REPUBLIQUE DU CAMEROUN
Paix – Travail – Patrie
****
UNIVERSITE DE YAOUNDE I
FACULTE DES SCIENCES
DEPARTEMENT DE INFORMATIQUE
****
Centre de Recherche et de Formation Doctorale en Sciences, Technologies et Géosciences

REPUBLIC OF CAMEROUN
Peace – Work – Fatherland
****
UNIVERSITY OF YAOUNDE I
FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCES
****
Postgraduate School For Science, Technology and Geoscience

Une Approche de Composition de Services Dynamiques : application aux Systèmes Collaboratifs

Docteur/Ph.D en Informatique

Par : **KENGNE KUNGNE Willy**
Master en Informatique

Sous la direction de
KOUAMOU Georges-E
Maître de Conférences, Université de Yaoundé I
TANGHA Claude
Professeur, Université de Yaoundé I

Année Académique : 2019 - 2020

RÉPUBLIQUE DU CAMEROUN
PAIX-TRAVAIL-PATRIE
****
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
****
UNIVERSITÉ DE YAOUNDÉ I
****
CENTRE DE RECHERCHE ET DE FORMATION DOCTORALE EN SCIENCES, TE*CHNOLOGIES* ET *GEOSCIENCES*
****

REPUBLIC OF CAMEROON
PEACE-WORK-FATHERLAND
****
MINISTRY OF HIGHER EDUCATION
****
THE UNIVERSITY OF YAOUNDE I
****
POSTGRADUATE SCHOOL OF SCIENCE, TECHNOLOGY AND GEOSCIENCES
****

DÉPARTEMENT D'INFORMATIQUE
DEPARTMENT OF COMPUTER SCIENCE

ATTESTATION DE CORRECTION DE LA THESE DE DOCTORAT / Ph.D

Nous soussignés, **TCHUENTE Maurice, Pr., UYI, Président du jury; ATSA ETOUNDI Roger, Pr, UYI, NDOUNDAM René, MC., UYI, Examinateurs; KOUAMOU Georges-Edouard, MC, UY1, Rapporteur,** membres du jury de la thèse de Doctorat/Ph.D présenté par **M. KENGNE KUNGNE Willy, Matricule 06U160**, intitulé: **«Une Approche de Composition de Services Dynamiques : Application aux Systèmes Collaboratifs»** et soutenue en vue de l'obtention du diplôme de **Doctorat/Ph.D en informatique**, attestons que toutes les corrections demandées par le jury de soutenance en vue de l'amélioration de ce travail, ont été effectuées.

En foi de quoi la présente attestation lui est délivrée pour servir et valoir ce que de droit.

Président	Rapporteur	Examinateurs
TCHUENTE Maurice, Pr., UYI	KOUAMOU Georges, MC, UYI	ATSA ETOUNDI Roger, Pr, UYI
		NDOUNDAM René, MC., UYI

Liste Protocolaire

Division de la Programmation et du Suivi des Activités Académiques
Liste des enseignants permanents

ANNÉE ACADEMIQUE 2019/2020
(Par Département et par Grade)
DATE D'ACTUALISATION 03 Mars 2020

DOYEN : TCHOUANKEU Jean- Claude, *Maitre de Conférences*
VICE-DOYEN / DPSAA : ATCHADE Alex de Théodore, *Maitre de Conférences*
VICE-DOYEN / DSSE : AJEAGAH Gideon AGHAINDUM, *Professeur*
VICE-DOYEN / DRC : ABOSSOLO Monique, *Maitre de Conférences*
Chef Division Administrative et Financière : NDOYE FOE Marie C. F., *Maitre de Conférences*
Chef Division des Affaires Académiques, de la Scolarité et de la Recherche DAASR : MBAZE

1- DÉPARTEMENT DE BIOCHIMIE (BC) (38)

N°	NOMS ET PRÉNOMS	GRADE	OBSERVATIONS
1	BIGOGA DIAGA Jude	Professeur	En poste
2	FEKAM BOYOM Fabrice	Professeur	En poste
3	FOKOU Elie	Professeur	En poste
4	KANSCI Germain	Professeur	En poste
5	MBACHAM FON Wilfried	Professeur	En poste
6	MOUNDIPA FEWOU Paul	Professeur	Chef de Département
7	NINTCHOM PENLAP V. épse BENG	Professeur	En poste
8	OBEN Julius ENYONG	Professeur	En poste
9	ACHU Merci BIH	Maître de Conférences	En poste
10	ATOGHO Barbara Mma	Maître de Conférences	En poste
11	AZANTSA KINGUE GABIN BORIS	Maître de Conférences	En poste
12	BELINGA née NDOYE FOE M. C. F.	Maître de Conférences	Chef DAF / FS
13	BOUDJEKO Thaddée	Maître de Conférences	En poste
14	DJUIDJE NGOUNOUE Marcelline	Maître de Conférences	En poste
15	EFFA NNOMO Pierre	Maître de Conférences	En poste

16	NANA Louise épouse WAKAM	Maître de Conférences	En poste
17	NGONDI Judith Laure	Maître de Conférences	En poste
18	NGUEFACK Julienne	Maître de Conférences	En poste
19	NJAYOU Frédéric Nico	Maître de Conférences	En poste
20	MOFOR née TEUGWA Clotilde	Maître de Conférences	Inspecteur de Service MINESUP
21	TCHANA KOUATCHOUA Angèle	Maître de Conférences	En poste

22	AKINDEH MBUH NJI	Chargé de Cours	En poste
23	BEBOY EDZENGUELE Sara Nathalie	Chargé de Cours	En poste
24	DAKOLE DABOY Charles	Chargé de Cours	En poste
25	DJUIKWO NKONGA Ruth Viviane	Chargé de Cours	En poste
26	DONGMO LEKAGNE Joseph Blaise	Chargé de Cours	En poste
27	EWANE Cécile Anne	Chargé de Cours	En poste
28	FONKOUA Martin	Chargé de Cours	En poste
29	BEBEE Fadimatou	Chargé de Cours	En poste
30	KOTUE KAPTUE Charles	Chargé de Cours	En poste
31	LUNGA Paul KEILAH	Chargé de Cours	En poste
32	MANANGA Marlyse Joséphine	Chargé de Cours	En poste
33	MBONG ANGIE M. Mary Anne	Chargé de Cours	En poste
34	PECHANGOU NSANGOU Sylvain	Chargé de Cours	En poste
35	Palmer MASUMBE NETONGO	Chargé de Cours	En poste

36	MBOUCHE FANMOE Marceline Joëlle	Assistante	En poste
37	OWONA AYISSI Vincent Brice	Assistante	En poste
38	WILFRIED ANGIE Abia	Assistante	En poste

2- DÉPARTEMENT DE BIOLOGIE ET PHYSIOLOGIE ANIMALES (BPA) (48)

1	AJEAGAH Gideon AGHAINDUM	Professeur	*VICE-DOYEN / DSSE*
2	BILONG BILONG Charles-Félix	Professeur	Chef de Département
3	DIMO Théophile	Professeur	En poste
4	DJIETO LORDON Champlain	Professeur	En poste
5	ESSOMBA née NTSAMA MBALA	Professeur	*Vice Doyen/FMSB/UYI*
6	FOMENA Abraham	Professeur	En poste
7	KAMTCHOUING Pierre	Professeur	En poste

8	NJAMEN Dieudonné	Professeur	En poste
9	NJIOKOU Flobert	Professeur	En Poste
10	NOLA Moïse	Professeur	En Poste
11	TAN Paul VERNYUY	Professeur	En Poste
12	TCHUEM TCHUENTE Louis Albert	Professeur	*Inspecteur de service Coord.Progr./MINSANTE*
13	ZEBAZE TOGOUET Serge Hubert	Professeur	En Poste

14	BILANDA Danielle Claude	Maître de Conférences	En poste
15	DJIOGUE Séfirin	Maître de Conférences	En poste
16	DZEUFIET DJOMENI Paul Désiré	Maître de Conférences	En poste
17	JATSA BOUKENG Hermine épse MEGAPTCHE	Maître de Conférences	En poste
18	KEKEUNOU Sévilor	Maître de Conférences	En poste
19	MEGNEKOU Rosette	Maître de Conférences	En poste
20	MONY Ruth épse NTONE	Maître de Conférences	En poste
21	NGUEGUIM TSOFACK Florence	Maître de Conférences	En poste
22	TOMBI Jeannette	Maître de Conférences	En poste

23	ALENE Désirée Chantal	Chargé de Cours	En poste
26	ATSAMO Albert Donatien	Chargé de Cours	En poste
27	BELLET EDIMO Oscar Roger	Chargé de Cours	En poste
28	DONFACK Mireille	Chargé de Cours	En poste
29	ETEME ENAMA Serge	Chargé de Cours	En poste
30	GOUNOUE KAMKUMO Raceline	Chargé de Cours	En poste
31	KANDEDA KAVAYE Antoine	Chargé de Cours	En poste
32	LEKEUFACK FOLEFACK Guy B.	Chargé de Cours	En poste
33	MAHOB Raymond Joseph	Chargé de Cours	En poste
34	MBENOUN MASSE Paul Serge	Chargé de Cours	En poste
35	MOUNGANG LucianeMarlyse	Chargé de Cours	En poste
36	MVEYO NDANKEU Yves Patrick	Chargé de Cours	En poste
37	NGOUATEU KENFACK Omer Bébé	Chargé de Cours	En poste
38	NGUEMBOK	Chargé de Cours	En poste
39	NJUA Clarisse Yafi	Chargé de Cours	Chef Div. UBA
40	NOAH EWOTI Olive Vivien	Chargé de Cours	En poste
41	TADU Zephyrin	Chargé de Cours	En poste
42	TAMSA ARFAO Antoine	Chargé de Cours	En poste
43	YEDE	Chargé de Cours	En poste

44	ALENE Désirée Chantal	Assistant	En poste
45	ATSAMO Albert Donatien	Assistant	En poste
46	BELLET EDIMO Oscar Roger	Assistant	En poste
47	DONFACK Mireille	Assistant	En poste
48	ETEME ENAMA Serge	Assistant	En poste

3- DÉPARTEMENT DE BIOLOGIE ET PHYSIOLOGIE VÉGÉTALES (BPV) (33)

N°	NOMS ET PRÉNOMS	GRADE	OBSERVATIONS
1	AMBANG Zachée	Professeur	Chef Division/UYII
2	BELL Joseph Martin	Professeur	En poste
3	DJOCGOUE Pierre François	Professeur	En poste
4	MOSSEBO Dominique Claude	Professeur	En poste
5	YOUMBI Emmanuel	Professeur	Chef de Département
6	ZAPFACK Louis	Professeur	En poste
7	ANGONI Hyacinthe	Maître de Conférences	En poste
8	BIYE Elvire Hortense	Maître de Conférences	En poste
9	KENGNE NOUMSI Ives Magloire	Maître de Conférences	En poste
10	MALA Armand William	Maître de Conférences	En poste
11	MBARGA BINDZI Marie Alain	Maître de Conférences	CT/ MINESUP
12	MBOLO Marie	Maître de Conférences	En poste
13	NDONGO BEKOLO	Maître de Conférences	*CE / MINRESI*
14	NGODO MELINGUI Jean Baptiste	Maître de Conférences	En poste
15	NGONKEU MAGAPTCHE Eddy L.	Maître de Conférences	En poste
16	TSOATA Esaïe	Maître de Conférences	En poste
17	TONFACK Libert Brice	Maître de Conférences	En poste
18	DJEUANI Astride Carole	Chargé de Cours	En poste
19	GOMANDJE Christelle	Chargé de Cours	En poste
20	MAFFO MAFFO Nicole Liliane	Chargé de Cours	En poste
21	MAHBOU SOMO TOUKAM. Gabriel	Chargé de Cours	En poste
22	NGALLE Hermine BILLE	Chargé de Cours	En poste
23	NGOUO Lucas Vincent	Chargé de Cours	En poste
27	NNANGA MEBENGA Ruth Laure	Chargé de Cours	En poste
25	NOUKEU KOUAKAM Armelle	Chargé de Cours	En poste

26	ONANA JEAN MICHEL	Chargé de Cours	En poste

27	GODSWILL NTSOMBAH NTSEFONG	Assistant	En poste
28	KABELONG BANAHO Louis-Paul-Roger	Assistant	En poste
29	KONO Léon Dieudonné	Assistant	En poste
30	LIBALAH Moses BAKONCK	Assistant	En poste
31	LIKENG-LI-NGUE Benoit C	Assistant	En poste
32	TAEDOUNG Evariste Hermann	Assistant	En poste
33	TEMEGNE NONO Carine	Assistant	En poste

4- DÉPARTEMENT DE CHIMIE INORGANIQUE (CI) (34)

1	AGWARA ONDOH Moïse	Professeur	*Chef de Département*
2	ELIMBI Antoine	Professeur	En poste
3	Florence UFI CHINJE épouse MELO	Professeur	*Recteur Univ.Ngaoundere*
4	GHOGOMU Paul MINGO	Professeur	*Ministre Chargé De Miss.PR*
5	NANSEU Njiki Charles Péguy	Professeur	En poste
6	NDIFON Peter TEKE	Professeur	*CT MINRESI*
7	NGOMO Horace MANGA	Professeur	*Vice Chancelor/UB*
8	NDIKONTAR Maurice KOR	Professeur	*Vice-Doyen Univ. Bamenda*
9	NENWA Justin	Professeur	En poste
10	NGAMENI Emmanuel	Professeur	*DOYEN FS Uds*

11	BABALE née DJAM DOUDOU	Maître de Conférences	*Chargée Mission P.R.*
12	DJOUFAC WOUMFO Emmanuel	Maître de Conférences	En poste
13	EMADACK Alphonse	Maître de Conférences	En poste
14	KAMGANG YOUBI Georges	Maître de Conférences	En poste
15	KEMMEGNE MBOUGUEM Jean	Maître de Conférences	En poste
16	KONG SAKEO	Maître de Conférences	En poste
17	NDI NSAMI Julius	Maître de Conférences	En poste
18	NJIOMOU C. épse DJANGANG	Maître de Conférences	En poste
19	NJOYA Dayirou	Maître de Conférences	En poste

20	ACAYANKA Elie	Chargé de Cours	En poste
21	BELIBI BELIBI Placide Désiré	Chargé de Cours	CS/ ENS Bertoua
22	CHEUMANI YONA Arnaud M	Chargé de Cours	En poste
23	KENNE DEDZO GUSTAVE	Chargé de Cours	En poste
24	KOUOTOU DAOUDA	Chargé de Cours	En poste
25	MAKON Thomas Beauregard	Chargé de Cours	En poste
26	MBEY Jean Aime	Chargé de Cours	En poste
27	NCHIMI NONO KATIA	Chargé de Cours	En poste
28	NEBA nee NDOSIRI Bridget NDOYE	Chargé de Cours	CT/ MINFEM
29	NYAMEN Linda Dyorisse	Chargé de Cours	En poste
30	PABOUDAM GBAMBIE A.	Chargé de Cours	En poste
31	TCHAKOUTE KOUAMO Hervé	Chargé de Cours	En poste

32	NJANKWA NJABONG N. Eric	Assistant	En poste
33	PATOUOSSA ISSOFA	Assistant	En poste
34	SIEWE Jean Mermoz	Assistant	En poste

5- DÉPARTEMENT DE CHIMIE ORGANIQUE (CO) (35)

1	DONGO Etienne	Professeur	Vice-Doyen
2	GHOGOMU TIH Robert Ralph	Professeur	Dir. IBAF/UDA
3	NGOUELA Silvère Augustin	Professeur	Chef de Département UDS
4	NKENGFACK Augustin Ephrem	Professeur	Chef de Département
5	NYASSE Barthélemy	Professeur	En poste
6	PEGNYEMB Dieudonné Emmanuel	Professeur	*Directeur/MINESUP*
7	WANDJI Jean	Professeur	En poste

8	Alex de Théodore ATCHADE	Maître de Conférences	Vice-Doyen / DPSAA
9	EYONG Kenneth OBEN	Maître de Conférences	En poste
10	FOLEFOC Gabriel NGOSONG	Maître de Conférences	En poste
11	FOTSO WABO Ghislain	Maître de Conférences	En poste
12	KEUMEDJIO Félix	Maître de Conférences	En poste
13	KEUMOGNE Marguerite	Maître de Conférences	En poste
14	KOUAM Jacques	Maître de Conférences	En poste
15	MBAZOA née DJAMA Céline	Maître de Conférences	En poste

16	MKOUNGA Pierre	Maître de Conférences	En poste
17	NOTE LOUGBOT Olivier Placide	Maître de Conférences	Chef Service/MINESUP
18	NGO MBING Joséphine	Maître de Conférences	Sous/Direct. MINERESI
19	NGONO BIKOBO Dominique Serge	Maître de Conférences	En poste
20	NOUNGOUE TCHAMO Diderot	Maître de Conférences	En poste
21	TABOPDA KUATE Turibio	Maître de Conférences	En poste
22	TCHOUANKEU Jean-Claude	Maître de Conférences	*Doyen /FS/ UYI*
23	TIH née NGO BILONG E. Anastasie	Maître de Conférences	En poste
24	YANKEP Emmanuel	Maître de Conférences	En poste

25	AMBASSA Pantaléon	Chargé de Cours	En poste
26	KAMTO Eutrophe Le Doux	Chargé de Cours	En poste
27	MVOT AKAK CARINE	Chargé de Cours	En poste
28	NGNINTEDO Dominique	Chargé de Cours	En poste
29	NGOMO Orléans	Chargé de Cours	En poste
30	OUAHOUO WACHE Blandine M.	Chargé de Cours	En poste
31	SIELINOU TEDJON Valérie	Chargé de Cours	En poste
32	TAGATSING FOTSING Maurice	Chargé de Cours	En poste
33	ZONDENDEGOUMBA Ernestine	Chargé de Cours	En poste

34	MESSI Angélique Nicolas	Assistant	En poste
35	TSEMEUGNE Joseph	Assistant	En poste

6- DÉPARTEMENT D'INFORMATIQUE (IN) (27)

1	ATSA ETOUNDI Roger	Professeur	*Chef Div.MINESUP*
2	FOUDA NDJODO Marcel Laurent	Professeur	*Chef Dpt ENS/Chef IGA.MINESUP*

3	NDOUNDAM Réné	Maître de Conférences	En poste

4	AMINOU Halidou	Chargé de Cours	*Chef de Département*
5	DJAM Xaviera YOUH – KIMBI	Chargé de Cours	En Poste
6	EBELE Serge Alain	Chargé de Cours	En Poste
7	KOUOKAM KOUOKAM E. A.	Chargé de Cours	En Poste

8	MELATAGIA YONTA Paulin	Chargé de Cours	En Poste
9	MOTO MPONG Serge Alain	Chargé de Cours	En Poste
10	TAPAMO Hyppolite	Chargé de Cours	En Poste
11	ABESSOLO ALO'O Gislain	Chargé de Cours	En Poste
12	MONTHE DJIADEU Valery M.	Chargé de Cours	En Poste
13	OLLE OLLE Daniel Claude Delort	Chargé de Cours	C/D Enset. Ebolo
14	TINDO Gilbert	Chargé de Cours	En Poste
15	TSOPZE Norbert	Chargé de Cours	En Poste
16	WAKU KOUAMOU Jules	Chargé de Cours	En Poste

17	BAYEM Jacques Narcisse	Assistant	En poste
18	DOMGA KOMGUEM Rodrigue	Assistant	En poste
19	EKODECK Stéphane Gaël Raymond	Assistant	En poste
20	HAMZA Adamou	Assistant	En poste
21	JIOMEKONG AZANZI Fidel	Assistant	En poste
22	MAKEMBE. S . Oswald	Assistant	En poste
23	MESSI NGUELE Thomas	Assistant	En poste
24	MEYEMDOU Nadège Sylvianne	Assistant	En poste
25	NKONDOCK. MI. BAHANACK.N.	Assistant	En poste

7- DÉPARTEMENT DE MATHÉMATIQUES (MA) (31)

1	EMVUDU WONO Yves S.	Professeur	*Inspecteur MINESUP*

2	AYISSI Raoult Domingo	Maître de Conférences	Chef de Département
3	NKUIMI JUGNIA Célestin	Maître de Conférences	En poste
4	NOUNDJEU Pierre	Maître de Conférences	*Chef service des programmes & Diplômes*
5	MBEHOU Mohamed	Maître de Conférences	En poste
6	TCHAPNDA NJABO Sophonie B.	Maître de	Directeur/AIMS

		Conférences	Rwanda

7	AGHOUKENG JIOFACK Jean Gérard	Chargé de Cours	Chef Cellule MINPLAMAT
8	CHENDJOU Gilbert	Chargé de Cours	En poste
9	DJIADEU NGAHA Michel	Chargé de Cours	En poste
10	DOUANLA YONTA Herman	Chargé de Cours	En poste
11	FOMEKONG Christophe	Chargé de Cours	En poste
12	KIANPI Maurice	Chargé de Cours	En poste
13	KIKI Maxime Armand	Chargé de Cours	En poste
14	MBAKOP Guy Merlin	Chargé de Cours	En poste
15	MBANG Joseph	Chargé de Cours	En poste
16	MBELE BIDIMA Martin Ledoux	Chargé de Cours	En poste
17	MENGUE MENGUE David Joe	Chargé de Cours	En poste
18	NGUEFACK Bernard	Chargé de Cours	En poste
19	NIMPA PEFOUKEU Romain	Chargé de Cours	En poste
20	POLA DOUNDOU Emmanuel	Chargé de Cours	En poste
21	TAKAM SOH Patrice	Chargé de Cours	En poste
22	TCHANGANG Roger Duclos	Chargé de Cours	En poste
23	TCHOUNDJA Edgar Landry	Chargé de Cours	En poste
24	TETSADJIO TCHILEPECK M. E.	Chargé de Cours	En poste
25	TIAYA TSAGUE N. Anne-Marie	Chargé de Cours	En poste

26	MBIAKOP Hilaire George	Assistant	En poste
27	BITYE MVONDO Esther Claudine	Assistant	En poste
28	MBATAKOU Salomon Joseph	Assistant	En poste
29	MEFENZA NOUNTU Thiery	Assistant	En poste
30	TCHEUTIA Daniel Duviol	Assistant	En poste

8- DÉPARTEMENT DE MICROBIOLOGIE (MIB) (18)

1	ESSIA NGANG Jean Justin	Professeur	*Chef de Département*

2	BOYOMO ONANA	Maître de Conférences	En poste
3	NWAGA Dieudonné M.	Maître de Conférences	En poste

4	NYEGUE Maximilienne Ascension	Maître de Conférences	En poste
5	RIWOM Sara Honorine	Maître de Conférences	En poste
6	SADO KAMDEM Sylvain Leroy	Maître de Conférences	En poste

7	ASSAM ASSAM Jean Paul	Chargé de Cours	En poste
8	BODA Maurice	Chargé de Cours	En poste
9	BOUGNOM Blaise Pascal	Chargé de Cours	En poste
10	ESSONO OBOUGOU Germain G.	Chargé de Cours	En poste
11	NJIKI BIKOÏ Jacky	Chargé de Cours	En poste
12	TCHIKOUA Roger	Chargé de Cours	En poste

13	ESSONO Damien Marie	Assistant	En poste
14	LAMYE Glory MOH	Assistant	En poste
15	MEYIN A EBONG Solange	Assistant	En poste
16	NKOUDOU ZE Nardis	Assistant	En poste
17	SAKE NGANE Carole Stéphanie	Assistante	En poste
18	TOBOLBAÏ Richard	Assistant	En poste

9. DEPARTEMENT DE PYSIQUE(PHY) (40)

1	BEN- BOLIE Germain Hubert	Professeur	En poste
2	EKOBENA FOUDA Henri Paul	Professeur	*Chef Division. UN*
3	ESSIMBI ZOBO Bernard	Professeur	En poste
4	KOFANE Timoléon Crépin	Professeur	En poste
5	NANA ENGO Serge Guy	Professeur	En poste
6	NDJAKA Jean Marie Bienvenu	Professeur	Chef de Département
7	NOUAYOU Robert	Professeur	En poste
8	NJANDJOCK NOUCK Philippe	Professeur	*Sous Directeur/ MINRESI*
9	PEMHA Elkana	Professeur	En poste
10	TABOD Charles TABOD	Professeur	Doyen Univ/Bda
11	TCHAWOUA Clément	Professeur	En poste
12	WOAFO Paul	Professeur	En poste

13	BIYA MOTTO Frédéric	Maître de Conférences	DG/HYDRO Mekin
14	BODO Bertrand	Maître de Conférences	En poste
15	DJUIDJE KENMOE épouse ALOYEM	Maître de Conférences	En poste
16	EYEBE FOUDA Jean sire	Maître de Conférences	En poste
17	FEWO Serge Ibraïd	Maître de Conférences	En poste
18	HONA Jacques	Maître de Conférences	En poste
19	MBANE BIOUELE César	Maître de Conférences	En poste
20	NANA NBENDJO Blaise	Maître de Conférences	En poste
21	NDOP Joseph	Maître de Conférences	En poste
22	SAIDOU	Maître de Conférences	MINERESI
23	SIEWE SIEWE Martin	Maître de Conférences	En poste
24	SIMO Elie	Maître de Conférences	En poste
25	VONDOU Derbetini Appolinaire	Maître de Conférences	En poste
26	WAKATA née BEYA Annie	Maître de Conférences	*Sous Directeur/ MINESUP*
27	ZEKENG Serge Sylvain	Maître de Conférences	En poste

28	ABDOURAHIMI	Chargé de Cours	En poste
29	EDONGUE HERVAIS	Chargé de Cours	En poste
30	ENYEGUE A NYAM épse BELINGA	Chargé de Cours	En poste
31	FOUEDJIO David	Chargé de Cours	Chef Cell. MINADER
32	MBINACK Clément	Chargé de Cours	En poste
33	MBONO SAMBA Yves Christian U.	Chargé de Cours	En poste
34	MELI'I Joelle Larissa	Chargé de Cours	En poste
35	MVOGO ALAIN	Chargé de Cours	En poste
36	OBOUNOU Marcel	Chargé de Cours	DA/Univ Inter Etat/Sangmalima
37	WOULACHE Rosalie Laure	Chargé de Cours	En poste

38	AYISSI EYEBE Guy François Valérie	Assistant	En poste
39	CHAMANI Roméo	Assistant	En poste
40	TEYOU NGOUPOU Ariel	Assistant	En poste

10- DÉPARTEMENT DE SCIENCES DE LA TERRE (ST) (43)

1	BITOM Dieudonné	Professeur	*Doyen / FASA / Uds*
2	FOUATEU Rose épse YONGUE	Professeur	En poste
3	KAMGANG Pierre	Professeur	En poste
4	NDJIGUI Paul Désiré	Professeur	Chef de Département
5	NDAM NGOUPAYOU Jules-Remy	Professeur	En poste
6	NGOS III Simon	Professeur	DAAC/Uma
7	NKOUMBOU Charles	Professeur	En poste
8	NZENTI Jean-Paul	Professeur	En poste

9	ABOSSOLO née ANGUE Monique	Maître de Conférences	*Vice-Doyen / DRC*
10	GHOGOMU Richard TANWI	Maître de Conférences	CD/Uma
11	MOUNDI Amidou	Maître de Conférences	*CT/ MINIMDT*
12	NGUEUTCHOUA Gabriel	Maître de Conférences	CEA/MINRESI
13	NJILAH Isaac KONFOR	Maître de Conférences	En poste
14	ONANA Vincent Laurent	Maître de Conférences	*Chef service Maintenance & du Matériel*
15	BISSO Dieudonné	Maître de Conférences	*Directeur/Projet Barrage Memve'ele*
16	EKOMANE Emile	Maître de Conférences	En poste
17	GANNO Sylvestre	Maître de Conférences	En poste
18	NYECK Bruno	Maître de Conférences	MINERESI
19	TCHOUANKOUE Jean-Pierre	Maître de Conférences	En poste
20	TEMDJIM Robert	Maître de Conférences	En poste
21	YENE ATANGANA Joseph Q.	Maître de Conférences	*Chef Div. /MINTP*
22	ZO'O ZAME Philémon	Maître de Conférences	*DG/ART*

23	ANABA ONANA Achille Basile	Chargé de Cours	En poste

24	BEKOA Etienne	Chargé de Cours	En poste
25	ELISE SABABA	Chargé de Cours	En poste
26	ESSONO Jean	Chargé de Cours	En poste
27	EYONG JOHN TAKEM	Chargé de Cours	En poste
28	FUH Calistus Gentry	Chargé de Cours	*Sec. D'Etat/MINMIDT*
29	LAMILEN BILLA Daniel	Chargé de Cours	En poste
30	MBESSE CECILE OLIVE	Chargé de Cours	En poste
31	MBIDA YEM	Chargé de Cours	En poste
32	METANG Victor	Chargé de Cours	En poste
33	MINYEM Dieudonné-Lucien	Chargé de Cours	*CD/Uma*
34	NGO BELNOUN Rose Noël	Chargé de Cours	En poste
35	NGO BIDJECK Louise Marie	Chargé de Cours	En poste
36	NOMO NEGUE Emmanuel	Chargé de Cours	En poste
37	NTSAMA ATANGANA Jacqueline	Chargé de Cours	En poste
38	TCHAKOUNTE J. épse NOUMBEM	Chargé de Cours	*Chef.cell / MINRESI*
39	TCHAPTCHET TCHATO De P.	Chargé de Cours	En poste
40	TEHNA Nathanaël	Chargé de Cours	En poste
41	TEMGA Jean Pierre	Chargé de Cours	En poste

42	FEUMBA Roger	Assistant	En poste
43	MBANGA NYOBE Jules	Assistant	En poste

Répartition chiffrée des Enseignants de la Faculté des Sciences de l'Université de Yaoundé I

NOMBRE D'ENSEIGNANTS

DÉPARTEMENT	Professeurs	Maîtres de Conférences	Chargés de Cours	Assistants	Total
BCH	9 (1)	13 (09)	14 (06)	3 (2)	**39 (18)**
BPA	13 (1)	09 (06)	19 (05)	05 (2)	**46 (14)**
BPV	06 (0)	11 (02)	9 (06)	07 (01)	**33 (9)**
CI	10 (1)	9 (02)	12 (02)	03 (0)	**34 (5)**
CO	7 (0)	17 (04)	09 (03)	02 (0)	**35(7)**
IN	2 (0)	1 (0)	13 (01)	09 (01)	**25 (2)**
MAT	1 (0)	5 (0)	19 (01)	06 (02)	**31 (3)**
MIB	1 (0)	5 (02)	06 (01)	06 (02)	**18 (5)**
PHY	12 (0)	15 (02)	10 (03)	03 (0)	**40 (5)**
ST	8 (1)	14 (01)	19 (05)	02 (0)	**43(7)**
Total	**69 (4)**	**99 (28)**	**130 (33)**	**46 (10)**	**344 (75)**

Soit un total de **344 (**75**)** dont :

- Professeurs **69 (4)**
- Maîtres de Conférences **99 (**28**)**
- Chargés de Cours **130 (33)**
- Assistants **46** (10**)**

( ) = Nombre de Femmes **75.**

Résumé

La composition des services est un processus qui consiste à assembler plusieurs services en un afin de remplir des fonctions plus importantes. Dans les dernières décennies, de nombreux langages ont été proposés pour leur modélisation. La plupart d'entre eux sont inspirés de la modélisation les *Workflows*. Ces approches produisent des langages impératifs, qui sont rigides au changement car ils se concentrent sur la façon dont les processus doivent être construits. Malgré le fait que la sémantique (via les annotations) soit introduite dans les langages pour augmenter leur flexibilité, le dynamisme se limite à trouver des services qui ont disparu ou qui sont devenus défectueux. Ces langages n'offrent pas la possibilité d'adapter le service composite à l'exécution. Pour pallier à ces manquements, des langages déclaratifs basés sur les règles ont été introduits. Ils décrivent des contraintes pour orienter la construction des services composites, ils sont centralisés et restent très dépendants de BPML qui est la technologie sous-jacente. Cette thèse propose la spécification d'un langage purement déclaratif basé sur des règles pour la composition des services dans un environnement pair-à-pair. Une approche déclarative a des propriétés de flexibilité, d'adaptabilité, de ré-utilisabilité et même de sémantique formelle. Tout d'abord, nous proposons un langage déclaratif nommé *GSLang* pour la composition des services. Nous définissons un service composite comme une règle de production d'une grammaire avec une partie gauche (LHS) qui est le service à définir et une partie droite (RHS) constituée de services requis pour réaliser le service LHS. Les catégories syntaxiques présentant les concepts de *GSLang* et une description formelle de la sémantique opérationnelle qui met en évidence le dynamisme, la flexibilité et l'adaptabilité sont également définis. Ensuite, un framework de vérification est conçu. En utilisant l'Ingénieure Dirigée par les Modèles, nous définissons un méta-modèle pour le langage *GSLang*. Le framework de vérification traduit les spécifications de services *GSLang* vers le langage *Promela* pour la vérification. Ce dernier permet particulièrement de vérifier qu'une spécification de services *GSLang* se termine. Enfin un ensemble d'outils est proposé constitué d'un éditeur des spécifications *GSLang*(DSL), d'un moteur de transformation et d'un moteur d'exécution. Finalement, une étude de cas portant sur un système simplifié de gestion des missions est présentée afin de montrer comment spécifier complètement un système en utilisant notre approche et la mise en évidence de ses propriétés.

Mots clés : Approche orientée Règles, Architecture pair à pair, Chorégraphie de Services, Flexibilité par Changement, Adaptabilité, Ingénierie Dirigée par les Modèles, Vérification de Modèles, DSL.

Abstract

The composition of services is a process of grouping several services into a single service to perform important functions. In recent decades, many languages have been proposed for their modeling. Most of them are based on process-oriented approaches. The latter produce imperative languages, which are rigid to change because they focus on how processes should be constructed. Despite the fact that semantics are introduced in languages to increase their flexibility, dynamism is limited to finding services that have disappeared or have become defective. They do not offer the possibility of adapting the composite service to the execution. Although rule-based languages have been introduced, they describe constraints to guide the construction of composite services, they remain very dependent on BPML which is the underlying technology and are centralized. This thesis proposes the specification of a purely declarative language based on rules for the composition of services in a peer-to-peer environment. A declarative approach has properties of flexibility, adaptability, reusability and even formal semantics. First, we propose a declarative language named *GSLang* for the composition of services. We define a composite service as a rule of production of a grammar with a left-hand side (LHS) which is the service to define and a right-hand side (RHS) being the services required to realize the LHS service. The syntactic categories presenting the concepts of *GSLang* and a formal description of the operational semantics which highlights dynamism, flexibility and adaptability are defined. Next, a verification framework is designed. Using the Model Driven Engineer, we define a meta-model for the *GSLang* language. The verification framework translates the *GSLang* service specifications into the *Promela* language for the verification. The latter makes it possible to verify in particular that a service specification *GSLang* is resolved. Finally, a set of tools is proposed consisting of an editor for the *GSLang* specifications, a transformation engine and an execution engine. Finally, a case study of a simplified mission management system is presented in order to show how to specify a system completely using our approach and highlighting its properties.

keywords : Rule-based Approach, Peer-to-peer architecture, Service Choreography, Flexibility by Change, Adaptability, Model Driven Engineering, Model Checking, DSL.

. *A la mémoire de mon Papa Kungne Pierre.*

Remerciements

Je voudrais ici exprimer ma gratitude et mes remerciements à mon directeur de thèse Pr Georges-Edouard KOUAMOU pour sa confiance, ses encouragements et sa patience durant toutes ces années.

Une pensée à la mémoire du Professeur Claude TANGHA qui a supervisé ma thèse.

Je remercie le Professeur TCHUENTE Maurice pour avoir accepté de présider le jury de ma thèse.

Je remercie le Professeur TCHUENTE Maurice et le Professeur BADOUEL pour leurs rapports d'expertise qui ont conduit à la soutenance. Je remercie également les Professeurs ATSA ETOUNDI Roger et NDOUNDAM René pour avoir accepté d'examiner ma thèse. Merci infiniment pour vos remarques qui m'ont permis d'améliorer la qualité de la thèse.

Je remercie le LIRIMA (Laboratoire International de Recherche en Informatique et en Mathémathiques Appliquées) pour leur soutien financier à travers l'équipe FUSCHIA. Je tiens aussi à remercier le CETIC (Centre d'Excellence Africain en Technologies de l'Information et de la Communication) pour leur accompagnement financier durant une partie de ma thèse.

Je remercie les enseignants du Département d'Informatique de l'Université de Yaoundé I pour leurs conseils et surtout pour la qualité de la formation dont j'ai bénéficié laquelle a contribué à l'aboutissement de cette thèse.

Je remercie les enseignants du Génie Informatique de ENSPY (École Nationale Supérieure Polytechnique de Yaoundé) pour l'accueil et les conseils durant ces années laborieuses de doctorat.

Je remercie mes amis et mes camarades de classe (Silvère Foyang, Omar Djiomou, Léonie Tamo, Achille Nyabeye, Franklin Wouleu, Brilland Nkuete, Carole Noutchegueme, Ghislaine Tiomo, Laure Tchuente, Vanessa Kamga, Ghislain Meleu, Brice Bessala, Wilfried Tedongmo, Maxime Tchinda, Stéphane Ekodeck, Waffo Austin, Billong IV Ismael...) pour vos aides et discussions enrichissantes.

Je remercie Darline Staelle, Léonie, Lucie, Anna, Laurie, Robert et Silvère pour les multiples relectures des articles et de la thèse. Merci infiniment.

Je remercie ma mère Yuego Genevieve ainsi que mes frères (Francis, Valère et Maxime) et mes Sœurs (Lucie et Carine) pour leur disponibilité, leur soutien inconditionnel durant ces années de doctorat.

A mon épouse Darline Staelle, pour la patience et le soutien dont elle a fait preuve pendant toute la durée de cette thèse. Thank you my Darling.

Contents

List of Figures

List of Tables

Abreviations

ACP	**A**lgebra of **C**ommunicating **P**rocesses
API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
ATL	**A**tlas **T**ransformation **L**anguage
AO4BPEL	**A**spect **O**riented for **BPEL**
BPEL	**B**usiness **P**rocess **E**xecution **L**anguage
BPEL2PNML	**BPEL** to **P**etri **N**et **M**arkup **L**anguage
BPEL2PN	**BPEL** to **P**etri **N**et
BPEL4chor	**BPEL** for **chor**eography
BPEL4WS	**BPEL** for **W**eb **S**ervice
BPML	**B**usiness **P**rocess **M**odeling **L**anguage
BPMN	**B**usiness **P**rocess **M**odel and **N**otation
CA	**C**ondition **A**ction
CCS	**C**alculus of **C**ommunicating **S**ystems
CSP	**C**ommunicating **S**equential **P**rocesses
DSL	**D**omain **S**pecific **L**anguage
EMF	**E**clipse **M**odeling **F**ramework
FARAO	**FrAmewoRk** for **A**daptive **O**rchestration
GAG	**G**uarded **A**ttribute **G**rammars
GSLang	**G**AG **S**ervice **L**anguage
GO-BPMN	**G**oal **O**riented **BPMN**
HoD	**H**ead **o**f **D**epartement
IDE	**I**ntegrated **D**evelopment **E**nvironment
IDM	**I**ngénierie **D**irigée par les **M**odèles

J2EE	**J**ava **E**nterprise **E**dition
JMS	**J**ava **M**essage **S**ervice
JSF	**J**ava **S**erver **F**aces
JVM	**J**ava **V**irtual **M**achine
LHS	**L**eft-**H**and **S**ide
LOTOS	**L**anguage of **T**emporal **O**rdering **S**pecification
LTL	**L**inear **T**emporal **L**ogic
MDA	**M**odel **D**riven **A**rchitecture
MDE	**M**odel **D**riven **E**ngineering
MMS	**M**ission **M**anagement **S**ystem
MOF	**M**eta **O**bject **F**acility
MOM	**M**essage **O**riented **M**iddleware
OMG	**O**bject **M**nangement **G**roup
PDM	**P**latform-**D**escription **M**odel
PIM	**P**latform-**I**ndependent **M**odel
PROMELA	**PRO**cess **M**eta **LA**nguage
PSM	**P**latform-**S**pecific **M**odel
QVT	**Q**uery **V**iew **T**ransformation
REST	**R**epresentational **S**tate **T**ransfer
RHS	**R**ight-**H**and **S**ide
SCA	**S**ervices **C**omponent **A**rchitecture
SELF-SERV	compo**S**ing w**E**b accessib**L**e in**F**ormation and bu**S**iness s**ERV**ices
SOA	**S**ervice **O**riented **A**rchitecture
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
SOC	**S**ervice-**O**riented **C**omputing
SPIN	**S**imple **P**romela **IN**terpreter
UML	**U**nified **M**odeling **L**anguage
WS-BPEL	**W**eb **S**ervice-**BPEL**
WS-CDL	**W**eb **S**ervice **C**horeography **D**escription **L**anguage
WSCI	**W**eb **S**ervice **C**horeography **I**nterface
WSTTS	**W**eb **S**ervice **T**imed **T**ransition **S**ystems

WSDL	**W**eb **S**ervices **D**escription **L**anguage
XML	**E**xtensible **M**arkup **L**anguage
Xpath	**X**ML **P**ath **L**anguage
XSD	**X**ML **S**chema **D**efinition

Introduction Générale

1.1 Contexte

Le développement des réseaux et la mobilité humaine accrue ont favorisé l'émergence des systèmes distribués. Pour répondre à cette préoccupation, plusieurs paradigmes ont été proposés parmi lesquels les Architectures Orientées Services (SOA). La SOA a été développée comme une solution palliative aux insuffisances des offres traditionnelles des intergiciels couramment utilisés pour distribuer les objets. L'ingénierie des services est une discipline du Génie Logiciel qui implique plusieurs types d'activités parmi lesquelles le développement des services et la composition des services. Cette dernière consiste à assembler plusieurs services, même hétérogènes pour construire une application [1, 2]. Les modèles sous-jacents à la composition de services s'inspirent des modèles de $Workflows$. Les approches de composition de services y afférents proposent un ensemble d'éléments qui permettent de décrire un service composite sous la forme d'un enchaînement de tâches. Ces approches sont inadéquates étant donné que le nombre de services à intégrer dans un composite peut être important. Et dans ces cas-là, la compréhension de chacun des services sous-jacents n'est pas aisée. En outre, les services peuvent être composés dans le cadre d'un partenariat à court terme et dissous instantanément avec la fin du contrat. Par conséquent, l'intégration d'un grand nombre de services nécessite des techniques évolutives et flexibles, telles que celles basées sur des langages déclaratifs. Sur le plan architectural, l'exécution d'un service composite basée sur ces approches de $Workflows$ est généralement centralisée. Un modèle centralisé entraîne de graves problèmes, notamment d'évolutivité, de disponibilité et même de sécurité [3]. Étant donné la nature hautement dynamique et distribuée des services, les techniques impliquant l'exécution de services dans un style architectural pair à pair sont mieux indiqués.

Les systèmes collaboratifs [4] impliquent plusieurs rôles ou acteurs. Pour atteindre le but, chaque rôle offre des services aux autres rôles du système et parfois requiert des services des tiers. Par conséquent, un système collaboratif est appréhendé comme un ensemble de services issus des différents rôles qui collaborent. Il peut aussi arriver que, le but varie en fonction des préférences des utilisateurs finaux qui ne peuvent être conçu à priori. Par exemple, dans un système de réservation de vol en ligne, l'objectif peut être différent en fonction des utilisateurs. La requête peut être une combinaison des situations suivantes:

1. réserver un vol;
2. réserver un hôtel;
3. réserver une voiture;
4. connaître la météo durant son séjour;

Ces exigences peuvent subvenir alors que le client demande l'achat d'un ticket de voyage. Le refus de la prise en compte de ces exigences peut entraîner un désistement provoquant ainsi des pertes économiques importantes pour l'entreprise de réservation. Il y a donc un besoin de faire évoluer les objectifs d'un système à tout moment en fonction des exigences des utilisateurs. D'un autre côté, ces besoins peuvent évoluer considérablement rendant ainsi le système complexe et difficile à construire et à maintenir.

Dans la mise sur pied d'un tel système collaboratif flexible et adaptable en permanence selon les exigences des utilisateurs finaux, toutes les collaborations ne peuvent pas être prévues à l'avance telles que promeuvent les paradigmes et langages qui les modélisent actuellement. Les langages utilisés pour modéliser les workflows [5] ou encore les systèmes de gestion des cas [6][7] ont des insuffisances. Ils sont définis sur des architectures rigides. Par exemple dans les systèmes de workfow toutes les tâches à accomplir sont définies à l'avance comme dans un automate. La modélisation des systèmes de gestion de cas existants [7] ont une architecture centralisée, ce qui signifie qu'il y a une seule perspective pour gérer le cas. De plus, ils mettent les rôles au second plan. Les langages qui permettent de les modéliser sont de type orchestration (BPEL[1] , Jolie [8]). Ils ne sont pas adaptés pour les systèmes collaboratifs où les rôles ont une importance capitale. De plus leur description est impérative ou procédurale

[1] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

comme BPMN[2], ils permettent une définition rigide et statique et ne favorise pas la flexibilité et l'adaptation des tâches lors de leur exécution. Quant aux langages de type chorégraphie existants (WS-CDL[3], choreographic programming [9], BPEL4Chor [10], GO-BPMN[11]), ils permettent de définir une perspective globale des interactions entre les participants. Une fois définis, ils sont projetés pour être exécutés. Ce principe ne favorise pas l'adaptation des interactions aux besoins des utilisateurs, lesquelles interactions sont prédéfinies et décrites à priori.

1.2 Problématique

Dès lors, on peut se poser la question de savoir comment prendre en compte en permanence les nouveaux besoins utilisateurs même pendant l'exécution d'un système collaboratif? Récemment, a été introduit le modèle de GAGs [12] pour la gestion des cas. Le GAG (Guarded Attribute Grammars) est un modèle de travail coopératif basé sur la résolution des tâches. Une tâche est résolue en la divisant en tâches plus petites, en résolvant ces sous-tâches et en combinant leurs résultats pour produire la sortie requise. La décomposition des tâches en tâches plus petites est modélisée comme des règles de réécriture en utilisant les productions d'une grammaire attribuée. Les attributs hérités et synthétisés de la grammaire modélisent respectivement les paramètres d'entrée et de sortie des tâches, tandis que les règles sémantiques sont utilisées pour imposer le flux de données entre une tâche et ses sous-tâches, et entre les tâches sœurs. Ainsi, il permet de suivre l'exécution d'un cas au sein des artefacts. Le modèle de GAGs est déclaratif c'est-à-dire qu'il est moins rigide contrairement aux approches existantes. Néanmoins, il est très formel même dans son implémentation actuelle basée sur la programmation fonctionnelle [13] très peu répandue. Il met en œuvre des mécanismes fortement couplés pour la communication des espaces de travail (rôle). Dans un environnement plus ouvert tel qu'internet, il convient d'utiliser des paradigmes orientés services. Étant donné que les systèmes collaboratifs se mettent en place dans des architectures pair-à-pair, nous pensons à la chorégraphie de services. Nous voulons prendre en compte les besoins utilisateurs c'est-à-dire de nouveaux services même durant l'exécution d'un service composite dans une chorégraphie. Les chorégraphies supportent une vision collaborative d'une application distribuée. Elles spécifient dans une perspective globale les interactions entre les rôles. Une chorégraphie

[2]http://www.bpmn.org/
[3]http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217

est une spécification de ce que les participants à la collaboration, ou les rôles, doivent suivre et atteindre. De part sa définition et les langages de spécifications des chorégraphies existantes, il est très difficile de modifier une chorégraphie en cours d'exécution afin de prendre en compte de nouvelles exigences des utilisateurs car ils décrivent les interactions globales entre rôles. En cas de changements dans les interactions, il faut tout arrêter, revenir dans la spécification et la projeter à nouveau ce qui est inhérent au caractère impératif des langages existants pour la description des interactions entre les services. Au lieu de ça, nous voulons pour un système collaboratif, décrire le comportement de chaque rôle de manière déclarative à l'aide d'un langage basé sur les règles.

1.3 Contributions de la thèse

Dans cette thèse, nous étendons le modèle des GAGs dans le domaine de la composition de services de type chorégraphique. Au lieu de décrire les interactions globales entre les rôles dans une chorégraphique, nous décrivons le comportement de chaque service contenu dans les rôles afin de prendre en compte à tout moment de nouveaux services ou changements dans une composition. Ainsi, nous proposons un langage déclaratif pour la composition de services, ainsi qu'un cadre de vérification et de validation.

1.3.1 Flexibilité à priori

Nous proposons un langage purement déclaratif dont le formalisme est basé sur les règles de production d'une Grammaire Attribuée Gardée. Un service composite est défini comme une règle de production d'une grammaire avec un côté gauche (LHS) qui est le service à définir et un côté droit (RHS) constitué de services requis pour réaliser le service LHS. Lorsque le RHS est vide, le service est élémentaire et peut être assimilé à un protocole standard ou à un style de service architectural tel que SOAP ou REST. Chaque service est protégé par des conditions (gardes) permettant son activation. Nous formalisons notre modèle en définissant les concepts généralement présents dans le domaine de la composition de services tels que: variables (paramètres), service, instance de service, garde, rôles, messages et actions. Nous décrivons les règles sémantiques qui incluent entre autres, les opérations suivantes: instanciation, envoi et réception de messages, raffinement et choix des services pour leur exécution. Les services sont regroupés dans des pairs localisés grâce à leurs ports publics. Cette

description purement déclarative et abstraite des services permet de prendre en compte des nouveaux besoins même pendant l'exécution d'un service car les composites sont définis à la volée (peuvent être modifiés ou ajoutés sans arrêter le système) rendant le langage flexible et adaptable.

1.3.2 Cohérence entre les modèles: transformation de modèles

Le passage de la spécification des services jusqu'à leur opérationnalisation doit être automatique. Cette exigence est un principe fondamental de l'ingénierie dirigée par les modèles (IDM). Le langage est décrit en utilisant le formalisme du pi-calcul [14]. Le choix de ce formalisme est motivé par la répartition des pairs à travers un réseau et leur interaction qui se fait via des ports dynamiques, créés lors de l'exécution. Une fois décrit, nous vérifions que les services spécifiés se terminent. Pour y arriver, nous transformons grâce à l'IDM (pour garder la cohérence) les spécifications vers PROMELA (PROcess Meta LAnguage). Nous avons proposé un arbre de syntaxe abstraite pour la spécification des services qui nous sert de modèles sources. Ensuite nous avons mis sur pied des règles de transformation vers les modèles PROMELA. Ce dernier nous permet d'effectuer la simulation des services spécifiés afin de s'assurer qu'ils s'exécuteront correctement et peuvent être opérationnalisés. Enfin, le raffinement pour sa prise en compte dans le système opérationnel est alors effectué.

1.3.3 Framework de vérification et de validation

Le framework proposé est constitué d'un environnement de conception et d'exécution pour le langage *GSLang*. Il permet :

1. de vérifier que toutes les spécifications de service sont cohérentes ou que les modifications apportées aux services sont cohérentes;
2. de raffiner la spécification de service pour le pair vers le système opérationnel lorsque le point 1 est correctement effectué. Ici, les éléments tels que la nature des services et leur emplacement sont ajoutés;
3. d'exécuter les services obtenus grâce au moteur d'exécution. Ce dernier utilise un système opérationnel basé sur XML pour répondre aux besoins des utilisateurs en combinant de manière dynamique différents services.

1.4 Plan de la thèse

La suite du document est structurée en quatre (4) chapitres organisés autour de nos contributions.

- Le **chapitre 1** présente dans un premier temps l'état de l'art des langages de composition de services en insistant sur l'aspect architectural (orchestration ou chorégraphie) et le paradigme (impératif ou déclaratif) utilisé pour la construction des composites. D'après la littérature, le paradigme déclaratif apporte plus de flexibilité dans la construction des composites. Ensuite, nous présentons des approches formelles permettant de raisonner lors de la composition de services dans le but de garantir l'interaction correcte des services car de nombreuses erreurs peuvent se produire: messages non reçus, interblocages, comportements incompatibles. Plusieurs méthodes formelles ont été proposées, la plupart avec une sémantique basée sur des systèmes de transition tels que les automates, les réseaux de Petri ou les algèbres de processus. Nous avons choisi d'utiliser l'algèbre de processus (pi-calcul). Le choix de ce formalisme est motivé par la répartition des pairs sur un réseau et leur interaction, qui se fait via des ports dynamiques qui sont créés lors de l'exécution.

- Le **chapitre 2** présente le langage proposé pour la composition de services. Ce langage est nommé GSLang pour *GAG Service Language* car nous exploitons la structure des GAGs qui est purement déclarative et orienté-données. Nous présentons d'abord les propriétés des approches déclaratives, ensuite nous présentons les concepts de base, la syntaxe et la sémantique opérationnelle de GSLang ainsi que la démonstration des propriétés de GSLang.

- Le **chapitre 3** présente un framework pour la vérification et la validation des spécifications GSLang. La vérification consiste à simuler une spécification GSLang afin de déterminer si elle est correcte afin de la projeter vers un système opérationnel. A cet effet, L'IDM (Ingénierie Dirigée par les Modèles) est utilisée pour automatiser la transformation des spécifications en PROMELA pour leur vérification d'une part et le raffinement vers le système opérationnel (XML) d'autre part. Nous proposons un méta-modèle source pour le langage GSLang et des règles de transformation vers *PROMELA (PROcess Meta LAnguage)* pour la vérification.

Enfin, nous proposons un mécanisme de raffinement.

- Le **chapitre 4** présente dans un premier temps l'environnement logiciel de l'éditeur des spécifications GSLang conçu, les règles de transformation et le moteur d'exécution. L'éditeur est développé comme une DSL par l'entremise de XText permettant d'effectuer un contrôle de syntaxe lors de l'édition des services et d'affiner la spécification GSLang vers le système opérationnel. Les règles de transformation sont écrites en ATL(Atlas Transformation Language). Le moteur d'exécution en J2EE a un aspect visuel (interface web) et un backend qui implémente la sémantique opérationnelle de GSLang et est basé sur le MOM(Message Oriented Middleware). Ensuite l'expérimentation de ces outils sur un exemple de gestion simplifiée des missions dans une organisation a été réalisé. Nous présentons tout d'abord une description détaillée du cas. Ensuite nous le décrivons dans l'éditeur, montrons comment vérifier les services spécifiés et présentons en détail l'exécution des services. Enfin une discussion est faite par rapport au style architectural, les approches déclaratives et orienté-données.

- La **conclusion générale** présente une synthèse du travail effectué dans cette thèse et des perspectives énonçant des pistes futures à explorer sur le plan de l'évolutivité du modèle, l'amélioration de l'environnement technique et la sécurité des échanges.

1.5 Liste de publications

1.5.1 Papiers de journaux

1. Kungne, W. K., Kouamou, G. E., and Tangha, C. (2020). A Rule-Based Language and Verification Framework of Dynamic Service Composition. Future internet, 12(2), 23. (Scopus)

2. Kouamou, G. E., and Kungne, W. K. (2017). A Structural and Generative Approach to Multilayered Software Architectures. Journal of Software Engineering and Applications, 10(8), 677-692.

1.5.2 Conférences

1. Kungne, W.K.; Kouamou, G.E.; Tangha, C. Introducing an Artifact-driven language for Service Composition. In Proceedings of the ArabWIC 6th Annual International Conference Research Track, Rabat, Morocco, 6-8 March 2019; pp. 1-6. (ACM Conference)

2. Willy Kengne Kungne, Georges Edouard Kouamou, Claude Tangha. Extending an artifact-driven workflow model to service composition. Conference de Recherche en Informatique (CRI) 2019, Yaoundé.

.

Chapter 1

État de l'art

Sommaire

1.1 Introduction

L'approche orientée-services, en anglais Service-Oriented Computing (SOC), est un terme qui a été largement exploré durant ces deux dernières décennies. Elle est conçue pour permettre l'intégration de systèmes logiciels hétérogènes et distribués. Cette approche propose un style architectural qui cherche à faciliter la construction des applications à partir d'entités logicielles faiblement couplées, hétérogènes et distribuées, connues sous le nom de services. Dans ce chapitre, nous présentons le cycle de vie de la

composition des services, nous étudions également les langages pour la composition de services en mettant l'accent sur sa description et le paradigme (impératif vs déclaratif) utilisé pour leur construction. Enfin nous présentons des approches formelles qui ont été proposées.

