 est consacrée à la présentation du formalisme des Systèmes Réactifs Bigraphiques (BRS : Bigraphical Reactive Systems). Elle contient les différentes définitions et notations propres à ce formalisme, et les moyens qu'il offre pour la spécification structurelle et comportementale d'un système.

Enfin, la section 3.4 introduit le langage Maude, ses fondements logiques et ses caractéristiques, et comment les utiliser et les adapter pour implémenter, exécuter et vérifier formellement la sémantique d'un système et son comportement. Un intérêt particulier est donné au langage de stratégie Maude, qui sert à guider la réécriture des états représentant le comportement des SoS.

3.2 Langages de description d'architecture

De manière générale, un ADL (Architecture Description Language) est un langage utilisé pour représenter une architecture logicielle complexe d'un système, qui offre des abstractions et mécanismes qui s'adaptent à la modélisation de cette architecture. Il est défini selon la norme ISO/IEC/IEEE 42010 comme "any form of expression for use in architecture descriptions". La majorité des ADLs ont une syntaxe et une sémantique graphique/textuelle bien définies. L'intérêt d'utiliser un ADL réside dans la possibilité de spécifier rigoureusement une architecture afin de pouvoir l'analyser. Un ADL permet de fournir à la fois un cadre conceptuel et une syntaxe concrète pour caractériser une architecture logicielle.

Un système est décrit dans ce contexte par un composant, alors que les liens entre les systèmes sont décrits par des connecteurs. La topologie structurelle fournies par cet ensemble de composants et connecteurs est appelée configuration. Cependant, Il n'y a pas d'accord universel sur ce que les ADL devraient représenter, notamment en ce qui concerne le comportement de l'architecture. En particuliers, les ADL doivent couvrir les aspects d'évolution et d'analyse du comportement des systèmes complexes modélisés.

Plusieurs ADL sont proposés dans ce contexte [Ozkaya and Kloukinas, 2013], nous citons dans ce qui suit :

Sofa : Un ADL qui définit des systèmes comme des composants, et des relations qui peuvent être sous forme de quatre styles de communication : appel de procédure, messagerie, streaming et tableau noir [Bures et al., 2006]. La sémantique de spécification dans Sofa est décrite et complétée par un protocole spécifié dans l'algèbre des protocoles de comportement BP (Behaviours Protocoles) [Plasil and Visnovsky, 2002]. BP est une forme simplifiée de CSP [Brookes et al., 1984], avec un support supplémentaire pour les expressions régulières.

Prisma : Un ADL orienté aspect qui vise à combiner l'ingénierie logicielle basée sur les composants avec celle orientée aspects [Pérez, 2006]. Un aspect peut servir à différentes tâches telles que la coordination pour interconnecter des composants, et la description des fonctionnalités pour des composants. Ces aspects sont formellement spécifiés avec une forme étendue du langage OASIS [López et al., 1995].

PiLar : Un ADL qui sépare la description d'architecture en deux niveaux : un niveau de base et un niveau méta, où le premier représente des éléments primitifs (contrôlés) et le deuxième les éléments complexes et composites (qui contrôlent) [Cuesta et al., 2002]. La sémantique de PiLar est définie à l'aide de l'algèbre de processus du π -calcul [Milner, 1999].

CONNECT : Cet ADL [Issarny et al., 2011] facilite la description d'interactions entre les composants en adoptant l'algèbre de processus FSP [Magee and Maibaum, 2006] plutôt que le plus complexe CSP. CONNECT prend en charge les tests stochastiques pour l'analyse de systèmes.

AADL : Un ADL qui vise à prendre en charge la spécification du logiciel, du matériel, et des architectures de systèmes mixtes spécialisées pour les systèmes embarqués [Feiler et al., 2006] [Franca et al., 2007]. En raison de sa spécialisation, AADL a le taux d'utilisation le plus élevé [Malavolta et al., 2012]. AADL classe ses notations en trois groupes : (1) un groupe pour spécifier les architectures logicielles : *thread*, *groupe de threads*, *processus*, *données* et

sous-programme, (2) Un autre groupe pour les architectures matérielles : *processeur*, *mémoire*, *périphérique et bus*, et (3) Le dernier groupe est pour les systèmes composés de constituants des deux autres groupes. AADL n'a pas été développé à l'origine avec une sémantique précise. Par la suite, plusieurs tentatives ont été faites dans ce sens [Benammar et al., 2008] [Chkouri and Bozga, 2009] et [Ölveczky et al., 2010] [Hatcliff et al., 2021].

3.3 Systèmes Réactifs Bigraphiques (BRS)

Les systèmes réactifs bigraphiques, ou "Bigraphical Reactive Systems" (BRS), sont un formalisme introduit par Milner dans ses travaux [Milner, 2001] [Milner, 2008] [Milner, 2009] pour la modélisation des systèmes ubiquitaires. Ce modèle est axé sur trois aspects : un aspect spatial qui définit la localité des éléments d'un système, un aspect temporel qui met l'accent sur la connectivité interactionnelle entre ces derniers, et un aspect dynamique qui spécifie la sémantique comportementale d'un système.

Un BRS est constitué de deux parties principales : (1) un modèle bigraphique représentant la structure d'un système et son état (2) un ensemble de règles de réaction décrivant l'évolution du modèle. Un bigraph est doté de deux représentations : une spécification graphique et une spécification textuelle équivalente.

3.3.1 Anatomie et définition des Bigraphes

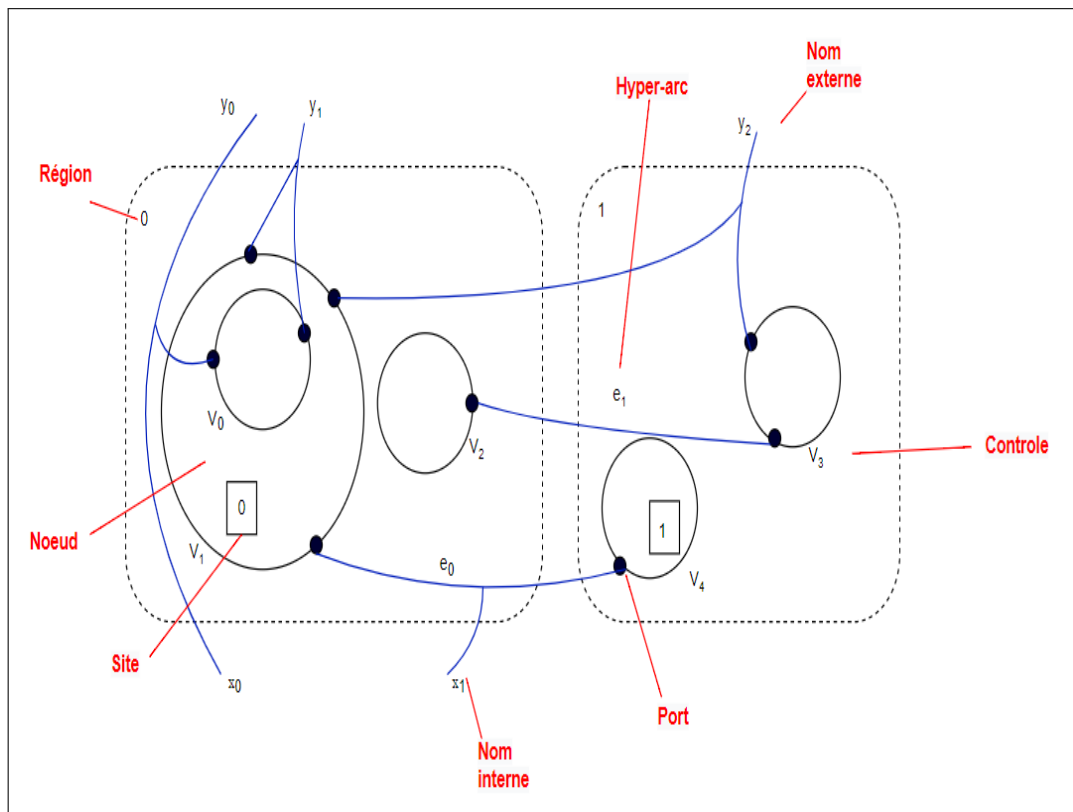


FIGURE 3.1: Anatomie des bigraphes

Dans sa forme graphique, un bigraphe est constitué de *nœuds* représentant des entités d'un système (logiques ou physiques), ces nœuds peuvent prendre des formes géométriques (ovale, carré, etc.). L'emplacement spatial des nœuds décrit une hiérarchie via leurs imbrications. Un nœud est dit *atomique* s'il ne peut contenir d'autres nœuds.

Les nœuds sont contenus dans des *régions* (rectangle en pointillés) indiquant les parties distinctes d'un système. Ils peuvent être insérés dans des *sites* (carré numéroté). Un nœud peut posséder zéro ou plusieurs *ports* (puce noir sur la membrane du nœud), les ports sont associés aux nœuds avec des *contrôles* (identifiant alphabétique) désignant la signature d'un bigraphe. Les liens entre les nœuds sont appelés *hyper-arcs*, et utilisent les ports comme des points de liaisons.

Un bigraphe possède aussi une interface, indiquant d'autres types d'interactions avec son environnement extérieur. Elle est définie à travers des noms internes et des noms externes. La figure 3.1 montre un exemple d'un bigraphe, ayant un ensemble de nœuds ($v_0, v_1 \dots$ etc) et un ensemble d'hyper-arcs (e_0, e_1) dans deux régions 0 et 1.

Finalement, un Bigraphe est défini comme une composition de deux graphes considérés distincts (d'où son nom Bi-graphe), un graphe de place (Figure 3.2) indiquant la localité à travers une distribution spatiale des entités, et un graphe de liens (Figure 3.3) exprimant la connectivité entre ses entités. La particularité entre ses deux graphes regroupant le même ensemble de nœuds, c'est qu'un arc dans un graphe de place indique une relation d'imbrication, alors qu'un hyper-arc dans un graphe de liens est une liaison entre les ports des nœuds.

Dans la suite de cette section, nous présentons quelques définitions mathématiques des éléments clés d'un BRS tirées de [Milner, 2008].

Définition 3.1 (Interface). *Nous distinguons deux type d'interfaces d'un bigraphe, définies via deux paires différentes $\langle m, X \rangle$ et $\langle n, Y \rangle$:*

- $\langle m, X \rangle$: représente l'interface interne d'un bigraphe, où m est le nombre de sites et X est l'ensemble de noms internes.
- $\langle n, Y \rangle$: représente l'interface externe d'un bigraphe, où n est le nombre de régions, et Y est l'ensemble de noms externes.

Définition 3.2 (Signature). *Une signature d'un bigraphe est un tuple (K, Ar) ou :*

- K : est un ensemble de contrôles (spécifiant le nombre de ports)
- Ar : est une fonction $Ar : K \rightarrow N$ qui assigne une arité N à chaque contrôle K . L'arité indique dans ce cas le nombre de ports attribués à chaque nœud, ou à chaque type particulier de nœuds.

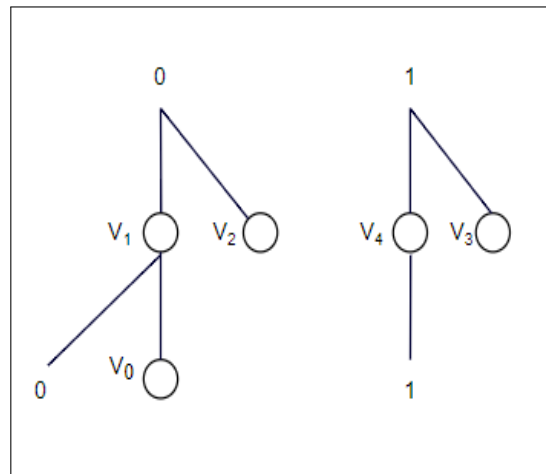


FIGURE 3.2: Exemple d'un graphe de place

Définition 3.3 (Graphe de place). *Un graphe de place est défini par un 4-tuple :*

$$G_p = \langle V, ctrl, prnt \rangle : m \rightarrow n$$

- V : est l'ensemble des nœuds d'un bigraphe .
- $ctrl$: est la signature (K, Ar) des nœuds .
- $prnt$: $m \uplus V \rightarrow V \uplus n$ est une fonction de parenté hiérarchique. Elle associe à chaque nœud ou site un parent hiérarchique qui l'imbrique. La notation $m \uplus V$ indique que les deux ensembles restent disjoints .
- $m \rightarrow n$: permet d'enregistrer les interfaces du graphe de places, où m et n représentent respectivement les interfaces internes et externes du graphe de places.

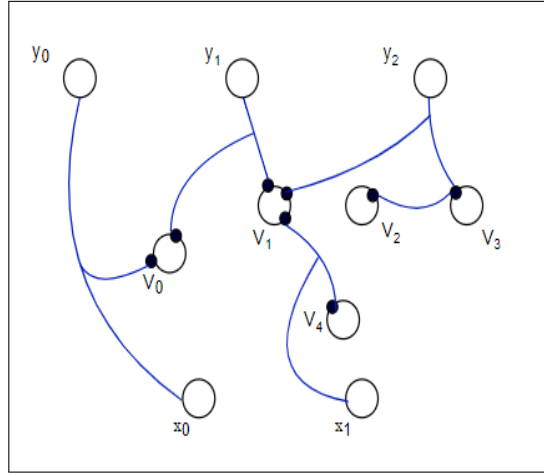


FIGURE 3.3: Exemple d'un graphe de liens

Définition 3.4 (Graphe de liens). *Un graphe de liens est défini par :*

$$G_l = \langle V, E, ctrl, link \rangle : X \rightarrow Y$$

- V : l'ensemble des nœuds d'un bigraphe .
- E : l'ensemble fini d'hyper-arcs reliant les nœuds .
- $ctrl$: est la signature (K, Ar) des nœuds .
- $link$: est une fonction qui a la forme : $X \uplus P \rightarrow E \uplus Y$. Elle indique que la connectivité est entre des noms internes X ou des ports P avec des noms externes Y ou des hyper-arcs E .

Définition 3.5 (Bigraphe). *Un bigraphe est formellement décrit par le 5-tuple :*

$$G = \langle V, E, ctrl, G_p, G_l \rangle : \langle m, X \rangle \rightarrow \langle n, Y \rangle$$

- V : l'ensemble des nœuds d'un bigraphe.
- E : l'ensemble fini d'hyper-arcs reliant les nœuds.
- $ctrl$: est la signature (K, Ar) de l'ensemble de nœuds V .
- G_p : $G_p = \langle V, ctrl, prnt \rangle : m \rightarrow n$: est le graphe de places associé au bigraphe G .
- G_l : $G_l = \langle V, E, ctrl, link \rangle : X \rightarrow Y$: est le graphe de liens associé au bigraphe G .
- $\langle m, X \rangle$: représente l'interface interne d'un bigraphe, où m est le nombre de sites et X est l'ensemble de noms internes.

- $\langle n, Y \rangle$: représente l'interface externe d'un bigraphe, où n est le nombre de régions, et Y est l'ensemble de noms externes.

Exemple :

Nous appliquons ces définitions sur l'exemple de la figure 3.1, il est formellement défini par :

$$G = \langle 2, \{x_0, x_1\} \rangle \rightarrow \langle 2, \{y_0, y_1, y_1\} \rangle$$

, c'est à dire le bigraphe G a deux sites, deux régions, et deux ensembles de noms externes $Y = (y_0, y_1, y_1)$ et internes $X = \{x_0, x_1\}$

- l'ensemble des noeuds V est donné par : $V = \{v_0, v_1, v_2, v_3, v_4\}$
- l'ensemble des hyper-arcs est donné par : $E = \{e_0, e_1\}$.
- la signature du bigraphes est représentée par : $\{v_0 : 2, v_1 : 3, v_2 : 1, v_3 : 2, v_4 : 1\}$.
- le graphe de place : $G_p : 2 \rightarrow 2$ (voir figure 3.2) .
- le graphe de liens : $G_l : \{x_0, x_1\} \rightarrow \{y_0, y_1, y_1\}$ (voir figure 3.3) .

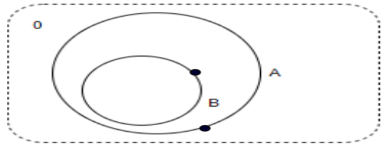
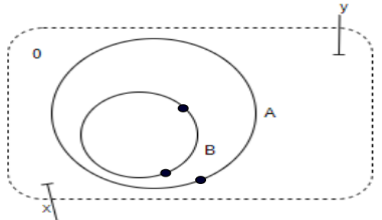
3.3.2 Langage à termes algébriques

En plus de leur forme graphique intuitive, les bigraphes sont dotés d'un langage à termes algébriques qui introduit plusieurs opérations permettant de construire les éléments d'un bigraphe.

Dans le tableau 3.1, nous donnons la définition algébrique des éléments d'un bigraphe illustratif G , ainsi que sa forme graphique équivalente. A chaque terme dans le tableau correspond un opérateur défini par un symbole de couleur rouge dans la forme algébrique, qui permet d'indiquer un concept ou une notion dans le bigraphe. A titre d'exemple, l'opérateur "." est un opérateur binaire qui permet d'imbriquer un deuxième nœud, dans le nœud qui précède l'opérateur.

Bien que cette forme permet de représenter les bigraphes algébriquement, il existe des cas où cette forme algébrique s'avère insuffisante pour capturer tout les éléments d'un système.

TABLE 3.1: Langage à termes algébriques

Terme	Forme Algébrique	Forme Graphique
Imbrication d'un nœud B dans un nœud A	$G = A.B$	
Nom interne x et externe y inactifs (pas liés)	$G = \backslash x y \backslash A.B$	

<p>Juxtaposition du nœud B avec C</p>	$G = \backslash x y \backslash A.(B C)$	
<p>Juxtaposition d'une racine 0 avec une racine 1</p>	$G = (\backslash x y \backslash A.(B C)) (D)$	
<p>Site 0 dans un nœud B</p>	$G = (\backslash x y \backslash A.(B.\square_0 C)) (D)$	
<p>Lien entre les nœuds B, C et un nom externe y</p>	$G = (\backslash x A.(B_y.\square_0 C_y)) (D)$	
<p>Lien entre un nœud B et un nœud C à l'aide d'un hyper-arc e_0</p>	$G = (\backslash x y \backslash A.(B(e_0).\square_0 C(e_0))) (D)$	

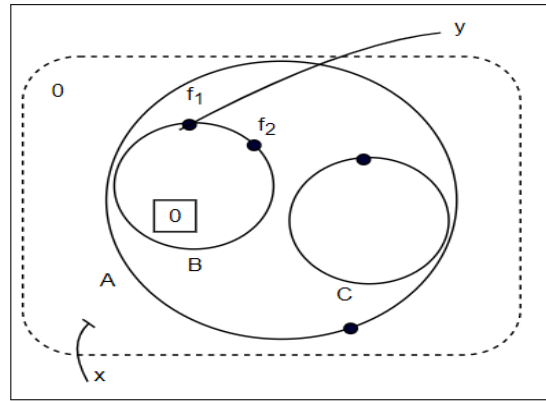
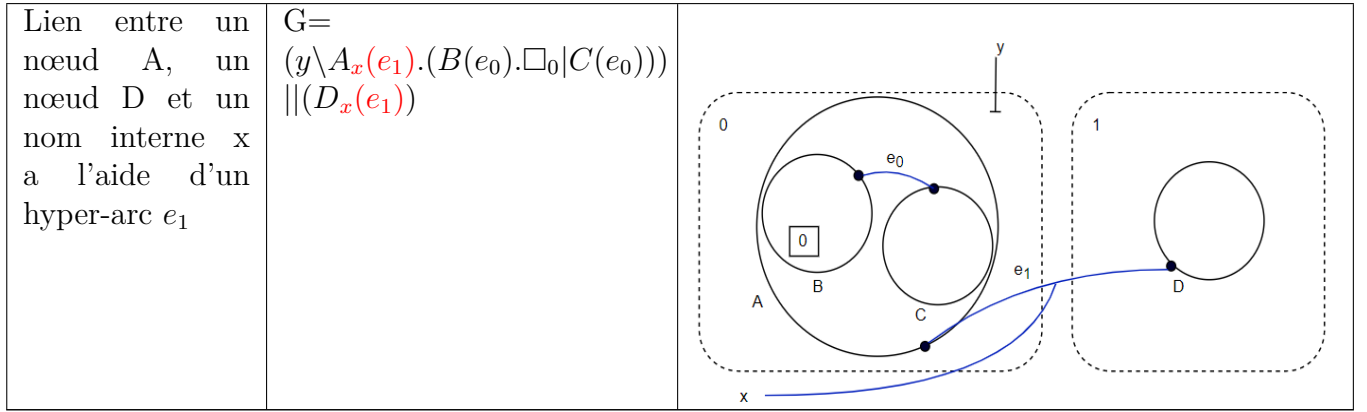


FIGURE 3.4: Bigraphe avec des ports explicites

Prenant le cas d'un système défini dans le bigraphe de la figure 3.4, sa forme algébrique est définie par : $G = \setminus x A.(B_y \cdot \square_0 | C)$. Dans cet exemple, les deux ports f_1 et f_2 du nœud B sont distincts : chaque port représente une fonctionnalité différente, et son lien avec le nom externe "y" donnera un résultat distinct qui dépend du port lié car la fonctionnalité va changer. Cependant, dans la forme algébrique telle qu'elle a été définie dans [Milner, 2008], nous ne retrouvons pas cette représentation des ports.

3.3.3 Dynamique des Bigraphes

Pour modéliser la dynamique des systèmes, les BRS sont équipés d'un ensemble de règles de réactions définissant comment les bigraphes peuvent évoluer et se reconfigurer. La définition formelle d'une règle de réaction est la suivante :

Définition 3.6 (Règle de réaction). *Une règle de réaction prend la forme $R \rightarrow R' : O$, où :*

- R est appelé *redex*, il représente le bigraphe à changer et à transformer, il est noté par $R : m \rightarrow J$
- R' est le *reactum*, il correspond au résultat après transformation, il est noté par $R' : m' \rightarrow J, n$
- O est une fonction de transformation d'ordinaux noté par $n : m' \rightarrow m$. Elle permet d'établir une correspondance entre les interfaces internes de R' et R .

L'application d'une règle de réaction est dite spontanée [Sevegnani and Calder, 2015] : dès qu'il y'a un matching du redex dans le bigraphe contextuel, celui-là est remplacé par le reactum. Nous distinguons principalement deux types de règles de réaction :

- Règles affectant les places : des règles qui permettent de rajouter, supprimer, ou déplacer un nœud.
- Règles affectant les liens : des règles qui permettent d'établir (ou détruire) des liens entre deux nœuds, ou entre un/plusieurs nœuds et un nom interne/externe.

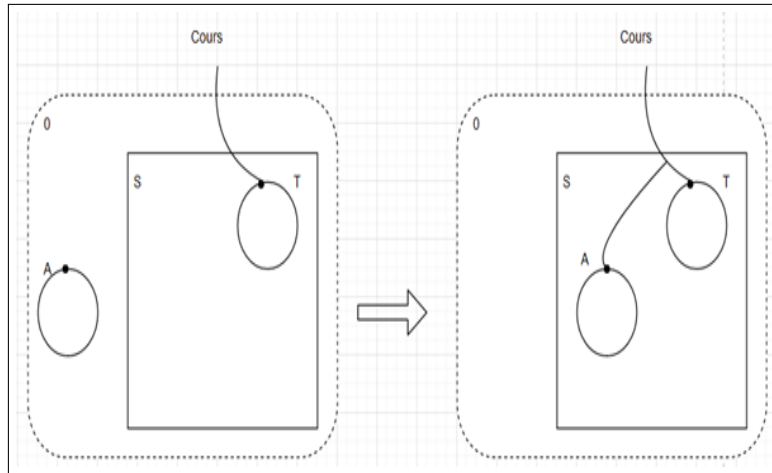


FIGURE 3.5: Exemple d'une règle de réaction

Exemple : Le redex dans la figure 3.5 contient un apprenant A, un tuteur T qui est dans une salle S, et qui assure un cours. En appliquant la règle de réaction dans cette figure, cela permet à l'apprenant A d'entrer dans la salle S et de suivre le cours assuré par le tuteur T, ce qui est représenté par l'établissement d'un lien entre l'apprenant, le tuteur et le cours.

3.3.4 Extension des Bigraphes

Les bigraphes ont subi plusieurs extensions depuis leur introduction, plusieurs ajouts et raffinements ont été proposés à la définition initiale des bigraphes. Quatre extensions principales sont introduites et expliquées dans ce qui suit, afin de montrer les nouvelles notions ajoutées pour augmenter les possibilités de modélisation à l'aide de BRS.

- **Bigraphes Contraignants** (ou "Binding Bigraphs") : Cette extension a été proposée par [Damgaard and Birkedal, 2005]. Son principe est d'offrir une séparation plus souple entre les deux notions principales des bigraphes, la localité et la connectivité. Cela est réalisé en mettant les noms des bigraphes aux seins des racines et des sites, ce qui donne la possibilité de savoir si une arête franchit la limite d'un nœud. Par conséquent, les arêtes d'un nœud peuvent uniquement être reliées aux ports situés au sein de ce nœud. Les bigraphes de cette extension sont beaucoup adaptés là où il faut limiter la communication dans certaines entités uniquement. Par exemple, une arête propre à un nœud ne peut relier que les ports situés au sein de ce nœud. Ce genre de contrôle sur les arêtes est bien utile dans des systèmes dotés de protocoles de sécurités privés par exemple, en limitant et observant la connectivité des arêtes de communication de ses entités.
- **Bigraphes Stochastiques** (ou "Stochastic Bigraphs") : Dans cette extension introduite par [Krivine et al., 2008], une valeur stochastique est associée aux règles de réaction. De ce fait, une chaîne de Markov à temps continu (CTMC) peut interpréter l'espace d'états généré. Vu qu'une réaction peut être le résultat de plusieurs différentes règles, le taux de réaction pour une règle est obtenu par le produit du taux constant et le nombre d'occurrence de la règle. Ce taux représente le comportement stochastique de la réaction. Cette extension est adaptée dans l'analyse stochastique des systèmes distribués généralement.

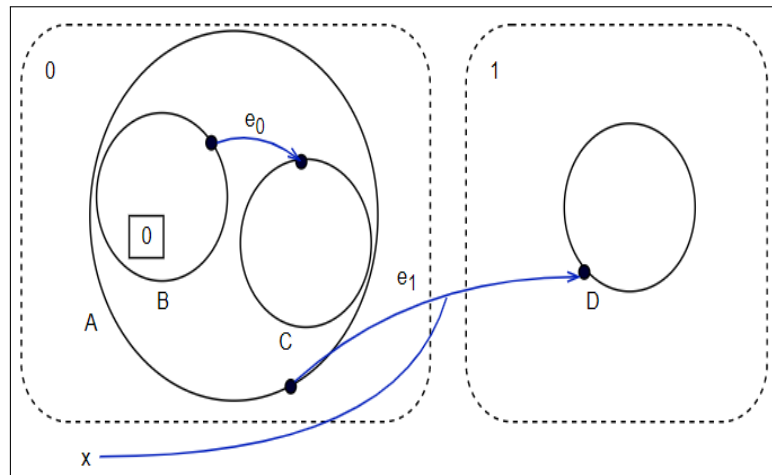


FIGURE 3.6: Exemple d'un bigraphe dirigé

- **Bigraphes Dirigés** (ou "Directed Bigraphs") : Les bigraphes dirigés introduits par [Grohmann and Miculan, 2007] forment une extension où une redéfinition du graphe de liens est faite en attribuant une direction aux arêtes. Les arêtes sont représentées comme des sommets d'un graphe. Cette extension est plutôt utile quand il s'agit de modéliser des systèmes où la transition et la communication doivent être orientées entre les composants d'un système. La figure 3.6 définit un bigraphe avec deux régions 0 et 1, contenant respectivement trois nœuds A, B, C et un nœud D, avec deux hyper-arcs e_0 et e_1 qui ont un sens de communication bien précis.
- **Bigraphes avec Partage** (ou "Bigraphs with Sharing") : Cette extension permet à une place d'un bigraphe d'avoir plusieurs parents, elle est introduite par [Sevegnani and Calder, 2015]. Elle offre une notion de partage qui permet de maintenir une copie d'un nœud à chacun de ses parents, et de relier ses copies par un lien unique.

3.3.5 Outils pratiques

La manipulation manuelle des bigraphes est une tâche complexe, car ils représentent un langage formel pour la modélisation des larges systèmes distribués en particulier, d'où la nécessité d'introduire un outil autour de ces derniers. Des environnements basés sur des langages à haut niveau qui permettent de spécifier, visualiser, et vérifier les bigraphes ont été proposés. Nous les résumons dans le tableau 3.2.

Bien que les BRS peuvent représenter différents systèmes informatiques d'une façon claire, structurée et non ambiguë, les outils existant autour de ce formalisme ne sont pas encore matures, la majorité représentent des prototypes ayant plusieurs lacunes. En particulier, BigMC est le seul outil qui prend en charge la vérification formelle des bigraphes, mais il s'avère très restrictif dans son expressivité, il ne permet pas d'exprimer des propriétés en LTL par exemple, aussi sa syntaxe représente un ensemble limité des expressions bigraphiques.

Plusieurs tentatives de recherches, spécialement issues de notre équipe de recherche GLSD (Génie Logiciel et Systèmes Distribués) du laboratoire LIRE, ont essayé de surmonter ce défi en projetant les modèles bigraphiques conçus dans le langage Maude. En effet, les BRS ont été appliqués pour modéliser un bon nombre de systèmes dans des travaux de recherche différents. A titre d'exemple, les BRS ont été utilisés dans [Benzadri et al., 2017] comme un moyen de donner une sémantique formelle aux concepts d'architecture Cloud, en précisant sa structure statique et son évolution dynamique. Les auteurs dans [Bouhroum et al., 2019] ont défini une

TABLE 3.2: Outils autour des BRS

Outil	Principe
BPL Tool : ou Bigraphical Programming Language Tool [Højsgaard and Glenstrup, 2011]	<ul style="list-style-type: none"> — Il permet de gérer et de simuler tout en visualisant des bigraphes. — Il est doté d'un parseur et de deux moteurs : un premier pour le parsing et un deuxième pour la normalisation. — Il est décrit dans un langage BPL qui est défini en terme de blocs du langage Standard ML (SML) [Milner et al., 1997]. — Il est utilisé pour la spécification des systèmes de téléphonie mobile [Højsgaard and Glenstrup, 2011], ou les modèles platographiques [Elsborg, 2009]. — Il peut être utilisé soit à travers une interface utilisateur dans le Web, ou bien comme étant une bibliothèque de programmation.
BigMC : ou Bi-graphical Model Checker [Perrone et al., 2012]	<ul style="list-style-type: none"> — Il est le premier outil conçu pour la vérification des bigraphes, en se basant sur la technique du matching dans le moteur de BPL. — Il permet de faire une exploration de tous les états possibles d'un BRS afin de vérifier la validité ou la violation de propriétés données. — Il possède une syntaxe bien appropriée pour la spécification.
BigRed [Faithfull et al., 2013]	<ul style="list-style-type: none"> — Il permet de spécifier et d'éditer visuellement des bigraphes, et d'exporter ces spécifications en fichiers sous différents formats, notamment XML. — Cet outil est extensible et peut avoir l'intégration de BigMC afin de l'utiliser sur les bigraphes définis.
BigraphER : ou Bigraph Evaluator and Rewriting [Sevgnani and Calder, 2016]	<ul style="list-style-type: none"> — Il sert à manipuler, visualiser et simuler des BRS. — Il prend en charge les BRS stochastiques et les BRS avec partage. — Il est supporté par un langage de spécification similaire à la forme algébrique des bigraphes, appelé BSL ("Bigraph Specification Language")

approche formelle pour répondre aux défis de sécurité dans une architecture Fog. Les BRS ont aussi été adaptés dans [Khebbab et al., 2019] pour la modélisation formelle et vérification des stratégies d'élasticité dans les systèmes Cloud.

3.4 Langage Maude

Maude [Clavel et al., 2007] [Clave et al., 2000] [McCombs, 2003] est un langage qui implémente la logique de réécriture [Meseguer, 1996] et ses théories mathématiques. Il est conçu pour formaliser et analyser des systèmes concurrents et répartis. Dans ce qui suit, nous présentons les concepts les plus importants de Maude, ainsi que ceux du langage de stratégies Maude qui représente une extension de ce dernier.

3.4.1 Syntaxe et notations

Maude permet de définir les deux aspects d'un système informatique : l'aspect statique à l'aide d'une théorie équationnelle définissant une syntaxe particulière, l'aspect dynamique est pris en charge par le biais des règles de réécriture qui, à partir d'un état initial S_i , font une transition vers un état suivant S_j , changeant par cela le comportement général du système. Les deux aspects forment une théorie de réécriture Maude. Par conséquent, toute théorie de réécriture a une théorie équationnelle sous-jacente.

Les spécifications Maude peuvent être exécutés et validés, et son avantage consiste à sa capacité de représenter les calculs concurrents et non déterministes. Maude met l'accent sur trois points : (1) la simplicité : un sens clair des spécifications (2) l'expressivité : puissance expressive grâce à la capacité d'exprimer des logiques (3) la performance : rend les performances du système compétitives avec d'autres langages de programmation.

Nous définissons dans ce qui suit les théories équationnelles et de réécriture du langage Maude.

Définition 3.7 (Théorie équationnelle Maude). *Une théorie équationnelle Maude est basée sur une logique mathématique d'appartenance, et elle a la forme :*

$$(\Sigma, E \cup A). \text{ où :}$$

- Σ : est la signature qui précise la structure typée de chaque élément d'un système.
- E : est la collection d'équations (éventuellement conditionnelles) déclarées agissant sur la structure.
- A : est la collection d'attributs équationnels déclarés pour les différents opérateurs, comme l'attribut de commutativité ou d'associativité entre les éléments.

Définition 3.8 (Théorie de réécriture Maude). *Une théorie de réécriture est une théorie équationnelle étendue par la prise en charge des aspects dynamiques. elle a la forme :*

$$(\Sigma, E \cup A, R). \text{ où :}$$

- Σ, E et A représentent la théorie équationnelle d'un système.
- R est l'ensemble de règles de réécritures (éventuellement conditionnelles) qui sont appliquées pour changer le comportement d'un système, faisant ainsi évoluer sa structure équationnelle.

Dans Maude, un module est l'unité basique de spécification et de programmation. Au sein de sa version standard, il existe deux principaux types de modules Maude selon les théories qu'il implémente :

- Un module fonctionnel qui implémente la théorie équationnelle en spécifiant la structure d'un système.
- Un module système qui implémente la théorie de réécriture en spécifiant les évolutions dynamiques d'un système.

Module fonctionnel : Le module fonctionnel permet de spécifier la syntaxe structurelle d'un système. Un module fonctionnel a la syntaxe suivante :

$$fmod \langle nom \rangle is \langle déclarations et instructions \rangle endfm.$$

Un module fonctionnel commence et fini respectivement par les mots clés *fmod* et *endfm*. Ce qui existe entre les deux c'est des déclarations ou des instructions que nous détaillons ci-dessous :

- Importations : nous pouvons importer d'autres modules Maude afin d'utiliser leurs spécifications.

- Sortes et sous sortes : elles sont une catégorie de valeurs qui décrit des types de données spécifiques.
- Variables : elles sont définies pour s'étendre sur une sorte particulière.
- Opérations : spécifie le domaine et le co-domaine des valeurs, elles ont des propriétés comme l'associativité ou la commutativité.
- Équations : il y'a deux types d'équations, les équations simples et les équations conditionnelles. Une équation simple a la forme

$$eq < Terme1 > = < Terme2 > [< Attributs >]$$

Une équation conditionnelle est notée par :

$$ceq < Terme1 > = < Terme2 > \textit{if} < EqCondition1 > \\ \dots < EqConditionk > [< Attributs >]$$

Pour qu'une équation soit exécutable, toutes les variables dans la partie droite (Terme2) de l'équation doivent apparaître dans sa partie gauche (Terme1), et les deux parties doivent être de la même sorte. Les attributs sont les propriétés associés aux équations.

Module système : Le module système Maude implémente la logique de réécriture, il utilise un module fonctionnel qui décrit la syntaxe d'un système, pour lui définir une sémantique opérationnelle et dynamique. Il a la forme :

$$\textit{mod} < nom > \textit{is} < déclarations \textit{et} \textit{instructions} > \textit{endm}.$$

Un module système dans Maude est déclaré avec le mot-clé `mod`, suivi de nom du module. En plus des éléments existants dans le module fonctionnel, un module système utilise des règles de réécritures, qui peuvent être conditionnées. Les règles (rl) et les règle conditionnelles (crl) sont définies comme suit :

$$\textit{rl}[< Label >] : < Terme1 > = > < Terme2 > [< Attributs >]. \\ \textit{crl}[< Label >] : < Terme1 > = > < Terme2 > \textit{if} < Condition k > \dots \\ < Condition n > [< Attributs >].$$

Label est l'étiquette d'une règle, tandis que les termes Terme1 et Terme2 sont des termes de la même sorte. Les règles peuvent être exécutées si la partie gauche d'une règle correspond à un fragment de l'état du système, et uniquement si les conditions qui suivent sont satisfaites. La transition spécifiée par la règle pourrait ainsi être appliquée, et le fragment identifié de l'état se transforme en l'instance qui correspond du coté droit.

Exemple : Afin d'illustrer ces notations Maude, nous présentons un exemple défini dans les figures 3.7 et 3.8, tiré de [Eker et al., 2007]. Il s'agit d'un simple jeu qui consiste à sélectionner deux entiers naturels depuis un tableau noir (BLACKBOARD), et d'écrire leur moyenne arithmétique après leur suppression du tableau.

La figure 3.7 définit un module système BLACKBOARD, ce module sert a construire des tableaux noir décrits par une sorte BLACKBOARD (qui est une sous sorte de la sorte Nat qui correspond aux entiers) à l'aide de l'opérateur "`__`" qui permet de concaténer des entiers dans le tableau. La règle de réécriture « `play` » permet de faire évoluer le tableau en exécutant l'opération arithmétique du jeu a l'aide de deux variables N et M.

Le deuxième module EXT-BLACKBOARD (figure 3.8) étend le premier, et permet de définir des équations qui permette de calculer le maximum et le minimum des entiers dans le tableau, (notées `max` et `min` respectivement), ainsi qu'une équation "`remove`" qui permet d'enlever un entier du tableau quand il contient au moins deux entiers (enlever X des deux entiers X et B dans l'exemple).

```

mod BLACKBOARD is
  protecting NAT .
  sort Blackboard .
  subsort Nat < Blackboard .
  op __ : Blackboard Blackboard -> Blackboard [assoc comm] .
  vars M N : Nat .
  rl [play] : M N => (M + N) quo 2 .
endm

```

FIGURE 3.7: Module Système BLACKBOARD [Eker et al., 2007]

```

fmod EXT-BLACKBOARD is
  including NAT .
  including BLACKBOARD .
  ops max min : Blackboard -> Nat .
  op remove : Nat Blackboard -> Blackboard .
  vars M N X Y : Nat .
  var B : Blackboard .
  eq max(N) = N .
  eq max(N B) = if N > max(B) then N else max(B) fi .
  eq min(N) = N .
  eq min(N B) = if N < min(B) then N else min(B) fi .
  eq remove(X, X B) = B .
endfm

```

FIGURE 3.8: Module fonctionnel EXT-BLACKBOARD [Eker et al., 2007]

3.4.2 Langage de Stratégies Maude

Maude est muni d'autres notations qui permettent d'exprimer différemment la structure et la dynamique des systèmes spécifiques. Il se base sur le paradigme orienté objet [Ölveczky and Meseguer, 2007] pour définir des modules orientés objet offrant une syntaxe appropriée dotée de classes, d'objets et de messages entre eux.

Il possède aussi une extension dédiée aux stratégies [Eker et al., 2007] [Martí-Oliet et al., 2009] qui a pour but la séparation modulaire entre les règles de réécriture et leurs contrôles (exécution) en utilisant des stratégies. Dans le cadre de notre travail, Nous utilisons cette extension de Maude afin de définir les comportements des systèmes complexes et distribués ; ce langage permet d'avoir un mécanisme pour guider la réécriture de leurs comportements.

Syntaxes et notations

Le comportement d'un système dans Maude dépend uniquement de ses règles de réécritures et de leurs applications, par contre dans le langage de stratégies Maude, ce n'est plus le cas. Grâce à la séparation offerte par ce langage, nous pouvons avoir plusieurs modules de stratégies pour le même module système Maude, où chaque module offre une définition d'un chemin d'exécution différent, ce qui offre plus de flexibilité par rapport aux différents scénarios exécutés.

Un comportement dépendra des stratégies qui contrôlent les règles de réécritures. Nous le définissons ci-dessous :

Définition 3.9 (Module de stratégies Maude). *Un module de stratégies déclaré par le langage de stratégies Maude peut être sous la forme :*

$$(\Sigma, E \cup A, S(R, SM)). \text{ où :}$$

- Σ , E et A représente la théorie équationnelle du système en évolution.
- S est une sémantique décrivant le comportement d'un système, elle est construite à partir d'un module R contenant des règles de réécritures, et d'un ensemble de stratégies SM qui va guider la réécriture de ces règles en utilisant des stratégies.

```

smod nomdumodule is
  protecting Module1 .
  including ModuleStrategieX .
  strat S1 @ MX.
  sd S1 := Expression .
  csd S1 := Expression' if Condition .
endsm

```

FIGURE 3.9: Exemple d'un module de stratégies

La syntaxe des modules de stratégies est la suivante :

$$smod < nom - module > is < déclarations - et - expressions > endsd.$$

où *smod* et *endsd* sont les mots clés indiquant le début et la fin du module de stratégies, tandis que les déclarations et les expressions représentent des variables, des importations d'autres modules et des déclarations de stratégies [Rubio Cuéllar et al., 2021]. En général, un module de stratégies Maude à la syntaxe générique présentée dans la figure 3.9, une explication de chaque instruction dans la figure est donnée dans le tableau 3.3.

