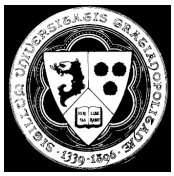

UNIVERSITÉ DE GRENOBLE

UNIVERSITÉ POLYTECHNIQUE DE BUCAREST

THESE

Pour obtenir les grades de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE
et
DOCTEUR DE L'UNIVERSITÉ POLYTECHNIQUE DE BUCAREST

Spécialité : INFORMATIQUE

Préparée dans le cadre d'une cotutelle entre
l'UNIVERSITÉ DE GRENOBLE et l'UNIVERSITÉ POLYTECHNIQUE DE BUCAREST

Arrêtés ministériels : 6 janvier 2005 - 7 août 2006

Préparée et soutenue publiquement par

Damien THIVOLLE

le 29 avril 2011

Langages modernes pour la modélisation et la vérification des systèmes asynchrones

---

Thèse dirigée par Hubert GARAVEL et co-dirigée par Valentin CRISTEA

JURY

M. Dumitru POPESCU	Professeur à l'Université Polytechnique de Bucarest	Président
M. Ioan JURCA	Professeur à l'Université Polytechnique de Timişoara	Rapporteur
M. Charles PECHEUR	Professeur à l'Université Catholique de Louvain	Rapporteur
Mme Mihaela SIGHIREANU	Maître de Conférences à l'Université Paris Diderot	Rapporteur
M. Valentin CRISTEA	Professeur à l'Université Polytechnique de Bucarest	Examinateur
M. Hubert GARAVEL	Directeur de Recherche à l'INRIA	Examinateur

Thèse préparée au sein de l'Université Polytechnique de Bucarest, du Centre de Recherche INRIA de Grenoble Rhône-Alpes et de l'Ecole Doctorale MSTII de l'Université de Grenoble

Résumé :

Cette thèse se situe à l'intersection de deux domaines-clés : l'ingénierie dirigée par les modèles (IDM) et les méthodes formelles, avec différents champs d'application. Elle porte sur la vérification formelle d'applications parallèles modélisées selon l'approche IDM. Dans cette approche, les modèles tiennent un rôle central et permettent de développer une application par transformations successives (automatisées ou non) entre modèles intermédiaires à différents niveaux d'abstraction, jusqu'à la production de code exécutable. Lorsque les modèles ont une sémantique formelle, il est possible d'effectuer une vérification automatisée ou semi-automatisée de l'application. Ces principes sont mis en œuvre dans TOPCASED, un environnement de développement d'applications critiques embarquées basé sur Eclipse, qui permet la vérification formelle par connexion à des boîtes à outils existantes.

Cette thèse met en œuvre l'approche TOPCASED en s'appuyant sur la boîte à outils CADP pour la vérification et sur son plus récent formalisme d'entrée : LOTOS NT. Elle aborde la vérification formelle d'applications IDM à travers deux problèmes concrets :

1) Pour les systèmes GALS (*Globalement Asynchrone, Localement Synchrone*), une méthode de vérification générique par transformation en LOTOS NT est proposée, puis illustrée sur une étude de cas industrielle fournie par Airbus : un protocole pour les communications entre un avion et le sol décrit dans le langage synchrone SAM conçu par Airbus.

2) Pour les services Web décrits à l'aide de la norme BPEL (*Business Process Execution Language*), une méthode de vérification est proposée, qui est basée sur une transformation en LOTOS NT des modèles BPEL, en prenant en compte les sous-langages XML Schema, XPATH et WSDL sur lesquels repose la norme BPEL.

Mots-clés : Ingénierie dirigée par les modèles, méthodes formelles, vérification, LOTOS NT, CADP, systèmes critiques, systèmes GALS, Services Web, BPEL

Rezumat:

Această teză de doctorat se situează la intersecţia a două domenii-cheie: ingineria dirijată de modele (IDM) şi metodele formale, cu diverse câmpuri de aplicare. Subiectul tratat este verificarea formală a aplicaţiilor paralele modelate conform abordării IDM. In această abordare, modelele joacă un rol central şi permit dezvoltarea unei aplicaţii prin transformări succesive (automatizate sau nu) între modele intermediare cu diferite niveluri de abstractizare, până la producerea de cod executabil. Când modelele au o semantică formală, este posibilă efectuarea unei verificări automatizate sau semi-automatizate a aplicaţiei. Aceste principii sunt puse în aplicare în TOPCASED, un mediu de dezvoltare a aplicaţiilor critice îmbarcate bazat pe Eclipse, care permite verificarea formală prin conectarea către toolbox-uri existente.

Această teză pune in aplicare abordarea TOPCASED bazându-se pe toolbox-ul CADP pentru verificare şi pe formalismul lui cel mai recent acceptat ca intrare: LOTOS NT. Verificarea formală a aplicaţiilor IDM este abordată prin prisma a două probleme concrete:

1) Pentru sistemele GALS (*Globally Asynchronous, Locally Synchronous*), o metodă de verificare generică prin transformare către LOTOS NT este propusă, apoi ilustrată cu ajutorul unui studiu de caz industrial furnizat de Airbus: un protocol pentru comunicaţiile între un avion şi sol, descris în limbajul sincron SAM conceput de Airbus.

2) Pentru serviciile Web descrise cu ajutorul standardului BPEL (*Business Process Execution Language*), o metodă de verificare este propusă, bazată pe o transformare a modelelor BPEL în LOTOS NT, ţinând seama de sublimbajele XML Schema, XPATH şi WSDL— pe care se bazează standardul BPEL.

Cuvinte cheie: Ingineria dirijată de modele, metodele formale, verificare, LOTOS NT, CADP, sistemele critice, sistemele GALS, Serviciile Web, BPEL

Abstract:

The work in this thesis is at the intersection of two major research domains: Model-Driven Engineering (MDE) and formal methods, and has various fields of application. This thesis deals with the formal verification of parallel applications modelled by the MDE approach. In this approach, models play a central role and enable to develop an application through successive transformations (automated or not) between intermediate models of differing levels of abstraction, until executable code is produced. When models have a formal semantics, the application can be verified, either automatically or semi-automatically. These principles are used in TOPCASED, an Eclipse-based development environment for critical embedded applications, which enables formal verification by interconnecting existing tools.

This thesis implements the TOPCASED approach by relying on the CADP toolbox for verifying systems, and on its most recent input formalism: LOTOS NT. This thesis tackles the formal verification of MDE applications through two real problems:

1) For GALS (*Globally Asynchronous, Locally Synchronous*), a generic verification method, based on a transformation to LOTOS NT, is proposed and illustrated by an industrial case-study provided by Airbus: a communication protocol between the airplane and the ground described in the synchronous langage SAM designed at Airbus.

2) For Web services specified with the BPEL (*Business Process Execution Language*) norm, a verification method is proposed. It is based on a BPEL to LOTOS NT transformation which takes into account XML Schema, XPATH, and WSDL, the languages on which the BPEL norm is built.

Keywords: Model-Driven Engineering, formal methods, verification, LOTOS NT, CADP, critical systems, GALS systems, Web services, BPEL

Remerciements

Je tiens à remercier:

- Hubert Garavel, *Directeur de Recherche à l'INRIA, qui a été l'instigateur de cette thèse et dont la persévérance sans faille a très certainement permis l'aboutissement des travaux présentés ici ;*
- Valentin Cristea, *Professeur à l'Université Polytechnique de Bucarest, qui m'a accueilli pendant deux ans dans son équipe et dont les conseils m'ont toujours été d'un grand secours ;*
- Charles Pecheur, *Professeur à l'Université Catholique de Louvain, et* Mihaela Sighireanu, *Maître de Conférences à l'Université Paris Diderot, pour l'attention qu'ils ont portée à la relecture de ce document et les commentaires précis et pertinents qu'ils ont émis et auxquels j'ai essayé de répondre de mon mieux dans la version finale de ce document ;*
- Ioan Jurca, *Professeur à l'Université Polytechnique de Timisoara, pour avoir accepté de juger ce travail ;*
- Dumitru Popescu, *Professeur à l'Université Polytechnique de Bucarest, pour l'honneur qu'il m'accorde en présidant le jury de cette thèse et pour l'accueil chaleureux qu'il m'a réservé.*

Je n'oublierai jamais l'amitié que m'a témoignée Radu Mateescu, *qui m'a épaulé dans de nombreuses démarches et m'a éclairé de ses connaissances en d'innombrables occasions.*

Mes remerciements vont ensuite à Frédéric Lang, Gwen Salaün et Wendelin, *pour la disponibilité et la gentillesse avec laquelle ils m'ont toujours reçu.*

Pour les bons moments passés ensemble, je voudrais associer à ces remerciements les membres passés et présents de l'équipe VASY, ainsi que du laboratoire d'informatique de l'Université Polytechnique de Bucarest, et plus particulièrement: Sylvain Robert, Rémi Hérillier *(qui a attentivement relu ma thèse)*, Vlad Posea, Gideon Smedding, Yann Genevois, Florin Pop, Ciprian Dobre, Romain Lacroix, Simon Bouland, Alexandru Costan, Catalin Leordeanu, Nicolas Coste, Etienne Lantreibecq, Jan Stöcker, Yves Guerte, Iker Bellicot, Vincent Powazny, Christine McKinty *et* Alain Kaufmann.

Ma reconnaissance va aussi à celles et ceux qui m'ont, ponctuellement, apporté leur aide: Helen Pouchot, Dominique Moreira, Patrick Farail, Pierre Gaufillet, Pascal Raymond, Xavier Clerc *et* Claude Helmstetter.

Enfin, je souhaite exprimer mon immense gratitude envers mes amis et ma famille, plus particulièrement mes parents et Cristiana, *ma compagne, qui m'ont soutenu et encouragé sans répit durant les trois années qui viennent de s'écouler.*

Table des matières

I Préambule **1**

Notations **7**

1 Ingénierie dirigée par les modèles **9**
1.1 Principes . . . 9
1.2 Implantation dans Eclipse . . . 11
1.2.1 Définition et manipulation de méta-modèles . . . 12
1.2.2 Editeurs de modèles . . . 15
1.2.3 Transformation de modèles vers modèles . . . 18
1.2.4 Transformations entre modèles avec ATL . . . 19
1.2.5 Transformations entre modèles avec Kermeta . . . 21
1.2.6 Transformations de modèles avec Java . . . 22
1.2.7 Génération de code (transformation de modèles vers texte) . . . 23
1.3 Le projet Topcased . . . 25

2 CADP et LOTOS NT **27**
2.1 Méthodes formelles . . . 27
2.2 CADP . . . 28
2.2.1 Les systèmes de transitions . . . 28
2.2.2 Algèbres de processus . . . 28
2.2.3 Techniques de vérification . . . 29
2.2.4 Les outils . . . 29
2.3 Présentation de LOTOS NT . . . 30
2.4 Sous-ensemble de LOTOS NT utilisé . . . 30
2.4.1 Identificateurs . . . 30
2.4.2 Définitions de modules . . . 31
2.4.3 Définitions de types . . . 31
2.4.4 Définitions de fonctions . . . 32
2.4.5 Instructions . . . 32
2.4.6 Expressions . . . 34
2.4.7 Définitions de canaux de communication . . . 35
2.4.8 Définitions de processus . . . 36
2.4.9 Comportements . . . 36
2.4.10 Exemple . . . 39

II Modélisation et vérification de systèmes GALS **41**

3 Motivations **43**
3.1 Paradigme synchrone . . . 43
3.1.1 Systèmes réactifs . . . 43
3.1.2 Modèle de calcul synchrone . . . 44
3.1.3 Langages de programmation synchrone . . . 45
3.1.4 Vérification des programmes synchrones . . . 46
3.2 Systèmes GALS . . . 47
3.3 Etat de l'art sur la vérification de systèmes GALS . . . 48
3.4 Contributions . . . 49

4 Approche proposée **53**
4.1 Description générale de l'approche . . . 53
4.2 Encodage des programmes synchrones . . . 53
4.3 Encapsulation dans un processus asynchrone . . . 54
4.4 Communication avec l'environnement . . . 56
4.5 Le langage SAM . . . 57
4.5.1 Présentation . . . 57
4.5.2 Sémantique . . . 58
4.6 Traduction de SAM en LOTOS NT . . . 60

5 Application à une étude de cas industrielle **63**
5.1 Description de l'étude de cas . . . 63
5.1.1 Protocole TFTP . . . 63
5.1.2 Variante Airbus du protocole TFTP . . . 64
5.2 Modélisation en LOTOS NT . . . 65
5.2.1 Modélisation d'entités TFTP simplifiées . . . 65
5.2.2 Modélisation d'entités TFTP réalistes . . . 66
5.2.3 Modélisation des liens de communication . . . 67
5.2.4 Composition parallèle des liens de communication et des entités TFTP . . . 69
5.3 Description formelle des propriétés de bon fonctionnement . . . 69
5.3.1 Logique temporelle RAFMC de CADP . . . 70
5.3.2 Logique temporelle MCL de CADP . . . 71
5.3.3 Propriétés exprimées en RAFMC . . . 72
5.3.4 Propriétés exprimées en MCL . . . 76
5.3.5 Classification des propriétés . . . 82
5.4 Vérification fonctionnelle des modèles . . . 83
5.4.1 Génération de l'espace d'états . . . 83
5.4.2 Génération directe . . . 84
5.4.3 Génération compositionnelle – niveau 1 . . . 85
5.4.4 Génération compositionnelle – niveau 2 . . . 87
5.4.5 Génération compositionnelle – niveau 3 . . . 88
5.4.6 Vérification à la volée . . . 90
5.4.7 Résultats de vérification . . . 91
5.5 Evaluation des performances par simulation . . . 92
5.5.1 Méthodologie de simulation avec CADP . . . 93
5.5.2 Résultats de simulation . . . 94

III Modélisation et vérification de services Web **97**

6 Les services Web **99**
6.1 Les origines des services Web . . . 99
6.2 Les solutions W3C : l'approche par les services Web . . . 100
6.3 Les langages de description de services Web . . . 101
6.4 Vérification des services Web . . . 102
6.5 Vérification des données uniquement . . . 103
6.6 Vérification du comportement uniquement . . . 103
6.7 Vérification du comportement en tenant compte des données . . . 105
6.8 Comparaison des approches de vérification des services Web . . . 106
6.9 Considérations pragmatiques . . . 108
6.10 Contributions . . . 108