1.2 Les services : définition et standards

Le service représente la brique de base dans une architecture orientée services. Il est assez difficile de donner une définition précise au service car il est utilisé dans plusieurs domaines. Il peut être défini tout simplement comme suit : un service est une action réalisée par une entité au profit d'une autre. Les définitions régulièrement utilisées dans cette étude sont celles mises en évidence par Papazoglou [15] [16] [17]

«Services are self-describing, platform agnostic computational elements. They are autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel way» [16]

Les services sont des entités logicielles indépendantes qui peuvent être publiées, découvertes par des tiers (consommateurs) et consommées sans savoir comment ils ont été implémentés. Ils sont donc indépendants des plateformes. Cette indépendance offre le faible couplage entre les consommateurs de services et les fournisseurs de services. Ce sont aussi des unités de calculs qui peuvent être utilisées seules ou être composées avec d'autres services. Une autre définition de services proposée est: «A service is a discrete unit of business functionality that is made available through a service contract. The service contract specifies all interactions between the service consumer and service provider. This includes: - Service interface; - Interface documents; - Service policies; - Quality of service (QoS); - Performance» [17].

Cette définition met l'accent sur le caractère description d'un service et met en évidence les éléments (service contract) qu'un fournisseur de services établit pour permettre à un client de consommer le service. Un service a un aspect externe matérialisé par l'interface (le contrat entre le fournisseur et le consommateur d'un service) connu du fournisseur mais surtout des consommateurs et un aspect interne matérialisé par l'implémentation connue uniquement par le fournisseur du service.

L'interface d'un service spécifie les opérations du service, qui est en fait ce que le service fait, les paramètres en entrée et en sortie d'une opération et les protocoles de communication.

L'implémentation d'un service quant à elle est la façon dont le service fournit les capacités de son interface. L'implémentation peut être basée sur des applications existantes, sur l'orchestration d'autres services afin de combiner leurs fonctionnalités ou sur le code écrit spécifiquement pour le service.

Il existe actuellement deux implémentations concrètes des services : les services Web traditionnels basés sur SOAP et les services Web RESTful plus simples sur le plan conceptuel [18].

L'un des challenges dans les architectures orientées services est la composition de services. Elle permet de proposer un service plus complexe en utilisant d'autres services de moindres envergures.

1.3 Composition de services

La composition des services est le processus consistant à regrouper plusieurs services en un seul service afin de remplir des fonctions plus complexes. En d'autres termes, elle permet de développer des applications en réutilisant des services existants pour créer d'autres services plus importants.

1.3.1 Phase de composition des services

Le processus de composition de services est réalisé à travers plusieurs phases qui permettent le passage de la spécification abstraite d'une composition vers la spécification concrète à l'exécution de services [2][19] :

- Phase de définition. Au cours de cette phase, les exigences de composition des services sont identifiées. Les exigences sont ensuite décomposées, en un modèle de processus abstrait (c'est-à-dire le service composite abstrait), qui spécifie un ensemble d'activités, le contrôle et le flux de données entre eux. Cette phase permet donc de définir des services composites d'une manière abstraite. La

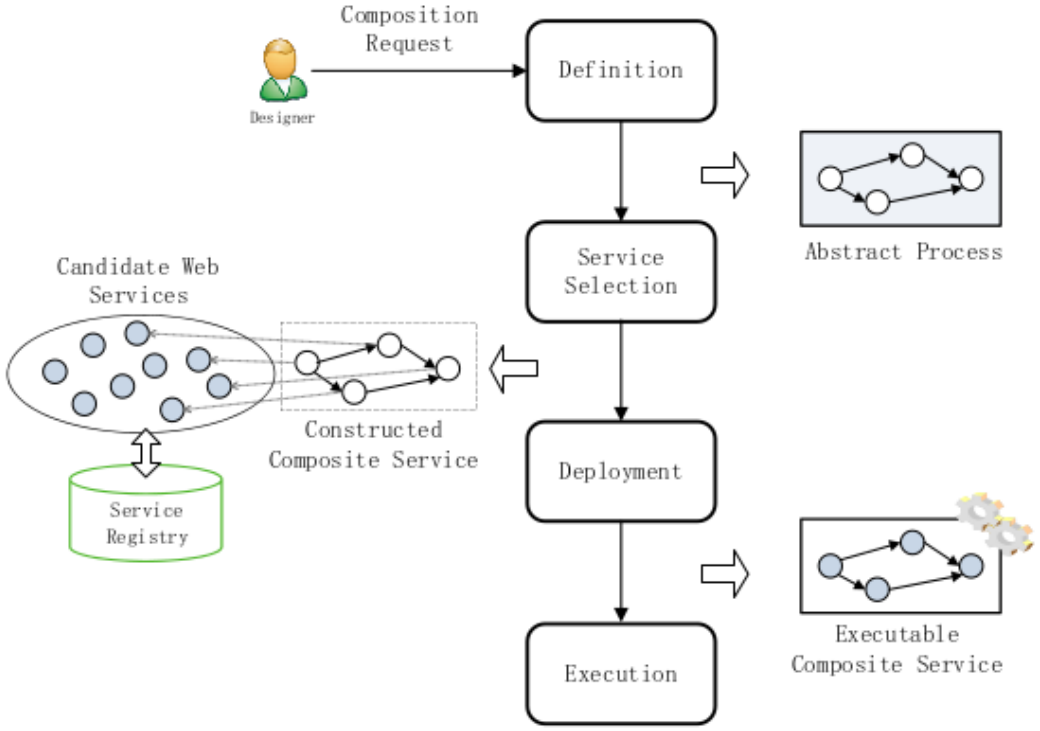

FIGURE 1.1: Processus de composition [2]

fonctionnalité fournie par la composition ainsi que les fonctionnalités apportées par les différents services participants doivent être identifiés.

- Phase de sélection des services. Dans cette phase, pour chaque activité du service composite abstrait, les services appropriés qui correspondent aux exigences de l'activité sont localisés en recherchant dans le registre des services par exemple. Il est probable que plus d'un service candidat satisfasse aux exigences. Par conséquent, le service le mieux adapté doit être sélectionné. Une fois tous les services requis identifiés et liés aux activités correspondantes, le service composite construit est produit. Cette phase vise donc à identifier l'ensemble des services requis conformes à la description abstraite de la composition définie dans la phase précédente.

- Phase de déploiement. Dans cette phase, le service composite construit est déployé pour permettre son instanciation et son invocation par les utilisateurs finaux. Le résultat de cette phase est le *service composite exécutable*.

- Phase d'exécution. Dans cette phase, l'instance de service composite sera créée et exécutée par le moteur d'exécution, qui est également responsable de l'appel des composants de service individuels.

Le processus (figure 1.1) de composition de services consiste ainsi tout d'abord à définir un service composite abstrait par les descriptions des services requis, ensuite à découvrir et à sélectionner les services parmi ceux qui sont disponibles, et enfin à réaliser les liaisons entre les services sélectionnés.

La découverte, la sélection et la liaison de services (c'est-à-dire, les phases de planification et de construction) peuvent être réalisées avant ou pendant la phase d'exécution caractérisant ainsi la composition. Lorsque tous les services participant à la composition sont sélectionnés et liés avant la phase d'exécution, la composition est appelée composition statique. Quand la sélection et la liaison de services peuvent être réalisées à l'exécution, la composition est nommée composition dynamique. Enfin, lorsque les sélections et les liaisons déjà réalisées peuvent être modifiées à l'exécution, la composition est appelée composition adaptative dynamique.

Malgré sa décomposition en plusieurs phases, le processus de composition de services reste une activité complexe. En effet, chacune des phases est divisée en tâches pas aisées. Par exemple, les services identifiés pour la composition peuvent présenter des problèmes d'incompatibilité (des types de données d'entrée et de sortie des services par exemple). Des mécanismes de médiation doivent ainsi être créés et utilisés dans la phase de construction afin d'adapter les services les uns aux autres et de résoudre les problèmes d'incompatibilité. Cela est une tâche difficile qui, dans la plupart des approches, est effectuée manuellement par les développeurs des applications. La complexité de la composition de services est associée non seulement à la complexité de la logique métier des applications, mais aussi à la complexité technique de la réalisation de l'approche à services. Les développeurs doivent avoir une double expertise : l'expertise métier et l'expertise technique nécessaire pour réaliser les compositions de services en utilisant des technologies particulières. Afin de permettre aux développeurs de se concentrer sur la logique métier des applications plutôt que sur les détails techniques, il est nécessaire de disposer de langages, de modèles et de mécanismes pour la réalisation de compositions de services.

1.3.2 Approches de composition de services

Il existe diverses approches pour la réalisation de compositions de services. Elles sont souvent classifiées par rapport à la façon dont le contrôle de la composition est géré : extrinsèque ou intrinsèque aux services. Ces deux façons de gérer le contrôle définissent

deux styles de composition : la composition par procédés, aussi appelée composition comportementale, et la composition structurelle.

1.3.2.1 Composition par procédés

Dans cette approche, la composition de services est définie en spécifiant la logique de coordination de services par un procédé. Un procédé est généralement représenté par un graphe orienté d'activités et le flot de contrôle qui établit l'ordre d'invocation des activités. Chaque activité représente une fonctionnalité réalisée concrètement par un service. La composition est décrite dans un langage spécifique interprété par un moteur d'exécution particulier qui réalise les invocations des services, le routage des données d'un service à un autre, et la gestion des erreurs.

Dans une composition par procédé, le flot de contrôle est explicite et le contrôle des invocations de services est externe aux services composés. Deux catégories de composition de services par procédés sont distinguées :

- l'orchestration de services qui décrit, du point de vue du service composite, les interactions entre les différentes activités : messages échangés, ordre d'invocation (séquence, parallèle, choix) ; ainsi que des opérations internes réalisées entre ces interactions (transformations de données, invocations de modules internes). La réalisation de chaque activité correspond à l'invocation d'un service concret. Le contrôle de la composition est centralisé (figure 1.2 A).
- la chorégraphie de services qui décrit la collaboration d'un ensemble de partenaires pour atteindre un but commun. La chorégraphie décrit la façon dont les services participants doivent collaborer afin de remplir le but commun : les échanges de messages, les règles auxquelles les interactions entre les différents participants sont soumises, la façon dont les différents participants se coordonnent. Le contrôle de la composition est distribué entre les participants (figure 1.2 B).

L'orchestration et la chorégraphie de services peuvent être combinées pour permettre l'intégration de systèmes d'entreprises différentes. Chaque partenaire modélise en interne, par orchestration, la réalisation des fonctionnalités rendues disponibles. La collaboration entre les différents partenaires est modélisée par chorégraphie.

Dans la suite, nous présentons deux standards d'orchestration et chorégraphie :

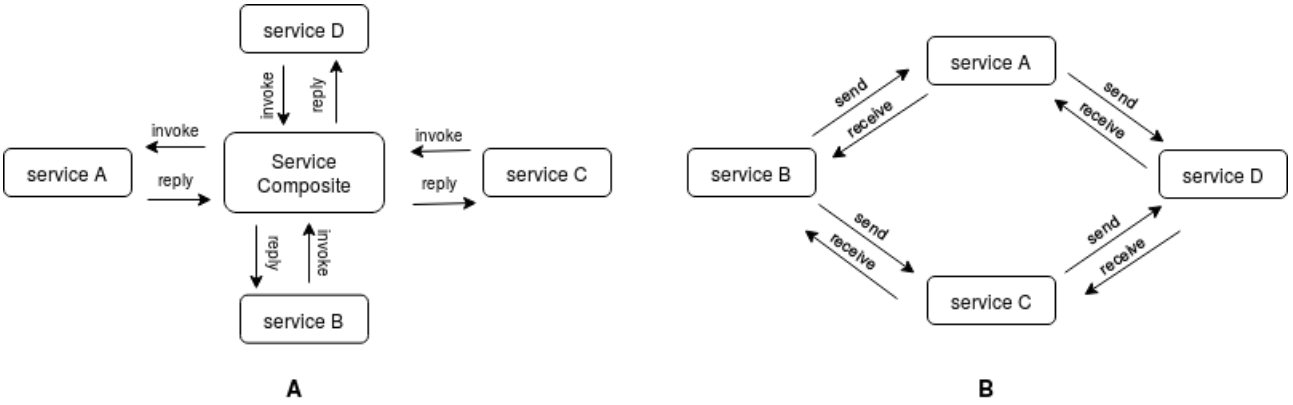

FIGURE 1.2: Orchestration vs Chorégraphie [19]

a. **WS-BPEL (Web Services Business Process Execution Language)**

WS-BPEL, ou BPEL en bref, est un langage XML pour la composition de services. Dans BPEL, le résultat de la composition est appelé *process*, les services participants sont appelés *partners* et l'échange de messages ou la transformation de résultat intermédiaire est appelé *activity*. BPEL introduit plusieurs types d'activités primitives pour 1) permettre l'interaction (*invoke*, *reply*, et *receive*); 2) attendre un peu (*wait*); 3) copier des données d'un endroit à un autre *assign*; 4) indiquer les conditions d'erreur *throw*; 5) mettre fin à l'instance de composition entière *exit*; 6) ne rien faire *empty*.

Ces activités primitives peuvent être combinées dans des activités plus complexes en utilisant des activités structurées fournies par BPEL telles que *sequence*, *while*, *foreach*, *flow*, etc.... Une construction particulière offerte par BPEL est la portée(*scope*), qui fournit un moyen de diviser un processus métier complexe en parties organisées hiérarchiquement. Le service composite est construit en XML à l'aide des éléments de construction définis précédemment.

b. **WS-CDL**

WS-CDL est un langage basé sur XML qui décrit les collaborations pair à pair des participants en définissant leur comportement observable. WS-CDL capture les interactions de service dans une perspective globale, ce qui signifie que tous les services participants sont traités de manière égale, ce qui est différent de BPEL où les interactions de service sont décrites dans une perspective de participant unique. WS-CDL prend en charge la gestion des exceptions via un type spécial d'unité de travail (nommé *Exception*), qui est associée à une chorégraphie et est activée lorsqu'une exception se produit. Un autre type spécial d'unité de travail est *Finalizer*, qui est activé lorsque sa chorégraphie associée se termine avec succès.

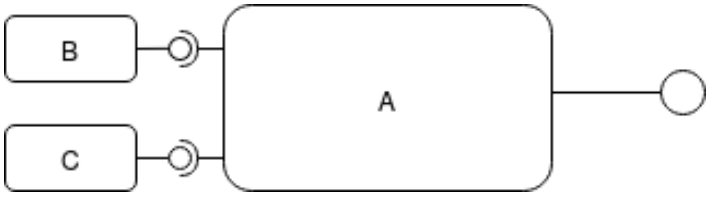

FIGURE 1.3: Composition structurelle de services [20]

1.3.2.2 Composition structurelle

Une composition structurelle consiste à décrire la structure d'une application en indiquant ses composants et les connexions entre eux. Chaque composant déclare explicitement les services qu'il fournit et ceux qu'il requiert. La composition assemble donc des composants dont les services requis par un composant correspondent aux services fournis par un autre composant. La logique de contrôle, spécifiant comment et à quel moment les opérations des services composés doivent être invoquées, est implicite et repartie entre les différents composants : le contrôle est exprimé à l'intérieur des composants. Par exemple, la figure 1.3 montre un assemblage de composants à services dont la logique de contrôle se trouve à l'intérieur du composant A.

Un exemple de technologie pour la composition structurelle de service est SCA (Services Component Architecture).

SCA [20] spécifie un modèle pour la création de composants et pour la construction d'applications. Une application SCA est un assemblage de composants, nommé composite, communiquant au travers de services (voir la Figure 1.3). Un composite SCA contient des composants (primitifs ou composites), des services, des références, des liaisons qui les connectent, et des propriétés qui peuvent être utilisées pour configurer les composants. Le modèle de composition SCA est hiérarchique et permet de lier des composants hétérogènes. Un composite SCA a donc la même vue externe qu'un composant primitif SCA. De la même façon que les composants primitifs SCA, un composite SCA précise ses services fournis et ses références de services. Les services fournis par un composite se retrouvent parmi les services fournis par ses composants internes et sont exportés à l'extérieur du composite par un mécanisme appelé promotion (promote). Le même mécanisme est utilisé pour les références de services. La promotion des références des composants internes les rend visibles à l'extérieur du composite. Ces références seront donc résolues à l'extérieur du composite. Les propriétés d'un composite peuvent être aussi promues afin de configurer les composants internes à partir des valeurs indiquées pour le composite.

A l'inverse de la composition par procédés, la composition structurelle ne permet pas facilement la réutilisation des composants puisque le contrôle est interne aux services. Par contre, elle est plus efficace puisque la communication entre services est directe, elle ne passe pas par un intermédiaire comme dans la composition par procédés. De plus, le contenu du service est réalisé par le développeur, les algorithmes et les interactions entre services peuvent être plus complexes que ceux des compositions par procédés, qui ont un langage plus restreint.

1.3.3 Paradigmes pour la composition de services

Dans cette section, on s'intéresse à la manière dont le service composition est décrit. Certains langages proposent de décrire complètement les composites: ils sont impératifs. Afin d'augmenter la flexibilité, des langages déclaratifs ont été proposés.

1.3.3.1 paradigme impératif

Les langages impératifs proposent des éléments pour décrire explicitement comment le service composite est mis sur pied. Les langages (WS-BPEL, WS-CDL, SCA etc...) présentés à la section précédente sont impératifs car ils décrivent comment les services composites sont conçus. Ces langages proposent de définir des services composites sous forme de blocs structurés rendant ainsi leur flexibilité et leur adaptation difficiles.

1.3.3.2 paradigme déclaratif

Les modèles déclaratifs se concentrent sur la spécification de ce qui doit être accompli en offrant tous les comportements possibles et en utilisant des contraintes pour éliminer le comportement que nous ne voulons pas voir se produire dans le processus,

Le paradigme des règles a été présenté comme une approche déclarative. Il présente les avantages suivants [21]:

- Flexibilité: les compositions basées sur des règles sont plus flexibles que les compositions de type BPML. Elles sont capables de poursuivre d'autres chemins d'exécution dans le cas où un chemin d'exécution particulier échoue sans redéfinir le composite et le redéployer;

- Adaptabilité : étant donné le caractère déclaratif des compositions de services basées sur des règles, elles peuvent être mises à jour pour s'adapter à des situations spécifiques, par exemple en termes de services externes ou de déploiement de plateforme;

- Ré-utilisabilité: Les règles étant isolées du contexte du processus métier, elles peuvent être réutilisées plus facilement dans d'autres contextes d'application de service;

- Sémantique formelle : les langages basés sur des règles exploitent un ensemble logique et/ou mathématique. Des approches formelles permettant de raisonner ont été proposées [22] [23] mais toutes utilisent le type de processus WS-BPEL pour leur implémentation.

1.3.3.3 Déclaratif vs Impératif : Synthèse

Un langage de composition de services est plus flexible lorsqu'il est basé sur un paradigme déclaratif plutôt que sur un paradigme impératif comme décrit dans [24].

La plupart des langages traditionnels qui ont été proposés pour spécifier la composition des services Web sont basés sur des processus, avec BPEL comme colonne vertébrale puisque tous les formalismes proposés sont traduits en BPEL pour leur exécution [25] [2] [1]. L'inconvénient de ce paradigme est que la description des services composites représentés comme des processus est centralisée et difficile à modifier au moment de l'exécution.

Pour surmonter ces difficultés, certains langages ont été proposées afin d'avoir plus de flexibilité [26] [27] [28] [29] . Ils traitent de la sémantique de la composition en offrant la capacité de décrire et de raisonner sur les services lors de l'exécution [30]. Ces langages sémantiques sont excellents dans la découverte, la sélection et la composition automatique de services. Leur flexibilité se limite à la recherche de services manquants ou à la création d'un plan de composition basé sur la requête d'un utilisateur et un système de planification prédéfini. Il est difficile d'ajouter de nouvelles exigences aux spécifications d'un service composite lorsque le système est en cours d'exécution.

Plusieurs approches déclaratives de la composition des services ont été proposées. Le travail de [31] définit les règles sous la forme de clauses $if..then$, la structure, les données et les règles de contraintes sur la base d'éléments tels que l'activité, la

condition, l'événement, le flux, le fournisseur, le rôle et le message. Les règles $if..then$ régissent la façon dont les choses doivent être faites dans la composition. Les règles $if..then$ impliquent la définition de toutes les possibilités entre les éléments de la composition. Il s'agit d'une première étape pour la définition de service composite flexible, mais elle est définie comme une extension des notations BPEL. Pour séparer les règles métier du code BPEL, Charfi et al. a suggéré un style orienté aspect (AO4BPEL) [32]. Les auteurs de [33] proposent une approche nommée FARAO.

Ils soutiennent que les règles métier peuvent être utilisées dans une composition de service sans avoir besoin d'un cadre BPEL. Cela augmente considérablement l'adaptabilité de l'orchestration. Au niveau du déploiement, un moteur de règles CA (Condition-Action) est introduit pour prendre en charge la composition de services basée sur des règles. Pour obtenir le service composite, une analyse du registre des services (contenant les WSDL) est effectuée afin d'avoir des dépendances entre les services et de créer des règles d'autorité de certification. Dans les règles CA, les règles et contraintes-métier seront ajoutées. Bien qu'utilisant les règles pour construire les services composites, cette approche a un niveau d'abstraction des règles, qui sont assez faibles (liées à WSDL). Comme les approches précédentes, FARAO se concentre sur l'orchestration au détriment de la distribution et de l'interaction.

Le tableau 1.1 récapitule les langages de composition de services présentés dans la littérature. En dimensionant notre problème, nous présentons en colonne les propriétés recherchées.

Le langage proposé dans cette thèse adopte une approche indépendante des blocs structurés tels que BPML qui favorisera et décrira un composite complètement avec des règles, en utilisant le formalisme des GAGs (Grammaire Attribuée Gardée).

L'adaptation du modèle proposé est possible au moment de l'exécution parce-que chaque règle dans le schéma d'exécution est identifiée et chargée lorsque la règle est appliquée. Étant donné que chaque espace de travail est considéré comme un homologue autonome, son propriétaire peut mettre à jour le schéma en offrant de nouvelles règles (déclaration de services) ou modifier le côté droit d'une règle (redéfinition d'un service composite).

TABLE 1.1: Synthèse des Langages de Composition de Services

	Déclaratif	Décentralisé	Flexibilité par C/S	Orienté-donnée	Formelle
WS-BPEL(T.Andrews.et.al.,2004)	×	×	×	×	×
SCA(Michael.Beisiegel.et.al.,2007)	×	×	×	×	×
OWL-S	×	×	×	×	×
Jolie (F Montesi et al.,2007)	×	×	×	×	×
WSCI (WWW.Consortium.,2002)	×	√	×	×	×
WS-CDL (N Kavantzas, 2004)	×	√	×	×	×
Choregraphy Prog. (F Montesi. et al., 2014)	×	√	×	×	×
Bart Orriëns et al. 2003	√	×	√	×	×
AO4BPEL (A Charfi et al., 2007)	√	×	√	×	×
FARAO (H Weigand et al., 2008)	√	×	√	×	×
BP-calculus (F Abouzaid et al., 2013)	×	×	×	×	√
AXML (S Abiteboul et al., 2009)	×	×	×	√	×

1.4 Approches Formelles pour la Composition de services

L'activité de composition de services met en évidence plusieurs services qui doivent interagir pour atteindre un objectif. Il faut garantir l'interaction correcte des services indépendants et communicants car de nombreuses erreurs subtiles peuvent se produire: messages non reçus, interblocages, comportements incompatibles, etc... Ces problèmes sont bien connus et récurrents dans les applications distribuées. Cependant, ils deviennent encore plus critiques dans le monde ouvert des services. Il convient donc d'utiliser des méthodes formelles (Les problèmes avec les approches les plus réalistes de la composition des services sont la vérification des services Web composés.). Le principal avantage de l'utilisation de langages et de modèles avec une sémantique claire et formelle est que cela permet d'utiliser des outils automatiques pour vérifier si un système correspond à ses besoins et fonctionne correctement. Plus précisément, des

méthodes et des outils formels peuvent être utilisés pour décider [34] : si deux services sont dans un sens précis équivalents et si un service satisfait certaines propriétés souhaitables (par exemple, la propriété selon laquelle le système n'atteindra jamais un certain état inattendu). Découvrir que la composition des services existants ne correspond pas à une spécification abstraite de ce qui est souhaité, ou qu'elle viole une propriété qui doit absolument être conservée, peut aider à corriger une conception ou à diagnostiquer des bogues dans un service existant. Très récemment, plusieurs méthodes formelles ont été proposé. La plupart avec une sémantique basée sur des systèmes de transition (automates [35], réseaux de Petri [36, 37], algèbres de processus [38]) pour garantir des compositions de services. Ces formalismes et modèles ont fait leurs preuves mathématiquement. Dans la suite, nous présentons un aperçu de certaines de ces approches.

1.4.1 Automates

Un automate [34, 39] est un modèle mathématique qui représente le comportement des systèmes à l'aide d'un nombre discret d'entrées et d'un nombre discret de sorties.

Les automates ou systèmes de transition étiquetés sont un modèle bien connu qui sous-tend les spécifications formelles des systèmes. Un automate se compose d'un ensemble d'états, d'un ensemble d'actions, d'un ensemble de transitions étiquetées entre états et d'un ensemble d'états initiaux. Les étiquettes représentent des actions et l'étiquette d'une transition indique l'action provoquant la transition d'un état à un autre. La manière intuitive dont un automate peut modéliser le comportement d'un système a conduit à une variété de modèles de spécification basés sur les automates. Leur base formelle permet une prise en charge automatique des outils et, par conséquent, des modèles basés sur des automates sont de plus en plus utilisés pour décrire, composer et vérifier formellement les compositions de services. Ci-dessous suivent quelques approches exemplaires.

Les modèles basés sur des automates sont de plus en plus utilisés pour décrire formellement, composer et vérifier les services Web. Ci-dessous quelques idées de modélisation de la composition de services à l'aide des automates.

Dans [40], les auteurs introduisent un cadre pour analyser et vérifier les propriétés des compositions de services Web des processus BPEL qui communiquent via des messages XML asynchrones. Leur framework traduit d'abord les processus BPEL en un type

particulier d'automates dont chaque transition est équipée d'une garde sous la forme d'une expression XPath [41], après quoi ces automates gardés sont traduits en Promela, le langage d'entrée du vérificateur de modèle SPIN [42]. Enfin, SPIN peut être utilisé pour vérifier si les compositions de services Web satisferont à certaines propriétés LTL.

Les auteurs de [43] montrent comment nous pouvons traduire les services Web décrits par WS-CDL en automates temporisés, et plus spécifiquement. Leur point de départ est la description des services Web écrite en WSBPEL - WSCDL (langages de description basés sur XML). Ces descriptions sont ensuite automatiquement traduites en automates temporisés, puis ils utilisent un outil bien connu qui prend en charge ce formalisme (UPPAAL [44]) pour simuler et analyser le comportement du système. .

[45] propose une approche pour la modélisation et l'analyse des propriétés temporelles des compositions de services Web définies comme un ensemble de processus BPEL4WS. Un formalisme, appelé Web Service Timed State Transition Systems (WSTTS) est présenté pour mettre en evidence le temps des services Web composites. Les auteurs exploitent aussi la logique temporelle pour décrire des hypothèses et des exigences temporelles complexes sur le comportement du système. Enfin ils fournissent des outils et des techniques pour vérifier les modèles BPEL4WS par rapport aux exigences liées au temps.

1.4.2 Les réseaux de Pétri

Les réseaux de Petri ont été introduits dans [46] comme cadre pour modéliser des systèmes concurrents. Leur principale attraction est la manière naturelle dont de nombreux aspects fondamentaux des systèmes concurrents sont identifiés à la fois mathématiquement et conceptuellement. Cela a grandement contribué au développement d'une riche théorie des systèmes concurrents basés sur les réseaux de Petri [47].

Les réseaux de Petri sont très populaires dans le BPM et les domaines connexes en raison de la variété des flux de contrôle de processus qu'ils peuvent capturer [48, 49]. En particulier, la technique d'élimination des voies mortes utilisée dans BPEL pour contourner les activités dont les conditions préalables ne sont pas remplies peut être facilement modélisée dans les réseaux de Petri. Dans [50], il est montré comment mapper toutes les constructions de flux de contrôle BPEL en réseaux de Petri étiquetés (incluant ainsi les flux de contrôle pour la gestion des exceptions et la compensation). Cette sortie peut ensuite être utilisée pour vérifier les processus BPEL au moyen des

outils open source BPEL2PNML et WofBPEL. Nous présentons par la suite quelques approches utilisant le réseaux de Petri.

Dans [37], les auteurs introduisent une algèbre basée sur le réseau de Petri pour modéliser les flux de contrôle. Ils décrivent la syntaxe et la sémantique informelle des opérateurs d'algèbre de service. Les concepts ont été choisis pour permettre des combinaisons de services Web communes et avancées. Nous avons par exemple ϵ qui représente un service qui n'exécute pas d'action et des concepts pour représenter l'exécution séquentielle ordonnée ou non, le choix, l'exécution parallèle etc...

Dans [51], un cadre de conception et de vérification basé sur Petri-net pour la composition des services Web est proposé, qui peut être utilisé pour visualiser, créer et vérifier les processus BPEL existants.

Dans [52], un langage de description architecturale basé sur $Petri-net$ (dans lequel des systèmes orientés service Web peuvent être modélisés) est présenté et une petite étude de cas est présentée. Afin de gérer des applications réelles et d'éliminer les erreurs de traduction manuelle, les auteurs développent actuellement un moteur de traduction automatique de WSDL vers leur langage.

Dans [53] une sémantique Petri-net complète et formelle pour BPEL est présentée, incluant ainsi la gestion des exceptions et les compensations. De plus, les auteurs présentent leur analyseur BPEL2PN qui peut automatiquement transformer les processus BPEL en réseaux de Petri. En conséquence, une variété d'outils de vérification Petri-net sont applicables pour analyser automatiquement les processus BPEL.

Les cadres et outils décrits ci-dessus présentent l'avantage de permettre de simuler et de vérifier le comportement de son modèle au moment de la conception, permettant ainsi la détection et la correction des erreurs le plus tôt possible. En tant que telles, ces approches contribuent à accroître la fiabilité des applications de services Web.

1.4.3 Algèbre de Processus

Les algèbres de processus sont un moyen populaire pour décrire et raisonner sur les comportements de processus. Leur fondement sémantique sous-jacent est basé sur des systèmes de transition étiquetés, c'est-à-dire des automates. Ils constituent des cadres adéquats pour l'écriture et la vérification de systèmes réactifs. De nombreuses variantes ont été définies et le domaine est accompagné d'une riche littérature. Les algèbres de

processus les plus connues sont le calcul de Milner des systèmes de communication (CCS [54]), le calcul de Hoare des processus séquentiels (CSP [55]), l'algèbre des processus de communication (ACP [56]) de Bergstra et Klop, et le Langage des systèmes ordonnés temporels (LOTOS [57]) normalisé par l'ISO.

Dans ces modèles, tout terme bien formé dénote un processus. L'abstraction fondamentale est qu'on ne s'intéresse au comportement d'un processus qu'à travers un certain nombre de points d'interaction également appelés *canaux*. La synchronisation et la communication sont alors exprimées par des lois de composition internes et externes. Aucune hypothèse n'est faite sur les vitesses d'exécution des différents processus.

La syntaxe du pi-calcul *polyadique* synchrone du premier ordre telle que définie dans [58] est la suivante :

$$P ::= 0 \mid \rho.P \mid P|P \mid P + P \mid [x = y]P \mid [x \neq y]P \mid v\ \overline{\overline{u}}\ P \mid A(\overline{\overline{u}})$$

où le préfixe ρ ou l'*action* est défini par :

$$\rho ::= \tau \mid \alpha(\overline{\overline{u}}) \mid \overline{\alpha}\langle\overline{\overline{u}}\rangle$$

et où l'abstraction de processus doit faire l'object de déclarations de la forme :

$$A(\overline{\overline{u}}) \stackrel{def}{=} P$$

$\overline{\overline{u}}$ signifie une liste, éventuellement vide, de variables également appelées *noms*.

De façon générale:

- le processus 0 ne fait rien, c'est-à-dire qu'il n'agit pas avec son environnement. Il dénote de la terminaison normale d'un processus.
- le processus $\rho.P$ est une loi de composition externe impliquant une action ρ et un processus P signifiant exécute l'action ρ puis se comporte comme P. ρ peut

être soit une action silencieuse τ soit une interaction avec l'environnement : $\alpha(\overline{\overline{u}})$ ou $\overline{\alpha}\langle\overline{\overline{u}}\rangle$ signifiant respectivement la reception et l'emission d'une liste de variables sur le canal α.

- $P|Q$ signifie que les processus P et Q s'exécutent en parallèle.
- le processus $P+Q$ dénote le choix indéterministe entre les processus P et Q. Ils ne peuvent pas s'exécuter tous les deux. De même ils n'interagissent pas.
- l'opérateur de *matching* permet de réaliser l'exécution conditionnelle d'un processus. Si $x = y$, alors $[x = y]P$ se comporte comme P sinon il se comporte comme 0. l'opérateur de *mismatching* a une sémantique symétrique c'est-à-dire si $x \neq y$ alors $[x \neq y]P$ se comporte comme P sinon il se comporte comme 0.
- $v\ \overline{\overline{u}}\ P$ dénote un processus dans lequel les variables de la liste $\overline{\overline{u}}$ ont une portée limitée à P, c'est-à-dire qu'elles sont inconnues dans les autres processus : on parle de variables *privées*. P ne peut pas communiquer dans l'environnement en utilisant ces canaux. Si l'un d'entre eux est émis sur un autre canal, sa portée est étendue au processus récepteur. Ainsi il est connu uniquement par ces deux derniers. Les éventuels conflits d'homonymie entre des variables privées exportées et des variables locales sont résolus par le biais d'alpha conversions.
- La définition d'une abstraction de processus $A(\overline{\overline{u}}) \stackrel{def}{=} P$ lie un identifiant de processus A et une liste de variables $\overline{\overline{u}}$ à un comportement de processus P

Comme les réseaux de Petri, les algèbres de processus sont des formalismes précis et bien étudiés qui permettent la vérification automatique de certaines propriétés. De même, ils fournissent une théorie riche sur l'analyse de *bisimulation*, c'est-à-dire que l'on peut établir si deux processus ont des comportements équivalents. De telles analyses sont utiles pour établir si un service peut remplacer un autre service dans une composition [59] ou pour vérifier la redondance d'un service. Nous n'insisterons pas sur la *bisimulation* puisque dans cette thèse nous ne nous intéressons pas à l'aspect sémantique de la composition de services qui conduit à la recherche des services équivalents lorsqu'un service est in-opérationnel ou défectueux. Dans la suite de cette section, nous présentons des approches utilisant l'algèbre de processus pour la composition de services.

Dans [60], les auteurs préconisent l'utilisation d'algèbres de processus pour décrire, composer et vérifier les services Web, avec un accent particulier sur leurs interactions.

À cette fin, ils présentent une étude de cas dans laquelle ils utilisent CCS pour spécifier et composer des services Web en tant que processus, et le Concurrency Workbench [61] pour valider des propriétés telles que la composition correcte des services Web. Pour appliquer cette approche à des applications réelles, il faut utiliser des calculs plus avancés que CCS (par exemple le pi-calcul) afin de prendre également en compte des questions telles que l'échange de données lors des interactions de services Web et les compositions dynamiques. En fait, dans [62] une cartographie bidirectionnelle est définie entre BPEL et l'algèbre de processus plus expressive LOTOS. Un avantage de cette traduction est qu'elle inclut les compensations et la gestion des exceptions. En tant que tel, il permet la vérification des propriétés temporelles avec la boîte à outils de vérification de modèle CADP [63].

[22] propose le BP-calcul qui est une extension du pi-calcul afin de vérifier qu'une composition WS-BPEL s'exécute correctement. En effet, il facilite la génération automatique de code BPEL vérifié, en définissant l'équivalence et la logique spécifiques afin de vérifier les implémentations BPEL à travers leur spécification formelle exprimée dans ce calcul.

Les outils algébriques de processus sont également bien adaptés pour améliorer la fiabilité du développement de services Web en simulant et en vérifiant le comportement de son modèle au moment de la conception.

1.4.4 Synthèse

1.4.4.1 Toujours WS-BPEL

La quasi-totalité des approches formelles parcourues pour la composition de services utilise WS-BPEL comme langage cible. WS-BPEL étant un langage impératif tel que présenté à la section 1.3.3, il convient donc que ces solutions souffrent des lacunes inhérentes à ce paradigme.

1.4.4.2 Choix de l'algèbre de processus

Dans l'algèbre de processus (pi-calcul), les processus exécutent des actions qui peuvent prendre trois formes: l'envoi d'un message sur un canal, la réception d'un message sur un canal et des actions silencieuses (τ), dont les détails ne sont pas observables.

Les actions d'envoi et de réception sont appelées actions de synchronisation, car la communication est mise en évidence lorsque les processus se synchronisent. La notion de transition représente l'exécution d'une expression de processus. Intuitivement, la relation de transition indique comment effectuer une étape de l'exécution du processus. Notez qu'étant donné qu'il peut exister de nombreuses façons d'exécuter un processus, la transition est fondamentalement non déterministe.

La transition d'un processus A vers un processus B en réalisant une action α est indiqué par A $\xrightarrow{\alpha}$ B. L'action α est l'observation de la transition.

La description des services ainsi définis et leurs comportements reposent sur des primitives d'instanciation, de raffinement et d'échange de messages que l'on retrouve également dans des langages plus abstraits comme les algèbres de processus. Cela nous permet de décrire la structure d'un service et de présenter formellement comment un service est défini et comment il interagit dans son environnement. Dans ce qui suit, nous décrivons formellement la syntaxe et la sémantique d'un langage de composition de service.

Cette sémantique est décrite en utilisant le formalisme pi-calcul [12]. Le choix de ce formalisme est motivé par la répartition des pairs sur un réseau et leur interaction, qui se fait via des ports dynamiques qui sont créés lors de l'exécution. Un service ou un pair est considéré comme un processus de Pi-calcul. Un pair reçoit et envoie les messages. Dans cette logique, un système se compose d'un ensemble de processus (pairs ou services) qui communiquent ensemble. Lors de leurs interactions, des canaux asynchrones peuvent être créés et utilisés pour échanger des messages. De plus, les canaux ainsi créés sont inclus dans les messages. Le fait que les canaux soient créés dynamiquement et envoyés au pair dans les messages, nous a conduit à choisir le Pi-calcul pour notre modélisation.

1.5 Conclusion

Dans ce chapitre nous avons présenté le cycle de vie de la composition. Il comprend généralement la définition du composite, la sélection des services, le déploiement et l'exécution du composite. Nous avons également présenté les approches standards pour la composition de services à savoir la composition par procédé et la composition structurelle. La majorité des langages qui en découle est basée sur un paradigme impératif c'est à dire que ces langages décrivent *comment* le service composition est

conçu. Ils ne favorisent pas la flexibilité. A côté de ces langages impératifs, les langages déclaratifs ont été définis afin de définir des processus plus flexibles. Certains de ces langages s'appuient toujours sur BPEL ce qui ne favorise pas l'ajout de nouveaux services (prise en compte des besoins utilisateurs) une fois défini et déployé. Nous voulons proposer une approche indépendante des blocs structurés tels que BPML qui permettra la définition des services à la volée sous forme de règles. Ce langage sera purement déclaratif présentant des propriétés de flexibilité, d'adaptabilité, de réutilisabilité et d'abstraction.

Étant donné que les composites seront définis à la volée, il convient de raisonner sur leurs propriétés. Notamment vérifier qu'un composite est correctement spécifié. C'est dans cette optique que nous avons présenté les approches formelles utilisées pour raisonner lors de la composition de services. Elles permettent de garantir l'interaction correcte des services communicants. Généralement ce sont des systèmes de transitions tels que les automates, les réseaux de Pétri ou les algèbres de processus. Dans cet écosystème, nous avons choisi le pi-calcul qui est une algèbre de processus afin de définir les interactions dynamiques entre les services composites communicants. Elle permet de simuler les ports dynamiquement lors de l'exécution des processus par exemple. Dans le prochain chapitre, nous présentons la syntaxe ainsi que la sémantique opérationnelle de notre langage déclaratif.

.

Chapter 2

Un langage déclaratif pour la composition de services : GSLang

Sommaire

2.1 Introduction

Ce chapitre présente le langage déclaratif proposé pour la composition de services. Ce langage est nommé GSLang pour *GAG Service Language* car nous exploitons la structure des GAGs qui est un modèle purement déclaratif, centré sur les *Artéfacts* et orienté-utilisateurs. Nous présentons d'abord les propriétés des approches déclaratives, ensuite nous décrivons les concepts de bases, la syntaxe et la sémantique opérationnelle de GSLang. Le chapitre se termine par des démonstrations des propriétés de GSLang.

2.2 Propriétés générales d'une approche déclarative

Le paradigme des règles est présenté comme une approche déclarative. Il présente les avantages suivants [21, 33, 64]:

- Flexibilité: les compositions basées sur les règles sont plus flexibles que les compositions de type BPML;
- Adaptabilité: étant donné la nature déclarative des compositions de services basées sur des règles, elles peuvent être mises à jour pour s'adapter à des situations spécifiques;
- Réutilisabilité: Les règles étant isolées du contexte du processus métier, elles peuvent être réutilisées plus facilement dans d'autres contextes d'application de service;
- Sémantique formelle: les langages basés sur les règles exploitent la logique et/ou les mathématiques. Les approches formelles permettant de raisonner ont été proposées au chapitre précédent mais tous utilisent le langage BPEL pour leurs implémentations.

Dans ce chapitre, nous adoptons une approche indépendante de la structuration en blocs telle que prônée par BPML. Nous décrivons un service composite complètement avec des règles, en utilisant le formalisme des GAGs [12] pour spécifier intentionnellement la composition des services. La définition intentionnelle de la composition des services les rend abstraits, ce qui augmente la flexibilité car le lien entre les intentions et la mise en œuvre se fait lorsqu'une règle est déclenchée. Toute mise à jour est possible sur une règle et elle sera prise en compte la prochaine fois que la règle sera déclenchée.

L'avantage de cette propriété est qu'elle permet de conformer les règles aux besoins de l'utilisateur même au moment de l'exécution en adaptant les règles aux nouvelles exigences. De plus, l'utilisation de règles dans un langage de composition de services apporte la sémantique intuitive formelle, la flexibilité, l'adaptabilité et la réutilisabilité.

2.3 Formalisme déclaratif pour la composition des services

L'un des principaux défis de l'ingénierie logicielle est de faire face aux changements (sociaux, besoins des utilisateurs, etc.). Concernant l'aspect de la composition du service, on parle de flexibilité par le changement, ce qui signifie modifier la structure d'un service composite même à l'exécution [24]. Nous proposons dans cette section un formalisme de spécification de service qui utilise des productions d'une grammaire hors contexte [65]. Cette spécification dite intentionnelle parce qu'elle reste abstraite, est constituée à droite (RHS) de la description des services concourant à résoudre le service à gauche (LHS). Dans notre approche, le schéma de composition reste aussi abstrait que possible, car même pendant l'exécution, seule la règle concernée est déclenchée et instanciée.

2.3.1 Définition de base

Un service est défini dans le contexte d'une activité. Une activité est un tuple
A = (id, S, C) où

- **Id** est un identifiant unique de l'activité (par exemple son nom);
- **S** est le schéma d'exécution qui décrit les services $\{s_0, s_1 ... s_n\}$ à exécuter pour résoudre l'activité;
- **C** est le contexte indiquant les informations de l'environnement dans lequel l'activité est exécutée. Ces informations peuvent contribuer à la modification du schéma d'exécution d'une activité. $C = (I, O)$ où I correspond aux entrées provenant de l'environnement et O aux sorties renvoyées à l'environnement à la fin de l'exécution de l'activité.

Chaque service s ∈ S peut être défini sous l'une des formes suivantes:

$$s(c_0...c_n)\langle r_0...r_m\rangle \rightarrow s_1(i_{1i}...i_{1u})\langle o_{1i}...o_{1k}\rangle \ ... \ s_j(i_{j1}...i_{jo})\langle o_{j1}...o_{jh}\rangle \qquad (1)$$

$$s(c_0...c_n)\langle r_0...r_m\rangle \rightarrow \qquad (2)$$

À cet égard, nous distinguons les services composites si le côté droit de la règle n'est pas vide et les services élémentaires dans le cas contraire. Un service composite dépend des services qui apparaissent sur le côté droit pour sa réalisation en d'autres termes, les services RHS agrégent le service qui apparaît au LHS. Un service élémentaire permet d'accéder à une application Internet via un protocole bien connu tel que SOAP ou le style REST. Ce concept de service contribuera à assurer l'interopérabilité entre les pairs. Pour chaque type de règle, une fonction de correspondances qui définit les règles sémantiques entre les paramètres peut être associée. Soit A une activité hébergée par un pair h. Lorsque A est définie comme:

$$s(c_0...c_n)\langle r_0...r_m\rangle \rightarrow s_1(i_{1i}...i_{1u})\langle o_{1i}...o_{1k}\rangle \ ... \ s_j(i_{j1}...i_{jo})\langle o_{j1}...o_{jh}\rangle$$

où s_1, .., s_j composent le service s. Les c_k sont les paramètres d'entrée de s et les r_k sont les paramètres de sortie. Les i_{kj} sont les paramètres en entrée de RHS et les o_{kj} sont les paramètres en sortie.

Les règles sémantiques sont données par les dépendances entre attributs qui permettent d'appliquer les substitutions

$$\begin{cases} i_{kl} = f(i_{ij}, o_{mn}, c_j) \ avec \ k \neq \ i, \ l \ \neq \ j \\ r_k = f(o_{mn}) \end{cases}$$

où les entrées des RHS dépendent des entrées des RHS d'autres services, les sorties des RHS et des entrées de LHS de s. Les sorties de LHS sont fonctions des sorties de RHS. f matérialise cette dépendance.

Exemple:

$travelService(username, date, destination)\langle numVol, hotelAdress\rangle \rightarrow$
. $flightBooking(username, date, destination)\langle numVol\rangle$
. $hotelReservation(username, date)\ \langle hotelAdress\rangle$

$numVol$ de $travelService$ correspond à $numVol$ de $flightBooking$.
D'autres dépendances similaires peuvent être considérés sur cet exemple pour $travelService$.

Les règles sémantiques s'expriment également par les conditions sur les paramètres d'entrée et de sortie qui servent de gardes et de post-conditions.

Cette définition dite intentionnelle se matérialise lors de son instanciation. L'instanciation des services a lieu en fonction de la disponibilité des données (paramètres). Lorsque le schéma de composition d'une activité est mis à jour, la modification sera appliquée la prochaine fois que la règle associée sera appliquée. Les règles sémantiques peuvent impliquer plusieurs variables qui appartiennent à des pairs différents. A ce moment, un canal dynamique privé est ouvert pour l'interaction entre les pairs. Ce sont les raisons pour lesquelles le pi-calcul est choisi dans la section suivante pour spécifier le langage de composition. En fait, on remarque que le pi-calcul fournit des primitives pour décrire et analyser l'infrastructure distribuée globale, en se concentrant sur la mutation de processus, les interactions de processus via des canaux dynamiques, la communication par canal privé [38]. Dans ce qui suit, nous décrivons formellement la syntaxe et la sémantique de GSLang.

2.3.2 Une syntaxe formelle et sémantique d'un langage de composition de service: GSLang

Le langage est nommé GSLang et a été présenté dans nos articles [66, 67]. Cette section présente une formalisation complète de GSLang de la spécification des services jusqu'à leurs opérationnalisations. Les éléments du langage GSLang sont:

- les variables et les termes
- les services et instances de services
- les actions

- les messages

2.3.2.1 Variables et termes

Une **variable** est une lettre; elle peut contenir une valeur. Les **termes** sont des valeurs, des variables, des variables définies (affectation), des fonctions sur les termes ou des expressions booléennes.

Dans la suite, nous définissons $\overline{\overline{x}}$ pour le tuple (x_1,... x_n).

$$\begin{aligned} t \;::=\; & x(variable) \\ \mid\; & u(value) \\ \mid\; & x_r(defined\ variable) \\ \mid\; & f(t_0 ... t_n) \end{aligned}$$

À la syntaxe de base du pi-calcul, nous ajoutons des expressions booléennes pour vérifier l'activation et la validation d'un service. Une **expression booléenne** lorsqu'elle est spécifiée et évaluée donne une valeur booléenne. En d'autres termes, c'est une expression logique sur les variables. Les expressions booléennes servent de gardes lors de l'activation d'un service. Dans ce cas, ce sont des conditions préalables; Elles contrôlent le déclenchement d'un service. À la fin de l'exécution d'un service, il peut y avoir des conditions à remplir. Ce sont des post-conditions.

$$\begin{aligned} e_b \;::=\; & true \mid false \mid t_j \leq t_i \\ & \mid t_i = t_j \mid e_b \mid e_b \wedge e_b \mid e_b \vee e_b \end{aligned}$$

L'**affectation** attribue une valeur à une variable.

$$x_r \;::=\; \epsilon \mid x_r\ (x \leftarrow t)(attribution\ de\ valeur)$$

Un paramètre est une sortie ou une variable d'entrée liée à un service. La portée d'une variable est liée au service associé. Le contexte d'activité est caractérisé par des variables d'entrée et des variables de sortie. Ces variables sont libres et uniques dans tout le système. On note qu'une variable est définie ou résolue lorsqu'une valeur lui est affectée ($x \leftarrow u$). Une variable est définie une fois et peut être reutilisée indéfiniment.

2.3.2.2 Service et Instance de Service

Un **service** est défini par un identifiant unique, des variables de sortie (paramètres de sortie), des variables d'entrée (paramètres d'entrée), des post-conditions, des gardes et un emplacement. Un service peut dépendre d'autres services.

$$\begin{aligned} S \ ::= \ & id(\ \overline{\overline{x}}, \overline{\mathrm{e_b}^x})\ \langle \overline{\overline{y}}, \overline{\mathrm{e_b}^y} \rangle\ [\alpha]\ \mid \\ & id(\ \overline{\overline{x}}, \overline{\mathrm{e_b}^x})\ \langle \overline{\overline{y}}, \overline{\mathrm{e_b}^y} \rangle\ [\alpha] \rightarrow \mathrm{S_1 ... S_n} \end{aligned}$$

un service est élémentaire ou composite. Il est caractérisé par des paramètres en sortie $\overline{\overline{y}}$, paramètres en entrée $\overline{\overline{x}}$, éventuellement les effets sur les paramètres en sortie $\overline{\mathrm{e_b}^y}$, éventuellement des pré-conditions sur les paramètres en entrée $\overline{\mathrm{e_b}^x}$ et un identifiant (son nom). Il peut être composé d'autres services $\mathrm{S_1 ... S_n}$. α représente l'emplacement du service. Il convient de noter que α peut être inutile pour les services du côté RHS s'ils sont mis en œuvre dans le même espace utilisateur que les services du côté LHS.

Pour des raisons de lisibilité, nous préférons la notation précédente. En notation pi-calcul, cela correspond à:

$$S \ ::= \ [\overline{e_b}^x]\ \alpha\ (id, \overline{\overline{x}}, p)\ .\ [\overline{e_b}^y]\ p!\overline{\overline{y}}\ \mid \qquad . \qquad (E1)$$

$$[\overline{e_b}^x]\ \alpha\ (id, \overline{\overline{x}}, p)\ .(vp_1\ \alpha\ \langle id_1, \overline{\overline{x}}_1, p_1 \rangle\ \mid ... \mid\ S_i ...\ \mid\ vp_n\ \alpha\ \langle id_n, \overline{\overline{x}}_n, p_n \rangle)\ .\ [\overline{e_b}^y]\ p!\overline{\overline{y}} \qquad (E2)$$

Le service S attend $\overline{\overline{x}}$ comme paramètre, s'exécute et retourne $\overline{\overline{y}}$. Il reçoit les données sur le port public α du pair où il est hébergé. Lorsque le RHS est présent ($E2$), il appelle les services qu'il contient pour construire y. Les services sur le RHS peuvent être exécutés en parallèle si les données sont indépendantes les unes des autres ou en séquence s'il y a dépendance d'où l'opérateur parallèle (|). Une fois un appel de service effectué, nous obtenons les instances de service. Ces notations sont également utilisées pour décrire les instances.

L'**instance de service** ou **Artifact** provient de l'instanciation d'un service. Elle permet de suivre l'exécution du service.

$$
\begin{aligned}
I \quad ::= \quad & id(\ \overline{\overline{x}}, \overline{\mathrm{e_b}}^{x})\ \langle \overline{\overline{y}}, \overline{\mathrm{e_b}}^{y} \rangle\ [\alpha]\ \mid \\
& id(\ \overline{\overline{\mathrm{x_r}}}, \overline{\mathrm{e_b}}^{x})\ \langle \overline{\overline{y}}, \overline{\mathrm{e_b}}^{y} \rangle\ [\alpha]\ \mid \\
& id(\ \overline{\overline{\mathrm{x_r}}}, \overline{\mathrm{e_b}}^{x})\ \langle \overline{\overline{\mathrm{y_r}}}, \overline{\mathrm{e_b}}^{y} \rangle\ [\alpha]\ \mid \\
& id(\ \overline{\overline{\mathrm{x_r}}}, \overline{\mathrm{e_b}}^{x})\ \langle \overline{\overline{y}}, \overline{\mathrm{e_b}}^{y} \rangle\ [\alpha] \rightarrow \mathrm{I_1 ... I_n}\ \mid \\
& id(\ \overline{\overline{\mathrm{x_r}}}, \overline{\mathrm{e_b}}^{x})\ \langle \overline{\overline{\mathrm{y_r}}}, \overline{\mathrm{e_b}}^{y} \rangle\ [\alpha] \rightarrow \mathrm{I_1 ... I_n}
\end{aligned}
$$

Une instance de service a plusieurs configurations:

- les paramètres en entrée et en sortie ne sont pas encore résolus;
- paramètres en entrée résolus et paramètres en sortie non encore résolus;
- paramètres en entrée et en sortie résolus.

Chaque élément qui apparaît à droite lorsqu'il existe peut avoir l'une des trois configurations précédentes. Les paramètres des instances de service sont résolus lors de leur exécution. Les explications detaillées concernant cet aspect sont fournies dans la section 2.4.7.

2.3.2.3 Action

Une **action** est un message d'envoi, un message de réception ou une interprétation silencieuse des instances de service.

$$
\begin{aligned}
act \quad ::= \quad & vp\ \overline{\alpha}\ \langle M, p \rangle \\
\mid \quad & \alpha(M, p) \\
\mid \quad & p(M) \\
\mid \quad & \overline{p}\ \langle M \rangle \\
\mid \quad & r
\end{aligned}
$$

Les **actions** définissent la communication à effectuer entre les espaces d'exécution des services:

- $vp\,\overline{\alpha}\,\langle M, p\rangle$ permet, tout d'abord, la création d'une variable représentant le port privé (p), puis envoyer un message M et p vers le port public α de l'espace distant;
- $\alpha(M, p)$ recevoir un message M et une variable p sur le port public α;
- $p(M)$ recevoir un message M sur un port privé p ;
- $\overline{p}\,\langle M\rangle$ envoi d'un message M sur un port privé p;
- r action silencieuse qui consiste à interpréter les paramètres des services à l'aide de règles sémantiques. Pas de communication avec l'environnement.

2.3.2.4 Message

Les **messages** sont des variables échangées sur le réseau. Ils contiennent des variables globales (variables de contexte). Ils sont composés de variables définies et/ou de variables non définies. Il existe deux types de messages: les messages de type requête (variables en entrée définies et variables en sortie non définies) et les messages de type réponse (variables en entrée définies et variables en sortie définies).

$$M ::= \ \overline{\overline{x}}_r\ \overline{\overline{y}}\ id\ \ (requete) \mid \overline{\overline{x}}_r\ \overline{\overline{y}}_r\ \ (reponse)$$

L'**envoi du message** (requête) comprend 3 parties: entrées résolues $\overline{\overline{\mathrm{x}}}_{\mathrm{r}}$, sorties à résoudre $\overline{\overline{\mathrm{y}}}$ et l'identifiant du service auquel la demande est destinée. La **réponse** se compose de 2 parties: entrées résolues $\overline{\overline{\mathrm{x}}}_{\mathrm{r}}$ et sorties résolues $\overline{\overline{\mathrm{y}}}_{\mathrm{r}}$. Comme nous le verrons dans la section suivante, la réponse est transmise via un port privé créé lors de la requête, d'où l'absence de l'identifiant de service.