TABLE 3.3: Les éléments clés dans un module de stratégies

Instruction	Signification
smod	Déclaration d'un module de stratégies Maude.
Module1	Le module dont nous voulons contrôler les réécritures
ModuleStrategieX	(Facultatif) Importation d'un autre module de stratégies Maude, afin de d'utiliser ses stratégies dans ce module
strat	Déclarer une nouvelle stratégie
S1	Le nom (identificateur) de la stratégie
@ MX	La stratégie déclaré va être appliqué aux termes d'une sorte MX
sd	Définition d'une stratégie (sd : strategy definition) suivi de l'identificateur et l'expression de cette stratégie
Expression	Un terme pour décrire la stratégie

Afin d'exécuter une stratégie dans le langage de stratégies Maude, il faut passer par la commande *srew* qui a la syntaxe suivante :

$$srew Terme \textit{ using Expression}$$

Où *Terme* est l'état initial (ou l'état à un instant donné) du système, *Expression* est la stratégie à appliquer sur ce terme.

Exemple : Nous introduisons un simple exemple pour montrer comment définir et exécuter une stratégie, supposant que nous avons a un système spécifié en Maude autant que : $System0 < Sub - system1, Sub - system2 >$. Cela définit un Système (d'une sorte Système) *System0* ayant deux systèmes constituants, *Sub-system1* et *Sub-system2*.

Il est doté d'un module système *SystèmeRules*, dans lequel figure une règle de réécriture qui permet de remplacer le constituant *Sub-System2* par un autre constituant qui est *Sub-System3*. la règle a la forme :

$$rl : [Replace13]System0 < Sub - system1, Sub - system2 > => System0 < Sub - system1, Sub - system3 >$$

Le module de stratégies adapté à cet exemple est représenté par la figure 3.10. La stratégie est exécutée par la commande :

`srew System0 < Sub - system1, Sub - system2 > using Replace`

Le résultat de l'exécution est le terme "`System0<Sub-system1, Sub-system3>`".

```
smod SystemStrategies is
    protecting SystemRules .
    strat Replace @ Système .
    sd Replace := Replace13 .
endsm
```

FIGURE 3.10: Exemple d'une stratégie

Types de stratégies

Une stratégie basique est décrite dans Maude comme une opération qui produit des termes (résultats) lorsqu'elle est appliquée. Elle consiste en l'application d'une règle de réécriture (en utilisant l'étiquette de la règle) sur un terme donné. Elle permet aux variables de la règle d'être instanciées via des substitutions avant que la règle soit appliquée. Des stratégies plus complexes peuvent être définies à partir de ces stratégies de base, nous les détaillons dans ce qui suit :

1- Stratégies Combinées : Les stratégies basiques sont combinées pour former des stratégies plus complexes à l'aide de nombreux opérateurs et expressions régulières de combinaisons.

Opérateurs de séparation « | » et « ; » : Ces opérateurs permettent de faire suivre une suite de stratégies/règles de réécritures selon un ordre et des propriétés particulières, Nous distinguons deux types d'opérateurs :

- « | » **Union :** opérateur commutatif et associatif qui permet de concaténer deux ou plusieurs stratégies, il a forme : $Strategy0 := Strategy1|Strategy2$. Cela est interprété par : La stratégie 1 a une priorité sur la stratégie 2 et exécutée avant elle dans le cas ou les deux peuvent être déclenchées au même temps. Cependant, aucune stratégie influence l'autre. Si la première ne peut pas être exécutée la deuxième s'exécutera toujours.

- « ; » **Concaténation** : opérateur associatif qui permet d'exécuter deux ou plusieurs stratégies séquentiellement, il prend la forme : $sdStrategy0 := Strategy1; Strategy2$ Cela indique que Strategy2 doit suivre la Strategy1, et si la première n'est pas exécutée, alors la deuxième ne le sera pas non plus.

Opérateurs de control « * », « ! » et « + » : Ils contrôlent le nombre d'itérations que le moteur de réécriture doit faire lors de l'exécution d'une stratégie, c'est plutôt utile lorsque nous ne voulons pas répéter l'exécution plusieurs fois

- **sd Strategy = : (.....)*** : Maude affichera toujours une solution à chaque itération sur les règles de la stratégie. Même s'il n'y a pas de substitutions possible, il affichera 0 réécritures et renvoie l'état initial. Cet opérateur est utile dans le où nous nous savons pas si nous avons vraiment besoin d'itérations dans la stratégie.
- **sd Strategy = : (.....)+** : Maude doit faire au moins une itération sur toute la stratégie pour afficher le résultat, cela forcera le moteur de réécriture à faire cette itération. C'est utile dans le cas où une des règles ne peut pas être exécutée dans la première application de la stratégie, et peut uniquement l'être une fois une règle venant après elle est exécutée, le moteur reviendra à la première dans l'itération.
- **sd Strategy = : (.....)!** : Maude fera toutes les combinaisons de substitutions possibles pour afficher uniquement le résultat final comme solution, donc le moteur de réécriture itère jusqu'à l'état final directement.

Définition 3.10 (Stratégies Combinées). *Une expression peut être associée à la combinaison de plusieurs stratégies ou règles de réécriture, ayant la syntaxe suivante :*

$sd S := (Ri \text{ Opérateur de séparation } Rj \dots \text{ Opérateur de séparation } Rn) \text{ Opérateur de control}$

Ou bien

$sd S := (Si \text{ Opérateur de séparation } Sj \dots \text{ Opérateur de séparation } Sn) \text{ Opérateur de control}$

- $Ri \ Rj \dots \ Rn$: Sont des règles de réécritures appliquées pour former une stratégie.
- $Si \ Sj \dots \ Sn$: Sont des stratégies appliquées pour former une stratégie.
- Les opérateurs de séparations servent à séquencer les règles de réécritures/stratégies selon des propriétés particulières, alors que les opérateurs de contrôle sont rajoutés à la fin de la déclaration pour délimiter les itérations.

Le tableau 3.4 résume les opérateurs, leurs syntaxes et leurs fonctionnalités.

TABLE 3.4: Opérateurs de stratégies

Opérateur	Syntaxe	fonctionnalités
;	$sdStrategy0 := S1, S2$	Concaténation
	$sdStrategy0 := S1 S2$	Union
*	$sdStrategy =: (.....)*$	Au moins 0 itérations
+	$sdStrategy =: (.....)+$	Au moins 1 itérations
!	$sdStrategy =: (.....)!$	Itérations jusqu'à la fin de l'exécution

2- Stratégies de type Si...Sinon :

Elles sont représentées par des conditions de la forme Si...Sinon entre des stratégies. Elles se présentent ainsi :

$$sd ST := S1 ? S2 : S3$$

Cela indique que dans une stratégie de type Si...Sinon nommée ST, si la stratégie S1 est exécutée, alors S2 est exécutée, sinon S3 sera exécutée à la place de S1 et S2.

3- Stratégies de "Matching" :

Une stratégie de ce type permet d'imposer des conditions sur un état en faisant du « matching », avant d'appliquer une règle de réaction dans la stratégie. Elle a la forme :

$$sd\ S1 := matchrew\ T\ s.t.\ C\ by\ T\ using\ X\ où :$$

- T est un terme.
- C est une condition que T doit satisfaire.
- "by T" indique que le résultat d'exécution va remplacer le terme T.
- X est soit une stratégie, soit une règle de réécriture.

Exemple :

```

smod BLACKBOARD-STRAT is
  protecting EXT-BLACKBOARD .
  var B : Blackboard .
  vars X Y : Nat .
  strat maxmin @ Blackboard .
  sd maxmin := (matchrew B s.t. X := max(B) /\ Y := min(B) by
                B using play[M <- X ; N <- Y] ) ! .
  strat maxmax @ Blackboard .
  sd maxmax := (matchrew B s.t. X := max(B) /\ Y := max(remove(X,B)) by
                B using play[M <- X ; N <- Y] ) ! .
  strat minmin @ Blackboard .
  sd minmin := (matchrew B s.t. X := min(B) /\ Y := min(remove(X,B)) by
                B using play[M <- X ; N <- Y] ) ! .
endsm

```

FIGURE 3.11: Module de stratégies BLACKBOARD-STRAT [Eker et al., 2007]

Afin d'illustrer les stratégies dans Maude, nous reprenons l'exemple précédent du tableau noir. Le but c'est de définir des stratégies qui permettent de choisir le "pattern" depuis lequel les nombres sont sélectionnés dans le tableau. La figure 3.11, qui importe le module EXT-BLACKBOARD et ses équations, définit un module de stratégies BLACKBOARD-STRAT contenant trois stratégies pour le tableau noir. Elles sont des stratégies de type "matching" pour imposer à chaque fois quelle équations à appliquer sur les deux nombres pour lesquels nous voudrions calculer la moyenne. Les trois stratégies sont :

- Maxmin : permet de prendre le maximum et le minimum du tableau (en mettant les deux variable X et Y comme résultats d'appel aux fonctions max et min) dans la condition de la stratégie, et en appliquant la règle play sur X et Y tout en enregistrant le résultat dans le tableau.
- Maxmax : fait la moyenne des deux maximum dans le tableau (le deuxième maximum est défini après suppression du premier avec l'équation remove).
- Minmin : calcule la moyenne des deux minimum dans le tableau noir.

Ces stratégies définissent chacune un moyen différent d'appliquer la règle "play" pour le même tableau noir B, la seule différence réside dans le choix des entiers à sélectionner dans chaque état du tableau, ce qui va influencer le résultat final. Nous notons que les trois stratégies

```

Maude> srew 2000 20 2 200 10 50 using maxmin .
result NzNat : 178
Maude> srew 2000 20 2 200 10 50 using maxmax .
result NzNat : 77
Maude> srew 2000 20 2 200 10 50 using minmin .
result NzNat : 1057

```

FIGURE 3.12: Résultats d'exécution des stratégies du BLACKBOARD [Eker et al., 2007]

sont suivies par l'opérateur '!', pour dire que la règle "play" va être répéter jusqu'à l'état final directement, c'est-à-dire jusqu'à avoir un seul entier.

Les résultats d'exécution de chaque stratégie sont affichés dans la figure 3.12, par exemple l'application de la stratégie Maxmax donnera l'entier 77, qui est différent du résultat d'exécution de chacune des autres stratégies. Ceci montre la présence d'une reconfiguration dynamique qui affecte le même état (le même tableau initial 2000,20...50) dans notre exemple pour donner des états évolutifs différents d'un système.

3.4.3 Vérification formelle et model-checking

Maude permet de déclarer des propriétés dans des modules systèmes, donc des modules de vérifications peuvent être spécifier dans Maude, dans le but de définir et de vérifier ces propriétés, ce qui amène à une analyse formelle des comportements d'un système. Cette analyse est faite par la validation (ou la non validation avec des contres exemples) de ces propriétés au cours de l'exécution des règles de réécritures.

Le système Maude est doté de plusieurs outils puissants dans l'analyse des comportements d'un système, nous pouvons noter le prouveur de théorèmes, la vérification par invariants, le LTL model-checker et l'analyse de cohérence. Nous adoptons le LTL model-checker [Rozier, 2011], que nous détaillerons ci-dessous. Ce choix est justifié par sa pertinence et sa richesse intuitive vis-à-vis le travail présenté dans cette thèse. Nous invitons les lecteurs à consulter [Clavel et al., 2007] et [Clave et al., 2000] pour plus de détails sur les autres techniques et outils d'analyse formelle avec Maude.

Le model-Checker Maude utilise la logique temporelle linéaire (LTL), qui est assez expressive et utilisée sur une large étendue, vu sa capacité à décrire et vérifier formellement plusieurs types de propriétés. Les plus importantes sont la sûreté : s'assurer que quelque chose d'indésirable arrive jamais, et la vivacité : quelque chose de bon va finir par arriver dans le système. Il existe plusieurs opérateurs et symboles LTL dans le model-checker, nous les notons dans le tableau 3.5.

TABLE 3.5: Tableau des symboles et opérateurs LTL

Opérateur / Symbole LTL	Significations
\wedge	Conjonction / ET logique
\vee	Disjonction / OU logique
\neg	Négation / NON logique
\bigcirc	Le prochain état
$\langle \rangle$	Éventuellement dans le futur
$[]$	Globalement / toujours dans le futur
\Rightarrow	Implication
\cup	Jusqu'à un état

Cette signature d'opérateurs/symboles est prédéfinie dans un module « modelchecker.Maude

». Ainsi, des propriétés et propositions LTL peuvent être utilisées dans un module Maude, de la manière suivante sous forme d’une équation conditionnelle :

$$ceq \langle \text{terme} \rangle \mid = \langle \text{Propriété} - 1 \rangle = \text{true} \text{ if } \langle \text{condition} \rangle == \text{true}.$$

Où *terme* est une partie du système représentant un état particulier à étiqueter avec la propriété "*Propriété -1*", si la condition "*condition*" est satisfaite. Plusieurs propriétés peuvent être combinées avec les opérateurs LTL.

Le model-checker de Maude est lancé avec une commande `red modelCheck`, ayant comme paramètres l’état initial (ou un état particulier d’un système), et la propriété à vérifier accompagnée de symboles LTL (globalement par exemple pour dire que la propriété sera toujours valide). Cela est fait avec la syntaxe suivante :

$$\text{red modelCheck}(\text{initial}, [] \text{Propriété}),$$

L’exécution retourne Vrai (true) si la propriété est validée, et un contre-exemple montrant la trace d’exécution dans le cas où la propriété est invalide.

La vérification formelle dans le langage de stratégie se fait exactement de la même manière que dans le langage Maude simple, avec un outil Model-Checker et la logique LTL. Cependant, la différence réside dans la façon dont le model-checking sur l’état initial est fait, parce qu’il doit prendre en charge à la fois la propriété à vérifier, et la stratégie qui contrôle l’évolution de l’état initial [Rubio et al., 2022]. Elle a la forme :

$$\text{red modelCheck}(\text{initial}, [] \text{Propriété}, 'Stratégie)$$

Où *Stratégie* est la stratégie à appliquer dans le processus de vérification.

Nous illustrons sa avec un exemple connu qui est celui des trois philosophes dans une table, où chaque philosophe doit récupérer une fourchette dans chacune de ses mains pour manger, ensuite il doit libérer les deux fourchettes dans la table. La figure 3.13 montre l’exemple complet pour illustrer la vérification dans les stratégies Maude, tiré de [Rubio et al., 2019]. Il contient quatre modules distincts :

- `fmod PHILOSOPHERS-Table` : un module fonctionnel qui permet de déclarer la table, les philosophes qui sont représentés par un identificateur et deux mains, gauche et droite contenant soit une fourchette ψ , soit vide o .
- `mod PHILOSOPHERS-DINNER` : un module système qui définit les règles de réécritures (`left`, `right` et `release`), qui sont traduites par prendre une fourchette dans une main (gauche/droite), et rendre les fourchettes dans la table respectivement.
- `smod DINNER-STRAT` : un module de stratégies qui déclare une stratégie ‘turns’, permettant aux philosophes de prendre des tours pour manger et ensuite relâcher les fourchettes.
- `mod DINNER-PREDS` : un module système qui déclare une propriété ‘eats’ afin de vérifier si un philosophe a pu manger à la fin de l’exécution.

Enfin, Nous vérifions la propriété `eats` sur les trois philosophes de deux façons : une avec le model-Checker Maude directement, et une avec la prise en charge de la stratégie `turns`. Nous pouvons constater dans la figure 3.14 qu’il y’a une situation de deadlock avec un contre-exemple d’un état qui le démontre, et que les trois philosophes ont pas pu manger. Par contre, les trois ont pu manger suite à la validation de la propriété `eats` en appliquant le Model-checker avec la stratégie `turns` dans la figure 3.15.

```

fmod PHILOSOPHERS-TABLE is *** functional module
protecting NAT . *** import a module (natural n.)
sorts Obj Phil List Table . *** declare some sorts
subsorts Obj Phil < List . *** establish subsort relations
op [_|_] : Obj Nat Obj -> Phil [ctor] . *** constructor
ops o  $\psi$  : -> Obj [ctor] .
op empty : -> List [ctor] .
op _ : List List -> List [ctor assoc id: empty] .
op <_> : List -> Table [ctor] .
var L : List . var P : Phil . *** declare a variable
eq <  $\psi$  L P > = < L P  $\psi$  > .
op initial : -> Table .
eq initial = < (o | 0 | o)  $\psi$  (o | 1 | o)  $\psi$  (o | 2 | o)  $\psi$  > .
endfm
mod PHILOSOPHERS-DINNER is *** system module
protecting PHILOSOPHERS-TABLE .
var Id : Nat .
var X : Obj .
var L : List .
rl [left] :  $\psi$  (o | Id | X) => ( $\psi$  | Id | X) .
rl [right] : (X | Id | o)  $\psi$  => (X | Id |  $\psi$ ) .
rl [left] : < (o | Id | X) L  $\psi$  > => < ( $\psi$  | Id | X) L > .
rl [release] : ( $\psi$  | Id |  $\psi$ ) =>  $\psi$  (o | Id | o)  $\psi$  .
endm

```

```

smod DINNER-STRAT is
protecting PHILOSOPHERS-DINNER .
strats turns @ Table .
sd turns(K, N) := left[Id <- K] ; right[Id <- K] ; release ;
turns(s(K) rem N, N) .
sd turns := turns(0, 3) .

```

```

mod DINNER-PREDS is
protecting PHILOSOPHERS-DINNER .
including SATISFACTION .
subsort Table < State .
op eats : Nat -> Prop [ctor] .
var Id : Nat .
vars L M : List .
eq < L ( $\psi$  | Id |  $\psi$ ) M > |= eats(Id) = true .
eq < L > |= eats(Id) = false [otherwise] .
endm

```

FIGURE 3.13: Modules de l'exemple des philosophes

```

Maude > red modelCheck(initial ,
  [] <> (eats (0) \ / eats (1) \ / eats (2))) .
ModelCheckerSymbol: Examined 4 system states.
rewrites: 43 in 4ms cpu (0ms real) (10750
rewrites/second)
result ModelCheckResult: counterexample(
{< (o | 0 | o)  $\psi$  (o | 1 | o)  $\psi$  (o | 2 | o)  $\psi$  >, 'left}
{< ( $\psi$  | 0 | o)  $\psi$  (o | 1 | o)  $\psi$  (o | 2 | o) >, 'left}
{< ( $\psi$  | 0 | o) ( $\psi$  | 1 | o)  $\psi$  (o | 2 | o) >, 'left},
{< ( $\psi$  | 0 | o) ( $\psi$  | 1 | o) ( $\psi$  | 2 | o) >, deadlock })

```

FIGURE 3.14: Vérification de l'exemple des philosophes avec le model-Checker de Maude

```

Maude > red modelCheck(initial ,
  [] (<> eats (0) /\ <> eats (1) /\ <> eats (2)), 'turns) .
rewrites: 131 in 0ms cpu (1ms real) (~ rewrites/second)
result Bool: true

```

FIGURE 3.15: Vérification des philosophes avec le model-Checker du langage des stratégies Maude

3.5 Conclusion

Dans ce Chapitre, nous avons évoqués les fondements formels adoptés dans notre travail. Dans un premier temps, nous avons introduits la théorie des systèmes bigraphiques réactifs. Nous avons d'abord décrit l'anatomie des bigraphes : leur forme graphique et leurs définitions formelles. Ensuite, on a décrit le langage à termes algébriques des bigraphes, équivalent à leur forme graphique mais contenant des limites en ce qui concerne l'expressivité. Nous avons également introduit la dynamique des systèmes réactifs bigraphiques, décrivant des règles de réactions qui définissent la sémantique d'un système. Enfin, nous avons présenté les différentes extensions des bigraphes, et les principaux outils autour de ce formalisme.

Un peut plus loin, nous avons présenté la syntaxe, les notations et la sémantique du langage Maude à travers des modules fonctionnels spécifiant des théories de réécritures, et des modules systèmes spécifiant des théories de réécritures. nous avons aussi décrit que les concepts du langage de stratégies Maude associé pour offrir une reconfiguration dynamique des théories de réécritures. Finalement, nous avons introduit les techniques de vérifications formelles et l'outil de model-checking dans Maude, et dans son langage de stratégies.

Modélisation et Analyse d'un SoS

Sommaire

4.1	Introduction	42
4.2	Travaux existants	43
4.2.1	Approches semi-formelles	43
4.2.2	Approches formelles	44
4.2.3	Synthèse	49
4.3	Principe de la solution basée ArchSoS	51
4.3.1	Syntaxe concrète d'ArchSoS	51
4.3.2	Syntaxe abstraite d'ArchSoS	52
4.3.3	Implémentation et vérification	52
4.4	Conclusion	53

4.1 Introduction

La description et la conception des SoS est une tâche complexe, dépendant de plusieurs facteurs tel-que la hiérarchie de leurs constituants, leurs évolutions indépendantes d'un côté et au sein des SoS de l'autre, et leurs interactions et coopérations. Le comportement d'un SoS est une mission que les systèmes constituants s'entraident pour achever. Chaque mission est due à un événement qui affecte un SoS, provoquant des changements dans la structure ou dans le comportement de ses constituants.

Dans le but de réduire la complexité des SoS, et de gérer leurs comportements évolutifs et imprévisibles, il est indispensable d'avoir une approche générique afin de répondre à ces défis. Une architecture logicielle est un moyen efficace pour concevoir ce type de systèmes, et pour les représenter structurellement et dynamiquement. Une architecture de référence, qui contient tous les éléments qui doivent être présents dans un SoS, ainsi que la relation entre les systèmes qui le constituent, va servir à en tirer une architecture logicielle.

Les architectures logicielles sont décrites par le biais d'un ADL (Architecture Description Language), qui contient souvent deux vues équivalentes, une graphique et visuelle, et une autre textuelle. L'aspect graphique est géré par des modèles contenant des entités et leurs relations, tandis qu'on attribue à l'aspect textuel des structures mathématiques. Un ADL conçu pour les SoS doit couvrir leurs deux aspects : leur structure hiérarchique et syntaxique, et leur sémantique comportementale et dynamique. Si l'aspect structurel est couvert par des modèles, il doit aussi avoir un mécanisme qui permet aux SoS, et à ses constituants d'évoluer.

Ce chapitre a pour but de servir comme un chapitre introductif qui va offrir une vision de l'ensemble des travaux présentés dans cette thèse, notamment en terme de formalisation des

SoS, leur implémentation et leur exécution et vérification. Par la suite, nous présentons une synthèse de ces travaux, avant de décrire après le principe de notre solution et ses différentes phases.

4.2 Travaux existants

Plusieurs travaux existants dans la littérature tentent de répondre aux besoins et aux défis complexes des SoS, où chaque travail est lié à son domaine. Dans le cadre de notre thèse, nous nous concentrons sur les travaux qui visent à abstraire les SoS, en considérant leurs composants évolutifs et leurs interactions.

Nous catégorisons nos travaux selon deux points de vues importants : Le premier point regroupe l'ensemble des approches qui ne sont pas entièrement formelles (semi-formelles), alors que le deuxième point regroupe les approches qui sont bien formalisées. Cette catégorisation se base sur la capacité de modélisation, de description et expressivité des approches, afin de discuter le besoin de formalisation. Nous synthétisons la discussion des approches par une comparaison dans la section 4.2.3, pour en déduire l'utilisation d'un formalisme qui sera le mieux adapté pour les SoS.

4.2.1 Approches semi-formelles

Cette catégorie regroupe majoritairement les approches qui sont basées sur des représentations UML et SysML, nous indiquons que SysML a été plus utilisé que UML dans cette catégorie vu qu'il est jugé plus adapté aux systèmes complexes et distribués. UML a été utilisé par quelques auteurs, nous citons [Osmundson et al., 2006] qui a utilisé les spécifications UML afin de traiter l'interopérabilité des informations dans les SoS. Dans le même aspect, une architecture qui modélise les SoS a été définie par [Axelsson et al., 2019], sous forme de diagrammes de classes UML définissant les composants et les fonctionnalités du système.

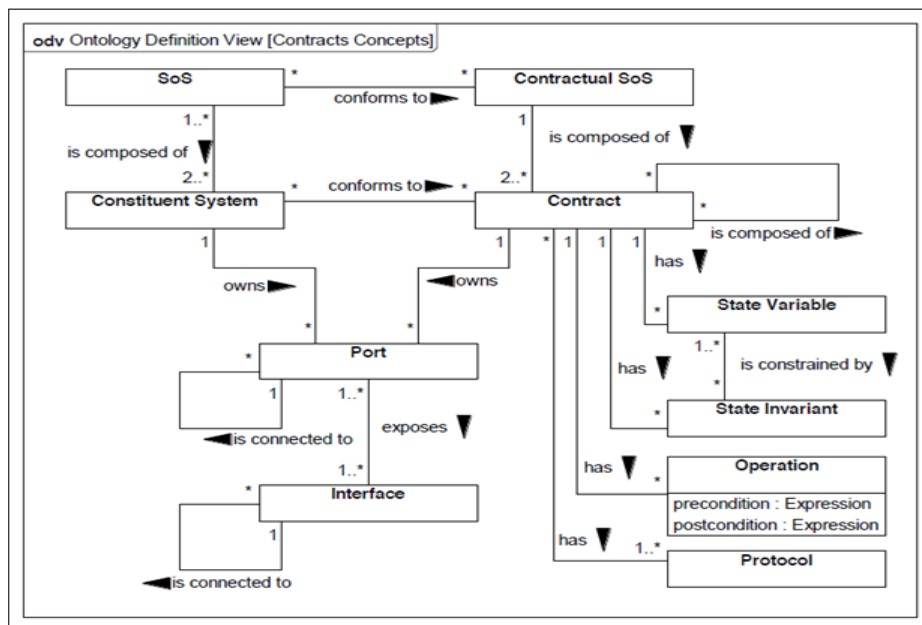


FIGURE 4.1: Modèle SoS à base de contrats [Bryans et al., 2014]

Les auteurs dans [Bryans et al., 2014] utilisent des profils SysML pour définir une spécification contractuelle des interfaces des systèmes constituant un SoS, appliquée sur une étude

de cas du domaine audio-visuelle. La figure 4.1 montre une vue d'un SoS et de ses constituants, ainsi que la notion d'un SoS contractuel qui regroupe la collection de contrats qu'il doit conformer.

D'autres chercheurs ont aussi opté pour SysML afin de décrire les SoS, notamment [Huynh and Osmundson, 2006] [Lane and Bohn, 2013] [Hause, 2014] [Hu et al., 2014] [Mori et al., 2016] et [Dahmann et al., 2017]. Ils ont utilisé les différents diagrammes consacrés à SysML, comme les diagrammes de « blocks-definition » dans le but de définir la structure des systèmes et leurs interactions, et les diagrammes de séquence pour définir des scénarios qui occurrent dans les systèmes constituants.

4.2.2 Approches formelles

Les approches formelles sont renforcées par des formalismes et techniques permettant, à l'aide d'une logique mathématique, de raisonner d'une façon plus rigoureuse et plus forte sur ce type de systèmes assez complexes.

Les auteurs dans [Woodcock et al., 2012] présentent CML (COMPASS Modelling Language) comme un langage formel spécifiquement conçu pour la modélisation et l'analyse des SoS. Il est basé sur les formalismes CSP (Constraint Satisfaction Problem) [Brookes et al., 1984] et Circus [Woodcock and Cavalcanti, 2002]. Il est présenté à l'aide d'un exemple d'un SoS des centrales téléphoniques indépendantes.

L'approche définie par [Nielsen and Larsen, 2012] utilise VDM-RT (Vienna Development Method Real-Time), une méthode formelle orientée objet qui permet de simuler la conception et le comportement des SoS. Nous distinguons dans cette approche trois types de reconfigurations : (1) ajout et suppression des systèmes constituants, (2) ajout et suppression des canaux de communications entre ces systèmes, (3) changement dans la topologie du réseau des systèmes, qui indique quel système va être connecté à quel canal. Cette approche a été appliquée sur une étude cas de gestion de voiture, appelée VeMo (Vehicle Monitoring). La figure 4.2 montre comment une classe d'un constituant est définie avec VDM équipée d'une application de feux de routes.

```

system VeMo
instance variables
const1 : Constituent := new Constituent(1E6);
t11 : TrafficLight := new TrafficLight(999,
                                new Position(20, -70));
public ve2consti : inmap VeMoEntity to
                                Constituent := {|->};

operations
public VeMo: () ==> VeMo
VeMo() ==(
const1.deploy(t11);
ve2cons:= VeMo`ve2consti munion
{t11 |-> const1} );
end VeMo

```

FIGURE 4.2: Classe système qui définit un constituant d'un SoS [Nielsen and Larsen, 2012]

Les agents sont utilisés dans [Muller IV, 2016] pour définir la coévolution dans les systèmes. Le comportement coévolutif est le cas où deux agents ou plus cherchent à développer

des systèmes avec des fonctionnalités spécifiques, et des contre-mesures pour vaincre des systèmes adverses. Chaque fois un agent développe de nouvelles technologies pour améliorer ses performances (par exemple, fiabilité, précision, coût), l'autre agent apporte des modifications à ses systèmes en réponse à atténuer tout nouvel avantage de l'ancien. L'approche consiste à simuler l'architecture d'un SoS de lutte contre la contrefaçon, dont les agents ont eu des comportements coévolutifs avec les architectures physiques des systèmes de contrebandiers. Les agents représentent les systèmes constituant un SoS, et ils communiquent avec des interactions agent-agent. Ces agents sont gouvernés par des règles qui délimitent leurs comportements, mais sont aussi dotés de propriétés qui influencent dynamiquement leurs comportements.

Les auteurs dans [Stary and Wachholder, 2016] définissent une représentation basée bi-graphes qui suggère une perspective SoS appliquée sur des systèmes d'aide à l'apprentissage polyvalents. Cette approche se concentre fortement sur les relations d'interactions entre les systèmes. Ces relations sont définies en termes de compositions horizontales (figure 4.3) et verticales (figure 4.4) des bigraphes, permettant par exemple à un étudiant d'avoir un enseignant, ou bien d'apprendre un cours respectivement.

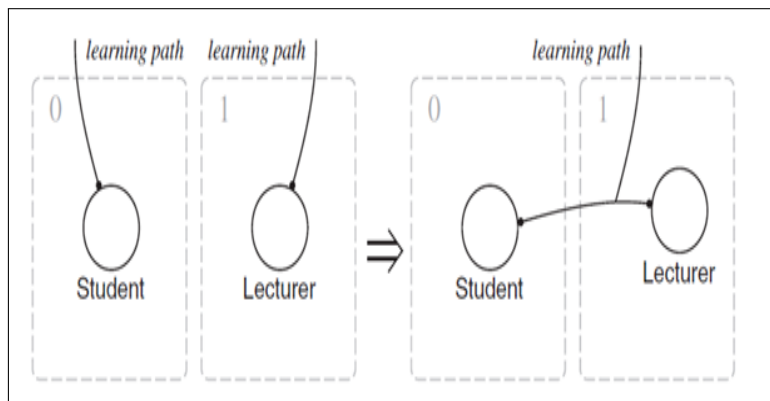


FIGURE 4.3: Composition Horizontale dans [Stary and Wachholder, 2016]

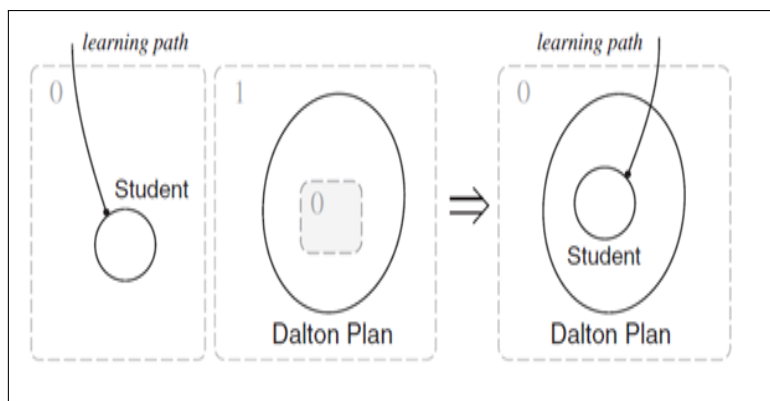


FIGURE 4.4: Composition Verticale dans [Stary and Wachholder, 2016]

[Gassara et al., 2017] proposent B3MS, une méthodologie basée BRS pour une modélisation multi-échelle. Cette approche suit un processus de raffinement entre les échelles. L'aspect dynamique est traité par des règles qui permettent une mobilité au sein des systèmes constituant. B3MS est appliquée sur une étude de cas de bâtiments intelligents. Les mêmes auteurs dans [Gassara et al., 2019] ont défini une solution pour exécuter des bigraphes en les transformant vers des graphes via un matching en spécifiant un foncteur. Les bigraphes, décrits depuis un outil appelé « Big Red », ainsi que leurs règles de réactions, sont encodées en graphes afin de les exécuter. Les résultats sont retournés ensuite afin d'être affichés en forme de bigraphes.

La figure 4.5 montre ce processus appliqué par un outil nommé BiGMTE (Bigraph matching and transformation engine).

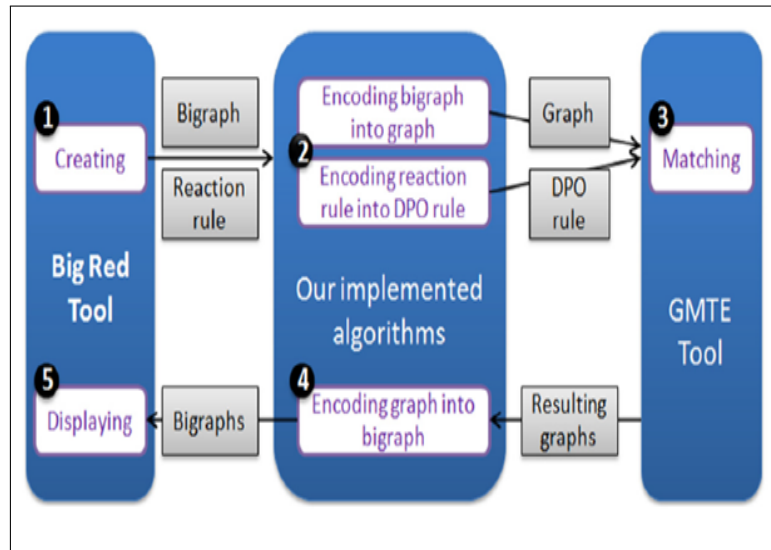


FIGURE 4.5: Architecture de l'outil BiGMTE [Gassara et al., 2019]

Les auteurs dans [Wang et al., 2015] définissent des réseaux de Pétri colorés CPN (Colored Petri Nets) [Jensen, 1987] comme formalisme pour simuler des structures définies dans une méthodologie de processus objets (ou OPM : Object process methodology) [Dori, 2011]. Les modèles décrits par OPM sont faciles à comprendre, et bien intuitifs. Ils sont convertis en modèles CPN dans le but d'analyser les aspects comportementaux d'un SoS, notamment l'aspect de performance. L'évolution est effectuée via des transitions d'états sémantiques.

Similairement, les CPN sont utilisés dans [Akhtar and Khan, 2019] pour la spécification de l'architecture et la vérification formelle du SoS de surveillance intelligente des inondations. Le modèle formel de cet SoS est spécifié pour garantir les propriétés de sûreté et de vivacité avec une vérification formelle en utilisant les LTS (Labelled Transition System) [Bert and Cave, 2000]. LTS effectue une analyse compositionnelle pour effectuer un model-checking en recherchant, de manière exhaustive, les violations des propriétés requises. Le SoS décrit recueille des informations auprès des prévisions météorologiques des observateurs de l'inondation. Ces informations sont traitées puis mises à la disposition des clients sous forme d'alertes. La figure 4.6 illustre le modèle de CPN pour un SoS de surveillance intelligente des inondations.

Les graphes sont utilisés dans [Derhamy et al., 2019], pour effectuer des requêtes afin d'extraire dynamiquement les informations de leur source, au sein d'un réseau IIoT (IoT Industriels ou Industrial IoT). Durant l'exécution de l'IIoT, les requêtes dynamiques pour extraire les données contextuelles construisent au fur et à mesure les constituants d'un SoS sous forme de services dans une architecture orientée services [Perrey and Lycett, 2003]. En utilisant la SOA et les SoS, il est possible de visualiser l'application IIoT comme des ensembles d'éléments de graphe.

Une approche à base d'ADL est définie dans [Oquendo and Legay, 2015] [Oquendo, 2016]. L'ADL est connu sous le nom de SoSADL, et se base sur le formalisme de " π -calculs. Il a une syntaxe abstraite formelle suivi d'une sémantique sous forme d'un système de transition, exprimée par des règles de transition. Cet ADL généralise le π -calculs avec la notion de calcul d'information partielle, basée sur le paradigme de contraintes concurrentes CCP (Concurrent Constraint Paradigm) [Olarde et al., 2013]. Ce dernier est muni d'un ensemble de primitives qui étendent le π -calculs sous forme d'actions. Par exemple, l'opérateur "tell" permet à un processus d'envoyer ses informations à son environnement, et l'opérateur "ask" sert à demander des informations de son environnement, qui vont affecter le comportement du processus. La figure

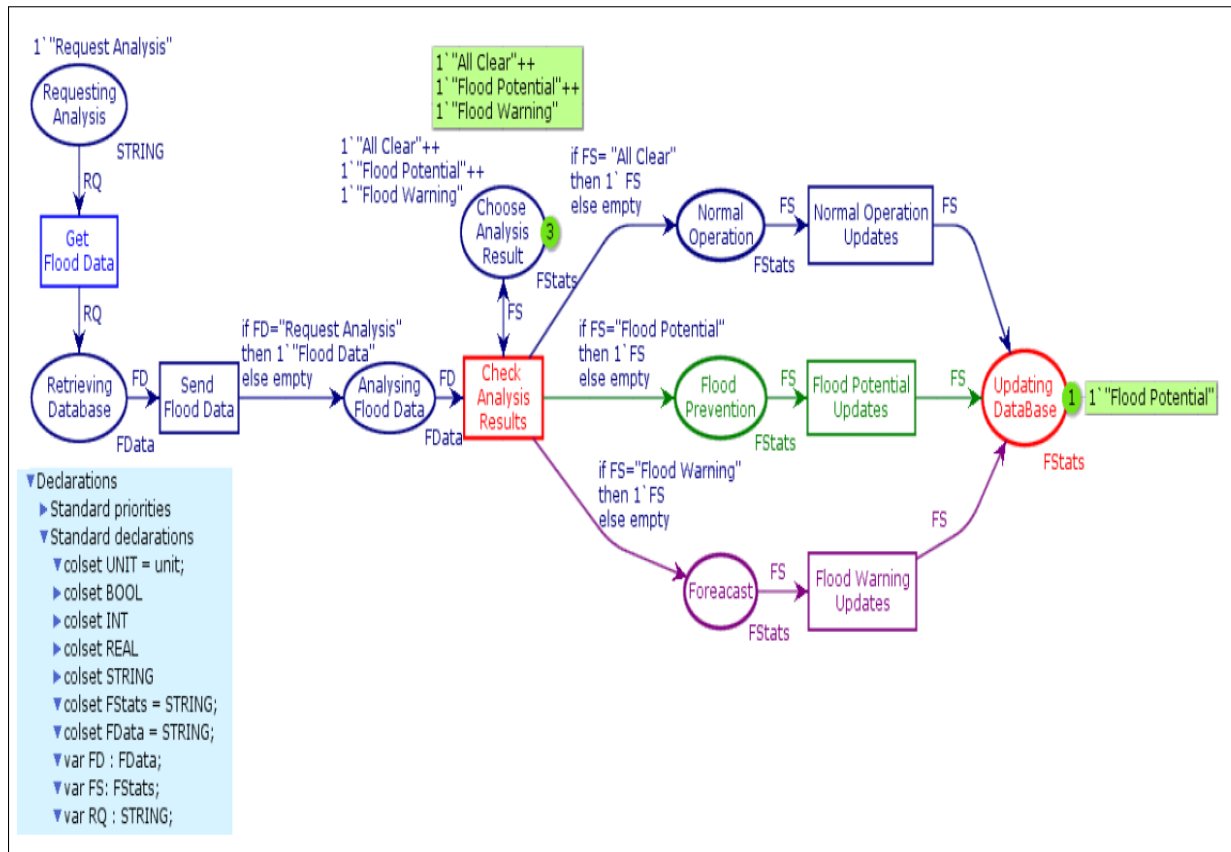


FIGURE 4.6: Modèle CPN pour l'analyse d'un SoS de surveillance des inondations [Akhtar and Khan, 2019].

4.7 résume la syntaxe des SoS dans SoSADL, où chaque SoS est décrit par un comportement, doté d'une séquence d'actions. Cette syntaxe peut évoluer grâce à une sémantique basée sur un système de transition, qui fait passer le SoS d'un état vers un autre. Cet ADL a été appliqué sur un SoS de surveillance des inondations. Les mêmes auteurs dans [Oquendo, 2018] ont utilisé SoSADL pour définir une architecture auto-organisée pour les SoS dans un contexte d'Internet des Objets. La particularité dans cette approche, est que des comportements émergents des SoS sont provoqués au moment de l'exécution.

Les auteurs dans [Neto, 2016] offrent une extension de l'architecture logicielle décrite par l'ADL précédent, en appliquant le formalisme de DEVS (Discrete Event System Specification) décrit par [Zeigler et al., 2013] pour simuler les comportements des SoS. Un mapping entre DEVS et SoSADL est défini avec une transformation de modèles pour bénéficier à la fois de l'architecture logicielle décrite par SoSADL, et l'expressivité logicielle offerte par DEVS. La figure 4.8 montre ce mapping, en offrant à chaque élément dans SoSADL son équivalent dans DEVS.

Dans le même sens, les auteurs dans [Silva et al., 2020] ont adapté une approche dédiée à la vérification des propriétés liées aux missions des SoS dans une conception architecturale. Le langage de modélisation de missions mKAOS [Silva et al., 2015] a été utilisé pour encapsuler les missions et les comportements émergents des SoS. Il a été renforcé par l'outil DynBLTL [Quilbeuf et al., 2016] comme un moyen de vérification formelle. Cet outil étend la logique temporelle bornée ou BTL (Bounded Temporal Logic), avec les mêmes opérateurs que LTL. Cet outil qui comprend, outre les opérateurs logiques traditionnels (et, ou, non, implique, etc.), des opérateurs de temps. Cette approche a été implémentée pour vérifier les modèles de l'ADL SoSADL.