7 Traduction des types XML Schema **111**
7.1 Les termes de XML . . . 111
7.2 Présentation de XML Schema . . . 113
7.2.1 Types spéciaux . . . 113
7.2.2 Types prédéfinis . . . 113
7.2.3 Types simples . . . 115
7.2.4 Types complexes . . . 117
7.2.5 Types complexes à contenu simple . . . 117
7.2.6 Types complexes à contenu complexe . . . 117
7.2.7 Attributs . . . 118
7.2.8 Eléments . . . 119
7.2.9 Récursivité . . . 120
7.2.10 Exemple d'utilisation de XML Schema . . . 120
7.3 Grammaire abstraite pour XML Schema . . . 121
7.4 Traduction en LOTOS NT . . . 122
7.4.1 Etat de l'art . . . 122
7.4.2 Principe général de traduction . . . 123
7.4.3 Domaines et types de base . . . 125
7.4.4 Les conversions . . . 126
7.4.5 Les itérateurs . . . 127
7.4.6 Traduction des types prédéfinis . . . 128
7.4.7 Conversion des types prédéfinis . . . 129
7.4.8 Itérateurs pour les types prédéfinis . . . 130
7.4.9 Traduction des types simples . . . 130
7.4.10 Conversion des types simples . . . 133
7.4.11 Itérateurs pour les types simples . . . 133
7.4.12 Traduction des attributs des types complexes . . . 135
7.4.13 Traduction des éléments racine . . . 136
7.4.14 Conversion des éléments racine . . . 137
7.4.15 Itérateurs pour les éléments racine . . . 137
7.4.16 Traduction des éléments des types complexes . . . 137
7.4.17 Traduction des types complexes . . . 139
7.5 Traduction des constantes BPEL . . . 140
7.5.1 Grammaire des constantes BPEL . . . 142
7.5.2 Principe général de traduction des constantes . . . 142
7.5.3 Traduction des constantes de type simple . . . 143
7.5.4 Traduction des constantes de type élément racine . . . 143
7.5.5 Traduction d'un attribut dans le contenu d'une constante . . . 144

7.5.6 Traduction des attributs dans le contenu d'une constante . . . 145
7.5.7 Traduction d'un sous-élément dans le contenu d'une constante . . . 146
7.5.8 Traduction des sous-éléments dans le contenu d'une constante . . . 148
7.5.9 Traduction du contenu simple d'une constante . . . 152
7.5.10 Traduction du contenu complexe d'une constante . . . 152

8 Traduction des expressions XPath 155
8.1 Modèle de données . . . 155
8.2 Partitionnement d'un arbre XML autour d'un nœud . . . 157
8.3 Syntaxe normalisée . . . 159
8.4 Sémantique statique normalisée . . . 161
8.5 Sémantique dynamique normalisée . . . 162
8.5.1 Contexte d'évaluation . . . 162
8.5.2 Sémantique dénotationnelle . . . 163
8.6 Traduction en LOTOS NT . . . 166
8.6.1 Etat de l'art . . . 166
8.6.2 Sous-ensemble utile de XPath . . . 167
8.6.3 Sémantique statique simplifiée . . . 168
8.6.4 Sémantique opérationnelle simplifiée . . . 177

9 Traduction des interfaces WSDL 181
9.1 Présentation du langage WSDL . . . 181
9.1.1 Messages . . . 182
9.1.2 Fonctions distantes . . . 182
9.1.3 Fonctionnalité d'un service . . . 183
9.1.4 Types de liens de communication . . . 183
9.1.5 Grammaire WSDL . . . 183
9.2 Traduction en LOTOS NT . . . 184
9.3 Etat de l'art des traductions d'interfaces WSDL . . . 184
9.4 Définition de la traduction . . . 184
9.5 Traduction d'une interface WSDL . . . 186
9.6 Traduction des messages . . . 186
9.7 Traduction des fonctionnalités de services . . . 186
9.7.1 Traduction des fonctionnalités de service . . . 187
9.7.2 Traduction des types de liens de communication . . . 187
9.7.3 Traduction des exceptions . . . 187

10 Traduction des services BPEL 189
10.1 Présentation du langage BPEL . . . 189
10.1.1 Modèle de communication . . . 189
10.1.2 Vue d'ensemble d'un service Web BPEL . . . 190
10.1.3 Relations entre activités, services et sous-services . . . 192
10.1.4 Abréviations syntaxiques . . . 192
10.1.5 Syntaxe concrète d'un service BPEL . . . 193
10.1.6 Attribut spécial "suppressJoinFailure" . . . 194
10.1.7 Variables . . . 195
10.1.8 Inclusion de fichiers XML Schema et WSDL . . . 195
10.1.9 Extensions du langage BPEL . . . 195
10.1.10 Liens de communication . . . 196
10.1.11 Mécanisme d'échanges de messages . . . 196

10.1.12 Mécanisme de corrélation . . . 197
10.1.13 Gestionnaires d'exceptions . . . 198
10.1.14 Gestionnaires d'événements . . . 199
10.1.15 Activités . . . 200
10.1.16 Activité "empty" . . . 200
10.1.17 Activité "exit" . . . 201
10.1.18 Activité "wait" . . . 201
10.1.19 Activité "receive" . . . 201
10.1.20 Activité "reply" . . . 202
10.1.21 Activité "invoke" . . . 202
10.1.22 Activité "throw" . . . 203
10.1.23 Activité "rethrow" . . . 203
10.1.24 Activité "assign" . . . 203
10.1.25 Activité "validate" . . . 205
10.1.26 Activité "sequence" . . . 205
10.1.27 Activité "if" . . . 205
10.1.28 Activité "while" . . . 206
10.1.29 Activité "repeatUntil" . . . 206
10.1.30 Activité "forEach" . . . 206
10.1.31 Activité "pick" . . . 207
10.1.32 Activité "flow" et liens de contrôle . . . 208
10.1.33 Activité "extensionActivity" . . . 210
10.1.34 Activité "scope" et notion de sous-service . . . 211
10.1.35 Gestionnaires de compensation . . . 212
10.2 Traduction en LOTOS NT . . . 213
10.2.1 Restrictions . . . 213
10.2.2 Syntaxe abstraite du sous-ensemble de BPEL considéré . . . 216
10.2.3 Paramètre de la fonction de traduction . . . 216
10.2.4 Traduction des liens de communications . . . 216
10.2.5 Préservation de l'atomicité des activités atomiques . . . 218
10.2.6 Gestion des variables partagées . . . 220
10.2.7 Gestion des exceptions . . . 221
10.2.8 Gestion des événements . . . 225
10.2.9 Traduction de l'activité "empty" . . . 227
10.2.10 Traduction de l'activité "exit" . . . 227
10.2.11 Traduction de l'activité "wait" . . . 228
10.2.12 Traduction de l'activité "receive" . . . 228
10.2.13 Traduction de l'activité "reply" . . . 228
10.2.14 Traduction de l'activité "invoke" . . . 229
10.2.15 Traduction de l'activité "assign" . . . 231
10.2.16 Traduction de l'activité "throw" . . . 233
10.2.17 Traduction de l'activité "rethrow" . . . 233
10.2.18 Traduction de l'activité "validate" . . . 234
10.2.19 Traduction de l'activité "sequence" . . . 234
10.2.20 Traduction de l'activité "if" . . . 234
10.2.21 Traduction de l'activité "while" . . . 235
10.2.22 Traduction de l'activité "repeatUntil" . . . 235
10.2.23 Traduction de l'activité "forEach" . . . 236
10.2.24 Traduction de l'activité "pick" . . . 237

10.2.25 Traduction de l'activité "flow" et des liens de contrôle 238
10.2.26 Traduction des activités au sein de l'activité "flow" 243

11 Conception d'un traducteur automatique **249**
11.1 Choix du langage de programmation 249
11.2 Réalisation 250
11.3 Espaces de noms 251

IV Conclusion **253**

Bibliographie **261**

Partie I

Préambule

Introduction

Cette thèse s'est effectuée en co-tutelle entre l'Université Joseph Fourier de Grenoble, France et l'Université Polytechnique de Bucarest, Roumanie. Les travaux de recherche ont été effectués au centre INRIA de Grenoble Rhône-Alpes, sous la direction de Hubert Garavel, et au Laboratoire d'Informatique de la Faculté d'Automatique et d'Ordinateurs de l'Université Polytechnique de Bucarest, sous la direction de Valentin Cristea.

Vérification des systèmes asynchrones

Un système asynchrone est constitué d'un ensemble de composants qui évoluent indépendamment et qui communiquent ponctuellement pour se synchroniser et éventuellement échanger des données. Les systèmes asynchrones se distinguent donc des systèmes synchrones dans lesquels les composants évoluent à l'unisson selon une horloge commune. En effet, même s'il a été démontré que le paradigme de programmation asynchrone et le paradigme de programmation synchrone sont équivalents [Mil83], les langages asynchrones et synchrones sont adaptés à la description de classes de systèmes différentes. De nos jours, les systèmes asynchrones sont omniprésents (protocoles de communication, processeurs multi-cœurs...) et tendent à s'imposer dans des domaines autrefois exclusivement constitués de systèmes synchrones, tels que les contrôleurs critiques embarqués (dans les avions, voitures, trains...) ou les circuits intégrés (généralisation des systèmes sur puce).

La validation du bon comportement de tels systèmes est un domaine de recherche important qui englobe diverses techniques telles que le test classique, le test unitaire, la simulation, l'analyse de code et la vérification formelle. La vérification formelle consiste à analyser un modèle du système pour vérifier qu'il remplit des critères d'exigences définis sous la forme de propriétés de bon fonctionnement (le plus souvent exprimées par des formules de logique temporelle), on parle alors de vérification exhaustive ou *model checking* [CES83], ou bien sous la forme d'un second modèle du système, on parle alors de vérification par équivalence ou *equivalence checking* [NH84].

La vérification formelle est très efficace[1], mais très complexe à mettre en oœuvre. En effet, cette technique nécessite d'avoir à disposition des outils performants mais aussi de connaître parfaitement les langages d'entrée de ces outils : les algèbres de processus pour la modélisation des systèmes ainsi que les logiques temporelles pour la spécification des propriétés de bon fonctionnement. Dans cette thèse, nous considérons les cas de la boîte à outils de vérification formelle CADP [GLMS11], et de son plus récent langage d'entrée : l'algèbre de processus LOTOS NT [Sig00, CCG+10].

[1] `http://ercim-news.ercim.eu/model-checking-of-safety-critical-software-for-avionics`

Ingénierie dirigée par les modèles

Ces dernières années ont vu l'apparition d'une nouvelle approche de développement logiciel appelé l'ingénierie dirigée par les modèles [Ken02] (MDE ou *Model-Driven Engineering* en anglais). Dans ce paradigme, les modèles d'une application ont un rôle fondamental. Chaque modèle représente l'application à un niveau d'abstraction particulier. La réalisation de l'application s'effectue par transformations successives, manuelles, automatisées ou semi-automatisées, des modèles jusqu'à l'obtention d'un modèle final.

Grâce à des outils appropriés, l'approche IDM facilite la création d'éditeurs, de transformateurs et de générateurs de code pour les modèles. La vérification formelle s'inscrit aisément dans l'approche IDM. En effet, en considérant comme modèle final d'une chaîne de transformations, un modèle décrit dans une algèbre de processus, il devient possible de combiner l'approche IDM et la vérification formelle. En appliquant ce principe, l'environnement TOPCASED[2], créé par extension de l'environnement de développement intégré (IDE ou *Integrated Developement Environment* en anglais) Eclipse, vise à connecter des langages de modélisation d'applications critiques (de l'industrie aéronautique notamment) à des outils de vérification formelle tels que CADP[3].

Langages dédiés

DSL [VDKV00] (*Domain-Specific Language*) est un terme générique désignant les langages de programmation ou de modélisation dédiés à un domaine particulier, par opposition aux langages de programmation (C, Java) ou de modélisation (UML [OMG97]) classiques qui ont une vocation plus générale. Un DSL exhibe usuellement un nombre restreint de constructions de façon à pouvoir décrire aussi succintement que possible les applications relevant du domaine qu'il couvre. Les modèles d'applications écrits dans les langages dédiés sont donc souvent simples et se prêtent bien à l'approche IDM. Les exemples de langages dédiés incluent : SQL [Cod70] pour interroger des systèmes de gestion de bases de données, YACC [Joh75] pour décrire des analyseurs syntaxiques ou encore les langages de *shell script* des systèmes UNIX tels que `sh` et `csh`.

Dans cette thèse, nous considérons les cas particuliers de deux DSLs : SAM [CGT08], dans le cadre des travaux effectués à l'INRIA et BPEL [Com01], dans le cadre des travaux effectués à l'Université Polytechnique de Bucarest. SAM est un langage de programmation synchrone défini par Airbus pour spécifier des systèmes de contrôle pour les avions tandis que BPEL est une norme du consortium OASIS[4] pour la définition de services Web. Ces deux langages, bien que sémantiquement très différents, possèdent de nombreux points communs :

- Ce sont des langages dédiés pour lesquels Eclipse possède des environnements de travail.
- Ils peuvent servir à la définition d'applications critiques : des contrôleurs avioniques pour SAM et des applications de commerce électronique pour BPEL.
- Ils peuvent définir des systèmes concurrents.
- Ils sont utilisés industriellement.
- Il existe un besoin de vérification formelle pour les applications décrites dans ces langages.

[2] `http://www.topcased.org`
[3] `http://vasy.inria.fr/cadp`
[4] `http://www.oasis-open.org`

Plan de la thèse et contributions

L'objectif de cette thèse est de rapprocher de nouveaux langages de modélisation pour les systèmes asynchrones et les méthodes de vérification formelle. Nous prenons l'exemple de deux langages : le premier, Sam, est un langage graphique conçu spécifiquement pour l'approche Ide tandis que le second, Bpel, est un langage à syntaxe Xml, compatible avec l'approche Ide. Nous établissons une continuité entre ces langages et les langages formels par le biais de traductions sémantiques afin de pouvoir appliquer des techniques de vérification formelle.

Les contributions de la thèse sont détaillées dans la présentation des chapitres qui suit.

Dans une **première partie**, nous introduisons les technologies autour desquelles les travaux de cette thèse ont été réalisés :

- Au chapitre 1, nous détaillons les principes de l'approche Idm puis décrivons l'environnement de travail Eclipse avant de dresser un état de l'art des technologies présentes dans Eclipse pour la mise en œuvre de l'approche Idm. Le projet Topcased qui vise à simplifier la combinaison des techniques de vérification formelle avec l'approche Idm est ensuite présenté.

- Au chapitre 2, nous présentons la boîte à outils Cadp : ses langages, ses outils et ses bibliothèques. Ensuite, nous exposons la sémantique du langage Lotos NT, l'un des formalismes d'entrée de Cadp, qui tient une place centrale dans cette thèse.