2.4 Description comportementale

Dans cette section, nous présentons la sémantique opérationnelle de GSLang. Elle décrit les mécanismes de résolution d'un service et est divisée en plusieurs opérations fondamentales:

- l'instanciation d'un service;

- l'envoi d'un message;
- la réception d'un message;
- le raffinement d'une instance de service;
- le choix local d'un service.

L'espace d'exécution ou le **pair**(Σ) heberge des services, des instances de services et se caractérise par son emplacement. Notons par $\mathrm{I_s}$ l'ensemble des instances de services, $\mathrm{S_s}$ l'ensemble des services et α son adresse (port principal ou emplacement). Ainsi, l'espace d'exécution est caractérisé par $\Sigma = (\mathrm{S_s},\ \mathrm{I_s},\ \alpha)$. $fn(e)$ est l'ensemble des variables liées à l'entité e.

2.4.1 Instanciation

L'instanciation est l'opération qui permet de créer les instances de services c'est-à-dire les services en cours d'exécution. Nous avons deux règles pour le faire selon que le service à instancier est simple (règle C_1) ou composite (règle C_2).

$$\frac{\Sigma' = \Sigma\ U\ I,\ p \notin fn(\Sigma)}{\Sigma : S = id(\overline{\overline{x}})\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{\alpha(M,p)} \Sigma' : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]} \qquad (C_1)$$

$$\frac{\Sigma' = \Sigma\ U\ I,\ p \notin fn(\Sigma)}{\Sigma : S = id(\overline{\overline{x}})\langle\overline{\overline{y}}\rangle[\alpha]\ \ \rightarrow \mathrm{S_1}...\mathrm{S_n} \xrightarrow{\alpha(M,p)} \Sigma' : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]\ \ \rightarrow \mathrm{I_1}...\mathrm{I_n}} \qquad (C_2)$$

Lorsque Σ reçoit sur son port public α le message nommé M et la variable nommée p, il cherche le service correspondant nommé S et crée l'instance I avec les paramètres en entrée définis $\overline{\overline{x}}_r$ et les paramètres en sortie $\overline{\overline{y}}$ non encore définis. I est ajouté à Σ qui devient Σ'. Si aucun service n'est trouvé, l'opération n'est pas appliqué.

C_2 est la version étendue de C_1 pour les services composite c'est-à-dire ayant une partie droite.

2.4.2 Requête

Une instance de service I envoie une requête M sur le port public α d'un espace distant. Cette opération ne change pas l'état de l'espace d'exécution émettrice; M est construit à partir des paramètres de l'instance à concrétiser. Ce dernier a l'une des configurations présentées dans la formule.

$$\frac{I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \; or \quad I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \mathrm{I}_1...\mathrm{I}_\mathrm{n} \; orI = I_0 \rightarrow I_1...id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]...I_n}{\Sigma : I \xrightarrow{vp\ \overline{\alpha}\ \langle M,p\rangle} \Sigma : I} \quad (Req)$$

2.4.3 Réponse

La réponse est le fait d'envoyer un message sur un port privé p précédemment reçu lors d'une requête.

$$\frac{}{\Sigma : I \xrightarrow{p\ \langle M\rangle} \Sigma : I} \qquad (Resp)$$

où M=$\overline{\overline{x}}_r \;\; \overline{\overline{y}}_r$

2.4.4 Raffinement

Le raffinement d'une instance de service est l'opération qui permet de matérialiser les parties non encore définies. L'action est soit silence ou la reception d'une réponse.

2.4.4.1 L'appel d'un service primitif

Une instance de service I dans l'espace Σ lorsqu'il est primitif c'est-à-dire à la forme $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]$ peut faire appel à un service externe via REST ou être manuel. Ce dernier est interprété comme une action silence dans l'espace Σ et permet de définir les sorties de I. Cette règle est nommé R_1

$$\frac{}{\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{r} \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]} \qquad (R_1)$$

2.4.4.2 Recevoir une réponse pour un service primitif

Lorsqu'une instance de service I dans l'espace Σ à la forme $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]$ reçoit un message M sur un port privé p, les sorties de I seront définies. Cette règle est nommée R_2

$$\frac{}{\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{p(M)} \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]} \qquad (R_2)$$

2.4.4.3 Définition des paramètres d'une instance à partir des paramètres déjà définis

Lorsque dans une instance, certains paramètres en partie droite ont déjà été définis et que dans l'instance, il y a des paramètres qui correspondent à ceux déjà définis, alors ces derniers seront eux aussi définis. Nous exprimons cela dans les règles R_3, R_4 et R_6.

$$\frac{\overline{\overline{x}}' \subseteq \overline{\overline{x}}}{\begin{array}{ll}\Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow & \Sigma : I = id_0(\overline{x}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \\ \quad I_1...id_i(\overline{\overline{x}}')\langle\overline{\overline{y}}'\rangle[\alpha'] & \xrightarrow{r} \quad I_1...id_i(\overline{\overline{x}}_r')\langle\overline{\overline{y}}'\rangle[\alpha'] \\ \quad ...I_n & \quad ...I_n\end{array}} \qquad (R_3)$$

$$\frac{\overline{\overline{x}}'' \subseteq \overline{\overline{y}}'}{\begin{array}{ll}\Sigma : I = I_0 \rightarrow I_1... & \Sigma : I = I_0 \rightarrow I_1... \\ \quad id_i(\overline{\overline{x'}_r})\langle\overline{\overline{y'}_r}\rangle[\alpha']... & \xrightarrow{r} \quad id_i(\overline{\overline{x'}_r})\langle\overline{\overline{y'}_r}\rangle[\alpha']... \\ \quad id_j(\overline{\overline{x''}})\langle\overline{\overline{y''}}\rangle[\alpha'']...I_n & \quad id_j(\overline{\overline{x''}_r})\langle\overline{\overline{y''}}\rangle[\alpha'']...I_n\end{array}} \qquad (R_4)$$

$$\frac{\overline{\overline{y}} \subseteq U\overline{\overline{y}}_i}{\begin{array}{ll}\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow & \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha] \rightarrow \\ \quad id_1(\overline{\overline{x}}_{1r})\langle\overline{\overline{y}}_{1r}\rangle[\alpha] & \xrightarrow{r} \quad id_1(\overline{\overline{x}}_{1r})\langle\overline{\overline{y}}_{1r}\rangle[\alpha] \\ \quad ...id_n(\overline{\overline{x}}_{nr})\langle\overline{\overline{y}}_{nr}\rangle[\alpha] & \quad ...id_n(\overline{\overline{x}}_{nr})\langle\overline{\overline{y}}_{nr}\rangle[\alpha]\end{array}} \qquad (R_6)$$

2.4.4.4 Recevoir une réponse pour un service composite

Lorsqu'un service en partie droite d'une instance reçoit la réponse d'un appel précédent, les paramètres en sorties deviennent définis. Dans la règle R_5 on suppose qu'en partie

droite de l'instance I nous avons $id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]$, après la réponse de M sur p, nous aurons $id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]$.

$$\frac{}{\begin{array}{l}\Sigma : I = I_0 \rightarrow I_1 ... \\ \quad id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]\ ...I_n\end{array} \xrightarrow{p(M)} \begin{array}{l}\Sigma : I = I_0 \rightarrow I_1 ... \\ \quad id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]\ ...I_n\end{array}} \quad (R_5)$$

2.4.5 Choix de service local (LoCh)

Lorsque dans une instance, un service de droite fait référence à un service défini dans l'espace Σ, alors le service sera instancié localement, l'instance remplace le service de droite. Une fois sélectionnées, les opérations décrites précédemment peuvent être appliquées.

$$\frac{id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \ \ match \ \ to \ \ I'_0 \rightarrow I'_1 \ \ ... \ \ I'_n}{\begin{array}{l}\Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \rightarrow \\ \quad I_1 ... id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle ... I_n\end{array} \xrightarrow{r} \begin{array}{l}\Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \rightarrow \\ \quad I_1 ... [I'_0 \rightarrow I'_1 \ \ ... \ \ I'_n] ... I_n\end{array}}$$

L'accent est mis sur les données qui influencent le choix des services, leurs créations et leurs raffinements. Le flux d'exécution dépend de la disponibilité des valeurs des variables d'entrée et de sortie. Dans un schéma composite de services, deux tâches s'exécutent en parallèle si les entrées de l'une ne dépendent pas des sorties de l'autre et en séquence si les entrées de l'une dépendent des sorties de l'autre. En reprenant les concepts de bases des langages de composition de services traditionnels [68], dans la prochaine une correspondance avec *GSLang* est présentée.

2.4.6 Correspondance avec les langages de création de services traditionnelles.

Un langage pour la composition des services comprend les différents éléments suivants: [69, 70]: activité, condition, événement, flux, message et rôle.

- Une activité: représente une fonction métier bien définie (similaire, par exemple, aux activités de base dans BPML). Elle contient un nom, les entrées et les sorties.

- Une condition: décrit le comportement de la composition en contrôlant les événements, en surveillant les activités et en appliquant les conditions préalables et les conditions postérieures.
- Un événement: décrit les événements au cours du processus de composition du service et son impact sur une activité. Celles-ci peuvent être à la fois de nature normale et exceptionnelle.
- Un *flow*: définit un bloc d'activités et comment elles sont connectées. Généralement en utilisant des activités de base. Ils montrent l'enchaînement des activités de base contenues dans les activités. Le bloc est explicitement défini sous la forme d'une séquence, d'une instruction conditionnelle ou même d'une boucle.
- Message: représente un conteneur d'informations. Les messages sont utilisés et générés par des activités. Ils décrivent les interactions entre les activités qui communiquent.
- Rôle: fournit une description abstraite pour un tiers participant à la composition du service. Ici, le rôle décrit l'emplacement d'un service partenaire.

Le *GSLang* est basé sur les règles qui décrivent un service en spécifiant son nom, les paramètres d'entrée et de sortie, les règles sémantiques (décrivant la correspondance entre les attributs) et les services à résoudre. Selon la dépendance des paramètres d'attributs et l'enchaînement des services dans les règles, nous avons des instructions séquentielles, des expressions conditionnelles, des boucles et la gestion de cas exceptionnel.

2.4.6.1 Séquence

Ceci arrive uniquement lorsque dans une règle définissant un service s, les paramètres en entrée du service s_2 dépendent des paramètres en sortie du service s_1. Alors le service s_1 s'exécutera avant le service s_2. Par exemple $s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s_2(y')\langle y\rangle$

2.4.6.2 Condition

Dans un espace d'exécution Σ. Un service peut avoir plusieurs définitions différentes, donc l'expression conditionnelle guidera le choix de l'une par rapport aux autres. Par exemple $a > 0$ alors la première définition sera choisie autrement, si $a = 0$ c'est le

deuxième.
$[a > 0]s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s_2(y')\langle y\rangle$
$[a = 0]s(x)\langle y\rangle \rightarrow$

2.4.6.3 Boucle

Dans la séquence d'une règle définissant un service s, nous pouvons avoir le service lui-même défini dans sa partie droite ou dans la partie droite d'un service présent dans la partie droite de s. Défini ainsi, nous avons une boucle. Par exemple
$[x > 0]s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s(y')\langle y\rangle$
$[x = 0]s(x)\langle y\rangle \rightarrow$

Il est très difficile en pratique de déterminer que le service exécutera jusqu'à la fin. Dans le chapitre suivant, nous proposons de transformer les spécifications de service en Promela/SPIN afin de valider qu'elles sont correctes.

2.4.6.4 Parallele (fork/join)

Si les deux services s_1 et s_2 présents sur le côté droit d'un service n'ont rien en commun, alors s_1 et s_2 peuvent s'exécuter en même temps. Il peut arriver que la sortie de s dépend des sorties de s_1 et s_2 comme dans l'exemple. $s(x)\langle y_1, y_2\rangle \rightarrow s_1(x)\langle y_1\rangle s_2(x)\langle y_2\rangle$

2.4.6.5 Traitement des cas exceptionnel

Comme la spécification des services est déclarative, l'ajout de règles spécifiques avec des conditions peut aider à gérer des cas exceptionnels. Cette description déclarative permet d'ajouter à tout moment des règles pour gérer des cas exceptionnels.

2.4.7 Résolution de services

La sémantique opérationnelle définie ci-dessus consiste à créer et matérialiser dynamiquement les instances de service lors de l'exécution en fonction des paramètres d'entrée et de sortie et du port créé lors de l'exécution. Dans le système, une instance de service I est résolue si tous ces paramètres sont définis (entrée et sortie). Toute action sur un

service résolu ne le modifie pas. c'est à dire.

I est résolu $\iff I^*$

La sémantique de cet opérateur est la suivante: si I^* et I $\xrightarrow{\alpha} I'$ alors I'^*.

Définition

Une instance de service I à laquelle sont attachées les variables d'entrée $\overline{\overline{x}}$ et les variables de sortie $\overline{\overline{y}}$ est résolue si toutes les instances de service associées I_i sont résolues, c'est-à-dire

$$\frac{I \to I_1...I_n,\ I_1^*,\ I_2^*...,\ I_n^*}{I^*}\ o\ \ I_i^* = id(\overline{\overline{x}}_{ir})\langle\overline{\overline{y}}_{ir}\rangle$$

Preuve

Si l'instance est simple, c'est-à-dire qu'elle est définie par $I \to$, alors I est résolue si les valeurs sont affectées aux paramètres en entrée x_r et aux paramètres en sortie y_r.

Si l'instance est composite, c'est-à-dire $I \to I_1...I_n$

- Une instance comme $I \to I_1$, par définition, $x \subseteq x_1$ et $y \subseteq y_1$ si I_1 est résolue c'est-à-dire nous avons x_{r1} et y_{r1} alors x et y sont définies, c'est-à-dire x_r et y_r.

- Une instance comme $I \to I_1...I_n$ par définition, les variables en entrée (resp. en sortie) de la partie gauche de x (resp. y) dépendent des paramètres en entrée (resp. en sortie) de la partie droite: $x \subseteq \cup_i x_i$ (resp. $y \subseteq \cup_i y_i$). si $\forall_i I_i$ est résolue c'est-à-dire nous avons x_{ir} et y_{ir}, $\Rightarrow$ nous avons x_r et y_r.

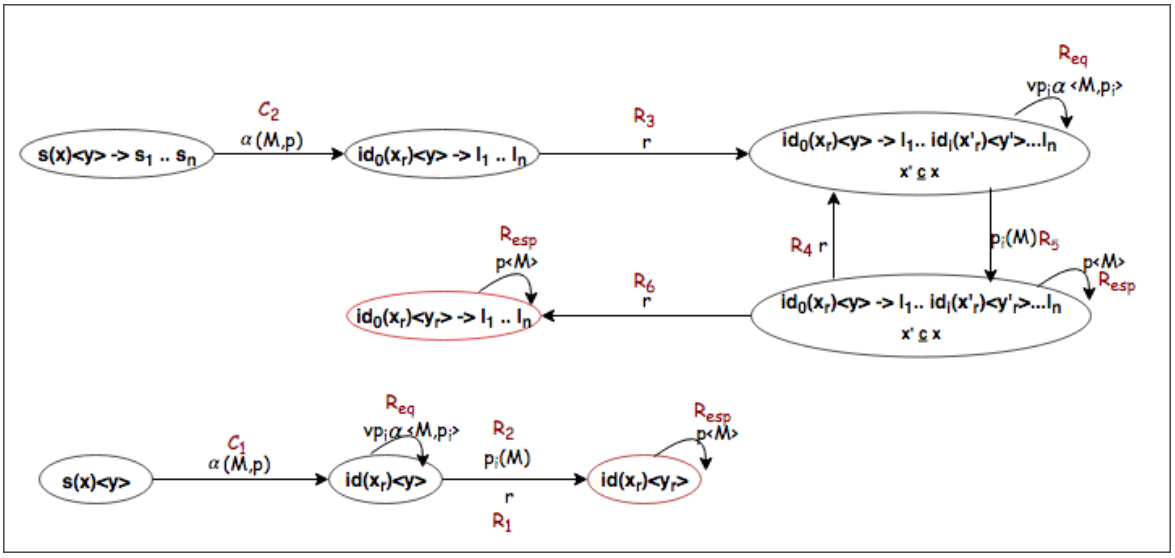

FIGURE 2.1: Progression d'un service instancié

Proposition

La sémantique opérationnelle génère un système $\perp$-transitions.

Preuve

Soit un système $\perp$-transition étiqueté défini par un n-tuple $(\Sigma, S, \rightarrow, \perp)$ où
- Σ est l'ensemble des **actions** qui étiquettent les transitions;
- $Q = S \cup I$ est l'ensemble des états où S est l'ensemble des **services** et I est l'ensemble des **instances**;
- $\delta \subseteq Q * \Sigma * Q$ la fonction de transition
- $\perp \in S * bool$ est un prédicat de terminaison tel que $\perp (s) = true$ et $s \xrightarrow{\alpha} s\prime$ alors $\perp (s\prime) = true$

Le système de transition décrit le comportement d'un service lorsqu'il est choisi dans un espace utilisateur. Dans la figure 2.1, toutes les opérations de la sémantique opérationnelle sont utilisées pour décrire comment un service instancié progressera pour être résolu.

Les opérations convertissent le système d'un état à un autre grâce à l'interprétation des attributs (action silencieuse), aux messages envoyés et reçus. Ces derniers correspondent dans le système de transition aux arcs étiquetés. Les services et les instances sont les nœuds.

Quand on regarde la règle d'instanciation C_1, si le système contenant un service de la forme $id(\overline{\overline{x}})\langle\overline{\overline{y}}\rangle$ (nœud du système de transition) reçoit un message ($\alpha(M,p)$) (arc dans le système de transition), le système crée une instance de la forme $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle$. Quand l'instance $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle$ est créé, on peut appeler le service associé en utilisant la requête $vp_i\alpha\langle M,p_i\rangle$ (opération Req). La réception de la réponse à cette requête $p_i(M)$ (opération R_2) permet de raffiner le service initial définissant ainsi la sortie $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle$.

D'un autre côté, pour un service composite, appliquer la règle C_2 lors de la réception d'un message sur le port public d'un pair $\alpha(M,p)$, crée l'instance avec les valeurs des services de la partie de gauche définis ($id_0(\overline{\overline{x}}_r)\langle y\rangle \rightarrow I_1..I_n$). Un raffinement peut donc être appliqué (règle R_3) qui permet à travers les règles sémantiques de définir certains paramètres de la partie droite. À ce stade, les services de la partie droite peuvent être appliqués, les requêtes $vp_i\alpha(M,p_i)$ sont envoyés à un autre pair, les réponses reçues sur les ports privés $p_i(M)$ (règle R_5) envoyé lors des requêtes.

Des raffinements sont appliqués (règles R_4 ou R_6). Lorsque toutes les sorties sont définies, nous atteindrons l'état final où les paramètres en sortie du service de la partie gauche sont définies.

La vérification de cette sémantique est démontrée à l'aide de l'outil SPIN [42]. Ce processus de vérification est présenté dans le chapitre suivant. La spécification de pi-calcul est traduite en Promela et SPIN est utilisée pour valider les services spécifiés. Nous voulons montrer que si le système est bien spécifié, chaque service instancié dans le système s'exécute jusqu'à la fin.

2.5 Caractéristiques de GSLang

Suivant la logique du pi-calcul, un espace utilisateur est modélisé comme un processus, qui contient des services matérialisés par les tâches. À cet égard, la figure 2.2 est composé de deux processus Σ_1 et Σ_2.

- L'espace 1 (Σ_1) contient la tâche s_{11} qui démarre le processus, puis se décompose en s_{12} et s_{21} qui se synchronisent pour terminer le processus. s_{21} est mis en œuvre au niveau de Σ_2
- L'espace 2 (Σ_1) contient la tâche s_{21}.

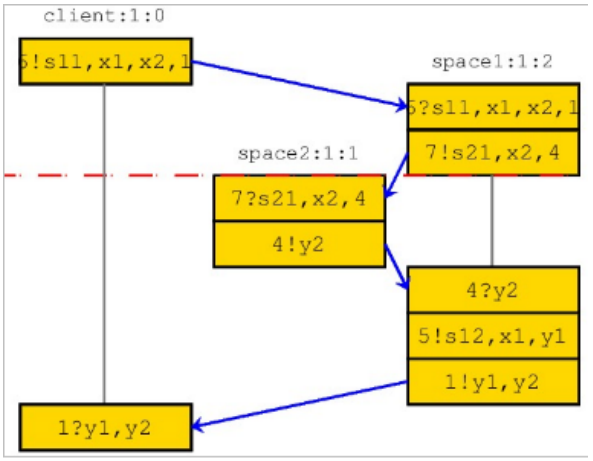

FIGURE 2.2: Exemple d'exécution

En utilisant ce langage, le processus décrit dans la figure 2.2 présente deux espaces utilisateurs Σ_1 et Σ_2. Initialement, Σ_1 contient les services $S_{\Sigma_1} = \{s_{11}, s_{12}\}$ et son port public α_1. Les services de Σ_1 sont définis par :

$s_{11}(x_1, x_2\{x_1 > 0, x_2 > 0\})\langle y_1, y_2\rangle \rightarrow s_{21}(x_2)\langle y_2\rangle s_{12}(x_1)\langle y_1\rangle$
$s_{12}(x_1)\langle y_1\rangle \rightarrow$

Le service s_{11} a une garde $x_1 > 0$ et $x_2 > 0$ et à besoin de s_{21} et s_{12} afin d'obtenir y_1 et y_2. s_{21} est un service distant implémenté dans l'espace Σ_2 et s_{12} est un service local simple de Σ_1.

Σ_2 contient le service s_{21}.($S_{\Sigma_2} = \{s_{21}\}$) et un port public α_2. Le service s_{21} est défini par: $s_{21}(x_2)\langle y_2\rangle \rightarrow$

Aussi dans la figure 2.2, un processus client c fait appel au service composite s_{11} de Σ_1. c définis x_1 et x_2, crée un port privé p dont la valeur est 1 et envoie le message contenant s_{11}, x_1, x_2 et p sur le port public α_1 (de valeur 5) de Σ_1 (ici la règle C_2 est mis en évidence). Une instance du service s_{11} sera créée dans Σ_1 parce-que s_{11} est trouvé et la garde vérifiée. La partie droite de s_{11} démarre, s_{12} et s_{21} exécute en parallèle puisqu'il n'y aucune dépendance entre leurs paramètres. Un appel distant vers Σ_2 sera fait lors de l'exécution de s_{21} (ici la règle *Req* est utilisée) c'est à dire en créant un port privé p_1 de valeur 4 et en envoyant un message contenant s_{21}, x_2 et p_1 sur le port public α_2 (de valeur 6) de Σ_2.

Sur Σ_2, en conformité avec les paramètres d'entrée, le service s_{21} est choisi (en appliquant la règle C_1), une instance du service est créée et exécutée. La réponse (principalement

y_2) sera envoyé à Σ_1 via le port privé p_1 (en appliquant la règle *Resp*). Σ_1 raffinera l'instance créée précédemment. La règle R_6 peut être utilisée et le port p_1 sera détruit. Finalement, la réponse sera envoyée au processus client via le port privé p précédent (de valeur 1).

Les ports privés asynchrones permettent de suivre individuellement l'exécution des instances de service. Une instance de service peut être indisponible pendant une période de temps; quand elle revient, elle peut continuer là où elle a été suspendu. De plus, les services peuvent être redéfinis à tout moment même pendant l'exécution car ils sont définis à la volée. Par exemple dans Σ_1 nous pouvons ajouter s_{13} alors que s_{11} est en cours d'exécution. C'est ce qu'on appelle la flexibilité par changement tel que défini dans [24], contrairement à la flexibilité par définition des approches de composition existantes [2][71]. De plus, les services sont entièrement définis sous forme de règles. Tel que décrit brièvement à la section 2.2, les règles ont été étudiées comme une approche déclarative présentant les avantages tels que:

- *Adaptabilité*: Étant donné la nature déclarative des compositions de services basées sur des règles, elles peuvent être modifiées et/ou étendues pour s'adapter à des situations spécifiques au contexte. L'adaptation du langage proposé dans cette thèse est possible au moment de l'exécution car chaque règle (service composite) est identifiée et chargée lorsque la règle est appliquée. La partie droite non encore activée peut être mise à jour même lors de l'exécution du service composite. Par exemple dans Σ_1, nous pouvons ajouter s_{13} lorsque s_{11} est en cours d'exécution. s_{11} deviendra :

  $s_{11}(x_1, x_2\{x_1 > 0, x_2 > 0\})\langle y_1, y_2\rangle \rightarrow s_{21}(x_2)\langle y_2\rangle s_{12}(x_1)\langle y_1\rangle s_{13}()\langle\rangle$

- *Flexibilité*: les compositions basées sur des règles sont plus flexibles que les compositions de type BPEL, étant donné leur capacité à poursuivre d'autres chemins d'exécution sans avoir à redéfinir le service composite et à le redéployer sur un moteur de service. Certaines langages comme BPEL4WS proposent un ensemble de balises (invoquer, répondre, recevoir, séquence, choix, flux, etc.) permettant de construire le service composite. Dans notre proposition, la définition et la composition des services sont décrites par les règles déclaratives, tandis que l'interaction est implicite à travers des attributs matérialisés par la transmission de paramètres. Les ports asynchrones privés (port dynamique) créés lors de l'exécution rendent la composition plus flexible. Le chemin d'exécution d'un

composite ne peut pas être déterminé à l'avance car les ports sont créés et détruits dynamiquement comme décrit dans l'exemple.

- *Sémantique intuitive formelle*: les langages basés sur des règles exploitent un ensemble logique et / ou mathématique de primitives sous-jacentes. Des approches formelles du raisonnement ont été proposées [22] [23], mais toutes utilisent le type de processus WS-BPEL pour leur implémentation. Nous proposons une définition intentionnelle des services qui permet une concrétisation paresseuse des services, favorisant ainsi un couplage faible avec la technologie sous-jacente et une adaptation (mise à jour des règles) du service même lors de son exécution. De plus, le langage proposé ne fait référence à aucune technologie. Le raisonnement peut être entrepris sur les services comme nous l'avons fait en définissant la sémantique opérationnelle dans la section 2.3.2.

- *Réutilisation et distribution*: les services composites définis principalement comme des règles peuvent être utilisés dans différents contextes. Les services sont distribués dans différents espaces utilisateurs selon l'architecture pair-à-pair. Dans l'exemple, nous avons les espaces Σ_1 et Σ_2 localisés par leurs ports publics respectifs α_1 et α_2

2.6 Conclusion

Ce chapitre a présenté un langage basé purement sur les règles, qui sert de cadre à la composition des services. Nous avons présenté une description formelle des concepts de base de ce langage et de leur comportement à travers une sémantique opérationnelle. Nous avons également présenté ce que nous entendons par service résolu par l'entremise d'un théorème et d'une proposition. Le langage proposé bénéficie des propriétés du modèle de *workflow* centré sur les données:

- Les services composites sont définis de manière déclarative (à la volée) sous la forme de règles, ce qui offre plus de flexibilité et d'adaptabilité.
- Les services participants à une composition collaborent dans un style pair-à-pair.
- Un service élémentaire ou composite peut être réutilisé dans différents contextes applicatifs.

Dans les chapitres suivants, nous déroulons la validation de notre langage de composition de services à travers le développement des outils logiciels sous-jacents tels que l'éditeur de services, les outils de vérification et de transformation. À cet égard, la sélection d'un environnement de vérification de modèle proche du pi-calcul est indiquée.

.

Chapter 3

Cadre de vérification et de validation des spécifications GSLang

Sommaire

3.1 Introduction

Dans le chapitre précédent, nous avons présenté le langage *GSLang* pour la composition de service dynamique. Ce langage permet une composition flexible des services. Il est abstrait, intentionnel et déclaratif. Dans ce chapitre, nous répondons à la question sur *comment savoir qu'une spécification GSLang est correcte?*. Nous présentons un *framework* de développement comprenant les phases de spécification des services en

GSLang, de simulation/vérification et d'opérationnalisation. L'approche MDE (*Model Driven Engineering*) en français IDM (*Ingénierie Dirigée par les Modèles*) est utilisée pour automatiser la transition entre les différentes phases. Par exemple, la transformation des spécifications en *PROMELA* pour leur vérification d'une part et le raffinement vers le système opérationnel (*XML*) d'autre part. Dans la section 3.2 nous décrivons ce processus. A la section 3.3 nous présentons les caractéristiques de l'IDM ensuite dans les sections 3.4 et 3.5 ngineeringdétaillent les différentes étapes de notre processus de développement.

3.2 Processus de développement avec GSLang

Un processus désigne l'enchaînement des activités dont le but est de créer un produit. En Génie Logiciel, un processus de développement décrit la séquence des activités d'ingénierie nécessaires pour transformer les besoins et les exigences en un produit logiciel [72]. Le processus de développement avec GSLang (figure 3.1) permet de définir un nouveau système opérationnel ou de faire évoluer un système existant. Le processus démarre par la définition des spécifications de services en GSLang. Dans cette phase, les services sont définis à la volée sur la base de syntaxe définie à la section 2.3.2. Une fois que les services GSLang sont syntaxiquement corrects, il convient de vérifier s'ils s'exécuteront correctement. Pour ce faire nous transformons les spécifications vers les processus Promela, une description littérale de la transformation est décrite à la section 3.4.1.1. Dans la phase de vérification, nous simulons l'ensemble des processus obtenus afin de savoir s'ils s'exécuteront jusqu'à la fin (section 2.4.7). Si tel n'est pas le cas, nous revenons à la phase de spécification sinon, nous passons à la phase de raffinement. Le raffinement consiste à projeter les spécifications GSLang vers des documents XML. Ces derniers seront ensuite distribués dans chaque pair. Cette distribution consiste à mettre à jour les documents XML dans les différents pairs. Enfin les services de chaque pair s'exécutent.

Afin d'automatiser les différentes phases et garder la cohérence entre les spécifications GSLang et le système opérationnel nous utilisons l'ingénierie dirigée par les modèles.

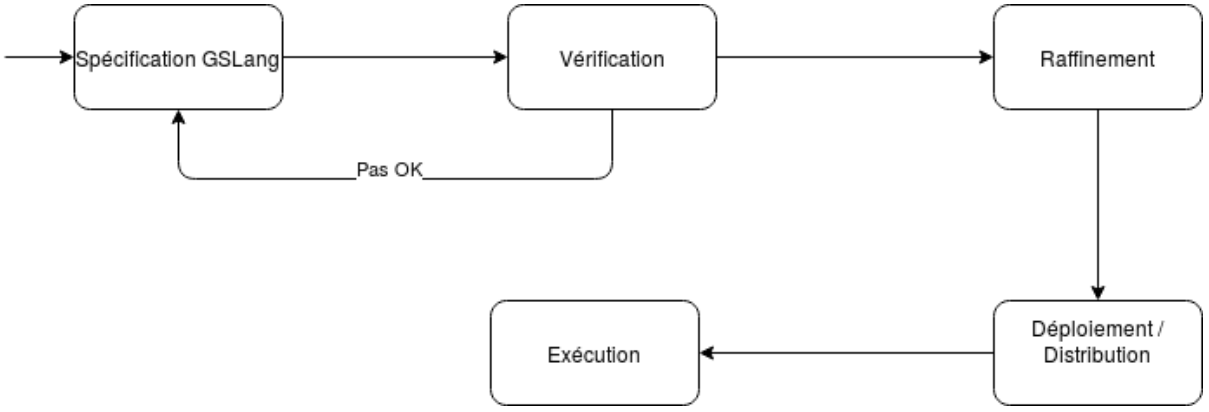

FIGURE 3.1: Processus de développement

3.3 Ingénierie Dirigée par les Modèles

La technologie *objets* a déclenché l'évolution vers les modèles. En effet, une fois la conception des systèmes informatiques sous la forme d'objets acquise, il s'est posé la question de les classifier en fonction de leurs différentes origines. L'IDM vise donc à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations. Le concept central de l'IDM est la notion de modèle pour laquelle il n'existe pas à ce jour de définition universelle. Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé. Pour concevoir un modèle, il faut définir un langage: on parle de meta-modèle. Le modèle doit être conforme au méta-modèle. Un *méta-modèle* est un modèle qui définit le langage d'expression d'un modèle [73], c.-à-d. le langage de modélisation. Un modèle est dit conforme à son méta-modèle comme un programme est conforme à la grammaire du langage de programmation dans lequel il est écrit [74]. À cet égard, l'OMG a introduit l'architecture à quatre niveaux illustrée à la figure 3.2. Au niveau inférieur, la couche $M0$ est le système réel. Un modèle représente ce système au niveau $M1$. Ce modèle est conforme à son méta-modèle défini au niveau $M2$ et le méta-modèle lui-même est conforme au méta-méta-modèle au niveau $M3$. Le méta-méta-modèle se conforme à lui-même. OMG a proposé MOF comme norme pour spécifier les méta-modèles. Par exemple, le méta-modèle UML est défini en termes de MOF. Sur la base de ces principes, l'OMG a défini MDA(Model Driven Architecture) en étendant UML(Unified Modeling Language) pour se conformer aux principes de MDE.

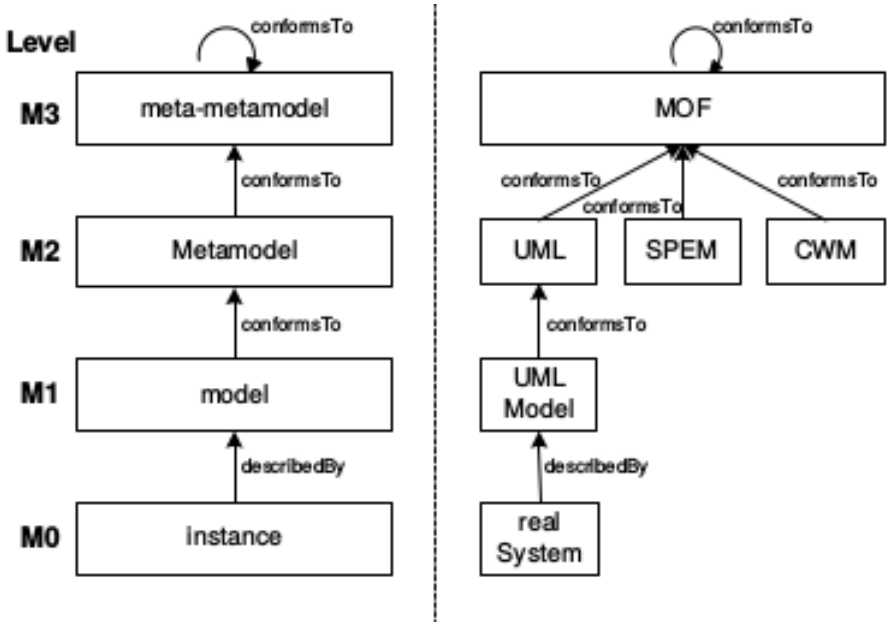

FIGURE 3.2: L'architecture de métamodélisation à quatre couches [74]

3.3.1 L'approche MDA

MDA est un ensemble de techniques de modélisation et de transformation de modèles normalisés par l'OMG [75]. Cette approche préconise l'utilisation de modèles dans les différentes phases du cycle de développement d'une application. Elle vise en particulier à développer les exigences du modèle, l'analyse et la conception des modèles et les modèles de codes. Dans l'approche MDA traditionnelle (illustrée sur la figure 3.3), l'objectif est de pouvoir générer des modèles spécifiques à une plateforme (PSM) à partir de modèles indépendants de cette plateforme (PIM) [75]. Afin de réaliser cette tâche automatiquement, des modèles précis des plates-formes cibles ont été définis tels que CORBA, DotNet, EJB, etc. Les modèles de description de plate-forme (PDM) semblent actuellement être le chaînon manquant de l'approche MDA. Son extension à travers l'IDM en apporte une solution générale. Les notions de modèle spécifique à la plate-forme (PSM) et de modèle indépendant de la plate- forme (PIM) seront ensuite utilisées.

Les transformations permettent de lier ces différents modèles. Les transformations de modèles permettent la mise en correspondance d'un modèle à l'autre. La transformation étant au cœur de MDA, OMG a standardisé Query/View/Transformation (QVT) [76] dont ATL [77] est une implémentation dans l'environnement Eclipse [78]. Ce langage permet de décrire des règles pour transformer un modèle en un autre ainsi que des requêtes permettant de convertir un modèle en texte.

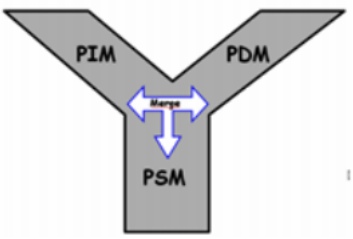

FIGURE 3.3: Étapes de l'approche MDA [75]

3.3.2 L'IDM comme une approche d'adaptation dynamique au contexte

L'adaptation dynamique au contexte est difficile à gérer dans le développement logiciel traditionnel, centré sur le code, car la notion de contexte varie fréquemment d'un système à l'autre [79, 80]. Nous voulons qu'une nouvelle spécification ajoutée dans le système le laisse dans un état cohérent. Ce problème peut être résolu à l'aide d'un développement logiciel basé sur les modèles. Les modèles nous permettent de définir des abstractions appropriées pour isoler et spécifier les différents contextes d'utilisation.

L'IDM nous permet :

- de définir un cadre de vérification automatique d'une spécification et
- de générer automatiquement les codes XML pour le système opérationnel.

3.4 Cadre de Vérification

Cette section présente une technique basée sur la transformation de modèle permettant de vérifier qu'une spécification *GSLang* est correcte. Cette transformation est aussi appliquée chaque fois qu'une mise à jour est faite sur un service. Le but ultime étant de vérifier la cohérence de la spécification *GSLang*.

Le processus de vérification et de validation de la spécification est décrit dans Figure 3.4. Il se compose de deux phases:

- (i) traduire une spécification de services GSLang vers une spécification Promela;
- (ii) simuler et vérifier la spécification Promela pour valider les propriétés.

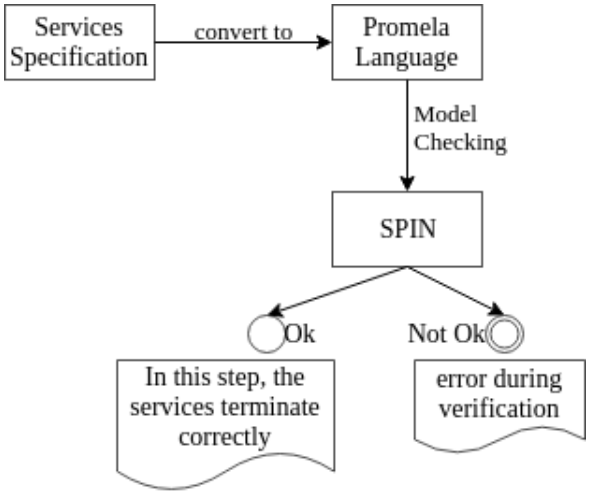

FIGURE 3.4: Processus de vérification

3.4.1 De GSLang vers PROMELA

Le langage *PROMELA*(PROcess MEta LAnguage) est un langage de spécification abstrait d'un système. Il modélise la synchronisation et la coordination des processus. C'est un langage permettant de définir des processus distribués. Chaque processus est donné par une déclaration *proctype* qui est constituée d'un ensemble d'instructions. *PROMELA* est proche du langage *C*. Les types de données sont *bit* ou *bool*, *byte*, *short* et *int*. La déclaration et l'utilisation des tableaux se font exactement comme en *C*.

Le type *channel* qui n'existe pas en *C* permet de définir un canal de transmission. Les canaux (*channels*) modélisent le transfert de données entre processus. Par exemple, *a!expr;* permet l'envoi de *expr* sur le canal *a*, de même, *a?expr* permet la réception de *expr* sur le canal *a*. *expr* peut être plusieurs valeurs.

Promela est choisi parce qu'il permet dans un message d'envoyer un canal similaire au pi-calcul [81] [82]. Un message contient un ensemble de données ainsi qu'un port sur lequel le processus distant transmettra sa réponse.

3.4.1.1 Description littérale de la transformation

Les tables 3.1 et 3.2 présentent les correspondances littérales entre une spécification *GSLang* et *PROMELA*.

TABLE 3.1: Règles de transformation littérale de la spécification GSLang vers le langage Promela. Les variables sont traduites en mtype, les ports en canaux, les services en structures de blocs et les rôles en processus.

Spécification GSLang	Code Promela	Commentaire
variable, valeur et port	mtype, byte, chan	utilisez *mtype* ou *byte* pour définir les variables et les valeurs Et *chan* pour définir le port
$x \leftarrow u \equiv x_r$	x = u	affectation de la valeur (u) à la variable (x)
$\overline{e}_b^x$	$==, <, >, <=, ! =, >=$	les expressions conditionnelles promela sur $\overline{\overline{x}}$ par exemple $x_1 == x_3$, $x1 > u$, $x_1 != x_3$. Nous les noterons par **c**.
$s(\ \overline{\overline{x}}, \overline{e}_b^x)\ \langle \overline{\overline{y}}, \overline{e}_b^y \rangle [\alpha]$	**begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** :: $(s == id\ \ \&\&\ \ c\ \ \&\&\ \ \overline{\overline{x}}) \rightarrow p!\overline{\overline{y}};$ **goto begin;** **fi**	où α est le port public d'un pair contenant **s** dans sa spécification de service ,**p** est le port privé contenu dans la requête et utilisé comme canal de retour. Traduction du code Promela: à la réception de $s, \overline{\overline{x}}$ *et* p sur α, le choix de l'implémentation appropriée en fonction de la garde et des paramètres d'entrée. Il retourne $\overline{\overline{y}}$ sur **p**.
$s(\ \overline{\overline{x}}, \overline{e}_b^x)\ \langle \overline{\overline{y}}, \overline{e}_b^y \rangle [\alpha] \rightarrow$ $s_1(\ \overline{\overline{x}}_1)\ \langle \overline{\overline{y}}_1 \rangle [\alpha_1]$ $s_2(\ \overline{\overline{x}}_2)\ \langle \overline{\overline{y}}_2 \rangle [\alpha_2]$ $s_3(\ \overline{\overline{x}}_3)\ \langle \overline{\overline{y}}_3 \rangle$ ... $s_n(\ \overline{\overline{x}}_n)\ \langle \overline{\overline{y}}_n \rangle [\alpha_n]$	**begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** :: $(s == id\ \ \&\&\ \ c\ \ \&\&\ \ \overline{\overline{x}}) \rightarrow$ $\alpha_1!s_1, \overline{\overline{x}}_1, p_1$ $p_1?\overline{\overline{y}}_1;$ $\alpha_2!s_2, \overline{\overline{x}}_2, p_2$ $p_2?\overline{\overline{y}}_2;$ y_3 //define y_3 ... $\alpha_n!s_n, \overline{\overline{x}}_n, p_n$ $p_n?\overline{\overline{y}}_n;$ $p!\overline{\overline{y}};$ **goto begin;** **fi**	Ce qui signifie qu'à la réception de $s, \overline{\overline{x}}$ *et* p sur α, le choix du service approprié est fait selon les paramètres en entrée et la garde (c). Ensuite, les services du côté droit sont traités en parallèle ou séquentiellement selon que les sorties correspondent ou non aux entrées. Le processus Promela est un pair. Les ports privés (p1, p2 ... pn) sont créés dans le processus représentant le pair. Enfin, envoie la réponse finale $\overline{\overline{y}}$ sur le canal **p**.

TABLE 3.2: Règles de transformation littérale de la spécification GSLang au langage Promela. Les variables sont traduites en mtype, les ports en canaux, les services en structures de blocs et les rôles en processus.

Spécification GSLang	Code Promela	Commentaire
$\Sigma = (S_s, I_s, \alpha)$	chan α = [k] of {mtype,mtype chan}; **Proctype peerName(){** chan p1 = [2] of {mtype,mtype,chan}; ... chan pn = [2] of {mtype,mtype,chan}; **begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** :: (c_1) $\rightarrow$ $block_1$ **goto begin;** :: (c_2) $\rightarrow$ $block_2$ **goto begin;** ... :: (c_k) $\rightarrow$ $block_k$ **goto begin;** **fi** }	Le processus promela est un pair. Les ports publics sont des canaux définis globalement, les ports privés sont locaux. Les différents services sont les alternatives de l'instruction if. Nous ne différencions pas les services de leurs instances. Le code promela sera simplement exécuté

De cette description littérale de la transformation d'une spécification GSLang vers Promela présentée, nous explicitons l'utilisation de l'Ingénierie Dirigée par les Modèles pour faire cette transformation. Dans un premier temps, nous présentons le modèle source issu de la description GSLang ainsi le modèle cible Promela. Ensuite nous définissons les règles de transformations.

3.4.1.2 Meta-modèle GSLang

La figure 3.5 décrit le formalisme permettant de spécifier un service GSLang dans son espace utilisateur. Le système représenté par la classe *System* est composé d'un ensemble de pairs représenté par la classe *role*. Un rôle est caractérisé par son nom et surtout un emplacement (son adresse ou sa *location*). A l'intérieur d'un rôle, on retrouve un ensemble de services (*Service*). Un service peut être simple (*Simple*) ou composite (*Composite*). Lorsqu'un service est simple, il peut être manuel c'est-à-dire juste un formulaire à remplir ou automatique c'est-à-dire une implémentation *REST* ou *SOAP*, dans ce cas, il est caractérisé par une localisation (*location*) permettant de l'invoquer. Lorsqu'un service est composite, il est constitué de services simples

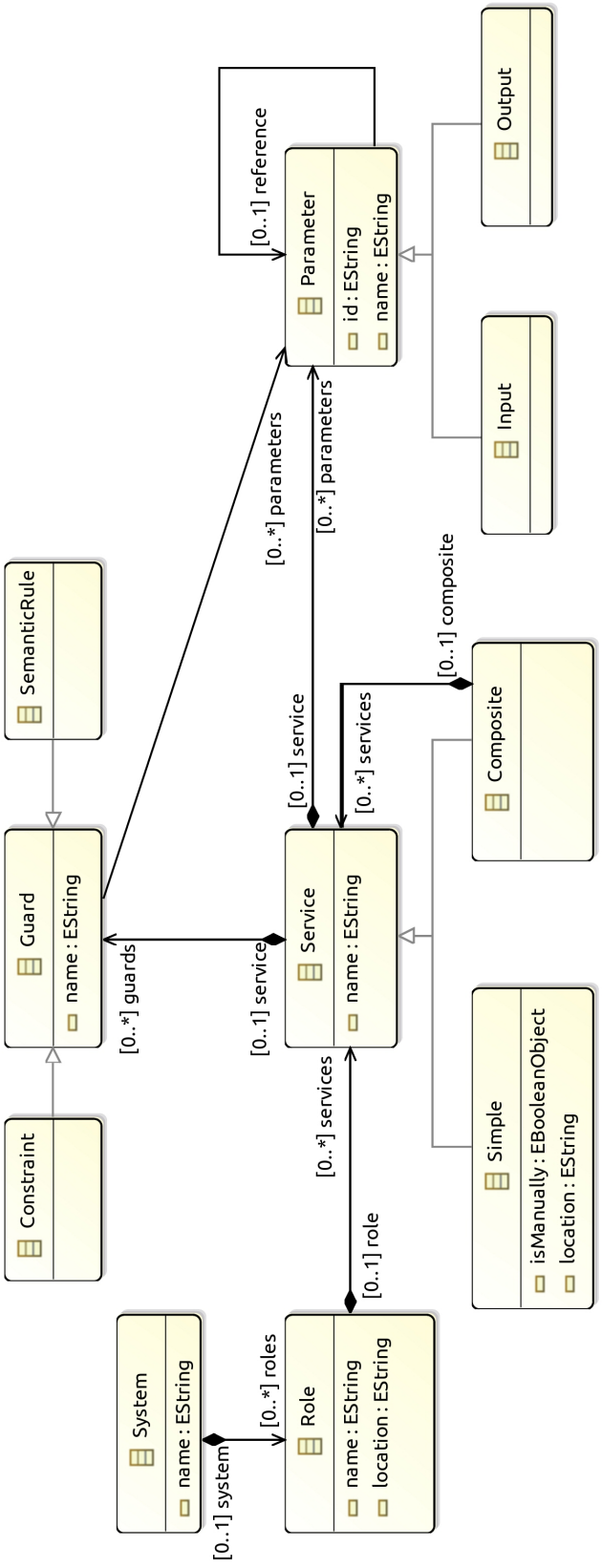

FIGURE 3.5: Méta-modèle GSLang

ou composites. Tout service possède des paramètres en entrée (*Input*) ou en sortie (*Output*). Un paramètre peut faire référence à d'autres paramètres. Enfin, un service a une garde (*Guard*) qui est une expression conditionnelle sur les paramètres des services.

3.4.1.3 Meta-modèle Promela

La figure 3.6 synthétise le meta-modèle Promela présenté dans [83–85].

Un modèle Promela (*Model*) contient des processus (*Process*). A l'intérieur d'un processus, nous avons des variables ($Variable$). Une variable peut être primitive (*Primitive*) c'est-à-dire de type simple (mtype, bit, byte, int ou chan) ou un canal (*Channel*). Un modèle contient lui aussi des variables qui seront globales à l'ensemble de processus. Un processus contient des paramètres (*Parameter*) ayant un nom et un type. Le processus contient également plusieurs instructions. Une instruction peut être simple (*SimpleStatement*) ou composite (*CompositeStatement*). Une instruction simple consiste en la réception d'éléments distants sur des ports alors qu'une instruction composition peut être un Do, un Id, un $Case$ ou un For. Naturellement une instruction composite est constituée d'instructions simples ou composites.

3.4.1.4 Règles de transformations

Dans cette section, nous nous appuyons sur les méta-modèles GSLang et Promela présentés dans les sous-section 3.4.1.2 et 3.4.1.3 respectivement. Pour obtenir les différentes règles de transformations, nous avons traduit les classes du méta-modèle GSLang vers les classes du méta-modèle Promela. Dans le prochain chapitre, ces règles seront décrites en ATL. Les principales règles de transformation sont:

- La règle **guard2Condition**: transforme les gardes associées au service en condition pour la déclaration if de promela.
- La règle **service2ChannelAndSimpleStatement**: transforme des services simples en variables de type canal et en une instruction simple dans un processus *proctype* promela.
- La règle **service2ChannelAndCompositeStatement**: transforme les services composites en variable de type canal et en instruction composite comme l'instruction If

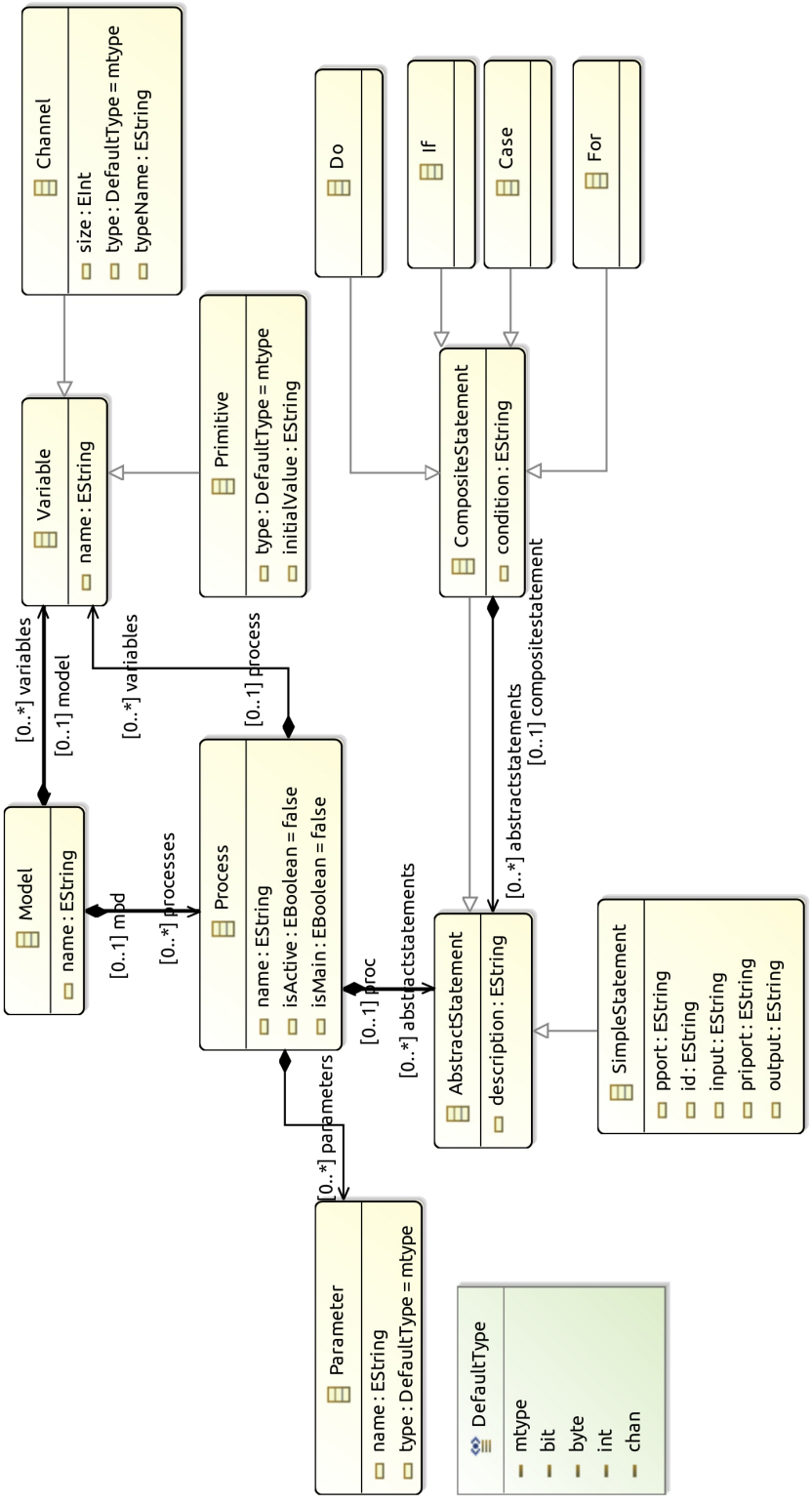

FIGURE 3.6: Méta-modèle Promela

- La règle **role2Process**: transforme un rôle (pair) en un processus de promela.
- La règle **system2Model**: transforme le système composé de l'ensemble des rôles en un modèle promela composé de processus.

3.4.2 Vérification de modèles

Les processus générés dans Promela sont simulés et vérifiés à l'aide de SPIN. Lorsqu'un nouveau service est ajouté ou qu'un service est mis à jour, il est transformé en Promela et vérifié. La vérification de modèle est explicitée sur un exemple. Le processus client envoie une demande et attend un résultat. Lorsque le processus client reçoit le résultat, la propriété de terminaison du processus appelé est vérifiée. Lorsque pour un service vérifié, tous les ports privés reçoivent les données, puis le service se termine. La propriété de terminaison est donc vérifiée. En LTL (Linear Temporal Logic): $[\,]\,(p?x)$ signifie que lorsqu'un port p est privé, il reçoit toujours des données. p étant un port privé créé lors de l'exécution d'un service s et x étant les données reçues.

La vérification formelle du modèle contribue à la démonstration de la propriété *soundness* trop difficile à prouver théoriquement [12].