Finalement, [Chaabane et al., 2019] propose une approche basée sur la norme «ISO / CEI /

Abstract syntax of π-Calculus for SoS	
constrainedBehavior ::= behavior ₁	
restriction ₁ . constrainedBehavior ₁	-- <i>Constrained Behavior</i>
behavior name ₁ (value ₀ ..., value _n) is { behavior ₁ }	-- <i>Definition</i>
constraint name ₁ is { constraint ₁ }	-- <i>Constraint Definition</i>
compose { constrainedBehavior ₀ ... and constrainedBehavior _n }	
behavior ::= baseBehavior ₁	
restriction ₁ . behavior ₁	-- <i>Unconstrained Behavior</i>
repeat { behavior ₁ }	-- <i>Repeat</i>
apply name ₁ (value ₀ ..., value _n)	-- <i>Application</i>
compose { behavior ₀ ... and behavior _n }	-- <i>Composition</i>
baseBehavior ::= action ₁ . behavior ₁	-- <i>Sequence</i>
choose { action ₀ . baseBehavior ₀	-- <i>Choice</i>
or action ₁ . baseBehavior ₁ ... or action _n . baseBehavior _n }	
if constraint ₁ then { baseBehavior ₁ } else { baseBehavior ₂ }	
done	-- <i>Termination</i>
action ::= baseAction ₁	
tell constraint ₁	-- <i>Tell</i>
unsaid constraint ₁	-- <i>Unsaid</i>
check constraint ₁	-- <i>Check</i>
ask constraint ₁	-- <i>Ask</i>
baseAction ::= via connection ₁ send value ₀	-- <i>Output</i>
via connection ₁ receive name ₀ : type ₀	-- <i>Input</i>
unobservable	-- <i>Unobservable</i>
connection ::= connection name ₁	
restriction ::= value name ₁ = value ₀ connection ₁	

FIGURE 4.7: Syntaxe abstraite de SoSADL. [Oquendo, 2016a]

SoSADL	SES/DEVS
Architecture	Coupled Model
Behavior	State Diagram
Connection	DEVS Port
Coalition	Coupled Model
Data Type	Data Type
Function	DEVS Function
Mediator	Atomic Model
SoS	Coupled Model
System	Atomic Model

FIGURE 4.8: Mapping entre SoSADL et DEVS [Neto, 2016]

IEEE 42010 : Ingénierie des systèmes et des logiciels - Description de l'architecture » pour décrire les architectures logicielles des SoS. Il propose des améliorations pour que cette norme soit adéquate avec les caractéristiques SoS. Le travail est modélisé par des graphes multi-étiquettes en utilisant l'outil GMTE (Graph Matching and Transformation Engine), et suivi d'une étude qualitative basé sur la notion Goal-Question-Metric (GQM) [Caldiera and Rombach, 1994] avec un groupe de discussion pour évaluer l'efficacité du travail, qui avait un SoS de ville intelligente comme une étude de cas.

4.2.3 Synthèse

Avant d'offrir une synthèse des travaux discutés et évoquer leurs apports ainsi que leurs limites, nous avons jugé important que les approches de modélisation et de conception des SoS utilisées dans ces travaux puissent répondre à un ensemble de critères tirés des défis des SoS. Nous regroupons ces critères en trois catégories principales pour la description architecturale des SoS comme indiqués sur le tableau 4.1, que nous détaillons ci-dessous :

- Description Syntaxique des SoS : la capacité de définir de la structure d'un SoS, en décrivant la hiérarchie des systèmes constituant un SoS, et leurs interactions en termes de liens.
- Description Sémantique des SoS : une prise en charge du comportement des SoS, cette catégorie contient les missions qu'un SoS est censé accomplir, les évolutions que les constituants d'un SoS peuvent subir pour atteindre ces missions, et les contraintes qui conditionnent cette évolution.
- Exécution et Vérification formelle : la troisième catégorie décrit l'aspect d'implémentation, en terme d'exécution à travers des méthodes et des outils adéquats, ainsi qu'en terme de reconfiguration dynamique d'un SoS pour s'adapter pour répondre à plusieurs événements.

Nous constatons que aucune des approches formelles ou semi-formelles ne fournisse une méthodologie complète qui permet de prendre en considération tous les critères de comparaison adoptés. En fait, Les approches semi-formelles SysML (et un peu moins UML) arrivent à décrire l'aspect structurel et syntaxique des SoS, notamment la hiérarchie des systèmes via des compositions de classes et d'entités. Les liens sont aussi bien représentés avec des relations entre ces classes. Cependant, les missions des SoS ne sont pas exprimées avec ces approches, et l'évolution est partiellement traitée avec des diagrammes de séquences et d'activités, qui ne permettent pas réellement d'avoir une sémantique pour la dynamique des SoS. Les approches formelles sont divergents, nous notons que :

- Les approches basées BRS prennent en charge les aspects structurels de manière appropriée grâce aux bigraphes en offrant une vue visuelle et graphique avec une forte abstraction. La hiérarchie des systèmes dans un SoS est définie avec la fonction de parenté des nœuds, alors que les relations entre les systèmes sont définies avec des hyper-arcs entre les nœuds. La notion de règles de réactions des BRS décrit l'évolution dynamique des SoS. Cependant ces approches ne traitent pas réellement les missions d'un SoS, ni les contraintes qui contrôlent leur évolution.
- Les approches basées ADL ne sont pas aussi représentatives que les bigraphes structurellement, mais elles sont orientées vers la description des relations dynamiques entre les systèmes, et comment ces derniers peuvent évoluer, grâce à la description architecturale réutilisable des configurations d'un SoS.
- Les autres approches comme CML, VDM ou les réseaux de Pétri sont aussi basées sur la description des relations entre les systèmes constituant un SoS et leurs évolutions, et ils ont de plus des outils appropriés pour effectuer des simulations et faire dérouler des scénarios concrets.

TABLE 4.1: Travaux existants pour la modélisation des SoS

Approche	Formalisme	Description Syntaxique des SoS		Description Sémantique des SoS			Exécution et Vérification formelle	
		Hiérarchie	Liens	Missions	Évolution	Contraintes	Exécution	Reconfiguration Dynamique
[Osmundson et al., 2006]	UML	+	+	-	+	-	-	-
[Axelsson et al., 2019]		++	+	-	+	-	-	-
[Huynh and Osmundson, 2006] [Lane and Bohn, 2013] [Hause, 2014] [Hu et al., 2014] [Bryans et al., 2014] [Mori et al., 2016] [Dahmann et al., 2017] et [Antul et al., 2018]	SYSML	++	++	-	+	-	-	-
[Woodcock et al., 2012]	CML	-	+	-	+	-	-	-
[Nielsen and Larsen, 2012]	VDM-RT	+	++	+	+++	-	+ : VDM-RT	-
[Muller IV, 2016]	Multi-Agents	-	-	++	++	-	-	+
[Stary and Wachholder, 2016]	Bigraphes	+++	++	+	+	-	-	-
[Gassara et al., 2017]		+++	++	-	+	-	-	-
[Gassara et al., 2019]		+++	++	-	+	-	+ : Outil BiGMTE	-
[Wang et al., 2015]	CPN	-	+	+	+	-	+ : Simulation CPN	-
[Akhtar and Khan, 2019]		+	+	-	++	+	+ : Simulation CPN	-
[Derhamy et al., 2019]	Modèle de graphes	+	++	+	+	-	-	-
[Axelsson, 2020]	Ontologies	+	+	+	+	+	-	-
[Nilsson et al., 2020]		+	++	-	+	-	-	-
[Oquendo and Legay, 2015] [Oquendo, 2016]	SoSADL	-	+++	+	+	++	-	-
[Neto, 2016]	SoSADL - DEVS	+	+++	+	++	+	+ : DEVS	-
[Neto, 2016]	SosADL - SoS d'auto-organisation	-	+++	+	++	++	+ : -	+
[Chaabane et al., 2019]	ADL de graphes multi-étiquettes	++	++	-	++	-	+ : GMTE	-
[Silva et al., 2020]	SoSADL - mKAOS	-	+++	+++	+	+	+ : Dyn-BLTL	-

+++ : Fortement supporté, ++ : Supporté, + : Partiellement supporté, - : Non-supporté

4.3 Principe de la solution basée ArchSoS

En vue de répondre aux critères identifiés, et suite à l'analyse des différents travaux existants sur la modélisation des SoS, nous proposons une solution basée sur la définition d'un ADL nommé ArchSoS, dédié à la description architecturale des SoS. Nos contributions essentielles sont présentées en trois phases de façon incrémentale :

- 1- Description des SoS en utilisant une architecture de référence à base de méta-modèles servant en plus à définir une syntaxe concrète pour ArchSoS.
- 2- Spécification formelle des deux aspects structurel et comportementale des SoS dans ArchSoS en utilisant les BRS qui offrent deux vues, une vue algébrique et une vue graphique.
- 3- Implémentation du comportement des SoS dans Maude, une vérification qualitative de leurs propriétés et leur possibilité de reconfiguration dynamique en utilisant les stratégies Maude sont aussi évoquées dans cette phase.

La figure 4.9 montre une vue globale de notre approche, chaque phase sera détaillée séparément dans ce qui suit.

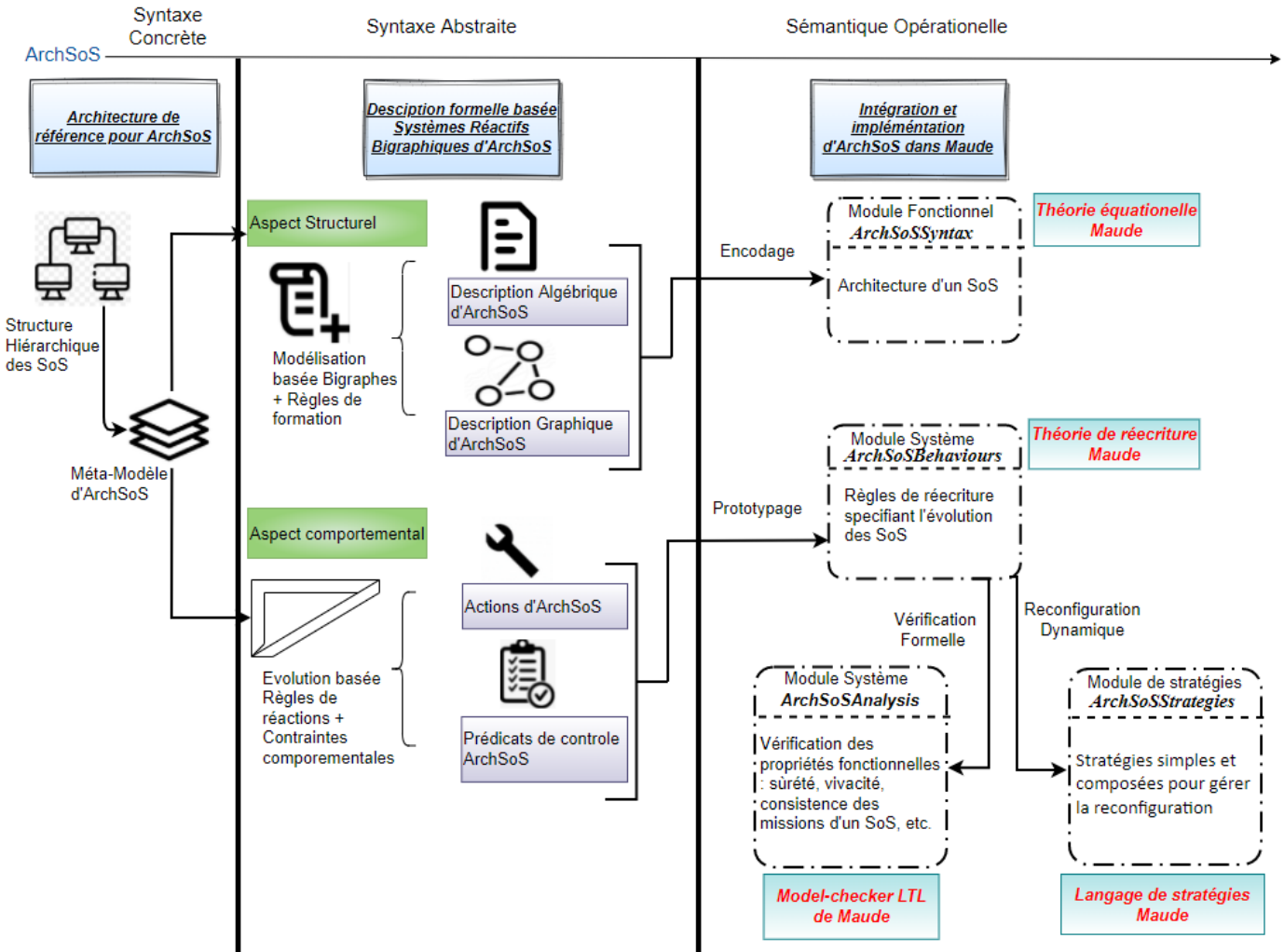


FIGURE 4.9: Vue globale de notre approche

4.3.1 Syntaxe concrète d'ArchSoS

Cette phase fournit une description d'une syntaxe concrète des SoS selon notre ADL ArchSoS en utilisant un méta-modèle résultant d'une approche MDA (Model Driven Architecture [Blanc

and Salvatori, 2011]. Ce méta-modèle est basé sur la norme ISO/IEC/IEEE 42010 adaptée pour décrire des architectures logicielles des SoS. Selon cette norme, une architecture logicielle englobe ce qui est essentiel dans un système par rapport à son environnement, ses constituants et comment ils sont censés interagir. À partir de cela, cette syntaxe concrète va contenir les éléments figurant dans un SoS, en termes de hiérarchie des systèmes constituants, leurs fonctionnalités, les missions qu'ils collaborent pour achever, les contraintes qu'ils sont obligés de respecter, et les événements affectant un SoS en provoquant les interactions de ces constituants, sous forme de liens et fonctionnalités combinés.

4.3.2 Syntaxe abstraite d'ArchSoS

Cette phase sert à formellement définir ArchSoS, introduisant ainsi une syntaxe abstraite. Elle est divisée en deux étapes principales :

- La première vise à associer une sémantique bien définie à tous les concepts de ce langage en se basant sur le formalisme des BRS. La structure est définie par des bigraphes, en mettant les points sur la localité des systèmes constituants un SoS, et la connectivité entre eux. Cela inclut la définition des bigraphes selon deux formes : (1) la forme graphique montrant les nœuds du bigraphe comme les entités d'un SoS, et les hyper-arcs délimitant la connectivité, (2) la forme algébrique ayant un aspect textuel. Les bigraphes ont un ensemble de règles de formations qui contraignent la construction des SoS.
- La deuxième partie cible le côté évolutif et comportemental des SoS. Plus précisément, elle sert à montrer comment les SoS et leurs constituants décrits par ArchSoS ont la capacité d'interagir et de répondre à des événements pour achever des missions de manière formelle. Ceci est réalisé en deux étapes :
 - 1- Recourir aux règles de réaction des BRS pour définir un ensemble d'actions. Ces actions ont pour but de faire évoluer le SoS d'un état initial i vers un autre état j , autrement dit faire évoluer le bigraphe initial d'un SoS vers un autre bigraphe. Les actions sont divisées en actions de mobilités changeant la structure des constituants d'un SoS, des actions de liens permettant aux systèmes d'interagir par rapport à un événement, et des actions de rôles (ou de fonctionnalités) permettant aux constituants de combiner leurs rôles pour avoir un nouveau rôle spécifiant une mission du SoS.
 - 2- Attribuer des prédicats de contrôle ϕ_i pour les actions définies précédemment. Ces prédicats assurent le bon fonctionnement des actions d'évolution, par exemple, on ne peut pas supprimer un lien qui n'existe pas, donc nous aurons un (ou plusieurs) prédicat qui va vérifier la présence d'un lien avant d'appliquer une règle qui peut le supprimer.

4.3.3 Implémentation et vérification

Les outils qui tournent autour des bigraphes ont été conçus dans un contexte bien étroit, ils ont un manque en termes d'expressivité, de vérification formelle et de performance. Pour cela, nous avons opté pour l'utilisation du langage Maude comme étant une alternative plus appropriée. Le modèle formel à base de BRS, issu de la phase précédente, est implémenté et interprété dans Maude afin d'offrir une sémantique opérationnelle exécutable, depuis laquelle on peut analyser les comportements des SoS dans ArchSoS. Cette phase est réalisable selon quatre étapes :

- 1- **Spécification syntaxique structurelle** : La première partie consiste à implémenter le modèle Bigraphique d'ArchSoS à l'aide d'un module fonctionnel Maude. Il consiste

à spécifier la structure d'un SoS dans une théorie équationnelle, en termes de types de constructions (sorts, opérateurs, etc.) pour chaque élément d'ArchSoS. Il est à noter que les règles de formation attribuées aux bigraphes sont conservées dans la spécification Maude.

- 2- **Spécification sémantique évolutive** : Elle sert à spécifier la sémantique opérationnelle des SoS dans ArchSoS grâce à un module système Maude. Les règles de réaction bigraphiques sont adaptées et enrichies autant que règles de réécritures conditionnées par des prédicats (sous forme d'équations Maude) pour définir une exécution autonome des comportements des SoS. Cette exécution permettra aux structures définies par le module fonctionnel d'évoluer et de passer d'un état global du système à un autre.
- 3- **Analyse et Vérification comportementale** : Cette étape est prise en charge par un module système Maude, qui définit des propriétés Maude dans le but d'effectuer une vérification formelle des comportements des SoS dans ArchSoS. Cette vérification assure que l'exécution autonome des règles de réécritures finit par aboutir à des comportements désirables, en satisfaisant, tout au long de l'exécution, des propriétés décrites. Ces propriétés permettent de : (1) Vérifier la consistance des missions d'un SoS, c'est à dire sa capacité d'aboutir ou non a des missions données, (2) vérifier des propriétés concernant les contraintes comportementales d'un SoS, ainsi que les propriétés de sûreté (quelque chose de mauvais n'arrive jamais), de vivacité (quelque chose de bon finit par arriver). La partie de la vérification s'appuie sur le Model-checker inclus dans Maude pour vérifier les comportements des SoS, et sur la logique temporelle linéaire (LTL) afin de définir ces propriétés.
- 4- **Reconfiguration dynamique** : Les SoS doivent pouvoir répondre à plusieurs événements en même temps, tout en adaptant leurs comportements selon ces événements distincts. Le moteur de réécriture de Maude précède souvent à une exécution séquentielle, donc si nous avons plusieurs événements, le système va répondre un par un, et nous pouvons avoir des cas dans lesquels il existe des conflits où un événement influence ou bloque un autre.
Dans les SoS critiques, où le temps de réponse est court, ces contraintes deviennent primordiales. Pour répondre a ce besoin, une reconfiguration dynamique s'avère nécessaire afin de délimiter des scénarios d'exécution parallèles, où chaque scénario s'occupera d'un événement particulier sans pour autant influencer ou affecter l'autre. Cela induira des missions différentes et parallèles pour le même SoS. Cette partie est prise en charge par un module de stratégies, qui est un module particulier de Maude basé sur le langage de stratégies Maude. Il a pour but de définir une stratégie à chaque mission, c'est à dire un ensemble particulier des règles à appliquer pour faire évoluer le système afin d'accomplir une mission.

4.4 Conclusion

Dans ce chapitre, nous avons étudiés et présentés plusieurs travaux liés à la spécification architecturale et à la modélisation des SoS. Dans un premier temps, nous avons discuté les approches semi-formelles autour des SoS. Ensuite, nous avons donné un aperçu d'un bon nombre d'approches et de modèles formels, ayant proposé des méthodologies et des solutions autour des spécifications de ce type de systèmes. Cet ensemble de travaux de la littérature a été ensuite évalué selon plusieurs critères, afin de mieux situer nos contributions, en dégagant leurs apports et leurs lacunes. Enfin, nous avons introduit la méthodologie proposée en expliquant le principe générale ainsi que les différentes phases de conception d'un ADL nommé ArchSoS, qui sert à décrire les architectures logicielles des SoS.

Syntaxes d'ArchSoS

Sommaire

5.1	Introduction	55
5.2	Concepts de base	56
5.3	Syntaxe concrète	59
5.4	Syntaxe abstraite basée BRS	64
	5.4.1 Aspect structurel	64
	5.4.2 Aspect comportemental	67
5.5	Conclusion	71

5.1 Introduction

La description architecturale des SoS est un aspect important pour délimiter la structure de ces systèmes, et la nature de leurs comportements. Nous avons défini, dans le chapitre précédent, les phases de conception d'une architecture logicielle dédiée aux SoS, par le biais de la définition d'un ADL spécifique à la représentation des architectures de ces systèmes, nommé ArchSoS.

Cependant, nous ne savons pas quels éléments doivent être présents dans les SoS, et comment ces éléments peuvent être interconnectés avec des associations ou des compositions particulières. Il est donc indispensable d'avoir une architecture de référence, qui servira comme une base pour instancier les architectures des SoS avec ArchSoS. Elle doit être expressive et efficace pour encapsuler à la fois les éléments structurels des SoS, et leurs comportements. Ces derniers dépendent de plusieurs facteurs assez complexes, tels que les événements qui affectent un SoS et ses constituants, les liaisons entre ces constituants, et la combinaison de leurs fonctionnalités pour en tirer de nouvelles fonctionnalités, représentant les missions des SoS.

Pour cela, un haut niveau d'abstraction est nécessaire afin de décrire ses systèmes dans ArchSoS. La structure d'un système établie doit être uniforme et correcte-par-construction, c'est à dire qu'il doit exister une spécification qui dénotent des règles et des contraintes sur la façon dont les constituants d'un SoS sont construits hiérarchiquement, et sur les relations entre eux.

En adaptant une méthodologie MDA (Model Driven Architecture) [Blanc and Salvatori, 2011], nous définissons la syntaxe concrète d'ArchSoS comme un méta-modèle normalisé par la norme ISO/IEC/IEEE 42010. Ce méta-modèle servira comme une architecture de référence pour ArchSoS, à partir de laquelle nous attribuons une formalisation, une implémentation et une vérification à ce langage.

Dans ce chapitre, nous nous occupons des deux premières phases de notre approche : (1) Nous définissons dans la section 5.2 les concepts de l'approche MDA et la norme ISO/IEC/IEEE

42010, afin d'introduire la syntaxe concrète d'ArchSoS dans la section 5.3 (2) Nous présentons dans la section 5.4 une approche de modélisation formelle pour les SoS, basée sur le formalisme des Systèmes Reactifs Bigraphiques (BRS), permettant de modéliser une syntaxe abstraite des SoS, qui prend en charge à la fois un aspect architectural et structurel des SoS dans ArchSoS, et un aspect comportemental pour illustrer l'évolution des SoS. Le choix de ce formalisme est justifié par la capacité des BRS de représenter la localité et la connectivité des systèmes, qui le diffère d'autres formalismes tels que les réseaux de Petri, la méthode B ou l'algèbre de processus.

5.2 Concepts de base

Afin de mieux décrire la structure d'ArchSoS, et les éléments des SoS qui doivent être présent dans cette structure, nous recurons à une méthodologie basée-modèles. Cela permet d'identifier ces éléments, leurs relations hiérarchiques et les liens qui leur permettent de communiquer, ce qui facilitera la compréhension des descriptions des SoS dans ArchSoS.

Approche MDA

OMG (Object Management Group) a défini une ingénierie guidée par les modèles MDA [MDA, 2008] afin de séparer les objectifs d'un système de sa plateforme utilisée, et cela en utilisant des modèles spécifiant une architecture bien précise. Cette ingénierie sert comme un guide pour structurer une architecture en se basant sur des langages de modélisation comme UML (Unified modeling language), Meta Object Facility (MOF), ou Common Warehouse Meta model (CWM). Ces langages servent comme des normes qui offrent une infrastructure de base pour la méthode MDA. Dans notre approche, MDA est utilisée pour définir les modèles décrivant un ADL dédié aux SoS.

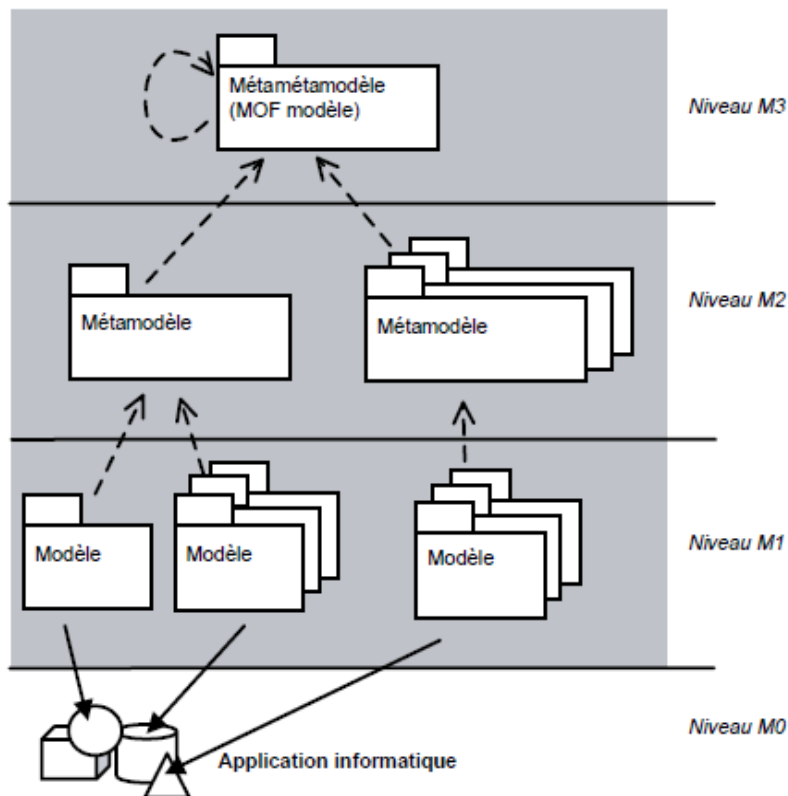


FIGURE 5.1: Architecture MDA [Blanc and Salvatori, 2011]

Les modèles jouent un rôle très important dans la description d'un ADL, ils permettent de définir des éléments de base qui doivent être présent dans un système. Chaque modèle a une syntaxe qui sert à son élaboration. MDA a défini une architecture à quatre niveaux qui pose les bases des relations qui existent entre les entités à modéliser, modèles, et leurs structures de modélisation (voir figure 5.1) :

- Entités à modéliser : qui correspondent plus souvent aux applications informatiques ou systèmes à modéliser. Dans notre cas, il s'agit des SoS et leurs constituants.
- Modèles : sont des représentations proches de la réalité qui contiennent des niveaux d'abstraction des informations utiles à la production ou évolution d'une application informatique.
- Méta-modèle : définit les entités du modèle et les propriétés de leurs connexions et leurs règles de cohérence. Chaque modèle doit être conforme a son méta-modèle.
- Structure MOF (aussi appelé méta-méta-modèle) : permet d'exprimer des méta-modèles (formalismes de modélisation). Tout l'intérêt de MOF est de définir des mécanismes génériques sur les méta-modèles à l'aide d'une structuration commune. MOF utilise des diagrammes de classes pour représenter les méta-modèles.

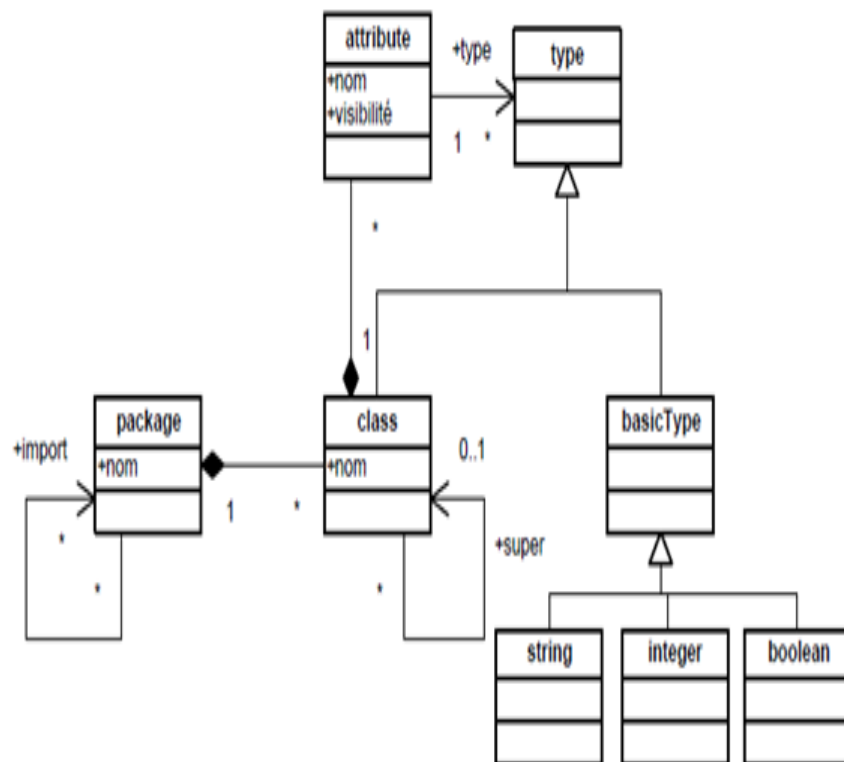


FIGURE 5.2: Syntaxe du méta-modèle d'un diagramme de classe [Blanc and Salvatori, 2011]

La figure 5.2 montre un exemple de méta-modèle d'un diagramme de classe, les concepts de base d'un diagramme de classe sont : les packages, les classes, les attributs, les types définis et les type de base (string, integer, boolean). Chaque composant d'un diagramme de classe est représenté par une classe et relié avec les autres composants par les relations d'associations suivantes : import (vise le composant package), super (pour la classe 'classe'), type (entre les classes type et attribut) et les relations d'agréations (entre package et classe et entre classe et attribut).

Norme ISO/IEC/IEEE 42010

Il existe un bon nombre de normes pour modéliser et décrire les systèmes complexes. Nous pouvons noter "ISO/IEC/IEEE DIS 24641" qui offre des outils et des méthodes pour l'ingénierie basée modèles des systèmes, ainsi que la norme "ISO/IEC/IEEE 21840 :2019" qui offre des lignes directrices pour adresser un SoS afin de le différencier d'un système complexe unique. Nous nous intéressons, dans le cadre de notre travail, à la norme "ISO/IEC/IEEE 42010" [ISO/IEC/IEEE, 2011] qui vise à décrire l'architecture logicielle d'un système en exprimant les éléments de ce système par rapport à son environnement, en termes de composants ou systèmes constituants, comment ils opèrent et comment ils interagissent entre eux. Le choix de cette norme par rapport aux autres est justifié par son aptitude à décrire une architecture pour un système dans ces deux aspects : structurel et comportemental, tout en offrant des exigences qui s'appliquent aux descriptions d'architecture et aux langages de description d'architecture.

La figure 5.3 illustre un méta-modèle conceptuel décrivant les éléments qui doivent être présents dans un système individuel, nommé système d'intérêt dans la figure (system of interest). Ce système est décrit par une architecture logicielle. Les éléments qui sont spécifiés et englobés par cette description sont détaillés ci-dessous :

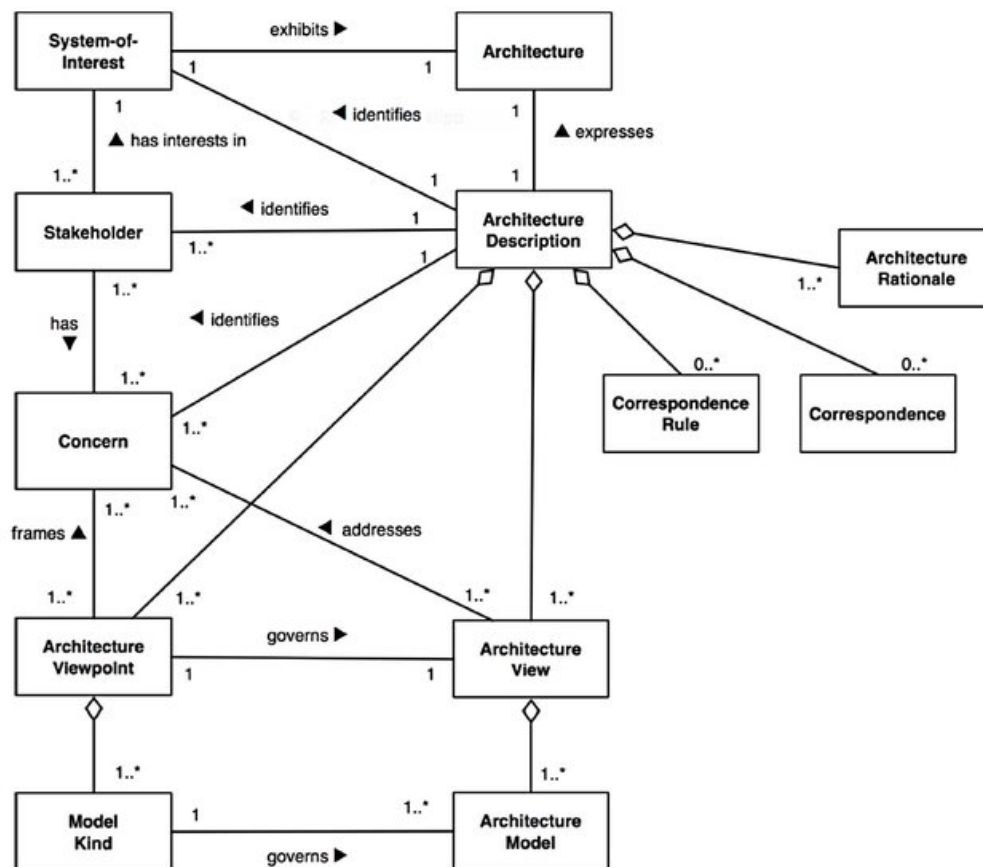


FIGURE 5.3: Méta-modèle issu de la norme ISO/IEC/IEEE 42010 [ISO/IEC/IEEE, 2011]

- 1- Vues et points de vues architecturaux : Une vue est un ensemble de modèles exprimant l'architecture d'un système dans une perspective spécifiques donnée, précisée par les préoccupations dans le cadre d'un point de vue. Un point de vue représente des conventions (comme les langages, les notations, etc.) pour la construction, l'interprétation et l'analyse des vues d' une architecture. Le point de vue est composé d'un ou plusieurs types de modèles, tandis qu'une vue contient un ou plusieurs modèles architecturaux.

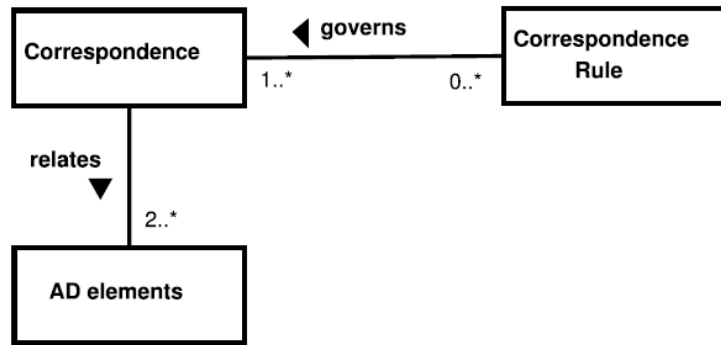


FIGURE 5.4: Modèle conceptuel des éléments AD et des correspondances [ISO/IEC/IEEE, 2011]

- 2- Partie prenante ("Stakeholder") : Désigne toute entité ayant un intérêt dans un système. La partie prenante dans une description architecturale est définie par un certain nombre de préoccupations.
- 3- Préoccupation ("Concern") : Représente les besoins attendus d'une architecture d'un système, comme les objectifs, l'adéquation de l'architecture pour atteindre ces objectifs, la faisabilité de sa mise en œuvre du système, la maintenabilité, etc.
- 4- Correspondances : La description architecturale inclut dans sa définition des règles de correspondances (figure 5.4) qui dictent les correspondances qui peuvent être établies entre les éléments d'une description architecturale (ou AD pour Architectural Description elements).

5.3 Syntaxe concrète

Les descriptions architecturales sont de plus en plus adaptées par les ingénieurs logiciels afin de gérer leurs systèmes. Ces descriptions leur permettent de travailler d'une façon intégrée et cohérente, ce qui améliore la conception des systèmes et facilite leurs compréhensions. Cependant, il n'existe pas de consensus sur les schémas et les modèles contenus dans une architecture, et ils sont souvent ambigus et informels. Le rôle des ADL (Architecture Description Language) est de servir comme un standard pour la description des architectures logicielles, en offrant une abstraction à haut niveau des éléments constituant un système. Pour décrire des systèmes complexes tels que les SoS, un ADL doit offrir deux vues distinctes :

- Une vue structurelle et hiérarchique, qui décrit l'ensemble des éléments constituant un système, cette vue est décrite graphiquement ou textuellement, pour une meilleure visibilité.
- Une vue comportementale et dynamique, qui spécifie l'aspect évolutif de l'architecture, c'est-à-dire la prise en charge des différentes évolutions qui se déroulent dans un système.

L'originalité du langage ArchSoS est qu'il traite non seulement les aspects hiérarchiques et structurels, mais aussi les aspects dynamiques qui peuvent survenir dans un comportement SoS, tels que la communication et les liens dynamiques entre les systèmes constituant. La figure 5.5 illustre deux niveaux d'abstraction sous forme d'une syntaxe concrète, détaillée ci-dessous, utilisée pour la description des éléments des SoS dans ArchSoS :

Format : Contient tous les systèmes indépendants qui composent un SoS, définis par leurs noms. Elle est divisée en :

- Le type des systèmes constituant déclaré via la balise "*Systems_Type*". Les constituants peuvent être des SoS eux-mêmes, ou des sous-systèmes atomiques (des systèmes qui ne sont pas composés d'autres systèmes).

- La structure hiérarchique des constituants d'un SoS, qui est donnée en utilisant la balise "*parent_of*" pour définir une relation arborescente d'appartenance .

Content : Spécifie le contenu fonctionnel du SoS et de ses constituants, elle contient :

```

ArchSoS id_SoS of id_SoS1; id_SoS2; id_SoS3; id_Subsystem1,
id_Subsystem2, id_Subsystem3, id_Subsystem4, id_Subsystem5,
id_Subsystem6...etc.
Format
    Systems_Type
        id_SoS: SoS; id_SoS1 SoS; ... id_Subsystem1: Sub_System;
        id_Subsystem2: Sub_System
        ...
    Hierarchy
        id_SoS parent_of id_SoS1; id_SoS2; id_SoS3/
        id_SoS1 parent_of id_Subsystem1; id_Subsystem2/
        id_SoS2 parent_of id_Subsystem3; id_Subsystem4/
        id_SoS3 parent_of id_Subsystem5 and id_Subsystem6
        ...
    Content
        Roles
            id_role11: id_Subsystem1; id_role12: id_Subsystem1;
            id_role21: id_Subsystem2; id_role22: id_Subsystem2;
            id_role31: id_Subsystem3; id_role32: id_Subsystem3;
            id_role33: id_Subsystem3;
            id_role41: id_Subsystem4;
            ...
        Behavioural_Constraints:
            Incompatible_Links: ~Link (id_SoS1, id_SoS2, Event1,
            link_type)...
            Incompatible_Roles: ~ [(id_role11: id_Subsystem1)
            (id_role32: id_Subsystem3)]
            ...
        Missions
            id_Mission1 {
                Events: id_event1, id_event2, id_event3;
                Links:
                    id_link1 : id_event1, link_type: id_Subsystem1 →
                    id_Subsystem2;
                    id_link2 : id_event2, link_type: id_Subsystem3
                    →id_Subsystem4;
                    id_link3 : id_event3, link_type: id_SoS1 → id_SoS2
                Roles_Combined
                    id_role11 + id_role21 + id_role32 + id_role41...
            }
            ...
    EndArch
    
```

FIGURE 5.5: Syntaxe concrete d'ArchSoS

- Rôles : Représentent les fonctionnalités du système. Un sous-système atomique a des rôles bien précis et prévisibles, tandis qu'un SoS a un seul rôle dynamique qui est inactif au début mais émerge suite à la combinaison de plusieurs rôles appartenant aux systèmes-constituants.
- Contraintes Comportementales : Elles contraignent les comportements dynamiques d'un SoS, car il n'y a pas contrôle sur la façon dont les systèmes peuvent réagir ou coopérer. Elles décrivent des conditions qui doivent être satisfaites dans un contexte spécifique qui diffère d'un SoS à l'autre.

Nous distinguons deux types de contraintes : (1) Liens Incompatibles : spécifient les liaisons interdites entre les systèmes, pour des raisons de sécurité ou d'autorité, les liens ont des types qui sont : "*a*" : *lien d'autorité*, "*u*" : *lien d'utilisation*, "*e*" : *lien d'échange de données*. La contrainte a la syntaxe :

$\sim \text{Link}(\text{id_SoS1}, \text{id_SoS2}, \text{Event1}, \text{link_type})$, cela indique que le *SoS1* et le *SoS2* ne peuvent pas être liés à l'événement *Event1* avec un lien d'un type particulier.

(2) Rôles Incompatibles : listent les combinaisons illégales de rôles et de fonctionnalités selon la syntaxe suivante :

$\sim [(id_role11 : id_Subsystem1)(id_role32 : id_Subsystem3)]$. Les rôles *role11* et *role32*, appartenant aux sous-systèmes *Subsystem1* et *Subsystem3* respectivement ne peuvent pas être combinés dans le SoS qui les contient.

- Missions : Une mission est l'objectif principal du SoS, elle peut être réalisée si les trois conditions suivantes sont réunies : (1) Un événement (ou plusieurs événements) externe est déclenché, affectant le SoS et/ou ses constituants, (2) Un lien (ou plusieurs liens) peut être créé entre les systèmes constituants affectés par l'événement, (3) Et finalement, des rôles, appartenant aux systèmes déjà liés peuvent être combinés pour accomplir la mission.