Dans une **seconde partie**, nous traitons de la vérification de systèmes Gals (*Globalement Asynchrone Localement Synchrone*). Les systèmes Gals ont été largement étudiés dans la littérature, mais quasiment exclusivement par des auteurs de la communauté des langages synchrones. Nous proposons une méthode générique pour vérifier formellement des composants synchrones immergés dans un environnement asynchrone. Nous illustrons cette méthode par son application à une étude de cas industrielle fournie par Airbus dans le cadre du projet Topcased. Il s'agit de vérifier formellement que deux programmes synchrones écrits en Sam et décrivant un protocole de communication, se comportent correctement si le lien qui les connecte est asynchrone. Afin de traiter cette étude de cas, nous utilisons les outils de Idm fournis par Topcased et faisons un retour d'expérience sur cette utilisation.

Le plan de cette partie est le suivant :

- Au chapitre 3, nous présentons les langages synchrones ainsi que les outils et techniques de vérification qui leur sont associés. Ensuite, nous introduisons les systèmes Gals et détaillons les techniques de vérification existantes pour ces systèmes hybrides.

- Au chapitre 4, nous définissons notre approche générique pour la vérification de systèmes Gals par combinaison de langages synchrones et algèbres de processus. Puis, nous illustrons cette méthode à l'aide des langages Sam et Lotos NT.

- Au chapitre 5, nous commençons par présenter l'étude d'un système de communication basé sur une variante du protocole Tftp, qui nous a été fournie par Airbus. Puis, nous modélisons, en Lotos NT, les composants synchrones Sam de ce système et les intégrons au sein d'un environnement asynchrone écrit en Lotos NT. Ensuite, nous présentons les propriétés de bon fonctionnement que le système doit vérifier avant d'expliquer leur expression dans les logiques temporelles supportées par Cadp. Après, nous présentons des techniques visant à réduire la taille de l'espace d'états de la spécification Tftp générée. Enfin, nous listons les erreurs découvertes et pour chacune d'elles, nous effectuons des simulations du système avec Cadp afin d'analyser son impact sur les performances.

Dans une **troisième partie**, nous traitons de la vérification formelle du langage Bpel. De nombreuses

approches ont été présentées dans la littérature afin de vérifier formellement des spécifications écrites avec BPEL. Pourtant, aucune ne donne entièrement satisfaction. C'est pourquoi nous formalisons une traduction de BPEL vers LOTOS NT, dans le but de vérifier les spécifications LOTOS NT produites avec CADP. Dans cette traduction, nous nous efforçons de traiter de la façon la plus complète possible non seulement BPEL, mais aussi les différents langages sur lesquels il repose pour la définition des types de données, des canaux de communication et des expressions de données. La mise en œuvre de cette traduction donne l'occasion de discuter de l'application de l'approche IDM à des transformations de grande complexité.

Le plan de cette partie est le suivant :

- Au chapitre 6, nous présentons un historique des services Web, puis nous détaillons les langages actuels pour leur définition (XML Schema pour les structures de données utilisées dans les services, XPATH pour les expressions figurant dans les services, WSDL pour les interfaces des services et BPEL pour le comportement des services). Ensuite, nous détaillons les approches de vérification de services Web existantes avant de les comparer à la nôtre. Enfin, nous présentons notre base d'exemples de services BPEL que nous avons constituée au fil de recherches sur Internet et qui nous a permis d'identifier les constructions de XML Schema, XPATH, WSDL et BPEL réellement utiles aux services Web.

- Au chapitre 7, nous présentons le langage XML Schema qui décrit les structures de données utilisées dans les services BPEL. Ensuite, nous définissons formellement la traduction des constructions XML Schema réellement utiles au services Web en LOTOS NT, en tenant compte des spécificités de la vérification formelle. Notamment, nous optimisons les types LOTOS NT produits afin de réduire autant que possible le temps de génération et la mémoire consommée lors de la génération de l'espace d'états.

- Au chapitre 8, nous présentons le langage XPATH dans lequel sont écrites les expressions figurant dans les constructions du langage BPEL. Ensuite, nous définissons formellement la traduction des expressions XPATH réellement utiles aux services Web en LOTOS NT, en distinguant deux cas, selon que les expressions XPATH figurent en partie gauche ou en partie droite dans l'opérateur d'affectation de BPEL.

- Au chapitre 9, nous présentons le langage WSDL dans lequel est définie l'interface du service. Ensuite, nous donnons la traduction des constructions WSDL en LOTOS NT afin de typer les liens de communication par lesquels le service communique.

- Au chapitre 10, nous présentons toutes les constructions du langage BPEL. Puis, nous éliminons celles qui ne figurent pas dans notre base d'exemples ou qui ne présentent pas d'intérêt pour la vérification formelle. Ensuite, nous donnons la traduction formelle des constructions restantes en LOTOS NT.

- Au chapitre 11, nous expliquons les choix d'implémentation que nous avons faits lors du développement du traducteur automatique de BPEL vers LOTOS NT et discutons de l'application de l'approche IDM à des transformations qui relèvent de la compilation de programmes.

Enfin, au chapitre 11.3, nous concluons ce document par une analyse des contributions réalisées et un panorama des perspectives ouvertes par notre travail.

Notations

La définition des notations utilisées dans ce document est donnée ici.

Symboles arithmétiques, logiques et ensemblistes

Les symboles arithmétiques usuels sont employés, ainsi que les opérateurs logiques suivants :

notation	*signification*
`false`	proposition toujours fausse
`true`	proposition toujours vraie
$\neg P_0$	négation de la proposition P_0
$P_1 \vee P_2$	conjonction des propositions P_1 et P_2
$P_1 \wedge P_2$	disjonction des propositions P_1 et P_2
$P_1 \implies P_2$	implication de la proposition P_2 par P_1
$P_1 \iff P_2$	équivalence des propositions P_1 et P_2

On utilise également les opérateurs ensemblistes classiques :

notation	*signification*
$\emptyset$	ensemble vide
$\{x_m, ..., x_n\}$	ensemble ($\emptyset$ si $m > n$)
$\text{card}(E)$	nombre d'éléments dans l'ensemble E
$E_1 \cap E_2$	intersection des ensembles E_1 et E_2
$E_1 \cup E_2$	union des ensembles E_1 et E_2
$E_1 - E_2$	différence des ensembles E_1 et E_2
$E_1 \times E_2$	produit cartésien des ensembles E_1 et E_2
$\langle\rangle$	liste[5] vide
$()$	*idem*
$\langle x_m, ..., x_n \rangle$	liste ($\langle\rangle$ si $m > n$)
$(x_m, ..., x_n)$	*idem*

Enfin on emploie fréquemment des notations abrégées de la forme "$N_m + ... + N_n$", "$P_m \wedge ... \wedge P_n$", "$E_m \cup ... \cup E_n$", ... dont la signification est définie formellement comme suit ; si "$\star$" représente une opération binaire associative admettant un élément neutre "ϵ", on note "$x_m \star ... \star x_n$" l'expression $f(m, n)$ définie par :

$$f(i,j) = \begin{cases} \textbf{si } i > j \textbf{ alors } \epsilon \\ \textbf{sinon } x_i \star f(i+1, j) \end{cases}$$

[5] ou n-uplet, ou séquence

Méta-langage de description syntaxique

Pour les définitions syntaxiques on utilise le méta-langage défini comme suit :

- les symboles terminaux sont notés en caractères gras
- les symboles non-terminaux sont notés en caractères italiques
- le méta-symbole "≡" introduit une règle syntaxique, qui associe au non-terminal situé en partie gauche sa définition figurant en partie droite
- la concaténation est notée par juxtaposition
- le méta-symbole "|" dénote le choix entre ses deux opérandes
- le méta-symbole "[ ]" dénote zéro ou une occurrence de l'opérande qu'il encadre[6]

Par commodité et pour différencier les différentes occurrences d'un même symbole non-terminal dans une partie droite de règle, on utilise des notations étendues[7] qui tiennent compte du contexte :

- si A est un non-terminal alors A', A'', A_i, A'_i et A''_i dénotent des non-terminaux ayant la même définition syntaxique que A
- la notation "$A_1, ..., A_n$" désigne la concaténation de n éléments A. La valeur de n ($n \geq 0$) est déterminée par l'analyse syntaxique et A_i dénote le i$^{\text{ème}}$ élément de cette séquence
- la notation "$A_0, ..., A_n$" désigne la concaténation de $n+1$ éléments A. La valeur de n ($n \geq 0$) est déterminée par l'analyse syntaxique et A_i dénote le i$^{\text{ème}}$ élément de cette séquence
- la notation "$\widehat{A}$" désigne la concaténation d'un nombre non nul d'éléments de A

Méta-langage de description sémantique

Pour définir la sémantique, on utilise des *grammaires attribuées* [Knu68], formalisme qui nous semble préférable aux systèmes de règles [Plo81] parce qu'il requiert que les définitions soient constructives, donc susceptibles d'être implémentées efficacement.

A chaque unité syntaxique on peut associer trois sortes de paramètres :

attributs hérités : ce sont des informations reçues de l'environnement. Leur valeur est calculée de manière *descendante* car, dans un arbre abstrait, les attributs hérités sont transmis par un noeud à ses fils. Dans la grammaire attribuée ces attributs sont précédés du méta-symbole "↓"

attributs synthétisés : ce sont des informations fournies à l'environnement. Leur valeur est calculée de manière *ascendante* puisque, dans un arbre abstrait, les attributs synthétisés sont transmis par un noeud à son père. Ces attributs sont précédés du méta-symbole "↑"

attributs locaux : ce sont des informations attachées aux unités syntaxiques et dont il est possible de consulter et de modifier la valeur

[6] ne pas confondre les méta-symbole "|", "[" et "]" avec les symboles terminaux "|", "`[`" et "`]`"
[7] variables syntaxiques

Chapitre 1

Ingénierie dirigée par les modèles

1.1 Principes

L'ingénierie dirigée par les modèles (IDM, en abrégé) est une méthodologie de développement logiciel dérivée des pratiques du monde de la programmation à objets. Ces pratiques consistent à définir, dans un langage graphique, un modèle qui représente les classes (nom, attributs, signatures des méthodes...) de l'application et leurs relations (héritage, relations d'inclusion...). Parmi ces langages graphiques, dits de *modélisation*, le plus connu est UML [OMG97] (*Unified Modeling Language*) qui a été défini, au milieu des années 1990, au sein de l'entreprise Rational Software et qui a été standardisé en 1997 par le consortium OMG (*Object Management Group*) dans le but d'unifier les multiples langages de modélisation qui cohabitaient à cette époque.

A l'origine, les langages de modélisation tels qu'UML servaient à représenter l'application aux différents acteurs engagés dans sa fabrication : clients, architectes, programmeurs. Ces représentations étaient alors transformées en squelettes d'applications (déclarations de classes avec leurs attributs et méthodes) dans le langage à objets visé. Il s'agissait, ensuite, de remplir manuellement les morceaux de code manquants, qui ne pouvaient être spécifiés dans le langage de modélisation considéré. Ces pratiques étaient mises en œuvre dans des outils dits CASE (*Computer-Aided Software Engineering*), tels que l'environnement de développement Rational Rose conçu par Rational Software. Après 1997, UML s'est imposé comme le langage de modélisation de référence, et la plupart des outils CASE l'ont peu à peu adopté. UML a été progressivement amélioré, sous l'impulsion de Rational Software, puis d'IBM (qui a acquis Rational Software en 2003), jusqu'à l'adoption de la norme UML 2 [OMG07b] par le consortium OMG en 2005.

UML est un exemple typique de l'évolution des pratiques de développement logiciel. Alors que la première version de la norme avait pour seul but la représentation de classes d'objets et de leurs interactions, le seconde version de la norme propose 14 types de diagrammes différents afin de modéliser d'autres aspects de la création d'une application : diagrammes d'activité pour le flux de contrôle, automates pour représenter le système comme une machine à états, diagrammes de communication pour représenter le passage de messages entre les objects... UML 2 propose aussi un mécanisme normalisé pour définir formellement de nouveaux types de diagrammes. Par conséquent, des outils comme Rational Rose peuvent désormais générer une grande partie du code d'une application à partir des différents diagrammes UML 2 décrivant cette application.

L'ingénierie dirigée par les modèles [Ken02] va au delà d'UML et pousse à l'extrême les pratiques de modélisation en prônant l'utilisation de modèles à toutes les étapes du développement logiciel. Il s'agit

d'utiliser des outils, eux-mêmes conçus à partir de modèles, pour effectuer la plupart des opérations sur les modèles (transformation, édition, génération de code...). Concrètement, l'idée principale est d'avoir des modèles pour chaque aspect du développement, par exemple :

- un modèle abstrait pour raisonner sur l'application et les interactions entre ses composants,
- un modèle concret dans lequel le comportement de l'application peut être spécifié et qui peut être transformé en code exécutable,
- un modèle intuitif dans lequel les clients peuvent exprimer le cahier des charges de l'application,
- un modèle pour décrire la base de tests...

L'approche IDM consiste à construire, tester et, éventuellement, vérifier formellement une application, uniquement à partir de modèles, à l'aide de transformations successives, automatisées ou semi-automatisées, qui peuvent prendre plusieurs modèles en entrée, pour produire un ou plusieurs modèles de sortie.

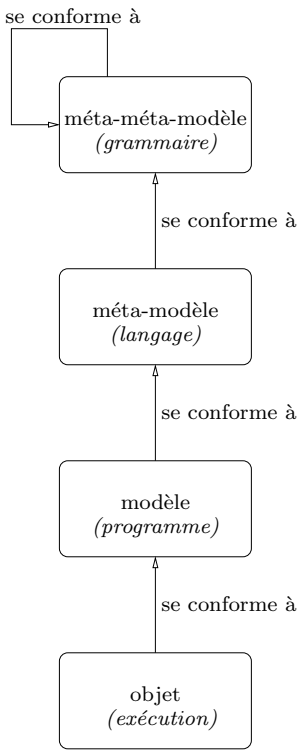

Figure 1.1: Hiérarchie des modèles dans l'approche IDM

Une hiérarchie de modèles à quatre niveaux (voir figure 1.1) définit les différents types de modèles. Nous les décrivons en dressant un parallèle avec les langages de programmation classiques :

- Au niveau **M3** : un *méta-méta-modèle* est un langage qui permet la définition de la syntaxe abstraite d'un méta-modèle, en suivant le même principe que les langages de grammaire (comme BNF) qui définissent une grammaire de syntaxe concrète. Un méta-méta-modèle doit pouvoir s'auto-définir.
- Au niveau **M2** : un *méta-modèle* est un langage pour la définition de modèles, tout comme les langages de programmation permettent la définition de programmes.