La table 3.3 décrit un exemple de services et de leurs transformations en Promela répartis sur trois espaces utilisateurs. La transformation de Promela est ensuite simulée. Trois scénarios sont présentés.

scenario 1. Les services ont été correctement spécifiés (Figure 3.7); Le processus client appelle le service s de *space1* sur son port public (7). Le processus de *space1* reçoit et instancie s. s appelle s_1 de *space2* en créant le port privé 4. *space*2 synthétise y_1 et l'envoie à *space*1. Ce dernier appelle $s2$ de *space*3 afin de synthétiser y_2 s_2 de *space3* crée le port privé 5 sur la figure, défini y_2 et l'envoie à *space*1. Enfin, le processus client reçoit la réponse attendue.

scenario 2. Les paramètres des services appelés ne correspondent pas (Figure 3.8). A partir de la table 3.3, avant le test, les paramètres d'entrée du service s_1 ne correspondent pas entre *space*1 et *space*2;

scenario 3. Le service appelé n'est pas défini (Figure 3.9). Dans le tableau 3.3, avant le test, le service s_2 a été supprimé.

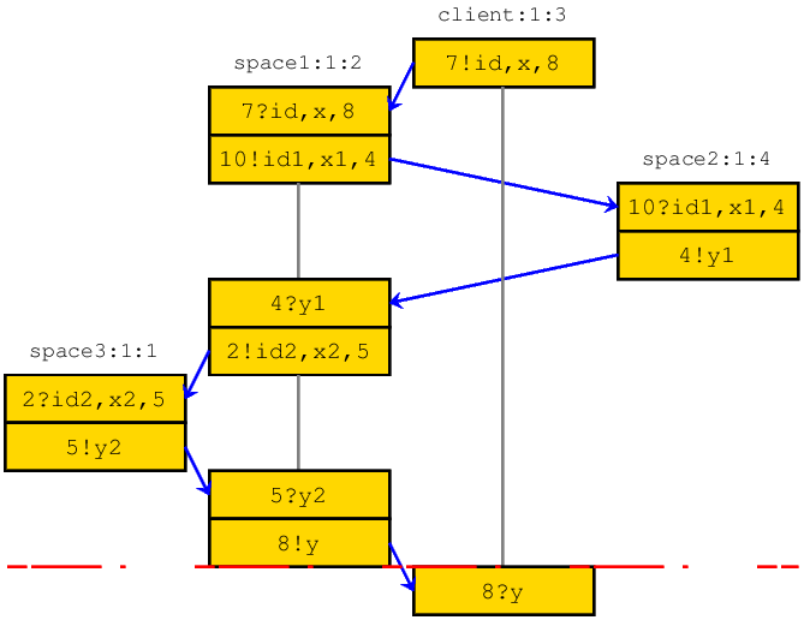

FIGURE 3.7: Exécution correcte

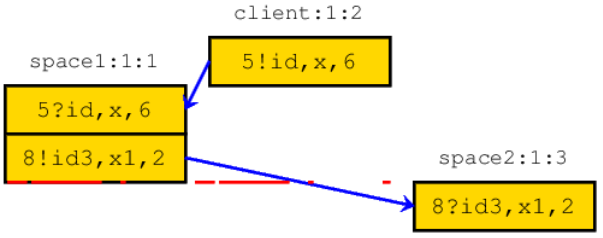

FIGURE 3.8: Services non correspondants

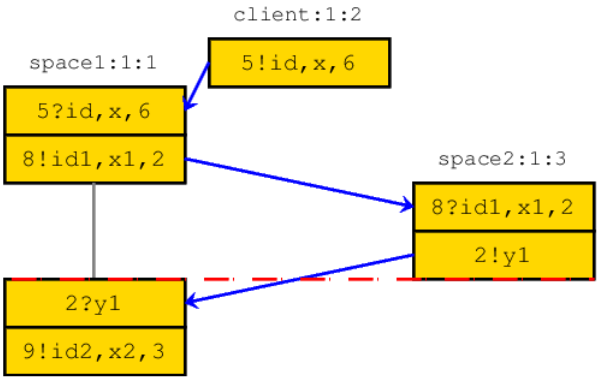

FIGURE 3.9: Service non défini

TABLE 3.3: convertir la spécification de service en processus Promela

Espace	Code Promela
espace 1 $s(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$ $s_1(\overline{\overline{x}}_1)\ \langle\overline{\overline{y}}_1\rangle[\alpha_1]$ $s_2(\overline{\overline{x}}_2)\ \langle\overline{\overline{y}}_2\rangle[\alpha_2]$ $s_3(\overline{\overline{x}}_3)\ \langle\overline{\overline{y}}_3\rangle$ $s_3(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space1(){** **begin:** $\alpha?s,x,p$ **if** $::(s == id) \rightarrow$ $\alpha_1!id_1,x_1,p_1;$ $p_1?y_1$ $\alpha_2!id_2,x_2,p_2;$ $p_2?y_2$ y_3 $//define y_3$ $p!y$ **goto begin;** **fi** }
espace 2 $s_1(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space2(){** **begin:** $\alpha_1?s,x,p$ **if** $::(s == id1) \rightarrow$ $p!y_1$ **goto begin;** **fi** }
espace 3 $s_2(\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space3(){** **begin:** $\alpha_2?s,x,p$ **if** $::(s == id2) \rightarrow$ $p!y_2$ **goto begin;** **fi** }

3.5 Spécification GSLang vers le système opérationnel : Cadre de Validation

Lorsque la spécification GSLang est correcte, elle est raffinée vers le système opérationnel. XML est utilisé comme langage cible. Nous présentons ici deux méta-modèles quasi identiques. Le premier (figure 3.10) est celui du Service et le second (figure 3.11) est celui d'une instance de services. Un service est simple ou composite. Lorsqu'il est composite, il est constitué d'un ou de plusieurs services. Ces derniers sont matérialisés

par la classe *Use* qui est caractérisée par un nom (*name*), un type (*type*) et une localisation (*location*) ainsi que des paramètres en entrée (*input*), des paramètres en sortie (*output*) et une garde (*Guard*) sur les paramètres en entrée.

L'instance de service quant à lui a aussi une structure de service initialement mais dont les *Use* seront concrétisés au fur et à mesure de l'instanciation des services y afférents. La balise *Use* ici matérialise les données intentionnelles qui seront enrichies lors de l'application des services. A l'exécution, lorsqu'une instance de services ne contiendra plus de balise *Use*, alors cette instance est résolue. En d'autres termes, l'instance de service se développe au fur et à mesure que la partie intentionnelle se concrétise c'est-à-dire les services en partie droite sont appliqués.

Ces méta-modèles représentent les services et leurs instances tels qu'ils seront utilisés par le moteur d'exécution.

Avec *ATL*, nous avons écrit des *Helpers* et une *Query* afin d'obtenir les fichiers XML des services dont le méta-modèle a été décrit précédemment.

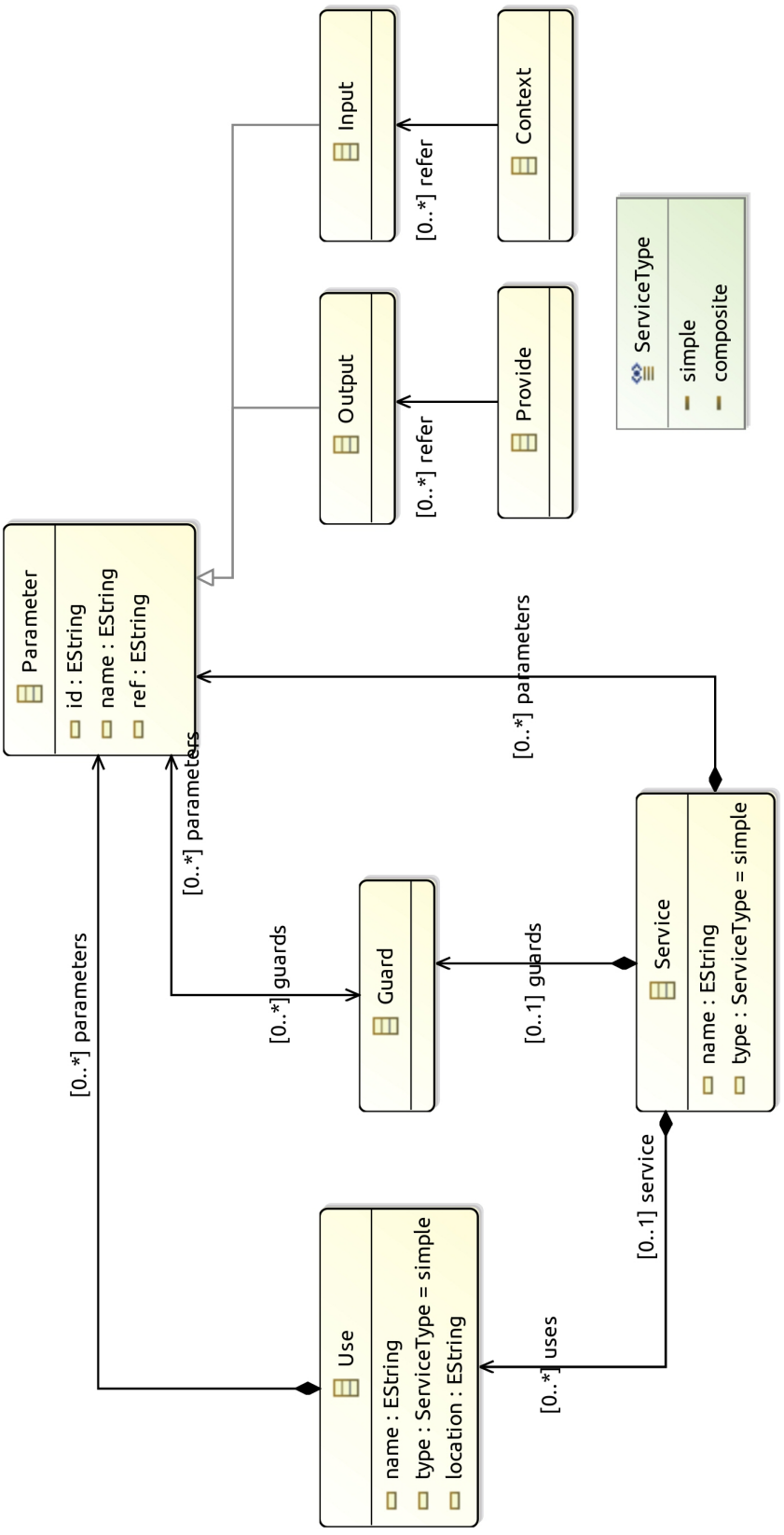

FIGURE 3.10: Méta-modèle des services

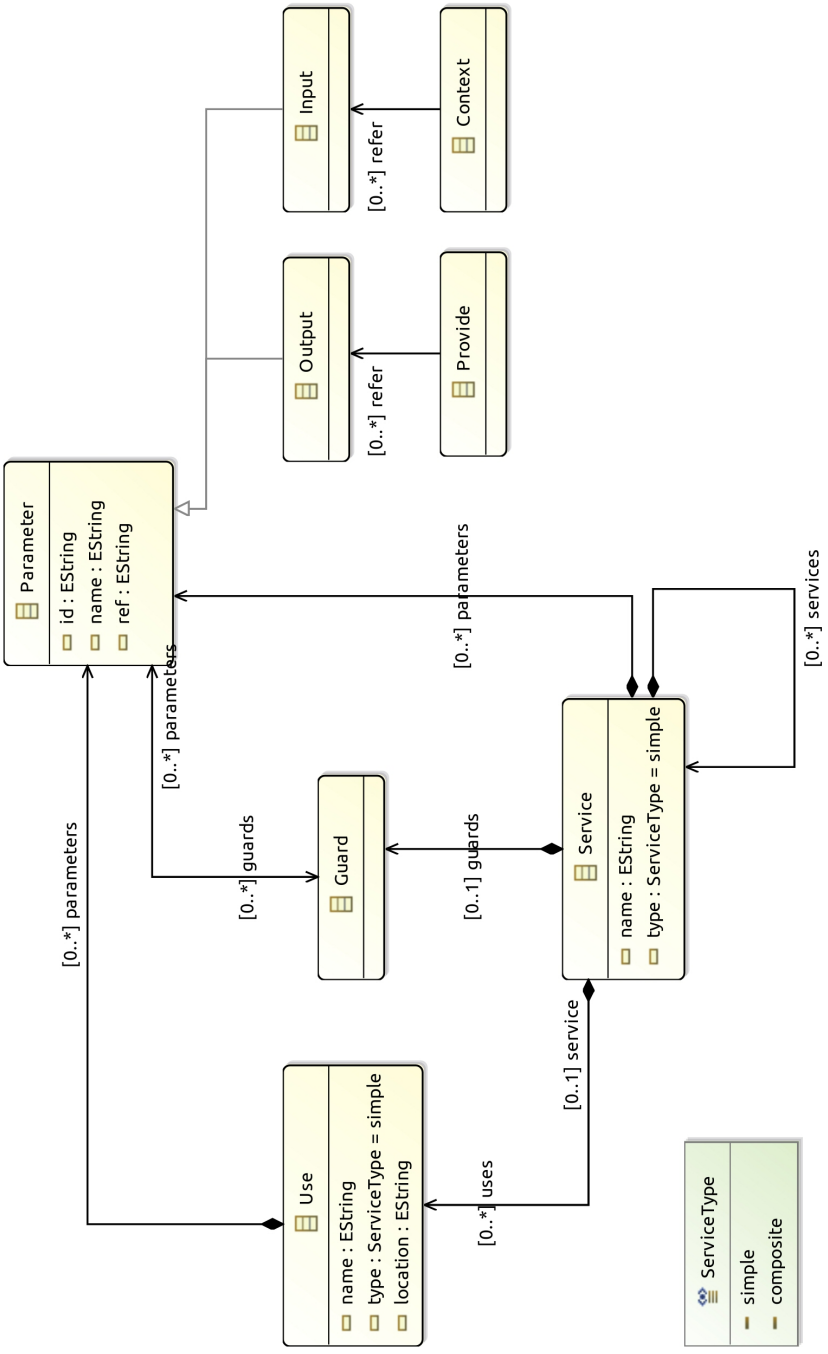

FIGURE 3.11: Méta-modèle des instances de services

3.6 Conclusion

Dans ce chapitre nous nous sommes appuyés sur l'Ingénierie Dirigée par les Modèles afin de vérifier automatiquement qu'une spécification GSLang est correcte. Nous avons commencé par définir et présenter le méta-modèle source (*GSLang*) et le méta-modèle cible (*PROMELA*). Ensuite, nous avons défini les règles de transformations et les éléments de vérification de modèles.

Dans le prochain chapitre, nous allons présenter les outils supports pour automatiser les activités du processus ainsi qu'une expérimentation.

.

Chapter 4

Outils et Expérimentation

Sommaire

4.1 Introduction

Dans le chapitre précédent, nous avons présenté d'une part le cadre théorique pour la transformation automatique des spécifications GSLang vers Promela pour leur vérification et d'autre part la génération de codes XML qui sera utilisée par le moteur d'exécution.

Dans le présent chapitre, nous présentons l'environnement logiciel de l'éditeur des spécifications GSLang conçu, les règles de transformation et le moteur d'exécution. Ensuite, nous expérimentons ces outils sur un exemple de gestion simplifiée des missions dans une organisation. Une description détaillée du cas est présentée, le cas est ensuite décrit dans l'éditeur et nous montrons comment vérifier les services spécifiés et présentons en détail l'exécution des services. Enfin une discussion est faite par rapport au style architectural, aux approches déclaratives et orientée-données.

4.2 Environnement logiciel

Un prototype a été développé. Dans une architecture distribuée pair à pair, chaque pair comprend un éditeur, un moteur de transformation et un moteur d'exécution. Ces derniers permettront de faciliter la conception, la vérification et l'exécution des spécifications *GSLang*.

4.2.1 Éditeur conçu grâce à Xtext

Pour obtenir notre éditeur, nous avons utilisé Xtext [1] qui est un framework de développement de langages et permet de développer rapidement des outils pour un langage textuel. Xtext fournit un ensemble de langages spécifiques au domaine et des API modernes pour décrire les différents aspects de votre langage de programmation. Il donne une implémentation complète du langage conçu exécuté sur une JVM. Les composants du compilateur de votre langage sont indépendants d'Eclipse et peuvent être utilisés dans n'importe quel environnement Java. Ils comprennent des éléments tels que l'analyseur syntaxique, l'arbre de syntaxe abstraite de type (AST), le *srialiseur*, le formateur de code et l'analyseur statique. Ces composants d'exécution sont basés sur le cadre de modélisation Eclipse (EMF)[86]. Xtext[87] [88] est aussi vu comme un framework Eclipse pour implémenter des langages de programmation et des DSL. Il vous permet d'implémenter des langages rapidement et, surtout, il couvre tous les aspects d'une infrastructure de langage, à partir de l'analyseur, du générateur de code ou de l'interpréteur, jusqu'à une intégration dans l'IDE Eclipse. A la fin nous obtenons un IDE Eclipse spécialement conçu pour notre langage.

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

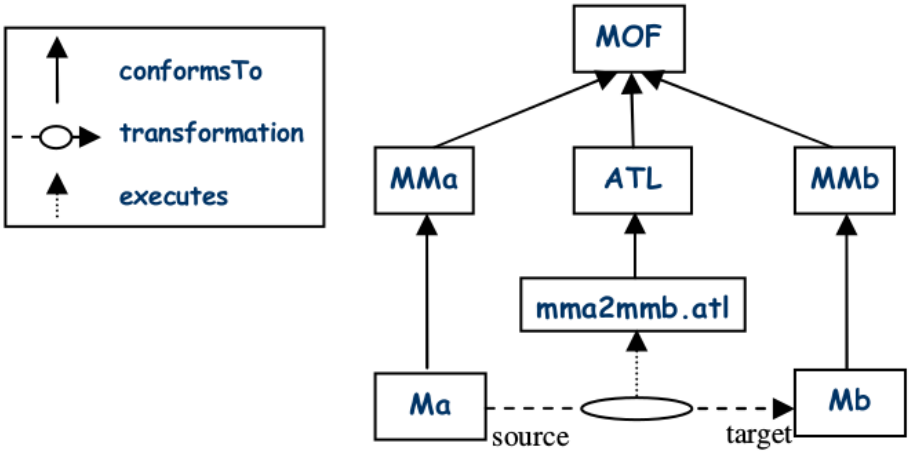

FIGURE 4.1: Aperçu de l'approche transformationnelle ATL [89]

4.2.2 Moteur de Transformation conçu grâce à ATL

ATL[89] est appliquée selon un modèle transformationnel illustré sur la figure 4.1. Dans ce modèle, un modèle source Ma est transformé en un modèle cible Mb selon une définition de transformation $mma2mmb.atl$ écrite dans le langage ATL. La définition de la transformation est un modèle. Les modèles source et cible et la définition de transformation sont conformes à leurs méta-modèles MMa, MMb et ATL respectivement. Les méta-modèles sont conformes au méta-méta-modèle MOF[73].

ATL est un langage de transformation hybride. Il contient un mélange de constructions déclaratives et impératives. Les transformations ATL sont unidirectionnelles, opérant sur des modèles source en lecture seule et produisant des modèles cibles en écriture seule. Pendant l'exécution d'une transformation, le modèle source peut être parcouru mais les modifications ne sont pas autorisées. Le modèle cible ne peut pas être parcouru. Une transformation bidirectionnelle est implémentée sous la forme de deux transformations: une pour chaque direction.

4.2.3 Moteur d'exécution conçu en J2EE

Le moteur d'exécution se compose de deux parties:

- Les interfaces Web développées en Java à l'aide du framework $JSF/Primefaces$. Chaque rôle peut utiliser ces interfaces pour visualiser les services contenus et leurs instances. L'interface Web est illustrée à la figure 4.6. Après la connexion

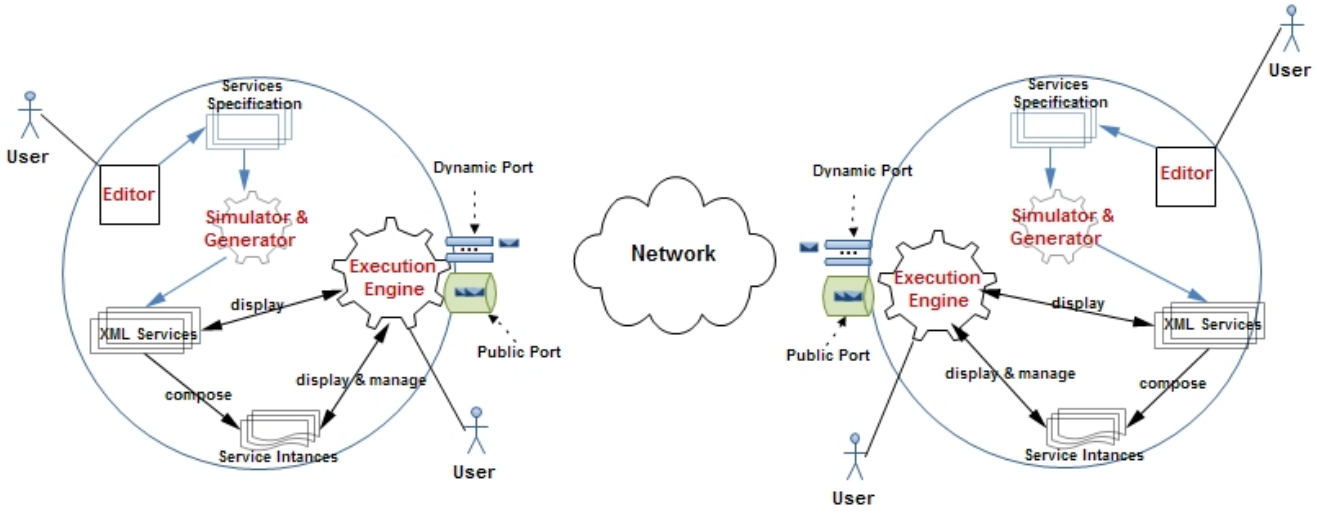

FIGURE 4.2: Le style architectural du moteur d'exécution est P2P. Le rôle échange des données via un middleware de type MOM.

d'un rôle, elle présente les services disponibles pour le rôle (en haut) et les instances de services en cours d'exécution (en bas).

- Un support de communication qui est un middleware de type Message Oriented Middleware (MOM) car les MOMs permettent des interactions entre les composants d'application dans un cadre faiblement couplé, asynchrone et fiable. À cet effet, JMS(Java Message Service) a été utilisé.

4.3 Style Architectural

Les services spécifiés dans GSLang sont regroupés dans des espaces utilisateurs qui communiquent entre eux (figure 4.2).

Comme nous détaillerons dans les prochaines sections, un pair contient un ensemble d'éléments :

- Éditeur;
- Moteur de transformation;
- Moteur d'exécution.

L'éditeur (*Editor*) permet aux concepteurs de mettre à jour les règles (les services *GSLang*). Il faut noter que chaque pair contient toute la spécification *GSLang* du système. Ceci facilitera l'étape de vérification. Lorsqu'une modification est validée, une mise à jour de la spécification est faite dans les autres pairs.

Le moteur de transformation (*simulator* & *generator*) transforme les services *GSLang* vers *Promela* pour la vérification et vers le système opérationnel (en XML) pour la validation. A l'exécution, les services XML sont composés dans des $Artfacts$ (instances de services).

Le moteur d'exécution (*Execution Engine*) suit l'évolution des instances de services. Il implémente la logique de la sémantique opérationnelle présentée à la section 2.4. Grâce au MOM, il crée des ports dynamiques à l'exécution pour suivre les requêtes entre les pairs.

4.4 Présentation des outils

A l'intérieur d'un pair, nous avons un ensemble d'éléments tels que l'éditeur des spécifications GSLang, le vérifieur et le moteur d'exécution.

1. le concepteur édite les services : le système vérifie que toutes les spécifications de services sont cohérentes ou que toute modification des services est cohérente. Les services spécifiés dans l'éditeur sont transformés en Promela pour la simulation;
2. le système raffine la spécification GSLang du pair vers le système opérationnel lorsque le point 1 est correctement effectué. Ici, les éléments comme le type des services (RESTfull ou Manuellement) et leur emplacement sont ajoutés. Les services spécifiés sont transformés en XML pour l'opérationnalisation;
3. le moteur d'exécution utilise un système opérationnel basé sur XML pour répondre aux besoins des utilisateurs en combinant dynamiquement différents services. Le moteur d'exécution gère les XML.

4.4.1 Éditeur

Nous avons développé une DSL pour le langage proposé en utilisant $Xtext$. Les plugins *Eclipse* obtenus permettent de faire un contrôle de syntaxe lors de l'édition

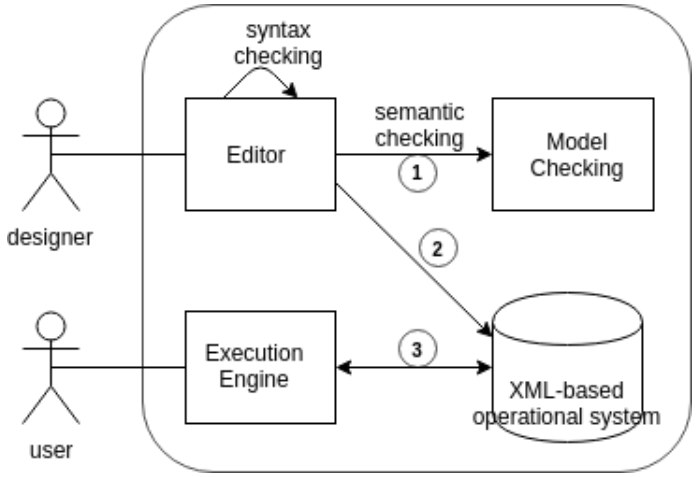

FIGURE 4.3: Framework de vérification et de validation

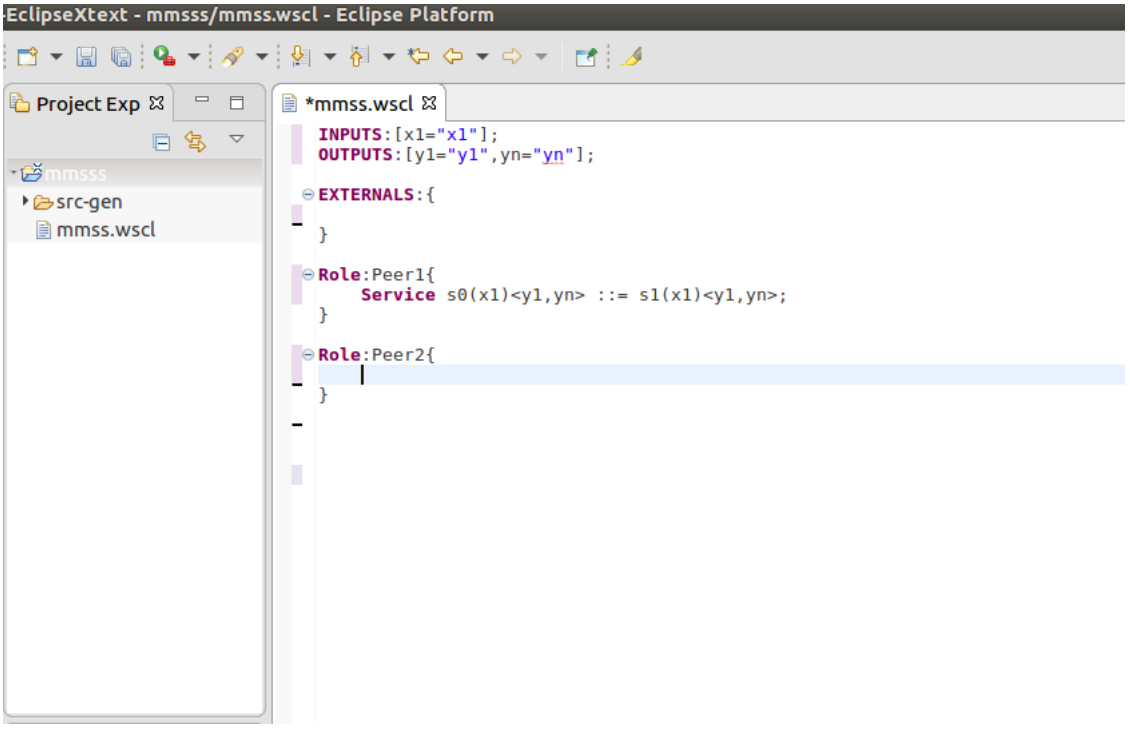

FIGURE 4.4: Interface principale de l'éditeur *GSLang*

des services et d'affiner la spécification *GSLang* vers le système opérationnel pour le rôle lorsque la vérification réussit. L'arbre de syntaxe abstraite basé sur *EMF* de la figure 3.4.1.2 (c'est le méta-modèle source présenté au chapitre précédent) permet de vérifier cette syntaxe.

L'interface principale de l'éditeur est présentée à la figure 4.4. Elle permet de définir :

1. des variables utilisées dans les paramètres des services

2. des services externes aux systèmes définis dans la rubrique *EXTERNALS*

3. des rôles définis dans chaque rubrique *ROLES*

4.4.2 Règle de Transformation

Moteur de transformation est écrit en ATL. Il prend en entrée les meta-modèles source figure 3.5 et cible figure 3.6. Les *helpers* sont d'abord présentés ensuite les différentes règles de transformation sont définies par exemple *Sytem2Model* transforme la classe Système du modèle source en la classe Model du modèle cible. De même la règle *Role2Process*, transforme un Rôle c'est à dire un pair en Process promela. Chaque service externe est aussi transformé en processus Promela.

De même pour obtenir les codes XML des services, des *helpers* ainsi qu'une *Query* ont été écrits.

```
create OUT : promela from IN : specService;


-- ------------------------------------------------------
-- ------------------------- HELPER ---------------------
-- ------------------------------------------------------

-- Retourner l'ensemble des roles du Systeme
helper def: getSystemRole(syst: specService!System) : Set(specService!Role) =
specService!Role.allInstances() -> select(e | e.system = syst)
;

-- Retourner l'ensemble des services simples d'un role
helper def: getSimpleServiceOfRole(rol1: specService!Role) : Set(specService!Simple) =
specService!Simple.allInstances() -> select(e | e.role = rol1)
;

-- Retourner l'ensemble des services composites d'un role
helper def: getCompositeServiceOfRole(rol2: specService!Role) : Set(specService!Composite) =
specService!Composite.allInstances() -> select(e | e.role = rol2)
;

-- Retourner l'ensemble des services d'un service composite
helper def: getServiceOfComposite(comp: specService!Composite) : Set(specService!Service) =
specService!Service.allInstances() -> select(e | e.composite = comp)
;

-- Retourner l'ensemble des parametres en entrees d'un services
helper def: getInputOfService(serv: specService!Simple) : specService!Input =
serv.parameters -> select(e | e.oclIsTypeOf(specService!Input)) -> first()
-- specService!Input.allInstances() -> select(e | e.service = serv)
```

```
;

-- Retourner l'ensemble des parametres en sorties d'un services
helper def: getOutputOfService(serv: specService!Simple) : specService!Output =
serv.parameters -> select(e | e.oclIsTypeOf(specService!Output)) -> first()
-- specService!Output.allInstances() -> select(e | e.service = serv)
;

-- Retourner l'ensemble des guards d'un services
helper def: getGuardOfService(serv: specService!Service) : Set(specService!Guard) =
specService!Guard.allInstances() -> select(e | e.service = serv)
;

helper def: getSystem(s : specService!Role) : specService!System =
s.system
;

helper def: getRoleee(s : specService!Service) : specService!Role =
s.role
;

-- ------------------------------------------------------------
-- -------------------------- END HELPER ----------------------
-- ------------------------------------------------------------


-- ------------------------------------------------------------
-- -------------------------- RULES ---------------------
-- ------------------------------------------------------------

rule Sytem2Model {
from syst: specService!System
to model : promela!Model (
name <- syst.name,
variables <-
Set {
syst.roles ->  collect (e | thisModule.resolveTemp(e, 'chan1')),
specService!Service.allInstances()  ->  collect(e | thisModule.resolveTemp(e, 'prim1')),
specService!Service.allInstances()  ->  collect(e | thisModule.resolveTemp(e, 'prim11')),
specService!Input.allInstances() -> collect(e | thisModule.resolveTemp(e, 'prim2')),
specService!Output.allInstances() -> collect(e | thisModule.resolveTemp(e, 'prim3'))
} ->flatten() ,

processes <- syst.roles -> collect(e |thisModule.resolveTemp(e, 'process1')) ->flatten()

)
}



rule Role2Process{
from rol3 : specService!Role
to chan1 : promela!Channel (
size <- 5,
type <- #byte,
```

```
typeName <- '{mtype,mtype,chan}',
name <- rol3.name+'pub',
model <- thisModule.resolveTemp(rol3.system, 'model')
),
process1 : promela!Process(
name <- rol3.name,
isActive <- true,
isMain <- false,
model <- thisModule.resolveTemp(rol3.system, 'model'),
variables <-  Set {rol3.services
-> select (e | e.isLHS)
-> collect (e | thisModule.resolveTemp(e, 'chan2')),
rol3.services
-> select (e | e.isLHS)
-> collect (e | thisModule.resolveTemp(e, 'chan22'))
},
abstractstatements <- Set{
rol3.services  ->  collect (e | thisModule.resolveTemp(e, 'simpleStat')),
rol3.services -> collect (a  | thisModule.resolveTemp(a, 'compositeStat'))
}
)
}

rule Simple2Primitive {
from serv1: specService!Simple
to prim1 : promela!Primitive (
type <- #byte,
name <- serv1.name,
process <- thisModule.resolveTemp(serv1.role, 'process1')
),

chan2 : promela!Channel (
size <- 5,
type <- #byte,
typeName <- '{mtype}',
name <- serv1.name+'priv',
process <- thisModule.resolveTemp(serv1.role, 'process1')
),
simpleStat : promela!SimpleStatement(
pport <- serv1.name+'pub',
id <- serv1.name,
input <- thisModule.getInputOfService(serv1).name, -- faire un control si c'est vide
priport <- serv1.name+'priv',
output <-  thisModule.getOutputOfService(serv1).name, -- faire un control si c'est vide
proc <- thisModule.resolveTemp(serv1.role, 'process1')
)

}

rule Composite2Primitive {
from serv1: specService!Composite
using {
-- coll : Set(specService!Simple) = Sequence{'a', 'b', 'c'};
}
to
```

```
prim11 : promela!Primitive (
type <- #byte,
name <- serv1.name,
process <- thisModule.resolveTemp(serv1.role, 'process1')
),

chan22 : promela!Channel (
size <- 5,
type <- #byte,
typeName <- '{mtype}',
name <- serv1.name+'priv',
process <- thisModule.resolveTemp(serv1.role, 'process1')
),

compositeStat : promela!CompositeStatement(
condition <- serv1.name+'conditionnnnnn',
abstractstatements <-  serv1.services
-- ->  select  (e | e.isSimple = false)
->  collect (e | thisModule.resolveTemp(e, 'simpleStat'))
,
proc <- thisModule.resolveTemp(serv1.role, 'process1')
)
}

rule Input2Primitive {
from input1: specService!Input
to prim2 : promela!Primitive (
type <- #byte,
name <- input1.name
)
}

rule Output2Primitive {
from output1: specService!Output
to prim3 : promela!Primitive (
type <- #byte,
name <- output1.name
)
}
```

4.4.3 Moteur d'exécution

Le moteur d'exécution a un aspect visuel (interface web) et un backend qui implémente les règles de la section 2.4 et est basé sur le MOM.

4.4.3.1 Interface Web

L'interface (figure 4.6) contient deux grandes parties:

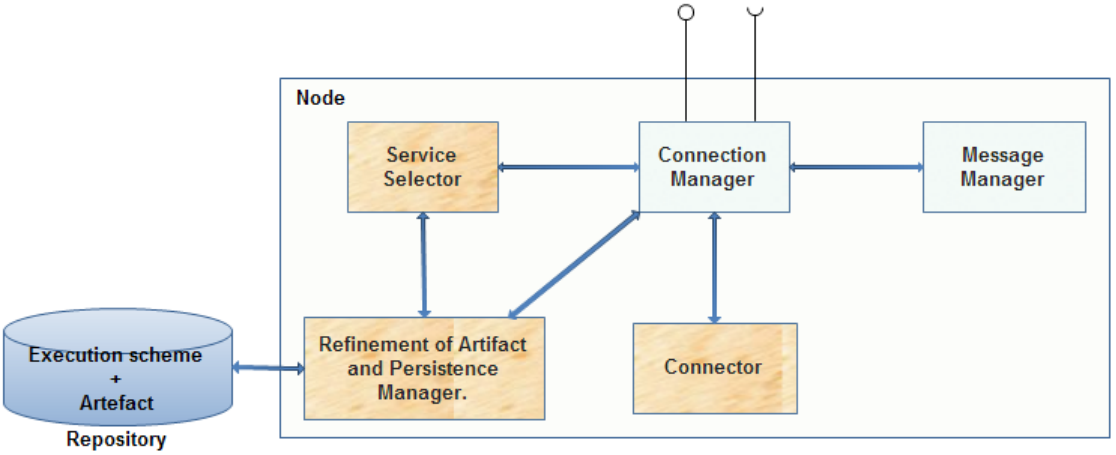

FIGURE 4.5: Les éléments du Backend

- une partie haute listant les services disponibles actuellement dans le pair. Sur la figure, on a la déclaration des services *checkDemand*, *preparingLogistic*, *rejectRequest*, *carRenting*, *generateMissionOrder* et *signMissionOrder*.
- une partie basse permettant de visualiser l'instance du service en cours d'exécution et les autres instances déjà exécutées. Sur la figure, l'instance du service *checkDemand* est en cours d'exécution. Sur chaque nœud de l'arborescence, nous pouvons visualiser les informations synthétisées et à synthétiser. Les nœuds rouges ne sont pas encore résolus et les noeuds noirs sont déjà résolus. Nous détaillerons sur un exemple dans la partie expérimentation.

4.4.3.2 Backend

Un conteneur est développé pour permettre l'exécution des services comme décrit. Il implémente la sémantique opérationnelle présentée à la section 2.4. Son rôle est de gérer les connexions pour composer les services sélectionnés au sein des artefacts, formater les entrées et sorties des messages d'un espace d'exécution et gérer la persistance des données. Les éléments constitutifs du conteneur sont présentés dans la figure 4.5:

a) *Connection Manager* gère les connexions créées lors de l'exécution d'un service. Il se connecte à d'autres services requis pour atteindre les résultats attendus.

b) *Message Manager* permet la construction de messages à envoyer et le formatage des messages entrants dans un pair (espace de travail).

c) *Service Selector* permet de choisir le service souhaité parmi ceux existant dans l'espace d'exécution.

d) *Refinement of Artifact* et *Persistence Manager* gèrent la persistance des services instanciés et la traçabilité de l'exécution d'un service instancié. Le raffinement consiste à résoudre des données définies de manière intentionnelle. L'artefact est une instance d'un service en cours d'exécution qui contient toutes les informations sur l'exécution du service de sa création jusqu'à son achèvement. Il s'agit d'une structure arborescente qui s'étend progressivement à mesure que le service composite est exécuté.

e) *Connectors* sont utilisés pour interagir avec des services distants en utilisant un style architectural bien connu spécifique tel que REST.

f) *Repository* contient des instances d'archivage des services en cours d'exécution ainsi que des artefacts et les services.

Au moment de l'exécution, le conteneur applique le ou les services sélectionnés. Le premier service sélectionné crée un artefact (pour surveiller la composition du service) dans l'espace utilisateur associé à ce service. Lorsque le service sélectionné a une partie droite, le conteneur continue avec les services de cette partie. Des services indépendants peuvent être exécutés en parallèle. L'exécution d'un service par le conteneur consiste à choisir le bon service, à créer l'instance associée à ce service et à la composer dans les artefacts associés. L'utilisateur peut modifier, créer ou supprimer les services présents dans le schéma d'exécution, ce qui peut avoir un impact sur l'exécution des services composites en cours. Ainsi, deux artefacts associés au même service peuvent être différents.

4.5 Expérimentation

4.5.1 Description détaillée du cas

Considérons un système de gestion de mission (MMS) simplifié utilisé par une organisation pour organiser les voyages d'affaires de ses employés. Cet exemple implique trois acteurs: l'employé, la secrétaire et le chef de département (HOD). Il y a également des services fournis par des structures externes. L'employé demande un ordre de mission en remplissant un formulaire qui est soumis au secrétaire. La secrétaire

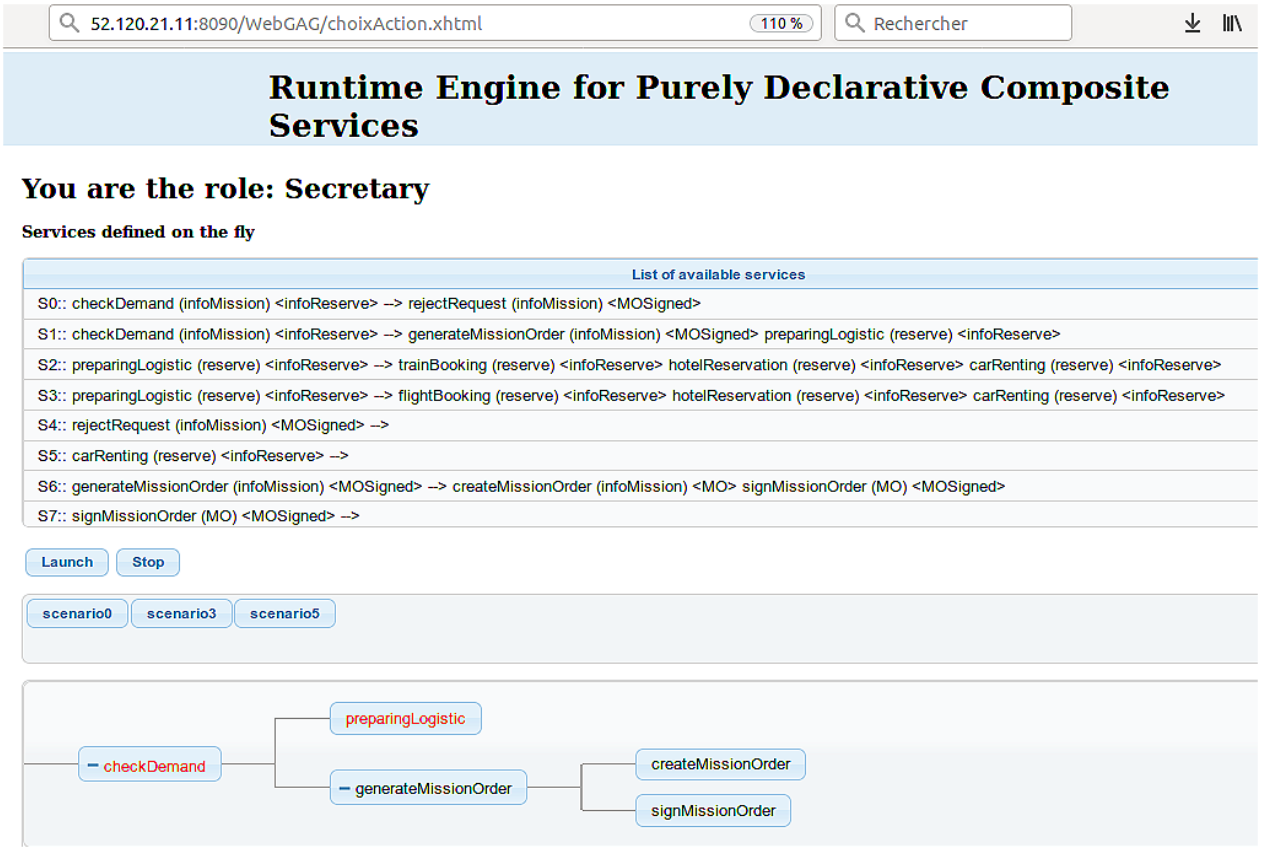

FIGURE 4.6: Interface principale du moteur d'exécution

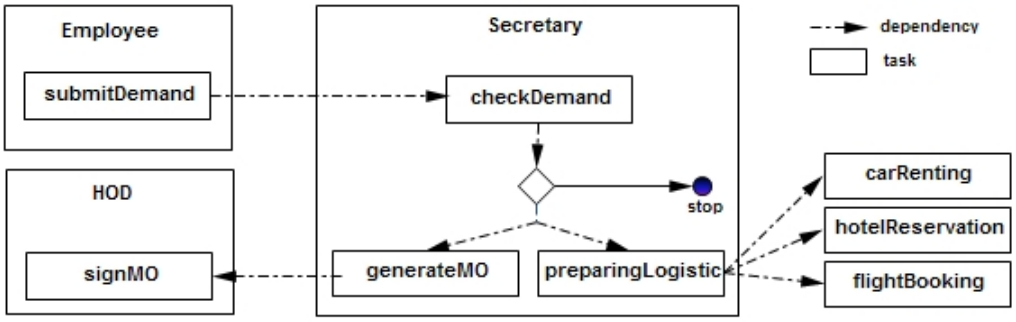

FIGURE 4.7: Structure du système de gestion de mission (MMS).

vérifie la validité de la demande, si elle est approuvée, le formulaire est transmis au Directeur du département (HOD) de l'employé. Pendant ce temps, la secrétaire prépare la logistique. Cette dernière action est composée de services externes. Il utilise le système de réservation de vol (*Flight Booking*) ou de train (*Train Booking*), d'hôtel(*Hotel Reservation*) et de location de voiture (*Car Rent*). La figure 4.7 montre la vue d'ensemble de l'exemple.

Dans les langages existants de composition de services, cet exemple sera converti en un fichier au format XML par exemple et soumis au moteur d'exécution. On peut dès lors se demander ce qui se passerait si de nouvelles exigences surgissaient. Par exemple, le service *preparingLogistic* peut nécessiter pour certains cas l'appel d'un service pour le Visa, pour d'autres cas la non réservation d'une Voiture, pour d'autres cas encore la réservation du train et non de l'avion. Dans de tels systèmes, tout ne peut pas être prévu à l'avance. Étant donné que ces modifications influencent la vue de la Secrétaire, le MMS est distribué afin de permettre à chaque acteur de faire des changements dans son espace de travail ainsi qu'une définition à la volée des différents services. Dans la section suivante, nous présentons comment l'approche explicitée dans cette thèse est utilisée pour permettre une flexibilité et une adaptation dynamique.

4.5.2 Mise en œuvre du cas

4.5.2.1 Édition des services

La toute première étape est l'initialisation du système. Pour cela, le concepteur spécifie le système dans l'éditeur. La version actuelle de l'éditeur est textuelle. Elle offre la possibilité d'éditer et de vérifier la syntaxe des services. Elle permet de spécifier :

- les variables globales dans les rubriques $INPUTS$ et $OUTPUTS$;
- les services externes au système par exemple les services de réservation d'hôtel, de vol, de train ou de location de voiture;
- les espaces utilisateurs (les rôles) par exemple Employé, Sécretaire ou Director (HoD).

La figure 4.8 montre l'édition des règles associées à l'exemple de la section précédente. Chaque rôle est représenté, pour chacun, nous avons les services associés:

- Dans le rôle *Employé*, nous avons le service *submitDemand*. Il prend en entrée $infoM$ qui représente les informations sur la mission et attend *mos* et $infoR$ représentant respectivement l'ordre de mission signé et les informations sur la réservation. Il dépend de *checkDemand* implémenté chez la secrétaire.

FIGURE 4.8: Modélisation dans l'éditeur

- Dans le rôle *Secrétaire*, nous avons un ensemble de services. *checkDemand* qui a deux variantes dans la spécification actuelle: l'une conduisant au rejet de la demande(*checkDemandReject*) et l'autre permettant de générer l'ordre de mission (*generateMissionOrder*) et de préparer la logistique (*preparingLogistic*). La génération de l'ordre de mission consiste à créer l'ordre de mission (*createMissionOrder*) et son envoi (*signMissionOrder*) chez le Director pour signature. La préparation de la logistique a deux variantes. La première variante permet de réserver le vol (*fightBooking*), l'hôtel (*hotelReservation*) et la location de véhicule (*carRenting*). Ces derniers sont des services externes. La seconde variante permet de réservation du train, du vol et la location des véhicules.

- Dans le rôle *Director*, nous avons un service *signMissionOrder* lui permettant de signer un ordre de mission, il prend en entrée *moIn* qui est l'ordre de mission et retourne *mos* qui est l'ordre de mission signé

Cette spécification initiale est présente intégralement dans chaque rôle. Si elle est modifiée par un rôle, elle doit être mise à jour sur tous les autres rôles.

4.5.2.2 Vérification et génération du système opérationnel

Une fois les services définis, nous appliquons le programme de transformation de la section 4.4.2. Ce programme génère le code PROMELA de la spécification qui est le suivant :

a. Transformation pour la vérification

```
#define N 3

mtype = {prepareLog1, prepareLog2, flightB, hotelR, rentCar, r1, seat,
         room, car, im, infoReserv, NULL, MOSigned, signMO1, trainB,
         submit1, generateMO1, createMO1, check1, check2, reject1, MO,
         MOSignedRoomCarSeat};

// define data structure
typedef reserve{
        mtype otherInfo;
        byte distance;
}

typedef infoMission1{
        mtype employeeInfo;
        reserve r;
}


// Declare the public ports
chan secret = [N] of {mtype,infoMission1,chan};
chan employe = [N] of {mtype,infoMission1,chan};
chan generateMO = [N] of {mtype,mtype,chan};
chan direct = [N] of {mtype,mtype,chan};
chan hotel = [N] of {mtype,reserve,chan};
chan carRental = [N] of {mtype,reserve,chan};
chan fightBook = [N] of {mtype,reserve,chan};
chan trainBook = [N] of {mtype,reserve,chan};

// Employee space
active proctype employee(){
        mtype a,s,x;

        // private ports
        chan p1 = [N] of {mtype};
        chan p = [N] of {mtype};

        //init infoMission
        infoMission1 im1;
        im1.r.otherInfo = r1;
        im1.r.distance = 6;
        im1.employeeInfo = im;
```

```
        //Test of submitDemand
        employe!submit1,im1,p;

        begin: employe?s,im1,p;
        if
        :: (s==submit1) ->
        atomic{
                secret!check1,im1,p1;
                p1?a;

                (a!=0) ->
                p!a; ////result = a;
                p?x;
                goto begin;
        }

        fi;
}

inline reject(){
        a = NULL;
}

inline createMO(){
        a = MO;
}

inline generateMOs(s){
        if
        :: (s==generateMO1) -> createMO();

            direct!signMO1,a,p4;
                p4?b;
        fi;
}

inline preparess(s1){
        if
        :: (s1==prepareLog1) ->  fightBook!flightB,r,p1a;
                        p1a?a1;
                        hotel!hotelR,r,p2a;
                        p2a?b1;
                        carRental!rentCar,r,p3a;
                        p3a?c1;
        :: (s1==prepareLog2) ->
                        trainBook!trainB,r,p1a;
                        p1a?a1;
                        hotel!hotelR,r,p2a;
                        p2a?b1;
                        carRental!rentCar,r,p3a;
                        p3a?c1;
        fi;
}
```

```
// hotel Space
active proctype hotl(){
        mtype id;
        reserve  r;
        chan p = [N] of {mtype};

        if
        :: hotel?id,r,p ->  p!room;
        fi;
}

// secretariat Space
active proctype secretariat(){
        //private ports
        chan p1 = [N] of {mtype};
        chan p2 = [N] of {mtype};
        chan p = [N] of {mtype};
        chan p4 = [N] of {mtype};
        chan pa = [N] of {mtype};
        chan p1a = [N] of {mtype};
        chan p2a = [N] of {mtype};
        chan p3a = [N] of {mtype};
        chan p4a = [N] of {mtype};
        chan p5a = [N] of {mtype};

        byte s;
        mtype a,b;
        mtype a1,b1,c1;//d1;
        infoMission1 im1;
        reserve  r;

        begin: secret?s,im1,p;

        r.otherInfo = im1.r.otherInfo;
        r.distance = im1.r.distance;

        if
        :: (s==check1) ->
                atomic{
                        generateMOs(generateMO1);
                        preparess(prepareLog1);

                        (a!=0 && b!=0 ) ->
                                        p!MOSignedRoomCarSeat;
                        goto begin;
                };
        :: (s==check2) ->
        atomic{
                reject();
                        (a!=0 ) -> p!NULL;
                goto begin;
                };
                :: (s==reject1) ->
                        p!NULL;
                :: (s==generateMO1) ->
```

```
                atomic{
                        p!a;
                goto begin;
        };
        :: (s==createMO1) ->
                atomic{
                        createMO();
                (a!=0) -> p!a;

                goto begin;
        };
        :: preparess(s); ->
                (a1!=0 && b1!=0 && c1 !=0) -> p!MOSignedRoomCarSeat;
                goto begin;
        fi
}

// fightBooking Space
active proctype fightBooking(){
        chan p = [N] of {mtype};
        reserve  r;
        mtype s;

        begin: fightBook?s,r,p;
        if
        :: (s==flightB) ->
                p!seat;
        fi;
}

// trainBooking Space
active proctype trainBooking(){
chan p = [N] of {mtype};
reserve  r;
mtype s;

begin: trainBook?s,r,p;
        if
        :: (s==trainB) ->
                p!seat;
        fi;
}

// director Space
active proctype director(){
        mtype id,x;
        chan p = [N] of {mtype};

        if
                :: direct?id,x,p -> p!MOSigned;
        fi;
}

// car Rental Space
active proctype carR(){
```

```
            mtype id;
            chan p = [N] of {mtype};
            reserve  r;

            if
            :: carRental?id,r,p -> p!car;
            fi;
    }
```

Chaque rôle ainsi que les services externes sont des processus PROMELA. Ce code PROMELA est simulé grâce à l'outil SPIN afin de vérifier si la spécification termine.

b. Vérification/Simulation

Le code obtenu à partir de la section 4.5.2.2.a est simulé dans l'environnement SPIN. L'exécution d'un service se fait par le choix des services dans les différents pairs (ici le processus). Nous vérifions d'abord que le service *submitDemand* se termine. La figure 4.9 montre sa trace d'exécution. Le processus *employee* soumet la demande de mission au processus *secretariat* via son port public (17 sur la figure 4.9) et attend la réponse sur le port privé (1 sur la figure 4.9). Le processus de secrétariat sous la valeur de infoMission (*im*) créera l'Ordre Mission, le fera signer par le directeur (le processus de secrétariat utilise le port privé 7, envoie une demande sur le port public du directeur et attend une réponse sur le port 7) et enfin préparer la logistique. Le choix du service logistique dépend de la valeur de la distance. Ici, le processus logistique menant au vol est choisi.Les processus *fightBooking*, *hotel* et *carR* seront appelés et exécutés et la réponse sera retournée via les canaux privés créés lors de leurs appels (port privé 9, 10 et 11 respectivement pour fightBooking, hotel et carRental). Enfin, le processus de secrétariat renvoie la réponse sur le port privé (1 sur la figure 4.9)créé lors de la requête.

La Figure 4.10 affiche une demande rejetée. Le processus secrétariat a retourné *NULL* car la demande de mission n'a pas été acceptée.

La Figure 4.11 montre un scénario où certains processus ne sont pas disponibles, en particulier le processus *fightBooking*.