En adoptant l'approche MDA, nous procédons à la définition de la syntaxe d'ArchSoS en lui associant un méta-modèle illustré la figure 5.6. Il contient les éléments de base des SoS dans ArchSoS sous forme de classes spécifiant des entités et des associations autant que relations entre ses entités. Nous les identifions dans ce qui suit :

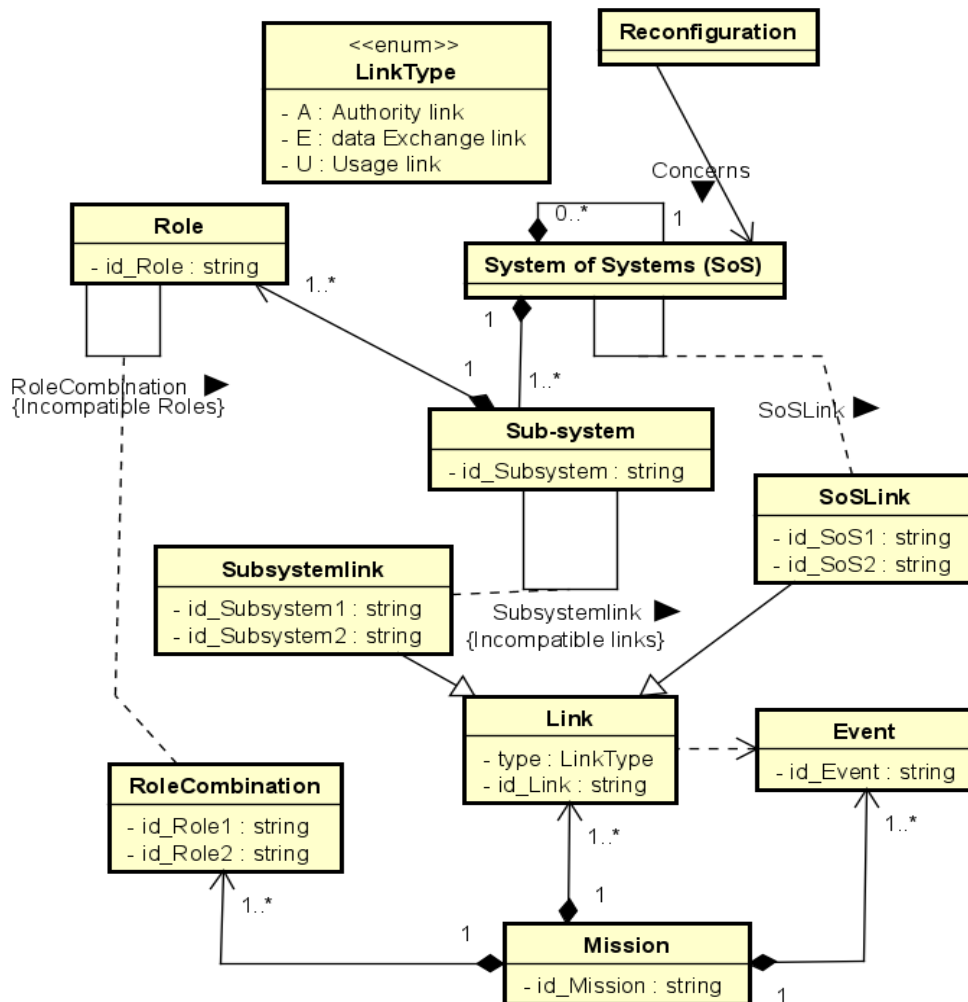


FIGURE 5.6: Méta-modèle d'ArchSoS

- Classe '*SoS*' : Cette classe décrit un SoS, elle est composée soit d'autres SoS, soit d'autres sous-systèmes avec une relation de composition. Entre deux SoS, il existe une classe associative qui est *SoSLink*, spécifiant un lien entre deux SoS connus par leurs noms. Chaque SoS est composé d'un seul rôle.

- Classe '*Sub-system*' : Une classe représentant les sous-systèmes atomiques des SoS. Entre deux sous-systèmes, il y'a une classe associative *Subsystemlink* qui indique la présence d'un lien entre eux. Chaque sous-système est composé d'au moins un rôle.
- Classe '*Role*' : La classe Rôle définie des fonctionnalités d'un système, où chaque SoS a un seul rôle, et chaque sous-systèmes en a plusieurs. Les rôles peuvent êtres combinés à l'aide d'une classe associative *RoleCombination* qui possède la contrainte comportementale *Incompatible Roles*.
- Classe '*Link*' : Les liens sont définis par une classe *Link*, qui est spécifiée par les deux classes *SoSLink* et *Subsystemlink*. Les liens dépendent d'un événement pour être établies, tandis que leur établissement dépend de la contrainte comportementale *Incompatible links*.
- Classe '*Event*' : Elle décrit des événements qui sont nécessaires à l'établissement des liens.
- Classe '*LinkType*' : C'est une énumération spécifiant les types de liens possibles entre les systèmes constituants (SoS ou sous-systèmes atomiques).
- Classe '*Mission*' : Cette classe est composée de trois autres classes : (1) *Event* (2) *Link* (3) *Role*.

La norme et le standard ISO/IEC/IEEE 42010 définit les éléments qui doivent être présents dans une description architecturale pour les systèmes informatiques. Cette norme peut être adaptée afin de traiter les SoS, leurs caractéristiques et leurs défis, en offrant une vue unifié pour ce type de systèmes.

Dans un premier pas, les auteurs dans [Chaabane et al., 2019] ont enrichi cette norme pour l'appliquer aux SoS, en remplaçant le système d'intérêt de la norme par une entité de systèmes constituants, liée à une entité de SoS. Cela a permis de redéfinir le modèle conceptuel de la norme pour qu'il soit apte à décrire les SoS. Similairement, nous étendons ce standard en remplaçant le système d'intérêt par une entité SoS contenant d'autres entités et éléments des SoS, inspirés du méta-modèle d'ArchSoS. Nous introduisons également la possibilité de reconfigurer les SoS, ainsi que les modèles formels adoptés dans la suite de notre travail. La figure 5.7 illustre cet ajout (les parties entourés par des rectangles bleus).

Ce nouveau méta-modèle décrit les architectures logicielles des SoS dans ArchSoS. Il est générique et sert comme une architecture de référence pour les SoS. Nous identifions les éléments composant cette architecture selon la nouvelle norme comme suit :

- (a) **Vues et point de vues architecturaux** : Nous définissons deux modèles formels pour la description architecturale d'ArchSoS : un modèle à base de BRS qui définit une vue graphique et algébrique pour ArchSoS, et un modèle basé Maude qui interprète et enrichit le premier pour définir une vue mathématique et équationnelle, qui est exécutable. Deux points de vues architecturaux sont identifiés pour nos modèles : (1) Un point de vue structurel conditionné par des règles de formations bigraphique. Ces règles gouvernent comment les éléments d'un SoS sont censés être modélisés et représentés, et elles doivent être respectées dans le langage Maude. (2) Un point de vue comportementale qui est délimité par des règles de réaction bigraphique, conditionnées par des contraintes comportementales.
- (b) **Partie prenantes** : Les entités qui ont un intérêt dans les SoS : (1) Les développeurs qui construisent et adaptent le système selon le méta-modèle normalisé d'ArchSoS (2) Un concepteur qui définit les contraintes comportementales, les événements et les missions d'un SoS. Il déclenche aussi les événements pour analyser et vérifier la consistance d'un SoS et sa capacité à aboutir aux missions définies, il peut aussi rajouter, remplacer ou supprimer des systèmes constituant un SoS.

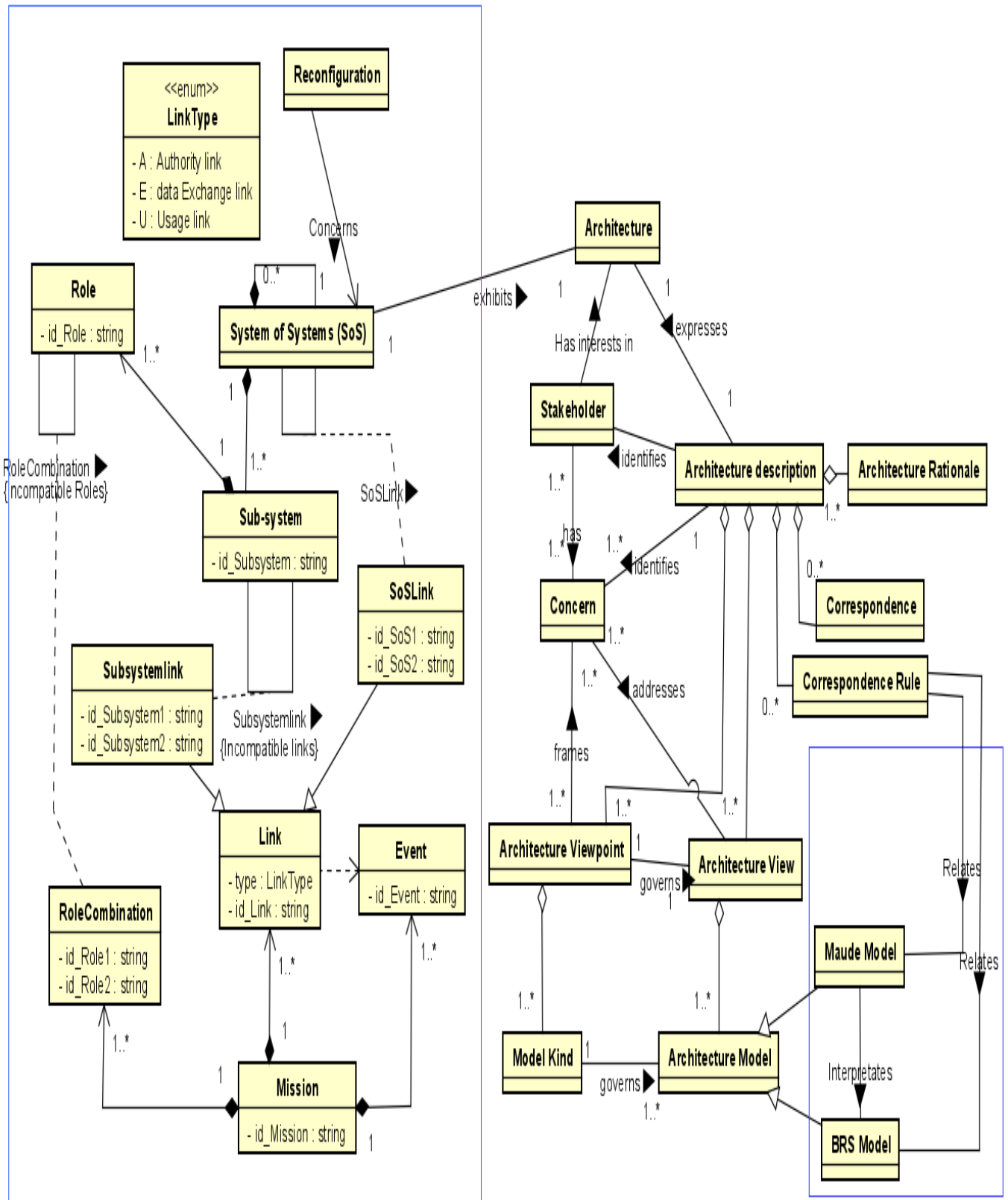


FIGURE 5.7: Méta-modèle normalisé d'ArchSoS

- (c) **Préoccupations** : Les préoccupations dans les SoS représentent globalement la mission du SoS, et plus précisément servent analyser comment les système constituant évoluent, interagissent et s'adaptent pour achever cette mission, d'une manière cohérente.
- (d) **Correspondances** : Il existe une correspondance entre les concepts de base d'ArchSoS,

et chaque modèle formel (les BRS, langage Maude) utilisé pour spécifier les SoS dans AchSoS.

5.4 Syntaxe abstraite basée BRS

Nous adaptons les BRS comme un formalisme pour définir la syntaxe d'ArchSoS. La structure des SoS est définie à l'aide de bigraphes dans la section 5.4.1, modélisant d'une façon intuitive les SoS et leurs constituants, construisant ainsi une architecture conceptuelle pour les SoS, dont les éléments sont tirés de l'architecture de référence définie au préalable. Dans la section 5.4.2, nous définissons l'aspect comportementale d'ArchSoS, décrivant ainsi la dynamique évolutive des SoS.

5.4.1 Aspect structurel

Les principaux éléments d'une instance architecturale des SoS dans ArchSoS sont définis selon le modèle de référence conceptuel vu précédemment. Pour attribuer une sémantique rigoureuse à leur représentation, nous établissons une correspondance entre chaque élément des SoS et élément des bigraphes :

- Une structure d'un SoS est représentée formellement par un bigraphe dont les nœuds sont soient des SoS, soient des sous-systèmes atomiques.
- La hiérarchie des constituants d'un SoS est gérée par le mécanisme d'imbrication des nœuds dans les bigraphes, où chaque nœud d'un SoS contient soient d'autres SoS, soient des sous-systèmes atomiques.
- Dans notre approche, nous donnons une signification à chaque port d'un nœud, c'est à dire les ports sont explicites et différent l'un de l'autre, nous en identifions deux types pour les SoS ou ses constituants :
 - (a) Port(s) R : Décrit un rôle (fonctionnalité) d'un système, les sous-systèmes atomiques ont des rôles prédéfinis décrivant leurs fonctionnalités, tandis que les rôles des SoS sont abstraits et représentent une mission lorsqu'ils sont liés à au moins deux autres ports de rôles appartenant à au moins deux systèmes constituants.
 - (b) Port L : Spécifie un point de liaison entre chaque deux SoS ou deux systèmes constituants, et avec un autre événement qui a provoqué cette liaison.
- Les événements qui affectent un SoS sont représentés par des noms internes détachés de base (pas liés) dans la région contenant les SoS, par défaut la région représente l'environnement contenant les événements.
- Les liens entre les SoS/systèmes constituants sont représentés par des hyper-arcs reliant les ports L de chacun avec un événement (nom interne). Similairement, les combinaisons de rôles (fonctionnalités) sont des hyper-arcs reliant le port spécifique du rôle d'un système avec le port du rôle d'un autre système, le même hyper-arcs relie cette combinaison avec le port abstrait d'un SoS pour spécifier une mission.
- Un SoS peut intégrer d'autres systèmes dans sa constitution, cela est spécifié par les sites présents dans un nœud d'un SoS.

Nous notons ici que nous avons utilisés les *les Bigraphes dirigés* [Grohmann and Miculan, 2007] comme extension, appropriée pour prendre en charge les directions des hyper-arcs définies dans les liens. Cela est traduit par la direction des flux dans les types liens (direction d'échange de données, direction de l'autorité et direction de l'utilisation entre chaque système).

Définition 5.1 (Bigraphe d'un SoS dans ArchSoS). *Formellement, le bigraphe d'un SoS B_{SoS} est donné par :*

$$B_{SoS} = (V_{SoS}, E_{SoS}, ctrl_{SoS}, Gl_{SoS}, Gp_{SoS}) : \langle m, X \rangle \rightarrow \langle n, Y \rangle$$

- V_{SoS} : Est l'ensemble fini de nœuds représentant soit des SoS, soit des sous-systèmes atomiques.
- E_{SoS} : Un ensemble fini d'hyper-arcs décrivant les connexions liant les constituants d'un SoS.
- $ctrl_{SoS}$: Est la signature (K, Ar, Ty) de l'ensemble des nœuds V_{SoS} où $Ar : K \rightarrow N$ assigne une arité N à chaque nœud en particulier, et $Ty : K \rightarrow M$ est une fonction introduite pour lister une identité explicite M pour chaque port d'un nœud ;
- $Gp_{SoS} = (V_{SoS}, ctrl_{SoS}, prnt_{SoS}) : m \rightarrow n$ est le graphe de places associé à B_{SoS} , m et n sont le nombre de sites et le nombre de régions. $prnt_{SoS} : m \uplus V_{SoS} \rightarrow V_{SoS} \uplus n$ est une fonction de parentalité décrivant la hiérarchie dans un SoS.
- $Gl_{SoS} = (V_{SoS}, E_{SoS}, ctrl_{SoS}, link_{SoS}) : X \rightarrow Y$ est le graphe de liens de B_{SoS} , avec la fonction $link_{SoS} : X \uplus P \rightarrow E_{SoS} \uplus Y$ spécifiant les interactions de chaque constituant dans un SoS sous forme d'hyper-arcs E_{SoS} où X , Y et P sont respectivement l'ensemble des noms interne, externe et l'ensemble de ports.
- $\langle m, X \rangle$: représente l'interface interne d'un bigraphe, où m est le nombre de sites qui peuvent représenter des constituants d'un SoS, et X est l'ensemble de noms internes représentant les événements d'un SoS.
- $\langle n, Y \rangle$: représente l'interface externe d'un bigraphe, où n est le nombre de régions, et Y est l'ensemble de noms externes.

Pour guider et contraindre la conception des SoS spécifiés avec un bigraph dans ArchSoS, des règles de formation sont définies et doivent être respectées. Le tableau 5.1 détaille les conditions des règles de formation $C_1 \dots C_{10}$ pour chaque type de nœud. Les architectures des SoS décrites à l'aide d'ArchSoS sont correctes-par-construction tant qu'elles respectent la définition du modèle bigraphique, ainsi que ces règles de formation.

La figure 5.8 illustre la forme graphique d'un bigraphe définissant un SoS nommé 'SoS', spécifié comme un nœud dans une région 0. Il contient deux constituants qui sont des nœuds de type sous-système Si et Sj . SoS a un port de liens L et un rôle inactif R , alors que Si et Sj ont les ensembles de ports respectifs : $\{Ri, Rj, Li\}$ et $\{Rk, Rl, Lj\}$, indiquant par exemple que Si possède deux fonctionnalités différentes illustrées par des rôles actifs Ri et Rj , et un port de liens Li . Nous notons que ces deux sous-systèmes peuvent être liés en réponse à un événement existant défini par le nom interne Ei .

En plus de leur forme graphique, les bigraphes sont munies d'une forme algébrique équivalente. Dans ce qui suit, nous définissons la forme algébrique qui décrit les SoS dans ArchSoS. Nous rappelons que le langage à termes algébriques qui définit les bigraphes est doté d'un ensemble de notations permettant de mettre en relation les éléments d'un bigraphe. A titre d'exemple, le '.' indique une imbrication de nœuds, l'opérateur '|' dénonce une juxtaposition entre des nœuds, alors que ' \square_i ' spécifie un site.

Afin d'inclure la notion des rôles explicites que nous avons rajouté et qui n'est pas prise en charge par le langage à termes algébriques initial, nous identifions un nouveau opérateur spécifiant un ensemble de ports pour chaque nœud dénoté par $\{Ri \dots Rn, Li\}$ ou $Ri \dots Rn$ est l'ensemble des rôles fonctionnels d'un système, et Li est le port de liens, séparant ainsi les ports et les rendant explicites pour les sous-systèmes. Par exemple un sous-système Si contenant les rôles Ri, Rj et Li est noté par : $Si\{Ri, Rj, Li\}$.

TABLE 5.1: Conditions de règles de formation pour les bigraphes des SoS

Condition	Description
C_1	Tous les nœuds d'une région 0 sont des SoS.
C_2	Tous les enfants d'un nœud SoS sont soit des SoS, soit des sous-systèmes atomiques
C_3	Tous les nœuds SoS ont exactement un port de liens L, et un port de rôles R
C_4	Tous les nœuds de type sous-système atomique ont au moins un port de rôles R, et un seul port de liens L
C_5	Tous les nœuds SoS et de sous-systèmes sont actifs.
C_6	Dans un nœud SoS, le port du rôle R est toujours lié à au moins deux ports de rôles R appartenant aux deux nœuds fils (SoS ou sous-systèmes) de cet SoS
C_7	Dans un nœud SoS, le port du rôle R peut être lié avec un autre port de rôle R d'un autre SoS grâce à un hyper-arc qui les relie tout les deux à un port de rôle R d'un SoS parent de ces deux SoS
C_8	Dans un nœud de type sous-système, chaque port de rôle R est lié à la fois avec un port de rôle R d'un autre sous-système, et au port de rôle R d'un SoS parent de ces deux sous-systèmes.
C_9	Dans un nœud SoS, le port L est toujours lié à la fois à un nom interne représentant un événement, et un autre port L d'un autre nœud SoS ayant le même parent hiérarchique
C_{10}	Dans un nœud sous-système, le port L est toujours lié à la fois à un nom interne représentant un événement, et un autre port L d'un autre nœud sous-système ayant le même parent hiérarchique

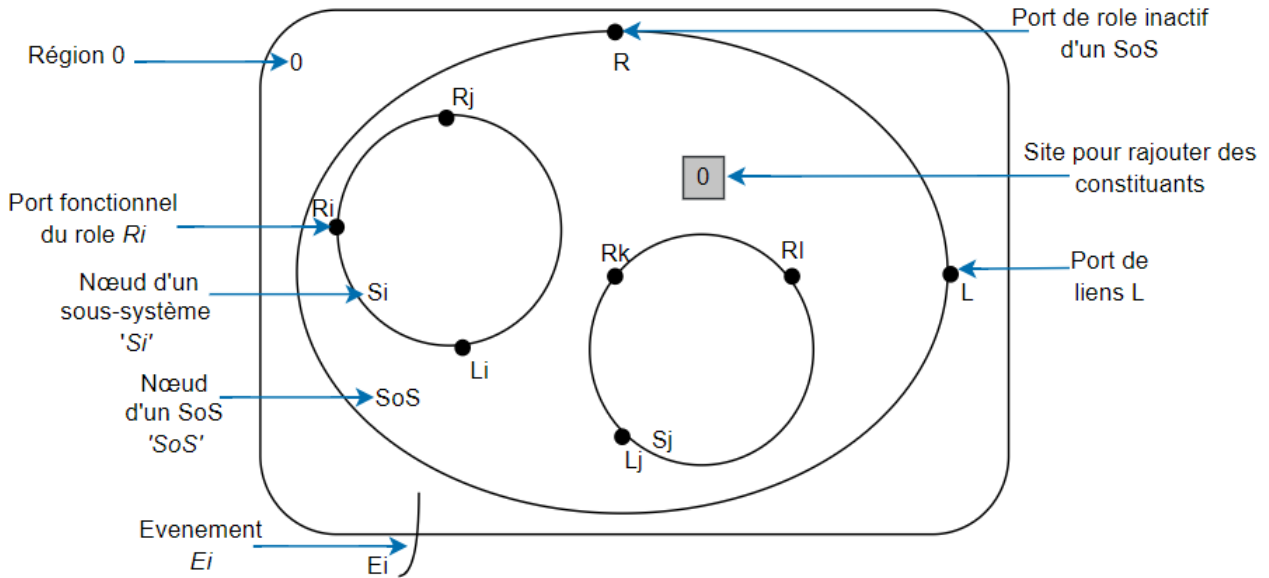


FIGURE 5.8: Description graphique d'un SoS dans ArchSoS

Définition 5.2 (Forme algébrique des SoS dans ArchSo). *La syntaxe algébrique d'un bigraphe dérivant les SoS dans ArchSoS est définie de manière récursive à l'aide de la forme de Backus Naur [McCracken and Reilly, 2003] par :*

$$\langle \text{SoS} \rangle ::= \langle \text{Event} \rangle \langle \text{id_SoS} \rangle \{ \langle \text{id_Role} \rangle \text{ , } \langle \text{id_Link} \rangle \} \text{ . } (\langle \text{SoS} \rangle / \langle \text{SoS} \rangle)$$

$$/ \langle \text{Event} \rangle \langle \text{id_SoS} \rangle \{ \langle \text{id_Role} \rangle \text{ , } \langle \text{id_Link} \rangle \} \text{ . } (\langle \text{SoS} \rangle)$$

$$\begin{aligned}
& | \langle \text{Event} \rangle \langle \text{id_SoS} \rangle \{ \langle \text{id_Role} \rangle \, , \, \langle \text{id_Link} \rangle \} \, . \, \langle \text{Sub_systems} \rangle \} \\
& \langle \text{Sub_systems} \rangle \, : \, : = \, \langle \text{id_Subsystem} \rangle \{ \langle \text{Roles} \rangle \, , \, \langle \text{id_link} \rangle \} \\
& | \langle \text{id_Subsystem} \rangle \{ \langle \text{Roles} \rangle \, , \, \langle \text{id_link} \rangle \} \, / \, \langle \text{Sub_systems} \rangle \\
& \langle \text{Roles} \rangle \, : \, : = \, \langle \text{id_role} \rangle \, | \, \langle \text{id_role} \rangle \, , \, \langle \text{Roles} \rangle \\
& \langle \text{Event} \rangle \, : \, : = \, \backslash \, \langle \text{id_event} \rangle \, | \, \backslash \, \langle \text{id_event} \rangle \, \langle \text{Event} \rangle \\
& \langle \text{id_event} \rangle \, : \, : = \, \text{String} \, \langle \text{id_Role} \rangle \, : \, : = \, \text{String} \\
& \langle \text{id_SoS} \rangle \, : \, : = \, \text{String} \, \langle \text{id_Subsystem} \rangle \, : \, : = \, \text{String} \\
& \langle \text{id_Link} \rangle \, : \, : = \, \text{String}
\end{aligned}$$

Exemple : Nous donnons la notation algébrique de l'exemple décrit graphiquement dans la figure 5.8 :

$\backslash EiSoS\{R, L\}.(Si\{Ri, Rj, Li\}|Sj\{Rk, Rl, Lj\})$ Ceci indique qu'un SoS, ayant le rôle R et le lien L (à l'intérieur de l'opérateur "... " qui indique ses ports), contient deux sous-systèmes Si et Sj via l'opérateur d'imbrication "...". Ils sont séparés par l'opérateur de juxtaposition "/", et ont respectivement les rôles Ri, Rj et Rk, Rl . Les ports Li et Lj sont respectivement les ports de liaison des sous-systèmes Si et Sj . Le SoS à un événement non lié Ei . Il est à noter que la représentation textuelle respecte les règles ci-dessus sur la façon dont un SoS est formé dans ArchSoS.

Le tableau 5.2 regroupe et résume l'ensemble des éléments syntaxiques d'ArchSoS, avec à la fois leurs notations algébriques et graphiques correspondantes.

5.4.2 Aspect comportemental

La partie bigraphique des BRS permet de modéliser l'aspect structurel d'un SoS dans ArchSoS, offrant une architecture formelle qui à deux vues : graphique et algébrique. Afin de prendre en charge les comportements des SoS et leurs évolutions, nous équipons ArchSoS d'une sémantique opérationnelle. Elle consiste à définir un ensemble d'états d'un SoS, et des transitions qui lui permettent de passer d'un état vers un autre pour évoluer.

Définition 5.3 (Etat d'un SoS). *Nous définissons formellement un SoS par :*

$$ST = (SoS, E)$$

Où :

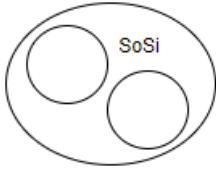

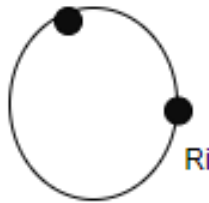
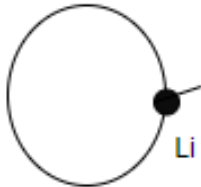
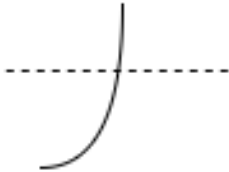
- SoS est la description bigraphique d'un SoS.
- E est un événement qui affecte cet SoS et ses constituants.

Actions d'évolution des SoS

L'évolution d'un SoS est exprimée par sa transition d'un état à un autre, pour répondre à un événement afin de finir par accomplir une mission donnée. Ceci est fait de la manière suivante : Un événement provoque un lien typé entre deux constituants d'un SoS. Une fois liés, ces deux constituants peuvent combiner certaines de leurs fonctionnalités dans le but d'obtenir un comportement qui représente une mission du SoS qui les contient. Cela permet à un SoS d'évoluer et d'avoir des transitions d'états. Ces transitions d'états sont obtenues grâce à l'application d'un ensemble d'actions.

Du point de vue bigraphique, une évolution est présenté par un bigraphe qui va évoluer vers un autre bigraphe en appliquant une action, qui est une règle de réaction bigraphique. Les bigraphes des SoS résultants de leurs évolutions doivent respecter les règles de formation définies, ainsi que les formes graphiques et algébriques correspondantes, notamment l'extension

TABLE 5.2: Description graphique et algébrique de la syntaxe d'ArchSoS

Concept d'ArchSoS	Forme algébrique bi-graphique	Forme graphique biraphique
SoS	$SoS\{.\}()$	Nœud contenant d'autres nœuds 
Sous-système	$Si\{\}$	Nœud atomique 
Rôle	$\{Ri, \dots Rn\}$	Ports de rôles 
Lien	Li	Ports de liens avec un hyper-arc 
Événement	$\setminus Ei$	Nom interne 

des ports explicites dans un bigraphe. Une action peut contenir des paramètres $pr1, pr2, \dots, prn$ pour transmettre des valeurs utilisées dans l'action, où chaque paramètre est considéré comme une variable.

Définition 5.4 (Evolution d'un SoS). *L'évolution d'un SoS en appliquant une action est définie par :*

$$A(pr1, pr2, \dots, prn) : ST \rightarrow ST'. \text{Où :}$$

— A : L'action à appliquer. Elle peut être de trois types (Tableau 5.3) :

- 1/ Actions de liens : Créer/détruire des liens typés entre les constituants d'un SoS avec un événement spécifique.
- 2/ Actions de rôles : acquérir/effacer des combinaisons des rôles pour achever (effacer) une mission d'un SoS.

TABLE 5.3: Description des actions d'évolution dans ArchSoS

Type	Action	Description	Forme algébrique
Actions de liens	Link_System($Ei, e : Si \rightarrow Sj$)	Crée un lien entre deux sous-systèmes Si et Sj , en reliant leurs ports respectifs Li et Lj à l'événement $Ei(Li_{Ei}$ et $Lj_{Ei})$, le lien est énoncé par l'hyper-arc eij	$\setminus Ei SoS\{R, L\}.(Si\{Ri, Rj, Li\} Sj\{Rk, Rl, Lj\})$ $\rightarrow SoS\{R, L\}.(Si\{Ri, Rj, Li_{Ei}(eij)\} Sj\{Rk, Rl, Lj_{Ei}(eij)\})$
	Destroy_Link($Ei, e : Si \rightarrow Sj$)	Supprime un lien entre les deux systèmes Si et Sj , en supprimant l'hyper-arc eij les liant avec l'événement Ei	$SoS\{R, L\}.(Si\{Ri, Rj, Li_{Ei}(eij)\} Sj\{Rk, Rl, Lj_{Ei}(eij)\})$ $\rightarrow \setminus Ei SoS\{R, L\}.(Si\{Ri, Rj, Li\} Sj\{Rk, Rl, Lj\})$
Actions de rôles	Acquire_Role($Ri : Si, Rk : Sj$)	Le rôle du SoS devient actif car les rôles Ri et Rk , appartenant respectivement aux sous-systèmes Si et Sj , sont combinés via l'hyper-arc x qui relie les 3 ports	$SoS\{R, L\}.(Si\{Ri, Rj, Li_{Ei}(eij)\} Sj\{Rk, Rl, Lj_{Ei}(eij)\})$ $\rightarrow SoS\{R(x), L\}.$ $(Si\{Ri(x), Rj, Li_{Ei}(eij)\} Sj\{Rk(x), Rl, Lj_{Ei}(eij)\})$
	Delete_Role($Ri : Si, Rk : Sj, x$)	Le rôle du SoS devient inactif en supprimant l'hyper-arc x liant son rôle aux rôles Ri et Rk des sous-systèmes Si et Sj	$SoS\{R(x), L\}.$ $(Si\{Ri(x), Rj, Li_{Ei}(eij)\} Sj\{Rk(x), Rl, Lj_{Ei}(eij)\})$ $\rightarrow SoS\{R, L\}.$ $(Si\{Ri, Rj, Li(eij)\} Sj\{Rk, Rl, Lj(eij)\})$
Actions de mobilité	Add_System(Sk, SoS)	Ajoute un sous-système Sk à un SoS, cette action est nécessaire lorsque tous les systèmes ne peuvent pas être liés par exemple	$\setminus Ei SoS\{R, L\}.(Si\{Ri, Rj, Li\} Sj\{Rk, Rl, Lj\})$ $\rightarrow \setminus Ei SoS\{R, L\}.(Si\{Ri, Rj, Li\} Sj\{Rk, Rl, Lj\} Sk\{Rm, Lk\})$
	Remove_System(SoS, Sk)	Supprime un sous-système Sj d'un SoS, lorsque le sous-système n'est pas nécessaire (ne peut être lié à aucun des autres sous-systèmes par exemple)	$\setminus Ei SoS\{R, L\}.(Si\{Ri, Rj, Li\} Sj\{Rk, Rl, Lj\} Sk\{Rm, Lk\})$ $\rightarrow \setminus Ei SoS\{R, L\}.(Si\{Ri, Rj, Li\} Sk\{Rm, Lk\})$
	Replace_System(SoS, Sj, Sk)	Lorsque deux systèmes ont des liens incompatibles, cette action remplace un sous-système Sj par le sous-système Sk afin d'avoir des liens compatibles dans le SoS	$SoS\{R, L\}.(Si\{Ri, Rj, Li_{Ei}(eij)\} Sj\{Rk, Rl, Lj_{Ei}(eij)\})$ $\rightarrow SoS\{R, L\}.(Si\{Ri, Rj, Li_{Ei}(eik)\} Sk\{Rm, Lk_{Ei}(eik)\})$

3/ *Actions de mobilité : Ajouter (respectivement effacer ou remplacer) des constituants dans un SoS*

— $pr1, pr2, \dots, prn$: L'ensemble des paramètres de l'action.

— ST et ST' : L'état initial d'un SoS, et l'état résultant après l'application de l'action.

Le tableau 5.3 résume l'ensemble des actions qui peuvent être appliquées à un SoS pour capturer son évolution, ainsi que leurs descriptions, et leurs représentations en utilisant la

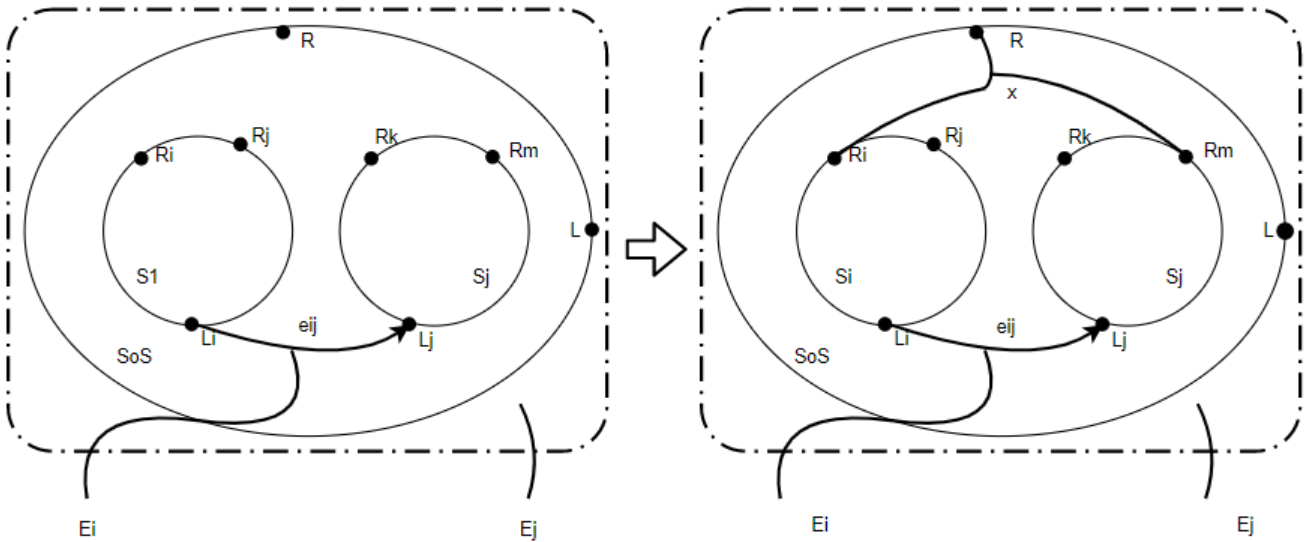


FIGURE 5.9: Action d'acquisition de rôle dans ArchSoS

forme algébrique d'une règle de réaction bigraphique.

Exemple : Un *SoS* qui acquiert un nouveau rôle est défini par l'action d'acquisition de rôle '*Acquire_Role*' dans le tableau. Les deux sous-systèmes S_i et S_j sont déjà liés au préalable par un lien de type '*e*' (hyper-arc e_{ij}) dirigé de S_i vers S_j en raison d'un événement E_i . L'action d'acquisition de rôle combine leurs rôles respectives R_i et R_k par le biais d'un hyper-arc ' x ' qui les relie avec le port R du *SoS* pour achever une mission. Cette action est illustrée dans la figure 5.9 afin de montrer sa forme graphique équivalente.

Transition guidée d'états

Les règles de réaction bigraphiques décrivant une évolution dans ArchSoS ont un déclenchement spontané, c'est à dire qu'une règle peut être automatiquement déclenchée dès que la partie du *redex* du bigraphe survient dans un contexte donné. Le comportement des *SoS* est incontrôlable et indéterministe, et peut aboutir à des états indésirables. Il faut imposer des restrictions pour pallier à ce problème.

Nous rappelons que ArchSoS est muni d'un ensemble de contraintes comportementales, interdisant certaines relations non autorisée entre systèmes. Ce sont des conditions qui doivent être satisfaites lorsque nous prenons en charge la dynamique d'un *SoS* et l'évolution de ses systèmes constituants. Elles se divisent en deux types de contraintes :

- Liens incompatibles : Ceci indique que certains systèmes ne peuvent pas être liés. La contrainte suit la syntaxe : $\sim \text{Link}(S_i, S_j, a)$, elle indique que deux systèmes S_i et S_j ne peuvent pas être liés avec un lien de type ' a '.
- Rôles incompatibles : représentent les combinaisons illégales de rôles entre systèmes constituants, notées : $\sim [(R_i : S_i), (R_j : S_j)]$, cela indique que les rôles R_i et R_j , correspondant respectivement aux sous-systèmes S_i et S_j , ne peuvent pas être combinés ensemble.

Bien que ces contraintes ont des syntaxes spécifiques, elles diffèrent d'un *SoS* à un autre en termes de ses constituants et de leurs relations. Le rôle d'un concepteur est d'affecter à chaque *SoS* un ensemble de contraintes comportementales.

Afin de réduire et de guider le comportement non déterministe des *SoS* dans ArchSoS, nous définissons, en plus des contraintes comportementales, d'autres conditions affectant l'application des actions dans ArchSoS, sous forme d'un ensemble de prédicats $\phi_1 \dots \phi_{10}$. Nous associons

TABLE 5.4: Prédicat de contrôle d'évolution des SoS dans ArchSoS

Prédicat	Description	Actions associé
ϕ_1	Le rôle d'un SoS est actif	Delete_role, Acquire_role
ϕ_2	Deux rôles sont compatibles	Acquire_role
ϕ_3	Tous les rôles sont incompatibles	Replace_System
ϕ_4	Un constituant est lié	Link_system, Destroy_Link
ϕ_5	Deux constituants sont déjà liés	Destroy_link, Link_System
ϕ_6	Un lien est possible entre deux constituants et un événement	Link_System
ϕ_7	Tous les liens sont incompatibles	Add_System
ϕ_8	Un système constituant peut être rajouté	Add_System
ϕ_9	Un système constituant peut être supprimé	Remove_System
ϕ_{10}	Un système constituant peut être remplacé par un autre	Replace_System

ces prédicats à chaque action A_i , afin d'obtenir un ensemble de prédicats ϕ_{A_i} qui restreint l'application de cette action uniquement quand les prédicats associés sont vérifiés. Les prédicats pour chaque type d'actions sont résumés dans le tableau 5.4. Par exemple, ϕ_4 vérifie si un système spécifique est lié ou non. On remarque que la négation des prédicats ϕ_2 et ϕ_6 est une représentation des contraintes comportementales définies précédemment (rôles incompatibles / liens incompatibles respectivement).

Définition 5.5 (Transition d'états). *Une transition d'états guidée d'un SoS sera définie comme suit :*

$$(A_i, \phi_{A_i}) : ST \rightarrow ST' \text{ if } \phi_{A_i}$$

Où :

- ST est un état donné d'un SoS.
- A_i est l'action à appliquer.
- ϕ_{A_i} est l'ensemble prédicat qui doit être vérifié pour faire l'action A_i .
- ST' est l'état résultant de l'application de l'action.

5.5 Conclusion

Dans ce chapitre, nous avons utilisé la méthode MDA et la norme ISO/IEC/IEEE 42010 comme des concepts de base afin de définir une architecture de référence pour la spécification des architectures logicielles des SoS. Plus précisément, nous avons adapté cette norme pour qu'elle soit adéquate avec les éléments des SoS dans ArchSoS, afin de représenter ce type de systèmes. Nous avons fini par obtenir un méta-modèle normalisé pour ArchSoS, qui décrit sa syntaxe concrète.

Nous avons également formalisé ArchSoS, en adaptant les BRS comme un langage formel pour lui définir une syntaxe abstraite, spécifiant ainsi les aspects structurels et comportementaux de notre ADL. Cette formalisation vise réduire la complexité de la modélisation et de la conception des architectures des SoS.

En particulier, Le coté bigraphique des BRS a servi pour décrire tous les éléments architecturaux d'ArchSoS d'une façon bien générique, cela est fait sur deux vues : une vue graphique facilitant la compréhension des SoS, et une vue algébrique équivalente. Nous avons étendue et enrichi les notions des bigraphe avec l'inclusion des ports explicites pour séparer les fonctionnalités des constituants d'un SoS, ainsi qu'avec un ensemble de règles de formations qui dictent comment un bigraphe décrivant les SoS est censé être formé.

En outre, nous avons défini l'aspect comportemental des SoS dans ArchSoS, via des transitions d'états d'un SoS en appliquant des actions basées sur les règles de réaction bigraphiques. Afin de guider cette évolution pour éviter d'avoir des comportements indésirable et indéterministes, nous avons introduits un ensemble de prédicats qui servent autant que conditions pour le déclenchement des actions.

Dans le prochain chapitre, nous montrons comment associer à ArchSoS une sémantique opérationnelle exécutable, en faisant recours au langage Maude pour spécifier et implémenter les éléments d'ArchSoS, ainsi que leur évolution et leur aspect comportemental.