- Au niveau **M1** : un *modèle* est une instance d'un méta-modèle, au même titre qu'un programme est une instance d'un langage.
- Au niveau **M0** : un *objet* est une instance d'un modèle, au même titre que l'exécution d'un programme est une instance de ce programme.

La différence entre les langages de programmation classiques et les méta-modèles réside dans le fait que, pour les premiers, on manipule principalement une syntaxe concrète, tandis que pour les seconds, on manipule exclusivement la syntaxe abstraite. L'approche IDM consiste donc à construire un langage en définissant sa syntaxe abstraite au moyen d'un méta-modèle pour ensuite créer des outils de manipulation, d'édition ou de transformation pour ce langage. Cette approche est particulièrement bien adaptée aux DSLs dont la syntaxe abstraite succincte permet de décrire des modèles de systèmes. Pour que l'approche IDM puisse fonctionner, il faut faciliter, pour un méta-modèle donné, le processus de création des outils d'édition, de transformation et de génération de code. A cette fin, des environnements de développement comme Eclipse (que nous présentons à la section suivante) proposent des infrastructures logicielles dédiées à la mise en œuvre de l'approche IDM.

En termes de normalisation, le consortium OMG a défini un ensemble de normes qui décrivent la vision qu'a ce consortium de l'approche IDM. Ces normes sont regroupées sous le nom commun : MDA [OMG01] (*Model-Driven Architecture*). En plus d'UML, les normes suivantes s'inscrivent dans le cadre de MDA :

- MOF [OMG06b] (*Meta-Object Facility*) est le méta-méta-modèle qui permet de définir tous les autres langages MDA (dont UML).
- OCL [OMG10] (*Object Constraint Language*) est un DSL permettant d'insérer des contraintes sémantiques (exprimées sous forme d'invariants) dans des modèles définis par des méta-modèles eux-mêmes définis en MOF.
- XMI [OMG05] (XML *Metadata Interchange*) est un format XML pour enregistrer des modèles et méta-modèles exprimés en MOF.
- MTL [OMG08a] (MOF *Model-To-Text Transformation Language*) est un langage pour la transformation de modèles vers du texte.
- QVT [OMG07a] (*Query/View/Transformation*) est une collection de trois DSL définis avec MOF et dédiés à la spécification de transformations entre modèles. Ces trois langages sont respectivement nommés *Relational*, *Core* et *Operational Mappings*. Relational est un langage de transformation déclaratif avec deux syntaxes concrètes, l'une textuelle et l'autre graphique. Operational Mappings est un langage textuel de transformation à syntaxe impérative. Core est un langage de transformation déclaratif de bas niveau vers lequel la norme définit des transformations depuis Relational et Operational Mappings. Un mécanisme pour invoquer des transformations écrites dans des langages comme XSLT [Gro03b] (*Extensible Stylesheet Language Transformations*) ou XQUERY [Gro07] est aussi décrit.

1.2 Implantation dans Eclipse

Eclipse est un environnement de développement intégré libre (sous license *open source*) qui est écrit en Java. Le développement d'Eclipse a été sponsorisé par IBM qui s'en sert de fondation pour ses outils commerciaux WebSphere et Rational. Eclipse fonctionne sur un principe de briques logicielles extensibles : chaque nouvelle brique (communément appelée *plug-in*) étend une fonctionnalité existante ou ajoute une nouvelle fonctionnalité, et peut, à son tour, être étendue. Ce système permet à

des entreprises de financer des extensions libres et gratuites pour ensuite les utiliser comme point de départ pour la construction de leurs outils commerciaux. Par exemple, Zend[8], la société qui contrôle l'évolution du langage Php, participe au développement d'une extension libre et gratuite pour Php dans Eclipse (Eclipse Php Development Tools) sur laquelle repose Zend Studio, leur produit commercial phare.

Le développement de la plupart des extensions libres d'Eclipse est financé selon ce schéma ou effectué par des volontaires, souvent issus du monde académique. Il n'est donc pas surprenant de constater que les acteurs du monde de l'ingénierie dirigée par les modèles (c'est-à-dire les membres du consortium Omg et les scientifiques travaillant sur la théorie des modèles) ont uni leurs efforts pour proposer, dans Eclipse, l'ensemble d'outils le plus complet pour la mise en œuvre de l'approche Idm. Les différentes solutions présentes dans Eclipse pour implémenter cette approche sont détaillées dans la suite de cette section :

- définition et manipulation de méta-modèles avec Emf,
- construction déditeurs avec Xtext et Gmf,
- transformation de modèles avec Atl et Kermeta et
- génération de code exécutable avec Xpand et Acceleo.

1.2.1 Définition et manipulation de méta-modèles

Emf [BMPS08] (*Eclipse Modeling Framework*) est une collection d'extensions d'Eclipse qui permet la définition et la manipulation de méta-modèles grâce au méta-méta-modèle Ecore, une implémentation de la norme Mof. Ecore est souvent représenté graphiquement à l'aide d'un diagramme de classe Uml, ce qui peut être troublant au premier abord, mais qui est plus ou moins équivalent à l'écriture, dans un langage de programmation dont la grammaire peut être décrite à l'aide du langage Bnf, d'un analyseur syntaxique (peu importe la classe de l'analyseur, LL, LR, Lalr...) pour traiter des grammaires écrites dans le langage Bnf. Ecore permet de définir un méta-modèle à l'aide de classes (au sens *objet* du terme) et de relations entre ces classes :

- nom des classes,
- attributs des classes,
- relations entre les classes (héritage, association),
- méthodes des classes,
- contraintes sur les classes et les relations (multiplicité),
- classes abstraites, interfaces...

La figure 1.2 présente un sous-ensemble simplifié du méta-méta-modèle Ecore dans lequel la gestion des types, des attributs et des noms des classes est allégée. Ce sous-ensemble est défini en terme de références (EReference), classes (EClass), d'attributs (EAttribute) et de types (EDataType).

EReference modélise le fait qu'une classe peut en référencer une autre. Une référence se compose :

- d'un nom,

[8] `http://www.zend.org`

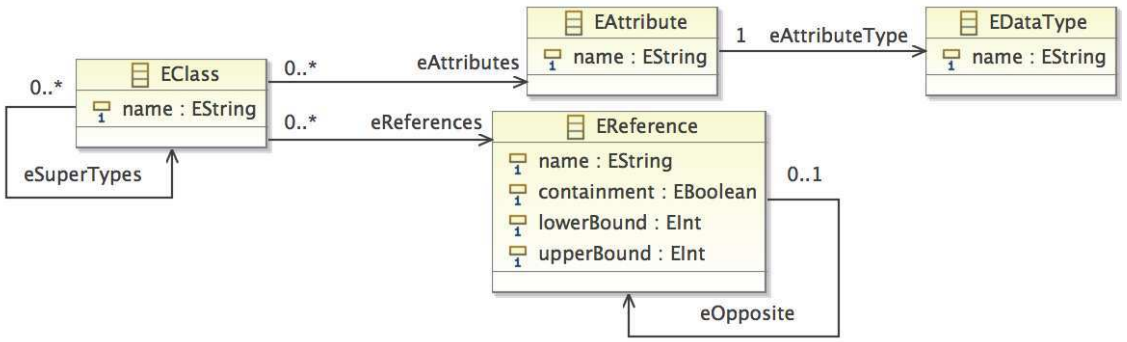

Figure 1.2: Représentation simplifiée du méta-méta-modèle Ecore

- d'un booléen (ici containment) qui indique si la classe définissant la réference possède les instances de la classe cible (dans le vocabulaire Uml, il s'agit d'une *composition* représentée par un losange plein à l'extrémité du conteneur) ou non (association simple),
- de deux bornes (ici lowerBound et upperBound) pour indiquer la multiplicité de la relation,
- et d'un type qui doit être une classe (représenté par une référence, dont le nom est type, à une classe).

Si deux classes A et B se référencent mutuellement, alors les deux références (A vers B et B vers A) se référencent mutuellement via la relation eOpposite.

EClass modélise les classes de l'application. Chaque classe a un nom et peut avoir des attributs et des références. Pour permettre l'héritage, une classe peut référencer d'autres classes en tant que ses super-classes.

EAttribute modélise les attributs d'une classe. Un attribut a un nom et un type (il s'agit en fait d'une référence vers son type).

EDataType modélise les types des attributs qui sont soit des classes, soit des types importés depuis un autre méta-modèle (par exemple, le type "int" du méta-modèle défini pour le langage Java).

Comme Ecore est un méta-méta-modèle, il se conforme à lui-même (autrement dit, on peut décrire Ecore en Ecore). Les lecteurs attentifs auront constaté que cette version simplifiée n'est pas un méta-méta-modèle minimal. En effet, elle peut parfaitement s'auto-décrire sans l'héritage (relation eSuperTypes) et les références inverses (relation eOpposite). En revanche, sans ces deux éléments, elle ne peut plus décrire Ecore qui se construit, entièrement, à partir de ce sous-ensemble simplifié par l'ajout d'un certain nombre de classes (EClass) et de références (EReference). Ces ajouts préservent la conformité de Ecore au sous-ensemble simplifié (et donc à lui-même).

Une méthode d'une classe est représentée en Ecore par une classe EOperation (qui peut être décrite dans le sous-ensemble simplifié) qui a un nom, des références vers une classe EParameter pour ses arguments, des références vers la classe EClass pour le type des exceptions qu'elle peut lever et son type de retour.

Les figures 1.3 et 1.4 montrent deux méta-modèles, définis grâce à Ecore, qui vont servir d'exemples tout au long de ce chapitre. Il s'agit respectivement d'un méta-modèle de structure d'arbre et d'un méta-modèle de structure de liste.

Ces exemples illustrent bien le fait qu'un méta-modèle est en réalité la syntaxe abstraite d'un langage, en l'occurrence des langages d'arbres et de listes. Cette syntaxe abstraite joue un rôle central dans

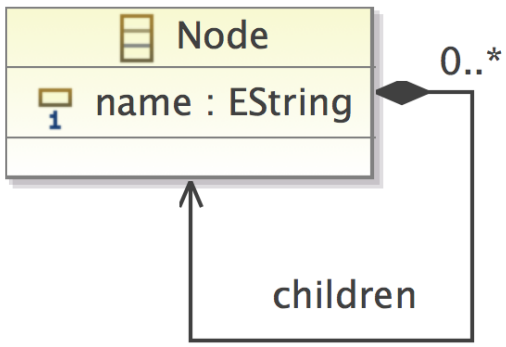

Figure 1.3: Méta-modèle de structure d'arbre modélisé en Ecore

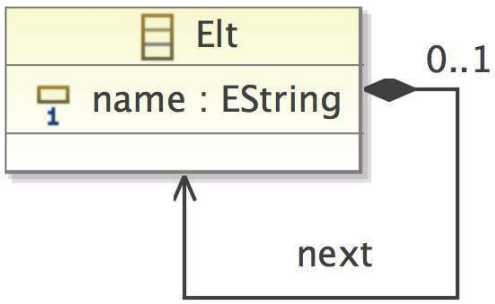

Figure 1.4: Méta-modèle de structure de liste modélisé en Ecore

l'ingénierie dirigée par les modèles et, par conséquent, dans Emf.

En plus de la simple définition de méta-modèles, Emf fournit divers outils logiciels que nous allons utiliser, directement ou indirectement, dans les sections suivantes :

- un générateur de squelettes d'applications en Java à partir de modèles (il s'agit de la vocation première de Emf),
- une implémentation de la norme Xmi pour sauvegarder les méta-modèles sous un format Xml,
- un éditeur graphique générique pour visualiser des modèles sous une forme arborescente et les modifier, il sera détaillé à la section 1.2.2,
- un ensemble de briques logicielles regroupées au sein du sous-projet Emf.Edit pour faciliter la construction d'éditeurs graphiques et textuels et
- une implémentation de la norme Ocl, dans un projet annexe nommé *Validation Framework*[9], pour extraire des informations des modèles et vérifier des invariants.

[9] `http://www.eclipse.org/modeling/emf/?project=validation`

1.2.2 Editeurs de modèles

Eclipse propose divers outils pour les éditeurs. En premier lieu, l'éditeur arborescent de modèles généré par Emf (dont nous montrons une capture d'écran en figure 1.5) est tout à fait fonctionnel. En revanche, il est très simplifié et peu ergonomique. En effet, cet éditeur permet de visualiser ou de modifier un modèle selon son arbre de syntaxe abstraite, sans tenir compte de la sémantique des constructions du modèle (ce qui est normal, car l'éditeur est générique). Pour nos modèles d'arbres et de listes, cet éditeur convient très bien, mais pour des modèles plus complexes, la repreśentation qu'il fait d'un modèle est trop peu concise pour être utilisable en pratique. De plus, la modification d'un objet (valeur d'un attribut ou réference à un autre objet) nécessite l'utilisation d'un second éditeur dit "éditeur de propriétés".

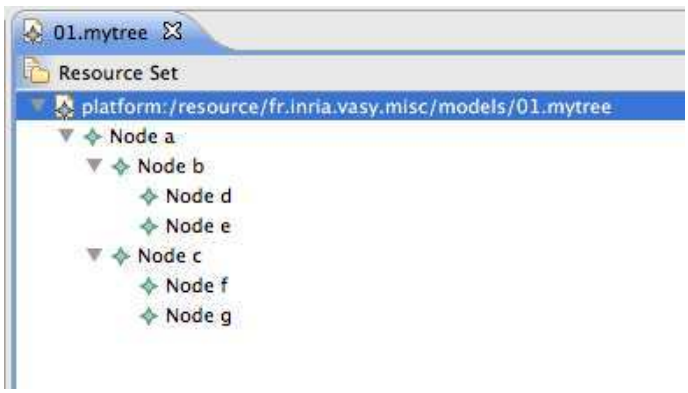

Figure 1.5: Editeur arborescent généré par Emf pour le méta-modèle d'arbre

Afin de permettre aux utilisateurs de créer aisément des éditeurs plus évolués que celui généré automatiquement par Emf pour chaque méta-modèle, Eclipse propose toute une infrastructure logicielle. Elle se présente sous la forme de briques logicielles qui contiennent un squelette d'éditeur et qu'il faut étendre pour obtenir un éditeur fonctionnel. Ces briques sont très complexes et font interagir des centaines de classes différentes. La construction d'un éditeur est donc une tâche ardue ; c'est pourquoi des outils sont rapidement apparus pour la rendre plus aisée. Deux catégories d'outils se distinguent : ceux pour la création d'éditeurs textuels et ceux pour la création d'éditeurs graphiques. Parmi ces outils, nous citons les premiers apparus dans chaque catégorie :

- Eclipse Imp[10] (Ide *Meta-Tooling Platform*), anciennement connu sous le nom de Safari, est un projet Eclipse financé par Ibm dont le but est de permettre la construction à moindre coût d'éditeurs textuels pour Eclipse à l'aide d'une grammaire Lalr. Imp permet la réutilisation d'un analyseur syntaxique existant, même sous une forme déjà compilée et n'est pas liée à un langage de programmation particulier pour l'encodage de l'arbre de syntaxe abstrait. Malheureusement, l'absence de documentation pour ces fonctionnalités les rend difficiles à mettre en œuvre.