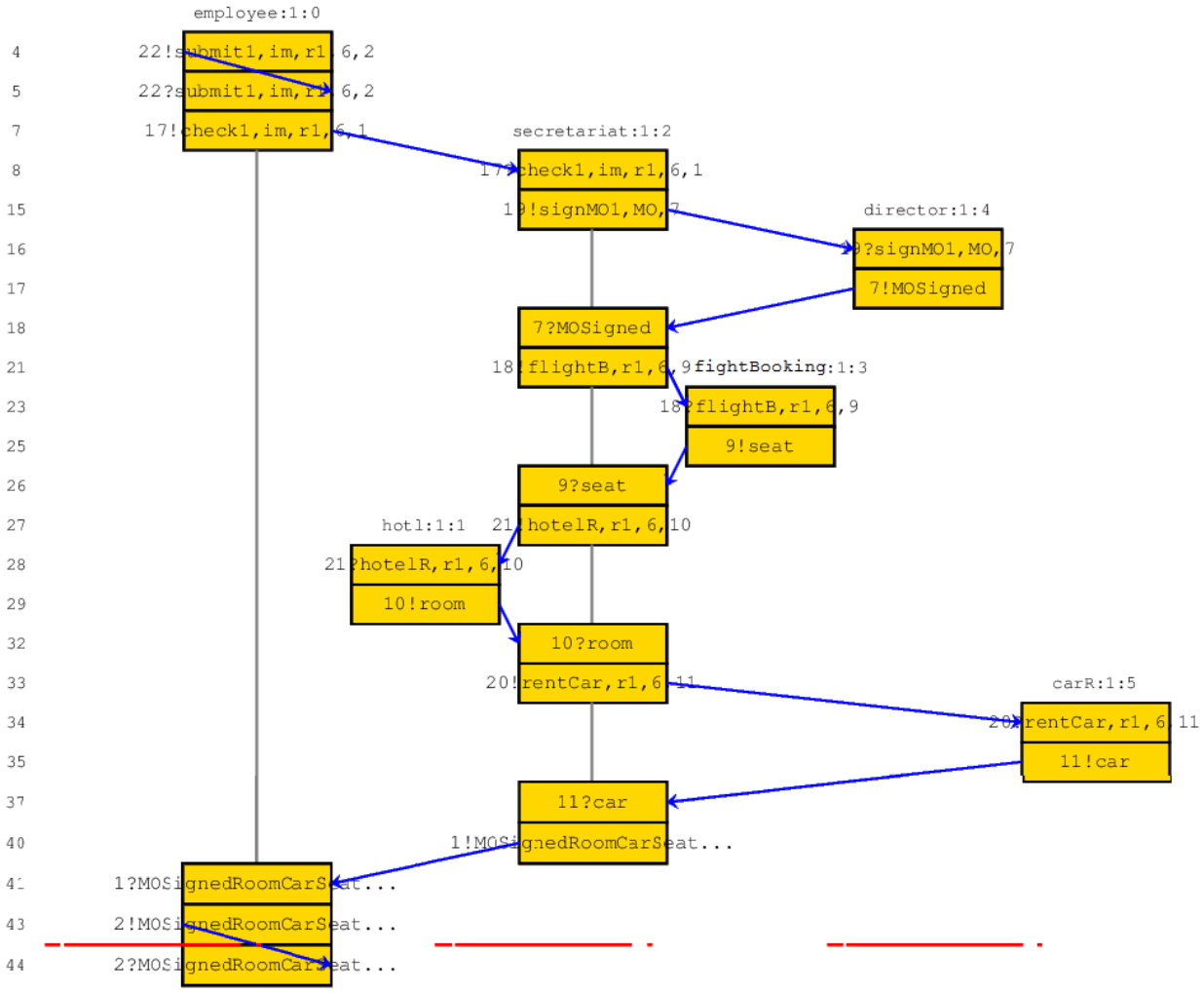

FIGURE 4.9: Une exécution possible

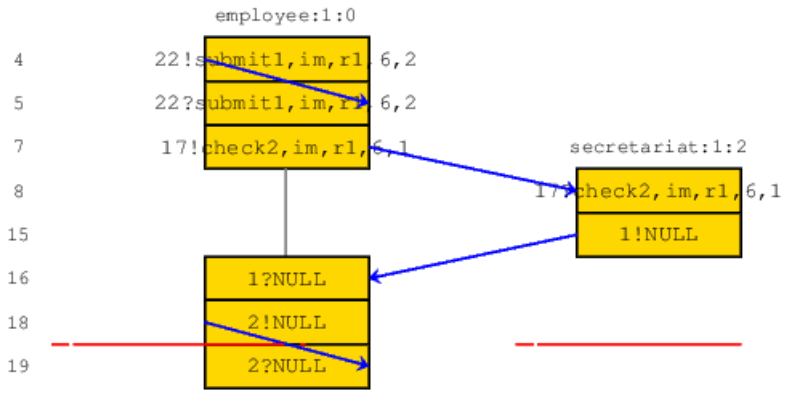

FIGURE 4.10: La demande rejetée

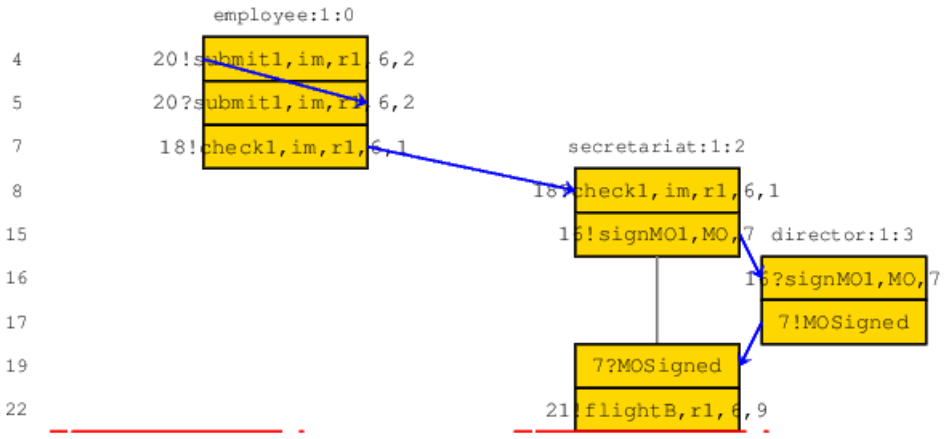

FIGURE 4.11: Le processus *fightBooking* n'est pas disponible

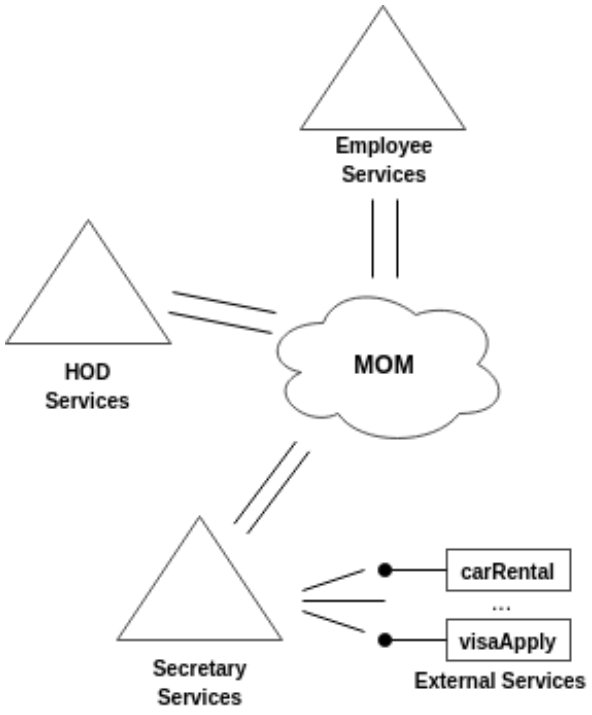

FIGURE 4.12: Le style architectural du moteur d'exécution

4.5.2.3 Exécution

La spécification est distribuée dans les différents espaces d'exécution selon un style architectural P2P au dessus de (MOM) comme présentée dans la figure 4.12.

Les figures 4.13, 4.14 et 4.15 présentent les interfaces principales respectivement des espaces d'exécution de l'employé, du secrétariat et du directeur. C'est une capture d'écran de la partie haute de chaque interface. Chez l'employé, nous avons le service *submitDemand*. Chez le secrétariat, nous avons actuellement les services suivants :

FIGURE 4.13: Présentation de l'espace de l'Employé.

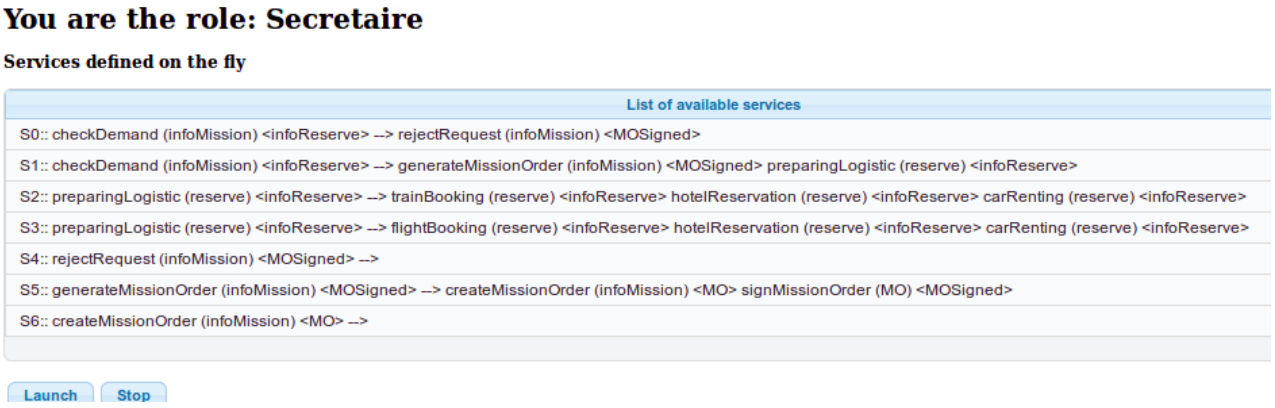

FIGURE 4.14: Présentation de l'espace du Secrétariat.

deux variantes pour *checkDemand*, deux variantes pour *preparingLogistic*, les services *rejectRequest*, *generateMissionOrder* et *createMissionOrder*. Et enfin chez le directeur, nous avons le service *signMissionOrder*.

Lorsque l'employé décide de faire une mission, il se connecte et choisit *submitDemand*. Ensuite, il définit le paramètre infoMission (figure4.16). Le moteur d'exécution se charge de faire appel à *checkDemand* via le MOM et instancie *submitDemand*. Une arborescence est créée (figure4.17) dans la partie basse permettant de suivre l'évolution du service instancié. Actuellement tout les nœuds sont rouges c'est-à-dire non résolus.

Une notification est envoyée au Secrétariat qui décide en fonction des informations reçues d'appliquer une action(c'est-à-dire choisir un service dans son espace). Considérons qu'il décide de choisir S_0 (figure 4.18), le service *checkDemand* est instancié (figure

FIGURE 4.15: Présentation de l'espace du Directeur.

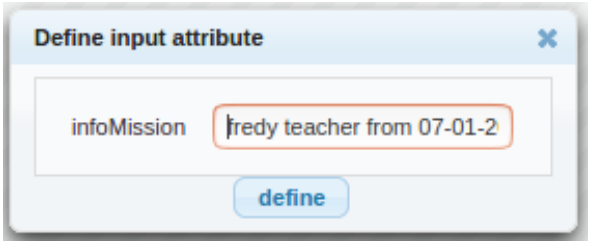

FIGURE 4.16: L'employé définit l'input de submitDemand.

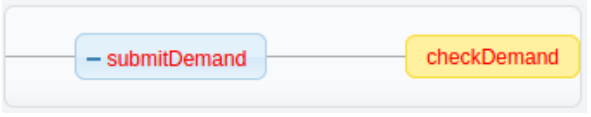

FIGURE 4.17: Le service submitDemand est instancié.

4.19) et *generateMissionOrder* sera automatiquement instancié étant donné que le paramètre en entrée est déjà défini.

La création de l'ordre de mission est faite (*createMissionOrder*) en renseignant le paramètre *MO* (figure 4.20). L'instance de ce service est ensuite résolue et l'appel du service *signMissionOrder* du Director est fait.

L'espace du directeur est notifié de cet appel. Il peut visualiser le service appelé et les paramètres reçus (figure 4.21). Il décide de signer l'order de mission (figure 4.22) matérialiser par la définition du paramètre *MOSigned*. Cette action permet de répondre au Secrétariat qui peut préparer la logistique.

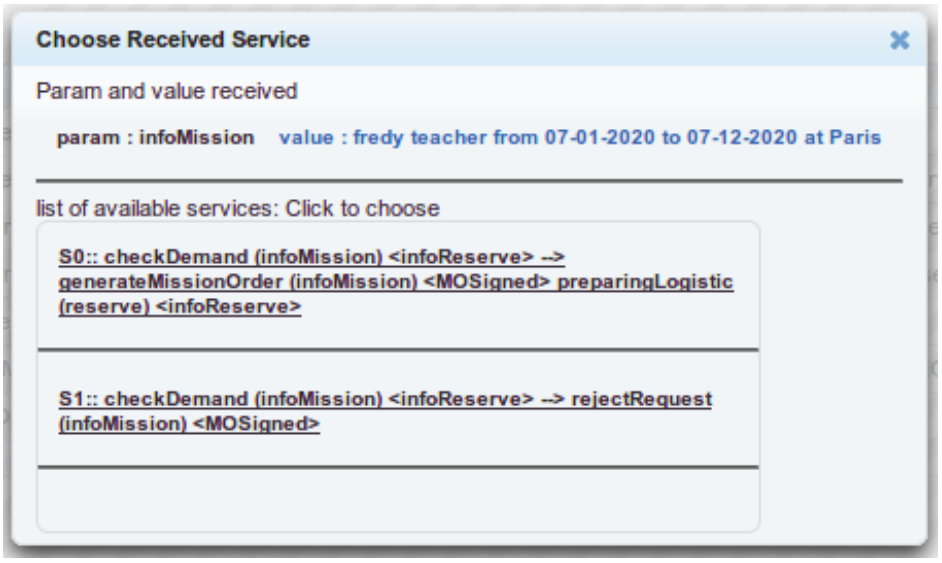

FIGURE 4.18: Le secrétariat choisit S_0.

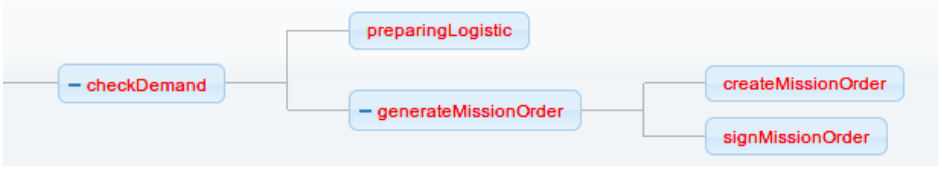

FIGURE 4.19: Le service checkDemand est instancié.

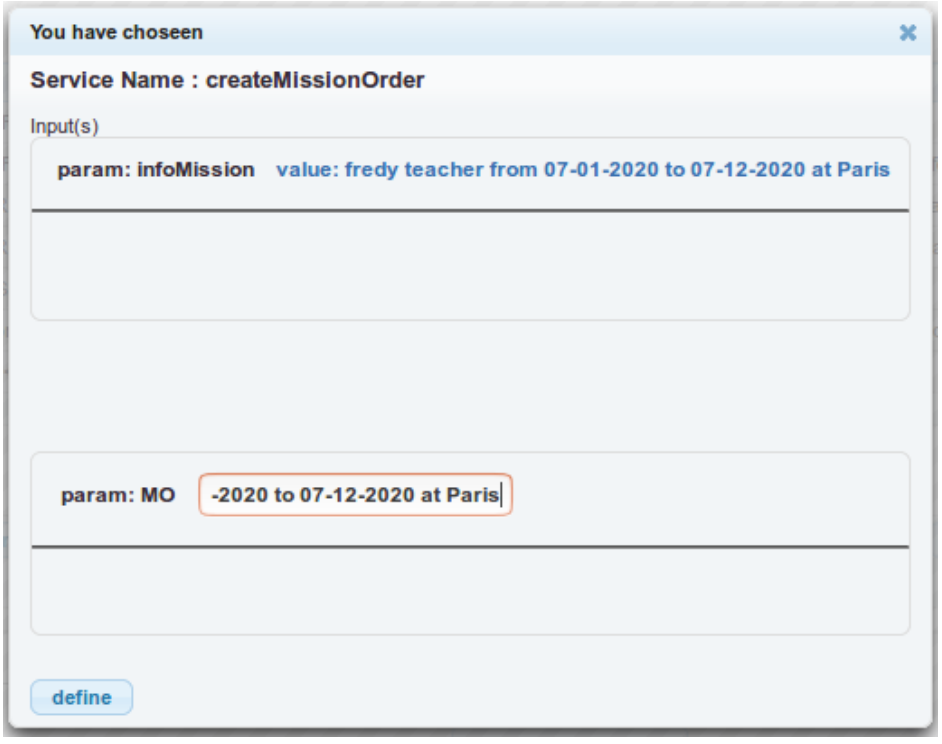

FIGURE 4.20: Création de l'ordre de mission.

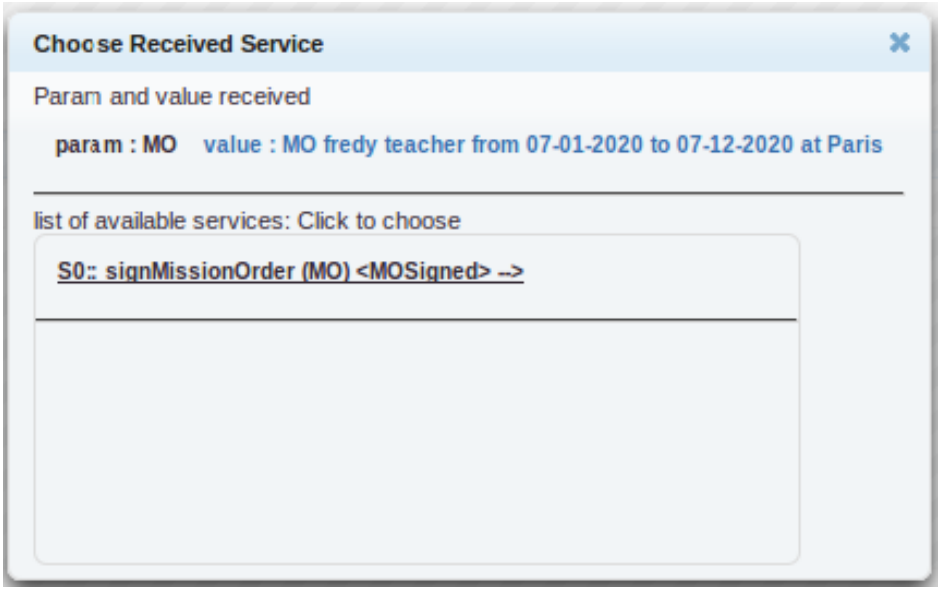

FIGURE 4.21: Information disponible chez le Director.

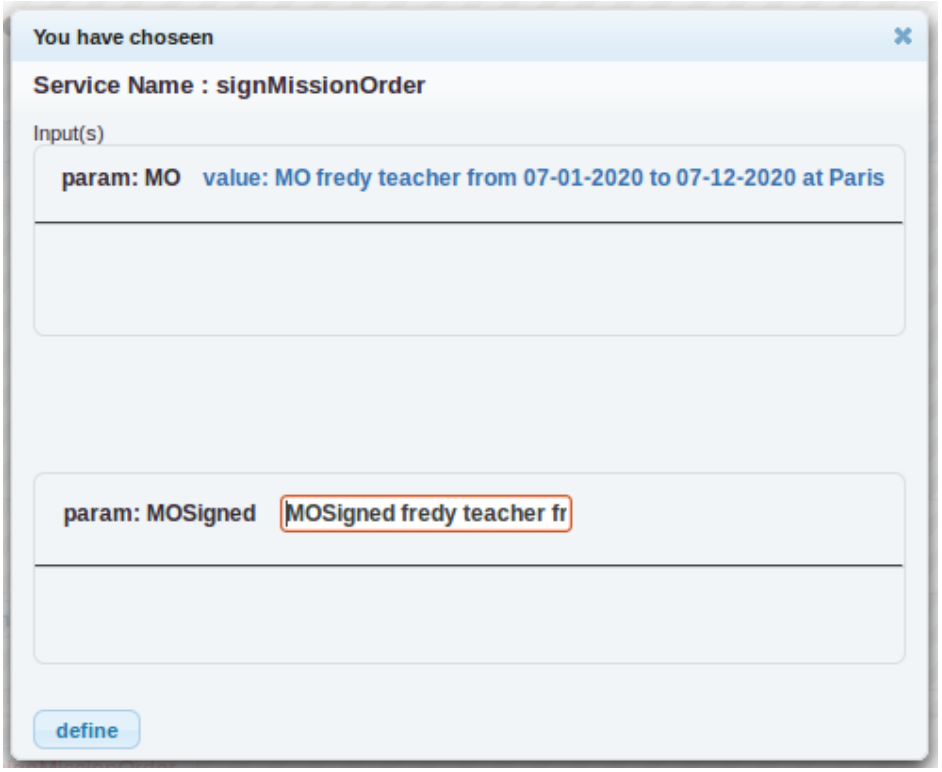

FIGURE 4.22: Le paramètre *signedMO* défini.

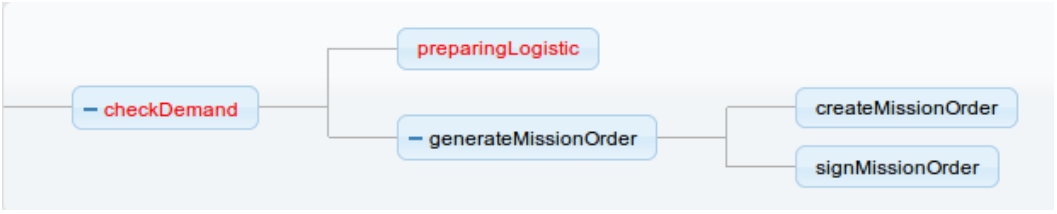

FIGURE 4.23: Évolution du service *checkDemand* : le service *generateMissionOrder* résolu

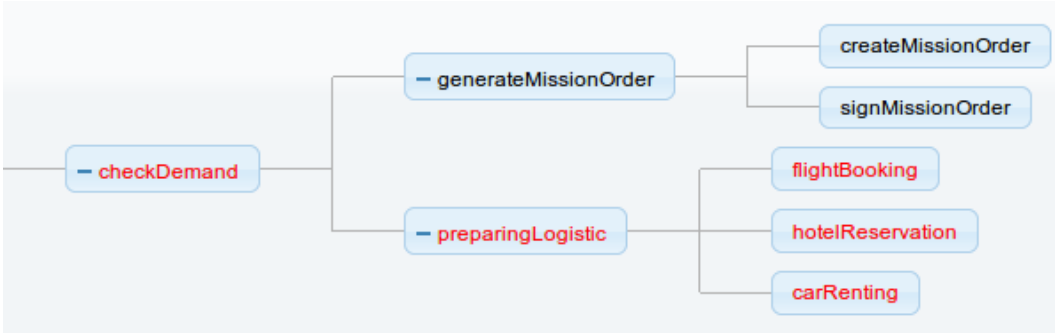

FIGURE 4.24: Evolution de checkDemand : le service *preparingLogistic* instancié

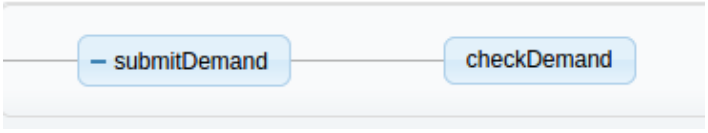

FIGURE 4.25: Le service *SubmitDemand* résolu.

A ce stade, dans l'espace du Secrétariat, les nœuds *generateMissionOrder*, *createMissionOrder* et *signMissionOrder* de l'instance du service *checkDemand* sont résolus comme présentés sur la figure 4.23.

Le secrétariat peut dès lors préparer la logistique en choisissant parmi les services *preparingLogistic* disponibles. Après le choix du service, l'instance de *preparingLogistic* est créée et l'instance de *checkDemand* est raffinée (figure 4.24). Les services *flightBooking*, *hotelReservation* et *carRenting* sont enfin appelés.

Finalement, la réponse est envoyée chez l'employé. On peut donc avoir l'instance de *submitDemand* avec des noeuds tous résolus (figure 4.25).

Au cours de l'exécution des services, les spécifications peuvent changer, dans ce cas, les étapes de vérifications et de raffinement doivent être effectuées afin de mettre à jour les schémas d'exécution. Ceci est possible car les spécifications sont définies à la volée (de manière déclarative).

4.5.3 Discussion

Plusieurs études se sont intéressées à la composition des services et plus récemment des langages déclaratifs ont été proposés afin de fournir plus de flexibilité et d'adaptation lors de l'exécution des services. Dans cette partie, la chorégraphie (architecture pair à pair) ainsi que le paradigme déclaratif de la composition des services sont discutés. Nous terminons en mettant en évidence le *data-driven* dans la composition des services.

4.5.3.1 Style architectural

Comme alternative aux langages et notations traditionnelles pour les compositions de services telles que BPEL [90] et BPMN [91], d'autres langages, comme JOpera [92] et Jolie [93] ont été proposé. Néanmoins, ils privilégient l'orchestration, un point de vue unique pour la composition du service. Etant de type orchestration, ils ne permettent pas d'envisager les collaborations de pair à pair. Notre langage est plutôt un type chorégraphique.

Quelques approches se focalisent sur la chorégraphie pour la composition de service. Inspirée des standards WS-CDL [29], au cours des dernières décennies, la chorégraphie a été étudiée pour soutenir un nouveau paradigme de programmation appelé programmation chorégraphique [94, 95]. Dans la programmation chorégraphique, le programmeur utilise la chorégraphie pour programmer les systèmes de service, puis un compilateur est utilisé pour générer automatiquement des implémentations conformes. Cela donne une méthode de précision par construction, garantissant des propriétés importantes telles que la liberté de blocage et l'absence d'erreurs de communication. Cet aspect de la vérification préalable au déploiement est important dans la collaboration entre les services, c'est pourquoi dans notre langage, nous transformons nos spécifications en promela. Néanmoins, la programmation chorégraphique permet une description des interactions entre les différents sites. Autrement dit, les interactions sont connues a priori ce que nous ne voulons pas. Il est difficile de modifier la chorégraphie en cours d'exécution lorsque des services sont ajoutés ou supprimés selon les besoins du concepteur.

Les approches telles que BPEL4Chor [10] ont tendance à ajouter la chorégraphie dans le style d'orchestration. Néanmoins, Elle s'appuie sur BPEL, qui a été critiqué pour son caractère rigide.

4.5.3.2 Approches déclaratives

Plusieurs langages ([92, 93, 96, 97]) ont été proposés; bien que plus faciles à utiliser et souvent plus expressifs que BPEL et BPMN [91], ils ne s'écartent pas du paradigme impératif et, par conséquent, partagent avec eux les mêmes limites qui ont motivé notre travail.

Pour fournir de la flexibilité par changement [24], les langages déclaratifs ont émergé. Parmi eux, des approches déclaratives telles que Declare [98]. Declare est le plus proche de notre travail compte tenu de son niveau d'abstraction. Dans Declare, les compositions de services sont définies *comme un ensemble d'actions et les contraintes qui les affectent*. Les actions et les contraintes sont modélisées, tandis que les contraintes ont une sémantique formelle donnée en logique temporelle linéaire (LTL).

Il existe plusieurs différences entre Declare et notre approche: Tout d'abord, Declare se concentre sur la modélisation des orchestrations de services pour prendre en charge l'audit et la surveillance tandis que notre langage se concentre sur la modélisation des chorégraphies. En outre, Declare se concentre sur la spécification et la vérification et ne fournit pas de mécanismes spécifiques pour gérer les échecs d'exécution.

GO-BPMN [11] est un autre langage déclaratif, conçu comme une extension orientée objectif pour le BPMN traditionnel. Dans GO-BPMN, les processus métier sont définis comme une hiérarchie d'objectifs et de sous-objectifs. Plusieurs plans BPMN sont attachés aux objectifs «feuille». Une fois exécuté, ils atteignent l'objectif associé. Ces plans peuvent être alternatifs ou ils peuvent être explicitement associés à des conditions spécifiques via des expressions de garde basées sur le contexte d'exécution. Bien que cette approche tente également de séparer les instructions déclaratives de la manière dont elles peuvent être accomplies, les plans alternatifs pour atteindre un objectif doivent être explicitement conçus par l'architecte de service et sont explicitement attachés à leurs objectifs. Le moteur ne décide pas automatiquement de la façon dont les plans sont construits ou remplacés; il choisit simplement entre les options données pour chaque objectif spécifique, et il le fait au moment de l'appel du service. GO-BPMN s'appuie sur BPMN connu pour être rigide au changement. Dans notre approche pour une instance d'un composite donné, nous pouvons modifier la structure des services non appliqués ce qui permet de changer sa manière de s'exécuter.

SelfMotion [97] est un langage déclaratif qui utilise des actions abstraites qui spécifient les opérations primitives qui peuvent être exécutées pour atteindre l'objectif; Actions

concrètes, une ou plusieurs pour chaque action abstraite, qui les mappent aux extraits de code exécutables qui les implémentent. Comme notre approche, il gère l'adaptation des services composites à l'exécution. Néanmoins, il ne vérifie pas afin d'éviter certains dysfonctionnements d'exécution car il se concentre davantage sur la mise en œuvre.

Concernant SELF-SERV [3], un résultat majeur du projet a été un système prototype dans lequel les services Web sont déclarativement composés. Dans SELF-SERV, le modèle de processus sous-jacent à un service composite est spécifié comme un diagramme d'états dont les états sont étiquetés avec des appels aux services Web et dont les transitions sont étiquetées avec des événements, des conditions et des opérations d'affectation de variables. Les diagrammes d'états possèdent une sémantique formelle et offrent la plupart des constructions trouvées dans les langages de modélisation de processus contemporains (séquence, branchement, boucles structurées, threads simultanés, synchronisation entre les threads, etc.). Cela garantit que les mécanismes de composition des services et les techniques d'orchestration développés dans le cadre du projet SELF-SERV peuvent être adaptés à d'autres langages de modélisation de processus pour les services Web tels que BPEL4WS et WSCI. Bien que fonctionnant dans un environnement pair à pair et prenant explicitement en compte les rôles qui sont l'une des caractéristiques de notre approche, il a basé sa conception sur des paradigmes impératifs héritant ainsi de toutes ses lacunes.

4.5.3.3 Approches orientées données

Les données sont l'un des points importants, elles permettent de préciser comment les sorties dérivent des entrées [21]. L'opérationnalisation du langage proposé implique une approche pilotée par les données puisque les modèles issus du langage ici défini sont pilotés par les données. Cette vision est proche de Active XML [99] où les services distants sont insérés dans des documents XML. Ces services, appelés données intentionnelles, sont activés lors du chargement du document XML. Dans notre approche, les artefacts (instance de services) contiennent les données concrètes qui représentent le service déjà exécuté et les données intentionnelles qui représentent les services en attente qui pourraient être appliqués. Néanmoins, Active XML est dédié à l'intégration des données plutôt qu'à la composition des services.

Les Mashups [100] rassemblent les services provenant de nombreuses sources sur une seule page Web. Certains inconvénients des mashups sont:

1. Les développeurs SOA doivent généralement consacrer des efforts importants à maîtriser de nombreuses technologies SOA, par exemple, BPEL, WSDL, SCA (Service Component Architecture), ainsi que des outils, par exemple, des outils IDE au moment de la conception et des serveurs middleware d'exécution (serveur SCA ou BPEL). serveur). La plupart de ces outils nécessitent des investissements majeurs sur l'infrastructure matérielle et logicielle;

2. Ces technologies ne peuvent pas prendre en charge la personnalisation de la composition des services à la volée car le processus de composition des services (conception, développement et tests) est généralement effectué dans l'IDE d'abord en fonction des besoins du client, puis déployé sur le serveur d'exécution. Après le déploiement, la logique de composition n'est pas facile à personnaliser en fonction des changements des exigences de service composite, car cela implique de revenir aux phases précédentes du processus de développement.

4.6 Conclusion

Dans ce chapitre, nous avons présenté l'environnement logiciel permettant l'exécution de notre langage. Nous avons tout d'abord présenté l'ensemble des outils conçus tels que:

- un éditeur XText permettant la spécification des services GSLang.
- par l'entremise d'ATL, nous avons écris les règles de transformation de modèles permettant d'une part de vérifier/simuler les services GSLang et d'autres par de générer les fichiers XML pour l'opérationalisation.
- un moteur d'exécution présentant une interface Web et un Backend.

Ensuite, nous avons expérimenté notre approche sur un système simplifié de mise en mission dans une organisation. Ce système a été présenté ainsi que sa modélisation dans l'éditeur, sa transformation vers Promela afin de vérifier notamment la propriété de terminaison et son exécution dans le moteur développé. Enfin une discussion a été réalisée.

.

Conclusion et Perspectives

Notre objectif dans cette thèse était de proposer un langage déclaratif pour la composition de services et un environnement permettant de le mettre en œuvre. Après avoir étudié les langages de la composition de services, nous avons constaté que la plupart d'entre eux préconise des blocs structurés pour la construction du composite et sont impératifs. Les langages impératifs proposent des éléments pour décrire explicitement comment le service composite est mis sur pied. Par conséquent, pour prendre en compte de nouveaux besoins c'est-à-dire l'ajout, la modification ou le retrait des services dans une composition il faut opérer statiquement. A côté de ces langages impératifs, les langages déclaratifs ont été proposés d'une part. Certains étendent les standards impératifs pour apporter de la flexibilité dans la composition de services. D'autres part, le paradigme de règles est utilisé pour spécifier une approche déclarative. Il présente les propriétés de *flexibilité*, d'*adaptabilité*, de *ré-utilisabilité* et de *sémantique formelle*.

1.1 Synthèse de la Recherche

- Nous avons proposé un formalisme de spécification de services qui utilise des productions d'une grammaire hors contexte. Cette spécification dite intentionnelle est décrite sous forme de règles constituées à droite (RHS), de la description des services concourant à résoudre le service à gauche (LHS). Dans cette approche, le schéma de composition reste aussi abstrait que possible, car même pendant l'exécution, seule la règle concernée est déclenchée et instanciée. Nous avons tout d'abord défini une syntaxe pour notre langage GSLang constitué de variables, de termes, d'expressions booléennes, d'expression d'affectation, de service, d'instance de services, d'actions et de messages. Ensuite nous avons défini une sémantique opérationnelle afin de décrire le comportement des services GSLang. Cette sémantique opérationnelle est constituée d'un ensemble d'opérations telles que

l'instanciation, la requête, la réponse, le raffinement et le choix local d'un service. Enfin sous forme de preuve et de théorème, nous avons montré qu'un service correctement défini s'exécutera convenablement.

- Une fois GSLang formalisé, nous avons mis sur pied grâce à l'IDM des règles pour la transformation d'une spécification GSLang vers le langage de vérification et simulation PROMELA. Le choix de PROMELA est motivé par son abstraction des processus distribués, les mécanismes offerts pour la synchronisation et la coordination qui prend en compte les canaux. Dans cette partie, nous avons tout d'abord défini un méta-modèle pour GSLang et nous avons étendu un des formalismes de PROMELA proposé dans la littérature. Nous avons ensuite proposé un ensemble de règles de transformation des modèles GSLang vers PROMELA pour vérifier que la spécification est correcte.

- Un prototype a été développé conforme au style d'architecture distribuée pair à pair. Chaque pair comprend un éditeur, un moteur de transformation et un moteur d'exécution. Ces derniers permettront de faciliter la conception, la vérification et l'exécution des spécifications GSLang. Pour construire l'éditeur, nous avons utilisé Xtext qui a produit un plugins Eclipse pour la description textuelle des spécification GSLang. Nous avons aussi écrit en ATL les règles de transformation de modèles GSLang vers PROMELA pour la vérification, puis la traduction vers XML pour le système opérationnel. Le moteur d'exécution en J2EE a été mis en oeuvre. Il est composé de deux parties : (i) une interface Web développée à l'aide du framework JSF/Primefaces permettant de visualiser les services définis et les instances de services; (ii) un backend qui est constitué d'un conteneur qui charge les documents XML de spécification des services et assure leur exécution. Son rôle est de gérer les connexions à travers la création de ports dynamiques, le déclenchement (localiser, instancier, exécuter) des services au sein des $Artfacts$, formater les entrées et sorties des messages et gérer la persistance des données. Les échanges de messages entre conteneurs est assuré par un MOM pour garantir la distribution des services.

  Nous avons expérimenté ce prototype sur un exemple de gestion simplifiée des missions dans une organisation afin d'évaluer notre approche.

1.2 Perspectives

Un objectif immédiat est d'étendre la modélisation de GSLang en prenant en compte les types de données complexe au niveau des attributs. Dans la spécification initiale des *GAGs*, les paramètres d'une tâche peuvent être des fonctions. Cette extension aura des répercussions au niveau des règles sémantiques qui décrivent le calcul de la valeur des attributs.

Un aspect qui nous semble important à explorer est d'associer GSLang aux techniques du Web sémantique pour une composition de service automatique. Nous pourrions donc générer les spécifications GSLang à partir des requêtes des utilisateurs. Aussi nous pourrions effectuer la découverte et la sélection automatique de nouveaux services; le choix des services selon les contraintes de QoS; la détection automatique des violations des exigences; l'ajustement des comportements des services composites en cas de changements des services externes [101–104].

L'éditeur développé a pour principal inconvénient d'être textuel. Il doit être amélioré afin d'obtenir un éditeur proposant des éléments graphiques. Ces éléments graphiques doivent être les plus simples et intuitifs afin de faciliter son utilisation.

Au-delà des aspects fonctionnels, les propriétés de composition non fonctionnelles telles que la sécurité est de la plus haute importance pour l'adoption des techniques de composition [105–107]. C'est particulièrement le cas lorsque les compositions se produisent dans des environnements ouverts tels que le cloud [108]. Il existe plusieurs problèmes de sécurité (par exemple, confidentialité, intégrité, confidentialité, authentification et autorisation) qui doivent être prises en compte pour donner aux utilisateurs l'assurance que leurs données sont traitées en toute sécurité.

Bibliography

[1] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: a survey of techniques and tools. *ACM Computing Surveys (CSUR)*, 48(3):1–41, 2015.

[2] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014.

[3] Boualem Benatallah, Quan Z Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE internet computing*, 7(1):40–48, 2003.

[4] Paul Pocatilu and Cristian Ciurea. Collaborative systems testing. *Journal of Applied Quantitative Methods*, 4(3):394–405, 2009.

[5] Stefan Jablonski and Christoph Bussler. *Workflow management: modeling concepts, architecture and implementation*, volume 392. International Thomson Computer Press London, 1996.

[6] Mike Marin, Richard Hull, and Roman Vaculín. Data centric bpm and the emerging case management standard: A short survey. In *International Conference on Business Process Management*, pages 24–30. Springer, 2012.

[7] Henk De Man. Case management: A review of modeling approaches. *BPTrends, January*, 2009, 2009.

[8] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[9] Fabrizio Montesi. *Choreographic programming*. IT-Universitetet i København, 2014.

[10] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. Bpel4chor: Extending bpel for modeling choreographies. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 296–303. IEEE, 2007.

[11] Dominic Greenwood and Giovanni Rimassa. Autonomic goal-oriented business process management. In *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, pages 43–43. IEEE, 2007.

[12] Eric Badouel, Loïc Hélouët, Georges-Edouard Kouamou, Christophe Morvan, and Nsaibirni Robert Fondze Jr. Active workspaces: distributed collaborative systems based on guarded attribute grammars. *ACM SIGAPP Applied Computing Review*, 15(3):6–34, 2015.

[13] Richard Bird et al. *Introduction to functional programming using Haskell*, volume 2. Prentice Hall Europe Hemel Hempstead, UK, 1998.

[14] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[15] Michael N Huhns and Munindar P Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet computing*, 9(1):75–81, 2005.

[16] Michael P Papazoglou and Dimitrios Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.

[17] Michael Rosen, Boris Lublinsky, Kevin T Smith, and Marc J Balcer. *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons, 2012.

[18] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[19] Jian Yang and Mike P Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.

[20] Hye-young Paik, Angel Lagares Lemos, Moshe Chai Barukh, Boualem Benatallah, and Aarthi Natarajan. Service component architecture (sca). In *Web Service Implementation and Composition Techniques*, pages 203–250. Springer, 2017.

[21] Yujie Yao and Haopeng Chen. A rule-based web service composition approach. In *2010 Sixth International Conference on Autonomic and Autonomous Systems*, pages 150–155. IEEE, 2010.

[22] Faisal Abouzaid and John Mullins. Model-checking web services orchestrations using bp-calculus. *Electronic Notes in Theoretical Computer Science*, 255:3–21, 2009.

[23] Yi Zhu, Zhiqiu Huang, and Hang Zhou. Modeling and verification of web services composition based on model transformation. *Software: Practice and Experience*, 47(5):709–730, 2017.

[24] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil MP van der Aalst. Towards a taxonomy of process flexibility. In *CAiSE forum*, volume 344, pages 81–84, 2008.

[25] Ayoub Sabraoui, Ahmed Ettalbi, Mohammed El Koutbi, Abdeslam En-Nouaary, et al. Towards an uml profile for web service composition based on behavioral descriptions. *Journal of Software Engineering and Applications*, 5(09):711, 2012.

[26] Yang Syu, Shang-Pin Ma, Jong-Yih Kuo, and Yong-Yi FanJiang. A survey on automated service composition methods and related techniques. In *2012 IEEE Ninth International Conference on Services Computing*, pages 290–297. IEEE, 2012.

[27] David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, et al. Bringing semantics to web services: The owl-s approach. In *International Workshop on Semantic Web Services and Web Process Composition*, pages 26–42. Springer, 2004.

[28] Mike P Papazoglou and Willem-Jan Van Den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[29] Hongli Yang, Xiangpeng Zhao, Zongyan Qiu, Geguang Pu, and Shuling Wang. A formal model forweb service choreography description language (ws-cdl). In *2006 IEEE International Conference on Web Services (ICWS'06)*, pages 893–894. IEEE, 2006.

[30] Jorge Cardoso and Amit P Sheth. Semantic web services and web process composition. In *First International Workshop, SWSWPC*, pages 2004–43, 2004.

[31] Bart Orriëns, Jian Yang, and Mike P Papazoglou. A framework for business rule driven service composition. In *International Workshop on Technologies for E-Services*, pages 14–27. Springer, 2003.

[32] Anis Charfi and Mira Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World wide web*, 10(3):309–344, 2007.

[33] Hans Weigand, Willem-Jan van den Heuvel, and Marcel Hiel. Rule-based service composition and service-oriented business rule management. In *Proceedings of the International Workshop on Regulations Modelling and Deployment (ReMoD'08)*, pages 1–12. Citeseer, 2008.

[34] Maurice ter Beek, Antonio Bucchiarone, and Stefania Gnesi. A survey on service composition approaches: From industrial standards to formal methods. *Technical Report 2006-TR-15*, 2006.

[35] Jacques Sakarovitch. *Elements of automata theory*. Cambridge University Press, 2009.

[36] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[37] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 191–200. Australian Computer Society, Inc., 2003.

[38] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[39] Maurice H Ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10, 2007.

[40] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM, 2004.

[41] James Clark, Steve DeRose, et al. Xml path language (xpath) version 1.0, 1999.

[42] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[43] Gregorio Diaz, Juan-José Pardo, María-Emilia Cambronero, Valentin Valero, and Fernando Cuartero. Automatic translation of ws-cdl choreographies to timed automata. In *Formal techniques for computer systems and business processes*, pages 230–242. Springer, 2005.

[44] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[45] Raman Kazhamiakin, Paritosh Pandya, and Marco Pistore. Modelling and analysis of time-related properties in web service compositions. *Engineering Service Compositions (WESC'05)*, page 59, 2005.

[46] Carl Adam Petri. Kommunikation mit automaten. 1962.

[47] Wolfgang Reisig and Grzegorz Rozenberg. Lectures on petri nets i: Basic models, volume 1491 of. *Lecture Notes in Computer Science*, pages 7–17, 1998.

[48] Petia Wohed, Wil MP van der Aalst, Marlon Dumas, and Arthur HM Ter Hofstede. Analysis of web services composition languages: The case of bpel4ws. In *International Conference on Conceptual Modeling*, pages 200–215. Springer, 2003.

[49] Bartek Kiepuszewski, Arthur HM ter Hofstede, and Wil MP van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.

[50] Chun Ouyang, Eric Verbeek, Wil MP Van Der Aalst, Stephan Breutel, Marlon Dumas, and Arthur HM Ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of computer programming*, 67(2-3):162–198, 2007.

[51] Xiaochuan Yi and Krys J Kochut. A cp-nets-based design and verification framework for web services composition. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 756–760. IEEE, 2004.

[52] Jia Zhang, Carl K Chang, Jen-Yao Chung, and Seong W Kim. Ws-net: a petri-net based specification model for web services. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 420–427. IEEE, 2004.

[53] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *International conference on business process management*, pages 220–235. Springer, 2005.

[54] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[55] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[56] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical computer science*, 37:77–121, 1985.

[57] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems*, 14(1):25–59, 1987.

[58] Robin Milner. The polyadic π-calculus: a tutorial. In *Logic and algebra of specification*, pages 203–246. Springer, 1993.

[59] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, and Massimo Mecella. When are two web services compatible? In *International Workshop on Technologies for E-Services*, pages 15–28. Springer, 2004.

[60] Gwen Salaun, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 43–50. IEEE, 2004.

[61] Rance Cleaveland, Tan Li, and Steve Sims. The concurrency workbench of the new century, version 1.2-user's manual. 2000.

[62] Andrea Ferrara. Web services: a process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251. ACM, 2004.

[63] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp a protocol validation and verification toolbox. In *International Conference on Computer Aided Verification*, pages 437–440. Springer, 1996.

[64] Florian Rosenberg and Schahram Dustdar. Business rules integration in bpel-a service-oriented approach. In *Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, pages 476–479. IEEE, 2005.

[65] Donald E Knuth. The genesis of attribute grammars. In *Attribute Grammars and Their Applications*, pages 1–12. Springer, 1990.

[66] Willy Kengne Kungne, Georges-Edouard Kouamou, and Claude Tangha. Introducing an artifact-driven language for service composition. In *Proceedings of the ArabWIC 6th Annual International Conference Research Track*, ArabWIC 2019, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360890. doi: 10.1145/3333165.3333173. URL https://doi.org/10.1145/3333165.3333173.

[67] Willy Kengne Kungne, Georges-Edouard Kouamou, and Claude Tangha. A rule-based language and verification framework of dynamic service composition. *Future internet*, 12(2):23, 2020.

[68] Willy Kengne Kungne, Georges-Edouard Kouamou, and Claude Tangha. Extending an artifact-driven workflow model to service composition. In *Proceedings of CRI 2019*.

[69] Bart Orriëns, Jian Yang, and Mike P Papazoglou. Model driven service composition. In *International Conference on Service-Oriented Computing*, pages 75–90. Springer, 2003.

[70] J Wen, Y Huang, Z Shu, and P Li. Research on dynamic web service composition with qos global optimization in bpel. *Journal of Computational Information Systems*, 8(2):521–529, 2012.

[71] Chang-ai Sun, Yan Zhao, Lin Pan, Huai Liu, and Tsong Yueh Chen. Automated testing of ws-bpel service compositions: A scenario-oriented approach. *IEEE Transactions on Services Computing*, 11(4):616–629, 2015.

[72] Ian Sommerville. *Software Engineering GE.* Pearson Australia Pty Limited, 2016.

[73] OBJECT MANAGEMENT GROUP et al. Meta object facility (mof) 2.0 core specification, 2003.

[74] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[75] Anneke G Kleppe, Jos Warmer, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional, 2003.

[76] Ivan Kurtev. State of the art of qvt: A model transformation language standard. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer, 2007.

[77] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, 2006.

[78] J Bézivin, F Jouault, and P Valduriez. An eclipse-based ide for the atl model transformation language. In *Submitted for publication, OOPSLA Workshop*, 2004.

[79] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *International symposium on handheld and ubiquitous computing*, pages 304–307. Springer, 1999.

[80] Sylvain Degrandsart12, Serge Demeyer, and Tom Mens. Using model transformation to facilitate dynamic context adaptation. 2010.

[81] Hosung Song and Kevin J Compton. Verifying π-calculus processes by promela translation. *University of Michigan, Tech. Rep. CSE-TR-472-03*, 2003.

[82] Peng Wu. Interpreting π-calculus with spin/promela. *Computer Science*, 8(7):9, 2003.

[83] Osmar M Dos Santos, Jim Woodcock, Richard F Paige, and Steve King. The use of model transformation in the iness project. In *International Symposium on Formal Methods for Components and Objects*, pages 147–165. Springer, 2009.

[84] Abbas Abdulhameed, Ahmed Hammad, Hassan Mountassir, and Bruno Tatibouet. An approach to verify sysml functional requirements using promela/spin. In *2015 12th International Symposium on Programming and Systems (ISPS)*, pages 1–9. IEEE, 2015.

[85] Ugo Gentile. *A Model-driven Approach for the Automatic Generation of System-Level Test Cases.* PhD thesis, University of Naples Federico II, Italy, 2016.

[86] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[87] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[88] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309, 2010.

[89] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *International Conference on Model Driven Engineering Languages and Systems*, pages 128–138. Springer, 2005.

[90] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11(120):5, 2007.

[91] Stephen A White and Derek Miers. *BPMN modeling and reference guide: understanding and using BPMN*. Future Strategies Inc., 2008.

[92] Cesare Pautasso and Gustavo Alonso. Jopera: a toolkit for efficient visual composition of web services. *International Journal of Electronic Commerce*, 9 (2):107–141, 2005.

[93] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Jolie: a java orchestration language interpreter engine. *Electronic Notes in Theoretical Computer Science*, 181:19–33, 2007.

[94] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[95] Dharmendra Shadija, Mo Rezai, and Richard Hill. Towards an understanding of microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)*, pages 1–6. IEEE, 2017.

[96] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The orc programming language. In *Formal techniques for Distributed Systems*, pages 1–25. Springer, 2009.

[97] Gianpaolo Cugola, Carlo Ghezzi, Leandro Sales Pinto, and Giordano Tamburrelli. Selfmotion: A declarative approach for adaptive service-oriented mobile applications. *Journal of Systems and Software*, 92:32–44, 2014.

[98] Marco Montali, Maja Pesic, Wil MP van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. Declarative specification and verification of service choreographiess. *ACM Transactions on the Web (TWEB)*, 4(1):1–62, 2010.

[99] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The active xml project: an overview. *The VLDB Journal*, 17(5):1019–1040, 2008.

[100] Martin Garriga, Cristian Mateos, Andres Flores, Alejandra Cechich, and Alejandro Zunino. Restful service composition at a glance: A survey. *Journal of Network and Computer Applications*, 60:32–53, 2016.

[101] Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal*, 17 (3):537–572, 2008.

[102] Anis Charfi, Tom Dinkelaker, and Mira Mezini. A plug-in architecture for self-adaptive web service compositions. In *2009 IEEE International Conference on Web Services*, pages 35–42. IEEE, 2009.

[103] Quan Z Sheng, Boualem Benatallah, Zakaria Maamar, and Anne HH Ngu. Configurable composition and adaptive provisioning of web services. *IEEE Transactions on Services Computing*, 2(1):34–49, 2009.

[104] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[105] Athman Bouguettaya, Quan Z Sheng, and Florian Daniel. *Web services foundations*. Springer, 2014.

[106] Fumiko Satoh and Takehiro Tokuda. Security policy composition for composite web services. *IEEE Transactions on Services Computing*, 4(4):314–327, 2010.

[107] Halvard Skogsrud, Boualem Benatallah, and Fabio Casati. Model-driven trust negotiation for web services. *IEEE Internet Computing*, 7(6):45–52, 2003.

[108] Talal H Noor, Quan Z Sheng, Sherali Zeadally, and Jian Yu. Trust management of services in cloud environments: Obstacles and solutions. *ACM Computing Surveys (CSUR)*, 46(1):1–30, 2013.

Appendix A

Publications scientifiques tirées des travaux

.

A.1. Conférence 1

Kungne, W.K. ; Kouamou, G.E. ; Tangha, C. *Introducing an Artifact-driven language for Service Composition.* In Proceedings of the ArabWIC 6th Annual International Conference Research Track, Rabat, Morocco, 6-8 March 2019 ; pp. 1-6. (published in the **ACM's International Conference Proceedings Series**, and indexed by **EI Compendex** and **Scopus**.)

Introducing an Artifact-driven language for Service Composition

Willy KENGNE KUNGNE*
Faculty of Sciences, Computer Sciences, University of Yaoundé I
Yaoundé, Cameroon
kengnekungnewilly@yahoo.fr

Georges-Edouard KOUAMOU
National Advanced School of Engineering, University Of Yaoundé I Yaoundé, Cameroon
georges.kouamou@polytechnique.cm

Claude TANGHA
Protestant University of Central Africa,
Yaoundé-Cameroun
Yaoundé, Cameroon
ctangha@gmail.com

ABSTRACT

The most recent service composition approaches rely on the mechanism, which involves scalable and decentralized execution of services. Although some formal tools have been used to this effect, they are influenced by the standard of web service orchestration and choreography based mainly on workflow languages or notation. In this paper, we describe the formal semantics of a novel service composition language through which the services are declaratively composed and executed following a peer-to-peer paradigm. The proposed language named *GSLang* is inspired by the *GAG* (Guarded Attribute Grammars) model that has been defined for the modeling collaborative systems. Pi-calculus is used to define the basic elements of the language and its operational semantics. Then its properties are highlighted through a case study.

KEYWORDS

Dynamic Service Composition, Formal Approach, GAG, Pi-calculus

ACM Reference format:

Willy KENGNE KUNGNE, Georges-Edouard KOUAMOU, and Claude TANGHA. 2019. Introducing an Artifact-driven language for Service Composition. In Proceedings of ArabWIC conference Research (ArabWIC'19). ACM, Rabat, Morocco, 6 pages.
https://doi.org/10.1145/3333165.3333173

1 Introduction

The concept of services in Software Engineering refers to a piece of software, which provides some little functionality to its environment. The most important benefit of services is their interoperability [2]. This feature allows a system to easily leverage the functionalities of another. Service oriented computing has emerged from this vision of software as a promising solution to enhance the functionality of the standard services by composing them into large structures. Complex systems can be built by integrating various independent services. Since most of the approaches are based on business process modeling languages and notations, this study extends also a collaborative case management model so called GAG (Guarded Attribute Grammars) [1] to propose a language for the service composition.

The GAG model defines the workspaces for each user in a formal way through Grammars. It makes it possible to follow the execution of a case in the artifacts and implements strongly coupled mechanisms for the communication of workspaces. This model was proposed as a solution to data-centric workflow modeling [12]. The proposed language (*GSLang*) allows the composite services to be defined on the fly within a workspace, therefore, resulting in a declarative, decentralized, user-centric, data-driven service composition approach.

We define a composite service as a rule of production of a grammar with a left-hand side (LHS) which is the service to define and a right-hand side (RHS) being the services required to realize the LHS service. When the RHS does not exist, then the service is elementary and can be assimilated to a standard protocol of service such as SOAP (Simple Object Access Protocol) or software architecture style as REST (representational state transfer). Each service is guarded by conditions that enable them to be activated. We formalize *GSLang* by defining the concepts generally present in the field of the composition of services such as: variables (parameters), service, service instance, guard, roles, messages and actions. We describe the semantic rules that include the following operations: instantiation, sending and receiving messages, and refinement and choice of services for their execution. This semantic is described using pi-calculus formalism [3]. The choice of this formalism is motivated by the distribution of the peers across a network and their interaction, which is done through dynamic ports whose are created during execution. A service or a peer is seen as a Pi-calculus process. A peer receives and sends the messages. In this logic, a system consists of a set of processes (Peers or Services) that communicate together. During their interactions, asynchronous channels can be created and used to exchange messages. In addition, the channels so-created are included in the messages. The fact that the channels are

*This is the corresponding author

ArabWIC 2019, March 7–9, 2019, Rabat, Morocco

ACM ISBN 978-1-4503-6089-0/19/03…$15.00
https://doi.org/10.1145/3333165.3333173

dynamically created and sent to the peer into the messages, led us to choose the Pi-calculus for our modeling.

The rest of the paper is organized as follows: section 2 presents the concepts of service composition in *GSLang* and a formal semantics for their execution. Section 3 highlights the properties of the *GSLang* through a case study. Section 4 concludes and issues perspectives to this work.

2 FORMAL DESCRIPTION OF A LANGUAGE FOR THE SERVICE COMPOSITION: *GSLANG*

The *GSLang* takes GAG model concepts [1] such as workspace that is assimilated to peer; an activity is assimilated to a composite or simple service. In addition, it promotes distributivity, flexibility and data-driven (Artifact). We want to transport these properties into the world of services.

Generally, the description of a language consists of two parts: the definition of the basics elements and their behaviors. In this section, pi-calculus will be used to this effect. The basic elements for the composition of services are defined as the concepts and the behavior is apprehended through the semantic rules.

In the pi-calculus [3], the processes perform actions, which can be of three forms: the sending of a message over channel x (written $\bar{x}$), the receiving of a message over channel x (written x), and internal actions (written τ), the details of which are unobservable. Send and receive actions are called synchronization actions, since communication occurs when the corresponding processes synchronize. The notion of a transition represents the execution of a process expression. Intuitively, the transition relation tells us how to perform one-step of the process execution. Note that since there can be many ways in which a process executes, the transition is fundamentally nondeterministic. The transition of a process P into a process Q by performing an action α is indicated $P \xrightarrow{\alpha} Q$. The action α is the observation of the transition.