Sémantique Opérationnelle d'ArchSoS basée Maude

Sommaire

6.1	Introduction	74
6.2	Intégration d'une spécification ArchSoS dans Maude	74
6.2.1	Principe	75
6.2.2	Aspects structurels	76
6.2.3	Aspects dynamiques	78
6.3	Réécriture et exécution d'une spécification ArchSoS	81
6.4	Analyse formelle d'un SoS	83
6.5	Conclusion	86

6.1 Introduction

Les systèmes réactifs bigraphiques représentent un moyen adéquat pour modéliser formellement les systèmes distribués et mobiles. de modélisation formelle, afin de décrire la structure et les comportements des SoS dans ArchSoS. Ils ont permis de représenter cet ADL à la fois graphiquement et visuellement, et ont offert des actions sous formes de règles de réactions, qui décrivent l'évolution des SoS. Cependant, ue la richesse d'expression d'ArchSoS dans la modélisation de la dynamique des SoS, nous avons constaté l'incapacité des BRS pour attribuer une sémantique opérationnelle complète à ArchSoS. Notre définition alternative de cette notion est faite sur la base de la théorie de réécriture, et particulièrement le langage Maude à travers une de ses extensions : le langage de stratégies Maude.

Dans ce chapitre, nous introduisant une définition de la sémantique d'ArchSoS en implémentant son modèle bigraphique sous le langage Maude, qui offre un cadre sémantique exécutable. Dans la section 6.2, nous introduisant le processus de transcription d'ArchSoS vers Maude. Ensuite, nous détaillons dans la section 6.3 l'apport des stratégies dans Maude pour la reconfiguration dynamique des SoS. Finalement, nous présentons un nombre de propriétés Maude qui permettent de vérifier formellement le bon fonctionnement des SoS définis dans ArchSoS.

6.2 Intégration d'une spécification ArchSoS dans Maude

Nous détaillons dans cette section comment transcrire les aspects structurels et comportementaux d'ArchSoS définis avec les BRS vers des spécifications Maude équivalentes. Nous utilisons un ensemble de modules Maude différents qui couvrent les deux aspects.

6.2.1 Principe

Maude, à sa forme basique, est doté de deux types de modules : des modules fonctionnels et des module systèmes. En utilisant l'extension du langage de stratégies Maude, nous y rajoutons un troisième module qui est un module de stratégies Maude pour la définition de la sémantique d'ArchSoS dans Maude.

- Modules fonctionnels $(\Sigma_{ArchSoS}, E_{ArchSoS} \cup A_{ArchSoS})$: Ils spécifient une théorie équationnelle Maude, qui est basée sur une logique mathématique d'appartenance, où $\Sigma_{ArchSoS}$ est la signature qui précise la structure typée de chaque élément d'un système (sortes, sous-types, opérateurs, etc.), $E_{ArchSoS}$ est la collection d'équations (éventuellement conditionnelles) déclarées agissant sur la structure, $A_{ArchSoS}$ est la collection d'attributs équationnels déclarés pour les différents opérateurs, comme l'attribut de commutativité ou d'associativité entre les éléments.
- Modules systèmes $(\Sigma_{ArchSoS}, E_{ArchSoS} \cup A_{ArchSoS}, R_{ArchSoS})$: Décritent une théorie de réécriture Maude où : $(\Sigma_{ArchSoS}, E_{ArchSoS} \cup A_{ArchSoS})$ représente la théorie équationnelle d'un système, $R_{ArchSoS}$ est l'ensemble de règles de réécritures (éventuellement conditionnelles) qui sont appliquées pour changer le comportement d'un système, et faire évoluer ainsi sa structure équationnelle.
- Modules de stratégies $(\Sigma_{ArchSoS}, E_{ArchSoS} \cup A_{ArchSoS}, S_{ArchSoS}(R_{ArchSoS}, SM_{ArchSoS}))$: Ils définissent un ensemble de stratégies qui contrôlent et guident l'application des règles de réécritures dans Maude. $(\Sigma_{ArchSoS}, E_{ArchSoS}$ et $A_{ArchSoS})$ représente la théorie équationnelle du système qui est censée évoluer. $S_{ArchSoS}$ est une sémantique décrivant le comportement d'un système, elle est construite à partir d'un ensemble $R_{ArchSoS}$ contenant des règles de réécritures, et un module de stratégies $SM_{ArchSoS}$ qui va guider la réécriture et l'utilisation de ces règles en utilisant des stratégies.

A partir de cela, nous définissons quatre modules complémentaires pour définir la sémantique d'ArchSoS, résumés ci-dessous :

- Module fonctionnel **ArchSoSSyntax** : Il définit la syntaxe structurelle (voir figure 6.1) d'un SoS dans ArchSoS, en implémentant les éléments bigraphiques (tirés du modèle graphique/algébrique) d'ArchSoS (SoS, sous-systèmes, liens, événements, rôles...etc) sous forme de sortes et d'opérations. Cette structure permet de construire des SoS complets avec tous leurs constituants.
- Module Système **ArchSoSBehaviours** : Il contient la sémantique offerte par l'ensemble des actions d'évolutions, qui permettent de faire évoluer la structure d'un SoS décrite dans le module fonctionnel (figure 6.1). Les actions basées sur les règles de réactions bigraphiques sont introduites par le biais de règle de réécritures Maude. Ce module inclut aussi les prédicats de contrôles qui conditionnent les actions d'évolution, en adaptant des règles de réécritures conditionnelles. Ses prédicats sont représentés par des équations Maude. En effet, les spécifications en langage Maude permettent d'implémenter des modèles bigraphiques d'une manière qui préserve leur sémantique d'une part, et permet de les enrichir d'une autre part.
- Module Système **ArchSoSAnalysis** : Un module système, tel qu'il a est présenté dans la figure 6.1, qui décrit un ensemble de propriétés LTL, introduites dans le but de vérifier le bon comportement des SoS, en vérifiant à la fois des propriétés de sûreté et de vivacité, et des propriétés qui vérifient l'atteignabilité des missions des SoS en cours de leurs évolution. Ces propriétés sont analysées en utilisant l'outil model-checker de Maude, qui lancera une vérification d'une propriété sur une simulation d'un état initial d'un SoS vers un état final, en appliquant l'ensemble des règles de réécritures.

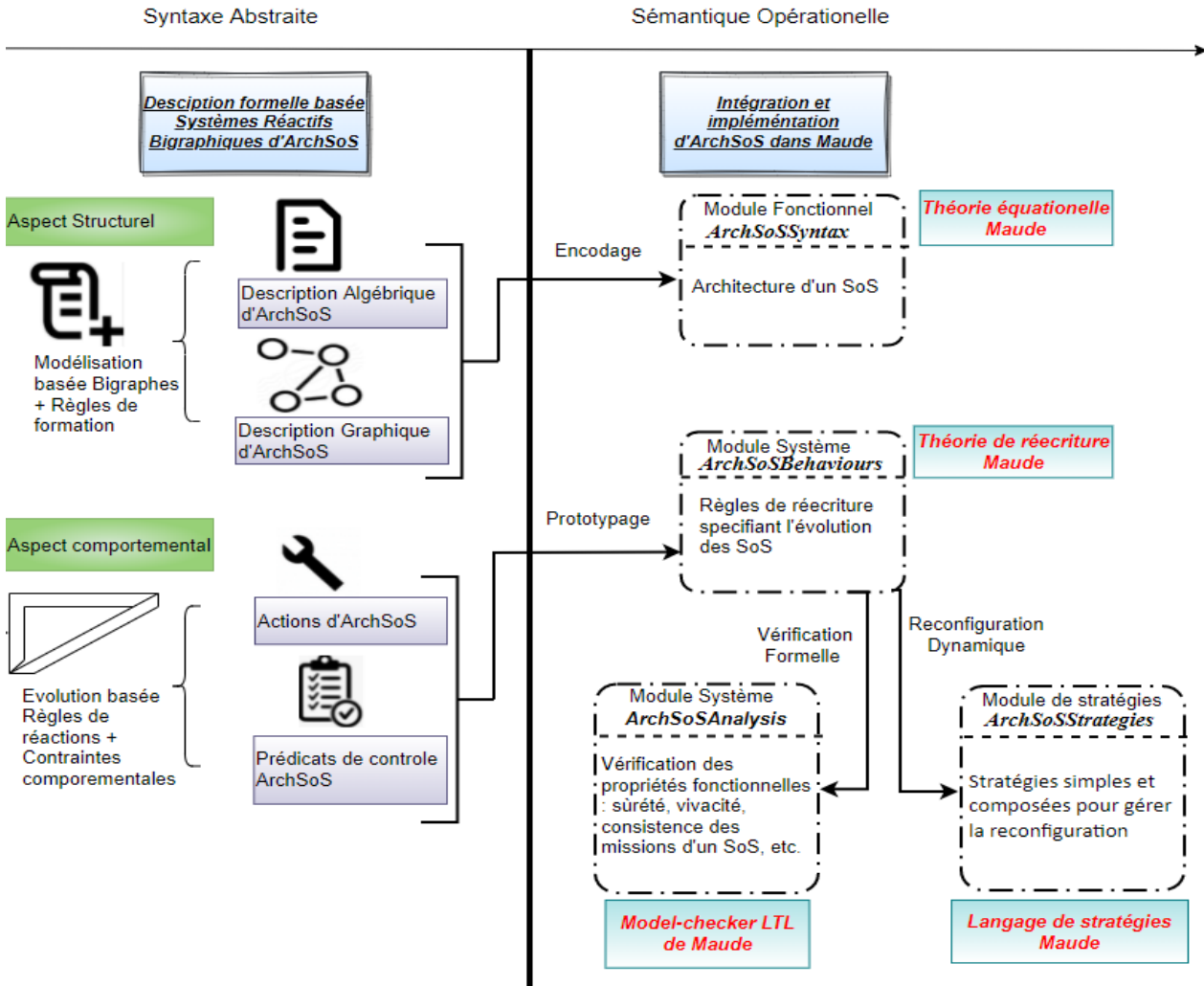


FIGURE 6.1: Du modèle BRS vers les modules Maude

- Module de stratégies **ArchSoSStrategies** : Il sert à définir un ensemble de stratégies spécifiques aux missions des SoS, où chaque stratégie définit un chemin particulier d'exécution, en regroupant des règles de réécritures définies dans le module *ArchSoSBehaviours*. Cela a pour but de guider l'exécution d'un SoS en reconfigurant son comportement pour avoir des résultats désirable et déterministes (voir figure 6.1).

6.2.2 Aspects structurels

La syntaxe d'ArchSoS est définie dans Maude par une théorie équationnelle décrite dans le module fonctionnel **ArchSoSSyntax**. Dans ce module, à chaque élément structurel dans le modèle bigraphique d'ArchSoS, est associé une sorte Maude, et des opérations qui permettent de construire ou d'agir sur ces sortes. La partie haute (syntaxe) du tableau 6.1 décrit ses éléments. Nous les détaillons dans ce qui suit :

- SoS : Les SoS ont une sorte *SoS* décrivant leurs types, et une sorte *idSoS* qui décrit l'identité de chaque SoS. Ils sont construits et définis via l'opérateur :

$$op\ opSoS < _, _ > .P[_].L[_].E[_] : idSoS\ Role\ SubSys\ Link\ Event \rightarrow SoS[ctor]$$

ou

$$op\ opSoS < _, _ > .P[_].L[_].E[_] : idSoS\ Role\ SoS\ Link\ Event \rightarrow SoS[ctor]$$

TABLE 6.1: Correspondance table between ArchSoS aspects and Maude.

Asp -ect	Élément Arch- SoS	Spécification Maude
Syntaxe ArchSoS	SoS	<pre> sort SoS idSoS . op opSoS < _, _ > .P[_].L[_].E[_] : idSoS Role SubSyS Link Event → SoS [ctor] . op opSoS < _, _ > .P[_].L[_].E[_] : idSoS Role SoS Link Event → SoS [ctor] . op null : → SoS [ctor] . op _ _ : SoS SoS → SoS [ctor assoc comm] . </pre>
	Sous-systèmes	<pre> sort subSyS idsubSys . op opsubSyS < _, _ > .L[_] : idsubSys Role Link → subSyS [ctor] . op _ _ : subSyS subSyS → subSyS [ctor assoc comm] . </pre>
	Événements	<pre> sort Event . op Event_Name → Role [ctor]. op null : → Role [ctor] . op _, _ : Role Role → Role [ctor assoc comm] . </pre>
	Roles	<pre> sort Role . op Role_Name → Role [ctor]. op null : → Role [ctor] . op _, _ : Role Role → Role [ctor assoc comm] . op _ + _ : Role Role → Role [ctor assoc comm] . </pre>
	Liens	<pre> sorts Link LinkT. op link < _ : _; _ --> _ > : Event LinkT idSoS idSoS →Link [ctor] . op link < _ : _; _ --> _ > : Event LinkT idsubSys idsubSys →Link [ctor] . op a : → LinkT [ctor] . op u : → LinkT [ctor] . op e : → LinkT [ctor] . </pre>
Sémantique ArchSoS	Prédicats de contrôles ϕ_i	<p>Équations Maude :</p> $\phi_1 : \text{IsActive}(\text{SoS}_i), \phi_2 : \text{CompRoles}(\text{Role}_i, \text{Role}_j),$ $\phi_3 : \text{NoComp}(\text{SoS}_i), \phi_4 : \text{isLinked}(\text{Subsystem}_i \text{ ou } \text{SoS}_i),$ $\phi_5 : \text{areLinked}(\text{Subsystem}_i, \text{Subsystem}_j, \text{Event}, \text{LinkT}),$ $\phi_6 : \text{PosLink}(\text{Subsystem}_i, \text{Subsystem}_j, \text{Event}, \text{LinkT}),$ $\phi_7 : \text{NoClinks}(\text{SoS}_i), \phi_8 : \text{CanAdd}(\text{SoS}_i, \text{SubSystem}_i),$ $\phi_9 : \text{CanRemove}(\text{SoS}, \text{SubSystem}_i), \phi_{10} :$ $\text{CanReplace}(\text{SoS}, \text{SubSystem}_i, \text{SubSystem}_j) .$
	Action	Règle de réécriture conditionnée Maude $\text{crl} [\text{Nom-de-règle}] : \text{Etat}_i \rightarrow \text{Etat}_j \text{ si } \phi_i .$
	Mission	Définition d'une stratégie Maude S_i

Reconfiguration	Combinaison des règles de réécritures dans une stratégies via des opérateurs
Propriétés a vérifier	Propriétés LTL dans Maude Prop _i

L'opérateur *opSoS* a 5 arguments pour construire un SoS, nous les notons respectivement : un identifiant du SoS, le rôle du SoS, les constituants qu'il contient (soient des sous-systèmes, ou bien des SoS), et les événements affectant cet SoS. Le mot clé *ctor* indique que c'est un opérateur constructeur. Nous définissons des opérateurs dédiés aux SoS comme l'opérateur *null* qui permet de construire un SoS vide, et l'opérateur *_|_* qui permet de concaténer, d'une façon associative et commutative via les mots clés *assoc* et *comm* respectivement, deux SoS formant une suite d'SoS.

- Sous-système : Les sous-systèmes sont des constituants atomiques d'un SoS. Ils sont construits grâce a l'opérateur *opsubSys* comme suit :

$$op\ opsubSys < _, _ > .L[_] : idsubSysRoleLink \rightarrow subSys[ctor].$$

Il a trois arguments pour construire un sous-système : un identificateur, un (ou plusieurs) rôles, et un lien. Similairement aux SoS, nous associons les opérateurs *null* et *_|_* aux sous-systèmes.

- Événements : Les événements sont définis à travers une sorte *Event*. Chaque événement potentiel doit être déclaré à part via un opérateur, selon la syntaxe suivante :

$$op\ Event_Name \rightarrow Role[ctor].$$

Où *Event_Name* est le nom(identifiant) de l'événement. L'opérateur *null* indiquera l'absence d'un événement, et plusieurs événements peuvent exister en les séparant par l'opérateur *_, _*.

- Rôles : Les rôles des SoS/Sous-systèmes sont identifiés par la sorte *Roles*. De la même façon que dans les événements, chaque rôle doit être déclaré à l'aide de l'opérateur suivant :

$$op\ Role_Name \rightarrow Role[ctor].$$

Role_name est le nom/identifiant du rôle qui le sépare des autres rôles. Un rôle *null* signifie qu'il n'est pas actif. Un sous-système peut avoir plusieurs rôles en les séparant via l'opérateur *_, _*, tandis que plusieurs rôles peuvent être combinés au sein d'un SoS via l'opérateur *_ + _*.

- Liens : Aux liens sont associés deux sortes différentes (1) Une sorte *Link* qui identifie le lien. (2) Une sorte *LinkT* qui indique les types potentiels d'un lien. Ainsi, un lien entre deux SoS est construit a partir de l'opérateur : $oplink < _ : _ ; _ \dashrightarrow _ > : EventLinkTidsubSysidsubSys \rightarrow Link[ctor]$. ou dans le cas d'un lien entre sous-systèmes : $oplink < _ : _ ; _ \dashrightarrow _ > : EventLinkTidSoSidSoS \rightarrow Link[ctor]$. Le premier argument est l'événement qui a causé le lien, le deuxième c'est le type du lien, suivi des identificateurs des constituants (SoS ou sous-systèmes) séparés par le symbol "*-->*" pour indiquer la direction du flux entre les deux.

6.2.3 Aspects dynamiques

La sémantique d'un SoS est définie dans un module système *ArchSoSBehaviours*, il contient un ensemble de règle de réécritures qui font passer la structure définie précédemment d'un état vers un autre. Afin de définir le comportement des SoS dans ArchSoS sous Maude, il faut passer par deux étapes :

TABLE 6.2: Prédicats de contrôles définis sous Maude

Prédicat	Équation Maude
ϕ_1 :IsActive	$\text{ceq isActive}(\text{opSoS} \langle \text{SoSi}, \text{Ri} \rangle . \text{P}[\text{Si} \mid \text{Sj}]. \text{L}[\text{Li}]. \text{E}[\text{Ei}]) = \text{false if } (\text{Ri} == \text{null}) .$
ϕ_2 :CompRoles	$\text{eq CompRoles}(\text{R1}, \text{R2}) = \text{true} .$ $\text{eq CompRoles}(\text{R1}, \text{R3}) = \text{true} .$ $\text{eq CompRoles}(\text{R2}, \text{R3}) = \text{false} .$
ϕ_3 :NoComp	$\text{ceq NoComp}(\text{opSoS} \langle \text{SoS1}, \text{null} \rangle . \text{P}[\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{null}] \mid \text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{null}] \mid \text{opsubSyS} \langle \text{S3}, \text{R3} \rangle . \text{L}[\text{null}]] . \text{L}[\text{null}]. \text{E}[\text{E1}]) = \text{true if } (\text{CompRoles}(\text{R1}, \text{R2}) == \text{false and CompRoles}(\text{R1}, \text{R3}) == \text{false and CompRole}(\text{R2}, \text{R3}) == \text{false})$
ϕ_4 :isLinked	$\text{ceq isLinked}(\text{opsubSyS} \langle \text{Si}, \text{Ri} \rangle . \text{L}[\text{Li}]) = \text{false if Li} == \text{null} .$ $\text{ceq isLinked}(\text{opSoS} \langle \text{SoSi}, \text{Ri} \rangle . \text{P}[\text{Si} \mid \text{Sj}]. \text{L}[\text{Li}]. \text{E}[\text{Ei}]) = \text{false if Li} == \text{null} .$
ϕ_5 :areLinked	$\text{ceq areLinked}(\text{opsubSyS} \langle \text{Si}, \text{Ri} \rangle . \text{L}[\text{Li}], \text{opsubSyS} \langle \text{Sj}, \text{Rj} \rangle . \text{L}[\text{Lj}], \text{Ei}, \text{type}) = \text{true if } ((\text{Li} == \text{link} \langle \text{Ei} : \text{type}; \text{Si} \dashrightarrow \text{Sj} \rangle \text{ and } \text{Lj} == \text{link} \langle \text{Ei} : \text{type}; \text{Ri} \dashrightarrow \text{Rj} \rangle) \text{ or } (\text{Li} == \text{link} \langle \text{Ei} : \text{type}; \text{Sj} \dashrightarrow \text{Si} \rangle \text{ and } \text{Lj} == \text{link} \langle \text{Ei} : \text{type}; \text{Sj} \dashrightarrow \text{Si} \rangle) .)$
ϕ_6 :PosLink	$\text{eq PosLink}(\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{Li}], \text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{Lj}], \text{Ei}, \text{e}) = \text{true}$ $\text{eq PosLink}(\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{Li}], \text{opsubSyS} \langle \text{S3}, \text{R3} \rangle . \text{L}[\text{Lj}], \text{Ei}, \text{e}) = \text{false}$ $\text{eq PosLink}(\text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{Li}], \text{opsubSyS} \langle \text{S3}, \text{R3} \rangle . \text{L}[\text{Lj}], \text{Ei}, \text{e}) = \text{true}$
ϕ_7 :NoClinks	$\text{ceq NoClinks}(\text{opSoS} \langle \text{SoS1}, \text{null} \rangle . \text{P}[\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{null}] \mid \text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{null}] \mid \text{opsubSyS} \langle \text{S3}, \text{R3} \rangle . \text{L}[\text{null}]] . \text{L}[\text{null}]. \text{E}[\text{E1}]) = \text{true if PosLink}(\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{Li}], \text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{Lj}], \text{Ei}, \text{e}) = \text{false and PosLink}(\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{Li}], \text{opsubSyS} \langle \text{S3}, \text{R3} \rangle . \text{L}[\text{Lj}], \text{Ei}, \text{e}) = \text{false and PosLink}(\text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{Li}], \text{opsubSyS} \langle \text{S3}, \text{R3} \rangle . \text{L}[\text{Lj}], \text{Ei}, \text{e}) == \text{false}$
ϕ_8 :CanAdd	$\text{ceq canAdd}(\text{opSoS} \langle \text{SoS1}, \text{R} \rangle . \text{P}[\text{Si} \mid \text{Sj}]. \text{L}[\text{Li}]. \text{E}[\text{Ei}], \text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}]) = \text{true if isLinked}(\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}]) == \text{false and NoClinks}(\text{opSoS} \langle \text{SoS1}, \text{R} \rangle . \text{P}[\text{Si} \mid \text{Sj}]. \text{L}[\text{Li}]. \text{E}[\text{Ei}], \text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}]) == \text{true} .$
ϕ_9 :CanRemove	$\text{ceq CanRemove}(\text{opSoS} \langle \text{SoS1}, \text{R} \rangle . \text{P}[\text{Si} \mid \text{Sj}]. \text{L}[\text{Li}]. \text{E}[\text{Ei}], \text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}]) = \text{true if isLinked}(\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}]) == \text{false and Si} == \text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}].$
ϕ_{10} :CanReplace	$\text{ceq CanReplace}((\text{opSoS} \langle \text{SoS1}, \text{R} \rangle . \text{P}[\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}] \mid \text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{L2}] \mid \text{opsubSyS} \langle \text{S3}, \text{R3} \rangle . \text{L}[\text{L3}]] . \text{L}[\text{Li}]. \text{E}[\text{Ei}], \text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}], \text{opsubSyS} \langle \text{S4}, \text{R4} \rangle . \text{L}[\text{L3}]) = \text{true if } (\text{isLinked}(\text{opsubSyS} \langle \text{S4}, \text{R4} \rangle . \text{L}[\text{L3}]) == \text{false and areLinked}(\text{opsubSyS} \langle \text{S1}, \text{R1} \rangle . \text{L}[\text{L1}], \text{opsubSyS} \langle \text{S2}, \text{R2} \rangle . \text{L}[\text{L2}]] . \text{L}[\text{Li}]. \text{E}[\text{Ei}]) == \text{true and CompRoles}(\text{R1}, \text{R2}) == \text{false and CompRoles}(\text{R2}, \text{R3}) == \text{true}$

- La première consiste à définir les prédicats de contrôles Φ_i , qui servent à contraindre les actions d'évolution dans ArchSoS. Un prédicat est traduit en Maude par une équation, qui retournera une valeur booléenne (True ou False).
- La deuxième est de spécifier les actions d'évolution d'ArchSoS par des règles de réécritures conditionnés Maude, où la condition est la validation d'un ou plusieurs prédicats de contrôle.

Exemple : Nous détaillons dans ce qui suit chaque étape, avec comme exemple un SoS illustratif décrit dans ArchSoS, sur lequel nous nous basons pour définir les prédicats/règles de réécritures, il est construit à travers l'opérateur *opSoS* comme suit :

$$opSoS < SoS1, null > .P[opsubSyS < S1, R1 > .L[null] | opsubSyS < S2, R2 > .L[null] | opsubSyS < S3, R3 > .L[null]].L[null].E[E1]$$

Cela indique qu'un SoS appelé SoS1, est constitué de deux sous systèmes S1, S2 et S3, ayant respectivement les rôles R1, R2 et R3, et pas de liens actifs (le lien est mis à *null*. SoS1 a pas de rôle ou de liens actif (valeur null attribuée aux deux champs). Nous notons la présence d'un événement E1 dans cet SoS.

Le tableau 6.2 décrit les prédicats de contrôles (voir tableau 6.1) dans le module ArchSoSBehaviours, qui va inclure le module fonctionnel ArchSoSFunctional. Dans ce tableau, SoSi est une variable qui fait référence à un SoS quelconque, de la même façon sont définies les variables (Si,Sj), (Ei,Ej),(Ri,Rj) et (Li, Lj) pour différencier des sous-systèmes, des événements, des rôles ou encore des liens respectivement.

Nous notons que certains prédicats sont spécifiques au SoS étudié, et doivent être impérativement définis en dépendant des constituants de cet SoS. A titre d'exemple, le prédicat *CompRoles*, qui prend en argument deux sous-systèmes, doit être établie comme une équation pour chaque deux rôles appartenant à deux systèmes différents d'un SoS, afin de définir une liste d'équations qui indique les combinaisons possibles de rôles. Cependant, le prédicat *isActive* par exemple indique si un SoS quelconque est actif (ayant un rôle différent de null) ou non, et sa peut être appliqué à n'importe quel SoS.

Une fois les prédicats de contrôles bien définis, nous passons à la définition de l'ensemble de règles de réécritures conditionnés par ces prédicats. D'une façon générale, des scénarios d'exécution sont choisis par un concepteur, et les règles seront appliquées pour correspondre à ces scénarios. Dans le cadre d'illustrer toutes les règles de réécritures et leurs syntaxes, nous allons représenter l'ensemble des règles pour l'exemple étudié. Le tableau 6.3 regroupe les différentes règle de réécritures conditionnées (mot clé *crl*) pour l'exemple. Prenons le cas d'une règle qui permet de lier les sous-systèmes S1 et S2 dans exemple, qui est labellée LinkSystemEi. Nous remarquons qu'elle est appliquée sur l'état :

$$opSoS < SoS1, null > .P[opsubSyS < S1, R1 > .L[null] | opsubSyS < S2, R2 > .L[null] | opsubSyS < S3, R3 > .L[null]].L[null].E[E1]$$

Le résultat est un autre état délimité par le symbol \Rightarrow qui est :

$$opSoS < SoS1, null > .P[opsubSyS < S1, R1 > .L[link < Ei : e; S1 --> S2 >] | opsubSyS < S2, R2 > .L[link < SEi : type; R1 --> R2 >] | opsubSyS < S3, R3 > .L[null]].L[null].E[E1]$$

Cela ce traduit par l'établissement d'un lien "link < Ei : e; S1 --> S2 >" qui représente un échange de données (type "e") entre les sous-systèmes S1 et S2 avec l'événement Ei. Le lien remplit les champs L des deux sous-systèmes à la fois. L'établissement du lien est conditionné par trois prédicats en conjonction par le mot clé *and* :

- (1) $\text{if PosLink}(\text{opsubSyS} < S1, R1 > .L[Li], \text{opsubSyS} < S2, R2 > .L[Lj], Ei, e) == \text{true}$: Vérifie si les sous-systèmes S1 et S2 sont compatibles pour être liés en réponse à l'événement Ei, avec un lien de type "e".
- (2) $\text{isLinked}(\text{opsubSyS} < S1, R1 > .L[\text{null}]) == \text{false}$: Vérifie si le système S1 est déjà lié.
- (3) $\text{isLinked}(\text{opsubSyS} < S2, R2 > .L[\text{null}]) == \text{false}$: Vérifie si le système S2 est déjà lié.

De la même façon, nous établissons le reste des règles de réécritures pour définir toutes les actions d'ArchSoS, tout en définissant les contraintes par les prédicats de contrôles, et en respectant la syntaxe définie précédemment.

6.3 Réécriture et exécution d'une spécification ArchSoS

Nous avons défini jusqu'à présent une spécification basée Maude des SoS dans ArchSoS. Cette spécification contient une syntaxe formelle pour la structure d'un SoS sous forme d'une théorie équationnelle décrite dans un module fonctionnel. Ce module est enrichi par une sémantique dynamique illustrée par des règles de réécritures conditionnelles Maude. Ces dernières sont conditionnées par des équations Maude, représentant ainsi les prédicats de contrôles pour les évolutions comportementales des SoS. Cette spécification permet de construire des SoS et les éléments qui les constituent dans ArchSoS, et d'exécuter leurs comportements en appliquant les règles de réécritures spécifiques sur un état donné d'un SoS, afin d'obtenir des états finaux et futurs à partir de cet état, tout en garantissant avec les prédicats de contrôles, le bon fonctionnement de ces comportements. Cela permet aux SoS d'avoir un comportement plus au moins autonome.

Cependant, l'application des règles de réécritures est spontanée : supposant que nous avons deux événements qui affectent l'état initial d'un SoS au même temps, on aura donc deux scénarios d'exécution, où le SoS répond dans chacun à l'événement spécifique pour accomplir une mission. Dans le cas où ces missions sont concurrentes, c'est à dire les constituants peuvent affecter l'un l'autre, le moteur d'exécution Maude va uniquement exécuter un seul scénario à partir de l'état initial donné.

Pour pallier à ce problème, il est indispensable d'avoir un mécanisme qui permet de contrôler le processus d'exécution dans Maude d'une façon dynamique, en reconfigurant les comportements des SoS pour faire face à des scénarios concurrents indépendamment. Le langage de stratégies Maude permet d'effectuer cela, en utilisant une stratégie Maude pour chaque scénario et pour chaque chemin d'exécution.

Le module de stratégies *ArchSoSStrategies* sert à définir ces stratégies dans ArchSoS. En incluant le module systèmes contenant les règles de réécritures *ArchSoSBehaviours*, le module de stratégie permet d'utiliser ces règles pour définir des chemins d'exécution, en les combinant avec les opérateurs du langage de stratégies.

Une stratégie qui répond à un événement donné est déclaré par :

$$\text{strat nom} - \text{stratégie} @ \text{SoS}.$$

Cette déclaration indique qu'une stratégie, spécifiée par un nom unique, sert à agir sur une sorte de type SoS. Ensuite, on a deux types de stratégies :

- Stratégie pour une mission d'un SoS : C'est une stratégie qui combine un ensemble de règles de réécritures Maudes, afin d'établir un chemin d'exécution désiré pour répondre à un événement donné, et réaliser la mission correspondante. Par conséquent, à chaque mission est associée une stratégie particulière. La définition de cette stratégie possède la syntaxe :

$$\text{sd Mission}_i\text{S} := (R1 ; R2 ; \dots ; RN) .$$

TABLE 6.3: Implementation des actions d'évolution

Actions dans ArchSoS	Règle de réécriture conditionnée Maude
Link_System($Ei, e : S1 \rightarrow S2$)	$\text{crl [Link-Ei] : opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS<S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1] \Rightarrow}$ $\text{opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow R2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow R2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1]}$ $\text{if (PosLink(opsubSyS< S1 , R1 >.L[Li] , opsubSyS< S2 , R2 >.L[Lj],Ei, e) == true and isLinked(opsubSyS< S1 , R1 >.L[null]) == false and isLinked(opsubSyS< S2 , R2 >.L[null]) == false) .}$
Destroy_Link($Ei, e : S1 \rightarrow S2$)	$\text{crl [DestroyLinkEi] : opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1] \Rightarrow}$ $\text{opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1]}$ $\text{if areLinked(opsubSyS< Si , Ri >.L[Li] , opsubSyS< Sj , Rj >.L[Lj], Ei, e) == true .}$
Acquire_Role($R1 : S1, R2 : S2$)	$\text{crl [AcquireRoleEi] : opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1] \Rightarrow}$ $\text{opSoS<SoS1, R1 + R2 >.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1]}$ $\text{if isActive(opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1]) == false and CompRoles (R1, R2) == true) .}$
Delete_Role($R1 : S1, R2 : S2$) $Sj, Sk,$	$\text{crl [DeleteRoleEi] : opSoS<SoS1, R1 + R2 >.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1] \Rightarrow}$ $\text{opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow R2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1]}$ $\text{if isActive(opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S2,R2>.L[link< Ei : e; S1 \dashrightarrow S2 >] opsubSyS <S3,R3>.L[null]].L[null].E[E1]) == true .}$
Add_System($SoS1, S4$)	$\text{crl [AddSystemEi] : opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1] \Rightarrow}$ $\text{opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null] opsubSyS <S4,R4>.L[null]].L[null].E[E1]}$ $\text{if canAdd(opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1], opsubSyS <S4,R4>.L[null]) == true .}$
Remove_System($SoS1, S3$)	$\text{crl [RemoveSystemEi] : opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1] \Rightarrow}$ $\text{opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null]].L[null].E[E1]}$ $\text{if CanRemove(opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1], opsubSyS <S3,R3>.L[null]) == true .}$
Replace_System($SoS, S1, S4$)	$\text{crl [ReplaceSystemEi] : opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1] \Rightarrow}$ $\text{opSoS<SoS1,null>.P[opsubSyS<S4,R4>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1]}$ $\text{if canReplace (opSoS<SoS1,null>.P[opsubSyS<S1,R1>.L[null] opsubSyS <S2,R2>.L[null] opsubSyS <S3,R3>.L[null]].L[null].E[E1], opsubSyS<S1,R1>.L[null] , opsubSyS<S4,R4>.L[null]) == true .}$

Cela indique qu'une stratégie appelée $Mission_iS$, établie pour $mission_i$, contient un ensemble de règles de réécritures séquencées avec l'opérateur de concaténation ";". Les règles seront exécutées l'une après l'autre dans cette stratégie. Ensuite, il faut établir d'autres stratégies $\{Mission_jS, Mission_kS \dots Mission_nS\}$ pour chaque mission j jusqu'à la mission n .

- Stratégie globale du SoS : Une fois les stratégies de chaque mission établie, il suffit d'avoir une stratégie globale pour définir le comportement du SoS, qui va regrouper les chemins d'exécution de chaque mission indépendante. La définition de la stratégie globale a la forme :

$$sdSoSS := (Mission_iS \mid Mission_jS \mid \dots \mid Mission_nS).$$

Cette notation indique que la mission globale d'un SoS, contient les stratégies $\{Mission_iS, Mission_jS \dots Mission_nS\}$, séparées par l'opérateur d'union "|". Cet opérateur permet d'exécuter toutes les stratégies de chaque mission en **parallèle**, offrant plusieurs solutions possibles, et ainsi plusieurs chemins d'exécutions possibles en même temps sans que l'un affecte l'autre. Par contre, la première mission à gauche a une **priorité** comparée à la deuxième.

6.4 Analyse formelle d'un SoS

Le système Maude est équipé de plusieurs autres outils et techniques d'analyse formelle afin de vérifier l'absence d'erreurs dans les comportements des SoS, qui est une tâche critique pour assurer la fiabilité des spécifications proposées. Nous récurons à la méthode de "model-checking" [Chechik and Gannon, 2001], [Baier and Katoen, 2008]. Cette méthode consiste à définir un ensemble de propriétés, et vérifier, d'un point de vue qualitatif, si elles sont satisfaites en faisant une analyse automatique sur les comportements des SoS.

Les propriétés représentent souvent des exigences attendues d'un système, et nous les décrivons par des formules LTL. La vérification d'une propriété donne toujours deux résultats : (1) Elle est valide et satisfaite, retournant la valeur True (vrai) pour indiquer que la propriété a tenu durant l'analyse automatique et exhaustive sur l'ensemble des états d'un SoS. (2) La propriété est invalide/n'est pas satisfaite en retournant un contre-exemple montrant le chemin où la propriété tient pas [Basin et al., 2011].

Les propriétés sont vérifiées à partir d'un état initial donné d'un SoS, vers des états accessibles à partir de cet état en appliquant un ensemble d'actions (règles de réécritures), jusqu'à obtenir un état final.

Le processus de définition des propriétés concernant les comportements des SoS dans Arch-SoS, qui est représenté par le module *ArchSoSAnalysis*, est divisé en deux étapes :

1/ Définition des prédicats d'états pour les SoS : Les prédicats d'états sont des équations conditionnelles Maude, indiquant différents états possibles d'un SoS. Les prédicats sont établis via la relation de satisfaction \models . Ils servent par la suite comme éléments d'une propriété Maude soumise à une vérification. Nous détaillons dans ce qui suit ces prédicats :

- **inactivity** : Un prédicat d'état qui indique qu'un SoS quelconque n'est pas actif. Nous le notons dans Maude comme suit :

$$ceq \text{ opSoS } \langle \text{SoSi}, Ri \rangle .P[\text{SoSV}i|\text{SoSV}j].L[Li].E[Ei] \models \text{inactivity} = \text{true if } (Ri == \text{null}).$$

Cela indique qu'un SoS ayant un identificateur (comme SoSi), un rôle Ri et des constituants, est inactif si son rôle Ri est égal à *null*.

- **roles-violation** : Indique s'il existe une combinaison de rôle qui est pas censée être établie (prohibée par la contrainte comportementale "incompatible-roles", le prédicat de contrôle *comproles* dans Maude). Nous le définissons ci-dessous :

$$ceq opSoS < SoSi, Rj+Rk > .P[opSoS < SoSj, Rj > .P[Sui|Suj].L[Lj].E[Ej]|opSoS < SoSk, Rk > .P[Suk|Sum].L[Lk].E[Ek]|SoSVn].L[Li].E[Ei] \models roles-violation = true\ if\ ((CompRoles(Rj, Rk)) == false).$$

Ça illustres un SoS identifié par l'id SoSi, contenant un rôle combiné qui est Rj+Rk, ainsi que trois autres SoS qui sont SoSj, SoSk et une variable qui abstrait un SoS SoSVn . Chacun des deux SoS SoSj et SoSk constituants a deux variables illustrant deux sous-systèmes respectifs (Sui, Suj et Suk, Sum). Le prédicat est valide si la combinaison de rôles (Rj + Rk) au sein du SoS est valide.

- **links-violation** : Dans le même sens que *roles-violation*, ce prédicat sert à identifier les liens illégaux ou "incompatible-links" dans l'évolution d'un SoS dans ArchSoS. Il est défini par :

$$ceq opSoS < SoSi, Ri > .P[opSoS < SoSj, Rj > .P[Sui|Suj].L[link < Ei : type; SoSj - - > SoSk >].E[Ej]|opSoS < SoSk, Rk > .P[Suk|Sum].L[link < Ei : type; SoSj - - > SoSk >].E[Ek]|SoSVn].L[Li].E[Ei] \models links-violation = true\ if\ ((PosLink(opSoS < SoSj, Rj > .P[Sui|Suj].L[null].E[Ej], opSoS < SoSk, Rk > .P[Suk|Sum].L[null].E[Ek], Ei, type) == false)).$$

La partie gauche de l'équation représente le SoS du prédicat précédent à l'exception de la combinaison de rôles existantes, mais avec l'établissement d'un lien entre les deux SoS *SoSj* et *SoSk*, avec un type quelconque et suite a un événement *Ei* du SoS *SoSi*. Le prédicat vérifie dans ce cas si ce lien établie est incompatible par en appliquant le prédicat de contrôle *PosLink* sur les deux SoS concerné avec l'événement qui a causé le lien, et le type de ce lien. *links-violation* retourne True (vrai) si *Poslink* est pas satisfait alors qu'un lien est déjà établie.

- **no-links** : Un prédicat qui délimite un état ou tout les systèmes d'un SoS peuvent pas être liés, en appliquant le prédicat de contrôle *NoClinks* sur un SoS quelconque Variable d'SoS SoSVn), il est décrit par :

$$ceq SoSVi \mid = no-links = true\ if\ NoClinks(SoSVi) == true.$$

- **no-roles** : Un prédicat qui indique qu'il n'existe pas de systèmes pouvant combiner leurs rôles dans un SoS.

$$ceq SoSVi \mid = no-roles = true\ if\ NoComp(SoSVi) == true.$$

- **adding, replacing** : Les états du SoS dans les quels des sous-systèmes sont rajoutés ou remplacés en utilisant les prédicats de contrôle *CanAdd*, *CanRemove* et *CanReplace*.

- **iMission** : Représente l'état d'un SoS ou la combinaison de rôles nécessaires a l'achèvement d'une mission *i* est présente comme rôle actif du SoS.

$$ceq opSoS < SoSi, Ri > .P[SoSVi|SoSVn].L[Li].E[Ei] \models iMission = true\ if\ (Ri == Role_{id1} + Role_{id2}).$$

$Role_{id1} + Role_{id2}$ est la combinaison de deux rôles existants au sein des constituants du SoS, ils sont à remplacer par les rôles nécessaires à chaque mission *i*, donc nous aurons autant de propriétés *iMission* que de mission à vérifier pour un SoS donné .