- Eclipse Gef[11] (*Graphical Editing Framework*) est un ensemble de briques logicielles dont le but est de simplifier la création d'éditeurs graphiques pour des langages dont l'arbre de syntaxe abstraite est décrit en Java. Gef encapsule les couches logicielles graphiques d'Eclipse afin de proposer une interface de programmation intuitive pour l'affichage d'objets sous diverses formes facilement modifiables.

[10] `http://www.eclipse.org/imp`
[11] `http://www.eclipse.org/gef/`

Par la suite sont apparus deux outils, XTEXT et GMF, pour la création d'éditeurs textuels et graphiques dans lesquels la syntaxe abstraite du langage considéré est un méta-modèle EMF. L'intérêt de ces deux outils est de pouvoir combiner les éditeurs qu'ils permettent de construire aux transformations de modèles ainsi qu'aux outils de génération de code.

XTEXT[12] est un outil qui permet d'associer une grammaire LL à un méta-modèle EMF dans le but de créer un éditeur textuel pour ce méta-modèle. En réalité, XTEXT est aussi un générateur de code et un environnement de transformation, mais il est beaucoup moins connu pour ses aspects. XTEXT est financé et développé par la société Itemis[13] dont l'une des principales activités est le support professionnel pour XTEXT. D'abord distribué au sein du projet *openArchitectureWare*[14], XTEXT est devenu un projet Eclipse à part entière en septembre 2009.

XTEXT est relativement facile à utiliser et nous a permis de créer un éditeur textuel pour notre méta-modèle de structure d'arbre. A partir de ce méta-modèle, XTEXT a généré une grammaire LL, dans un langage de grammaire propre à XTEXT, que nous avons légèrement simplifiée (suppression d'une règle intermédiaire inutile). Ensuite nous avons lancé la génération et la compilation de l'éditeur dont la figure 1.6 montre une capture d'écran. Les éditeurs générés par XTEXT peuvent comprendre des fonctionnalités usuelles comme la coloration syntaxique, la nagivation du modèle en cours d'édition, la complétion de code, des vues d'ensemble... Il suffit de modifier les fichiers correspondants.

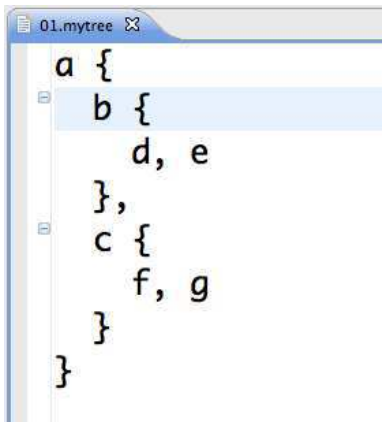

Figure 1.6: Editeur textuel créé avec XTEXT pour le méta-modèle d'arbre

Eclipse GMF[15] est un outil construit sur GEF, qui permet de définir, très rapidement et sans manipuler de code Java, des éditeurs graphiques pour des méta-modèles EMF. GMF est l'un des nombreux outils Eclipse financés par IBM. Il donne une interface graphique à GEF, c'est-à-dire que tout se fait visuellement avec le minimum d'édition de code Java. La création d'un éditeur pour notre méta-modèle de structures d'arbre a été étonnamment facile et a consisté en une succession de clics sur le bouton "Suivant", interrompue seulement par le choix des représentations graphiques des éléments du méta-modèle. La figure 1.7 montre une capture d'écran de cet éditeur.

La seule difficulté à laquelle nous avons été confrontés, lors de l'utilisation de GMF, était liée à la façon dont les éditeurs générés automatiquement par GMF instancient les nouveaux objets. Dans ces éditeurs, chaque objet est créé séparément avant que ne soient ajoutées les références entre objets. De plus, il doit y avoir, durant l'édition graphique d'un modèle, une correspondance permanente entre

[12] http://www.eclipse.org/xtext
[13] http://www.itemis.com
[14] http://www.openarchitectureware.org
[15] http://www.eclipse.org/gmf

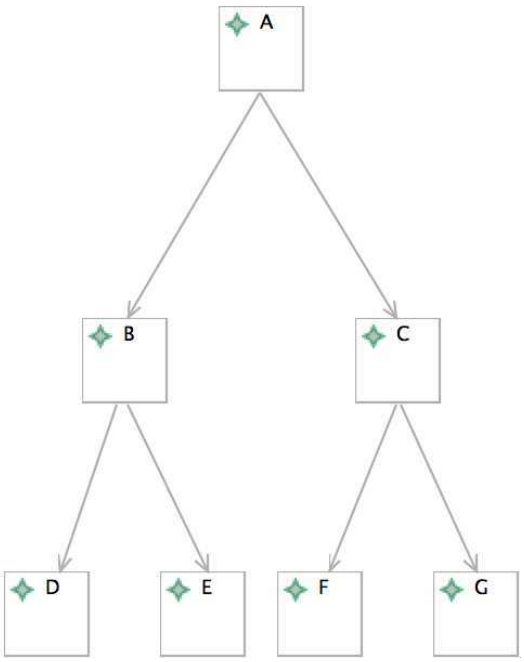

Figure 1.7: Editeur graphique créé avec GMF pour le méta-modèle d'arbre

ce qui est affiché dans l'éditeur et la structure définie par le méta-modèle (auquel le modèle se conforme). Il faut donc s'assurer que les classes du méta-modèle que l'on souhaite éditer graphiquement ne "contiennent" pas d'autres classes (références dont l'attribut booléen "containment" est vrai, cf. section 1.2.1), car, dans ce cas, les objets des classes "contenues" ne peuvent être créés séparément (puisqu'ils n'existent que dans le cadre de la référence qui les lie à l'objet qui les contient).

Dans le cas de notre méta-modèle de structure d'arbre, chaque nœud contient ses nœuds fils, c'est-à-dire qu'un nœud fils est créé en même temps que la relation qui le lie à son nœud père et est détruit lors que cette relation est détruite. En termes d'édition graphique, cela veut dire qu'il est possible de créer le nœud racine, mais pas ses nœuds fils, car il faut d'abord créer la relation de référence qui lie un nœudsà son père. Il n'est pas non plus possible de créer la référence entre un nœud fils et son nœud père, car GMF ne permet de lier que des objets déjà créés dans l'éditeur. Ce comportement peut être changé, mais il est pour cela nécessaire de modifier manuellement le code Java généré par GMF pour l'éditeur.

Pour résoudre ce problème, nous avons procédé à deux changements dans notre méta-modèle de structure d'arbre. Nous avons ajouté une nouvelle classe nommée Resource et nous avons changé, de "`true`" à "`false`", la valeur de l'attribut booléen containment de la référence qui lie un nœud à ses nœuds fils. Dans un modèle se conformant au méta-modèle ainsi obtenu, une instance de la classe Resource (que nous choisissons de ne pas afficher explicitement dans l'éditeur) contient tous les nœuds de l'arbre. Ces nœuds peuvent alors être créés séparément et liés par la suite. Le nouveau méta-modèle obtenu, dont on ne se sert que pour l'édition graphique est illustré par la figure 1.8. Le petit losange noir à la racine de la relation entre les classes Resource et Node indique qu'un objet de la classe Resource contient les objets de la class Node.

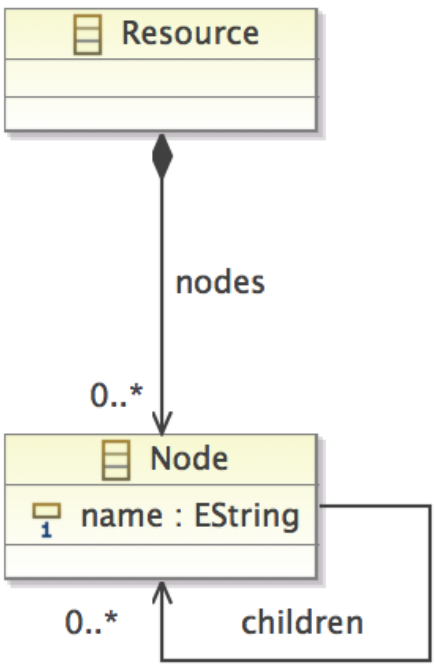

Figure 1.8: Méta-modèle d'arbre modifié pour la construction de l'éditeur graphique

1.2.3 Transformation de modèles vers modèles

Une des caractéristiques principales de l'ingénierie basée sur les modèles est de tout considérer comme un modèle. La norme QVT [OMG07a], que nous avons introduite à la section 1.1, suit ce principe et les transformations de modèles n'échappent pas à cette règle.

Dans la norme QVT [OMG07a], une transformation M_t entre modèles définit, étant donnés deux méta-modèles MM_a et MM_b, comment générer un modèle M_b conforme à MM_b à partir d'un modèle M_a conforme à MM_a. Toute transformation M_t est vue comme un modèle (c'est-à-dire, un programme) qui se conforme à (est écrit dans) un méta-modèle (c'est-à-dire un DSL) MM_t. A leur tour, les méta-modèles MM_a, MM_b et MM_t doivent se conformer à un méta-méta-modèle MMM (MOF dans le cas de QVT) comme illustré par la figure 1.9. L'intérêt d'un tel système est de permettre de transformer des transformations. Une transformation de modèles écrite dans un langage de programmation classique ne s'inscrit pas normalement dans l'approche IDM. Toutefois, ce problème se résoud par la définition d'un méta-modèle pour capturer la syntaxe abstraite du langage considéré, comme c'est le cas pour le langage Java pour lequel EMF fournit un méta-modèle.

Dans cette section, nous présentons en détail trois solutions différentes de transformation de modèles présentes dans Eclipse, qui reprennent les principes de la norme QVT (mais sans reprendre forcément les langages de transformations qu'elle propose) et qui sont adaptées à différentes complexités de transformation :

- ATL [BBDV03] (ATLAS *Transformation Language*) est un langage très proche de QVT adapté à des transformations simples ;
- Kermeta[16] [DFFV06] est un langage dédié à la manipulation de méta-modèles, qui permet

[16] `http://www.kermeta.org`

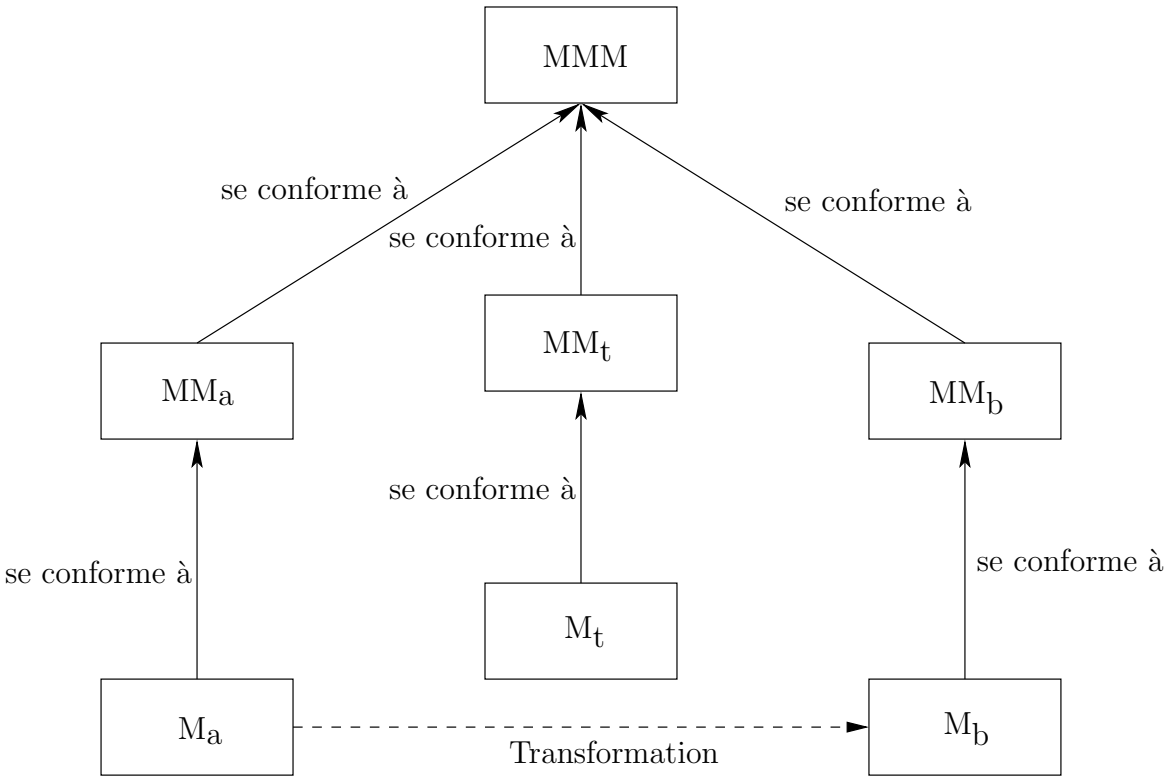

Figure 1.9: Schéma des transformations de modèles dans l'approche QVT

d'écrire des transformations moyennement complexes ;

- Java n'est pas un langage de transformation, mais il est possible d'y recourir pour écrire des transformations très complexes relevant de la compilation.

Il existe deux autres solutions fonctionnelles de transformation de modèles dans Eclipse : Smart QVT[17], une implémentation du langage *Operational* de la norme QVT et VIATRA [VB07], un environnement pour la transformation et la vérification de modèles qui repose sur des techniques de manipulations de graphes (les modèles étant considérés, dans VIATRA, comme des graphes). Nous ne détaillons pas ces deux langages dans cette section car il s'agit ici de présenter des langages de transformation s'appliquant à trois différents niveaux de complexité. Or, Smart QVT et VIATRA s'appliquent aux mêmes niveaux de complexité qu'ATL et Kermeta respectivement.