2.1 Basic elements of *GSLang*

The different elements of the *GSLang* are as follows:

2.1.1 ***Terms and variables***. A **variable** is characterized by a letter; it is an entity that may contain a value. **Terms** are variables, values, defined variables (assignments), Boolean expressions or functions on the terms. We define the following element $\bar{\bar{x}}$ by the tuples $(x_1 \dots x_n)$.

$$t ::= x(variable) \mid u(value) \mid x_r(defined\ variable) \mid f(t_0 \cdots t_n)$$

We extend the basic syntax of pi-calculus with Boolean expressions to verify the activation and the validation of a service.

Boolean expressions when specified and evaluated give a Boolean value.

$$e_b ::= true \mid false \mid t_j \leq t_i \mid t_i = t_j \mid e_b \mid e_b \wedge e_b \mid e_b \vee e_b$$

The **assignment** consists in solving for the variables; that is, assigning values to them. A Parameter is an input or an output variable related to a service.

$$x_r ::= \varepsilon \mid x_r\,(x \leftarrow t)(Assignment\ of\ value)$$

2.1.2. ***Service and Service Instance***. A **service** is an entity defined by a unique identifier, input variables (input parameters), output variables (output parameters), guards, post-conditions and a location which represents the associated peer. A service may depend on other services.

$$S ::= id\left(\bar{\bar{x}}, \overline{e_b}^{x}\right)\langle \bar{\bar{y}}, \overline{e_b}^{y}\rangle[\alpha] \mid id\left(\bar{\bar{x}}, \overline{e_b}^{x}\right)\langle \bar{\bar{y}}, \overline{e_b}^{y}\rangle[\alpha] \rightarrow s_1 \cdots s_n$$

Such as presented a service is simple or composite. It is characterized by an identifier (its name), input parameters $\bar{\bar{\boldsymbol{x}}}$, output parameters $\bar{\bar{\boldsymbol{y}}}$, possible preconditions on input parameters $\overline{\boldsymbol{e_b}}^{\boldsymbol{x}}$ and possible effects on the output parameters $\overline{\boldsymbol{e_b}}^{\boldsymbol{y}}$. It may be composed of other services $S_1 \dots S_n$. α represents the service location. It should be noted that α may be unnecessary for services on the RHS if they are implemented in the same user space as the services from the LHS.

For reasons of readability, we have preferred the previous notation for services in the paper. In the pi-calculus notation, it corresponds (simple or composite) to:

$$S ::= \left[\overline{e_b}^{x}\right]\alpha(id, \bar{\bar{x}}, p).\left[\overline{e_b}^{y}\right] p!\,\bar{\bar{y}} \mid \left[\overline{e_b}^{x}\right]\alpha(id, \bar{\bar{x}}, p).\,(vp_1\ \alpha\langle id_1, \overline{\bar{x}_1}, p_1\rangle \mid \dots \mid S_i \dots \mid \ vp_n\ \alpha\langle id_n, \overline{\bar{x}_n}, p_n\rangle).\left[\overline{e_b}^{y}\right] p!\,\bar{\bar{y}}$$

The service S expects $\bar{\bar{\boldsymbol{x}}}$ as the input parameter, when executed, it returns $\bar{\bar{\boldsymbol{y}}}$. It receives the data via the public port of the peer where it is accommodated. When there is a RHS, it calls the services it contains to build *y*. The services on the RHS can be executed in parallel if the data are independent of each other or in sequence if there is dependency hence the parallel operator (|) inside the brackets in bold. When a service call is initiated, a corresponding service instance is created. The same notations can be used for instances.

The **Artifact** or **Service instance** is the concrete representation of a service after the instantiation of the corresponding rule. It allows to track the execution of the service.

$$I ::= id\left(\bar{\bar{x}}, \overline{e_b}^{x}\right)\langle \bar{\bar{y}}, \overline{e_b}^{y}\rangle[\alpha]$$
$$id\left(\overline{\bar{x}_r}, \overline{e_b}^{x}\right)\langle \bar{\bar{y}}, \overline{e_b}^{y}\rangle[\alpha] \mid$$
$$id\left(\overline{\bar{x}_r}, \overline{e_b}^{x}\right)\langle \bar{\bar{y}}_r, \overline{e_b}^{y}\rangle[\alpha] \mid$$
$$id\left(\overline{\bar{x}_r}, \overline{e_b}^{x}\right)\langle \bar{\bar{y}}, \overline{e_b}^{y}\rangle[\alpha] \rightarrow I_1 \dots I_n \mid$$
$$id\left(\overline{\bar{x}_r}, \overline{e_b}^{x}\right)\langle \bar{\bar{y}}_r, \overline{e_b}^{y}\rangle[\alpha] \rightarrow I_1 \dots I_n$$

A service instance has several configurations: (i) the input and the output parameters are not yet resolved; (ii) resolved input parameters and output parameters not yet resolved; (iii) resolved input and output parameters. Each element, which appears at the

right-hand side when it exists, should have one of the three previous configurations. The parameters of the service instances are resolved progressively during execution.

2.1.3 ***Message***. **Messages** are variables that transit on the network. They contain global variables (context variables). They are composed of defined variables and/or undefined variables. There are two types of messages: request message (variables in defined inputs and variables in undefined output) and response message (defined input variables and defined output variables).

$$M ::= \overline{\overline{x_r}}\ \bar{y}\ id\ (request) | \ \overline{\overline{x_r}}\ \overline{\bar{y}_r}(response)$$

A **sending message** (request) comprises 3 parts: resolved input $\overline{\overline{x_r}}$, outputs to be resolved $\bar{y}$ and the identifier of the service to which the request is intended. A response message consists of 2 parts: resolved inputs $\overline{\overline{x_r}}$ and resolved outputs $\overline{\bar{y}_r}$. As we will see in the next section, the response is transmitted along a private port created during the request, hence the absence of the service identifier.

2.1.4 ***Action***. An **action** could be any of the following: sending message, receiving message, or silent interpretation of service instances.

$$act ::= vp\ \bar{\alpha}\ \langle M,p\rangle | \ \alpha(M,p) \mid p(M) | \ \bar{p}\langle M\rangle \mid r$$

2.1.5 **Role**. The **peer** or **execution space** (Σ) contains services, instances of services and is characterized by a single public port (its location). Let denotes by S_s the set of services, I_s the set of service instances and α its address (main port or location). Thus, the execution space is characterized by
$\Sigma = (S_s, I_s, \alpha)$.

2.2 Behavioral description

The following operational semantics describe the mechanisms for resolving services, which is broken down into several fundamental operations: instantiation, sending and receiving message, refinement and choice of services.

Instantiation

$$\frac{\Sigma' = \Sigma \cup I, p \notin fn(\Sigma)}{\Sigma: S = id(\bar{x})\langle\bar{y}\rangle[\alpha] \xrightarrow{\alpha(M,p)} \Sigma': I = id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha]}\ C_1$$

$$\frac{\Sigma' = \Sigma \cup I, p \notin fn(\Sigma)}{\Sigma: S = id(\bar{x})\langle\bar{y}\rangle[\alpha] \to S_1 \dots S_n \xrightarrow{\alpha(M,p)} \Sigma': I = id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \to I_1 \dots I_n}\ C_2$$

When Σ receives on its main port α the message M and the variable p, it finds the corresponding service S and creates the instance I with the defined input $\overline{\overline{x_r}}$ and the awaited outputs $\bar{y}$. I is added to Σ which becomes Σ′. If no service is found, then the operation will not be applied. C_2 is the extended version of C_1, the found service is composed.

Request

$$\frac{\begin{array}{l} I = id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] or \\ \quad = id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \to I_1 \cdots I_n \ or \\ \quad = I_0 \to I_1 \dots id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \dots I_n \end{array}}{\Sigma: I \xrightarrow{vp\ \bar{\alpha}\langle M,p\rangle} \Sigma: I}\ Req$$

The instance I of the space Σ sends the message M on α (main port of another space). This doesn't change the state of the execution space; M is constructed from parameters of the instance to be concretized.

Response

$$\frac{}{\Sigma: I \xrightarrow{p\langle M\rangle} \Sigma: I}\ Resp$$

Response on a private port (p) of a previously sent request.
M=$\overline{\overline{x_r}}\ \overline{\bar{y}_r}$

Refinement

$$\frac{}{\Sigma: I = id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \xrightarrow{r} \Sigma: I = id(\overline{\overline{x_r}})\langle\overline{\bar{y}_r}\rangle[\alpha]}\ R_1$$

$$\frac{}{\Sigma: I = id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \xrightarrow{p(M)} \Sigma: I = id(\overline{\overline{x_r}})\langle\overline{\bar{y}_r}\rangle[\alpha]}\ R_2$$

$$\frac{\overline{\bar{x}'} \subseteq \bar{x}}{\begin{array}{l}\Sigma: I = id_0(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \to \\ I_1 \dots id_i(\overline{\bar{x}'})\langle\bar{y}'\rangle[\alpha'] \\ \dots I_n\end{array} \xrightarrow{r} \begin{array}{l}\Sigma: I = id_0(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \to \\ I_1 \dots id_i(\overline{\overline{x_r}'})\langle\bar{y}'\rangle[\alpha'] \\ \dots I_n\end{array}}\ R_3$$

$$\frac{\overline{\overline{x''}} \subseteq \overline{\bar{y}'}}{\begin{array}{l}\Sigma: I = I_0 \to I_1 \dots \\ id_i(\overline{\overline{x_r}})\langle\overline{\overline{y_r}}'\rangle[\alpha'] \dots \\ id_j\left(\overline{\overline{x'}}\right)\langle\overline{\overline{y''}}\rangle[\alpha''] \dots I_n\end{array} \xrightarrow{r} \begin{array}{l}\Sigma: I = I_0 \to I_1 \dots \\ id_i(\overline{\overline{x_r}})\langle\overline{\overline{y_r}}'\rangle[\alpha'] \dots \\ id_j\left(\overline{\overline{x_r}''}\right)\langle\overline{\overline{y''}}\rangle[\alpha''] \dots I_n\end{array}}\ R_4$$

$$\frac{\begin{array}{l}\Sigma: I = I_0 \to I_1 \dots \\ id_i(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \dots I_n\end{array} \xrightarrow{p(M)} \begin{array}{l}\Sigma: I = I_0 \to I_1 \dots \\ id_i(\overline{\overline{x_r}})\langle\overline{\bar{y}_r}\rangle[\alpha] \dots I_n\end{array}}\ R_5$$

$$\frac{\bar{\bar{y}} \subseteq \cup\, \overline{\bar{y}_i}}{\begin{array}{l}\Sigma: I = id(\overline{\overline{x_r}})\langle\bar{y}\rangle[\alpha] \to \\ id_1(\overline{x_{1r}})\langle\overline{y_{1r}}\rangle[\alpha] \dots \\ id_n(\overline{x_{nr}})\langle\overline{y_{nr}}\rangle[\alpha]\end{array} \xrightarrow{r} \begin{array}{l}\Sigma: I = id(\overline{\overline{x_r}})\langle\overline{\bar{y}_r}\rangle[\alpha] \to \\ id_1(\overline{x_{1r}})\langle\overline{y_{1r}}\rangle[\alpha] \dots \\ id_n(\overline{x_{nr}})\langle\overline{y_{nr}}\rangle[\alpha]\end{array}}\ R_6$$

The refinement of an instance consists in materializing the parts not yet defined. The action is silent (materialization of the parameters from those already defined in an instance) or the receipt of a response on a private port.
R1: Calling a simple service (automatic or manual)

R2: Receiving information on a private port for the instance of a simple service. This action results in the materialization of the output parameters.
R3, **R4** and **R6:** Allows the definition of the parameters of certain service instances to the right from the parameters already defined. Semantic rules are used at this level to match attributes.
R5: Upon receipt of a response, materialize the part of the service's instance that made the request.

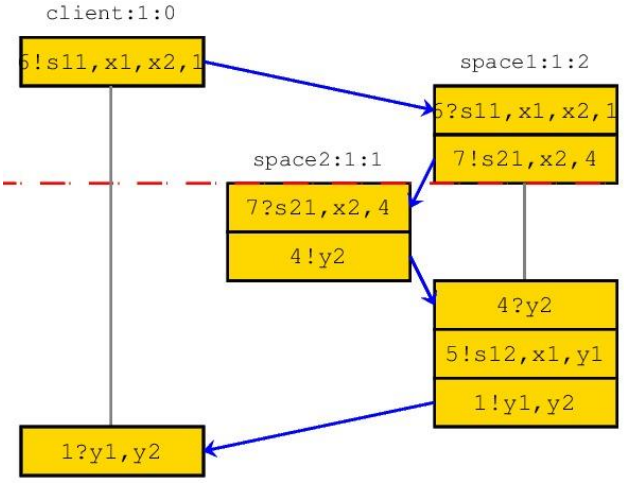

Figure 1: Execution Example

Local choice of Service (LoCh)

$$\frac{id_i(\overline{x_r})\langle\bar{y}\rangle \; match \; to \; I_0' \rightarrow I_1' \ldots I_n'}{\begin{array}{c}\Sigma: I = id_0(\overline{x_r})\langle\bar{y}\rangle \rightarrow \\ I_1 \ldots id_i(\overline{x_r})\langle\bar{y}\rangle[\alpha] \ldots I_n\end{array} \xrightarrow{r} \begin{array}{c}\Sigma: I = id_0(\overline{x_r})\langle\bar{y}\rangle \rightarrow \\ I_1 \ldots [I_0' \rightarrow I_1' \ldots I_n'] \ldots I_n\end{array}} LoCh$$

The selection of a service is made locally when the inputs are defined. Once selected, the previously described operations can be applied.

The emphasis here is on the data that influence the choice of services, their instantiation and their refinements. The execution flow depends on the availability of input and output variable values. In a service execution schema, two tasks are executed in sequence if the entries of one depend on the outputs of the other. They are executed in parallel if the inputs of one do not depend on the outputs of the other. It is for this reason that we have not explicitly defined the parallel operator of the pi-calculus. The conditional choice represents the behaviour of a peer. In a peer, under certain conditions/actions, a service or instance of services can be executed.

3 PROPERTIES OF THE LANGUAGE

This section opens with an example that will serve as a guide in order to highlight the properties of the language.

Following the logic of pi-calculus, a user space is modeled as a process, which holds services materialized by the tasks. In this regard, figure 1 is made up of two processes Σ_1 and Σ_2 representing the user space.

- The space 1 (Σ_1) contains the task s_{11} which starts the process and then decomposes into s_{12} and s_{21} which synchronizes to complete the process. s_{21} is implemented at the level of Σ_2.
- The space 2 (Σ_2) contains the task s_{21}.

Using this language, the process described in Figure 1 presents two user spaces Σ_1 and Σ_2. Initially, Σ_1 contains the services $S_{\Sigma_1} = \{s_{11}, s_{12}\}$, its public port and α_1. Σ_1 services are defined by:

$s_{11}(x_1, x_2\{x_1 > 0, x_2 > 0\})\langle y_1, y_2\rangle \rightarrow s_{21}(x_2)\langle y_2\rangle s_{12}(x_1)\langle y_1\rangle$
$s_{12}(x_1)\langle y_1\rangle \rightarrow$

The service s_{11} has a guard $x_1 > 0$ and $x_2 > 0$ and requires s_{21} and s_{12} in order to obtain y_1 and y_2. s_{21} is a remote service implemented in space Σ_2 and s_{12} is a simple local service to Σ_1.

Σ_2 contains the service s_{21} $S_{\Sigma_2} = \{s_{21}\}$ and a public port α_2. The service s_{21} is defined by:

$s_{21}(x_2)\langle y_2\rangle \rightarrow$

Also in the figure 1, a client process c executes the composite service s_{11} of Σ_1. c defines x_1 and x_2, creates a private port p whose value is 1 and sends the message containing s_{11}, x_1, x_2 and p on the public port α_1 (Of value 5) of Σ_1 (the rule C_2 is highlighted). An instance of the s_{11} service will be created in Σ_1 because s_{11} is found and the guard checked. The RHS of s_{11} starts, s_{12} and s_{21} run in parallel since there is no dependency between their parameters. A remote call on Σ_2 will be made to execute s_{21} (the semantic rule Req is used) i.e. creating a private port p_1 of value 4 and sending a message containing s_{21}, x_2 and p_1 on the public α_2 port (Of value 6) of Σ_2.

On arrival at Σ_2, with respect to the input parameters, the s_{21} service is chosen (by applying the C_1 rule), an instance of the service is created and executed. The response (principally y_2) will be sent to Σ_1 via the private port p_1 (applying the *Resp* rule). Σ_1 will refine the previously created instance. The R_6 rule can be used and the p_1 port will be destroyed. Finally, the response will be sent to the client process via the previous private port p (Of value 1).

The asynchronous private ports make it possible to track the execution of the service instances individually. An instance of service may be unavailable for a period of time; when it returns it can continue there where it was suspended. In addition, the services can be redefined at any time even during the execution since they are defined on the fly. For example in Σ_1 we can add s_{13} while s_{11} is running. This is called flexibility by change as defined in [6], contrary to flexibility by definition of existing composition approaches [2][11]. In addition, the services are fully

defined in the form of rules. The rules paradigm has been studied as a declarative approach, presenting the advantages [7] [5] [8] as:

- Adaptability: Given the declarative nature of rule-based service compositions, they can be modified and/or expanded to adapt to context-specific situations. The adaptation of the proposed language in this paper is possible at runtime because each rule (composite service) is identified and loaded when the rule is enacted. RHS not yet enabled can be updated even while running the composite service (LHS). For example in Σ_1 we can add s_{13} while s_{11} is running. s_{11} will become as follows :

$$s_{11}(x_1, x_2\ \{x_1 > 0, x_2 > 0\})\langle y_1, y_2\rangle \rightarrow\ s_{21}\ (x_2)\langle y_2\rangle s_{12}\ (x_1)\langle y_1\rangle\ s_{13}\ (\ )\langle\ \rangle$$

- Flexibility: rule-based compositions are more flexible than BPEL-type compositions, given their ability to pursue other execution paths without having to redefine the composite service and Redeploy it on a service engine. Some languages such as BPEL4WS offer a set of tags (invoke, reply, receive, sequence, choice, flow, etc.) allowing to build the composite service. In our proposition, the definition and the composition of services are described by the declarative rules, while the interaction is implicit through attributes materialized by the transmission of parameters. The private asynchronous ports (dynamic port) created at runtime make the composition more flexible. The execution path of a composite cannot be determined in advance because ports are created and destroyed dynamically as described in the example.
- Formal intuitive semantics: rule-based languages exploit a logical and/or mathematics set of underlying primitives. Formal approaches to reasoning have been proposed [9][10] but all of them use the WS-BPEL process type for their implementation. We propose an intentional definition of services that allows a late concretization of the services, thus favoring a weak coupling with the underlying technology and an adaptation (updating of the rules) of the service even during its execution. Moreover, the proposed language does not refer to any technology. The reasoning can be undertaken on services as we have done in defining operational semantics in section 2 using the pi-calculus.
- Reusability and Distribution: The composite services being defined primarily as rules can be used in different contexts. The services are distributed in different user spaces. The architecture is peer-to-peer. In the example, we have the spaces Σ_1 and Σ_2 located by their public ports α_1 and α_2.

4 RELATED WORK

A service composition language is more flexible when it is based on a declarative paradigm rather than an imperative paradigm as described in [6].

Most of the traditional languages which have been proposed to specify the composition of web services are based on processes, with BPEL as the backbone since all the proposed formalisms are translated into BPEL for their execution [14][2][21]. The disadvantage of this paradigm is that the description of the composite services represented as processes is centralized and difficult to change at runtime.

To overcome this difficulties, some languages have been proposed in order to have more flexibility [15][16][22]. They deal with the semantics of the composition by providing the ability for describing and reasoning over services at runtime [17]. These semantic-based languages are excellent in the discovery, the selection and the automatic composition of services. Their flexibility is limited to searching for missing services or building a composition plan based on a user's query and predefined planning system. It is difficult to add new requirements to the specifications of a composite service when the system is running.

Several declarative approaches to the composition of services have been proposed. The work in [18] defines the rules in the form of if..then clauses, the structure, the data and the constraint rules under the basis of elements such as Activity, Condition, Event, Flow, Provider, Role, and Message. The if..then rules govern how things are to be done in the composition. The if..then rules imply the definition of all the possibilities between the elements of the composition. This is a first step for the flexible composite service definition, but it is defined as an extension of the BPEL notations. To separate the business rules from the BPEL code, Charfi et al. suggested an aspect oriented style (AO4BPEL) [19]. Authors in [20] propose an approach named FARAO. Theyargue that business rules can be used in a service composition without the need for a BPEL framework. This greatly increases the adaptability of the orchestration. At the deployment level, a CA (Condition-Action) rule engine is introduced to support rule-based service composition. To obtain the composite service, an analysis of the services' registry (containing the WSDLs) is performed in order to have dependencies between services and to build CA rules. In CA rules, business rules and constraints will be added. Although using the rules to build the composite services, this approach has an abstraction level of the rules, which are quite low (linked to WSDL). As previous approaches, FARAO focuses on the orchestration on the detriment of distribution and interaction.

The proposed language adopts an independent approach of structured blocks such as BPML promotes by describing a composition service completely with rules, using the GAG formalism.

The adaptation of the proposed model is possible at runtime because each rule in the execution scheme are identified and loaded when the rule is enacted. Since each workspace is considered as an autonomous peer, its proprietary can update the scheme by adding new rules (service declaration) or modify the right hand side of a rule (redefinition of a composite service).

5 CONCLUSIONS AND PERSPECTIVES

This study introduces an artifact-driven language, which can be served as a framework for service composition. In this paper, we have presented a formal description of the basic concepts of this language and their behavior through the semantic rules. An example is shown on two processes to simulate the execution of a composite service. The proposed language benefits from the properties of the data-centric workflow model, it is built upon:

- The composite services are defined declaratively in the form of rules, which provides more flexibility and adaptability.
- The services participating in a composition collaborate in a peer-to-peer style.
- A service elementary or composite can be reused in different application context.

The further works will develop the support software tools for our service composition language such as the services editor, verification and translation tools. In this regard, the selection of a model-checking environment close to pi-calculus is indicated. To meet the challenges raised by the second iteration of service computing, the language shall evolve to cope with the problems of micro-services paradigm [13].

5 ACKNOWLEDGMENTS

This work was realized in the FUSCHIA project with the support of LIRIMA.

REFERENCES

[1] Eric Badouel and al, Active Workspaces: Distributed Collaborative Systems based on Guarded Attribute Grammars, Apply Computing Review, ACM, Vol 15(3), pp 6-34, 2015.
[2] Quan Z. Sheng and al, Web services composition: A decade's overview, Inf. Sci. , 280: 218-238 (2014).
[3] Sangiorgi, D., Walker, D., The pi-calculus: a Theory of Mobile Processes, Cambridge university press, 2003.
[4] Bultan, T. and al, Conversation Specification: A New Approach to Design and Analysis of E-service Composition, ACM 1-58113680-3/03/0005, WWW, 2003.
[5] Weigand, H., van den Heuvel, W.J., Hiel, Rule-based service composition and service-oriented business rule management, International Workshop on ReMoD, (2008).
[6] Mulyar, N., Schonenberg, M., et al., Towards a taxonomy of process flexibility (extended version), , (2007).
[7] Yao, Y., Chen, H., A rule-based web service composition approach, Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference, pp. 150-155. IEEE (2010).
[8] Rosenberg, F., Dustdar, S, Business rules integration in bpela service-oriented approach, E-Commerce Technology, 2005. CEC 2005. Seventh IEEE International Conference, pp. 476-479. IEEE (2005).
[9] Zhu, Y., Huang, Z., Zhou, H., Modeling and verification of web services composition based on model transformation, Software: Practice and Experience , 47(5), 709-730 (2017).
[10] Abouzaid, F., Mullins, J., Model-checking web services orchestrations using bp-calculus, Electronic Notes in Theoretical Computer Science , 255, 3-21 (2009).
[11] Sun, Chang-Ai, et al, Automated testing of WS-BPEL service compositions: A scenario-oriented approach, IEEE Transactions on Services Computing, (2015).
[12] COHN, David et HULL, Richard, Business artifacts: A datacentric approach to modeling business operations and processes, IEEE Data Eng. Bull, 32, 3, p. 3-9. (2009).
[13] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin et R., Safina, Microservices: yesterday, today, and tomorrow, In Present and Ulterior Software Engineering. Springer, Cham, pp. 195-216 (2017)
[14] Sabraoui, A., Ettalbi, A., El Koutbi, M., EnNouaary, A., Towards an uml profile for web-service composition based on behavioral descriptions, Journal of Software Engineering and Applications, 5(09), 711 (2012)
[15] Papazoglou, M.P., Heuvel, W.J., Service oriented architectures: approaches, technologies and research issues, The VLDB Journal The International Journal on Very Large Data Bases, 16(3), 389415 (2007)
[16] Yang, H., Zhao, X., Qiu, Z., Pu, G., Wang, S., A formal model forweb service choreography description language (ws-cdl), ICWS06. International Conference on Web Service, pp.893894. IEEE (2006)
[17] Martin, D. et al, Bringing semantics to web services: The owl-s approach, International Workshop on Semantic Web Services and Web Process Composition. Springer, Berlin, Heidelberg., pp. 26-42 (2004)
[18] Orriens, B., Yang, J., Papazoglou, M., A framework for business rule driven service composition, Technologies for E-Services. pp. 1427 (2003)
[19] Charfi, A., Mezini, M., Ao4bpel: An aspect-oriented extension to bpel, World wide web. 10(3), 309344 (2007)
[20] Weigand, H., van den Heuvel, W.J., Hiel, M., Rule-based service composition and service-oriented business rule management, Proceedings of the International Workshop on Regulations Modelling and Deployment (ReMoD08), pp. 112. June (2008)
[21] Lemos, A. L., Daniel, F., Benatallah, B., Web service composition: a survey of techniques and tools, ACM Computing Surveys (CSUR), 48(3), 33. (2016)
[22] Syu, Y., Ma, S. P., Kuo, J. Y., FanJiang, Y. Y. , A survey on automated service composition methods and related techniques, In Services Computing (SCC), 2012 IEEE Ninth International Conference on (pp. 290-297), (2012) (2012, June).

.

A.2 Conférence 2

Willy Kengne Kungne, Georges Edouard Kouamou, Claude Tangha. *Extending an artifact-driven workow model to service composition.* Conférence de Recherche en Informatique (CRI) 2019, Yaoundé.

CRI'2019

Extending an artifact-driven workflow model to service composition

Willy KENGNE KUNGNE* — Georges-Edouard KOUAMOU** — Claude TANGHA**

* Faculty of Sciences, University of Yaoundé I, PoBox 812 Yaoundé-Cameroun, kengnekungnewilly@yahoo.fr

** National Advanced School of Engineering, University of Yaoundé I, PoBox 8390 Yaoundé-Cameroun, georges.kouamou@polytechnique.cm

*** Protestant University of Central Africa, PoBox 4011 Yaoundé-Cameroun, c.tangha@gmail.com

RÉSUMÉ. Traditionnellement, les langages de composition de service reposent sur un flux de travail orienté processus dont les plus connus sont BPEL4WS, WS-CDL ou SCA. Ils sont impératifs et centrés sur la manière dont les services composites doivent être construits, ils sont donc rigides aux changements à l'exécution. Avec l'avènement des workflows pilotés par les artefacts, nous avons l'intention d'exploiter l'un des modèles proposés dans ce cas, le modèle GAG (Guarded Attributed Grammars), afin de mettre en évidence ses propriétés pour la composition de services.

ABSTRACT. Traditionally, the service composition languages rely on process oriented workflow of whom the well-known are BPEL4WS, WS-CDL or SCA. They are imperative and focused on how composite services must be constructed, therefore they are rigid to change at runtime. With the advent of the artifact-driven workflows, we intent to exploit one of the proposed models in this case, the Guarded Attributed Grammars (GAG) model, for the services composition in order to highlight its properties for the composition of services.

MOTS-CLÉS : workflow orienté artefact, composition de service, modele de GAG, System collaboratif.

KEYWORDS : Artifact-driven workflow, Service Composition, GAG model, Collaborative System.

1. Introduction

Services are usually defined as independent software entities that can be published, discovered by third parties (consumers) and consumed regardless of their implementation [7, 3, 9]. The expansion of the web has favored their growth. However, one can note that these services, considered alone are not enough to respond to the request of users. Therefore, in many situations, they must be combined to achieve a particular goal. The traditional approaches of services composition are based mostly on a description in the form of business processes [8, 1]. This allows to describe the composition scheme of service as a process e.g. BPEL4WS. Most of these languages describe explicitly the selection and order of execution of elementary services within the composite. They propose predefined elements to obtain structures represented as graphs.

Faced to the requirements of increasingly growing for service composition such as distribution, availability and mobility of users and also their preferences that are changing, the existing languages are limited. They have the disadvantage of being rigid because the partner services are selected and assembled during design. It is not possible to change the composition scheme during the execution of a composite service in order to adapt it to new user requirements that can occur during execution.

In view of the taxonomy of the different types of flexibilities contained in [6] the composition languages of services workflow-based lead to flexibility by definition or by deviation because they allow defining several paths possible execution in the composite service at design step. This is inherent to the imperative description of these languages. As shown in [6], in order to have a flexibility by change (i.e. at runtime, the composite service is modified) or under-specification (i.e. at runtime, composite service is enriched), the service composition languages should be used in a declarative description in order to consider enhancements or changes during execution of the composite service.

This study highlights the properties of an artifact-oriented workflow model used in the framework of the service composition. Here, the Guarded Attributed Grammar (GAG) model proposed in [4] for the modeling of user distributed and asynchronous systems. To achieve this goal, we briefly present the GAG model in section 2. The section 3 presents the elements of a service composition language as well as a correspondence between the GAG model and the service composition languages. The section 4 highlights the properties of our approach for the service composition. Finally, the section 5 concludes the paper and gives some outlook.

2. Overview of the GAG model

The traditional Workflow models put emphasis on the orchestration of the individual activities. In this context one puts stress on the control and coordination of the tasks required for the achievement of a particular goal. Such systems are usually modeled using centralized and state-based formalisms like automata, or Petri nets. They can also be directly specified in some dedicated programming language like BPEL, or notations like BPMN. In these approach, the data that are exchanged during the processing of a task play a secondary role when they are not simply ignored [9]. By contrast, the model of artifact-oriented workflow systems, highlighted by IBM, puts stress on the exchanged documents, the so-called Business Artifacts [5]. GAG has also been proposed to satisfy this need. In the GAG, the actions in the system are modeled as the productions of grammar.

For e.g we note:

$P :: A_0(x_1..x_n)\langle y_1, ..y_m\rangle \rightarrow A_1(x_{11}, ..x_{1k})\langle y_{11}, ..y_{1l}\rangle, ...A_p(x_{p1}, ..x_{pi})\langle y_{p1}, ..y_{pj}\rangle$ which means that the production P contains on the left hand side the activity A_0 which is solved by combining the activities on the right hand side A_1, A_2 ... A_p. In other words, the activity A_0 is composed of the activities A_1, A_2 ... A_p. More concretely, if we have

$preparingLogistic(reserve)\langle infoReserve\rangle \rightarrow$
$flightBooking(reserve)\langle airlineComp, fligthNum, seatNum\rangle$
$hotelReservation(reserve)\langle roomNum, hotelNum\rangle$
$carRental(reserve)\langle taxiNum\rangle$

The definition means that the preparing Logistic activity is composed of the flightBooking, hotelReservation and CarRental activities. If a production doesn't hold right-hand side, it is an elementary activity.

The productions could be associated with the conditions so-called guard, related to the input parameters of the activity which constraint its triggering. The parameter are defined as attributes, so the input parameter are inherited attributes and the output parameter are synthesized attributes.

A production is triggered when the input attributes are available, and the constraints are checked. Let's consider the following production (P1):

$P1 :: preparingLogistic(reserve, reserve.distance > 60Km)\langle infoReserve\rangle \rightarrow$
$flightBooking(reserve)\langle infoReserv1\rangle[bookingAg]$
$hotelReservation(reserve)\langle infoReserv2\rangle[hotel]$
$carRenting(reserve)\langle infoReserv3\rangle[carRentalA]$

P_1 is triggered if the reserve variable is set and the distance is greater than 60Km. We will say that the choice of a production is driven by the data. The flow of execution is therefore not explicit; it depends to the input parameters and the satisfaction of guards (i.e. the availability of the values of the parameters).

A production set is associated with a role.

3. A composition service language

3.1. Basics of a service composition language

A language for the composition of services has the following different elements: [2] [10]: activity, condition, event, flow, message and role.

– An Activity: represents a well-defined business function (similar to e.g. basic activities in BPML). It contains a name, the inputs and the outputs.

– Condition: describes the behavior of the composition by controlling event occurrences, guarding activities and enforcing pre-conditions, and post-conditions.

– Event: describes occurrences during the process of service composition and its impact on an activity. These can be both of a normal and exceptional nature.

– Flow: defines a block of activities and how they are connected. Generally using basic activities. They show the enchainment of basic activities contained in the activities. The block is explicitly defined in the form of a sequence, a conditional statement or even a loop.

– Message: represents a container of information. The Messages are used and generated by activities. It describes the interactions between the activities that communicate.

– Role: provides an abstract description for a party participating in the service composition. Here the role describes where a partner service is located.

3.2. Correspondence between a service composition language and the GAG model

The activities in service composition languages are the production with empty right hand side in the model of GAG.

A right-hand side of the production is actually a Flow in the traditional service composition languages. A flow explicitly defines the order of execution of the different associated activities while a production declares the associated activities. If a composite service is mapped as a production, then the service is declarative as opposed to the imperative character of the existing service composition languages. If an activity s is defined by $s \rightarrow s_1, s_2$. The flow consisting of s_1 and s_2 will be $sequential$ when for example the output of s_1 is used as the input of s_2 ($s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s_2(y')\langle y\rangle$) . and $parallel$ when there is no dependency between s_1 and s_2 ($s(x)\langle y_1, y_2\rangle \rightarrow s_1(x)\langle y_1\rangle s_2(x)\langle y_2\rangle$).

The conditions are the inputs constraints associated with the productions, it is the same in the existing service composition languages.

The events are linked to the triggering of the productions: either by the availability of attributes or by a user. The definitions of events in this regards satisfy the same vision as in the common language.

The roles will be a set of productions defining the grammar linked to a user. The activities on the right-hand side of the production not yet activated can be modified. This is not the case for the existing service composition languages where any change in the schema of a composite service is done statically when the system is stopped.

4. Prototype

A prototype has been developed to show how service instances are created and run from the defined services. Web interfaces developed in Java using the JSF/Primefaces framework. Each role can use these interfaces to visualize the services contained and their instances. The web interface is shown in figure 1. After the login of a role, it presents the services available for the role (at the top) and the service instances that are running (at the bottom).

The debugging aspect of this work is the possibility offered to the users to observe the evolution of the service instances as shown in figure 2 where an instance of the checkDemand service is presented. The refined nodes as well as related information can be viewed. We can request each node of the checkDemand service for the details of these informations.

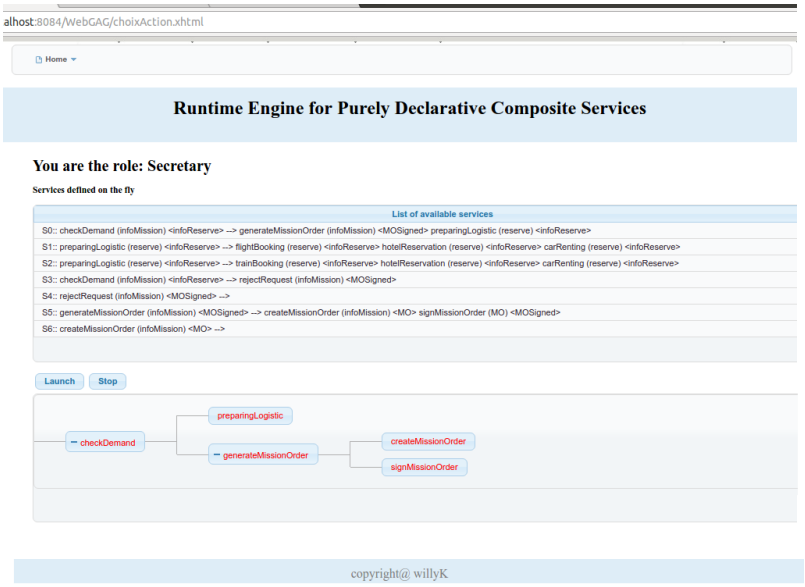

Figure 1. *Main interface of the Execution Engine*

The black nodes are the services already treated and the red nodes are the pending services waiting for an event or a information to continue their execution.

Service schemas can be updated at any time, and the runtime engine will take these changes into account when running.

5. Highlighted Properties of the approach

The services composition model defined from GAG enables to change the definition of the service composition execution at runtime. This opportunity is the consequence of

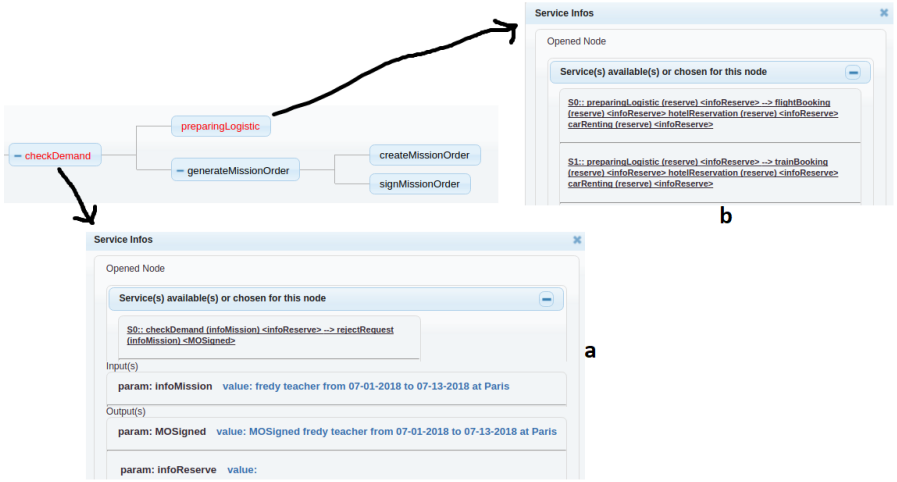

Figure 2. *A runtime instance of the CheckDemand service*

many properties which are highlighted in the following.:

– Declarative: the language described is based on the rules (Production) that describe a service by specifying its name, input and output parameters, semantic rules (describing the correspondence between attributes) and services to solve it (right hand side).

– Abstract Specification: The intentional definition of services allows a late concretization of the services thus favoring a weak coupling with the underlying technology and an adaptation (updating of the rules) of the service even during its execution.

– Connectivity (Asynchronous) and Distribution: sends and receives asynchronous messages for the creation and refinement of services. Indeed, a service is implemented in an execution space having, among other things, a public port for communication. Dynamically created ports work together to refine instances of services. Services are distributed in different locations so-called user workspace

– Adaptation and Flexibility: Services can be defined at any time, verified and integrated into the execution spaces.

– User-centric: Users can define or update services. Users are put forward; the services are related to user space.

– Data oriented: The services parameters guide for the choice of the services. Depending on these parameters, the user or execution engine does the choice of the service to instantiate.

6. Conclusion and Future Works

The composition of services allows obtaining more complex services in order to achieve specific objectives. Since the models for the service composition are based for the most part on the traditional workflow models whose main drawback is their rigidity, we have proposed in this paper to extend the GAGs for the composition of services. We first briefly presented the elements of the GAG model and we recalled the basic elements of service composition languages. Then we performed equivalence between the GAG model and the service composition languages. This gives us the opportunity to extend the GAG model for service composition. What brings new properties (declarative, adaptability, flexibility by change, Abstract Specification, distribution, user centric) in the languages of composition of service.

As a perspective, we intend to fully describe the GAG-based service-composing language and an environment to support the specification and the execution of the composite services. Given that the present paper presents a structural correspondence, a study of the operational semantics of GAGs and the proposed services composition languages can also be done.

7. Bibliographie

[1] ALSEDRANI, A. & TOUIR, A. , « Web service composition processes: A comparative study », *International Journal on Web Service Computing (IJWSC)*, vol. 7, n° 1, 1-21., 2016.

[2] ORRIËNS, B., YANG, J. & PAPAZOGLOU, M. P. , « Model driven service composition », *In International Conference on Service-Oriented Computing. Springer, Berlin, Heidelberg.*, pp. 75-90., 2003.

[3] PELTZ, C., « Web services orchestration and choreography. », *IEEE Computer*, vol. 36, n° 10, pp. 46-52., 2003.

[4] BADOUEL, E., HÉLOUËT, L., KOUAMOU, G. E., MORVAN, C., & FONDZE JR, N. R., « Active workspaces: distributed collaborative systems based on guarded attribute grammars. », *ACM SIGAPP Applied Computing Review*, vol. 15, n° 3, pp. 6-34., 2015.

[5] HULL, R., DAMAGGIO, E., DE MASELLIS, R., FOURNIER, F., GUPTA, M., HEATH III, F. T., ... & SUKAVIRIYA, P. N., « Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. », *In Proceedings of the 5th ACM international conference on Distributed event-based system*, pp. 51-62., 2011.

[6] SCHONENBERG, H., MANS, R., RUSSELL, N., MULYAR, N., & VAN DER AALST, W. M., « Towards a Taxonomy of Process Flexibility. », *In CAiSE forum*, vol. 344,pp. 81-84 2008.

[7] M. P. PAPAZOGLOU & W. J. VAN DEN HEUVEL., « Service oriented architectures: approaches, technologies and research issues », *VLDB Very Large Database Journal*, 2007.

[8] QUAN Z. SHENG, XIAOQIANG QIAO, ATHANASIOS V. VASILAKOS, CLAUDIA SZABO, SCOTT BOURNE & XIAOFEI XU, « Web services composition: A decade's overview. », *Inf. Sci. 280* , 218-238 2014.

[9] ABITEBOUL, S., BENJELLOUN, O., & MILO, T., « The Active XML project: an overview », *The VLDB Journal—The International Journal on Very Large Data Bases*, 1019-1040 2008.

[10] WEN, J., HUANG, Y., SHU, Z., AND LI, P., « Research on dynamic Web ser- vice composition with QoS global optimization in BPEL », *Journal of Computational Information Systems*, vol. 8, n° 2, 2012

.

A.3 Journal 1

Kungne, W. K., Kouamou, G. E., and Tangha, C. (2020). *A Rule-Based Language and Verification Framework of Dynamic Service Composition.* Future internet, 12(2), 23. (journal indexed by **EI Compendex** and **Scopus**)

future internet

MDPI

Article

A Rule-Based Language and Verification Framework of Dynamic Service Composition

Willy Kengne Kungne [1,*], Georges-Edouard Kouamou [2] and Claude Tangha [3]

[1] Department of Computer Sciences, Faculty of Sciences, University of Yaoundé I, P.O. Box 812 Yaoundé, Cameroon
[2] Department of Computer Sciences, National Advanced School of Engineering, University of Yaoundé I, P.O. Box 8390 Yaoundé, Cameroon; georges.kouamou@polytechnique.cm
[3] Faculty of Information Technologies and Communication, Protestant University of Central Africa, P.O. Box 4011 Yaoundé, Cameroon; c.tangha@gmail.com
* Correspondence: kengnekungnewilly@yahoo.fr; Tel.: +237 675 978 859

Received: 8 December 2019; Accepted: 23 January 2020; Published: 26 January 2020

Abstract: The emergence of BPML (Business Process Modeling Language) has favored the development of languages for the composition of services. Process-oriented approaches produce imperative languages, which are rigid to change at run-time because they focus on how the processes should be built. Despite the fact that semantics is introduced in languages to increase their flexibility, dynamism is limited to find services that have disappeared or become defective. They do not offer the possibility to adapt the composite service to execution. Although rules-based languages were introduced, they remain very much dependent on the BPML which is the underlying technology. This article proposes the specification of a rule-based declarative language for the composition of services. It consists of the syntactic categories which make up the concepts of the language and a formal description of the operational semantics that highlights the dynamism, the flexibility and the adaptability of the language thus defined. This paper also presents a verification framework made of a formal aspect and a toolset. The verification framework translates service specifications into Promela for model checking. Then, a validation framework is proposed that translates the verified specifications to the operational system. Finally, a case study is presented.

Keywords: rule-based approach; service choreography; flexibility by change; adaptability; DSL; model checking.

1. Introduction

Services are increasingly developed and published on the Web. Often, these services taken alone serve lightweight functionality. In many cases, a single service is not sufficient to satisfy user requests. So several services need to be combined to achieve a specific goal [1,2]. Several languages were proposed to define composite services. These languages rely for the most part on process models with the BPML language as the reference [2]. The latter provides structured and rigid blocks based on graphs which are then translated into static composite services.

The ability to manage the flexibility and the evolution of a composite service at runtime is an essential requirement. For example, a user in order to prepare its travel, needs to buy a flight ticket and reserve a room. The user get in touch with a travel agent which will communicate with a hotel service and an airline service. May be the travel agent make available the possibility to rent a car. This new service can be provided by a car rent service or one of the initial agent (hotel or airline). The new service will be added in the travel agent space without stop the running service. In addition, the car rent service is optional for the users who need it.

The changes in travel agent requirements have an immediate impact on the specification of the composite services. This situation can occur even at runtime since the services which participate to a composite can be removed or added any time.

Therefore, the need to have the flexible and adaptive services composition approach is a necessity. Several studies have been carried out to make them more adaptable during execution [1–6]. To this end, it is appropriate to offer languages that promote more flexibility (as soon as the services are specified) and adaptability (in the execution framework). To meet these requirements, rule-based declarative approaches were proposed because they promote reusability, adaptability and flexibility for the composition of services [7].

How can we take into account the user needs or the new services even during the execution of a composite service? The research question in this paper deals with the proposition of a language for the definition of flexible and adaptable services. Rather than providing runtime techniques to adapt to the changes [3], the required need consists of anticipating the changes by relying on a language which is purely declarative and rules oriented which facilitates a greater flexibility and adaptation by describing the execution schema of the composition (the *what*) rather than the execution path of the composition (the *how*). The proposed language here is based on the production rules with the purpose to operationalize the Guarded Attributed Grammar (GAG) model [8] through the specification of a service composition language. Basically, GAG models are designed principally in order to contribute to the modelling of artifact-driven workflows. They are declarative rules-based, data-oriented, user-centric, and distributed. The transposition of all these properties in the domain of the services composition makes it possible to have a more flexible composition environment because the language proposed in this article is purely declarative and based on the rules.

A composite service is defined by using grammar productions as composition rules. The left hand side of a production rule defines a new service that is obtained by assembling the entities on the right hand side. Any change in the description of the composition scheme is automatically taken into account when the affected rule is applied. The services are connected at runtime through the dependencies between their attributes. The composition is controlled by the data and the associated conditions so-called guards. The operational semantics which consists of a description of the services selection, the creation of the instances of the services and their refinements is presented using process algebra (pi-calculus) [9]. The choice of pi-calculus is supported by the fact of the distribution of peers in a network and they interactions through dynamic ports created at runtime.

The rest of the paper is structured as follows. Section 2 presents the state of the art on rule-based service composition languages. In Section 3, a declarative language for the composition of services is proposed, which consists of the syntax definition, the specification of the operational semantics and its verification. In Section 4, a verification framework of the proposed language is presented based on the PROMELA language and the SPIN tool. Section 5 describes a validation framework that presents a prototype for evaluation of our approach, this prototype consists of an editor (DSL) for the proposed language, model transformation rules to Promela/SPIN and an execution engine. Section 6 highlights the strengths of our work compared to the existing in the field of the composition of services. Section 7 concludes the work and presents some future directions.

2. A State of the Art on Ruled based Specification of Service Composition

Traditional languages proposed for specifying web services composition are process-based hence their imperative character on the one hand. On the other hand, they are very related to the underlying technology.

Languages that are standards such as BPEL [10], BPML [11], WS-CDL [12] or WSCI [13] propose to define composite services in the form of structured blocks thus making their flexibility and adaptation difficult. Several other languages and tools have been proposed based on these standardized languages to improve their flexibility [1,2,14]. The other disadvantage of these languages is that the processes are centralized and difficult to modify during execution. Moreover, the composite services are described

by WSDL [15] interfaces therefore the orchestration of other types of services that are not compliant with WSDL is not possible. The level of abstraction of the specification of the composition is quite low.

To overcome these difficulties, some languages propose increasing the autonomy of the peers and effective distribution [16,17]. They also present the semantics of the composition by offering the possibility of describing and reasoning about the services during the execution [18,19]. These languages based on semantics are excellent to discover, select and automatically compose services. However, their flexibility is limited to search the missing services or to build a composition plan based on the query of user and a predefined planning system. It is difficult to add new requirements to a specification of composite service when it is executed.

Instead, the rules paradigm has been presented as a declarative approach. It presents the following advantages [7,20,21]:

- Flexibility: compositions based on rules are more flexible than BPML-type compositions. It is able to pursue other execution paths in case where a particular execution path fails without to redefine the composite and redeploy it;
- Adaptability: given the declarative nature of the rule-based service compositions, they can be update to adapt to specific situations, for example in terms of external services or platform deployment;
- Reusability: Since rules are isolated from the context of the business process, they can be reused more easily in other service application contexts;
- Formal semantics: languages based on rules exploit a logical and/or mathematics set. Formal approaches allowing reasoning have been proposed [22–24] but all use the WS-BPEL process type for their implementation.

Considerable efforts have been invested in providing the rule engines to support flexible and adaptable service compositions.

In [4,25], a service-oriented rule engine was introduced, allowing access to business rules by invoking the RuleML engines. [23] introduces an alternative service execution environment in which rules can be defined and then injected into WSDL specifications, after which they can be deployed to a service executor.

[26] defines in the form of clauses *if..then*, the structure, the data and the constraint rules under the basis of elements such as Message, Event, Condition, Provider, Flow, Role, and Activity. This is a first step for the flexible composite service specification but it is presented as an extension of the BPEL notations. To separate the business rules from the BPEL code, Charfi et al. suggested an aspect-based style (AO4BPEL) [27].

[7] argues that business rules can be used in a service composition without the need for a BPEL framework. This increases the adaptability of the orchestration. At the deployment level, a *CA* (Condition-Action) rule engine supports rules-based service composition. To obtain the composite, an analysis of the registry (containing WSDLs) of the services is performed in order to extract the dependencies between services and to build the CA rules. The business rules and the constraints added to the CA rules. Although using the rules to build the composites, this approach has an abstraction level of the rules which is quite low (linked to WSDL). Moreover, it focuses, as do the previous ones, on the orchestration thus leaving the distribution and interaction in the background.

This study adopts an independent approach of the block structuration such as BPML. We promote describing a composition service completely with rules, using the GAG formalism. [8] to intentionally specify the composition of the services. The intentional definition of services composition makes them abstract, which increases the flexibility because the link between the intentions and the implementation is done when a rule is enacted. Any update is possible on a rule and it is taken into account the next time the rule is enacted. The advantage of this property allows conforming the rules with the user needs even at runtime by adapting the rules to the new requirements. Moreover, the use of rules in a service composition language brings the formal intuitive semantics, flexibility, adaptability, and reusability.

3. Declarative Formalism for the Service Composition

One of the main challenges of software engineering is dealing with the changes (social changes, user requirements changes, etc.). Concerning the aspect of service composition, one talks about flexibility by changing, which means modifying the structure of a composite service even at runtime [28]. We propose, in this section, a service specification formalism that uses productions of a context-free grammar [29]. This so-called intentional specification, because it remains abstract, is constituted at the right hand side (RHS) of the description of the services that concur to solve the left-hand side (LHS) service. In our approach, the composition scheme remains as abstract as possible, because even during execution, only the concerned rule is enacted and instantiated.

3.1. Basic Definition

A service is defined within the frame of an activity. An activity is a tuple
A = (id, S, C) where

- **Id** is a unique identifier of the activity (for example, its name);
- **S** is the execution schema that describes the services $\{t_0, t_1 ... t_n\}$ to be performed to resolve the activity;
- **C** is the context indicating information from the environment in which the activity is performed. This information can contribute to the modification of the execution scheme of an activity. $C = (I, O)$ where I is the inputs getting from the environment and O the outputs returned to the environment at the end of the execution of the activity.

Each service s ∈ S can be defined in one of the following forms:

$$s(c_0 ... c_n)\langle r_0 ... r_m\rangle \rightarrow s_1(i_{1i} ... i_{1u})\langle o_{1i} ... o_{1k}\rangle \; ... \; s_j(i_{j1} ... i_{jo})\langle o_{j1} ... o_{jh}\rangle \quad (1)$$

$$s(c_0 ... c_n)\langle r_0 ... r_m\rangle \rightarrow \quad (2)$$

In this respect, we distinguish composite services if the right hand side of the rule is not empty, and elementary services otherwise. A composite service depends on the services which appear on the right hand side for its completion in other words the RHS services aggregate the service that appear at the LHS. An elementary service provides access to Internet-based application via a well-known protocol such as SOAP or REST architecture. This concept of service will help to ensure interoperability among the peers. For each type of rule, a function of correspondences that defines the semantic rules between the parameters can be associated. Let A be an activity accommodated by a peer h. If A is defined as:

$$s(c_0 ... c_n)\langle r_0 ... r_m\rangle \rightarrow s_1(i_{1i} ... i_{1u})\langle o_{1i} ... o_{1k}\rangle \; ... \; s_j(i_{j1} ... i_{jo})\langle o_{j1} ... o_{jh}\rangle$$

Where s_1, .., s_j make up the service s, the c_k are the input parameters to s and the r_k are the output parameters, the i_{kj} are the input parameters of the RHS and the o_{kj} are the output parameters.

The semantic rules are given by the dependencies between attributes which make possible to apply the substitutions

$$\begin{cases} i_{kl} = f(i_{ij}, o_{mn}, c_j) with\ k \neq\ i,\ l\ \neq\ j \\ r_k = f(o_{mn}) \end{cases}$$

Where the inputs of the RHS depends on the inputs of the RHS of other services, the outputs of the RHS and the inputs of LHS of s. The outputs of the LHS are functions of the outputs of the RHS. f materializes this dependency Example:

$travelService(username, date, destination)\langle numVol, hotelAdress\rangle \rightarrow$
. $flightBooking(username, date, destination)\langle numVol\rangle$
. $hotelReservation(username, date)\ \langle hotelAdress\rangle$

$numVol$ of $travelService$ corresponds to $numVol$ to $flightBooking$. Other examples can be taken for this $travelService$ service.

The semantic rules are also expressed by the conditions on the input and the output parameters which serve as guards and post-conditions.

This so-called intentional definition is materialized during its instantiation. The service instance takes place according to the availability of the data (parameters). When the composition scheme of an activity is updated, the modification will be applied next time the associated rule be enacted. The semantic rules may involve several variables which belong to different peers. At this moment a private dynamic channel is opened for the interaction between the peers. These are the reasons why the pi-calculus is chosen in the next section to specify the composition language. In fact, one can notice that calculi based on the pi calculus provide primitives for describing and analyzing global distributed infrastructure, focusing on process migration between peers, process interactions via dynamic channels, and private channel communication [9]. In the process algebra (pi-calculus) [9], the processes execute actions that can have three forms: the sending of a message over x channel (written $\overline{x}$), the receiving of a message over x channel (written x), and silent actions (written τ), whose details are unobservable. Send and receive actions are called actions of synchronization, since communication is highlighted when the processes synchronize. The transition notion represents the executing a process expression. Intuitively, the transition relation indicates how to perform one-step of the process execution. Note that since there may be many ways to execute a process, the transition is basically non-deterministic. The transition of a process A into a process B by performing an action α is indicated A $\xrightarrow{\alpha}$ B. The action α is the observation of the transition.

The description of the services thus defined and their behaviors are based on primitives of instantiation, refinement and exchange of messages which can also be found in more abstract languages such as process algebras. This permits us to describe the structure of a service and present formally how a service is defined and how it interacts in its environment. In the following, we describe formally the syntax and the semantics of a service composition language.