2/ Définition des formules LTL des propriétés à vérifier : Les propriétés des comportements des SoS à vérifier dans Maude sont identifiés par des formules LTL. Une formule LTL est une équation Maude regroupant les prédicats d'états avec l'ensemble des opérateurs LTL ($[]$, $\langle \rangle$, $->$ et \sim). Ces formules définissent des propriétés dynamiques sur l'évolution d'un SoS, nous les notons dans ce qui suit :

- **vivacity** : Une formule exprimant une propriété qui garantit que quelque chose de bien sera toujours produite (vivacité). Elle vérifie que chaque SoS sur lequel le prédicat d'inactivité tient, finira toujours par avoir un état où il est actif (négation du prédicat inactif). Nous la définissons dans Maude par :

$$eq \text{ vivacity} = [] (\text{inactivity} -> \langle \rangle \sim \text{inactivity}).$$

- **safety** : Elle assure la bonne évolution des comportements SoS, en respectant les contraintes comportementales définies. Cela est effectué par la vérification des deux contraintes comportementales pour les liens et les rôles. La propriété est dite valide si les deux contraintes (conjonction des négations des prédicats links-violation et roles-violation) ne sont pas violées à travers tous les états d'évolution du SoS. Elle est notée :

$$eq \text{ safety} = [] \langle \rangle ([] \sim \text{roles} - \text{violation} / \wedge [] \sim \text{links} - \text{violation}).$$

- **consistency** : Permet à un SoS de qui satisfait la propriété de vivacité de passer vers un état actif où une mission donnée (iMission dans ce cas, à remplacer par le prédicat de la mission qu'on veut vérifier) est achevée. Elle a la syntaxe :

$$eq \text{ consistency} = [] (\sim \text{vivacity} / \wedge \langle \rangle \text{iMission}).$$

- **nodeadlinks** : Indique quand tous les systèmes d'un SoS peuvent pas être liés et qu'un ajout d'un système est disponible, ce système sera ajouté pour finir par avoir un état ou il existe au moins un lien possible dans le SoS.

$$eq \text{ nodeadlinks} = [] (\text{no-links} / \wedge \text{adding} -> \langle \rangle \sim \text{no-links}).$$

- **nodeadroles** : Quand un SoS est dans un état ou tous ses constituants ne peuvent pas combiner leurs rôles, et qu'un remplacement d'un système par un autre est envisageable par un autre, cela résulte par avoir des systèmes ayant des rôles compatibles dans le SoS.

$$eq \text{ nodeadroles} = [] (\text{no-roles} / \wedge \text{replacing} -> \langle \rangle \sim \text{no-roles}).$$

Une fois que nous finalisons ses deux étapes pour la création et la définition des propriétés de vérification pour les SoS dans ArchSoS, Nous aurons tous les éléments nécessaires pour lancer la vérification dans l'outil model-checker de Maude. Nous importons le module MODEL-CHECKER dans ArchSoSAnalysis. La commande qui permet d'exécuter la vérification des propriétés est la suivante :

$$red \text{ modelCheck}(\text{initial}, \text{LTLoperators } P)$$

Où *red modelCheck* est la commande qui lance le model-checker. *initial* est l'état initial d'un SoS donné, que nous déclarons dans le module ArchSoSAnalysis. *P* est la propriété à vérifier, et *LTLoperators* sont des opérateurs LTL qui précède cette propriété pour décrire si elle doit être globalement valide, ou uniquement présente une fois dans l'évolution des états du SoS. Le résultat d'exécution est soit la valeur True pour montrer la validation, ou un contre-exemple qui indique sa non-satisfaction.

6.5 Conclusion

Dans ce chapitre, nous avons défini une sémantique opérationnelle à ArchSoS en utilisant le langage formel Maude. Cette définition respecte les règles de formations structurelles définies dans les bigraphes, donc préserve naturellement la constitution correct-par-construction des SoS dans ArchSoS.

Nous avons introduit des stratégie Maude dans un module de stratégies, afin d'enrichir l'aspect sémantique d'ArchSoS par une possibilité de gérer la reconfiguration dynamique dans les SoS. Enfin, nous avons procédé à la vérification formelle qualitative basée sur la technique du model-checking. Dans le chapitre suivant, nous présenterons une étude de cas complète pour illustrer toutes nos contributions, qui consiste à modéliser l'architecture logicielle d'un SoS de réponses de crise.

Aspects d'implémentation

Sommaire

7.1	Introduction	88
7.2	Etude de cas (Crisis Response SoS)	89
7.2.1	Syntaxe concrète de CRSoS	90
7.2.2	Syntaxe abstraite de CRSoS	93
7.2.3	Sémantique opérationnelle basée Maude de CRSoS	95
7.3	Évaluation	99
7.3.1	Simulation et reconfiguration dynamique de CRSoS	99
7.3.2	Vérification qualitative des comportements de CRSoS	102
7.4	Vers un framework pour l'implémentation d'ArchSoS	104
7.4.1	Phase de spécification	106
7.4.2	Phase de simulation :	110
7.4.3	Phase d'analyse	111
7.5	Conclusion	112

7.1 Introduction

Les systèmes de réponse aux crises constituent un bon nombre d'SoS complexes d'aujourd'hui ; ils génèrent des situations caractérisées par des conséquences néfastes, une faible probabilité et un temps de décision court. La réponse à la crise en général représente un défi et doit être consciencieusement soutenue. Ces systèmes impliquent plusieurs parties, chacune avec sa propre autonomie et ses propres capacités, ce qui conduit à des différenciations dans la structure, les objectifs et les stratégies, ainsi que les contraintes de coopération [Van Veelen et al., 2008].

Par conséquent, un SoS de réponse aux crises (ou CRSoS : Crisis Response SoS) doit prendre en charge une adaptation distribuée et continue en raison de changements imprévisibles dans les objectifs et les plans de l'organisation actuelle de gestion des crises. Différentes approches d'adaptation et de (re)configuration doivent être gérées pour structurer, configurer et reconfigurer dynamiquement les organisations d'un CRSoS, et se conformer ainsi aux objectifs réels et aux accords sur les plans dans des conditions de gestion de crise.

Dans ce chapitre, nous considérons CRSoS comme une étude de cas réelle, sur laquelle nous appliquons les modèles et les implémentations tirés de nos contributions. Les auteurs dans [Hachem et al., 2020] indiquent que l'utilisation d'une nouvelle méthodologie pour implémenter un système existant peut être considérée comme un moyen idéal de validation, et fournit des

informations utiles sur les avantages et limites de la solution. Dans le même cas d'esprit, nous procédons à cette étape à la validation de notre solution ArchSoS. La section 7.2 présente une vue globale de l'étude de cas, afin de la spécifier dans ArchSoS. Les résultats de la simulation, de reconfiguration et de vérification formelle de CRSoS dans ArchSoS sont décrits et discutés dans la section 7.3.

Finalement, nous abordons dans la section 7.4 l'implémentation d'un Framework autour du langage.

7.2 Etude de cas (Crisis Response SoS)

CRSoS est un SoS qui a pour objectif de réduire l'effet de surprise après une crise ou une catastrophe, produisant en moins de temps, la réaction appropriée pour fournir une assistance immédiate, des avertissements et des évacuations nécessaires. CRSoS regroupe un ensemble de systèmes indépendants qui doivent coopérer pour atteindre cet objectif, bien que ces systèmes soient totalement autonomes, complexes, et possiblement hétérogènes.

L'approche SoS est justifiée par la capacité des constituants à répondre à une crise qu'aucun d'eux ne peut gérer tout seul. Si nous considérons différentes crises comme des événements externes, qui ont un temps de décision court et se produisent sans préavis, le besoin que les systèmes constituants réagissent à ces événements est essentiel. Ils doivent être dynamiquement adaptatifs et mobiles afin de tirer de nouveaux comportements en fonction de ces événements survenants. Ainsi, la gestion de l'évolution des constituants de CRSoS et de leurs coopérations est vitale; ils doivent interagir en étant liés ensemble et en combinant leurs fonctions pour atteindre les missions du CRSoS.

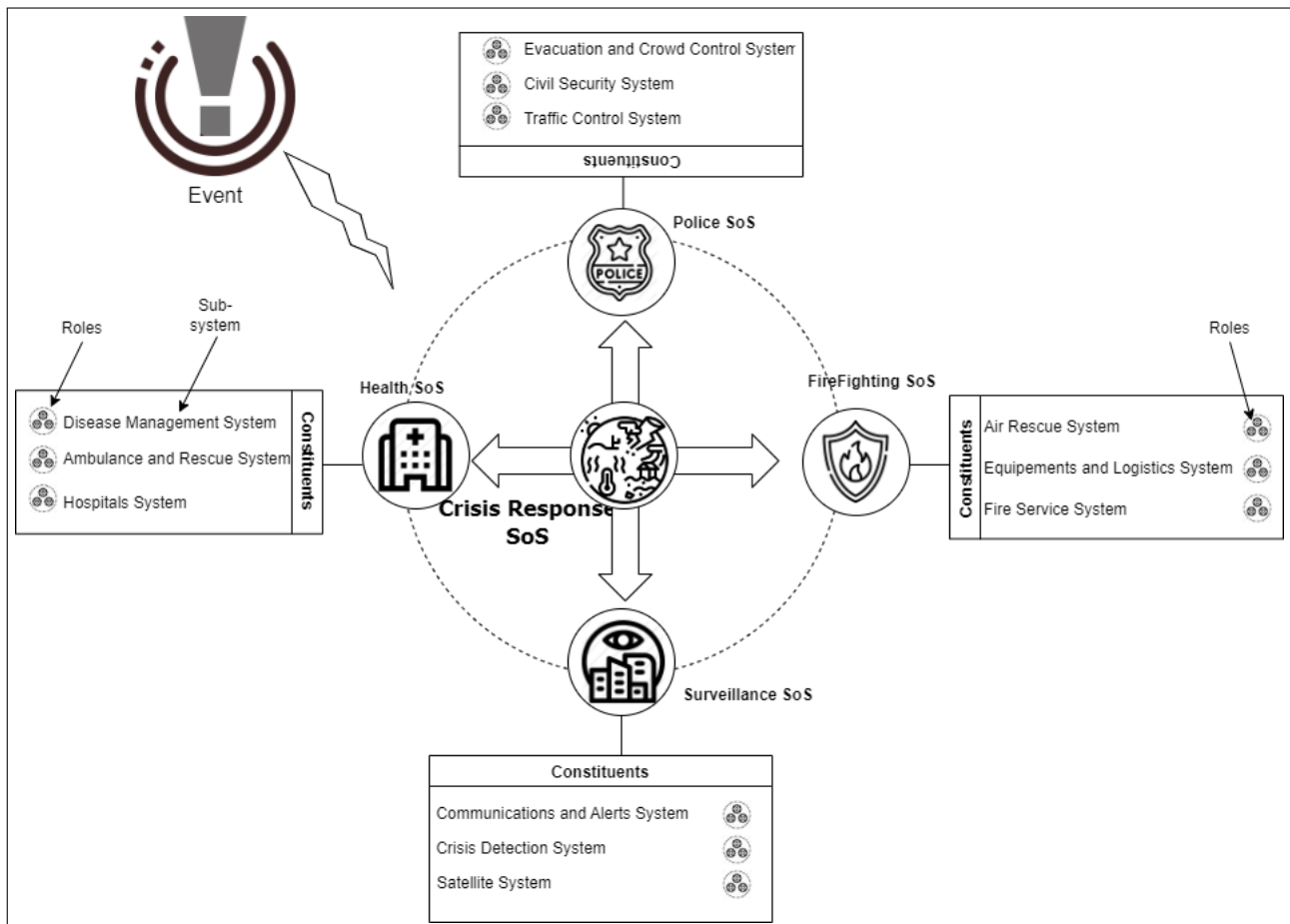


FIGURE 7.1: Vue d'ensemble des constituants de CRSoS

CRSoS, en étant ainsi réactif et reconfigurable aux événements pour atteindre des missions, offre plusieurs solutions pour préserver la vie humaine, et pour minimiser les dommages une fois une crise détectée et signalée. Les systèmes constituant de CRSoS ont des objectifs et des rôles différents mais ils appartiennent à cet SoS qui leur permet de réaliser des objectifs communs. Toutefois, ils maintiennent encore leur indépendance, leur autonomie de gestion, et leur propre nature évolutive à part.

La figure 7.1 offre une vue globale de CRSoS et de ses constituants : il contient quatre autres SoS qui sont : Police (SoS de police), Surveillance (SoS de surveillance de crises), Health (SoS de santé) et FireFighting (SoS de pompiers). Chacun de ses SoS contient un ensemble de sous-systèmes atomiques, ayant des rôles et des fonctionnalités prédéfinies. Par exemple, le SoS de Surveillance possède les sous-systèmes : "Communications and Alerts", "Crisis Detection", et "Satellite", où chaque sous-système a ses propres rôles.

7.2.1 Syntaxe concrète de CRSoS

Nous définissons dans ce qui suit CRSoS en utilisant la syntaxe concrète d'ArchSoS. Dans la figure 7.2, nous illustrons sa déclaration. La première ligne spécifie CRSoS et liste tout ses constituants.

La clause "Format" décrit : (1) le type de chaque système avec la balise **Systems_Type**. Les constituants peuvent être eux même des **SoS**, ou bien des sous-systèmes avec le mot clé **Sub_System**, et (2) la hiérarchie du SoS via une relation d'appartenance entre les systèmes constituants, déclarée dans la balise **parent_of**.

A titre d'exemple, *CRSoS parent_of Health* indique que le SoS de santé *Health* est un constituant de CRSoS, tandis que *Health parent_of Air Rescue* indique que le sous-système *Air Rescue* est un constituant du SoS *Health*.

La clause "Content" définit le contenu de CRSoS et de ses constituants en termes de rôles (fonctionnalités), de contraintes qui guident le comportement de CRSoS, et les missions qu'il est censé achever, incluant les événements, les liens et les rôles impliqués dans ces missions. Des exemples de contraintes sont présentés dans le tableau 7.1. La première contrainte " \sim Link(Surveillance, Health, Fire, a)" indique qu'un lien d'autorité ne peut pas être établie entre les SoS *Health* et *Surveillance* en réponse à un événement de feu.

De plus, la contrainte notée " \sim [(DiseaseM : Disease Management), (Evacuation : Evacuation and crowd control)]" indique que les rôles *DiseaseM* et *Evacuation*, appartenant respectivement aux sous-systèmes *Disease Management* et *Evacuation and crown control*, ne peuvent pas être combinés pour atteindre une mission de CRSoS.

TABLE 7.1: Contraintes comportementales de CRSoS

Contrainte	Syntaxe
Liens incompatibles (Incompatible Links)	\sim Link(Surveillance, Health, Fire, a) \sim Link(Surveillance, FireFighting, Fire, a) \sim Link(Surveillance, Health, Fire, u)
Rôles incompatibles (Incompatible Roles)	\sim [(Equipements : Equipement and logistics),(detection : Crisis Detection)] \sim [(DiseaseM : DiseaseManagement),(TrafficC : TrafficC]


```

ArchSoS SCRSoS of Health; FireFighting; Surveillance; Police; Disease
Management...; Crisis Detection...; Air Rescue...; Evacuation and Crowd
Control...etc.
Format
  Systems_Type
    Health: SoS; FireFighting SoS; ... CrisisDetection: Sub_System;
    Air Rescue: Sub_System
    ...
  Hierarchy
    SCRSoS parent_of Health; FireFighting; Surveillance; Police/
    Health parent_of Air Rescue/
    Surveillance parent_of Crisis Detection; Communications and
    alerts system /
    Police parent_of Evacuation and Crowd Control
    FireFighting parent_of Fire Service; Equipment and logistics
    ...
Content
  Roles
    DiseaseM: Disease Management; Evacuation: Evacuation and
    Crowd Control; Crisis Detection: detection;
    Communications and alerts: communication; Fire Service:
    Ffighting; Equipment and logistics: equipment;
    ...
  Behavioural_Constraints:
    Incompatible_Links: ~Link(Surveillance, Health, Fire, a)...
    Incompatible_Roles: ~[(DiseaseM: Health), ( TrafficC:
    Police)] ...
  Missions
    FireDistinguish Mission {
      Events: Fire Appearance, FireSignal, Fire
      Links:
        Fire Appearance, e: Crisis detection → Communications and
        alerts;
        FireSignal, u: Fire Service →Equipment and logistics;
        Fire, e: Surveillance → FireFighting
      Roles_Combined
        detection + communication + Ffighting + equipement
    }
    ...
EndArch

```

FIGURE 7.2: Déclaration de CRSoS dans ArchSoS

Un exemple de missions considérée dans cette étude de cas pourrait être : l'extinction de feu "*FireDistinguish*", pouvant être liée à trois événements possibles *Fire*, *FireAppearance* et *Firesignal*.

Scénarios d'exécution de CRSoS : Pour illustrer le concept de missions dans CRSoS et son évolution selon certains événements, nous définissons deux scénarios possibles de deux crises survenant en même temps. CRSoS doit être reconfiguré pour répondre aux événements et accomplir deux missions correspondantes qui sont : *FireDistinguish* et *HurricaneEvacuation*.

Nous rappelons qu'une mission est accomplie lorsque CRSoS arrive à combiner certains combinaison rôles, qui résultent des liens spécifiques qui ont été établis entre des systèmes composant le SoS en réponse à un nombre d'événements.

Les deux scénarios sont résumés dans le tableau 7.2, qui montre les SoS affectés avec certains événements, et quels systèmes constitutants sont reconfigurés. Cela conduit à réaliser deux

TABLE 7.2: Scénarios d'exécution de CRSoS

Événements	SoS affectés	Systèmes re-configurés	Liens établies	Rôles combinés
Scénario 1 : Mission <i>FireDistinguish</i>				
Fire-Appearance	Surveillance	Sous-système Crisis detection + Sous-système Communications and alerts	Link(Fire-Appearance : e, CrisisDetection, Communications and Alert)	R1 : (Surveillance SoS) [(detection :CrisisDetection :) + (:communication : Communications and Alert)]
FireSignal	FireFighting	Fire Service system + Sous-système Equipment and Sous-système logistics	Link(FireSignal : u, FireService, Equipment and logistics)	R2 : (FireFighting SoS) [(Fighting :FireService)+ (equipment :Equipment and logistics :)]
Fire	CRSoS	SoS Surveillance + SoS FireFighting	Link(Fire : e, Surveillance, FireFighting)	R : CRSoS [(R1 :Surveillance)+ (R2 :FireFighting)]
Scénario 2 : Mission <i>HurricaneEvacuation</i>				
Hurricane-Appearance	Surveillance	Sous-système Crisis detection + Sous-système Communications and alerts	Link(Hurricane-Appearance : e, CrisisDetection, Communications and Alert)	R1 : (Surveillance SoS) [(detection :CrisisDetection :) + (communication : Communications and Alert)]
Hurricane-Signal	Police	Sous-système Evacuation and crowd control + Sous-système Traffic control	Link(HurricaneSignal : u , Evacuation and crowd control, Traffic control)	R3 : (Police SoS) [(CManagement :Evacuation and crowd control) + (TManagement :Traffic control)]
Hurricane	CRSoS	SoS Surveillance + SoS Police	Link(Hurricane : e, Surveillance, Police)	R' : CRSoS [(R3 :Surveillance)+ (R4 :Police)]

missions distinctes de CRSoS. Nous remarquons que plusieurs reconfigurations sont nécessaires pour répondre à des événements extérieurs qui peuvent survenir en même temps.

Par exemple, pour atteindre la mission *FireDistinguish*, le *CRSoS* doit répondre à trois événements séquentiels, qui sont *FireAppearance*, *FireSignal* et *Fire*. Les systèmes constituants concernés (par exemple, *CrisisDetection*, *Communications and Alert*) sont liés avec un événement de type "e" en réponse à l'événement *Fireappearance*. Ensuite, ils ont leur rôles *detection* et *communication*, appartenant aux sous-systèmes *CrisisDetection* et *Communications and Alert* respectivement, combinés afin de faire émerger d'autres rôles, comme le rôle *R1* du SoS Sur-

veillance indiquant que la mission est accomplie.

A travers ces deux scénarios simultanés de CRSoS, nous gérons l'évolution de CRSoS et de ses constituent dans une hiérarchie structurée, en appliquant les actions d'évolution d'ArchSoS sur CRSoS pour achever des missions et répondre à des événements de crises. La mission *FireDistinguish* par exemple est achevée en ayant la combinaison des rôles *R1* et *R2*, appartenant aux SoS *Surveillance* et *FireFighting* respectivement. Le même principe est appliqué pour la mission *HurricaneEvacuation* et les événements qui la concernent. A cette étape et pour cet exemple, nous notons deux principaux défis qui peuvent surgir lorsque nous procédons à la gestion de plusieurs événements en même temps :

- 1/ CRSoS peut être reconfiguré pour répondre à plusieurs événements parallèles sans que les uns influencent les autres.
- 2/ CRSoS peut être reconfiguré pour répondre à une catastrophe et à une crise spécifique avant d'autres, en fonction de leur gravité ou leur ordre de réponse, par exemple, l'événement *FireAppearance* doit être traité avant l'événement *FireSignal*, et l'événement *Fire* devrait suivre après. Un autre exemple serait que l'événement *Fire* soit priorisé sur *Hurricane*, car naturellement un incendie a besoin d'une réponse plus rapide qu'un ouragan qui prend du temps pour faire des dommages.

7.2.2 Syntaxe abstraite de CRSoS

En respectant les règles de formation bigraphiques dans ArchSoS, les constituants de CRSoS sont modélisés selon deux vues : graphique et algébrique.

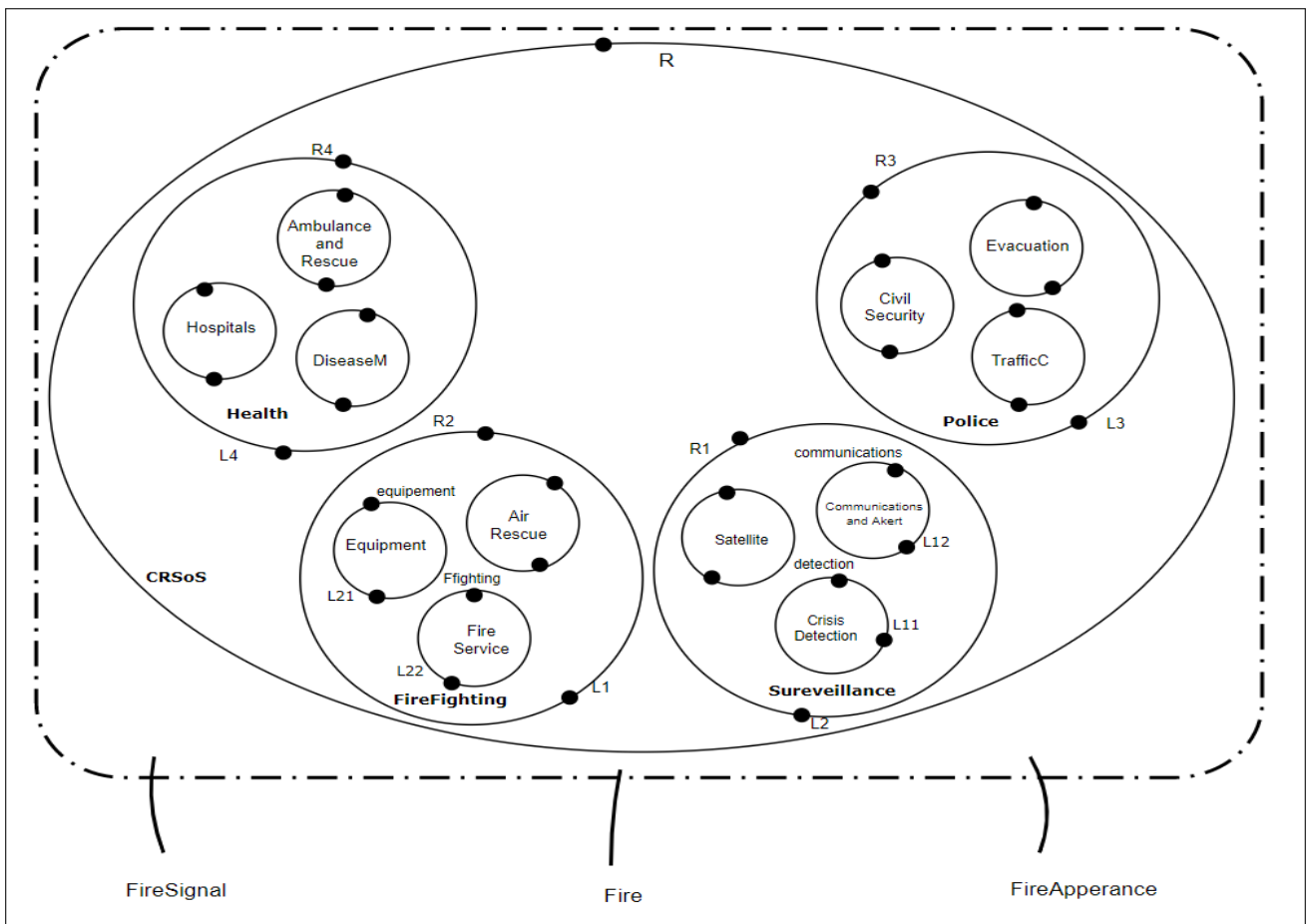


FIGURE 7.3: CRSoS selon la syntaxe abstraite d'ArchSoS : vue graphique

```

\FireAppearance \FireSignal \Fire CRSoS{R,L}.
(Surveillance{R1,L1}.(CrisisDetection{detection,L11} | Communications
and alert{communications,L12}.Satellite.... )
|FireFighting{R2,L2}. (Equipment and logistics{equipment,L21}
|FireService{Firefighting,L22})
|Police{R3,L3}.Evacuation.....
|Health{R4,L4}.Hospitals....)

```

FIGURE 7.4: CRSoS selon la syntaxe abstraite d'ArchSoS : vue algébrique

CRSoS est un nœud global contenant quatre autres nœuds, qui spécifient les quatre SoS qui le constituent, où chaque SoS contient un ensemble de sous-systèmes respectifs. A chaque SoS est attribué deux ports, un port de rôle initialement inactif R_i , et un port de liens L_i . En plus des ports des liens, les sous-systèmes ont de ports significatifs qui correspondent à leurs rôles, par exemple le sous-système *CrisisDetection* a le rôle *detection* (figure 7.4).

Nous définissons dans la figure 7.3 un état initial de CRSoS en respectant la syntaxe d'ArchSoS, où aucun système est lié, et aucun rôle est combiné, tous les SoS ont des rôles inactifs. CRSoS a aussi une vue graphique équivalente présentée dans la figure 7.4, qui offre une meilleure visualisation de ces constituants modélisés par un bigraphe.

Les noms internes dans la vue graphique, dont l'équivalent est la partie en gras de la vue algébrique, représentent les trois événements inclus dans le scénario. Les nœuds concernant les SoS sont représentés par les systèmes en rouges, tandis que les nœuds des sous-systèmes sont les systèmes en bleus. Les ports de chaque systèmes sont décrit via l'opérateur $\{...\}$.

Nous notons que cette description de CRSoS permet de décrire cet SoS structurellement avec précision, en incluant tous les constituants, leurs rôles, et leurs potentiels liens via les ports des liens et les événements.

L'aspect comportemental de CRSoS représente une évolution de sa structure. Plus précisément, nous appliquons les actions d'évolution d'ArchSoS, qui sont basées sur les règles de réactions bigraphiques, ainsi que les prédicats de contrôle qui les conditionnent, pour faire évoluer CRSoS de son état initial, vers des états futurs.

Dans notre exemple, nous appliquons un ensemble d'actions sur CRSoS afin de lui permettre d'accomplir le premier scénario concernant la mission *FireDistinguish*. D'après le tableau 7.2, nous avons exactement six actions à appliquer pour atteindre cette mission. Le résultat de cette application est défini par la figure 7.5. Nous détaillons dans ce qui suit ces actions :

- Un lien de type "e", causé par l'événement *Fire-Appearance* entre les sous-systèmes *CrisisDetection* et *Communications and Alert*. Ce lien est décrit par l'hyper-arc "e1".
- Une combinaison de rôles entre les rôles *detection* et *communications*, appartenant aux sous-systèmes *CrisisDetection* et *Communications and Alert*. Le rôle *R1* du SoS surveillance devient actif et aura comme fonctionnalité la combinaison des deux rôles.
- Un lien e type "u", provoqué par l'événement *FireSignal* entre sous-systèmes *FireService* et *Equipment and logistics*, et décrit par l'hyper-arc "u1".
- Le rôle *R2* du SoS *FireFighting* devient actif et aura la combinaison entre les rôles *Ffighting* et *equipment*, des sous-systèmes *FireService* et *Equipment and logistics*) respectivement.

- Un lien de type "e", entre les SoS *Surveillance* et *Firefighting* suite à l'événement *Fire*, l'hyper-arc "e2" représente ce lien.
- Finalement, une combinaison des rôles *R1* et *R2* est faite, qui sont eux même des fonctionnalités complexes des SoS *Surveillance* et *Firefighting*, résultants des combinaisons de rôles de leurs sous-systèmes respectifs. Le rôle *R* de CRSoS détient cette combinaison, dénotée par l'hyper-arc "x". Cette combinaison lui permet de répondre à l'événement *Feu* pour achever la mission *FireDistinguish*, qui nécessite cette combinaison de rôles particulière.

De la même manière, nous appliquons les actions du deuxième scénario pour aboutir à la mission *HurricaneEvacuation*.

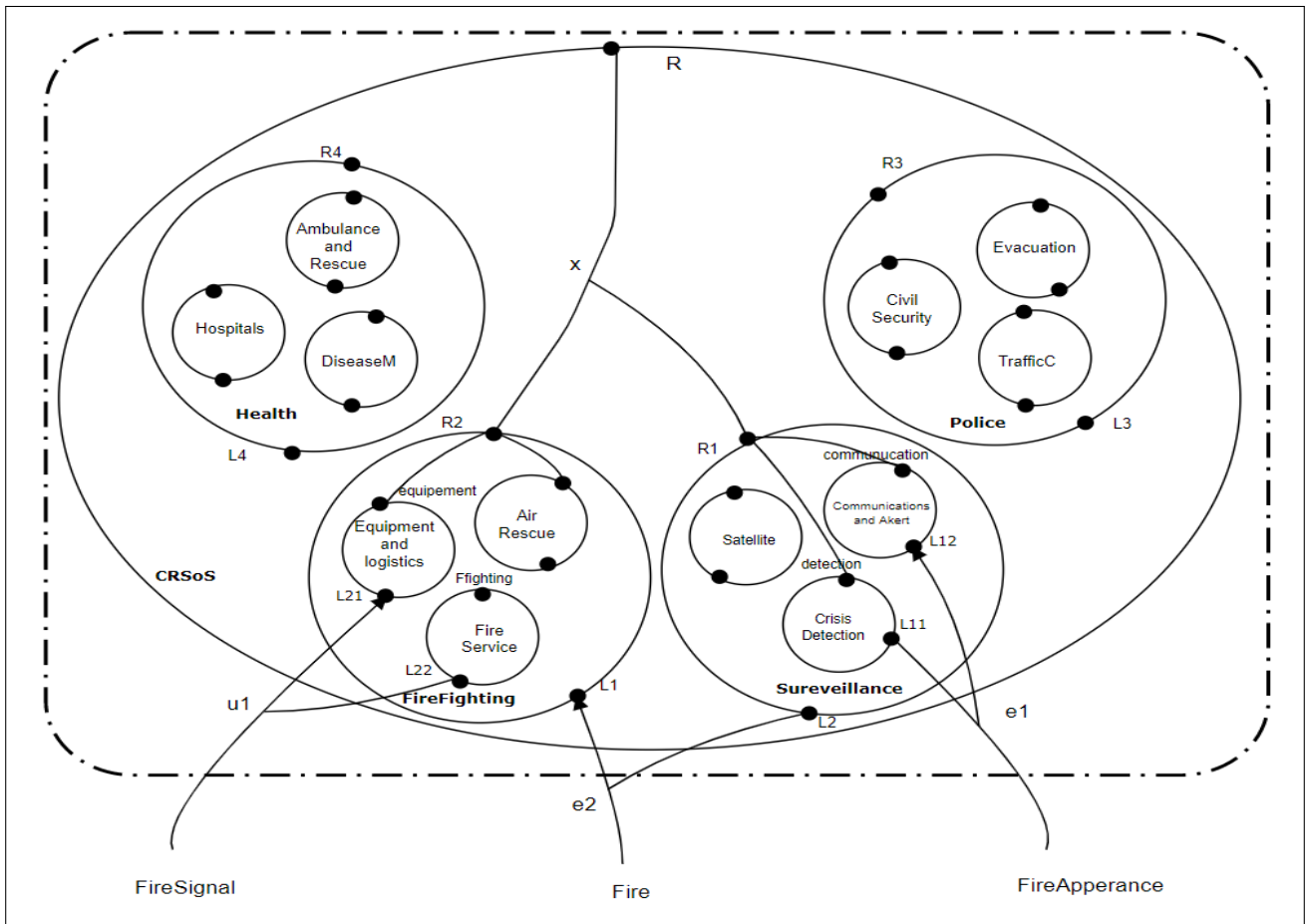


FIGURE 7.5: CRSoS selon la syntaxe abstraite d'ArchSoS : l'état final du scénario 1

7.2.3 Sémantique opérationnelle basée Maude de CRSoS

Dans cette partie, nousinstancions les trois modules *ArchSoSSyntax*, *ArchSoSBehaviours* et *ArchSoSStrategies* définissant la sémantique d'évolution de CRSoS, pour décrire respectivement l'aspect structurel, l'aspect comportemental, et la reconfiguration dynamique de CRSoS dans ArchSoS.

Dans le module *ArchSoSSyntax*, nous spécifions de plus des opérations qui permettent de définir les identifiants de chaque SoS et de chaque sous-système, ainsi que les événements, les types de liens possibles et l'ensemble des rôles (de sous-systèmes) qui seront utiles pour décrire CRSoS. La figure 7.6 démontre un bon nombre de ces opérations dans le module ArchSoSSyntax. Par exemple, l'opération $op\ Fire : \rightarrow Event [ctor]$ permet de définir l'événement *Fire* dans

CRSoS (sorte Event), tandis que $op\ CrisisControl : \rightarrow idSoS [ctor]$. définit l'identifiant (sorte idSoS) de CRSoS comme "CrisisControl". La figure 7.7 décrit un état initial de CRSoS spécifié par l'opération opSoS, qui dénote CRSoS et tous ses constituants d'une façon hiérarchique et structurée.

```

op a : -> LinkT [ctor] .
op u : -> LinkT [ctor] .
op e : -> LinkT [ctor] .

op Fire : -> Event [ctor] .
op FireAppearance : -> Event [ctor] .
op FireSignal : -> Event [ctor] .
op Hurricane : -> Event [ctor] .
op HurricaneAppearance : -> Event [ctor] .
op HurricaneSignal : -> Event [ctor] .

op communications : -> Role [ctor] .
op detection : -> Role [ctor] .
op equipements : -> Role [ctor] .
op Ffighting : -> Role [ctor] .
op FExtinguish : -> Role [ctor] .
op CManagement : -> Role [ctor] .
op TManagement : -> Role [ctor] .
op CProtection : -> Role [ctor] .

op CrisisControl : -> idSoS [ctor] .
op Police : -> idSoS [ctor] .
op Health : -> idSoS [ctor] .
op FireFighting : -> idSoS [ctor] .
op Surveillance : -> idSoS [ctor] .

op EvacuationCC : -> idsubSys [ctor] .
op CSecurity : -> idsubSys [ctor] .
op TrafficC : -> idsubSys [ctor] .

op DiseaseM : -> idsubSys [ctor] .
op AmbulanceR : -> idsubSys [ctor] .
op Hospitals : -> idsubSys [ctor] .

.....
    
```

FIGURE 7.6: Opérations d'identifications des éléments de CRSoS

Dans le module *ArchSoSBehaviours*, nous rajoutons les règles de réécritures conditionnées donnant la sémantique d'évolution des scénarios de CRSoS, où les conditions sont des équations Maude décrivant les prédicats correspondants. Ainsi, chaque scénario a six règles de réécritures

```

opSoS< CrisisControl , null >.P[
  opSoS< Surveillance , null >.P[
    opsubSys< CAlert , communications >.L[ null ]|
    opsubSys< CDetection , detection >.L[ null ] ]
    .L[ null ].E[ FireAppearance , HurricaneAppearance ]
| opSoS< FireFighting , null >.P[opsubSys< EquipementsL , equipements >.L[ null ] |
  opsubSys< FireS , Ffighting >.L[ null ] ]
  .L[ null ].E[ FireSignal , HurricaneSignal ]
| opSoS< Police , null >.P[opsubSys< TrafficC , TManagement >.L[ null ] |
  opsubSys< EvacuationCC , CManagement >.L[ null ] ].L[ null ].E[ null ] ]
.L[ null ].E[ Fire , Hurricane ] .

```

FIGURE 7.7: État initial de CRSoS avec tous les événements des deux scénarios

conditionnés, qui correspondent aux actions de chaque scénario (voir tableau 7.2). La première action de chaque scénario est appliquée sur le même l'état initial de CRSoS. Les règles sont labellisées (selon les événements de CRSoS) comme suit :

- Scénario 1 : LinkFireApperance, AcquireRoleFireAppearance, LinkFireSignal, AcquireRoleFireSignal, LinkFire, AcquireRoleFire.
- Scénario 2 : LinkHurricaneAppearance, AcquireRoleHurricaneApperance, LinkHurricaneSignal, AcquireRoleHurricaneSignal, LinkHurricane, AcquireRoleHurricane.

En plus des prédicats de contrôle standards dans ArchSoS, nous définissons les prédicats définissant les rôles/liens compatibles dans CRSoS, car ses prédicats dépendent de l'étude de cas considérée.

A titre d'exemple, nous avons deux équations *CompRoles* pour indiquer que les combinaisons de rôles (*communications*, *detection*) et (*Ffighting*, *equipements*) sont valides, aussi les liens possibles (de chaque événement *FireAppearance* et *HurricaneAppearance*) entre les sous-systèmes *CAlert* (Communications and Alert) et *CDetection* (Communications and Detection).

$$\begin{aligned}
 eqCompRoles(communications, detection) &= true. \\
 eqCompRoles(Ffighting, equipements) &= true. \\
 eqPosLink(opsubSys < CAlert, communications > .L[null], opsubSys < \\
 &CDetection, detection > .L[null], FireAppearance, e) = true. \\
 eqPosLink(opsubSys < CAlert, communications > .L[null], opsubSys < \\
 &CDetection, detection > .L[null], HurricaneAppearance, e) = true.
 \end{aligned}$$

Nous donnons dans la figure 7.8 le code en Maude de la règle *LinkHurricaneAppearance* qui permet de lier les deux sous-systèmes *CAlert* et *CDetection* en réponse à l'événement *HurricaneAppearance*. La première partie avant le symbole \Rightarrow représente l'état initial de CRSoS, tandis que la deuxième partie représente la transition résultante de l'application de l'action (règle) de lien.

Nous remarquons l'apparition du lien "*link < HurricaneAppearance : e; CAlert --> CDetection >*" dans la partie *L[...]* de chacun des deux sous-systèmes, indiquant qu'un lien de type "e", qui concerne l'événement *HurricaneApperance* est établi entre eux. Cette règle est uniquement appliquée si certaines conditions sont valides telles que : (1) si *CAlert* n'est pas déjà lié, (2) si *CDetection* n'est pas déjà lié, (3) et si le lien est possible.

Ces conditions sont respectivement prises en charge par les prédicats *isLinked*, qui doit retourner faux (false) pour chacun des systèmes, et par le prédicat *posLink* qui doit retourner

vrai (true) pour indiquer que les deux sous-systèmes sont compatibles pour avoir ce lien précis. De la même manière, nous définissons le reste des actions d'évolution de CRSoS sous forme de règles de réécritures conditionnées par les prédicats de contrôle, pour les deux scénarios distincts de CRSoS.

```

cr1 [ LinkHurricaneAppearance ] :
opSoS< CrisisControl , null >.P[
  opSoS< Surveillance , null >.P[
    opsubSys< CALert , communications >.L[ null ] | opsubSys< CDetection , detection >.L[ null ] ]
  .L[ null ].E[ FireAppearance , HurricaneAppearance ] |
  opSoS< FireFighting , null >.P[
    opsubSys< EquipementsL , equipements >.L[ null ] | opsubSys< FireS , Ffighting >.L[ null ] ]
  .L[ null ].E[ FireSignal , HurricaneSignal ] |
  opSoS< Police , null >.P[
    opsubSys< TrafficC , TManagement >.L[ null ] | opsubSys< EvacuationCC , CManagement >.L[ null ] ]
  .L[ null ].E[ null ] ]
.L[ null ].E[ Fire , Hurricane ]
=>
opSoS< CrisisControl , null >.P[
  opSoS< Surveillance , null >.P[
    opsubSys< CALert , communications >.L[ link< HurricaneAppearance : e ; CALert --> CDetection > ] |
    opsubSys< CDetection , detection >.L[ link< HurricaneAppearance : e ; CALert --> CDetection > ] ]
  .L[ null ].E[ HurricaneAppearance ] |
  opSoS< FireFighting , null >.P[
    opsubSys< EquipementsL , equipements >.L[ null ] | opsubSys< FireS , Ffighting >.L[ null ] ]
  .L[ null ].E[ null ] |
  opSoS< Police , null >.P[opsubSys< TrafficC , TManagement >.L[ null ] |
    opsubSys< EvacuationCC , CManagement >.L[ null ] ].L[ null ].E[ HurricaneSignal ] ]
.L[ null ].E[ Hurricane ]
if ( isLinked(opsubSys< CALert , communications >.L[ null ] ) == false
and isLinked( opsubSys< CDetection , detection >.L[ null ] ) == false
and PosLink(opsubSys< CALert , communications >.L[ null ] ,
  opsubSys< CDetection , detection >.L[ null ] , HurricaneAppearance , e ) == true ) .
    
```

FIGURE 7.8: Exemple de règle de réécriture conditionnée de CRSoS : la règle LinkHurricaneAppearance

Dans le troisième module *ArchSoSStrategies*, nous introduisant la déclaration d'un ensemble de stratégies qui guident l'évolution de CRSoS dans ArchSoS. Nous prévoyons une stratégie pour chaque scénario d'exécution, et une stratégie qui englobe le comportement global de CRSoS en incluant les deux scénarios en même temps. Nous notons que les stratégies sont appliquées sur l'état initial de CRSoS. La figure 7.9 décrit ce module de stratégies, où les trois stratégies en question sont déclarées pour agir sur des sortes de type SoS (strat ... @ SoS). Elles sont définies comme suit :

- Stratégie *FireDistinguishS* : Stratégie pour le scénario qui mène à la mission *FireDistinguish*, en séquençant les règles de réécritures de ce scénario via leurs labels dans le module *ArchSoSBehaviours*. Les règles sont combinées via l'opérateur de concaténation ";", pour indiquer qu'une règle doit être complètement exécutée avant de passer à l'exécution de la règle suivante dans la séquence.
- Stratégie *HurricaneEvacuationS* : Similairement, les règles du deuxième scénario sont combinées avec le même opérateurs dans cette stratégie pour achever la mission *HurricaneEvacuation*.
- Stratégie *CrisisResponseS* : Cette stratégie représente le comportement global de CRSoS, en combinant les deux stratégies précédentes avec l'opérateur d'union "|". Ainsi, nous définissons deux chemins d'exécution parallèles pour chaque scénario dans la même stratégie, sans que l'un affecte l'autre. Cependant, si les deux chemins sont exécutables en même temps, la stratégie *FireDistinguishS* aura une priorité sur la stratégie *ArchSoSBehaviours*, en termes de temps de réponse et d'ordre de réécritures dans le moteur de Maude.


```

smode ArchSoSStrategies is
protecting ArchSoSBehaviours .
including STRATEGY-MODEL-CHECKER .
including ArchoSoSAnalysis .
strat FireDistinguishS @ SoS .
strat HurricaneEvacuationS @ SoS .
strat CrisisResponseS @ SoS .

sd FireDistinguishS := ( LinkFireApperance ; AcquireRoleFireApperance ; LinkFireSignal
; AcquireRoleFireSignal ; LinkFire ; AcquireRoleFire ) .

sd HurricaneEvacuationS := ( LinkHurricaneApperance ; AcquireRoleHurricaneApperance
; LinkHurricaneSignal ; AcquireRoleHurricaneSignal ; LinkHurricane ; AcquireRoleHurricane) .

sd CrisisResponseS := ( FireDistinguishS | HurricaneEvacuationS ) .

op initial : -> SoS .
eq initial = opSoS< CrisisControl , null >.P[
  opSoS< Surveillance , null >.P[
    opsubSyS< CALert , communications >.L[ null ]|
    opsubSyS< CDetection , detection >.L[ null ] ]
  .L[ null ].E[ FireApperance , HurricaneApperance ]
| opSoS< FireFighting , null >.P[opsubSyS< EquipementsL , equipements >.L[ null ] |
  opsubSyS< FireS , Ffighting >.L[ null ] ]
  .L[ null ].E[ FireSignal , HurricaneSignal ]
| opSoS< Police , null >.P[opsubSyS< TrafficC , TManagement >.L[ null ] |
  opsubSyS< EvacuationCC , CManagement >.L[ null ] ].L[ null ].E[ null ] ]
  .L[ null ].E[ Fire , Hurricane ] .
endsm

```

FIGURE 7.9: Stratégies d'évolution de CRSoS

7.3 Évaluation

Nous avons défini jusque-là les modules Maude nécessaires d'une part à la spécification architecturale de CRSoS dans ArchSoS, et d'autre part à la description dynamique de son comportement via des règles de réécritures et des stratégies Maude. Afin d'évaluer le bon fonctionnement de CRSoS, nous procédons à sa simulation en exécutant les règles de réécritures définies avec le moteur de réécriture de Maude, ensuite nous effectuons une analyse formelle de ce comportement à travers l'outil model-checker de Maude.