1.2.4 Transformations entre modèles avec ATL

ATL est l'approche officielle pour les transformations de modèles dans Eclipse. Il s'agit d'un langage de transformation créé par l'équipe ATLAS de l'INRIA pour répondre à l'appel à propositions de l'OMG qui a lancé le processus d'élaboration de la norme QVT. Les différentes réponses reçues ont convergé vers un document unique, la norme QVT finale. Les auteurs d'ATL n'ont pas accepté tous les compromis faits dans le cadre de cette évolution et c'est pourquoi ATL n'a pas convergé vers le

[17] `http://smartqvt.elibel.tm.fr`

même résultat que QVT. Les différences entre ATL et QVT ont fait l'objet d'une étude [JK06] par les auteurs du langage ATL.

ATL représente les modèles d'entrée et de sortie par leurs arbres de syntaxe abstraite respectifs. Les transformations ATL sont écrites dans un langage textuel qui présente un système de règles dont la syntaxe abstraite est définie par un méta-modèle écrit en ECORE. ATL propose deux types de règles pour transformer un modèle :

- les *matched rules*, avec une syntaxe déclarative, correspondent aux règles du langage *Relational* de QVT, filtrent un nœud du modèle d'entrée et créent un élément correspondant dans le modèle de sortie, et
- les *called rules*, avec une syntaxe impérative, correspondent aux règles du langage *Operational* de QVT et sont appelées explicitement.

En plus des règles, ATL permet d'écrire des requêtes (*queries*) vers les nœuds du modèle et d'enrichir (par des *helpers*) les classes des nœuds des modèles à transformer.

ATL est parfaitement adapté lorsque les structures des modèles source et cible sont proches et que la transformation ne demande pas de parcours complexes. Pour effectuer une comparaison, ATL est à la transformation de modèles ce que les outils UNIX `sed`, `grep`, `cut`... sont à la transformation de texte.

Afin d'illustrer les limitations d'ATL, nous avons écrit une transformation qui prend en entrée un modèle (`IN`) conforme à notre méta-modèle (`MyTree`) de structure d'arbre et qui produit un modèle (`OUT`) conforme à notre méta-modèle (`MyList`) de structure de listes. Ces méta-modèles ont été définis à la section 1.2.1. La principale contrainte d'ATL (et de QVT) est que chaque règle doit créer exactement (ni plus, ni moins) un nœud du modèle de sortie, ce qui empêche de réaliser un parcours d'arbre classique, en traitant tous les fils d'un même nœud dans la même règle. Dans notre transformation, cela complique l'écriture en obligeant à ajouter une règle auxiliaire `SeqTree2List`.

```
module Tree2List;
create OUT : MyList from IN : MyTree;

rule Tree2List {
  from
    t : MyTree!Node (t.isRoot ())
  to
    l : MyList!Elt  (
      name <- t.name,
      next <- thisModule.SeqTree2List (t.toSeq ())
    )
}
```

```
rule SeqTree2List (seq : Sequence (MyTree!Node)) {
  to
    l : MyList!Elt (
      name <- seq.first ().name
    )
  do {
    if (seq.size() <> 1) {
      l.next <- thisModule.SeqTree2List (seq.excluding (seq.first ()));
    }
    l;
  }
}
```

La première règle, `Tree2List`, est une *matched rule* qui n'est exécutée que pour le noeud racine de l'arbre (le seul qui remplit la condition `t.isRoot ()`). Elle crée le premier élément de la liste, puis, appelle une méthode `t.toSeq ()` qui est définie, en code impératif, dans une bibliothèque ATL annexe, et qui construit une séquence de noeuds de l'arbre (en effectuant un parcours en profondeur préfixe de gauche à droite de l'arbre). Ensuite, cette séquence est passée à `SeqTree2List`. Il est important de noter que les méta-modèles que nous avons définis ne permettent pas de définir des arbres et des listes vides. Pour cette raison, ce cas particulier n'est pas géré.

La seconde règle, `SeqTree2List`, est une *called rule* qui crée une liste par un parcours récursif de la séquence de nœuds passée en argument. Chaque nœud ainsi rencontré est transformé en un élément et ajouté à la liste à renvoyer.

Les fonctions `isRoot ()` et `toSeq ()` appartiennent à une bibliothèque ATL annexe et sont définies de la manière suivante :

```
helper context MyTree!Node def : isRoot () : Boolean =
  self.refImmediateComposite ().oclIsUndefined ();

helper context MyTree!Node def : toSeq () : Sequence (MyTree!Node) =
  if self.isRoot () then
    self.nodes->iterate (
      n ; res : Sequence (MyTree!Node) = Sequence {} | res.union (n.toSeq ())
    )
  else
    self.nodes->iterate (
      n ; res : Sequence (MyTree!Node) = Sequence {self} | res.union (n.toSeq ())
    )
  endif;
```

1.2.5 Transformations entre modèles avec Kermeta

Kermeta est un environnement de méta-modélisation surtout utilisé pour écrire des transformations entre modèles. Cet outil est développé par l'équipe TRISKELL[18] de l'IRISA. Le langage Kermeta est un langage de programmation impératif, à objets et statiquement typé. Il se conforme aux méta-méta-modèles MOF et ECORE. Sa syntaxe concrète est très proche de celle de Java vers lequel le langage Kermeta peut être "compilé".

Kermeta supporte les concepts classiques des langages à objets comme la définition de méthodes et

[18] `http://www.irisa.fr/triskell`

d'attributs de classe, l'héritage simple, l'héritage multiple, la surcharge et le masquage de méthodes ou encore les paramètres *template*. La particularité de Kermeta repose sur sa faculté à pouvoir importer des modèles et les manipuler. Pour manipuler un modèle M_a conforme à un méta-modèle MM_a, il suffit d'importer le méta-modèle MM_a à l'aide d'une directive prévue à cet effet. Les classes du méta-modèle sont alors traduites vers des classes de Kermeta et lorsque le modèle M_a est chargé, il est simplement instancié en objet Kermeta.

Kermeta se prête bien à des transformations de tout type, mais qui ne relèvent pas d'un réel exercice de compilation. Kermeta est à la transformation de modèles ce que sont à la transformation de texte les langages de *script* (comme Ruby, Perl, Python...).

La transformation d'un modèle conforme à notre méta-modèle de structure d'arbre vers un modèle conforme à notre méta-modèle de structure de liste est très rapide à écrire en Kermeta. Il suffit de commencer par importer les définitions des méta-modèles source et destination. Il faut ensuite charger le modèle à transformer à l'aide de directives Kermeta qui sont clairement documentées dans le manuel de référence, mais que nous omettons ici pour nous concentrer sur l'écriture de la transformation. Ensuite, la transformation est appelée sur le modèle. Enfin le résultat de la transformation est sauvegardée dans un fichier liste qui prend automatiquement le format Xmi de notre méta-modèle de liste. Le code de la transformation est le suivant (`result` est une variable spéciale à laquelle doit être affecté la valeur de retour de la fonction) :

```
operation tree2list (n : MyTree::Node) : MyList::Elt is do
  var l : MyList::Elt init MyList::Elt.new
  l.name := n.name
  n.nodes.each { c |
    var m : MyList::Elt init tree2list (c)
    l := merge (l, m)
  }
  result := l
end
```

Cette fonction construit, pour un nœud t, la liste ordonnée des nœuds traversés lors d'un parcours en profondeur (parcours préfixe de gauche à droite) du sous-arbre dont t est la racine. Pour chaque nœud de t, la fonction est rappelée récursivement afin de construire la sous-liste ordonnée correspondant à ce nœud, cette sous-liste est ensuite concaténée à la liste principale.

Le code de la fonction de concaténation de liste est tout aussi intuitif :

```
operation merge (left : MyList::Elt, right: MyList::Elt) : MyList::Elt is do
  if left == void then
    result := right
  else
    var list : MyList::Elt init MyList::Elt.new
    list.name := left.name
    list.next := merge (left.next, right)
    result := list
  end
end
```

1.2.6 Transformations de modèles avec Java

Le langage Java n'est pas particulièrement adapté à l'écriture de transformations entre modèles. Pourtant, lorsque la transformation devient si difficile et complexe qu'elle tient plus d'un véritable

algorithme de compilation que d'une simple transformation syntaxique, l'utilisation du langage Java, des bibliothèques et des outils qui lui sont associés, devient alors justifiée. Le principal obstacle de Java à l'écriture de transformations simples est le chargement et la sauvegarde des modèles d'entrée et de sortie. Dans l'exemple de transformation d'arbre en liste, le code de ces deux opérations est deux fois plus long que le code de la transformation. A titre d'exemple, voila comment s'écrit la transformation :

```
private static Elt tree2list (Node node, Elt elt, MyListFactory factory) {
  elt.setName(node.getName());
  Elt current = elt;
  Iterator<Node> iterator = node.getChildren().iterator();
  while (iterator.hasNext()) {
    Elt newElt = factory.createElt();
    current.setNext(newElt);
    current = tree2list (iterator.next(), newElt, factory);
  }
  return current;
}
```

Cette transformation utilise les classes Java automatiquement définie par Emf pour le méta-modèle de structure d'arbre et le méta-modèle de structure de liste. Elle prend en entrée trois arguments :

- `node`, le nœud à traiter,
- `elt`, l'élément de la liste de sortie correspondant à `node`, et
- `factory`, une instance de la classe `MyListFactory` qui possède la méthode `createElt`. Invoquer cette méthode est le seul moyen de créer de nouvelles instances de la classe `Elt`.

Pour chaque nœud de `node`, une liste contenant les nœuds du sous-arbre dont `node` est la racine est construite. Chaque nouvelle liste ainsi obtenue est ajoutée à la liste courante qui correspond aux nœuds de `node` déja visités.

1.2.7 Génération de code (transformation de modèles vers texte)

Même si, dans une certaine mesure, les éditeurs textuels comme Xtext génèrent du texte pour un modèle, il s'agit d'une représentation textuelle de ce modèle dont le but est d'être lisible et modifiable par un utilisateur. Par "génération de code", nous entendons transformation d'un modèle en code exécutable. Cette transformation est désignée sous le terme M2T (*Model To Text*). Le consortium Omg a édité une norme, Mtl [OMG08a] (Mof *Model-To-Text Transformation Language*), qui définit un langage pour effectuer ce type de transformations. Eclipse propose deux outils[19] pour réaliser les transformations M2T :

- Acceleo[20] a été développé par la société française Obeo qui l'a distribué librement sur le site `http://www.acceleo.org`. Depuis juin 2009, Acceleo est devenu un projet Eclipse. Le langage de transformation d'Acceleo est une implémentation de la norme Mtl de l'Omg.
- Xpand[21], comme Xtext, faisait à l'origine partie du projet *openArchitectureWare*[22] avant de devenir un projet Eclipseen septembre 2009. Le langage de transformation Xpand, contraire-

[19] `http://www.eclipse.org/modeling/m2t`
[20] `http://www.eclipse.org/acceleo`
[21] `http://www.eclipse.org/modeling/m2t/?project=xpand`
[22] `http://www.openarchitectureware.org/`

ment au langage de transformation d'Acceleo, a une syntaxe qui lui est propre, mais qui reste toutefois proche de la norme Mtl.

Afin d'illustrer le fonctionnement de ces deux outils, nous avons décidé de créer une transformation qui génère une expression C++ à partir d'un modèle se conformant à notre méta-modèle de structure d'arbre. Cette expression renvoie une valeur de type `Node`, une classe C++ qui représente la classe de même nom dans notre méta-modèle. Nous avons défini pour cette classe :

- la fonction `create_node`, qui prend comme argument une chaîne de caractères `s` et renvoie un nouveau nœud de nom `s` et
- la méthode `add_child`, qui ajoute le nœud passé en argument à la liste des fils du nœud auquel elle est appliquée.

Pour l'exemple d'arbre de la figure 1.5, l'expression générée doit être la suivante :

```
create_node ("a")
  .add_child (
    create_node ("b")
      .add_child (
         create_node ("d")
      )
      .add_child (
         create_node ("e")
      )
  )
  .add_child (
    create_node ("c")
      .add_child (
        create_node ("f")
      )
      .add_child (
        create_node ("g")
      )
  )
```

En Acceleo, l'expression C++ est générée à partir d'un modèle d'arbre comme suit :

```
[template public Tree2c(n : Node) ? (e.eContainer().oclIsUndefined())]
  [file (n.name.concat('.cc'), false, 'UTF-8')]
    [generateNode(n)/]
  [/file]
[/template]

[template public generateNode(n : Node)]
  create_node ("[n.name/]")
  [for (c : Node | n.children)]
    .add_child (
      [generateNode(c)/]
    )
  [/for]
[/template]
```

La notation `for (c : Node | n.children)` désigne la boucle dans laquelle la variable `c` itère sur les nœuds fils de `n`.

En Xpand, la même transformation s'écrit ainsi :

```
<<DEFINE Tree2c FOR MyTree::Node>> -- Argument anonyme
  <<FILE name + ".cc">>
    <<EXPAND generateNode>> -- Appel de generateNode sur l'argument anonyme
  <<ENDFILE>>
<<ENDDEFINE>>

<<DEFINE generateNode FOR MyTree::Node>>
  create_node("<<name>>")
  <<FOREACH children AS n >>
    .add_child (<<EXPAND generateNode FOR n >>)
  <<ENDFOREACH>>
<<ENDDEFINE>>
```

Les deux langages sont similaires et fonctionnent sur un système de règles, un peu comme Atl. Chaque règle produit un fragment du texte dans lequel peuvent être insérées, à l'aide de balises, des chaînes de caractères résultant de l'évaluation d'expressions. Dans les deux examples, la première règle fait office de point d'entrée de la transformation et ouvre le fichier qui va contenir le résultat de la transformation. Pour Acceleo, une garde booléenne (`n.eContainer().oclIsUndefined()`) restreint l'application de cette règle au nœud racine uniquement (en Xpand, cette restriction est implicite). Ensuite, cette première règle appelle la seconde qui effectue un parcours récursif de l'arbre et génère le code correspondant.

La principale différence entre les deux outils tient surtout à la façon dont sont lancées les transformations. Acceleo privilégie l'interface graphique d'Eclipse pour l'exécution des transformations : il suffit de spécifier le modèle d'entrée et le répertoire cible dans lequel le (ou les) fichier(s) généré(s) seront placés. Xpand préfère spécifier l'exécution des transformations au moyen d'un langage de *script* à syntaxe Xml (à la façon d'un *Makefile*) dont la syntaxe est malheureusement obscure et mal documentée.