3.2. A Formal Syntax and Semantic of the Service Composition Language: GSLang

The language is called GSLang and was introduced in [30]. This section presents GSLang and shows how the defined services can be resolved. The elements of the GSLang language are:

3.2.1. Variables and Terms

A **variable** is a letter; it can contain a value. The **terms** are values, variables, defined variables (assignment), functions on the terms or Boolean expressions.

In this paper we define, $\overline{\overline{x}}$ for tuple (x_1,... x_n).

$$\begin{array}{rcl} t & ::= & x(variable) \\ & | & u(value) \\ & | & x_r(defined\ variable) \\ & | & f(t_0 ... t_n) \end{array}$$

To the basic syntax of pi-calculus, we add Boolean expressions to check the activation and validation of a service. **Boolean expressions** when specified and evaluated gives a Boolean value. In other words it is a logical expression on the variables. They serve as guards when activating a service.

In this case, they are pre-conditions; they control the triggering of a service. At the end of the execution of a service, there may be conditions to be satisfied. These are post-condition.

$$\begin{aligned} e_b \quad ::= \quad & true \mid false \mid t_j \leq t_i \\ & \mid t_i = t_j \mid e_b \mid e_b \wedge e_b \mid e_b \vee e_b \end{aligned}$$

The **assignment** affects a value to a variable.

$$x_r \quad ::= \quad \epsilon \mid x_r \ (x \leftarrow t)(value\ assignment)$$

A parameter is an output or an input variable related to a service. The scope of a variable is related to the associated service. The context of activity is characterized by input variables and output variables. These variables are free and unique throughout the system. One notes that a variable is defined or resolved when a value is assigned to it (x ← u). A variable is define once and can be used several times.

3.2.2. Service and Service Instance

A **service** is defined by a unique identifier, output variables (output parameters), input variables (input parameters), post-conditions, guards and a location. A service can depend on other services.

$$\begin{aligned} S \ ::= \ & id(\ \overline{\overline{x}}, \overline{e_b}^x)\ \langle \overline{\overline{y}}, \overline{e_b}^y \rangle\ [\alpha] \mid \\ & id(\ \overline{\overline{x}}, \overline{e_b}^x)\ \langle \overline{\overline{y}}, \overline{e_b}^y \rangle\ [\alpha] \rightarrow S_1 ... S_n \end{aligned}$$

A service is elementary or composite. It is characterized by output parameters $\overline{\overline{y}}$, input parameters $\overline{\overline{x}}$, possibly the effects on the output parameters $\overline{e_b}^y$, possibly preconditions on input parameters $\overline{e_b}^x$ and an identifier (its name). It can be composed of other services $S_1 ... S_n$. α represents the location of service. It should be noted that α may be unnecessary for services on the RHS if they are implemented in the same user space as the services from the LHS.

For reasons of readability, we prefer the previous notation. In pi-calculus notation, it corresponds to:

$$\begin{aligned} S \ ::= \ & [\overline{e_b}^x]\ \alpha\ (id, \overline{\overline{x}}, p)\ .\ [\overline{e_b}^y]\ p!\overline{\overline{y}}\ \mid \\ & [\overline{e_b}^x]\ \alpha\ (id, \overline{\overline{x}}, p)\ .\mathbf{(}vp_1\ \alpha\ \langle id_1, \overline{\overline{x}}_1, p_1 \rangle\ \mid \\ & ...\ \mid\ S_i ...\ \mid\ vp_n\ \alpha\ \langle id_n, \overline{\overline{x}}_n, p_n \rangle \mathbf{)}\ .\ [\overline{e_b}^y]\ p!\overline{\overline{y}} \end{aligned}$$

The S service expects $\overline{\overline{x}}$ as the parameter, executes and returns $\overline{\overline{y}}$. It receives the data on the public port of the peer where it is hosted. when the RHS is present, it calls the services it contains to build y. The services on the RHS can be executed in parallel if the data are independent of each other or in sequence if there is dependency hence the parallel operator(|) inside the brackets in bold. Once a call of service performed, we get the service instances. These notations are also used to describe the instances.

The **Service instance** or **Artifact** is the instantiation of a service. This makes it possible to follow the execution of the service.

$$
\begin{aligned}
I \quad ::= \quad & id(\ \overline{\overline{x}}, \overline{e_b}^{x})\ \langle \overline{\overline{y}}, \overline{e_b}^{y} \rangle\ [\alpha]\ \mid \\
& id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}, \overline{e_b}^{y} \rangle\ [\alpha]\ \mid \\
& id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}_r, \overline{e_b}^{y} \rangle\ [\alpha]\ \mid \\
& id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}, \overline{e_b}^{y} \rangle\ [\alpha] \rightarrow I_1 ... I_n\ \mid \\
& id(\ \overline{\overline{x}}_r, \overline{e_b}^{x})\ \langle \overline{\overline{y}}_r, \overline{e_b}^{y} \rangle\ [\alpha] \rightarrow I_1 ... I_n
\end{aligned}
$$

A service instance has several configurations:

- the input and the output parameters are not yet resolved;
- resolved input parameters and output parameters not yet resolved;
- resolved input and output parameters.

Each element that appears on the right, when it exists, can have one of the three previous configurations. The parameters of the service instances are resolved as they are executed.

3.2.3. Action

An **action** is a send message, a receive message or a silent interpretation of the service instances.

$$
\begin{aligned}
act \quad ::= \quad & vp\ \overline{\alpha}\ \langle M, p \rangle \\
\mid \quad & \alpha(M, p) \\
\mid \quad & p(M) \\
\mid \quad & \overline{p}\ \langle M \rangle \\
\mid \quad & r
\end{aligned}
$$

The **actions** define the communication to be performed between the services execution spaces:

- $vp\ \overline{\alpha}\ \langle M, p \rangle$ allows, firstly, the creation of a variable representing the private port (p), then send a message M and p to the public port α of a remote space;
- $\alpha(M, p)$ receiving a message M and a variable p on the public port α;
- $p(M)$ receive a message M on a private port p ;
- $\overline{p}\ \langle M \rangle$ sending a message M on a private port p ;
- r silent action which consists of interpreting the services parameters using semantic rules. No communication with the environment.

3.2.4. Message

Messages are variables that are exchanged on the network. It contains global variables (context variables). It is composed of defined variables and/or not defined variables. There are two types of messages: request message (variables in defined inputs and variables in undefined output) and response message (defined input variable and output variable defined).

$$M ::= \ \overline{\overline{x}}_r\ \overline{\overline{y}}\ id\ \ (request) \mid \overline{\overline{x}}_r\ \overline{\overline{y}}_r\ \ (response)$$

A **sending message** (request) comprises 3 parts: resolved input $\overline{\overline{x}}_r$, outputs to be resolved $\overline{\overline{y}}$ and the identifier of the service to which the request is intended. A **response message** consists of 2 parts: resolved inputs $\overline{\overline{x}}_r$ and resolved outputs $\overline{\overline{y}}_r$. As we will see in the next section, the response is transmitted along a private port created during the request, hence the absence of the service identifier.

3.3. Behavioral Description

The following operational semantics describes the mechanisms for resolving services, which is divided into several fundamental operations: sending and receiving of the message, refinement, instantiation and choice of services.

The execution space or **peer**(Σ) contains services, instances of services and is characterized by its location. Let us denote by I_s the set of service instances, S_s the set of services and α its address (main port or location). Thus the space of execution is characterized by $\Sigma = (S_s, I_s, \alpha)$. In the following, the different operations to be applied on an execution space for the resolution of the services, $fn(e)$ is set of bound variables of the entity e.

Intanciation

$$\frac{\Sigma' = \Sigma \cup I,\ p \notin fn(\Sigma)}{\Sigma : S = id(\overline{\overline{x}})\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{\alpha(M,p)} \Sigma' : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]}\, C_1$$

$$\frac{\Sigma' = \Sigma \cup I,\ p \notin fn(\Sigma)}{\Sigma : S = id(\overline{\overline{x}})\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow[\rightarrow \mathrm{S_1}...\mathrm{S_n}]{\alpha(M,p)} \Sigma' : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \mathrm{I_1}...\mathrm{I_n}}\, C_2$$

When Σ receives on its main port α the message named M and the variable named p, it finds the corresponding service named S and creates the instance named I with the input which are defined $\overline{\overline{x}}_r$ and the outputs $\overline{\overline{y}}$ awaited. I is added to Σ which becomes Σ′. If no service is found, the operation is not applied.

C2 is the extended version of C1, the service found is composed.

Response

$$\frac{}{\Sigma : I \xrightarrow{p\ \langle M\rangle} \Sigma : I}\, Resp$$

Response on p (private port) of a previously sent request. M=$\overline{\overline{x}}_r\ \ \overline{\overline{y}}_r$

Request

$$\frac{\begin{array}{l} I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]\ or \\ = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \mathrm{I_1}...\mathrm{I_n}\ or \\ = I_0 \rightarrow I_1...id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha]...I_n \end{array}}{\Sigma : I \xrightarrow{vp\ \overline{\alpha}\ \langle M,p\rangle} \Sigma : I}\, Req$$

The I of Σ sends the M on α. This does not change the state of the execution space; M is constructed from parameters of the instance to be concretized.

Refinement

$$\frac{}{\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{r} \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]}\, R1$$

$$\frac{}{\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \xrightarrow{p(M)} \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha]}\, R2$$

$$\frac{\overline{\overline{x}}' \subseteq \overline{\overline{x}}}{\begin{array}{ll}\Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow & \Sigma : I = id_0(\overline{x}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow \\ \quad I_1 ... id_i(\overline{\overline{x}}')\langle\overline{\overline{y}}'\rangle[\alpha'] & \quad I_1 ... id_i(\overline{\overline{x}}'_r)\langle\overline{\overline{y}}'\rangle[\alpha'] \\ \quad ... I_n & \quad ... I_n\end{array}} R3$$

$$\frac{\overline{\overline{x}}'' \subseteq \overline{\overline{y}}'}{\begin{array}{ll}\Sigma : I = I_0 \rightarrow I_1 ... & \Sigma : I = I_0 \rightarrow I_1 ... \\ \quad id_i(\overline{\overline{x'}_r})\langle\overline{\overline{y'}_r}\rangle[\alpha'] ... & \quad id_i(\overline{\overline{x'}_r})\langle\overline{\overline{y'}_r}\rangle[\alpha'] ... \\ \quad id_j(\overline{\overline{x''}})\langle\overline{\overline{y''}}\rangle[\alpha''] ... I_n & \quad id_j(\overline{\overline{x''}_r})\langle\overline{\overline{y''}}\rangle[\alpha''] ... I_n\end{array}} R4$$

$$\frac{}{\begin{array}{ll}\Sigma : I = I_0 \rightarrow I_1 ... & \Sigma : I = I_0 \rightarrow I_1 ... \\ \quad id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] ... I_n & \quad id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha] ... I_n\end{array}} R5$$

$$\frac{\overline{\overline{y}} \subseteq U\overline{\overline{y}}_i}{\begin{array}{ll}\Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle[\alpha] \rightarrow & \Sigma : I = id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle[\alpha] \rightarrow \\ \quad id_1(\overline{\overline{x}}_{1r})\langle\overline{\overline{y}}_{1r}\rangle[\alpha] & \quad id_1(\overline{\overline{x}}_{1r})\langle\overline{\overline{y}}_{1r}\rangle[\alpha] \\ \quad ... id_n(\overline{\overline{x}}_{nr})\langle\overline{\overline{y}}_{nr}\rangle[\alpha] & \quad ... id_n(\overline{\overline{x}}_{nr})\langle\overline{\overline{y}}_{nr}\rangle[\alpha]\end{array}} R6$$

The refinement of an instance is the materialization of not yet defined parts. The action is silent or the receipt of a response on a private port.

R1: Calling a primitive service (automatic or manual)

R2: receive information on a private port for an instance of service.

R3, R4 and **R6:** Allow the definition of the parameters of some service instances to the RHS from the parameters already defined.

R5: Upon receipt of a response, materialize the part of the service instance that made the request.

Local choice of Service (LoCh)

$$\frac{id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \;\; match \;\; to \;\; I'_0 \rightarrow I'_1 \;\; ... \;\; I'_n}{\begin{array}{ll}\Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \rightarrow & \Sigma : I = id_0(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle \rightarrow \\ \quad I_1 ... id_i(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle ... I_n & \quad I_1 ... [I'_0 \rightarrow I'_1 \;\; ... \;\; I'_n] ... I_n\end{array}}$$

select a service locally.

The focus is on the data that influences the choice of services, their creations and their refinements. The execution flow depends on the availability of the values of the input and output variables. In a schema of service composite, two tasks execute in parallel if the inputs of one do not depend on the outputs of the other and in sequence if the entries of one depend on the outputs of the other.

3.4. Correspondence with Traditional Process Creation Activities

The *GSLang* is based on the rules that describe a service by specifying its name, input and output parameters, semantic rules (describing the correspondence between attributes) and services to solve it. According to the dependency of the attributes parameters and the enchainment of services within the rules, we have sequential instructions, conditional expressions, loops, and exceptional case handling.

3.4.1. Sequence

It can happen only in a rule defining a service s, the input parameters to a service s_2 correspond to the output parameters of a service s_1. Then services s_1 will execute before service s_2. For e.g., $s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s_2(y')\langle y\rangle$

3.4.2. Condition

In a Σ execution space, a service may have several different definitions, so the conditional expression will guide the choice of one over the others. For example if $a > 0$ then the first definition will be chosen otherwise, if $a = 0$ it is the second one.
$[a > 0]s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s_2(y')\langle y\rangle$
$[a = 0]s(x)\langle y\rangle \rightarrow$

3.4.3. Loop

In the sequence of a rule defining a service s, we can have the service s itself in its RHS or in the RHS of a service present in the RHS of s. Defined thus, we have a loop. For example
$[x > 0]s(x)\langle y\rangle \rightarrow s_1(x)\langle y'\rangle s(y')\langle y\rangle$
$[x = 0]s(x)\langle y\rangle \rightarrow$
It is very difficult in practice to determine that the service will run until the end. To do this, we transform the service specifications into Promela/SPIN in order to validate them.

3.4.4. Parallele (fork/join)

If the two services s_1 and s_2 present on the right side of a service have nothing in common, then s_1 and s_2 can run at the same time. It can happen that the output of s depends on the outputs of s_1 and s_2 as in the example. $s(x)\langle y_1, y_2\rangle \rightarrow s_1(x)\langle y_1\rangle s_2(x)\langle y_2\rangle$

3.4.5. Exceptional Case Handling

As the specification of services is declarative, adding specific rules with conditions can help manage exceptional cases. This declarative description allows you to add rules at any time to handle exceptional cases.

3.5. Service Resolution

The operational semantics defined above consist in creating and materialize the service instances dynamically at runtime according to the input and output parameters and the port created during execution. In the system a service instance I is resolved if all these parameters are defined (input and output). Any action on a resolved service does not change it. i.e.,

$$\text{I is resolved} \iff I^*$$

The semantics of this operator are as follows: if I^* and $\text{I} \xrightarrow{\alpha} I'$ then I'^*.

Theorem
A service instance I to which are attached the input variables $\overline{\overline{x}}$ and the output variables $\overline{\overline{y}}$ is resolved if all the associated service instances I_i are resolved i.e.,

$$\frac{I \rightarrow I_1 ... I_n,\ I_1^*,\ I_2^* ...,\ I_n^*}{I^*} where\ I_i^* = id(\overline{\overline{x}}_{ir})\langle\overline{\overline{y}}_{ir}\rangle$$

Proof

If the instance is simple i.e., it is defined by $I \rightarrow$, then I is resolved if the values are assigned to the input parameters x_r and the output parameters y_r.

If the instance is composite i.e., $I \rightarrow I_1 ... I_n$

- A instance like $I \rightarrow I_1$, by definition, $x \subseteq x_1$ and $y \subseteq y_1$ if I_1 is resolved i.e. we have x_{r1} and y_{r1} then x and y are defined i.e. x_r and y_r.
- A instance like $I \rightarrow I_1 ... I_n$ by definition, The input (resp. output) variables of the LHS x (resp. y) depends on the input (resp. the output) parameters of the RHS: $x \subseteq \cup_i x_i$ (resp. $y \subseteq \cup_i y_i$). if $\forall_i I_i$ is resolved i.e. we have the x_{ir} and y_{ir}, $\Rightarrow$ we have x_r and y_r.

Proposition

The operational semantics generate a $\perp$-transitions system.

Proof

Let a labeled $\perp$-transition system defined by a n-tuple $(\Sigma, S, \rightarrow, \perp)$ where
- Σ is a set of **actions** which label the transitions;
- $Q = S \cup I$ is the set of states where S is the set of **services** and I is the set of **instances**;
- $\delta \subseteq Q * \Sigma * Q$ the transition function
- $\perp \in S * bool$ is a termination predicate such that $\perp (s) = true$ and $s \xrightarrow{\alpha} s\prime$ then $\perp (s\prime) = true$

The transition system describes the behavior of a service when it is chosen in a user space. In Figure 1 , all the operations of the operational semantics are used to describe how an instantiated service will progress to be solved.

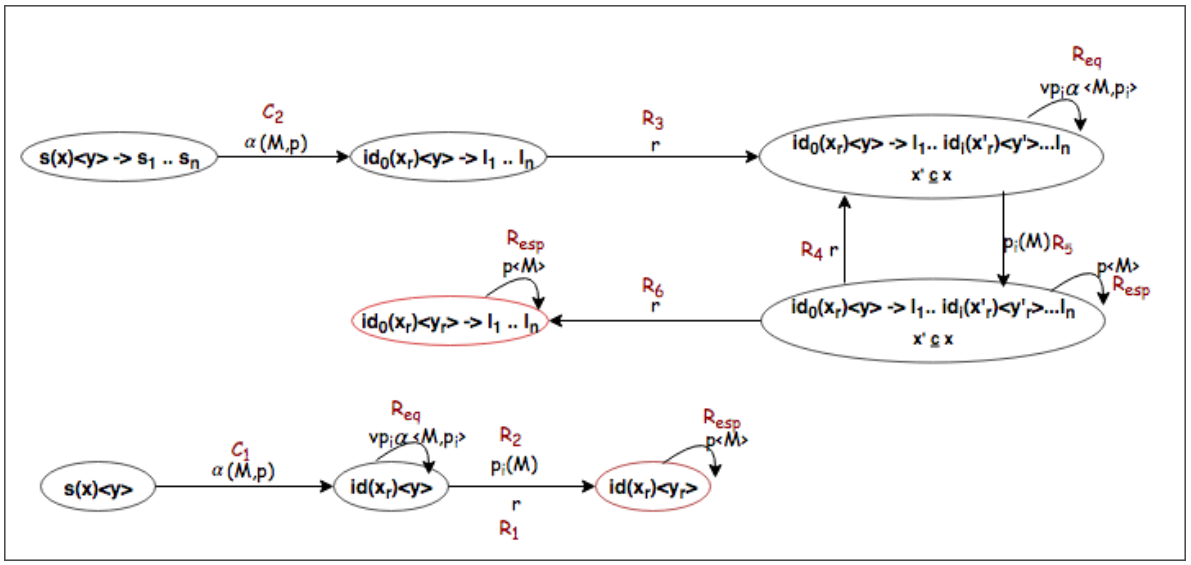

Figure 1. Progression of an instantiated service. Each transition is a rule of the operational semantics presented in Section 3.3

The operations convert the system from one state to another through attribute interpretation (silent action), sent and received messages. These latter correspond in the transition system to labelled edges. The services and the instances are the nodes.

When we look at the instantiation rule C1, if the system containing a service of the form id($\overline{\overline{x}})\langle\overline{\overline{y}}\rangle$ (node of the transition system) receives a message ($\alpha(M, p)$) (edge in the transition system) then the system creates an instance of the form $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle$. When the instance $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}\rangle$ is created, one can call the related service using the request $vp_i\alpha\langle M, p_i\rangle$ (rule Req). The receiving of the response to this $p_i(M)$ request (rule R2) allows refining the initial service thus defining the output $id(\overline{\overline{x}}_r)\langle\overline{\overline{y}}_r\rangle$.

On the other hand, for a composite service, applying rule C2 when receiving a message on the public port of a peer $\alpha(M, p)$, creates the instance with the values of the services of the LHS defined $(id_0(\overline{\overline{x}}_r)\langle y\rangle \rightarrow I_1..I_n)$. Therefore a refinement can be applied (rule R3) which allows through the semantic rules to define certain parameters of the RHS. At this point, the RHS services can be applied, requests $vp_i\alpha(M, p_i)$ are sent to another peer, responses received on private ports $p_i(M)$ (R5 rule) sent during requests. Refinements are applied (rules R4 or R6). When all the outputs are set then we will reach the final state where the outputs of the LHS service are defined.

The verification of this semantics is demonstrated using SPIN tools [31]. The pi-calculus specification is translated into Promela and SPIN is used to validate the proposed model. We want to show that if the system is well specified, each instantiated service in the system will run until the end.

4. Verification Framework

The process of verifying and validating the above mentioned specification is described in Figure 2. It consists of the two phases:

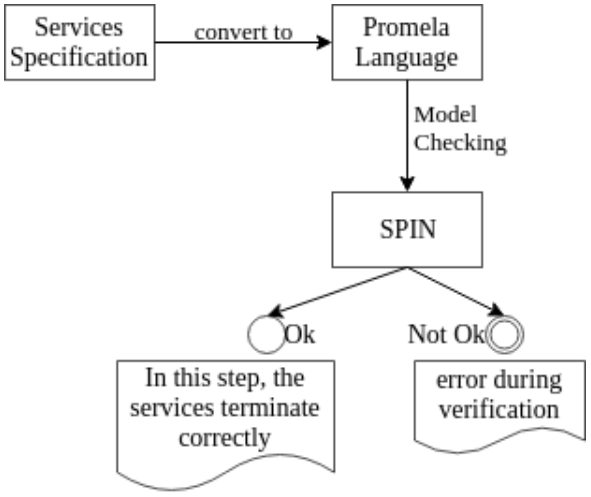

Figure 2. Verification process: the service specification is converted into PROMELA code for model checking.

- (i) Translate a service definition to a Promela specification;
- (ii) Simulate and verify the Promela specification to validate the properties.

4.1. Translate Service Specification to Promela Language (Translation Template)

The Promela language is an abstract specification language of a system. It models the synchronization and coordination of processes. Our choice was based on Promela, because it allows in a message to send a channel a bit like in the pi-calculus [32,33]. A message contains a dataset as well as a port on which the remote process will transmit its response.

Table 1 presents the correspondences between pi-calculus concepts and Promela code.

Table 1. Transformation rules from theoretical aspect to Promela language. Variables are translated into mtype, ports into channels, services into block structures and roles into processes.

Service Specification	Promela Code	Comment
variable, value and port	mtype, byte, chan	use *mtype* or *byte* to define variable and value And *chan* to define port
$x \leftarrow u \equiv x_r$	x = u	assignment the value (u) to the variable (x)
$\overline{e}_b^x$	==, <, >, <=, ! =, >=	promela conditional expressions on $\overline{\overline{x}}$ for e.g $x_1 == x_3$, $x1 > u$, x_1 != x_3. We will note them by **c**.
$s(\ \overline{\overline{x}}, \overline{e}_b^x)\ \langle \overline{\overline{y}}, \overline{e}_b^y \rangle [\alpha]$	**begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** :: $(s == id\ \ \&\&\ \ c\ \ \&\&\ \ \overline{\overline{x}}) \rightarrow p!\overline{\overline{y}}$; **goto begin;** **fi**	where α is the public port of user space(peer) containing **s** in the service specification, **p** is the private port contained in the request and used like the return channel. The Promela code translates the receipt of $s, \overline{\overline{x}}\ and\ p$ on α, the choice of the appropriate implementation according to the guard and the input parameters. it returns $\overline{\overline{y}}$ on **p**.
$s(\ \overline{\overline{x}}, \overline{e}_b^x)\ \langle \overline{\overline{y}}, \overline{e}_b^y \rangle [\alpha] \rightarrow$ $s_1(\ \overline{\overline{x}}_1)\ \langle \overline{\overline{y}}_1 \rangle [\alpha_1]$ $s_2(\ \overline{\overline{x}}_2)\ \langle \overline{\overline{y}}_2 \rangle [\alpha_2]$ $s_3(\ \overline{\overline{x}}_3)\ \langle \overline{\overline{y}}_3 \rangle$ ... $s_n(\ \overline{\overline{x}}_n)\ \langle \overline{\overline{y}}_n \rangle [\alpha_n]$	**begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** :: $(s == id\ \ \&\&\ \ c\ \ \&\&\ \ \overline{\overline{x}}) \rightarrow$ $\alpha_1!s_1, \overline{\overline{x}}_1, p_1$ $p_1?\overline{\overline{y}}_1$; $\alpha_2!s_2, \overline{\overline{x}}_2, p_2$ $p_2?\overline{\overline{y}}_2$; y_3 //define y_3 ... $\alpha_n!s_n, \overline{\overline{x}}_n, p_n$ $p_n?\overline{\overline{y}}_n$; $p!\overline{\overline{y}}$; **goto begin;** **fi**	Meaning that upon receipt of $s, \overline{\overline{x}}\ and\ p$ on α, the choice of the appropriate service is made according to the input parameters and the guard (c). Then the services on the right side are processed in parallel or sequentially depending on whether the outputs correspond to inputs or not. Promela process is a peer. Private ports (p1, p2 ... pn) are created within the process representing the peer. Finally, sends the final response $\overline{\overline{y}}$ on channel **p**.
$\Sigma = (S_s, I_s, \alpha)$	chan α = [k] of {mtype,mtype chan}; **Proctype peerName(){** chan p1 = [2] of {mtype,mtype,chan}; ... chan pn = [2] of {mtype,mtype,chan}; **begin:** $\alpha?s, \overline{\overline{x}}, p$ **if** :: $(c_1) \rightarrow block_1$ **goto begin;** :: $(c_2) \rightarrow block_2$ **goto begin;** ... :: $(c_k) \rightarrow block_k$ **goto begin;** **fi** }	The promela process is a peer. The public ports are globally defined channels, private ports are local. The different services are the alternatives of the if instruction. We do not differentiate services from their instances. The code promela will be simply executed

4.2. Model Checking

The processes generated in Promela are simulated and verified using SPIN. When a new service is added, it is transformed into Promela and verified. The client process sends a request and waits for a result. When the client process receives the result, then the termination property for the called process (service) is verified. When for a verified service, all private ports receive the data, then the

service terminates. The property of termination is therefore verified. In LTL (Linear Temporal Logic) this means: $[\,]\,(p?x)$ meaning that when a private port p is created, it always receives data. p being a created private port when running a service s and x being the received data.

The formal verification of the model contributes to the demonstration of the soundness property which is too difficult to proof theoretically [8].

Table 2 describes an example of services and their transformations into Promela distributed on three user spaces. The Promela transformation is then simulated. Three scenario are presented

scenario 1. The services were correctly specified (Figure 3); The client process calls the service s from *space1* on his public port. The *space1* process receives and instantiates s. s calls s_1 in *space2* by creating private port 4 on the $figure$, s_2 in *space3* by creating private port 5 on the figure and define y_3 locally. Finally, the client process receives the reponse awaited.

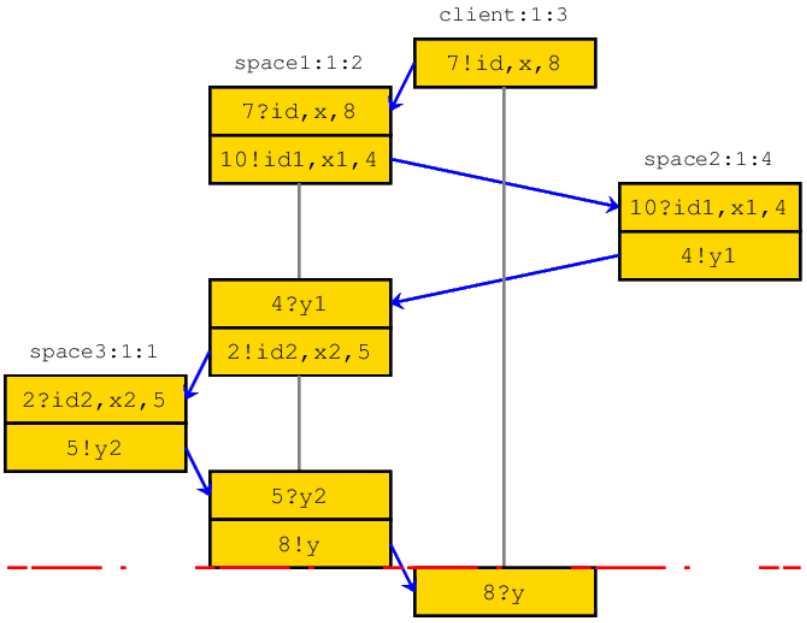

Figure 3. Correct execution: the client process obtains the requested response.

scenario 2. The parameters of the called services do not match (Figure 4). From Table 2, before test, the input parameters for the service s_1 does not match the space 1 and 2;

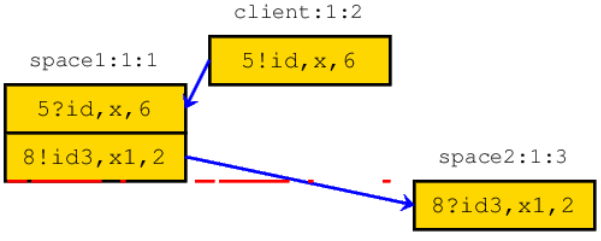

Figure 4. Not matching service: service parameters do not match. This implies the call of the client process does not execute until the end.

scenario 3. The called service is not defined (Figure 5). From Table 2, before test, s_2 service has been removed.

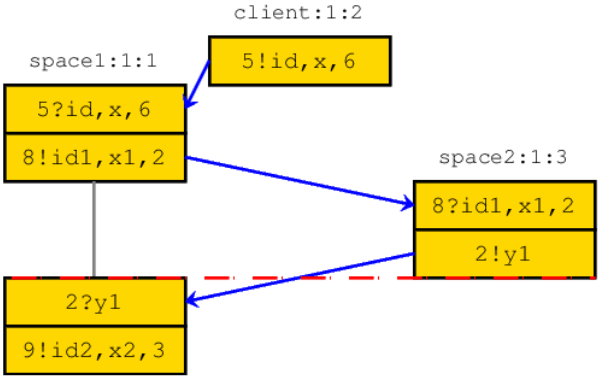

Figure 5. Undefined service: the requested process does not exist. This implies the call of the client process does not execute until the end.

Table 2. Translation of each of spaces 1, 2 and 3 to PROMELA processes

Space	Promela Code
space 1 $s(\,\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$ $s_1(\,\overline{\overline{x}}_1)\ \langle\overline{\overline{y}}_1\rangle[\alpha_1]$ $s_2(\,\overline{\overline{x}}_2)\ \langle\overline{\overline{y}}_2\rangle[\alpha_2]$ $s_3(\,\overline{\overline{x}}_3)\ \langle\overline{\overline{y}}_3\rangle$	**Proctype space1(){** **begin:** $\alpha?s,x,p$ **if** $::\ (s == id) \rightarrow$ $\alpha_1!id_1,x_1,p_1;$ $p_1?y_1$ $\alpha_2!id_2,x_2,p_2;$ $p_2?y_2$ $y_3\ \ //define y_3$
$s_3(\,\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	$p!y$ **goto begin;** **fi** }
space 2 $s_1(\,\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space2(){** **begin:** $\alpha_1?s,x,p$ **if** $::\ (s == id1) \rightarrow$ $p!y_1$ **goto begin;** **fi** }
space 3 $s_2(\,\overline{\overline{x}})\ \langle\overline{\overline{y}}\rangle \rightarrow$	**Proctype space3(){** **begin:** $\alpha_2?s,x,p$ **if** $::\ (s == id2) \rightarrow$ $p!y_2$ **goto begin;** **fi** }

5. Technical Framework

This section presents a design and execution environment for the proposed language, an example of modeling in the environment, and the contributions of our approach. Figure 6 summarizes our validation framework.

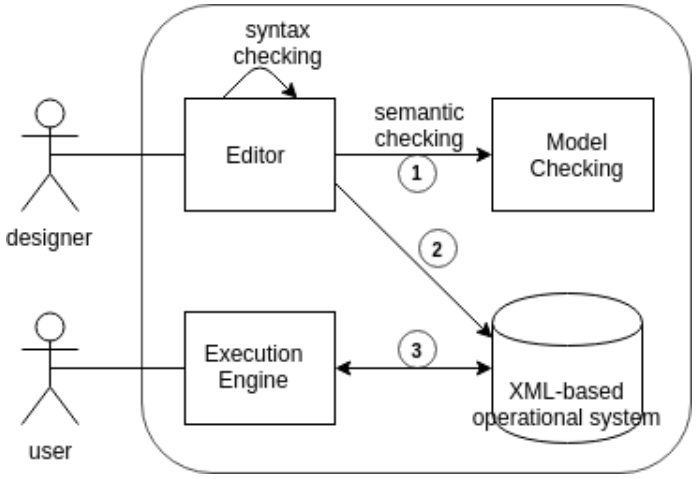

Figure 6. Validation Framework: 1. The services specified in the editor are transformed into Promela for model checking. 2. The services specified are transformed into XML for operationalization. 3. The execution engine manages the XMLs.

1. check that all service specifications are consistent or that any changes in the services are consistent.;
2. refinement of the service specification for the peer to the operational system when the point 1 is correctly done. Here the elements like the nature (RESTfull or Manually) of the services and their location are added;
3. the execution engine uses XML-based operational system to meet users needs by dynamically combining different services.

5.1. Tools

A prototype has been developed. It consists of an editor, a transformation engine and an execution engine.

5.1.1. **The editor**

We have developed a DSL for the proposed language using Xtext [34,35]. The eclipse plugins obtained allows doing a syntax check when editing the services and refine the specification to the operational system for the role when the verification succeeds. The EMF-based Abstract Syntax Tree in Figure 7 makes it possible to check this syntax.

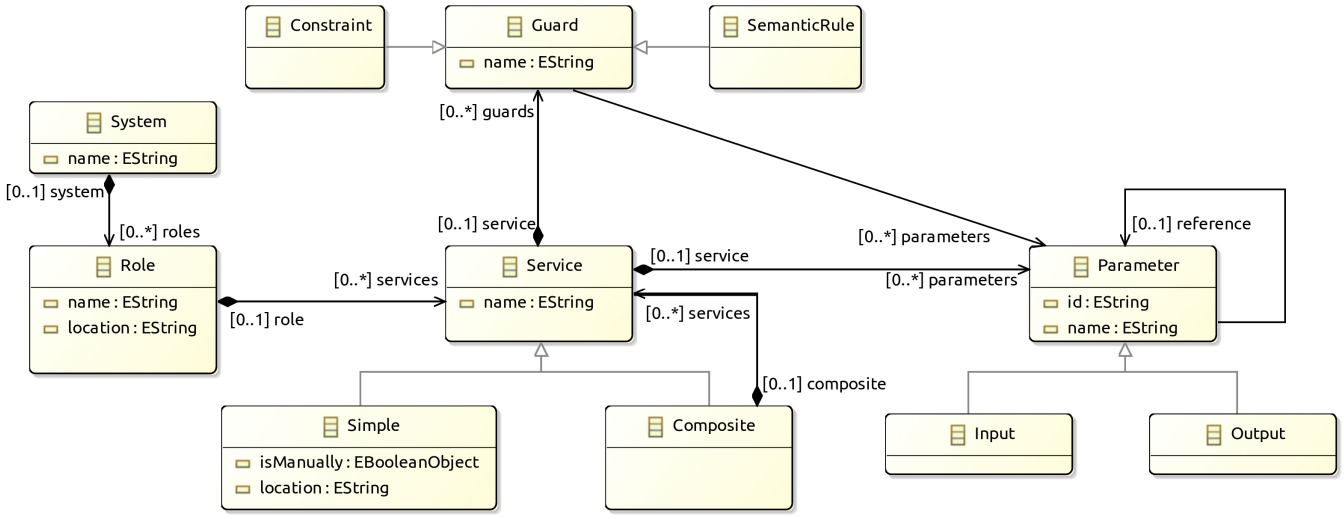

Figure 7. EMF-based Abstract Syntax Tree for the Services Specification

5.1.2. The transformation and semantic verification engine

The transformation of Table 1 has been automated using the ATL language [36]. The source is the Abstract Syntax Tree in Figure 7 and the target is the Promela meta-model presented in document [37]. The main transformation rules are:

- **guard2Condition** rule: transforms the guards associated with the service into a condition for IF statement of promela.
- **service2ChannelAndSimpleStatement** rule: transforms simple services into channel-type variables and simple statement in a promela proctype process.
- **service2ChannelAndCompositeStatement** rule: transforms composite services into channel-type variable and composite statement such as the If statement
- **role2Process** rule: transforms a role (peer) into a promela process.
- **system2Model** rule: transforms the system composed of the set of roles into a promela model composed of all the processes.

A skeleton of the Transformation Rules Code is shown in Figure 8.

SpecServToProme | Promela class d | specService cla | promela.ecore | specServExample

```
rule Sytem2Model {
    from syst: specService!System
    to model : promela!Model (
        name <- syst.name,
        variables <-
            Set {
                syst.roles ->  collect (e | thisModule.resolveTemp(e, 'chan1')),
                specService!Service.allInstances()  ->  collect(e | thisModule.resolveTemp(e, 'prim1')),
                specService!Service.allInstances()  ->  collect(e | thisModule.resolveTemp(e, 'prim11')),
                specService!Input.allInstances() -> collect(e | thisModule.resolveTemp(e, 'prim2')),
                specService!Output.allInstances() -> collect(e | thisModule.resolveTemp(e, 'prim3'))
            } ->flatten() ,

        processes <- syst.roles -> collect(e |thisModule.resolveTemp(e, 'process1')) ->flatten()

    )
}
rule Role2Process{
    from rol3 : specService!Role
    to chan1 : promela!Channel (
        size <- 5,
        type <- #byte,
        typeName <- '{mtype,mtype,chan}',
        name <- rol3.name+'pub',
        model <- thisModule.resolveTemp(rol3.system, 'model')
    ),
    process1 : promela!Process(
```

Figure 8. skeleton of transformation rules. For example, System is translated into Promela Model, Role is translated to Promela Process.

5.1.3. The Execution Engine

The execution engine consists of two parts:

- Web interfaces developed in Java using the JSF/Primefaces framework.Each role can use these interfaces to visualize the artifact contained and their instances. The web interface is shown in Figure 9. After the login of a role, it presents the services available for the role (at the top) and the service instances that are running (at the bottom).
- A communication medium that is a middleware of the Message Oriented Middleware (MOM) type because the MOMs allow interactions between application components in a loosely coupled, asynchronous and reliable framework. For this purpose, JMS (Java Message Service) was used.

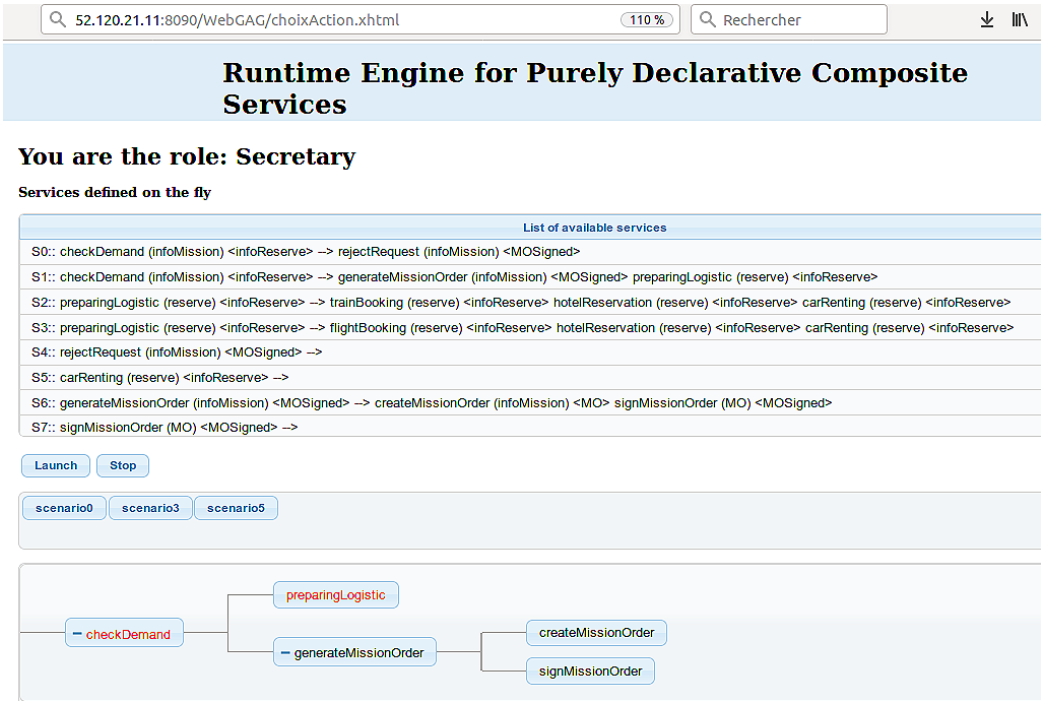

Figure 9. Main interface of the Execution Engine, the top part shows the specified services and the bottom part the instance of the running service.

In the following section, an example shows how modeling and execution are implemented.

5.2. *Case study*

5.2.1. Case Description

Let us consider a simplified mission management system (MMS) used by an organization to organize the business travel of its employees. This case study completes the description of the example which have been presented at the beginning of this paper. It is selected firstly to show the use of the proposed language on a concrete example. Secondly, to validate the environment which is put in place. In this regard, this example involves three actors: Employee, Secretary, Head of Department (HOD), and four services provided by external structures. The employee requests a mission order by filling a form which is submitted to the Secretary. The Secretary checks the validity of the demand. If it is approved, the form is transmitted to the HOD while the logistic is prepared. The latter action is made up with external services. It uses *Flight Booking* or *Train Booking* service, *Hotel Reservation* service and *Car Rent* service. Figure 10 shows the overview of the example.

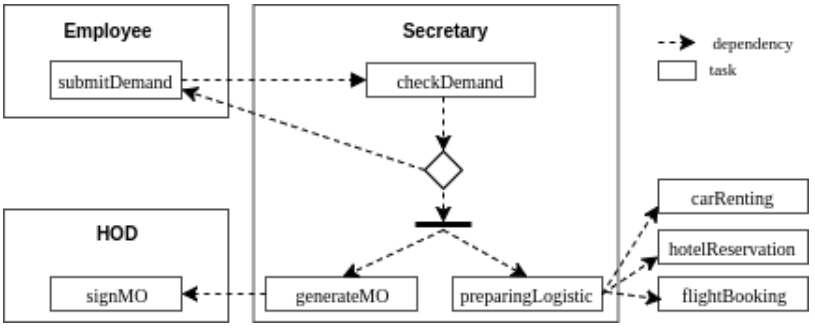

Figure 10. Structure of the Mission Management System (MMS)

5.2.2. Modelling in the Environment and Verification

The current version of the editor is textual. It offers the possibility of editing and checking the syntax of the services.

Figure 11 shows the edition of the rules associated with the example of the previous section. Each role is represented, in each role we have the associated services. For example in the role Employee we have the *submitDemand* service.

Figure 11. Modeling in the Editor : The variables, the external services and the defined services of each role are presented.

Once the modeling is completed, the code is verified and in case of success, the transformation engine translate it into Promela for semantic verification. Then, the editor generates the XML model which is loaded in the execution engine.

In summary

- each peer contains the editor and all the service specification of the system because the verification is done over the entire specification;
- on a peer, the operational services generated are only those of the roles;
- an update of the service specification is reflected in the different peer after the verification has been successful.

5.2.3. Execution

The architectural pattern induced by the model is a P2P style implemented on the top of the chosen MOM as shown in Figure 12.

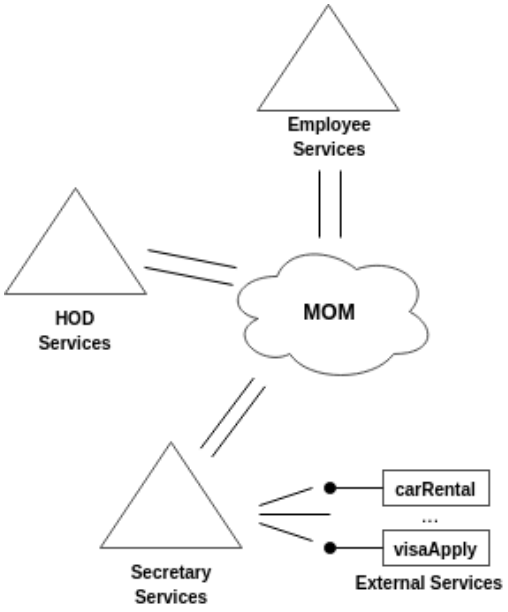

Figure 12. The architectural style of the execution engine is P2P. The employee, the HOD and the Secretary exchange data via MOM-type middleware.

The execution engine has a visual aspect (web interface) and a backend that implements the rules of section 3.3 and is based on the MOM.

The debugging aspect of this work is the possibility offered to the users to observe the evolution of the service instances as shown in Figure 13 where an instance of the *checkDemand* service is presented. The refined nodes as well as related information can be viewed. We can request each node of the *checkDemand* service for the details of these informations.

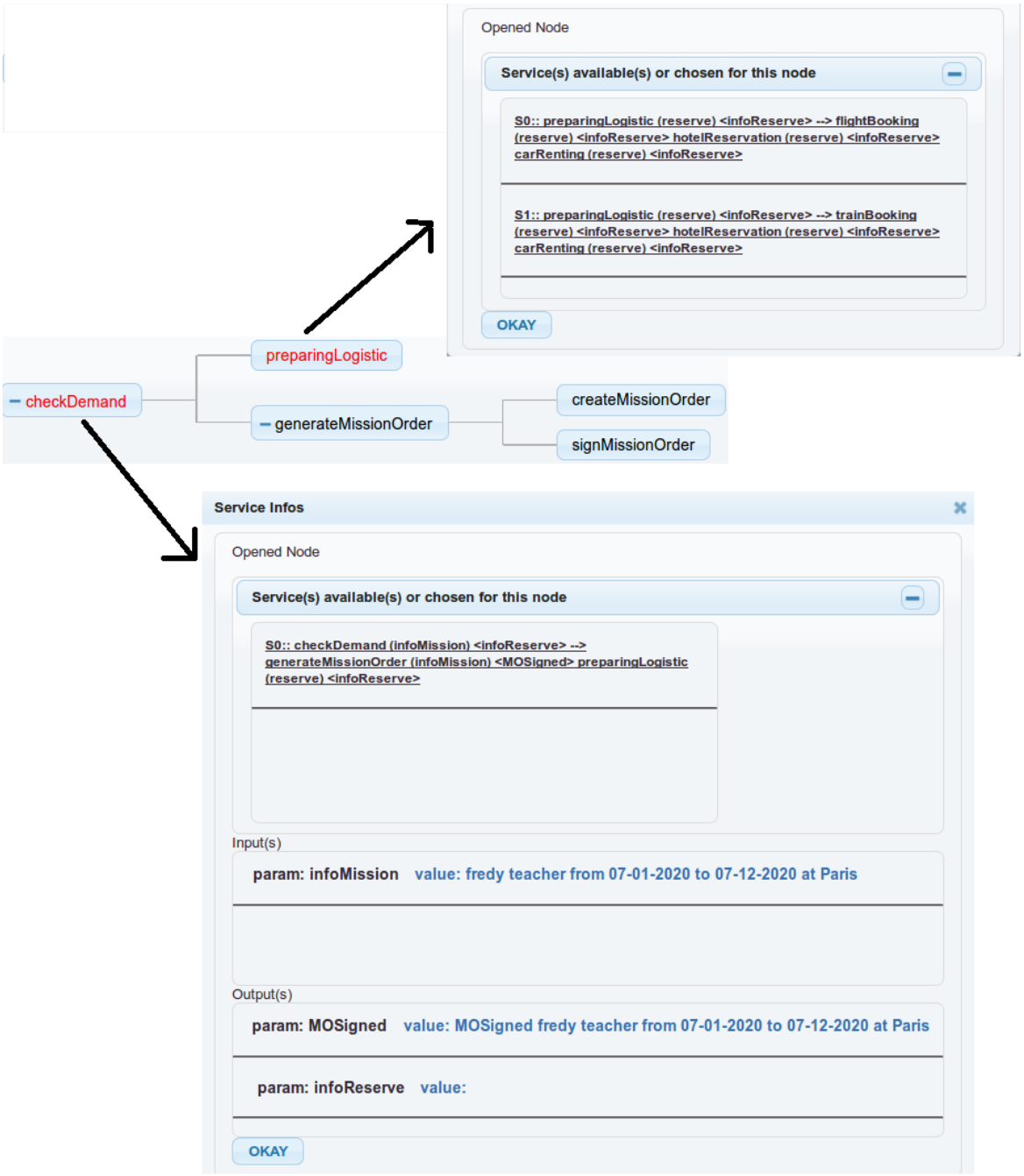

Figure 13. A runtime instance of the CheckDemand service. At this stage, preparingLogistic proposes to choose between S_0 and S_1 and the checkDemand service is still waiting for the values of the infoReserve variable.

The black nodes are the services already treated and the red nodes are the pending services waiting for an event or information to continue their execution.

The schema of a service can be updated in the editor and reload in the execution engine without stopping the running processes. This modification will take effect next time the given service is chosen to be applied.

6. Discussion

Several studies have been interested in the composition of services and more recently declarative languages have been proposed in order to provide more flexibility and adaptation during the execution

of services. In this part, The choreography (peer to peer architecture) as well as the declarative paradigm for the composition of services are discussed. We end by highlighting the data-driven in the composition of services.

6.1. Architecture Style

As an alternative to traditional languages and notations for service compositions such as BPEL [10] and BPMN [38], other languages, such as JOpera [39] and Jolie [40] have been proposed. Nevertheless, they favor the orchestration, a single point of view for the composition of service. Being of type orchestration, they do not allow to consider the collaborations of peer to peer. Our language is instead of a choreographic type.

Few approaches focus on choreography for compositional modeling. Inspired by the WS-CDL [12] standards, over the past decade, choreography has been studied to support a new programming paradigm called choreographic programming [41,42]. In choreographic programming, the programmer uses choreography to program the service systems, and then a compiler is used to automatically generate compliant implementations. This gives an accuracy method by construction, guaranteeing important properties such as freedom of blocking and the absence of communication errors. This aspect of pre-deployment verification is important in the collaboration between services, which is why in our language we transform our specification into promela. Nevertheless, the choregraphy programming allows a global description of the interactions between the different sites. In other words, the interactions are known a priori what we do not want. It is difficult to change running choreography when services are added/removed as needed by the designer.

The approaches such as BPEL4Chor [43] tend to add the choreography in orchestration style. Nevertheless, it relies on BPEL, which has been criticized for its rigid character.

6.2. Declarative Approaches

Many languages ([39,40,44]) have been proposed. Although easier [45] to use and often more expressive than BPEL and BPMN [38], they do not deviate from the imperative paradigm and, therefore, share with them the same limitations that motivated our work.

To provide flexibility by change [28], the declarative languages have emerged. Among them, declarative approaches such that Declare [46]. Declare is closest to our work considering its abstraction level. In Declare, service compositions are defined *as a set of actions and the constraints that affect them*. Actions and constraints are modeled, while constraints have a formal semantics given in linear temporal logic (LTL).

There are several differences between Declare and our approach: First, Declare focuses on the modeling of service orchestrations to support auditing and monitoring while our language focuses on the modeling of choreographies. In addition, Declare focuses on specification and verification and does not provide specific mechanisms for handling run-time failures.

GO-BPMN [47] is another declarative language, designed as a goal-oriented extension for traditional BPMN. In GO-BPMN, business processes are defined as a hierarchy of goals and sub-goals. Multiple BPMN plans are attached to the "leaf" goals. When executed, they achieve the associated goal. These plans can be alternative or they can be explicitly associated with specific conditions through guard expressions based on the context of execution. Although this approach also tries to separate the declarative statements from the way they can be accomplished, the alternative plans to achieve a goal must be explicitly designed by the service architect and are explicitly attached to their goals. The engine does not automatically decide how the plans are built or replaced; it just chooses between the given options for each specific goal, and it does it at service invocation time. GO-BPMN relies on BPMN known as being rigid to change. In our approach for an instance of a given composite, we can modify the structure of the services not applied which makes it possible to change its way of executing itself.

SelfMotion [45] is a declarative language that uses Abstract Actions which specify the primitive operations that can be executed to achieve the goal; Concrete Actions, one or more for each abstract action, which map them to the executable snippets that implement them. As our approach, it manages the adaptation of composite services at runtime. Nevertheless, it does not check in order to prevent some runtime malfunction as it focuses more on implementation.

Concerning SELF-SERV [48], A major outcome of the project has been a prototype system in which Web services are declaratively composed. In SELF-SERV, the process model underlying a composite service is specified as a statechart whose states are labeled with invocations to Web services, and whose transitions are labeled with events, conditions, and variable assignment operations. Statecharts possess a formal semantic and offers most of the constructs found in contemporary process modeling languages (sequence, branching, structured loops, concurrent threads, inter-thread synchronization, etc.). This ensures that the service composition mechanisms and orchestration techniques developed within the SELF-SERV project can be adapted to other process modeling languages for Web Services such as BPEL4WS and WSCI . Although operating in a peer-to-peer environment and explicitly taking into account roles that is one of the characteristics of our approach, it based its design on imperative paradigms thus inheriting all its shortcomings.

6.3. Data-driven

The data are one of the important points, they allow to specify how the outputs derive from the inputs [21]. The operationnalization of the proposed language implies a data-driven approach since the models issued from the here defined language are piloted by the data. This vision is close to Active XML [49] where the remote services are inserted in XML documents. These services, so called intentional data, are enabled when loading the XML document. In our approach, the artifacts contain the concrete data which represents the already enacted service and the intentional data that represent the pending services which could be applied. Nevertheless Active XML is dedicated to data integration rather than service composition.

Mashups [50] assemble the services coming from many sources in one web page. Some disadvantages of the mashups are:

1. The SOA developers usually need to spend major effort to master many SOA technologies, e.g., BPEL, WSDL, SCA (Service Component Architecture), as well as tools, e.g., design-time IDE tools, and runtime middleware servers (SCA server or BPEL server). Most of these tools require major investment on hardware and software infrastructure;

2. These technologies cannot support service composition's customization on the fly because the process of service composition (design, development and testing) is usually conducted in IDE tool first according to customer requirements, and then deployed on the runtime server. After deployment, the composition logic is not easy to customize according to the changes of composite service requirements, as this involves to come back at the earlier phases of the development process.

7. Conclusions and Future Works

In this paper, we proposed a rule-based approach to the definition of composite services in order to support adaptability and flexibility by change at runtime. The formalism used is that of attributed grammars extended with guards where the structure of a service is represented in a declarative purely manner. This approach focuses on the description of the execution schema of the composition while the process-based approaches describe the execution path of the composition. Formal description and verification ensures the correct composition of web services. It also ensures that system works as it is expected. The syntactic elements as well as an operational semantics of this language are described in pi-calculus formalism. The operational semantics makes it possible to highlight the dynamism of the services defined through out the asynchronous ports created at runtime. A verification framework which consists of translating the defined services into promela is proposed. This is done in order to check the consistency of the specified services and their ending. The validation framework allows

the evaluation by proposing the tools. It includes an editor, a checker and an engine to execute the specification.

The relevance of our approach could be appreciated especially in the case of composing a very large number of services, or in the case of services that take time to execute. Many properties are highlighted in the model presented in this paper:

- **Declarative:** the language described is based on the rules that describe a service by specifying its name, input and output parameters, semantic rules (describing the correspondence between attributes) and services to solve it.
- **Abstract Specification:** The intentional definition of services allows a late concretization of the services thus favoring a weak culprit with the underlying technology and an adaptation (updating the rules) of the service even during its execution.
- **Connectivity (Asynchronous):** Sends and receives asynchronous messages for the creation and refinement of services. Indeed, a service is implemented in an execution space with, among other things, a public port for communication. Dynamically created ports work together to refine instances of services.
- **Adaptation and Flexibility:** Services can be defined at any time, verified and integrated into the execution spaces.
- **Distribution:** Services are distributed in different locations (user space or service provider).
- **Data driven:** The services parameters guide for the choice of the services. Depending on these parameters, the user or execution engine does the choice of the service to instantiate.