7.3.1 Simulation et reconfiguration dynamique de CRSoS

Dans cette section, nous montrons la simulation du comportement de CRSoS de deux manières différentes afin d'essayer d'exécuter les deux scénarios de CRSoS sur le même état initial.

La première consiste à une réécriture Maude de l'état initial (de la figure 7.7), en laissant le moteur de réécritures Maude choisir quelle règle à appliquer. La commande qui permet de faire cette simulation est *rew* (pour rewrite). Les résultats de simulation sont illustrés dans les figures 7.10 et 7.11. Dans la figure 7.10, les règles du premier scénario sont définies avant les règles du deuxième scénario dans le module *ArchSoSBehaviours* (les règles de haut en bas dans la séquence). Par conséquent, le moteur de réécriture Maude a exécuté uniquement les règles du premier, et les règles du deuxième scénario ne sont pas exécutées.

Cependant, dans la figure 7.11, les règles du deuxième scénario sont placées avant celles du premier, évidemment l'exécution des règles du deuxième scénario est faite sans celle du premier

scénario.

```

Maude> rewrite in ArchSoSBehaviours : opSoS< CrisisControl,null >.P[(
  opSoS< Police,null >.P[opsubSyS< EvacuationCC,CManagement >.L[null] |
  opsubSyS< TrafficC,TManagement >.L[null]].L[null].E[null] | opSoS<
  FireFighting,null >.P[opsubSyS< EquipementsL,equipements >.L[null] |
  opsubSyS< FireS,Ffighting >.L[null]].L[null].E[FireSignal,HurricaneSignal])
  | opSoS< Surveillance,null >.P[opsubSyS< CDetection,detection >.L[null] |
  opsubSyS< CALert,communications >.L[null]].L[null].E[FireAppearance,
  HurricaneAppearance]].L[null].E[Fire,Hurricane] .
rewrites: 54 in 0ms cpu (0ms real) (~ rewrites/second)
result SoS: opSoS< CrisisControl,communications + detection + equipements +
  Ffighting >.P[opSoS< Police,null >.P[opsubSyS< EvacuationCC,CManagement
  >.L[null] | opsubSyS< TrafficC,TManagement >.L[null]].L[null].E[null] |
  opSoS< FireFighting,equipements + Ffighting >.P[opsubSyS< EquipementsL,
  equipements >.L[link< FireSignal : u ; EquipementsL --> FireS >] |
  opsubSyS< FireS,Ffighting >.L[link< FireSignal : u ; EquipementsL --> FireS
  >]].L[link< Fire : e ; Surveillance --> FireFighting >].E[FireSignal] |
  opSoS< Surveillance,communications + detection >.P[opsubSyS< CDetection,
  detection >.L[link< FireAppearance : e ; CALert --> CDetection >] |
  opsubSyS< CALert,communications >.L[link< FireAppearance : e ; CALert -->
  CDetection >]].L[link< Fire : e ; Surveillance --> FireFighting >].E[
  FireAppearance]].L[null].E[Fire]
Maude>
    
```

FIGURE 7.10: Simulation de CRSoS (mission *FireDistinguish* avant *HurricaneEvacuation* dans la séquence d'exécution)

```

Maude> rewrite in ArchSoSBehaviours : opSoS< CrisisControl,null >.P[(
  opSoS< Police,null >.P[opsubSyS< EvacuationCC,CManagement >.L[null] |
  opsubSyS< TrafficC,TManagement >.L[null]].L[null].E[null] | opSoS<
  FireFighting,null >.P[opsubSyS< EquipementsL,equipements >.L[null] |
  opsubSyS< FireS,Ffighting >.L[null]].L[null].E[FireSignal,HurricaneSignal])
  | opSoS< Surveillance,null >.P[opsubSyS< CDetection,detection >.L[null] |
  opsubSyS< CALert,communications >.L[null]].L[null].E[FireAppearance,
  HurricaneAppearance]].L[null].E[Fire,Hurricane] .
rewrites: 54 in 0ms cpu (0ms real) (~ rewrites/second)
result SoS: opSoS< CrisisControl,communications + detection + CManagement +
  TManagement >.P[opSoS< Police,CManagement + TManagement >.P[opsubSyS<
  EvacuationCC,CManagement >.L[link< HurricaneSignal : u ; EvacuationCC -->
  TrafficC >] | opsubSyS< TrafficC,TManagement >.L[link< HurricaneSignal : u
  ; EvacuationCC --> TrafficC >]].L[link< Hurricane : e ; Surveillance -->
  Police >].E[HurricaneSignal] | opSoS< FireFighting,null >.P[opsubSyS<
  EquipementsL,equipements >.L[null] | opsubSyS< FireS,Ffighting >.L[
  null]].L[null].E[HurricaneSignal] | opSoS< Surveillance,communications +
  detection >.P[opsubSyS< CDetection,detection >.L[null] | opsubSyS< CALert,
  communications >.L[link< HurricaneAppearance : e ; CALert --> CDetection
  >]].L[link< Hurricane : e ; Surveillance --> Police >].E[
  HurricaneAppearance]].L[null].E[Hurricane]
    
```

FIGURE 7.11: Simulation de CRSoS (mission *HurricaneEvacuation* avant *FireDistinguish* dans la séquence d'exécution)

Nous arrivons à la conclusion que lorsque nous avons deux scénarios d'exécution différents et concurrents, qui sont applicables sur le même état initial de CRSoS, le moteur de réécriture Maude est incapable d'exécuter les deux scénarios à la fois, et exécutera toujours le scénario dont les règles apparaissent en premier dans le module *ArchSoSBehaviours*. Cela peut être justifié par la capacité des règles de réécritures de chaque scénario de changer l'état initial après leur exécution.

Afin de remédier à ce problème, nous récurons à la deuxième manière qui consiste à une simulation basée sur les stratégies Maude. Plus précisément nous considérons les stratégies du

module *ArchSoSStrategies* lors de l'exécution des règles de réécritures. La stratégie *CrisisResponseS* de ce module regroupe les deux stratégies de chaque scénario, et représente ainsi le comportement global de CRSoS.

```

Maude> srewrite in ArchSoSStrategies : initial using CrisisResponseS .

Solution 1
rewrites: 163 in 0ms cpu (0ms real) (~ rewrites/second)
result SoS: opSoS< CrisisControl,communications + detection + equipements +
  Ffighting >.P[opSoS< Police,null >.P[opsubSys< EvacuationCC,CManagement
  >.L[null] | opsubSys< TrafficC,TManagement >.L[null]].L[null].E[null] |
  opSoS< FireFighting,equipements + Ffighting >.P[opsubSys< EquipementsL,
  equipements >.L[link< FireSignal : u ; EquipementsL --> FireS >] |
  opsubSys< FireS,Ffighting >.L[link< FireSignal : u ; EquipementsL --> FireS
  >]].L[link< (Fire).Event : e ; Surveillance --> FireFighting >].E[
  FireSignal] | opSoS< Surveillance,communications + detection >.P[opsubSys<
  CDetection,detection >.L[link< FireAppearance : e ; CALert --> CDetection
  >] | opsubSys< CALert,communications >.L[link< FireAppearance : e ; CALert
  --> CDetection >]].L[link< (Fire).Event : e ; Surveillance --> FireFighting
  >].E[FireAppearance]].L[null].E[Fire]

Solution 2
rewrites: 163 in 0ms cpu (0ms real) (~ rewrites/second)
result SoS: opSoS< CrisisControl,communications + detection + CManagement +
  TManagement >.P[opSoS< Police,CManagement + TManagement >.P[opsubSys<
  EvacuationCC,CManagement >.L[link< HurricaneSignal : u ; EvacuationCC -->
  TrafficC >] | opsubSys< TrafficC,TManagement >.L[link< HurricaneSignal : u
  ; EvacuationCC --> TrafficC >]].L[link< Hurricane : e ; Surveillance -->
  Police >].E[HurricaneSignal] | opSoS< FireFighting,null >.P[opsubSys<
  EquipementsL,equipements >.L[null] | opsubSys< FireS,Ffighting >.L[
  null]].L[null].E[HurricaneSignal] | opSoS< Surveillance,communications +
  detection >.P[opsubSys< CDetection,detection >.L[null] | opsubSys< CALert,
  communications >.L[link< HurricaneAppearance : e ; CALert --> CDetection
  >]].L[link< Hurricane : e ; Surveillance --> Police >].E[
  HurricaneAppearance]].L[null].E[Hurricane]

No more solutions.
rewrites: 163 in 0ms cpu (43ms real) (~ rewrites/second)

Maude>

```

FIGURE 7.12: Simulation de CRSoS avec les stratégies Maude

Nous simulons l'évolution de CRSoS à partir de son état initial (variable *initial* qui décrit cet état dans la figure 7.9) en appliquant cette stratégie. La commande qui permet cette exécution est la suivante :

srew initial using CrisisResponseS

Les résultats de cette simulation sont affichés dans la figure 7.12. Ils indiquent que la stratégie *FireDistinguishS* et la stratégie *HurricaneEvacuationS*, qui correspondent à chaque scénario, sont exécutées simultanément en obtenant deux solutions distinctes, où chaque solution représente un résultat de simulation indépendant.

La première solution revient au premier scénario qui traite la mission *FireDistinguish*, vu que sa stratégie était la première (de gauche à droite) dans la définition de la stratégie globale. Ainsi, nous pouvons déduire que :

- L'exécution de la stratégie globale de CRSoS offre la capacité de gérer des événements parallèles et simultanés. Ainsi, CRSoS peut répondre à plusieurs crises en même temps sans s'influencer les unes les autres.

- De plus, tout en définissant des stratégies de manière séquentielle et ordonnée, une priorité est donnée à certains scénarios par rapport à d'autres, cela est interprété comme un traitement par degré d'urgence de certaines crises comparées à d'autres, compte tenu du temps de réaction nécessaire à cette crise.

7.3.2 Vérification qualitative des comportements de CRSoS

Dans cette étape, nous utilisons l'outil model-checker de Maude, qui est basé sur la logique temporelle linéaire LTL, pour vérifier la satisfaction des propriétés proposées. Dans un premier temps, nous rappelons les deux propriétés nécessaires au bon fonctionnement des comportements d'un SoS dans ArchSoS, qui sont les propriétés de sûreté (*safety* dans ArchSoS) et de vivacité (*vivacity*). Afin de vérifier la satisfaction de ses propriétés, nous les appliquons sur l'état initial de CRSoS comme argument du model-checker de Maude, suivi des opérateurs `[] <>` et la propriété à vérifier, pour indiquer que cette propriété sera toujours satisfaite, et cela peut importe le chemin d'exécution pris.

La figure 7.13 montre le résultat de vérification quantitative des propriétés *safety* et *vivacity* d'ArchSoS, appliquée sur l'état initial de CRSoS dans le model-checker. *

Le résultat True (vrai) est retourné, indiquant que la propriété est assurée dans les deux scénarios de CRSoS, et que CRSoS finira par avoir un rôle actif, traitant une mission donnée (*vivacity*), et les contraintes comportementales sur les rôles et les liens possibles sont toujours respectées (*safety*).

```

Ready.
reduce in ArchoSoSAnalysis : modelCheck(initial, [[]<>
    safety) .
rewrites: 336 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> reduce in ArchoSoSAnalysis : modelCheck(initial, [[]<> vivacity) .
rewrites: 154 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

red modelCheck(initial, [[]<> vivacity).

```

FIGURE 7.13: Vérification des propriétés de *sûreté* et de *vivacité*

En plus, afin d'illustrer d'autres propriétés fonctionnelles de CRSoS, nous tenons à vérifier la consistance (propriété *consistency* dans ArchSoS) des missions de CRSoS par exemple. Il suffit de trouver des états futurs à partir de l'état initial de CRSoS, où les rôles qui permettent d'achever ses missions (*FireDistinguish* ou *HurricaneEvacuation*) sont présents en tant que rôles combinés dans le comportement de CRSoS.

Afin de vérifier cette propriété, il faut d'abord définir des prédicats appropriés. La propriété *consistency* est alors définie :

$$eq \text{ consistency} = [[](\sim \text{vivacity} \wedge \langle \rangle i\text{Mission}).$$

*i*mission est le prédicat d'état de la mission à vérifier. La figure 7.14 indique les deux prédicats pour chaque mission de CRSoS *FireDistinguishMission* et *HurricaneEvacuationMission*, pour vérifier la consistance des missions *FireDistinguish* et *HurricaneEvacuation* respectivement.

```

ceq opSoS< Sidi , Ri >.P[ Si | Sj ].L[Li].E[Ei] |= FireDistinguishMission = true
if (Ri == detection + communications + equipements + Ffighting) .

ceq opSoS< Sidi , Ri >.P[ Si | Sj ].L[Li].E[Ei] |= HurricaneEvacuationMission = true
if (Ri == detection + communications + CManagement + TManagement) .

eq consistency = []<>( vivacity /\ FireDistinguishMission ) .

```

FIGURE 7.14: Prédicat pour la propriété *consistency*

Nous illustrons dans la même figure, à titre d'exemple, l'adaptation de la propriété "*consistency*" afin de vérifier la mission *FireDinstiguish*, où *iMission* est remplacée par le prédicat *FireDistinguishMission*.

```

reduce in ArchoSoSAnalysis : modelCheck(initial, []<>
  FireDistinguishMission) .
rewrites: 134 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample({opSoS< CrisisControl,null >.P[opSoS<
  Police,null >.P[opsubSyS< EvacuationCC,CManagement >.L[null] | opsubSyS<
  TrafficC,TManagement >.L[null]].L[null].E[null] | opSoS< FireFighting,null
  >.P[opsubSyS< EquipementsL,equipements >.L[null] | opsubSyS< FireS,
  Ffighting >.L[null]].L[null].E[FireSignal,HurricaneSignal] | opSoS<
  Surveillance,null >.P[opsubSyS< CDetection,detection >.L[null] | opsubSyS<
  CAlert,communications >.L[null]].L[null].E[FireAppearance,
  HurricaneAppearance]].L[null].E[Fire,Hurricane],'LinkHurricaneAppearance} {
.....
.....

HurricaneAppearance]].L[null].E[Hurricane],'AcquireRoleHurricane), {opSoS<
CrisisControl,communications + detection + CManagement + TManagement >.P[
opSoS< Police,CManagement + TManagement >.P[opsubSyS< EvacuationCC,
CManagement >.L[link< HurricaneSignal : u ; EvacuationCC --> TrafficC >] |
opsubSyS< TrafficC,TManagement >.L[link< HurricaneSignal : u ; EvacuationCC
--> TrafficC >]].L[link< Hurricane : e ; Surveillance --> Police >].E[
HurricaneSignal] | opSoS< FireFighting,null >.P[opsubSyS< EquipementsL,
equipements >.L[null] | opsubSyS< FireS,Ffighting >.L[null]].L[null].E[
HurricaneSignal] | opSoS< Surveillance,communications + detection >.P[
opsubSyS< CDetection,detection >.L[null] | opsubSyS< CAlert,communications
>.L[link< HurricaneAppearance : e ; CAlert --> CDetection >]].L[link<
Hurricane : e ; Surveillance --> Police >].E[HurricaneAppearance]].L[
null].E[Hurricane],deadlock))

```

FIGURE 7.15: Vérification de la mission *FireDistinguish* dans Maude

Nous appliquons l'outil model-checker Maude, avec l'état initial de CRSoS, et la propriété "*consistency*" pour vérifier cette mission. Le résultat d'exécution est affiché dans la figure 7.15, en retournant un contre-exemple pour indiquer que la propriété "*consistency*" n'est pas satisfaite sur l'évolution de CRSoS. Ce contre-exemple représente l'exécution du deuxième scénario où la mission *FireDistinguish* n'est pas traitée.

Nous interprétons cela par l'absence d'un mécanisme de contrôle qui guide le moteur de réécriture de Maude pour exécuter le premier scénario précisément. Nous avons le même résultat si nous vérifions la consistance de la mission *HurricaneEvacuation*.

L'utilisation des stratégies de Maude permet de remédier à ce problème. Nous récurons à l'outil de vérification formelle applicable sur ces stratégies, appelé "STRATEGY-MODEL-CHECKER". La différence avec le model-checker de Maude, c'est que nous pourrions spécifier, au moment de l'exécution de l'outil Model-checker, la stratégie à appliquer dans la simulation dans laquelle se déroule la vérification. La commande qui permet cela est la suivante :

$$red\ modelCheck(initial, LTLoperators\ P, 'S)$$

Où *initial* est l'état initial de CRSoS, P est la propriété à vérifier, et S est la stratégie à appliquer pendant cette vérification. Les figures 7.16 et 7.17 montrent la vérification de la propriété "consistency" de chaque mission de CRSoS. Évidemment La vérification retourne True(Vrai) dans les deux cas de figure.

```
Maude> reduce in ArchSoSstrategies : modelCheck(initial, []<> consistency,
  'FireDistinguishS) .
rewrites: 113 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude>
```

FIGURE 7.16: Vérification de la consistance de la mission *FireDistinguish* avec les stratégies Maude

```
Maude> reduce in ArchSoSstrategies : modelCheck(initial, [
  ]<> consistency, 'HurricaneEvacuationS) .
rewrites: 168 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude>
```

FIGURE 7.17: Vérification de la consistance de la mission *HurricaneEvacuation* avec les stratégies Maude

7.4 Vers un framework pour l'implémentation d'Arch-SoS

ArchSoS combine les deux formalismes des BRS et Maude : (1) Les BRS sont adaptés pour modéliser graphiquement la structure d'ArchSoS, ainsi que les actions d'évolutions à l'aide de règles de réactions définies graphiquement. (2) Maude est utilisé comme un moyen d'exécution d'une sémantique dans ArchSoS, grâce à son moteur de réécriture qui permet de simuler des SoS, avant de les vérifier par la suite avec son outil de Model-checking LTL.

Afin d'illustrer notre approche appliquée sur cette étude de cas dans un aspect pratique, nous présentons dans ce qui suit, une proposition d'un Framework dédié à l'implémentation

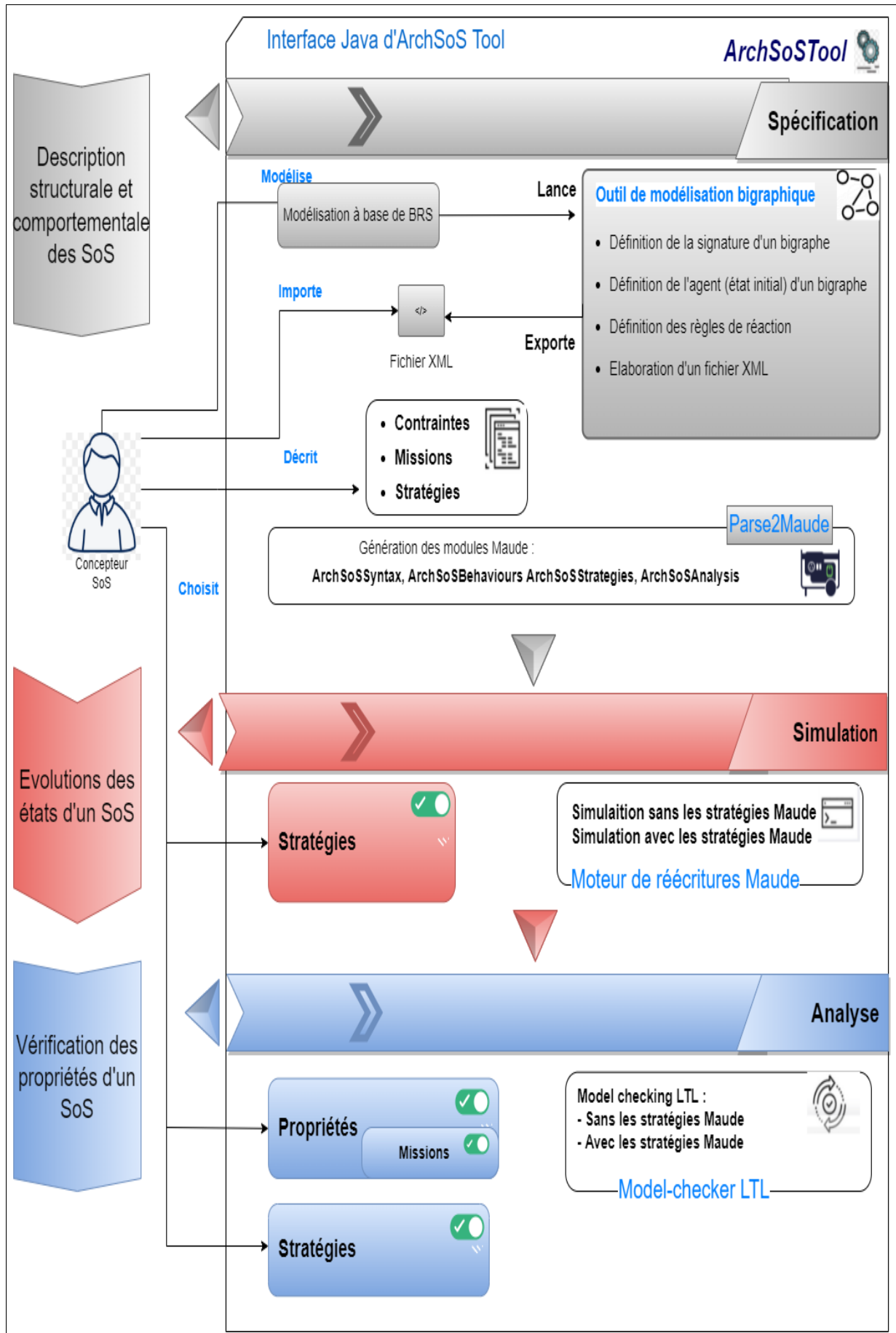


FIGURE 7.18: Vue globale d'ArchSoSTool

d'ArchSoS, que nous appelons ArchSoSTool. Il s'agit d'une plateforme permettant de spécifier, simuler et vérifier les SoS dans notre ADL.

Nous avons implémenté ArchSoS dans le langage Java pour des raisons d'efficacité et de simplicité, à travers Eclipse qui est un environnement de développement intégré (Integrated Development Environment ou IDE), compatible avec la presque totalité des langages de programmation actuels. Java est équipé d'une bibliothèque graphique libre "Swing", qui offre la possibilité de créer des interfaces graphiques identiques quel que soit le système d'exploitation sous-jacent, nous profitons de cela pour créer une interface graphique propre à ArchSoSTool.

La figure 7.18 donne une vue d'ensemble du processus de description architecturale d'un SoS dans ArchSoSTool par un concepteur, qui est divisée en trois phases essentielles :

7.4.1 Phase de spécification

Cette phase vise à offrir une description architecturale d'un SoS avec ArchSoSTool, aussi bien d'un point de vue structurel et comportementale. La figure 7.19 montre les fonctionnalités principales de cette phase.

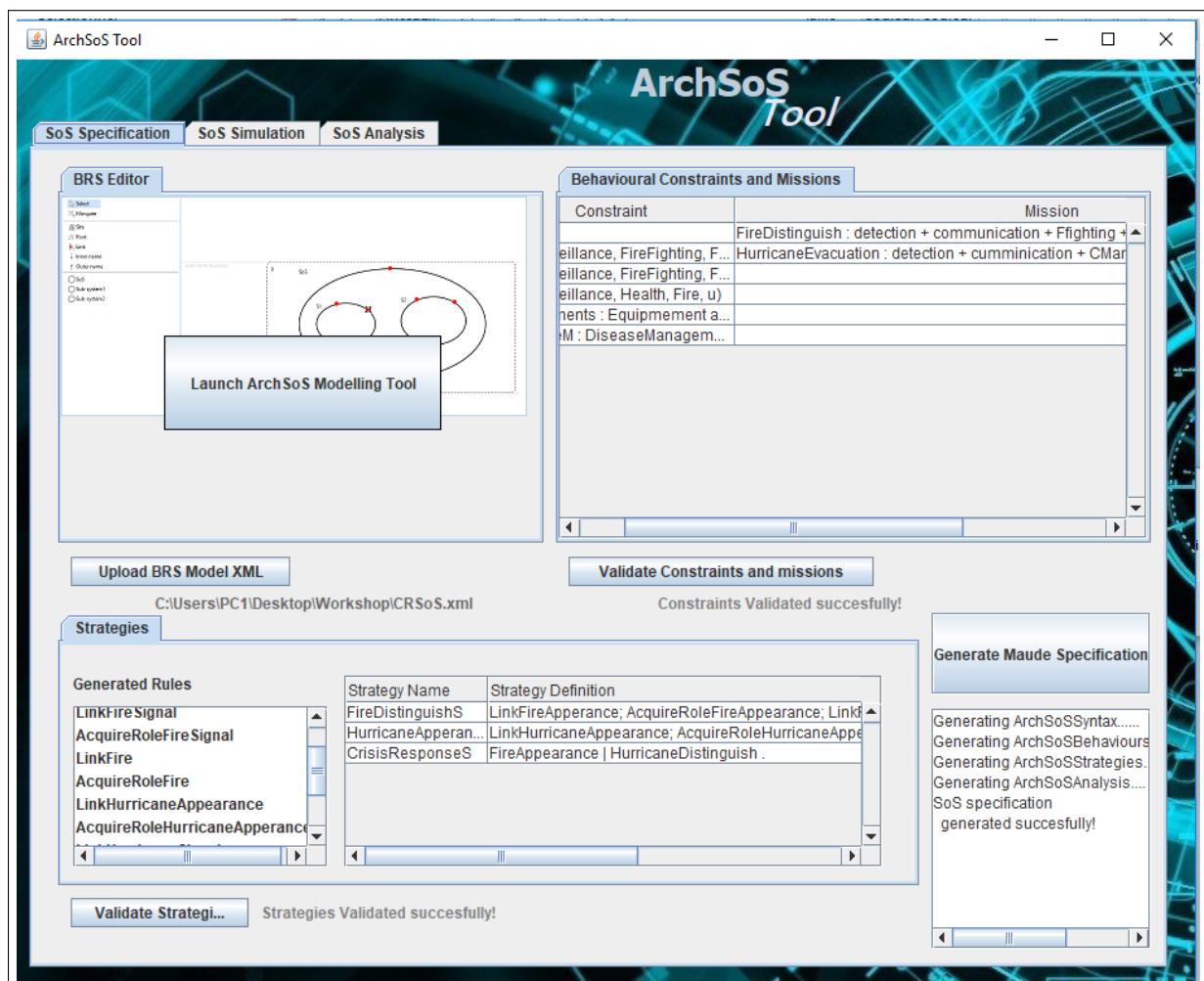


FIGURE 7.19: Spécification de CRSoS dans ArchSoSTool

Un concepteur voulant spécifier un SoS doit tout d'abord décrire son SoS, nous adaptons pour cela les BRS, qui sont le formalisme le plus simple pour ce genre de description, car sa forme graphique et visuelle est plus simple à comprendre et à manipuler. Cela est effectué grâce à un plugin offrant une extension de l'outil BigRed, dont le choix est justifié par sa capacité d'interfaçage élevée, et son aptitude à exporter les spécifications au format XML (eXtensible

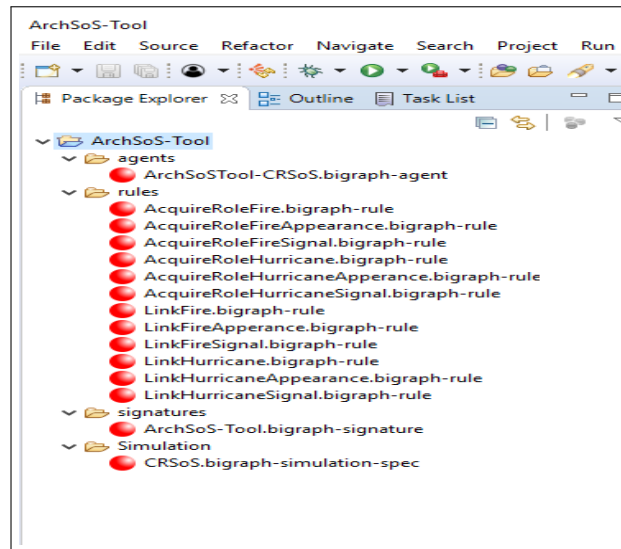


FIGURE 7.20: Exemple de modélisation dans ArchSoS-Tool

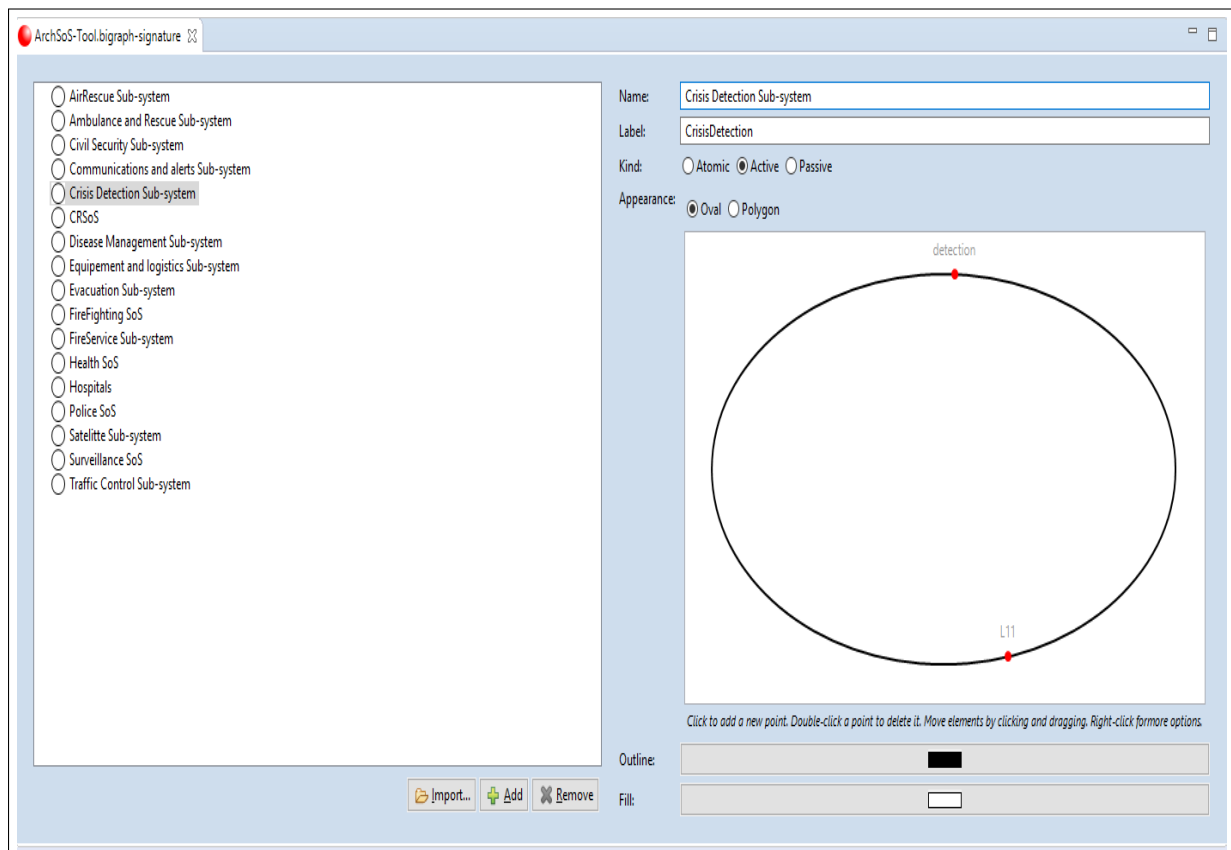


FIGURE 7.21: Création de noeuds de CRSoS dans ArchSoS-Tool

Markup Language), ce qui permet d'être utilisés aisément par n'importe quel langage et n'importe quelle plateforme, sans subir de dépendances.

Ce plugin de modélisation bigraphique est lancé depuis le bouton "Launch ArchSoS Modelling Tool". Par la suite, quatre types d'entités sont à définir par l'utilisateur. La figure 7.20 montre l'architecture de l'outil de modélisation d'ArchSoS-Tool pour modéliser le cas de CRSoS. En effet, la spécification doit passer par quatre (04) fonctionnalités :

- 1- "*bigraph-signature* ." Une fonctionnalité pour définir les noeuds d'un SoS et de ses constituants, et les ports associés à chaque noeud. La figure 7.21 montre la création de la

signature de CRSoS.

- 2- "*bigraph-agent* :" Cette fonctionnalité offre la possibilité de représenter l'état initial d'un SoS, modélisé par des éléments de la signature qui apparaissent dans une palette graphique. La figure 7.22 montre la modélisation de l'état initial de CRSoS avec ses constituants, et les événements du premier scénario qui permettent d'accomplir la mission *Fire-Distinguish*.

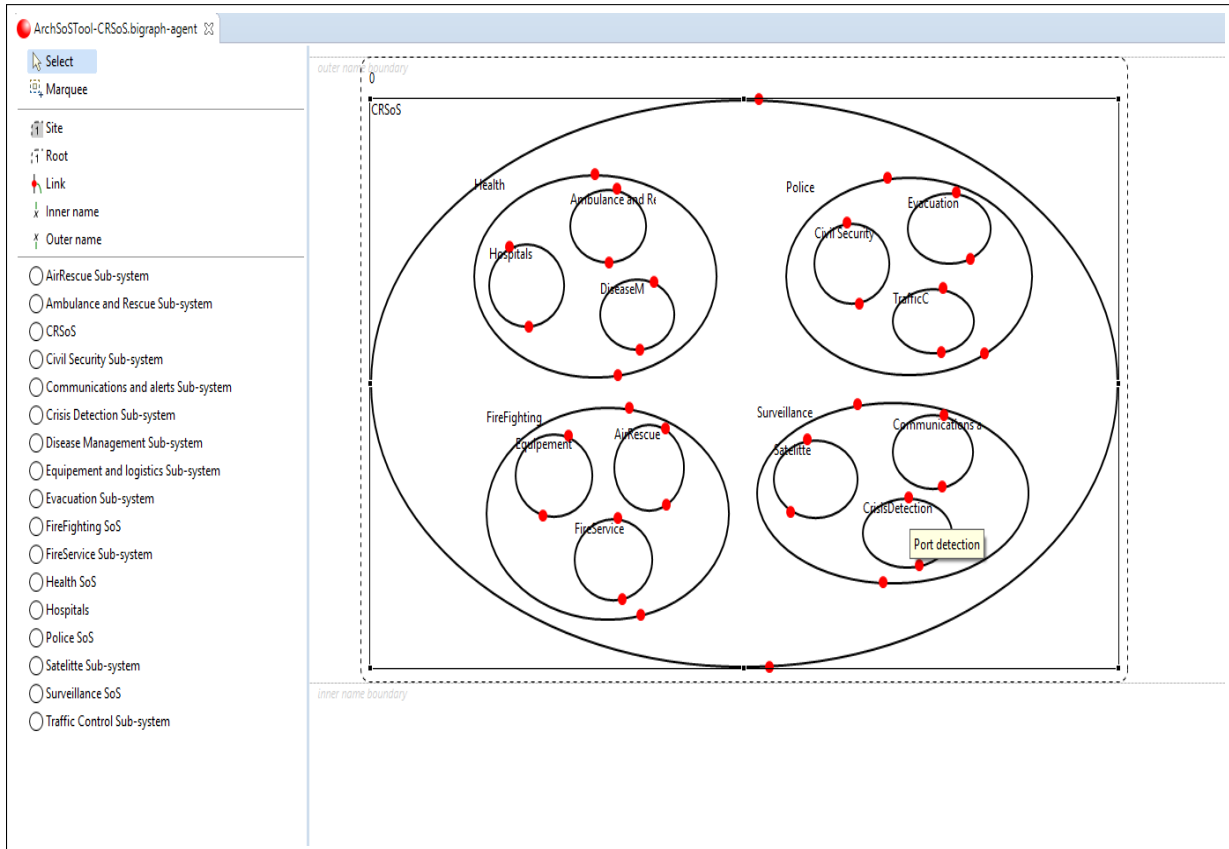


FIGURE 7.22: Modèle Bigraphique (état initial) de CRSoS avec ArchSoSTool

- 3- "*bigraph-rule* :" Elle permet de créer l'ensemble des règles de réactions qui permettent l'évolution dynamique d'un SoS. Chaque règle à deux parties : un redex et un reactum, spécifiant la transition d'un SoS à partir d'un état donné, vers un état futur. La figure 7.23 montre la définition de la règle "*LinkFireAppearance*" de CRSoS.
- 4- "*bigraph-simulation-spec* :" Enfin, cette fonctionnalité permet de préparer une simulation, en choisissant la signature d'un SoS, les règles à appliquer, son modèle d'état initial (*bigraph-agent*) qui seront exportés en tant que fichier XML.

Une fois le fichier XML de la spécification généré par l'outil de modélisation, le concepteur l'importe dans la partie spécification d'ArchSoSTool. Ensuite, il doit définir et valider les contraintes, les missions (les deux dans l'onglet *Behavioural Constraints and Missions*) et les stratégies (onglet *Strategies*) qui concernent son SoS, les règles sont importées automatiquement à partir du fichier XML pour faciliter la définition des stratégies.

Cette phase est concrétisée par la génération des quatre modules Maude indispensables pour la définition sémantique opérationnelle d'un SoS dans ArchSoS. La génération passe par le générateur *Parse2Maude*, permettant de générer ses modules et de les lancer dans le moteur de réécriture Maude intégré à ArchSoSTool.

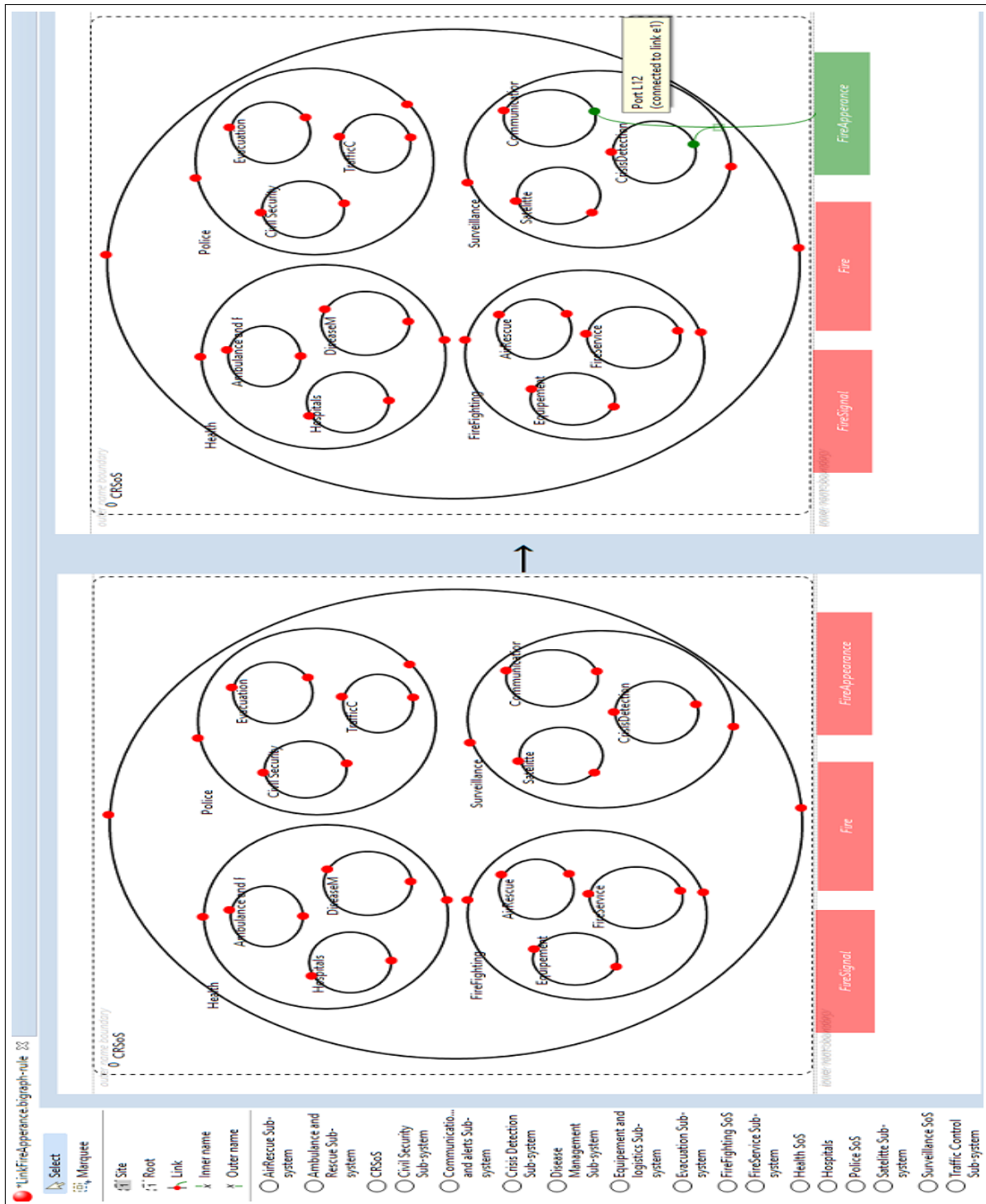


FIGURE 7.23: Saisie d'une règle de réaction "LinkFireAppearance" de CRSoS sous ArchSoSTool

7.4.2 Phase de simulation :

Cette phase suit la première phase, et permet d'appliquer une simulation d'un état initial modélisé par l'utilisateur précédemment dans l'outil de modélisation, via l'entité bigraph-agent. Elle permet d'avoir comme retour des états d'évolution d'un SoS avec une simulation spécifique. Les règles de réactions bigraphiques, qui sont traduites vers des règles de réécritures Maude sont exécutées dans cette phase, pour faire une simulation des comportements d'un SoS dans ArchSoS, en utilisant ArchSoSTool.