1.3 Le projet Topcased

Cette thèse s'inscrit en partie dans le cadre du projet Topcased[23] (2006-2011), un projet du pôle de compétitivité mondial Aese (*Aerospace Valley*). Ce projet regroupe :

- des grandes entrepries des industries aéronautique, spatiale et de l'automobile comme Airbus, Eads Astrium, Rockwell & Collins, Siemens Vdo et Thales2,
- des Ssiis comme Atos Origin, Cap Gemini et Cs,
- des entreprises éditrices de logiciels comme Obeo, Anyware et Adacore,
- des universités et grandes écoles comme l'Université Paul Sabatier, l'Insa de Toulouse, l'Ensieta et l'Enseeiht,
- des laboratoires de recherche comme le Cea, le Cnes, l'Inria, et le Laas-Cnrs.

[23] `http://www.topcased.org`

TOPCASED vise à faire progresser les méthodes de développement des systèmes critiques des industries aéronautique, spatiale et de l'automobile. Pour cela :

- TOPCASED fournit un environnement de travail pour le développement et la vérification des logiciels critiques embarqués. Ces logiciels doivent fonctionner sans erreur, car une défaillance peut avoir des coûts humains et financiers importants. C'est pourquoi il est impératif de garantir le bon comportement de ces systèmes et, donc, d'avoir à disposition des outils adaptés pour les développer.

- TOPCASED se focalise sur les langages de modélisation pour le logiciel embarqué, tels que : AADL [SAE09], SYSML [OMG08b], SAM [CGT08], SILDEX [Eur05], UML 2 [OMG07b]...

- TOPCASED implémente l'approche IDM. La représentation des systèmes critiques sous la forme de modèles permet de réutiliser les langages de transformation de modèles pour réaliser des passerelles entre différents langages de modélisation et formalismes intermédiaires.

- TOPCASED est bâti sur l'environnement Eclipse afin de bénéficier de l'infrastructure logicielle que met à disposition cet environnement pour implémenter l'approche IDM : EMF, XTEXT, GMF, ATL... De nombreuses contributions ont été faites à différents projets d'Eclipse dans le cadre de TOPCASED : EMF, GMF, ATL, Kermeta, ACCELEO ainsi que tous les projets relatifs à UML 2 (l'éditeur Papyrus[24], par exemple).

- TOPCASED a choisi un modèle de développement *open source* pour permettre une collaboration plus facile entre les différents partenaires, ainsi que pour en attirer de nouveaux. Ce type de développement favorise les contributions extérieures spontanées : rapports et corrections de bogues, propositions de fonctionnalités... Autour de ce noyau libre et ouvert, viennent se connecter des outils (académique ou commerciaux) de simulation et de vérification. Enfin, le modèle *open source* assure une relative pérennité des outils développés.

- TOPCASED propose un vaste panel de fonctionnalités. Il fournit des éditeurs, textuels et graphiques, pour différents langages de modélisation. Ensuite, TOPCASED intègre des outils de génération de code, tels que Gene-Auto [TNP+], pour la production de code exécutable certifié à partir des systèmes spécifiés. Enfin, TOPCASED connecte, au moyen du format intermédiaire FIACRE [BBF+08], les langages de modélisation aux formalismes d'entrée de divers outils existants (tels que CADP [GLMS07], TINA [BRV04], OBP [DPC+09] (*Observer-Based Prover*), Sigali [BMR06]...) afin de simuler et de vérifier formellement le fonctionnement des systèmes spécifiés. Cette dernière fonctionnalité est primordiale puisqu'elle rend possible la vérification d'un même système critique par le biais de plusieurs outils de vérification, sans qu'il soit nécessaire de spécifier explicitement le système à l'aide des formalismes d'entrée des différents outils de vérification.

Le projet TOPCASED est en cours d'achèvement. Les outils qu'il a produits sont d'ores et déjà utilisés, non seulement par les partenaires du projet, mais aussi par des acteurs extérieurs. Ainsi, le programme des journées TOPCASED[25], (2 au 4 février 2011) indique des utilisations des industriels (Deutsche Bahn et Altstom) et des académiques (Université de Düsseldorf, Institut Fraunhofer, Université de Paris Est Créteil et Université de Franche Comté) extérieurs au projet.

[24] `http://www.eclipse.org/papyrus`
[25] `http://gforge.enseeiht.fr/docman/view.php/52/4122/Program_V6.pdf`

Chapitre 2

CADP et LOTOS NT

2.1 Méthodes formelles

Les méthodes formelles sont des méthodes à caractère mathématique pour la modélisation et la vérification de systèmes informatiques (logiciels ou matériels). La modélisation de systèmes est donc un objectif commun aux méthodes formelles et à l'ingénierie dirigée par les modèles. Tandis que l'approche IDM insiste sur la syntaxe abstraite des langages de modélisation et les techniques de transformation, les méthodes formelles se focalisent sur la sémantique de ces langages. Cette complémentarité entre méthodes formelles et approche IDM a été mise en évidence dans le projet TOPCASED (cf. section 1.3).

L'intérêt des méthodes formelles réside dans les langages de modélisation formelle, mais aussi dans les outils qui leur sont associés. Ces outils se déclinent selon plusieurs écoles dont nous présentons les principales :

- L'analyse statique consiste à analyser le code source d'un programme en tenant compte de la sémantique du langage dans lequel ce programme est écrit. L'interprétation abstraite [CC77] est une technique d'analyse statique qui traite un programme en raisonnant sur les constructions de langage utilisées par son code source, mais en effectuant les calculs de manière symbolique. Cette technique s'applique à la compilation de programmes pour vérifier la correction de certaines optimisations, ainsi qu'à la certification de programmes, notamment séquentiels. Parmi les outils d'analyse statique, nous pouvons citer ASTRÉE [CCF+05] commercialisé par ABSINT[26] et Polyspace Verifier commercialisé par MATHWORKS[27].

- La preuve de théorèmes consiste à formuler le fait qu'un programme satisfait une propriété à l'aide d'un système de règles qu'un outil informatique va ensuite résoudre. Les outils qui calculent des preuves de théorèmes sont appelés des démonstrateurs informatiques ou des assistants de preuve. Ce dernier nom vient du fait que la complexité de la tâche rend souvent nécessaire une interaction avec un humain, qui va alors guider la résolution. Parmi les assistants de preuve, nous pouvons citer COQ [BBC+08] qui a servi à valider [Gon08] la preuve du théroème des quatre couleurs [AH76], et ISABELLE [Pau89] qui a validé le théorème de complétude de Gödel [Mar04].

- La vérification "énumérative" consiste à construire un modèle explicite représentant toutes les

[26] http://www.absint.com/astree/
[27] http://www.mathworks.com/products/polyspace/

exécutions possibles du système à vérifier. Le plus souvent, il s'agit d'un système concurrent, c'est-à-dire contenant du parallélisme. Il existe trois grandes techniques pour vérifier, par cette approche, le bon fonctionnement du système : par évaluation de formules de logique temporelle sur le modèle (*model checking*), par vérification d'équivalence avec un autre modèle (*equivalence checking*) ou par vérification visuelle du modèle obtenu (*visual checking*), cette dernière n'étant pas une technique formelle à proprement parler.

Dans ce document, nous nous concentrons sur la vérification énumérative. De nombreux outils de vérification énumérative existent[28], parmi lesquels SPIN [Hol04], SMV [CCGR00], NUSMV [CCGR00], UPPAAL [PL00]... Nous considérons le cas de CADP [FGK+96, GJM+97, GLM02b, GLMS07, GLMS11], développé par l'équipe VASY du centre de recherche INRIA de Grenoble Rhône-Alpes dans laquelle s'est principalement déroulée cette thèse.

Dans la suite de ce chapitre nous présentons CADP (à la section 2.2) avant d'aborder la syntaxe et la sémantique de LOTOS NT (à la section 2.3), son plus récent formalisme d'entrée.

2.2 CADP

CADP[29] [GLMS11] (*Construction and Analysis of Distributed Processes*) est une boîte à outils de vérification formelle. CADP permet la modélisation et l'analyse des systèmes distribués et des protocoles de communication. Cette boîte à outils cible les grands groupes industriels et le monde académique. Elle a été utilisée dans de nombreuses études de cas[30].

2.2.1 Les systèmes de transitions

Le formalisme utilisé dans CADP pour représenter les systèmes à analyser est celui des STEs (*Systèmes de Transitions Etiquetées*). Le comportement d'un système est découpé en un ensemble d'états reliés par des transitions. Chaque transition porte une étiquette que l'on appelle "action". L'évolution du système correspond au passage d'un état à un autre par le franchissement d'une transition dont l'action est alors déclenchée. Des systèmes ainsi modélisés peuvent être composés par synchronisation sur des actions communes afin de créer un modèle pour un système plus complexe. Chaque système prenant part à cette composition évolue en parallèle des autres, indépendamment (les actions des différents systèmes en interaction sont entrelacées), sauf pour les synchronisations qui s'effectuent à l'unisson. On parle alors de parallélisme asynchrone. Les STEs permettent aussi de modéliser des systèmes dont le comportement n'est pas déterministe, c'est-à-dire qu'à un moment donné, un système peut choisir une transition parmi plusieurs sans que les systèmes avec lesquels il est composé ne puissent influencer ce choix. De telles transitions comportent une action invisible (notée "i" ou "τ") et sont couramment appelées *transitions tau*.

2.2.2 Algèbres de processus

Pour la modélisation des systèmes, CADP utilise les algèbres de processus qui permettent la définition de processus parallèles qui peuvent communiquer et s'échanger des données par des synchronisations sur des portes de communication. La sémantique des algèbres de processus peut être exprimée par traduction vers des systèmes de transitions étiquetées. Dans ce cas, les actions des transitions ne

[28] `http://anna.fi.muni.cz/yahoda/`
[29] `http://vasy.inria.fr/cadp`
[30] `http://vasy.inria.fr/cadp/case-studies/`

représentent que les communications entre les différents processus, les autres évolutions des processus (affectation d'une variable, calcul d'une expression...) étant représentées par des transitions *tau*.

Initialement, CADP utilisait LOTOS [ISO89] (*Language Of Temporal Ordering Specification*). Les version récentes de CADP utilisent LOTOS NT [CCG+10] (cf. section 2.3), une version simplifiée de la norme internationale E-LOTOS [ISO01], plutôt qu'à LOTOS. Un traducteur automatique qui transforme des spécifications LOTOS NT en spécifications LOTOS est développé au sein de la boîte à outils CADP. Les spécifications LOTOS générées peuvent être, à leur tour, traduites vers des systèmes de transitions étiquetées, puis analysées avec CADP.

2.2.3 Techniques de vérification

Le formalisme des systèmes de transitions étiquetées permet l'analyse des systèmes par plusieurs techniques, toutes implémentées dans CADP, notamment :

- Le *model checking* [CES86] consiste à évaluer une formule de logique temporelle (du temps linéaire ou du temps arborescent) sur un système de transitions étiquetées dans le but de vérifier que le système satisfait la propriété représentée par la formule ;
- La vérification par équivalence (*equivalence checking*) consiste à vérifier que deux systèmes de transitions étiquetées sont égaux, modulo une certaine relation d'équivalence ;
- La simulation consiste à analyser des traces d'exécutions extraites aléatoirement du système de transitions étiquetées.

2.2.4 Les outils

CADP comporte de nombreux outils et bibliothèques permettant de manipuler les systèmes de transitions étiquetées, les spécifications LOTOS et LOTOS NT, et les formules de logique temporelle :

- CÆSAR.ADT est un compilateur qui transforme les déclarations de types algébriques LOTOS en déclarations de fonctions et de types C.
- CÆSAR est un compilateur qui transforme les processus LOTOS, soit en C, soit en STE pour les vérifier formellement.
- OPEN/CÆSAR est un environnement logiciel générique qui sert au développement d'outils pour explorer les STEs à la volée. Il s'agit d'un composant central de CADP qui permet de connecter les outils basés sur les langages (algèbres de processus) aux outils basés sur les STEs. Un nombre important d'outils ont été développés grâce à cet environnement, parmi lesquels : EVALUATOR (le *model checker* de CADP), BISIMULATOR (pour la vérification d'équivalence entre STEs) ou encore EXECUTOR (pour la génération de séquences de transitions aléatoires dans un STE).
- BCG (*Binary Coded Graphs*) est à la fois un format de fichier pour la représentation de grands STEs et un environnement logiciel pour la manipulation de ce format. De nombreux outils de CADP reposent sur cet environnement, parmi lesquels : BCG_DRAW (pour l'affichage des STEs), BCG_EDIT (pour l'édition graphique de STEs) et BCG_MIN (pour la réduction de STEs selon diverses relations d'équivalence).
- La connexion entre les modèles explicites (comme les STEs) et les modèles implicites (explorés à la volée) est assurée par les compilateurs suivants, qui se conforment à l'environnement OPEN/CÆSAR :

- Cæsar.Open pour les modèles exprimés comme des spécifications Lotos,
- Bcg_Open pour les modèles représentés par des graphes Bcg,
- Exp.Open pour les modèles exprimés comme des automates communicants et
- Seq.Open pour les modèles représentés par des ensembles de séquences d'exécution.

Tous les outils de Cadp sont distribués pour 13 architectures informatiques différentes[31] (processeurs et systèmes d'exploitation).

2.3 Présentation de LOTOS NT

Entre 1993 et 2001, l'Iso a entrepris de réviser la norme Lotos. Ce travail de révision a abouti à la norme E-Lotos [ISO01]. L'équipe Vasy a défini une version simplifiée appelée Lotos NT. Ce langage est une algèbre de processus de seconde génération [Gar03, Gar08], qui combine le parallélisme asynchrone des algèbres de processus avec les constructions usuelles des langages de programmation, dans un style d'écriture à la fois impératif et fonctionnel. La syntaxe, la sémantique statique et la sémantique dynamique de Lotos NT ont été définies par Mihaela Sighireanu [Sig99].

Deux compilateurs pour Lotos NT ont été réalisés au sein de l'équipe Vasy :

- Traian [SCG+08] est un outil qui traduit les types de données et les fonctions de Lotos NT vers le langage C. La vocation première de Traian était d'être un compilateur complet pour Lotos NT, au même titre que les compilateurs Cæsar et Cæsar.adt pour Lotos. Cependant, la compilation des processus n'a pas été implémentée. L'équipe Vasy utilise actuellement cet outil pour le développement de compilateurs [GLM02a], ce qui confirme l'expressivité de Lotos NT.

- Lnt2Lotos [CCG+10] est un traducteur qui produit une spécification Lotos à partir d'une spécification Lotos NT ; cela permet de réutiliser les compilateurs Cæsar et Cæsar.adt pour Lotos afin de produire un Lts correspondant à un processus Lotos NT. Le développement de Lnt2Lotos a débuté en 2005, grâce à un financement de Bull. Ce traducteur a commencé par traduire les types de données, puis les fonctions et a progressivement évolué jusqu'à traiter, à l'ecriture de ce document, un large sous-ensemble de Lotos NT.