Future work could be associated with the concerns of the semantic web with respect to the problems of automatic search for missing services and the automatic generation of composite services from user requests. To the language thus described could be added constraints that could guide the automatic composition of services and the specification of the non-functional requirements that would favor the construction of composite services based on user requests.

Author Contributions: Conceptualization, W.K.K. and G.K.; Methodology, W.K.K. and G.K.; software, W.K.K.; formal analysis, W.K.K. and G.K.; writing–original draft preparation, W.K.; writing–review and editing, G.K.; supervision, C.T.; funding acquisition, G.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: This research was supported by FUCHSIA project-team of LIRIMA funded by Inria and the CETIC's Programme of the World Bank.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

ATL	Atlas Transformation Language
BPEL	Business Process Execution Language
BPML	Business Process Modeling Language
GAG	Guarded Attribute Grammars
LHS	Left-Hand Side
REST	REpresentational State Transfer
RHS	Right-Hand Side
SELF-SERV	compoSing wEb accessibLe inFormation & buSiness sERVices
SOAP	Simple Object Access Protocol
SPIN	Simple Promela INterpreter
BPEL4WS	Business Process Execution Language for Web Services
WS-CDL	Web Services Choreography Description Language
WSDL	Web Services Description Language
WSCI	Web Service Choreography Interface
PROMELA	PROcess Meta LAnguage

References

1. Rao, J.; Su, X. A survey of automated web service composition methods. In Proceedings of the International Workshop on Semantic Web Services and Web Process Composition, San Diego, CA, USA, 6 July 2004; pp. 43–54.

2. Sheng, Q.Z.; Qiao, X.; Vasilakos, A.V.; Szabo, C.; Bourne, S.; Xu, X. Web services composition: A decade's overview. *Inf. Sci.* **2014**, *280*, 218–238.
3. Barakat, L.; Miles, S.; Luck, M. Adaptive composition in dynamic service environments. *Future Gener. Comput. Syst.* **2018**, *80*, 215–228.
4. Nagl, C., Rosenberg, F., and Dustdar, S. VIDRE–A Distributed Service-Oriented Business Rule Engine based on RuleML. In Proceedings of the 2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), Hong Kong, China, 16–20 October 2006; pp. 35–44.
5. Kamada, A.; Mendes, M. Business rules in a service development and execution environment. In Proceedings of the 2007 International Symposium on Communications and Information Technologies, Sydney, Australia, 17–19 October 2007; pp. 1366–1371.
6. Papazoglou, M.P.; Benbernou, S.; Andrikopoulos, V. On the evolution of services. *IEEE Trans. Softw. Eng.* **2012**, *3*, 609–628.
7. Weigand, H.; van den Heuvel, W.J.; Hiel, M. Rule-based service composition and service-oriented business rule management. In Proceedings of the International Workshop on Regulations Modelling and Deployment (ReMoD'08), Montpellier, France, 17 June 2008.
8. Badouel, E.; Hélouët, L.; Kouamou, G.E.; Morvan, C.; Fondze J.N. Active workspaces: distributed collaborative systems based on guarded attribute grammars. *ACM SIGAPP Appl. Comput. Rev.* **2015**, *15*, 6–34.
9. Sangiorgi, D.; Walker, D. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press: Cambridge, UK, 2003.
10. Jordan, D.; Evdemon, J.; Alves, A.; Arkin, A.; Askary, S.; Bloch, B.; Curbera,F.; Ford, M.; Goland, Y.; Guízar, A.; et al. Web services business process execution language version 2.0. *OASIS stand.* **2007**, *11*, 5.
11. Dijkman, R.; Hofstetter, J.; Koehler, J. Business Process Model and Notation. Springer: Berlin, Germany, 2011, pp. 10–12.
12. Kavantzas, N.; Burdett, D.; Ritzinger, G.; Lafon, Y. *Web services choreography description language version 1.0, w3c candidate recommendation*. Technical Report; W3C: Cambridge, MA, USA, November 2005.
13. Assaf, A.; Intalio, A.A.; Intalio, S.A.; Fordin, S.; Sap, W.J.; Kawaguchi, K.; Orchard, D. Web service choreography interface 1.0. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.879&rep=rep1&type=pdf (accessed on 25 January 2020).
14. Sabraoui, A.; Ettalbi, A.; El Koutbi, M.; En-Nouaary, A. Towards an UML profile for web service composition based on behavioral descriptions. *J. Softw. Eng. Appl.* **2012**, *5*, 711.
15. Newcomer, E. Understanding Web Services: XML, Wsdl, Soap, and UDDI. AddisonWesley Professional: Boston, MA, USA, 2002.
16. Papazoglou, M.P.; Van Den Heuvel, W.J. Service oriented architectures: approaches, technologies and research issues. *VLDB j.* **2007**, *16*, 389–415.
17. Yang, H.; Zhao, X.; Qiu, Z.; Pu, G.; Wang, S. A formal model forweb service choreography description language (WS-CDL). In Proceedings of the 2006 IEEE International Conference on Web Services (ICWS'06). Chicago, IL, USA, 18–22 September 2006; pp. 893–894.
18. Martin, D.; Paolucci, M.; McIlraith, S.; Burstein, M.; McDermott, D.; McGuinness, D.; Parsia, B.; Payne, T.; Sabou, M.; Solanki, M.; et al. Bringing semantics to web services: The OWL-S approach. In Proceedings of the International Workshop on Semantic Web Services and Web Process Composition, San Diego, CA, USA, 6 July 2004; pp. 26–42.
19. Berners-Lee, T.; Connolly, D.; Kagal, L.; Scharf, Y.; Hendler, J. N3logic: A logical framework for the world wide web. *Theory Pract. L. Program.* **2008**, *8*, 249–269.
20. Rosenberg, F.; Dustdar, S. Business rules integration in BPEL-a service-oriented approach. In Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05), Munich, Germany, 19–22 July 2005; pp. 476–479.
21. Yao, Y.; Chen, H. A rule-based web service composition approach. In Proceedings of the 2010 Sixth International Conference on Autonomic and Autonomous Systems, Cancun, Mexico, 7–13 March 2010; pp. 150–155.
22. Zhu, Y.; Huang, Z.; Zhou, H. Modeling and verification of web services composition based on model transformation. *Softw. Pract. Exp.* **2017**, *47*, 709–730.

23. Abouzaid, F.; Mullins, J. Model-checking web services orchestrations using bp-calculus. *Electron. Notes Theor. Comput. Sci.* **2009**, *255*, 3–21.
24. Bianculli, D.; Ghezzi, C.; Spoletini, P. A model checking approach to verify BPEL4WS workflows. In Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA '07), Newport Beach, CA, USA, 19–20 June 2007; pp. 13–20.
25. Paschke, A.; Kozlenkov, A. A Rule-based Middleware for Business Process Execution. Available online: https://vsis-www.informatik.uni-hamburg.de/events/mas2/paschke.pdf (accessed on 23 January 2020).
26. Orriëns, B.; Yang, J.; Papazoglou, M.P. A framework for business rule driven service composition. In *International Workshop on Technologies for E-Services*. Springer: Berlin/ Heidelberg, Germany, 2003; pp. 14–27.
27. Charfi, A., and Mezini, M. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web* **2007**, *10*, 309–344.
28. Schonenberg, H.; Mans, R.; Russell, N.; Mulyar, N.; van der Aalst, W.M. Towards a Taxonomy of Process Flexibility. In Proceedings of the Forum at the CAiSE'08 conference, Montpellier, France, 18–20 June 2008; pp. 81–84.
29. Knuth, D.E. The genesis of attribute grammars. In *Attrib. Gramm. Their Appl.* Springer: Berlin/Heidelberg, Germany, 1990; 1–12.
30. Kungne, W.K.; Kouamou, G.E.; Tangha, C. Introducing an Artifact-driven language for Service Composition. In Proceedings of the ArabWIC 6th Annual International Conference Research Track, Rabat, Morocco, 6–8 March 2019; pp. 1–6.
31. Holzmann Gerard, J. *SPIN Model Checker: The Primer and Reference Manual.* Addison Wesley: Boston, MA, USA, 2003.
32. Wu, P. Interpreting π-calculus with Spin/Promela. *Comput. Sci.* **2003**, *8*, 9.
33. Song, H.; Compton, K.J. Verifying π-calculus processes by Promela translation. Available online: https://www.researchgate.net/profile/Kevin_Compton/publication/244354070_Verifying_-calculus_Processes_by_Promela_Translation/links/00b4953232bc753d31000000.pdf (accessed on 23 January 2020).
34. Eysholdt, M.; Behrens, H. Xtext: implement your language faster than the quick and dirty way. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. Reno/Tahoe, NE, USA, 17–21 October 2010; pp. 307–309.
35. Bettini, L. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing: Birmingham, UK, 2016.
36. Jouault, F.; Allilaire, F.; Bézivin, J.; Kurtev, I. ATL: A model transformation tool. *Sci. Comput. Program.* **2008**, *72*, 31–39.
37. Gentile, U. A Model-driven Approach for the Automatic Generation of System-Level Test Cases. Ph.D. Thesis, University of Naples Federico II, Naples, Italy, 2016.
38. White, S.A. *BPMN modeling and reference guide: understanding and using BPMN*. Future Strategies Inc.: Toronto, ON, Canada, 2008.
39. Pautasso, C.; Alonso, G. JOpera: a toolkit for efficient visual composition of web services. *Int. J. Electron. Commer.* **2005**, *9*, 107–141.
40. Montesi, F.; Guidi, C.; Lucchi, R.; Zavattaro, G. Jolie: a java orchestration language interpreter engine. *Electron. Notes Theor. Comput. Sci.* **2007**, *181*, 19–33.
41. Thönes, J. Microservices. *IEEE softw.* **2015**, *32*, 116–116.
42. Shadija, D.; Rezai, M.; Hill, R. Towards an understanding of microservices. In Proceedings of the 2017 23rd International Conference on Automation and Computing (ICAC). Huddersfield, UK, 7–8 September 2017; pp. 1–6.
43. Decker, G.; Kopp, O.; Leymann, F.; Weske, M. BPEL4Chor: Extending BPEL for modeling choreographies. In Proceedings of the IEEE International Conference on Web Services (ICWS 2007), Salt Lake City, UT, USA, 9–13 July 2007; pp. 296–303.
44. Kitchin, D.; Quark, A.; Cook, W.; Misra, J. The Orc programming language. In *Formal techniques for Distributed Systems*, Springer: Berlin/Heidelberg, Germany, 2009, pp. 1–25.
45. Cugola, G.; Ghezzi, C.; Pinto, L.S.; Tamburrelli, G. Selfmotion: A declarative approach for adaptive service-oriented mobile applications. *J. Syst. Softw.* **2014**, *92*, 32–44.
46. Montali, M.; Pesic, M.; van der Aalst, W.M.; Chesani, F.; Mello, P.; Storari, S. Declarative specification and verification of service choreographiess. *ACM Trans. Web (TWEB).* **2010**, *4*, 3.

47. Greenwood, D.; Rimassa, G. Autonomic goal-oriented business process management. In Proceedings of the Third International Conference on Autonomic and Autonomous Systems (ICAS'07), Athens, Greece, 19–25 June 2007; pp. 43–43.
48. Benatallah, B.; Sheng, Q.Z.; Dumas, M. The self-serv environment for web services composition. *IEEE internet comput.* **2003**, *7*, 40–48.
49. Abiteboul, S.; Benjelloun, O.; and Milo, T. The Active XML project: an overview. *VLDB J.* **2008**, *17*, 1019–1040.
50. Garriga, M.; Mateos, C.; Flores, A.; Cechich, A.; Zunino, A. RESTful service composition at a glance: A survey. *J. Netw. Comput. Appl.* **2016**, *60*, 32–53.

.

A.4 Journal 2

Kouamou, G. E., and Kungne, W. K. (2017). *A Structural and Generative Approach to Multilayered Software Architectures.* Journal of Software Engineering and Applications, 10(8), 677-692.

Journal of Software Engineering and Applications, 2017, 10, 677-692
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

A Structural and Generative Approach to Multilayered Software Architectures

Georges Edouard Kouamou[1], Willy Kengne Kungne[2]

[1]Department of Computer Engineering, National Advanced School of Engineering, Yaounde, Cameroon
[2]Department of Computer Science, Faculty of Science, University of Yaounde I, Yaounde, Cameroon
Email: georges.kouamou@polytechnique.cm, willy.kengne@uy1.uninet.cm

How to cite this paper: Kouamou, G.E. and Kungne, W.K. (2017) A Structural and Generative Approach to Multilayered Software Architectures. *Journal of Software Engineering and Applications*, **10**, 677-692.
https://doi.org/10.4236/jsea.2017.108037

Received: May 2, 2017
Accepted: July 8, 2017
Published: July 11, 2017

Abstract

The layered software architecture is the model commonly adopted for the development of information systems since it favors the modularity and the scalability of the systems. On the other hand, the emergence of model engineering aims to raise the level of abstraction to allow developers to reason on models, and less in code. The research question is to combine the two approaches to facilitate the work of developers. The proposal resulting from this study is based on a set of concepts defined using the UML profiles. These concepts include services, business components, and data persistence. Then the Kruchten model is adopted to represent the development cycle according to several views, each view being represented by UML diagrams derived from the previously defined profiles. Finally, rules are available for checking inter-view consistency, from refinement to code generation. The result is a step towards the definition of a domain specific ADL and a development process as much as it includes the expected characteristics of such a language, namely: the fundamental concepts, the support tools and the multiview development.

Keywords

ADL, Architectural Style, Model Driven Engineering, UML, Service, Software Development Process

1. Introduction

The software architectures describe in a symbolic and schematic manner the various constituent elements of computer systems, their interrelations and their interactions. The architectural styles specify the nature of the components, the connectors, the topological distribution of these components, indicating their relations and a set of semantic constraints. The development of software architectures in the software industry has led to the development of a generation of

languages so-called Architectural Description Language (ADL).

An ADL is a language that provides functionality for modeling the conceptual architecture of a software system. An ADL provides a concrete syntax and a conceptual framework for characterizing architectures. It should explain the basic concepts of the software architectures that are: components, connectors and configurations. To be valued, an ADL must provide a set of support tools for the development of architectures and their evolution [1]. In this context, multiple architectural views must be offered to developers through which they appreciate the consistency of the system being built. However, most of the known languages in this domain have not been imposed on software developers and builders for two reasons: 1) they require advanced knowledge in formal theories, 2) they are limited only to the description of the architecture and its verification, without worrying about the implementation of the functionalities of the application [2].

UML provides simple graphical notations with understandable semantics for specifying, viewing, modifying, and building the necessary documents for software development. Recent developments in this modeling language have explicitly introduced the fundamental concepts of software architectures. It thus positions itself as a candidate for the massification of software architectures [3]. First because it is accepted by the software manufacturers and the academic milieu. Then it is commonly used by most developers. In this situation, it is important to be able to represent the whole development cycle within an ADL. Although UML through the diagrams it offers, presents the different abstractions of the system to be designed and it complies with the various architectural views as defined by the Kruchten model [4], it does not define a process of refinement to lead to the implementation. Apart from the transformation of the class diagram into code that is automatic, the refinement process for new architectural concepts in UML is usually done manually. From these facts, UML considered alone does not offer all the features that are expected of an ADL.

In this paper, we propose a framework for the construction of software architectures based on UML, which we combine with the Kruchten model to satisfy certain missing characteristics, in particular the definition of architectural views. Once this choice is adopted, we focus on the development of the refinement mechanisms to ensure the coherence between views and to transform the conceptual elements into structural components of the application. We will take into account the use case view that materializes the services offered by the system, and the logical view that includes the structural elements to derive the implementation view in a layered architectural style. For this purpose, we define a UML profile and transformation rules described with Atlas Transformation Language (ATL) [5].

The remainder of this article is organized as follows. Section 2 presents a state of the art on software architectures and the study of some ADLs in order to position our preoccupation with the existing ones. In Section 3 we propose the models underlying our approach to design a domain specific ADL. This approach is based on the use of UML metamodels through the Model Driven Architecture

(MDA) approach advocated by the OMG. Section 4 details the experimental framework in which a tool is implemented to support the proposed approach. Next the section 5 presents the validation of this tool which is carried out on the construction of the back-end of an application for managing registration in a university. Section 6 presents the conclusion and some thinking for future studies.

2. Background

2.1. ADL

An ADL provides a concrete syntax and a conceptual framework for characterizing architectures. Each ADL must rely on a set of fundamental concepts namely components, connectors and configurations. In addition to these concepts, an ADL must have some minimal features which are mainly support tools: architecture editor, refinement, code generator and architecture evolution management [1]. In the literature, we distinguish three approaches in the definition of ADL.

The first approach consists of native languages specifically designed to specify software architectures: Wright, Darwin, Rapide. The fundamental concepts mentioned above are the elementary entities they offer. If they offer the main characteristics among those expected of an ADL in this case specification editors, static and dynamic analysis tools, sometimes code generators in a programming language, the main disadvantage of these Languages remain their heterogeneous terminology and their restriction to specific communities or application domains [6].

The second approach consists to extend a common programming language by incorporating the basic concepts of ADLs. ArchJAVA is an illustration [7]. If these ADLs allow the user to remain in his familiar language by exploiting his usual environment, this approach has the disadvantage of covering only the implementation view.

The third type consists of the languages that can be used as a common interchange format for architecture design tools. Acme is an illustration [8]. They are used to exchange one architecture format to another. Therefore they can be used as a pivot language between two architectures defined in two different ADLs. Some may provide a basis for developing new ADL. The principle is based on a simple structure that takes up the basic structure of all the ADLs and the properties allowing to define auxiliary information, which supplement the description of the architecture. Their principal shortcoming is the absence of mechanisms for analysis and code generation [6].

2.2. UML as an Approach to the Dissemination of ADL

UML is a modeling language based on graphical symbols to represent a system. It provides users with different diagrams that put together, form a complete modeling of the system. UML does not impose any design methodology, that is, UML does not impose a particular way for the use of the diagrams it offers. Each

diagram must respect the syntax defined in its specification. Starting with version 2.0 UML improves its component diagram which explicitly takes into account the basic concepts of an ADL.

UML has a well-defined syntax. It is widely adopted by developers and its evolution is supported by several manufacturers. Each diagram can be used in one of the views of the Kruchten architectural model [4]. This symbiosis between UML and the 4 + 1 views model promotes an architectural design method to which it will be necessary to associate support tools to build a true ADL [9]. Taking this into account, the concept of architectural style in the method should be made explicit and the coherence between the different diagrams should be ensured.

3. Modeling Approach

The traditional IT organizational structures of the most companies closely matches the layered architecture style. Nowadays this style is adopted for most Enterprise Information Systems consisting of four standard layers: presentation, business, persistence, and database [10]. This purpose of the approach is to develop a specific language dedicated to the description of layered software architectures.

With regard to the characteristics as defined by [11], it is essential to have within the same ADL several views presenting different aspects of the architecture of the system. Krutchten's 4 + 1 views model allows to appreciate all the views of the system throughout the development. Each view can be represented by UML diagrams. In order to manipulate these models, OMG standardizes a model driven approach so-called Model Driven Architecture (MDA) which introduces UML extension mechanisms.

3.1. Kruchten Model

During the software development process, each step presents a different abstraction from the system to be designed. These different abstractions in **Figure 1** derive the notion of view in the process model.

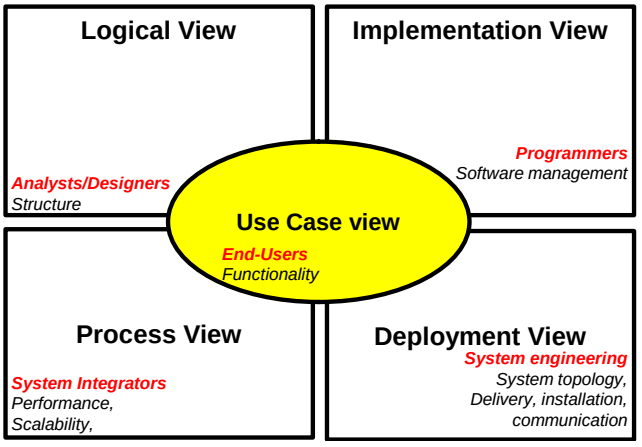

Figure 1. The 4 + 1 views model.

The logical view includes the structural elements (classes, components) of the design. It is the view that is at the heart of reuse. The process view captures the dynamics and timing of design aspects. The physical or deployment view describes how the software is mapped to the hardware and reflects its distributed configuration. The implementation view describes the organization of the system source code. Finally, the central view shows the use cases that represent the functional requirements of the application.

Each view of this model is represented by UML diagrams. The central view (use case view) is illustrated by the use case diagram, the logical view can be represented by the class and component diagrams, the process view can be represented by the activity diagram, State Chart diagram, Interaction diagram. The implementation view can be represented by the class diagram and also the component diagram since a UML component is a database, a source file, a Dynamic Link Library (DLL) and the deployment view can be represented by the deployment diagram.

Recent versions of UML aim to increase the level of abstraction by advocating model engineering [12]. This is equivalent to making the UML models perennial and allowing them to be free from the execution platforms. Among the new characteristics introduced are [13]:

- The elaboration of a Document Type Definition (DTD) for UML2.0 according to the XMI standard. It shows the importance of XMI which is the standard par excellence capable of exchanging models.
- Component-based development: UML2.0 supports the component paradigm, it defines profiles to support Corba Component Model (CCM) and Enterprise Java Bean (EJB) component model and it enables profiles for other component platforms.

These characteristics favor the definition of reusable architectures and their storage in XMI format. The MDA approach intervenes in the definition of models and also in their transformations.

3.2. The MDA Approach

MDA is a set of modeling and model transformation techniques standardized by the OMG [14]. This approach advocates the use of models in the different phases of the development cycle of an application. Specially, it aims to develop model requirements, model analysis and design and code models. The transformations make it possible to link these different models.

3.2.1. Requirements Model

The first thing to do when building an application is to specify the client's requirements to define what services are offered by the future application. Requirements are specified in a requirements model called Computation Independent Model (CIM). It allows to clearly express the links of traceability with the models that will be built in the other phases of the development cycle of the application. With UML, a requirements model can be summarized as a use case diagram.

3.2.2. PIM Analysis and Design Models

It is during analysis and design that the software architecture of the application is realized. In the MDA approach, this phase also uses a Platform Independent Model (PIM). PIMs assure the transition from the requirements model (functionality) to the implementation model so-called Platform Specific Model (PSM).

3.2.3. PSM Code Models

Code generation begins after the PIM is obtained. This phase is tricky because the code patterns and the source code of the application can be confused.

3.2.4. Transforming Models

The transformations of models allow the mapping from one model to another. Because the transformation is at the heart of MDA, OMG has standardized Query/View/Transformation (QVT) whose ATL is an implementation in the Eclipse environment [15]. This language allows to describe rules to transform a model into another one as well as query queries making it possible to convert a model into text.

3.3. The Steps of the Proposed Approach

The approach includes three steps that are similar to analysis, design and implementation. The first step concerns the requirements analysis that lists the expected services. These services are grouped in the boundary layer between the clients and the business application. The introduction of a service layer aims to satisfy the needs of the modern information systems characterized by the adoption of Service-Oriented Architecture (SOA) principles [16]. The second step deals with the identification of business components that implement the services identified during the requirements analysis. Because these components manipulate persistent entities, they must be identified and gathered to create the structure of the database in the third step.

The approach consists of three views. Each view represents a different aspect of the system and contains a modeling formalism to represent its elements. These three views are:

- The central view (Service view): this is the view allowing to specify the user requirements in the form of use cases.
- The logical view: this is the view used to specify the class models as well as the business components.
- The implementation view: it is the view to check the consistency between the two other modules, to transform and refine their models to have a skeleton of code for which we preferred, without harming the generality, the technology Java for experimentation.

3.3.1. The Service View

This view describes the use cases that are the services provided by the application. Its modeling formalism is presented by the UML metamodel of **Figure 2**.

The meta-class *Service* allows to specify the functionalities of the application

as a services. It is composed of a set of exceptions represented by the *Exception* meta-class. The services are performed by actors represented by the meta-class *Actor*. In order to regroup the services by packages we defined a meta-class *ServiceSet* which gathers a set of services. This meta-model characterizes a CIM for an application that translates the functional requirements into the framework of services.

3.3.2. The Logical View

This view describes the database model in the form of a class diagram as well as the business components. A business component specifies the provided services and the required services as methods signature *i.e.* with their input and output parameters. Figure 3 describes the model formalism for this module. This figure contains the following classes:

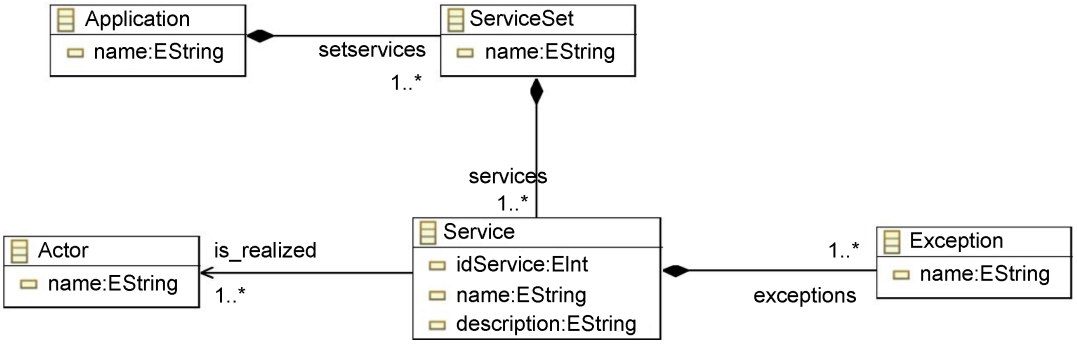

Figure 2. Service meta model.

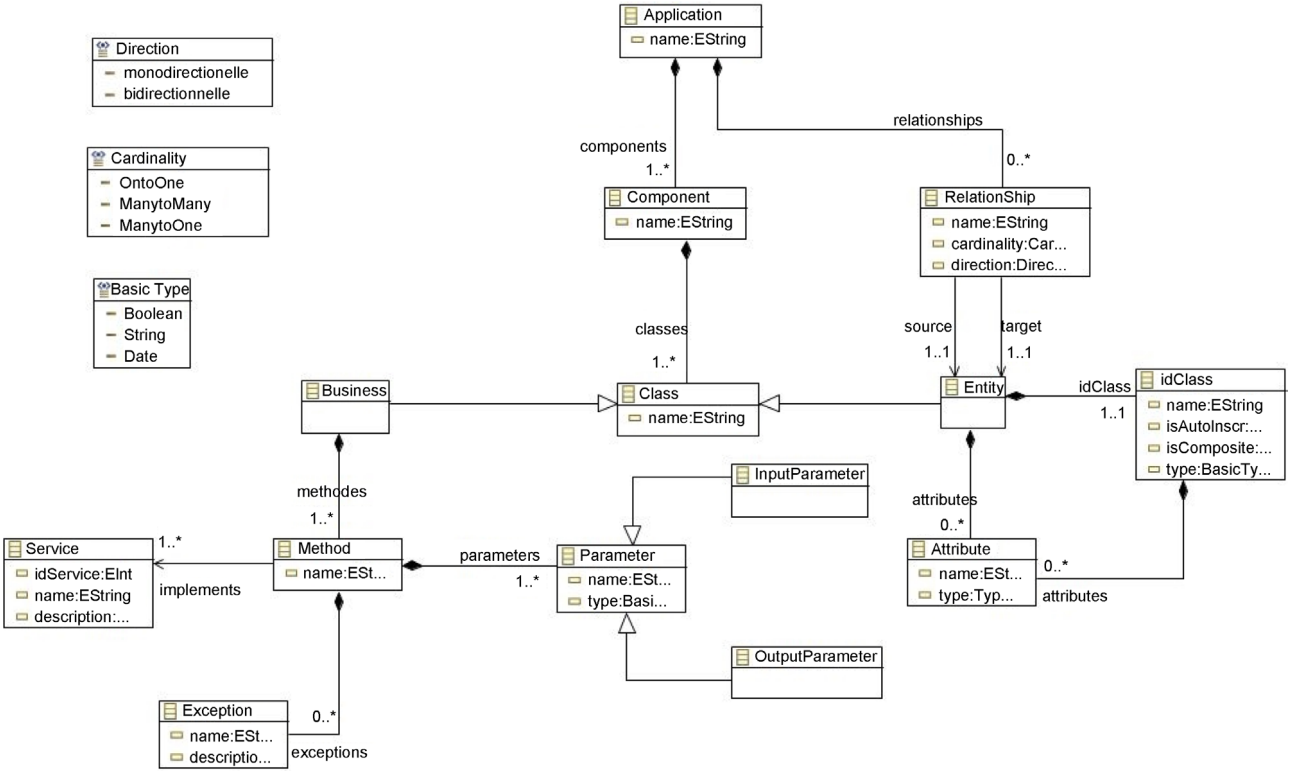

Figure 3. Meta model of the logical view.

- Entity: coupled with Relationship describes persistent entities that require a Data Base Management System (DBMS) for their storage. *Attribute* and **Id-Class** respectively represent the concepts that describe respectively the attributes and the keys of the different tables.
- Business: describes the interfaces of the business components. The services of these components are represented by Methods that have parameters (Parameter) that can be input parameters (Input Parameter) or output parameters (Output Parameter). The services declared in the *cohesion ADL Service* module are implemented by the methods of this module.
- *Direction*, *Cardinality* and *Type Primitive* are enumerated types. They define new types.

The models conforming to this formalism are considered as PIM in the sense that they are independent to the implementation platform.

3.3.3. The Implementation View

This third view makes it possible to:

- Check the consistency between the other two views.
- Transform the obtained PIM model into PSM close to the EJB3 components.
- Refine the previous PSM to obtain a code skeleton in Java including a layer of web services to manage client heterogeneity, an EJB component layer and a data access layer.

Consistency check

It consists in ensuring that each identified service is implemented at the level of the business. Let's consider C the set of business components and S the set of registered services. Let us define on C the "use" relation. Two business components c_1 and c_2 are related if one uses at least one service provided by the other. Groups are thus formed which are similar to equivalence classes whose constituent elements are subsets of components which interact with one another. Let us note $[C]$ the new set constituted by these groups.

Let's define the application: $f:[C]\to S; f[c]=\{s_i \in S, i=1\cdots k\}$ which identifies the set of services implemented by a group of components.

1) If there is an element of C that does not belong to any equivalence class and does not implement any service of S then this business component is irrelevant. A warning message is generated containing these irrelevant components so that the designer can take corrective action.

2) If an element of $[C]$ has no image in S, then this group is irrelevant. In this case, the business components that constitute it appear in the warnings file.

3) If f is surjective then any service $s \in S$ has an implementation in C. Otherwise, the service with no antecedent is reported in the warnings log.

This rules ensures consistency between the service view and the logical view. The coherence intra-view (logical) is ensured by the associations defined between the entities of the metamodel.

Transformation and refinement of models

After the step of checking the consistency, if the set difference is empty, we transform the obtained model by federating the two models from the first two

views (service and logic) into a model of code conforming to the formalism represented in Figure 4. As the JEE platform is chosen for the validation of the approach, it only remains to refine the models in order to deliver the Java code skeleton consisting of EJB3 components. The ATL script used for this purpose has 10 rules and 17 helper.

1) Refinement of the service layer. The transformation program contains rules that map a *SetService* to an EJB3 Component for the Service Layer and the methods for the interface and implementation of the EBJ3 component are obtained by transforming the Business Class methods (Figure 3) that perform the Services that make up the *SetService*. The model is decorated with JAX-WS annotations (Java Annotation XML-Web Service). The Web Services will have to allow interoperability with presentation layer whatever its nature (Web, Desktop, Android, etc…).

2) Refinement of the business layer. The business layer contains EJB3 components constructed from the formalism presented in the logical view. In the latter, the Business class is transformed into the EJB3 component and its methods are

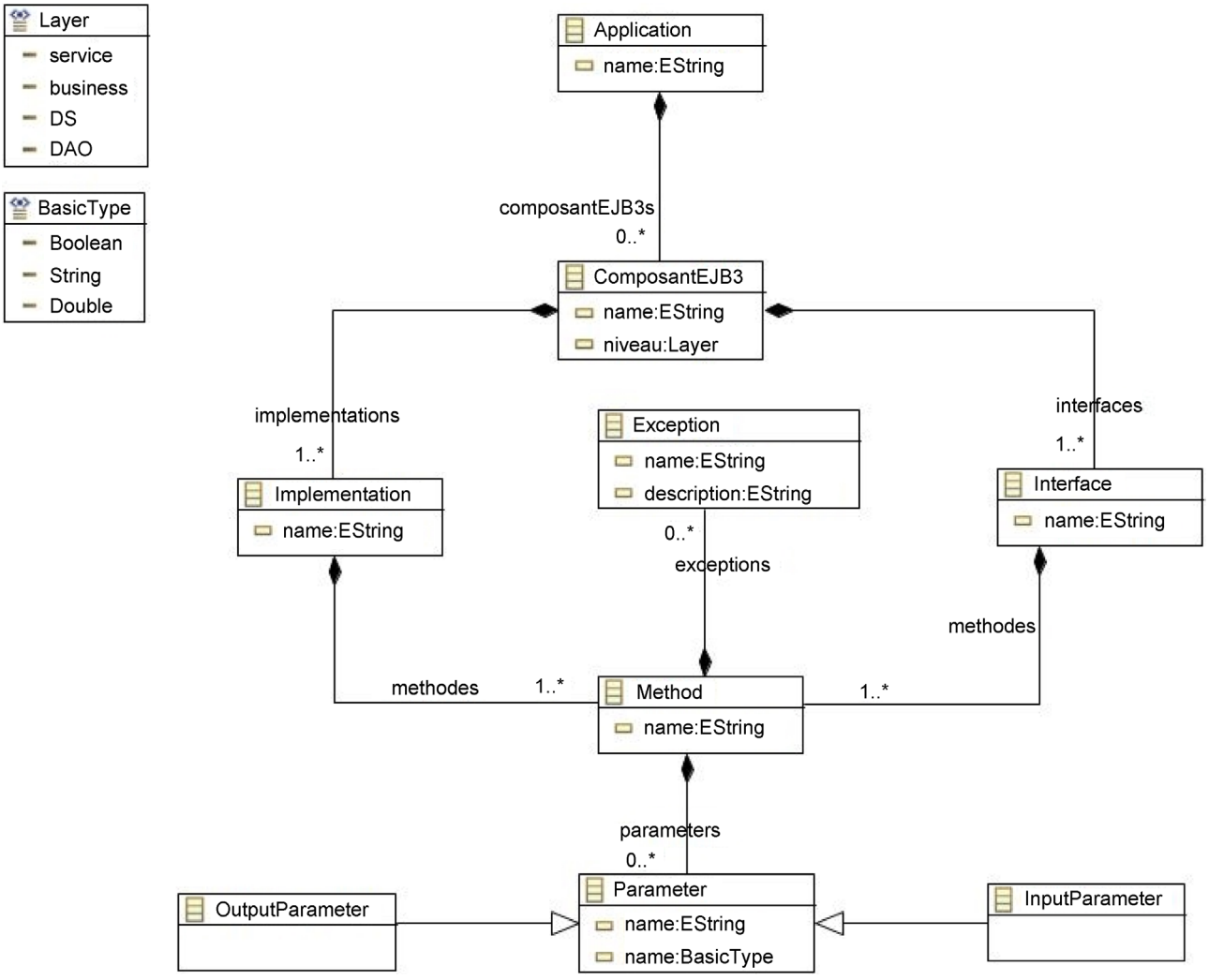

Figure 4. Metamodel towards EJB3 components.

transformed into methods for the interface and the implementation of the EJB3 component.

3) Refinement of the Data Access Object (DAO) layer. The DAO layer is built from the Entity, Attribute, IdClass, and RelationShip classes of the logical view. It of a sublayer containing of the entities that will be mapped to the tables in the database and another sub-layer (Data Service) that provides creation, modification, deletion, and persistent feature search services. The DS layer encapsulates the services offered by the DAO layer and presents them to the business layer.

Since the EJB3 components for these two layers provide basic services for inserting, deleting, creating, and searching data entities, they will be closely refined from the Entity.

4. Experimentation Framework

To facilitate the creation of software architectures according to the approach and the reference style described in this paper, the support tools are built as an Eclipse plug-in using the following software: Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF) and ATL.

4.1. Presentation of the Tools Used

EMF is a modeling framework that includes code generation from a data model. This is a Java implementation of a subset of the OMG MOF standard. To avoid ambiguities with MOF, the EMF models conform to the eCORE meta-model. We use EMF to construct the formalism of our models. Each view corresponds to a formalism represented by an .ecore extension file. However EMF does not offer graphic tools for modeling that is why GMF is also used. GMF is a framework allowing to create, from a data model, a graphics editor based on the Eclipse platform. This tool is composed of EMF and GEF. GEF is composed of two parts:

- Graphical Definition Model: represented by the extension file .gmfgraph allows to specify the graphic elements of the model.
- Tooling Definition Model: represented by the .gmftool extension file is used to specify the elements of the palette.

In order to link EMF models to GEF, GMF assembles through the mapping model, a file with the gmfmap extension. For each element of the Graphical Definition Model, it is assigned a node and an action as well as the corresponding class of the data model. After this step, you can generate a new .gmfgen file, gathering all the information in the project.

ATL allows to specify the transformation rules for models from the service and logical views to obtain the implementation view. Once the different views are modeled (logical view and service view), their representation are merged into a unique XMI. The XMI file that results from this merging is transmitted to the refinement module whose code is written in ATL which checks the consistency, transforms and generates the corresponding structured Java code.

4.2. Case Study: A University Registration Application

The on-line registration application allow the students of a university to register from a computer (Laptop, Desktop) or a mobile terminal (phone, PDA, etc.). For this purpose, they must be able to pay their registration fees, complete the registration forms, get the medical form from an authorized doctor, and upload the documents which justifies their status. Then the "Registration Agent" must be able to consult and manage the students' files in order to validate or reject a file.

Some constraints: 1) the payment of fees will be done by a mobile or electronic means of payment, 2) any person must subscribe to the service in order to have a user account on the platform. The components of the different views are summarized in Table 1.

Once the services are identified, they are assigned to the business components that can perform them. Service behavior is entirely delegated to the business components. In anticipation of interactions with client devices that may be of a varied nature, we consider this layer of services as adapters capable of ensuring compatibility between the system and the user interfaces [17].

The cartography of the business components is defined by factoring behaviors at the service level. This technique avoids having a large number of components in proportion to the services. Other components are added by necessity, for example if the system requires an external component. This is the case of the *PaymentManager* component, which requires an electronic payment API to manage the financial transactions. Each component includes boxes that each materialize the service provided. In addition to the signature of the service, there are the exceptions that are taken into account.

The graphical representations of the constituents of the various views, shown in Figures A1-A3, are supplemented by textual properties, given by the designer, which describe the inter-view and intra-view relationships. These properties are data provided to instantiate the components from the metamodel. About the relationships between the entities, the developer gives a name, the cardinalities, and the direction. Concerning the business component, one needs to define the operation signature (its name and the type of each parameter), the exceptions

Table 1. Summary of the views.

Services	Business Components	Entities
Login Logout	Session Manager	User Account Role
Subscribe	User Manager	User Account
Upload Document	Upload Manager	Document
Create File Fill Registration Form Consult File Save File	File Manager	Registration Form
Pay Registration Fees	Payment Manager	Payment

The details and graphical overview are shown in Figures A1-A3.

that can be thrown. At the level of services, the association with the business operation that must implement each of them is required.

5. Evaluation

The tool in its current state allows to generate the back-end code of a given application. Figure 5 shows the corresponding architecture of the generated code. The Service layer, DataService layer and DAO layer are generated integrally therefore do not need to be completed. Only the instructions of the business layer are written manually because, at this stage of thinking, the dynamic view is not taken into account.

Automatic generation as well as the proper organization of the code have an impact on the productivity of the developers and the quality of the final software for many reasons:

- The reduction of the programming delay since the programmer deals only with the business components.
- The lisibility of the application code.
- The improvement of the reuse and the extensibility of the software because of the adoption of a component-based approach.

Among the 4 layers, 3 are fully generated, *i.e.* 75%. These are the DAO, DataService and Services layers. The 4th layer contains the technical requirements and the beans corresponding to the service calls. On this basis, we estimate that the development team will be responsible for writing 20% of the volume of the code needed to implement an application. This code corresponds to the bean body instructions.

6. Conclusions and Further Works

Software architectures have proven importance in the software development process. Among the architectural styles reference, layered architectures are predominant especially for structuring information systems, because they favor the modularity, flexibility and scalability of systems. However many platforms like JEE are based on the layered architecture model, but they leave it up to the developers to organize the code manually from the conceptual model.

This study proposes an approach and a support tool for the design of multi-layered software architectures. The approach combines the 4 + 1 process model

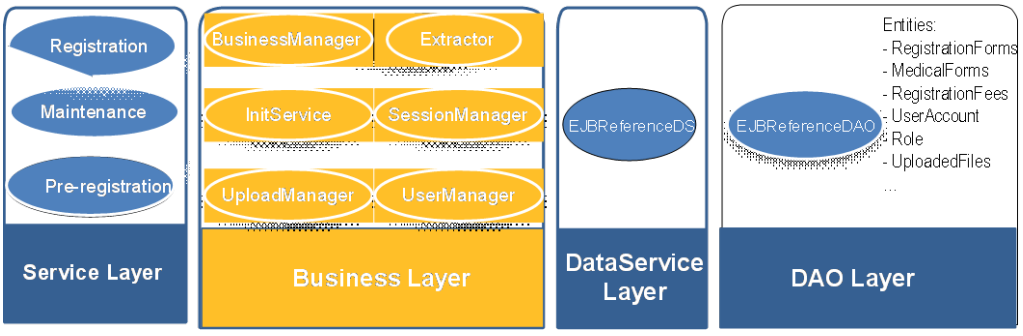

Figure 5. Architecture of the generated application.

from Krutchten and the use of UML diagrams to facilitate its dissemination. The importance of this work resides in the definition of a domain specific language dedicated to the specification of layered architectures, the assistance to the development team which will focus more on the models and will be less interested in the structure of the code.

The resulting tool is built in the form of an Eclipse plug-in. It consists of three modules: the first one allows to specify the requirements of the application to be constructed in the form of use cases, the second serves to describe the models of classes and components of the application and finally the transformation module including ATL rules allows to obtain the implementation view from the files obtained from the first two modules. The validation of the model and the experimentation of the prototype are carried out through the application of online registration in an academic institution.

The tool whose the design is presented in this article does not take into account the dynamic and the deployment view of the Kruchten model in order to complete all the views of a software system. These aspects will be taken into account in future versions in order to have a comprehensive tool that facilitates the work of the development team throughout the development process. The deployment aspect is all the more important as most current systems are distributed. It would be interesting if the resulting environment supports the placement of the constituents on the different nodes of the network.

The basis of our reflection integrates ubiquity and distribution as fundamental aspects of modern systems such as state in [18]. The ubiquity deals with the presentation layer where the work carried out by [19], will be helpful to think on a generic metamodel that will enable us to build the front-end for Information systems, that is independent to GUI libraries and device platforms (desktop, Web, Android…). For this purpose, we are thinking on coupling the MVC model to the layered style where the model is represented by the backend conforming to the layered structure mentioned in this paper, the controller acts as an adapter to ensure the compatibility between the presentation layer and the services layer.

References

[1] Medvidovic, N. and Taylor, R. (2000) A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering,* **126**, 70-93. https://doi.org/10.1109/32.825767

[2] Kouamou, G.E. (2012) Coherence of Views in the Specification of Software Architectures. *ARIMA,* **14**, 205-216.

[3] Medvidovic, N., Rosenblum, D.S., Redmiles, D.F. and. Robbins, J.E (2002) Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology,* **11**, 2-57. https://doi.org/10.1145/504087.504088

[4] Kruchten, P. (1995) Architectural Blueprints—The "4+1" View Model of Software Architecture. *IEEE Software,* **12**, 42-50. https://doi.org/10.1109/52.469759

[5] Randak, A., Martínez, S. and Wimmer, M. (2011) Extending ATL for Native UML Profile Support: An Experience Report. *Proceedings of the* 3*rd International Workshop on Model Transformation with ATL*, Zürich, Switzerland, CEUR-

WS.org, Vol-742, Jul 2011, 49-62.

[6] ACCORD (2002) Etat de l'Art sur les Langages de Description d'Architecture. INRIA. https://pdfs.semanticscholar.org/5fb0/c9409a903b296aa86b4e49dc59e7eaff0ef3.pdf

[7] Aldrich, J., Chambers, C. and Notkin, D. (2002) ArchJava: Connecting Software Architecture to Implementation. *Proceedings of the ISCE*, Orlando, Florida, 19-25 May 2002, 187-197. https://doi.org/10.1145/581339.581365

[8] Garlan, D., Monroe R. and Wile, D. (1997) Acme: An Architecture Description Interchange Language. *Proceedings of CASCON'97*, November 1997, Toronto, Ontario, 169-183.

[9] Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A. and America, P. (2007) A General Model of Software Architecture Design Derived from Five Industrial Approaches. *Systems and Software,* **80**, 106-126. https://doi.org/10.1016/j.jss.2006.05.024

[10] Richards, M. (2015) Software Architecture Patterns. O'Reilly Media, Inc.

[11] Taylor, N., Medvidovic, N. and Dashofy, E.M. (2009) Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons Publishing, Hoboken.

[12] Muchandi, V. (2007) Applying 4 + 1 View Architecture with UML 2. Sparx Systems.

[13] Blanc, X. (2005) MDA in Action. Model Driven Software Engineering, Eyrolles.

[14] Kleppe, A., Warmer, J. and Bast, W. (2003) MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston.

[15] Bézévin, J., Jouault, F. and Valduriez, P. (2004) An Eclipse-Based IDE for the ATL Model Transformation Language. Nantes.

[16] Razavian, M. and Lago, P. (2010) A Frame of Reference for SOA Migration. In: Di Nitto, E. and Yahyapour, R., Eds., *Towards a Service-Based Internet. ServiceWave* 2010. *Lecture Notes in Computer Science*, Vol. 6481, Springer, Berlin, Heidelberg, 150-162. https://doi.org/10.1007/978-3-642-17694-4_13

[17] Lorenz, A. (2013) Architectural Patterns for Applications with External User Interface Elements. *Pervasive and Mobile Computing*, **9**, 269-280.

[18] Nabi, F. and Mullins, R. (2011) Moving from Traditional Software Engineering to Componentware. *Journal of Software Engineering and Applications*, **4**, 283-292. https://doi.org/10.4236/jsea.2011.45031

[19] Belangour, A, Sadik, S. and Abbar, A. (2017) Towards a Platform Independent Graphical User Interface. *American Journal of Software Engineering and Applications*, **6**, 5-12. https://doi.org/10.11648/j.ajsea.20170601.12

Annex

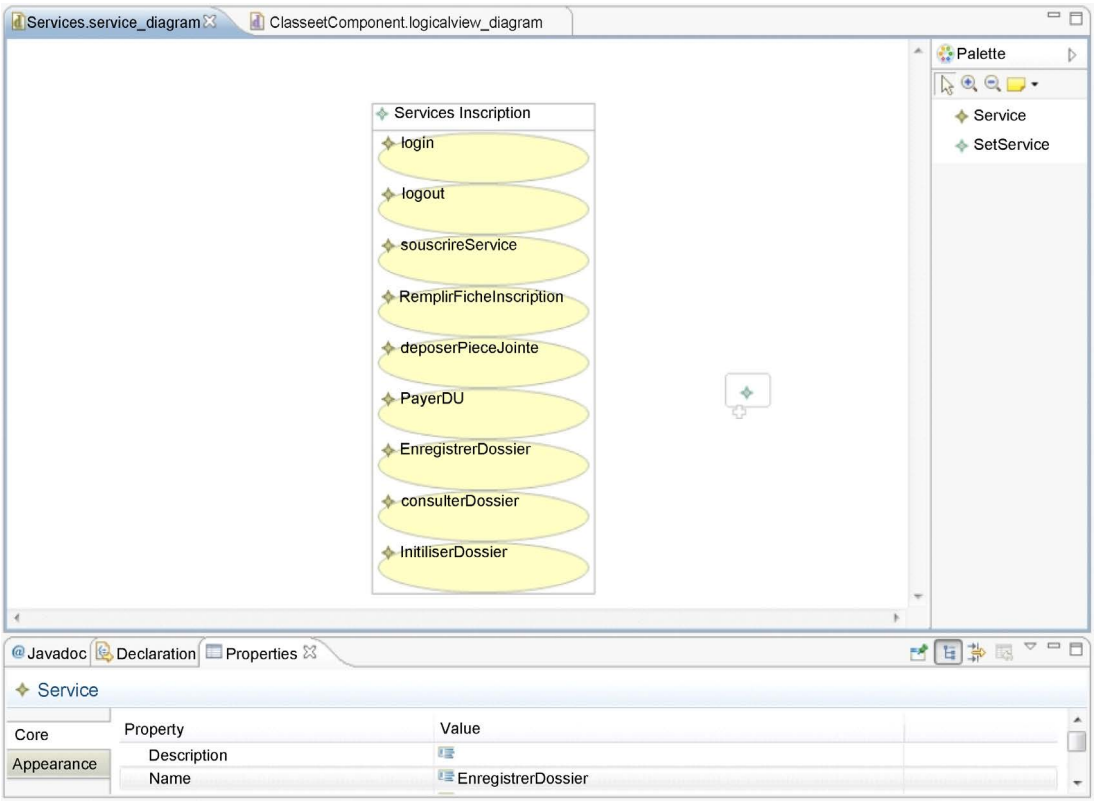

Figure A1. The service requirement model.

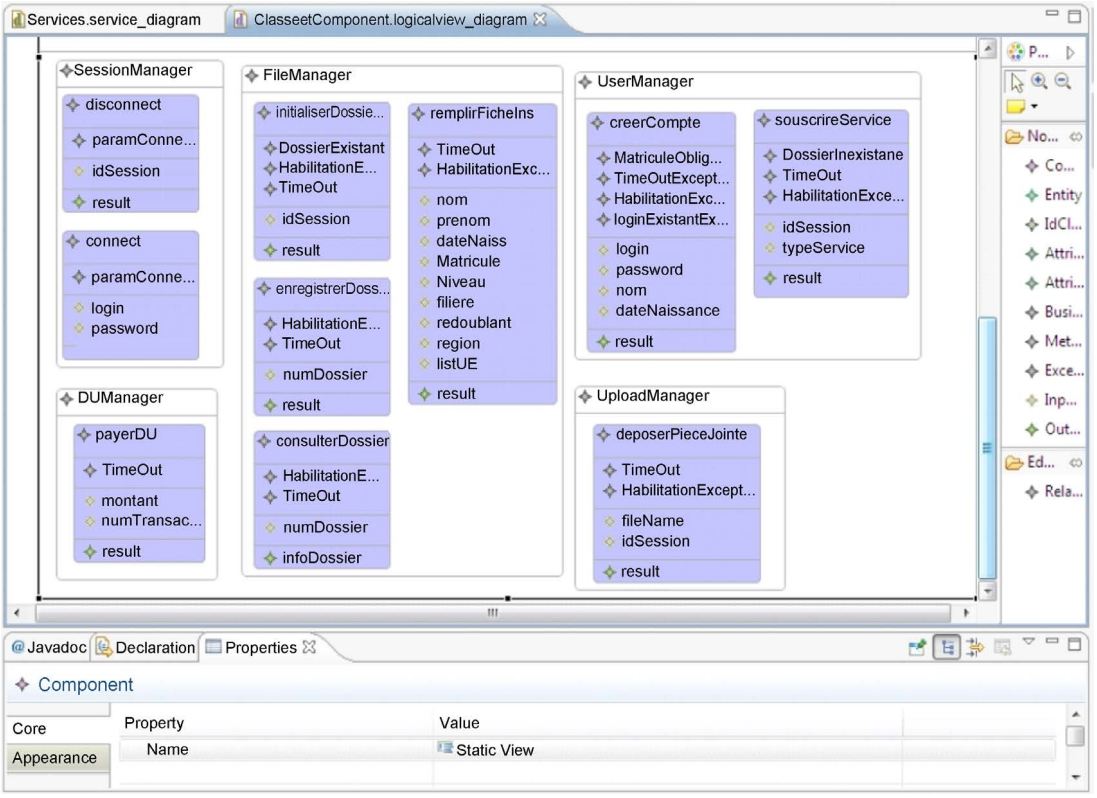

Figure A2. Business components represented by an Instance of the PIM.

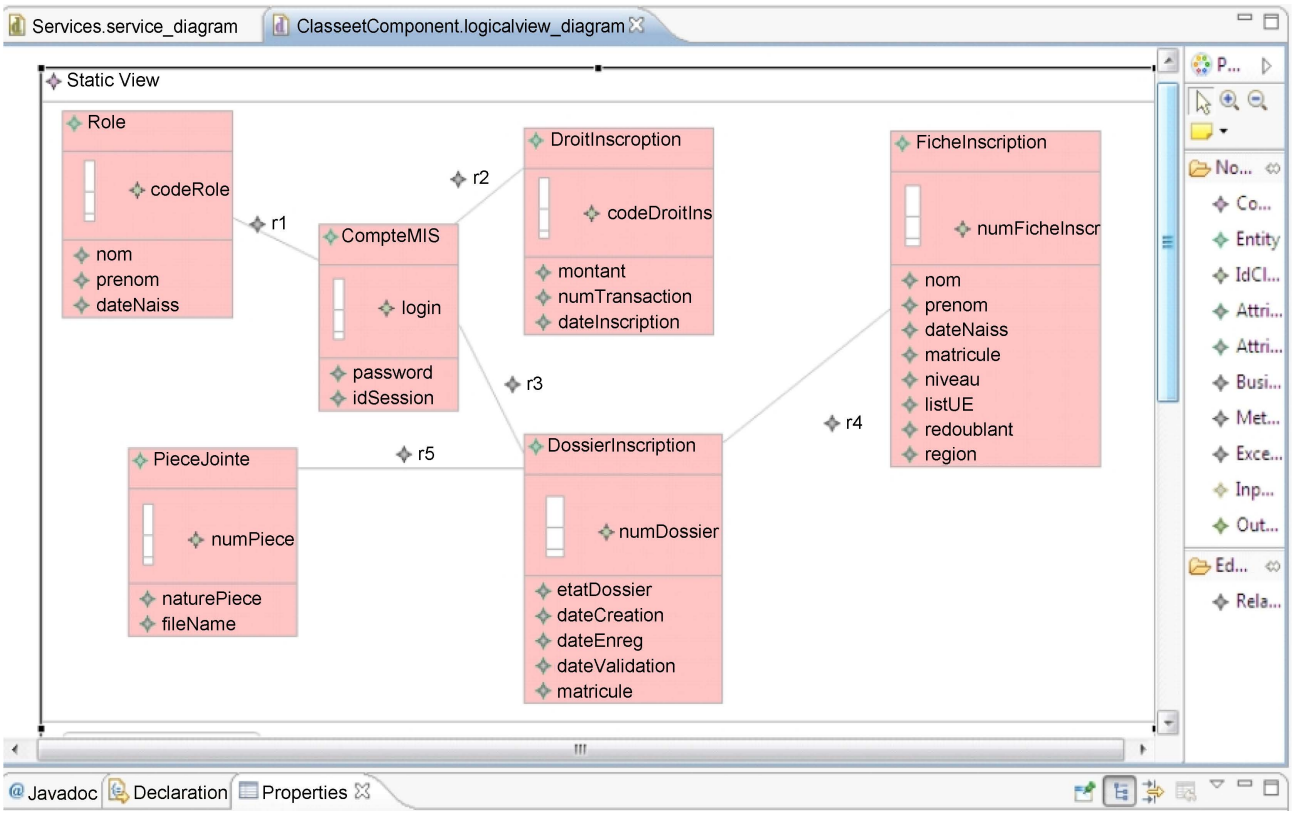

Figure A3. Data models from persistent classes.