Le concepteur peut choisir quelle stratégie à appliquer (ou sans stratégies dans le cas échéant) pendant la simulation de son SoS. La figure 7.24 montre la simulation de CRSoS avec l'application de la stratégie *CrisisResponseS*, où deux solutions distinctes sont affichés dans le moteur de réécriture Maude.

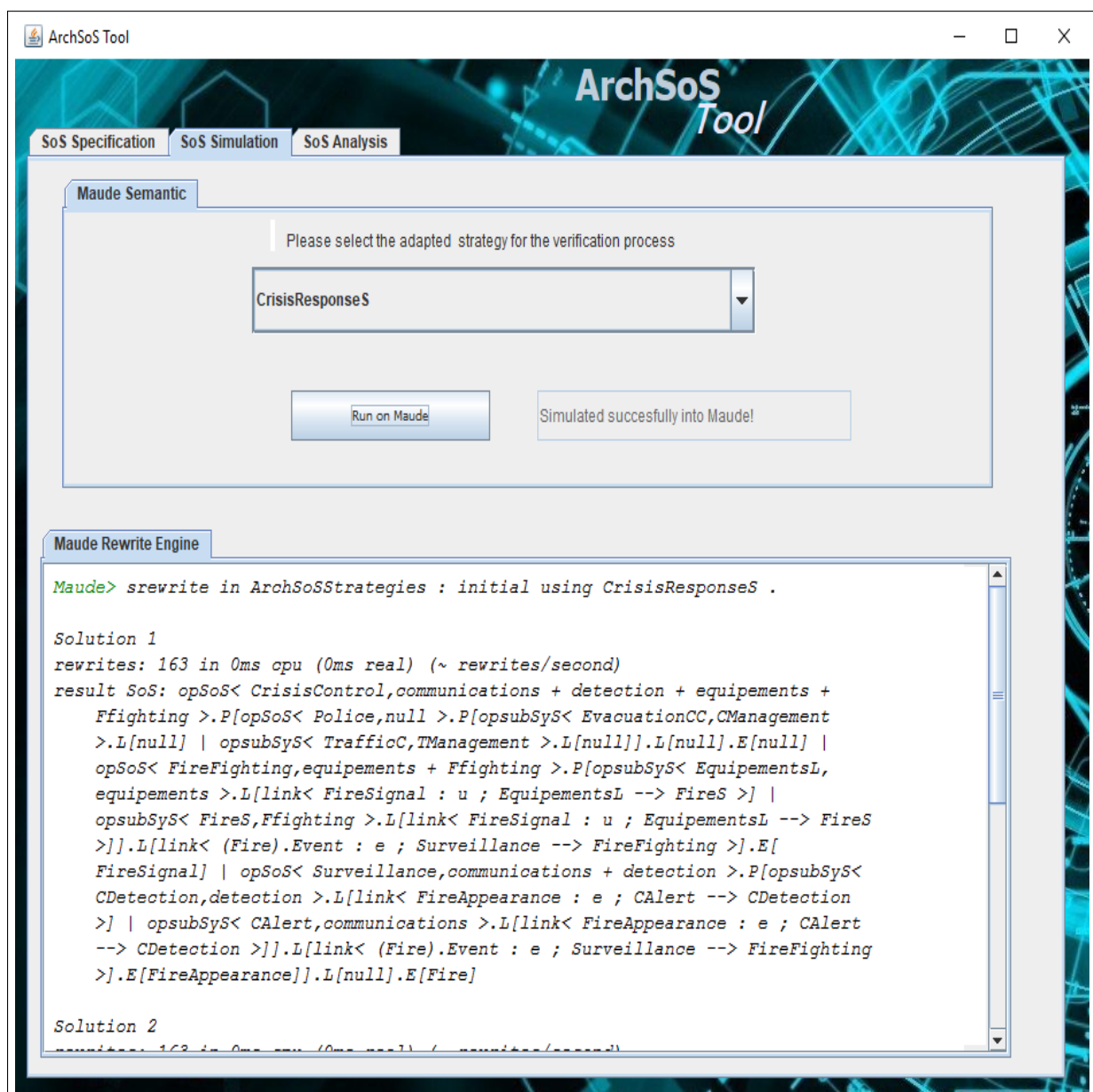


FIGURE 7.24: Simulation de CRSoS avec la stratégie *CrisisDetectionS* dans ArchSoSTool

7.4.3 Phase d'analyse

La dernière phase consiste à l'analyse formelle des comportements d'un SoS avec ArchSoS-Tool. Elle permet de vérifier, à l'aide de l'outil model-checker LTL de Maude, les propriétés de sûreté, de vivacité d'un SoS, ainsi que la consistance de chacune de ses missions définies. Pour cela, le concepteur doit choisir quelle propriété à vérifier. Dans le cas des missions (qui sont importés automatiquement depuis la spécification), il doit choisir aussi quelle mission dont il veut vérifier la consistance.

En plus, il peut sélectionner la stratégie à adapter pendant le processus de vérification, s'il veut les vérifier dans des chemins d'exécution particuliers. La figure 7.25 montre le résultat de la vérification formelle qualitative de la propriété de vivacité pour CRSoS.

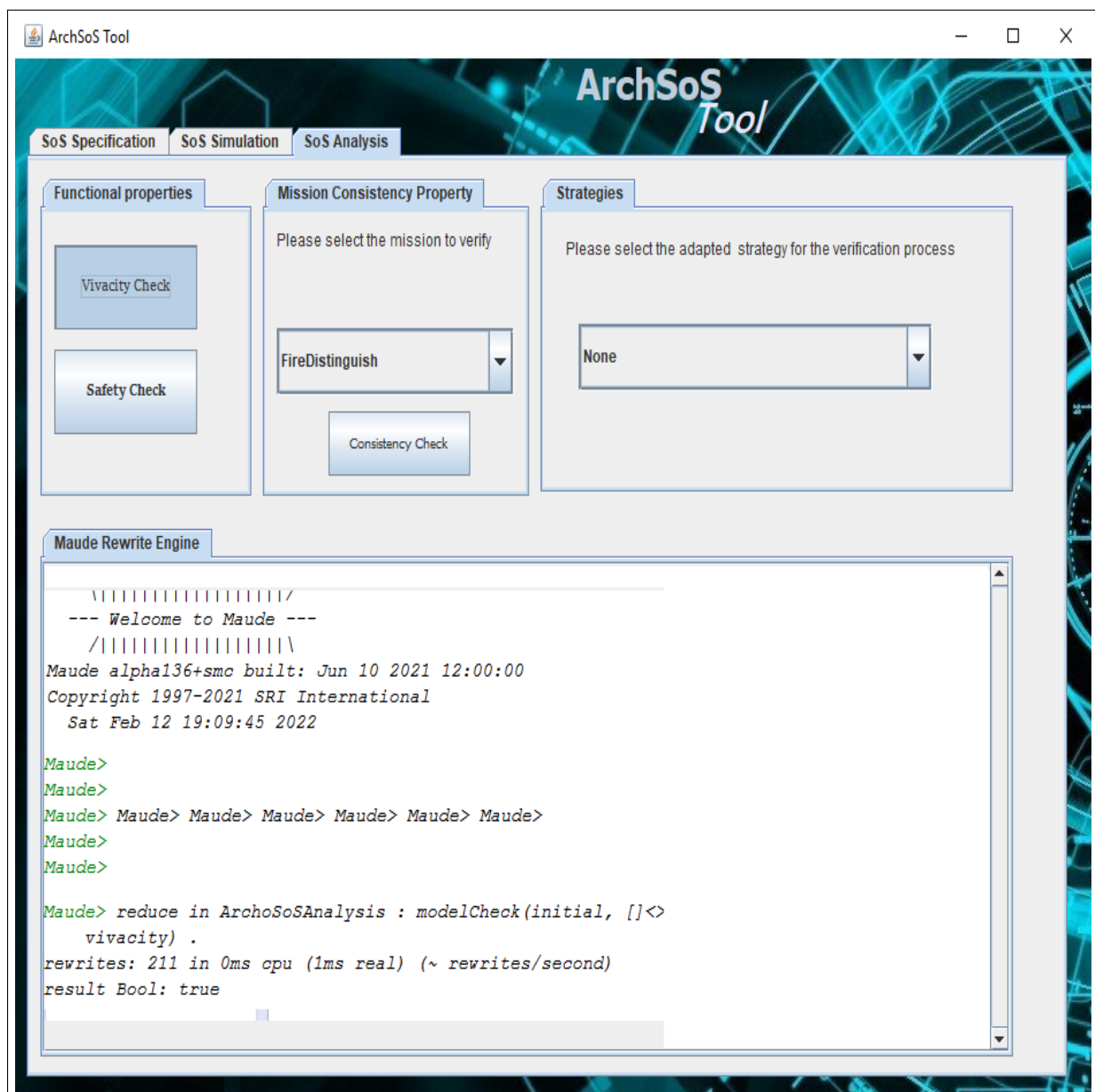


FIGURE 7.25: Vérification de la propriété de vivacité de CRSoS dans ArchSoS-Tool

7.5 Conclusion

Dans ce chapitre, nous avons présenté une étude de cas concrète d'un SoS réel, qui s'agit de CRSoS : un SoS de gestion de crises, où les crises sont des événements qui affectent le SoS, et la réponse est la mission à accomplir pour réagir à cette crise.

Dans un premier temps, nous avons appliqué notre approche basée ArchSoS afin d'attribuer une syntaxe concrète à CRSoS, contenant tous les éléments contenue de cet SoS (constituants, rôles, événements, missions, etc.). Deux scénarios d'évolution de CRSoS sont considérés dans cette étude de cas, qui permettent d'accomplir deux missions différentes pour cet SoS.

CRSoS est ainsi modélisé selon deux vues, une vue graphique et une vue algébrique en s'inspirant du formalisme des BRS.

Nous avons par la suite associé une sémantique opérationnelle basée sur le langage Maude à CRSoS. Cette sémantique offre la possibilité de simuler l'évolution comportementale de CRSoS en utilisant un ensemble de règles de réécritures, conditionnées par un ensemble de prédicats de contrôle de CRSoS. L'exécution du modèle a été enrichie par la prise en charge des stratégies Maude, afin de guider cette simulation, où chaque stratégie concerne un chemin de simulation particulier menant vers une mission précise.

Les stratégies sont combinées pour définir le comportement global de CRSoS, offrant ainsi la possibilité d'avoir plusieurs missions en parallèle à partir de la même configuration de CRSoS.

De plus, les comportements de CRSoS sont analysés et vérifiés, dans un aspect qualitatif, par l'outil model-checker LTL de Maude, afin de garantir leurs cohérences, ainsi que leurs bons fonctionnements.

Finalement, nous avons proposé une ébauche d'un Framework pour implémenter n'importe quel exemple de SoS dans ArchSoS.

Conclusion générale

“How you start is important, but it is how you finish that counts. In the race for success, speed is less important than stamina. The sticker outlasts the sprinter.”

B. C. Forbes

L’essor de la numérisation que nous connaissons actuellement dans tous les domaines de notre vie courante donne naturellement lieu à diverses hypothèses et théories, à des espoirs et même à des frustrations. Les réussites et les échecs des transformations induites par les TIC (Technologies de l’Information et de la Communication) ont montré que les technologies elles-mêmes ne sont ni positives, ni négatives, ni nécessairement neutres.

Les TIC évoluent à une vitesse vertigineuse, ils sont au centre des nouvelles démarches d’ingénierie des systèmes (modélisation 3D, virtualisation, simulation, prototypage numérique, etc.) pour la conception et le développement de systèmes intelligents présents dans les entreprises de haute technologie dans de nombreux domaines tels que : les télécommunications, l’automobile, l’aéronautique, le militaire, le médical, etc.

Les objectifs majeurs d’une démarche d’ingénierie des systèmes d’actualité sont les suivants :

- A. Concevoir et réaliser des systèmes utilisant des circuits électroniques, de l’informatique embarquée, des technologies réseaux et de transmission sans fil, des composants multimédias et du traitement de signal,
- B. Concevoir l’architecture de systèmes exploitant les technologies numériques émergentes,
- C. Gérer des équipes et des projets en prenant en compte des contraintes économiques, sociales et écologiques.

Les travaux de cette thèse se positionnent dans ce contexte en proposant une démarche de modélisation des systèmes de systèmes (SoS) qui cible particulièrement l’objectif B suscitée.

Les Systèmes de systèmes (SoS), se caractérisent par l’autonomie opérationnelle et managérielle de leurs constituants hiérarchiques et hétérogènes, ayant des évolutions dynamiques et imprévisibles pour accomplir les missions d’un SoS. Ce type de système a pris une popularité et un intérêt d’ampleur aussi bien dans les domaines académiques qu’industriels. Ils ont la capacité d’aboutir à des fonctionnalités complexes qu’aucun des constituants du SoS peut accomplir

seul. Ce côté évolutif des constituants à deux échelles (singulier et au sein du SoS) ouvre la voie à plusieurs défis qui font face à cette croissance exponentielle en terme de complexité qui réside notamment dans la description de la hiérarchie d'un SoS, les interactions entre ses constituants, leurs évolutions, ainsi que les missions qu'un SoS est sensé accomplir, tout en s'assurant que ses défis sont décrits avec une cohérence qui peut être analysée et vérifiée, afin d'assurer son bon fonctionnement.

Ses défis contraignants ont rendu les SoS un type de systèmes très attrayant par rapport aux chercheurs du génie logiciel. Les travaux existants dans la littérature sont soit trop conceptuels et théoriques, sans outils d'implémentation ou de simulation, soit capables de traiter uniquement l'aspect structurel des SoS sans prendre en charge leur dynamique et d'autant plus leurs missions émergentes. L'objectif principal de cette thèse était de réaliser un compromis entre ces deux voies de recherche qui ont existé en contribuant à la réduction de la complexité des SoS, tout en leur offrant un niveau de description assez abstrait qui permet de répondre aux défis évoqués.

Notre choix s'est porté sur la description architecturale d'un SoS, nous avons proposé ArchSoS : un langage de description architecturale (ADL) dédié aux SoS. Nous avons adopté les méthodes formelles, pour définir la syntaxe et la sémantique opérationnelle de ce langage afin de pouvoir vérifier formellement l'exécution et l'analyse des SoS ainsi décrits.

Nous recensons dans ce qui suit nos principales contributions et quelques perspectives futures pouvant constituer les continuations envisageables pour ce travail.

- **Classification des approches de modélisation des SoS :** La réalisation d'un état de l'art concernant la conception des SoS a été entreprise pour motiver le besoin d'adopter une nouvelle approche intégrée à base de description d'architecture et méthode formelle.
- **Définition d'ArchSoS :** Nous avons donné une importance à la définition aussi bien de la syntaxe et de la sémantique de ArchSoS.
 - **Dans sa partie syntaxique,** nous avons adopté une architecture de référence dédiée à la description des SoS dans ArchSoS, tirée de la norme ISO/IEC/IEEE 42010, afin de décrire une syntaxe concrète pour ce langage. Elle s'est basée sur deux vues : une vue textuelle contenant tous les éléments d'un SoS comme les constituants, leurs types, les fonctionnalités (rôles), etc., et une vue graphique qui s'inspire des modèles de l'approche MDA, utilisant un méta-modèle conceptuel de l'architecture d'un SoS décrit par ArchSoS. Ce méta modèle a servi de base pour attribuer une syntaxe plus formelle à ArchSoS, il s'agit d'une syntaxe abstraite basée sur les systèmes réactifs bigraphiques. Les éléments structurels d'ArchSoS tels que : les SoS, leurs constituants, leurs fonctionnalités, et les événements pouvant les affecter pour effectuer des missions particulières, ainsi que les dépendances et les relations interactionnelles entre éléments, sont décrits par un bigraphe bien construit respectant un ensemble de règles de formation que nous avons élaboré. Ce modèle comprend une vue graphique visuelle, et une vue algébrique textuelle. De plus, nous avons défini l'aspect évolutif dynamique d'un SoS grâce à un ensemble d'actions qui sont décrites par des règles de réaction et un ensemble de prédicats de contrôles permettant de contraindre le déroulement de ces actions pour aboutir à un fonctionnement désirable.
 - **La partie sémantique** d'ArchSoS a été définie selon trois étapes essentielles :
 - a) Nous avons tout d'abord défini une sémantique opérationnelle au langage ArchSoS en transformant les spécifications bigraphiques des SoS en des modules

Maude. Les actions d'évolutions dans ArchSoS sont représentées par des règles de réécritures conditionnelles, les prédicats de contrôle sont définis en tant qu'un ensemble d'équations.

- b) D'autre part, la reconfiguration dynamique des comportements des SoS dans ArchSoS a été aussi considérée dans un module Maude afin de pouvoir traiter plusieurs missions, d'une façon séquentielle ou même en parallèle en appliquant des stratégies du langage de stratégies Maude.
 - c) Dans une dernière étape, nous avons procédé à une analyse formelle des comportements des SoS dans ArchSoS en utilisant le model-checker de Maude.
- **Etude de cas** : Nous avons illustré notre approche par une étude de cas reposant sur un SoS qui gère les réponses aux crises (CRSoS). L'outil d'analyse et de simulation de Strategy-Maude nous a permis de voir les différentes étapes de simulation du comportement de deux scénarios d'exécution où la gestion d'une crise est prise en charge de manière différente.

Les travaux que nous avons proposés peuvent être étendus et poursuivis. Nous identifions dans ce qui suit des perspectives à court, moyen et long termes, que nous classifions en points de vue théoriques, et points de vue pratiques :

- **D'un point de vue théorique** : Nous projetons raffiner la définition d'ArchSoS, afin de prendre en considération d'autres aspects pertinents des SoS. L'intégration des spécifications des BRS dans Maude, est faite selon une correspondance entre des éléments des BRS et les théories équationnelles et de réécritures Maude. Une piste de recherche serait de trouver une correspondance fondée et judicieuse entre ses deux formalismes complémentaires. Nous envisageons l'intégration d'autres propriétés pour la vérification qualitative des SoS, voir même quantitatif, dont les critères peuvent être définis par l'utilisateur.
- **D'un point de vue pratique** : Il serait intéressant d'optimiser l'automatisation des spécifications des SoS dans l'outil défini, avec l'intégration d'une interface de modélisation bigraphique indépendante et intuitive. L'aspect temps peut être considéré avec l'extension RT-Maude et traiter ainsi l'évolution temporelle des SoS dans ArchSoS.

Bibliographie

- [Akhtar and Khan, 2019] Akhtar, N. and Khan, S. (2019). Formal architecture and verification of a smart flood monitoring system-of-systems. *Int. Arab J. Inf. Technol.*, 16(2) :211–216.
- [Antul et al., 2018] Antul, L., Ricks, S., Cho, L. M. K., Cotter, M., Jacobs, R. B., Markina-Khusid, A., Kamenetsky, J., Dahmann, J., and Tran, H. T. (2018). Toward scaling model-based engineering for systems of systems. In *2018 IEEE Aerospace Conference*, pages 1–9. IEEE.
- [Axelsson, 2020] Axelsson, J. (2020). Achieving system-of-systems interoperability levels using linked data and ontologies. In *INCOSE International Symposium*, volume 30, pages 651–665. Wiley Online Library.
- [Axelsson et al., 2019] Axelsson, J., Fröberg, J., and Eriksson, P. (2019). Architecting systems-of-systems and their constituents : A case study applying industry 4.0 in the construction domain. *Systems Engineering*, 22(6) :455–470.
- [Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press.
- [Basin et al., 2011] Basin, D., Clavel, M., and Egea, M. (2011). A decade of model-driven security. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, pages 1–10.
- [Benammar et al., 2008] Benammar, M., Belala, F., and Latreche, F. (2008). Aa dl behavioral annex based on generalized rewriting logic. In *2008 Second International Conference on Research Challenges in Information Science*, pages 1–8. IEEE.
- [Benzadri et al., 2017] Benzadri, Z., Bouanaka, C., and Belala, F. (2017). Big-caf : a bigraphical-generic cloud architecture framework. *International Journal of Grid and Utility Computing*, 8(3) :222–240.
- [Bert and Cave, 2000] Bert, D. and Cave, F. (2000). Construction of finite labelled transition systems from b abstract systems. In *International Conference on Integrated Formal Methods*, pages 235–254. Springer.
- [Blanc and Salvatori, 2011] Blanc, X. and Salvatori, O. (2011). *MDA en action : Ingénierie logicielle guidée par les modèles*. Editions Eyrolles.
- [Boardman and Sauser, 2006] Boardman, J. and Sauser, B. (2006). System of systems-the meaning of of. In *2006 IEEE/SMC International Conference on System of Systems Engineering*, pages 6–pp. IEEE.
- [Bouhroum et al., 2019] Bouhroum, R., Boulkamh, A., Asia, L., Lebarillier, S., Ter Halle, A., Syakti, A., Doumenq, P., Malleret, L., and Wong-Wah-chung, P. (2019). Concentrations and fingerprints of pahs and pcbs adsorbed onto marine plastic debris from the indonesian cilacap coast and thenorth atlantic gyre. *Regional Studies in Marine Science*, 29 :100611.

- [Brook, 2016] Brook, P. (2016). On the nature of systems of systems. In *INCOSE International Symposium*, volume 26, pages 1477–1493. Wiley Online Library.
- [Brookes et al., 1984] Brookes, S. D., Hoare, C. A., and Roscoe, A. W. (1984). A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3) :560–599.
- [Bryans et al., 2014] Bryans, J., Fitzgerald, J., Payne, R., Miyazawa, A., and Kristensen, K. (2014). Sysml contracts for systems of systems. In *2014 9th International Conference on System of Systems Engineering (SOSE)*, pages 73–78. IEEE.
- [Bures et al., 2006] Bures, T., Hnetynka, P., and Plasil, F. (2006). Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pages 40–48. IEEE.
- [Caldiera and Rombach, 1994] Caldiera, V. R. B. G. and Rombach, H. D. (1994). The goal question metric approach. *Encyclopedia of software engineering*, pages 528–532.
- [Chaabane et al., 2019] Chaabane, M., Rodriguez, I. B., Colomo-Palacios, R., Gaaloul, W., and Jmaiel, M. (2019). A modeling approach for systems-of-systems by adapting iso/iec/ieee 42010 standard evaluated by goal-question-metric. *Science of Computer Programming*, 184 :102305.
- [Chechik and Gannon, 2001] Chechik, M. and Gannon, J. (2001). Automatic analysis of consistency between requirements and designs. *IEEE transactions on Software Engineering*, 27(7) :651–672.
- [Chkouri and Bozga, 2009] Chkouri, M. Y. and Bozga, M. (2009). Prototyping of distributed embedded systems using aadl. *ACESMB 2009*, page 65.
- [Clave et al., 2000] Clave, M., Durán, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., and Quesada, J. F. (2000). Towards maude 2.0. *Electronic Notes in Theoretical Computer Science*, 36 :294–315.
- [Clavel et al., 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2007). *All About Maude-A High-Performance Logical Framework : How to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350. Springer.
- [Comfort, 2007] Comfort, L. K. (2007). Crisis management in hindsight : Cognition, communication, coordination, and control. *Public administration review*, 67 :189–197.
- [Cuesta et al., 2002] Cuesta, C. E., Barrio-Solórzano, M., Beato, E., et al. (2002). Coordination in a reflective architecture description language. In *International Conference on Coordination Languages and Models*, pages 141–148. Springer.
- [Dahmann et al., 2017] Dahmann, J., Markina-Khusid, A., Doren, A., Wheeler, T., Cotter, M., and Kelley, M. (2017). Sysml executable systems of system architecture definition : A working example. In *Systems Conference (SysCon), 2017 Annual IEEE International*, pages 1–6. IEEE.
- [Dahmann and Baldwin, 2008] Dahmann, J. S. and Baldwin, K. J. (2008). Understanding the current state of us defense systems of systems and the implications for systems engineering. In *2008 2nd Annual IEEE Systems Conference*, pages 1–7. IEEE.
- [Damgaard and Birkedal, 2005] Damgaard, T. C. and Birkedal, L. (2005). Axiomatizing binding bigraphs (revised). Technical report, Citeseer.
- [DAU, 2010] DAU (2010). Defense acquisition guidebook (dag).
- [Derhamy et al., 2019] Derhamy, H., Eliasson, J., and Delsing, J. (2019). System of system composition based on decentralized service-oriented architecture. *IEEE Systems Journal*, 13(4) :3675–3686.

- [Dori, 2011] Dori, D. (2011). Object-process methodology. In *Encyclopedia of Knowledge Management, Second Edition*, pages 1208–1220. IGI Global.
- [Eker et al., 2007] Eker, S., Martí-Oliet, N., Meseguer, J., and Verdejo, A. (2007). Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science*, 174(11) :3–25.
- [Elsborg, 2009] Elsborg, E. (2009). Bigraphs : Modelling, simulation, and type systems—on bigraphs for ubiquitous computing and on bigraphical type systems.
- [Faithfull et al., 2013] Faithfull, A. J., Perrone, G., and Hildebrandt, T. T. (2013). Big red : A development environment for bigraphs. *Electronic Communications of the EASST*, 61.
- [Feiler et al., 2006] Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The architecture analysis & design language (aadl) : An introduction. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.
- [FERC, 2014] FERC, . N. (2014). *FERC/NERC Staff Report on the September 8, 2011 Blackout*.
- [Franca et al., 2007] Franca, R. B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., and Thomas, D. (2007). The aadl behaviour annex—experiments and roadmap. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 377–382. IEEE.
- [Gassara et al., 2017] Gassara, A., Rodriguez, I. B., Jmaiel, M., and Drira, K. (2017). A bigraphical multi-scale modeling methodology for system of systems. *Computers & Electrical Engineering*, 58 :113–125.
- [Gassara et al., 2019] Gassara, A., Rodriguez, I. B., Jmaiel, M., and Drira, K. (2019). Executing bigraphical reactive systems. *Discrete Applied Mathematics*, 253 :73–92.
- [Grohmann and Miculan, 2007] Grohmann, D. and Miculan, M. (2007). Directed bigraphs. *Electronic Notes in Theoretical Computer Science*, 173 :121–137.
- [Hachem et al., 2020] Hachem, J. E., Chiprianov, V., Babar, M. A., Khalil, T. A., and Aniorte, P. (2020). Modeling, analyzing and predicting security cascading attacks in smart buildings systems-of-systems. *Journal of Systems and Software*, 162 :110484.
- [Hatcliff et al., 2021] Hatcliff, J., Belt, J., Carpenter, T., et al. (2021). Hamr : An aadl multi-platform code generation toolset. In *International Symposium on Leveraging Applications of Formal Methods*, pages 274–295. Springer.
- [Hause, 2014] Hause, M. C. (2014). Sos for sos : A new paradigm for system of systems modeling. In *Aerospace Conference, 2014 IEEE*, pages 1–12. IEEE.
- [Højsgaard and Glenstrup, 2011] Højsgaard, E. and Glenstrup, A. J. (2011). *The BPL Tool—a Tool for Experimenting with Bigraphical Reactive Systems*. IT University of Copenhagen.
- [Hu et al., 2014] Hu, J., Huang, L., Chang, X., and Cao, B. (2014). A model driven service engineering approach to system of systems. In *Systems Conference (SysCon), 2014 8th Annual IEEE*, pages 136–145. IEEE.
- [Hutton et al., 2011] Hutton, J. P., Mackin, M., Traci, M., Carrigg, W., Derricotte, T., Echard, J., Fish, L., Gomez, C., Hassinger, K., Lee, J., et al. (2011). Coast guard : Action needed as approved deepwater program remains unachievable. Technical report, GOVERNMENT ACCOUNTABILITY OFFICE WASHINGTON DC.
- [Huynh and Osmundson, 2006] Huynh, T. V. and Osmundson, J. S. (2006). A systems engineering methodology for analyzing systems of systems using the systems modeling language (SysML).
- [IEEE, 1990] IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

- [INCOSE, 2018] INCOSE (2018). *Systems of Systems Primer*.
- [ISO, 2015] ISO (2015). Iso/iec/ieee international standard - systems and software engineering – system life cycle processes. *ISO/IEC/IEEE 15288 First edition 2015-05-15*, pages 1–118.
- [ISO, 2019] ISO (2019). Iso/iec/ieee international standard – systems and software engineering – system of systems (sos) considerations in life cycle stages of a system. *ISO/IEC/IEEE 21839 :2019(E)*, pages 1–40.
- [ISO/IEC/IEEE, 2011] ISO/IEC/IEEE (2011). Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010 : 2011 (E)(Revision of ISO/IEC 42010 : 2007 and IEEE Std 1471-2000)*, pages 1—46.
- [Issarny et al., 2011] Issarny, V., Bennaceur, A., and Bromberg, Y.-D. (2011). Middleware-layer connector synthesis : Beyond state of the art in middleware interoperability. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 217–255. Springer.
- [Jamshidi and Sage, 2009] Jamshidi, M. and Sage, A. P. (2009). *System of systems engineering : innovations for the 21st century*, volume 58. John Wiley & Sons Incorporated.
- [Jensen, 1987] Jensen, K. (1987). Coloured petri nets. In *Petri nets : central models and their properties*, pages 248–299. Springer.
- [Keating et al., 2003] Keating, C., Rogers, R., Unal, R., Dryer, D., Sousa-Poza, A., Safford, R., Peterson, W., and Rabadi, G. (2003). System of systems engineering. *Engineering Management Journal*, 15(3) :36–45.
- [Khebbeb et al., 2019] Khebbeb, K., Hameurlain, N., Belala, F., and Sahli, H. (2019). Formal modelling and verifying elasticity strategies in cloud systems. *IET Software*, 13(1) :25–35.
- [Kinder et al., 2015] Kinder, A., Henshaw, M., and Siemieniuch, C. (2015). A model based approach to system of systems risk management. In *2015 10th System of Systems Engineering Conference (SoSE)*, pages 122–127. IEEE.
- [Krivine et al., 2008] Krivine, J., Milner, R., and Troina, A. (2008). Stochastic bigraphs. *Electronic Notes in Theoretical Computer Science*, 218 :73–96.
- [Lane and Bohn, 2013] Lane, J. A. and Bohn, T. (2013). Using sysml modeling to understand and evolve systems of systems. *Systems Engineering*, 16(1) :87–98.
- [López et al., 1995] López, O. P., Salavert, I. R., and Cerdá, J. H. C. (1995). Oasis v2 : A class definition language. In *International Conference on Database and Expert Systems Applications*, pages 79–90. Springer.
- [Magee and Maibaum, 2006] Magee, J. and Maibaum, T. (2006). Towards specification, modelling and analysis of fault tolerance in self managed systems. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 30–36.
- [Maier, 1998] Maier, M. W. (1998). Architecting principles for systems-of-systems. *Systems Engineering : The Journal of the International Council on Systems Engineering*, 1(4) :267–284.
- [Malavolta et al., 2012] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2012). What industry needs from architectural languages : A survey. *IEEE Transactions on Software Engineering*, 39(6) :869–891.
- [Martí-Oliet et al., 2009] Martí-Oliet, N., Meseguer, J., and Verdejo, A. (2009). A rewriting semantics for maude strategies. *Electronic Notes in Theoretical Computer Science*, 238(3) :227–247.
- [McCombs, 2003] McCombs, T. (2003). Maude 2.0 primer. *Department of Computer Science, University of Illinois and Urbana-Champaign, Urbana-Champaign, Ill., USA*.

- [McCracken and Reilly, 2003] McCracken, D. D. and Reilly, E. D. (2003). Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131.
- [MDA, 2008] MDA, O. (2008). Object management group model driven architecture.
- [Meseguer, 1996] Meseguer, J. (1996). Rewriting logic as a semantic framework for concurrency : a progress report. In *International Conference on Concurrency Theory*, pages 331–372. Springer.
- [Milner, 1999] Milner, R. (1999). *Communicating and mobile systems : the pi calculus*. Cambridge university press.
- [Milner, 2001] Milner, R. (2001). Bigraphical reactive systems. In *International Conference on Concurrency Theory*, pages 16–35. Springer.
- [Milner, 2008] Milner, R. (2008). Bigraphs and their algebra. *Electronic Notes in Theoretical Computer Science*, 209 :5–19.
- [Milner, 2009] Milner, R. (2009). *The space and motion of communicating agents*. Cambridge University Press.
- [Milner et al., 1997] Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The definition of standard ML : revised*. MIT press.
- [Mori et al., 2016] Mori, M., Ceccarelli, A., Lollini, P., Bondavalli, A., and Frömel, B. (2016). A holistic viewpoint-based sysml profile to design systems-of-systems. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 276–283. IEEE.
- [Muller IV, 2016] Muller IV, G. A. (2016). *A meta-architecture analysis for a coevolved system-of-systems*. Missouri University of Science and Technology.
- [Neto, 2016] Neto, V. V. G. (2016). Validating emergent behaviours in systems-of-systems through model transformations. In *SRC@ MoDELS*.
- [Nielsen and Larsen, 2012] Nielsen, C. B. and Larsen, P. G. (2012). Extending vdm-rt to enable the formal modelling of system of systems. In *System of Systems Engineering (SoSE), 2012 7th International Conference on*, pages 457–462. IEEE.
- [Nielsen et al., 2015] Nielsen, C. B., Larsen, P. G., Fitzgerald, J., Woodcock, J., and Peleska, J. (2015). Systems of systems engineering : basic concepts, model-based techniques, and research directions. *ACM Computing Surveys (CSUR)*, 48(2) :1–41.
- [Nilsson et al., 2020] Nilsson, R., Dori, D., Jayawant, Y., Petnga, L., Kohen, H., and Yokell, M. (2020). Towards an ontology for collaboration in system of systems context. In *INCOSE International Symposium*, volume 30, pages 666–679. Wiley Online Library.
- [Office, 2009] Office, N. A. (2009). *The National Offender Management Information System*.
- [Office, 2011] Office, N. A. (2011). The national programme for it in the nhs. *An update on the delivery of detailed care records systems. Forty-Fifth Report, Session 2010–12, (July)*,, page 1–80.
- [Olarte et al., 2013] Olarte, C., Rueda, C., and Valencia, F. D. (2013). Models and emerging trends of concurrent constraint programming. *Constraints*, 18(4) :535–578.
- [Ölveczky et al., 2010] Ölveczky, P. C., Boronat, A., and Meseguer, J. (2010). Formal semantics and analysis of behavioral aadl models in real-time maude. In *Formal Techniques for Distributed Systems*, pages 47–62. Springer.
- [Ölveczky and Meseguer, 2007] Ölveczky, P. C. and Meseguer, J. (2007). Semantics and pragmatics of real-time maude. *Higher-order and symbolic computation*, 20(1-2) :161–196.

- [Oquendo, 2016] Oquendo, F. (2016). pi-calculus for SoS : A foundation for formally describing software-intensive systems-of-systems. *System of Systems Engineering Conference (SoSE), 11th*.
- [Oquendo, 2018] Oquendo, F. (2018). Formally describing self-organizing architectures for systems-of-systems on the internet-of-things. In *European Conference on Software Architecture*, pages 20–36. Springer.
- [Oquendo and Legay, 2015] Oquendo, F. and Legay, A. (2015). Formal architecture description of trustworthy systems-of-systems with SoSADL. *ERCIM News*, (102).
- [Osmundson et al., 2006] Osmundson, J. S., Huynh, T. V., and Shaw, P. (2006). Developing ontologies for interoperability of systems of systems. In *Conference on Systems Engineering Research*.
- [Ozkaya and Kloukinas, 2013] Ozkaya, M. and Kloukinas, C. (2013). Are we there yet? analyzing architecture description languages for formal analysis, usability, and realizability. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 177–184. IEEE.
- [Pérez, 2006] Pérez, J. (2006). Prisma : Aspect-oriented software architectures. *Spain : Universitat Politècnica de València, PhD thesis*.
- [Pernin et al., 2012] Pernin, C. G., Axelband, E., Drezner, J. A., Dille, B. B., Gordon, I., Held, B. J., McMahon, K. S., Perry, W. L., Rizzi, C., Shah, A. R., et al. (2012). Lessons from the army’s future combat systems program. Technical report, RAND ARROYO CENTER SANTA MONICA CA.
- [Perrey and Lycett, 2003] Perrey, R. and Lycett, M. (2003). Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.*, pages 116–119. IEEE.
- [Perrone et al., 2012] Perrone, G., Debois, S., and Hildebrandt, T. T. (2012). A model checker for bigraphs. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1320–1325.
- [Plasil and Visnovsky, 2002] Plasil, F. and Visnovsky, S. (2002). Behavior protocols for software components. *IEEE transactions on Software Engineering*, 28(11) :1056–1076.
- [Popper et al., 2004] Popper, S. W., Bankes, S. C., Callaway, R., and DeLaurentis, D. (2004). System of systems symposium : Report on a summer conversation. *Potomac Institute for Policy Studies, Arlington, VA*, 320.
- [Quilbeuf et al., 2016] Quilbeuf, J., Cavalcante, E., Traonouez, L.-M., Oquendo, F., Batista, T., and Legay, A. (2016). A logic for the statistical model checking of dynamic software architectures. In *International Symposium on Leveraging Applications of Formal Methods*, pages 806–820. Springer.
- [Rebovich et al., 2009] Rebovich, G., DeRosa, J. K., and Norman, D. O. (2009). Improving the practice of systems engineering : Boot strapping grass roots success. In *2009 3rd Annual IEEE Systems Conference*, pages 323–326.
- [Rozier, 2011] Rozier, K. Y. (2011). Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2) :163–203.
- [Rubio et al., 2019] Rubio, R., Martí-Oliet, N., Pita, I., and Verdejo, A. (2019). Model checking strategy-controlled rewriting systems (system description). In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [Rubio et al., 2022] Rubio, R., Martí-Oliet, N., Pita, I., and Verdejo, A. (2022). Model checking strategy-controlled systems in rewriting logic. *Automated Software Engineering*, 29(1) :1–62.

- [Rubio Cuéllar et al., 2021] Rubio Cuéllar, R. R., Martí Oliet, N., Pita Andreu, I., and Verdejo López, J. A. (2021). The semantics of the maude strategy language.
- [Sevegnani and Calder, 2015] Sevegnani, M. and Calder, M. (2015). Bigraphs with sharing. *Theoretical Computer Science*, 577 :43–73.
- [Shortell, 2015] Shortell, T. M. (2015). In cose systems engineering handbook : a guide for system life cycle processes and activities. *John Wiley & Sons*.
- [Silva et al., 2015] Silva, E., Batista, T., and Oquendo, F. (2015). A mission-oriented approach for designing system-of-systems. In *2015 10th system of systems engineering conference (SoSE)*, pages 346–351. IEEE.
- [Silva et al., 2020] Silva, E., Batista, T., and Oquendo, F. (2020). On the verification of mission-related properties in software-intensive systems-of-systems architectural design. *Science of Computer Programming*, 192 :102425.
- [Stafford, 1979] Stafford, B. (1979). *The heart of enterprise : companion volume to Brain of the firm*. Wiley.
- [Stary and Wachholder, 2016] Stary, C. and Wachholder, D. (2016). System-of-systems support—a bigraph approach to interoperability and emergent behavior. *Data & Knowledge Engineering*, 105 :155–172.
- [Van Veelen et al., 2008] Van Veelen, J., Van Splunter, S., Wijngaards, N., and Brazier, F. (2008). Reconfiguration management of crisis management services. In *The 15th conference of the International Emergency Management Society (TIEMS 2008)*.
- [Wang et al., 2015] Wang, R., Agarwal, S., and Dagli, C. H. (2015). Opm & color petri nets based executable system of systems architecting : A building block in fila-sos. In *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pages 554–561. IEEE.
- [Woodcock and Cavalcanti, 2002] Woodcock, J. and Cavalcanti, A. (2002). The semantics of circus. In *International Conference of B and Z Users*, pages 184–203. Springer.
- [Woodcock et al., 2012] Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., and Perry, S. (2012). Features of cml : A formal modelling language for systems of systems. In *System of Systems Engineering (SoSE), 2012 7th International Conference on*, pages 1–6. IEEE.
- [Zeigler et al., 2013] Zeigler, B. P., Sarjoughian, H. S., Duboz, R., and Souli, J.-C. (2013). *Guide to modeling and simulation of systems of systems*, volume 516. Springer.