Le début de cette thèse a coïncidé avec l'apparition de la traduction des processus dans Lnt2Lotos. Cela nous a permis d'utiliser le langage Lotos NT comme langage d'entrée de Cadp dans les travaux que nous présentons. Ces travaux ont conduit à une cinquantaine de suggestions de rapports de bogues concernant Lnt2Lotos et ont donc contribué à amener l'outil à son niveau de développement et de stabilité actuel, qui rend possible son utilisation.

2.4 Sous-ensemble de LOTOS NT utilisé

Nous présentons dans cette section le sous-ensemble du langage Lotos NT que nous considérons dans ce document. Pour chaque construction Lotos NT utilisée, nous présentons sa syntaxe et décrivons, de façon intuitive, sa sémantique. Les lecteurs intéréssés par la définition complète de Lotos NT et de sa sémantique formelle pourront se référer à [Sig99].

Il convient aussi de préciser que l'évaluation d'une spécification Lotos NT ne peut se faire que si l'on peut prouver statiquement que toutes les variables sont initialisées avant d'être lues.

[31] `http://vasy.inria.fr/cadp/status`

2.4.1 Identificateurs

Nous adoptons les conventions suivantes pour noter les différents identificateurs LOTOS NT :

- M pour les modules,
- T pour les types,
- C pour les constructeurs de types,
- X pour les variables,
- F pour les fonctions,
- L pour les étiquettes de boucles,
- Γ pour les canaux,
- G pour les portes de communication et
- Π pour les processus.

2.4.2 Définitions de modules

Un fichier LOTOS NT contient un module M, c'est-à-dire un ensemble de déclarations de types, fonctions, canaux et processus, et peut importer les déclarations des modules $M_0 ... M_n$.

lnt_file ::= `module` M [`(`M_0`,...,`M_n`)`] `is`
definition$_0$...*definition*$_n$
`end module`

definition ::= *type_definition*
| *function_definition*
| *channel_definition*
| *process_definition*

2.4.3 Définitions de types

Nous considérons deux sortes de types LOTOS NT : les types listes qui contiennent une séquence d'éléments de même type, et les types construits qui sont définis par une suite de constructeurs. Le domaine de valeurs d'un type construit T est l'union des domaines de valeurs des constructeurs de T ; le domaine de valeurs d'un constructeur est le domaine de valeurs du n-uplet formé par ses paramètres.

type_definition ::= `type` T `is`
type_expression
`end type`

type_expression ::= *construction_definition*$_0$...*constructor_definition*$_n$
| `list of` T

constructor_definition ::= C [`(`*constructor_parameters*$_0$`,...,`*constructor_parameters*$_n$`)`]

constructor_parameters ::= X_0`,...,`X_n`:`T

2.4.4 Définitions de fonctions

L'entête d'une définition de fonction LOTOS NT comprend le nom de la fonction, une liste (éventuellement vide) de paramètres et un type de retour optionnel. Un paramètre de fonction peut être passé par valeur (`in`), par résultat (`out`) ou à la fois par valeur et par résultat (`inout`).

function_definition ::= `function` F
`[(`*formal_parameters*$_1$`,...,`*formal_parameters*$_n$`)]` `[:`T`]`
I
`end function`

formal_parameters ::= *parameter_mode* X_0`,...,`X_n`:`T

parameter_mode ::= `[in]` | `out` | `inout`

2.4.5 Instructions

Les instructions I pouvant figurer au sein d'une définition de fonction sont détaillées ci-après :

I ::= `null`
| ...

`null` désigne l'instruction qui ne fait rien.

I ::= ...
| I_1 `;` I_2
| ...

I_1 `;` I_2 désigne l'exécution de l'instruction I_1 suivie de l'exécution de l'instruction I_2.

I ::= ...
| `return` `[`V`]`
| ...

`return` `[`V`]` termine l'exécution d'une fonction ou d'une procédure. Dans le cas d'une fonction, l'expression optionnelle V doit être présente et son type doit être compatible avec le type de retour de la fonction.

I ::= ...
| X `:=` V
| ...

X `:=` V désigne l'affectation du résultat de l'évaluation de V à la variable X.

I ::= ...
| F `(`*actual_parameter*$_1$`,...,`*actual_parameter*$_n$`)`
| ...

actual_parameter ::= V | `?`X | `!?`X

Cette règle de grammaire désigne un appel à une procédure F. Les paramètres de la forme V sont des paramètres d'entrée dont la valeur sera uniquement lisible par la fonction. Les arguments de la forme `?`X sont des paramètres de sortie dont la valeur est calculée à l'intérieur de la fonction. Les arguments de la forme `!?`X sont des paramètres d'entrée/sortie dont la valeur sera à la fois lisible et

modifiable par la fonction.

I ::= ...
| `var` *var_declaration*$_0$ `,` ... `,` *var_declaration*$_n$ `in`
  I
  `end var`
| ...

var_declaration ::= X_0 `,` ... `,` X_n `:` T

L'instruction `var` déclare des variables locales, dont la portée est restreinte à l'instruction I.

I ::= ...
| `case` V `in`
  [`var` *var_declaration*$_0$ `,` ... `,` *var_declaration*$_n$ `in`]
    match_clause$_0$ `->` I_0
  | ...
  | *match_clause*$_n$ `->` I_n
  `end case`
| ...

match_clause ::= P_0 `|` ... `|` P_n | `any`
P ::= X | `any` T | C [`(`P_0 `,` ... `,` P_n`)`]

L'instruction `case` exécute la premiere instruction I_i ($i \in [0...n]$) dont la *match_clause*$_i$ associée filtre l'expression V (*pattern matching*). Les variables déclarées dans *var_declaration*$_0$...*var_declaration*$_n$ doivent être initialisées dans une *match_clause*$_i$ avant d'être utilisées dans l'instruction I_i. Une expression V est filtrée par une *match_clause* de la forme P_0 `|` ... `|` P_n s'il existe un motif P_i qui filtre V. Une expression V est toujours filtrée par une *match_clause* de la forme "`any`". Si X est une variable non initialisée, alors une expression V est filtrée par un motif de la forme "X" si et seulement si le type de X est égal au type du résultat de l'évaluation de V, auquel cas cette valeur est liée à la variable X. Si X est une variable initialisée, alors une expression V est filtrée par un motif de la forme "X" si la valeur de X est égale au résultat de l'évaluation de V. Une expression V satisfait un motif de la forme `any` T si le résultat de l'évaluation de V est de type T. Une expression V satisfait un motif de la forme C [`(`P_0 `,` ... `,` P_n`)`] si le résultat de l'évaluation de V est une instance du constructeur C avec pour paramètres V_0...V_n et chaque expression V_i parmi V_0...V_n satisfait le motif P_i lui correspondant.

I ::= ...
| `if` V_0 `then` I_0
  [`elsif` V_1 `then` I_1
  ...
  `elsif` V_n `then` I_n]
  [`else` I_{n+1}]
  `end if`
| ...

L'instruction `if` effectue le branchement conditionnel usuel. Les clauses `elsif` et `else` sont optionnelles.

```
I ::= ...
    | loop L in
        I
      end loop
    | break L
    | ...
```

La clause `loop` exécute en boucle l'instruction I. Cette boucle est identifée par L, ce qui permet à l'instruction `break` L, lorsqu'elle figure dans l'instruction I, de faire s'interrompre la boucle.

```
I ::= ...
    | while V loop
        I
      end loop
    | ...
```

La clause `while` exécute en boucle l'instruction I tant que le résultat de l'évaluation de l'expression V est vrai.

```
I ::= ...
    | for I0 while V by I1 loop
        I2
      end loop
```

La clause `for` exécute l'instruction I_2 tant que le résultat de l'évaluation de l'expression V est vrai. I_0 est exécutée une seule fois, initialement. I_1 est exécutée automatiquement après chaque exécution de I_2.

2.4.6 Expressions

Les expressions LOTOS NT sont détaillées dans cette section.

```
V ::= X
    | ...
```

"X" désigne une variable X. Utiliser une variable non initialisée dans une expression entraîne la levée d'une erreur à la compilation. La définition de LOTOS NT utilise des contraintes de sémantique statique pour s'assurer que toute variable est affectée avant d'être utilisée.

```
V ::= ...
    | C [(V1,...,Vn)]
    | ...
```

"C [(V_1,...,V_n)]" désigne l'instantiation du constructeur C avec les paramètres V_1...V_n.

```
V ::= ...
    | F [(V1,...,Vn)]
    | ...
```

"F [(V_1,...,V_n)]" désigne l'appel de la fonction F avec les paramètres V_1...V_n.

$$\begin{array}{lcl} V & ::= & ... \\ & | & V_1\ F\ V_2 \\ & | & ... \end{array}$$

"$V_1\ F\ V_2$" désigne l'appel de la fonction infixe F (typiquement, un opérateur prédéfini) avec les paramètres V_1 et V_2.

$$\begin{array}{lcl} V & ::= & ... \\ & | & V\,.\,X \\ & | & ... \end{array}$$

Si le résultat de l'évaluation de V est un terme ayant la forme $C\ (X_1 : V_1, ..., X_n : V_n)$ et que le i-ème paramètre X_i (s'il existe) désigne le même identifiant que X, alors "$V\,.\,X$" renvoie V_i et lève une exception sinon.

$$\begin{array}{lcl} V & ::= & ... \\ & | & V\,.\,\{X_0\texttt{=>}V_0\,\texttt{,}\,...\,\texttt{,}\,X_n\texttt{=>}V_n\} \\ & | & ... \end{array}$$

Si le résultat de l'évaluation de V est un terme ayant la forme $C\ (X'_1 : V'_1, ..., X'_m : V'_m)$ alors "$V\,.\,\{X_0\texttt{=>}V_0\,\texttt{,}\,...\,\texttt{,}\,X_m\texttt{=>}V_m\}$" renvoie une valeur égale à $C\ (X'_1 : V'_1, ..., X'_m : V'_m)$, à l'exception des paramètres $X_0...X_n$ qui, s'ils apparaissent parmi $X'_0...X'_m$, prennent respectivement comme nouvelles valeurs les résultats des évaluations des expressions $V_0...V_n$.

$$\begin{array}{lcl} V & ::= & ... \\ & | & \{V_1\,\texttt{,}\,...\,\texttt{,}\,V_n\} \\ & | & ... \end{array}$$

"$\{V_1\,\texttt{,}\,...\,\texttt{,}\,V_n\}$" est une abbréviation syntaxique qui désigne la liste de valeurs $V_1...V_n$; ces valeurs doivent être du même type.

2.4.7 Définitions de canaux de communication

En LOTOS NT, les processus communiquent par rendez-vous sur des portes de communications. Au cours de ces rendez-vous, des données peuvent être échangées entre les processus. Contrairement à LOTOS où les portes ne sont pas typées, LOTOS NT type les portes de communication au moyen d'un canal de communication, qui définit le domaine des valeurs pouvant être échangées. La déclaration d'un canal de communication Γ est la suivante :

$$\begin{array}{lcl} \textit{channel_definition} & ::= & \texttt{channel}\ \Gamma\ \texttt{is} \\ & & \quad \textit{gate_profile}_0\,\texttt{,} \\ & & \quad ..., \\ & & \quad \textit{gate_profile}_n\,\texttt{,} \\ & & \texttt{end channel} \\ \textit{gate_profile} & ::= & (T_1\,\texttt{,}\,...\,\texttt{,}\,T_n) \end{array}$$

Le canal de communication prédéfini `any` désigne une porte non typée, afin de préserver la compatibilité avec LOTOS.

2.4.8 Définitions de processus

Une définition de processus LOTOS NT comprend le nom du processus, les portes de communication et le canal de communication associé à chacune des portes, les paramètres du processus et son comportement :

process_definition ::= `process` Π [[*gate_declaration*$_0$, ... , *gate_declaration*$_n$]]
[(*formal_parameters*$_1$, ... , *formal_parameters*$_n$)]
B
`end process`

gate_declaration ::= G_0 , ... , G_n : Γ
| G_0 , ... , G_n : `any`

2.4.9 Comportements

Le comportement B qui correspond au corps d'un processus LOTOS NT est très similaire à l'instruction I qui définit le corps d'une fonction LOTOS NT. Les seules différences résident dans l'ajout de constructions spécifiques au parallélisme asynchrone. Nous commençons par présenter (mais sans les détailler) les comportements B pour lesquels il existe une instruction I similaire (cf. section 2.4.5) :

B ::= `null`
| B_1 `;` B_2
| X `:=` V
| `var` *var_declaration*$_0$, ... , *var_declaration*$_n$ `in`
B
`end var`
| `case` V `in`
[`var` *var_declaration*$_0$, ... , *var_declaration*$_n$ `in`]
match_clause$_0$ `->` B_0
| ...
| *match_clause*$_n$ `->` B_n
`end case`
| `if` V_0 `then` B_0
[`elsif` V_1 `then` B_1
...
`elsif` V_n `then` B_n]
[`else` B_{n+1}]
`end if`

```
  | loop L in
      B
    end loop
  | break L
  | while V loop
      B
    end loop
  | for I0 while V by I1 loop
      B2
    end loop
  | ...
```

Nous détaillons à présent les constructions spécifiques aux comportements.

```
B ::= ...
    | stop
    | ...
```

Le comportement "`stop`" exprime le blocage. Contrairement à "`null`" qui est l'élément neutre (à gauche et à droite) pour la composition séquentielle "`;`", "`stop`" est l'élément absorbant à gauche : `stop ; B` = `stop`.

```
B ::= ...
    | X := any T [where V]
    | ...
```

Ce comportement désigne l'affectation à la variable X d'une valeur v choisie de façon non-déterministe parmi les valeurs du type T. Si la clause `where` est présente, alors v est choisie parmi les valeurs du domaine de T pour lesquelles le résultat de l'évaluation de l'expression V (en remplaçant X par v) est vrai. Si aucune telle valeur v satisfaisant cette condition n'existe, alors l'instruction est équivalente à "`stop`".

```
B ::= ...
    | eval F (actual_parameter1,...,actual_parametern)
    | ...
```

Ce comportement désigne un appel d'une fonction F avec pour paramètres effectifs $actual_parameter_1, ..., actual_parameter_n$ au sein d'un processus. Ici, le mot clé `eval` sert à lever l'ambiguïté syntaxique qui existe entre l'appel de fonction et la communication non gardée sur une porte qui est présentée ci-après.

```
B ::= ...
    | Γ [[G1,...,Gn]] [(actual_parameter1,...,actual_parametern)]
    | ...
```

Ce comportement désigne l'appel du processus Γ avec pour portes de communication $G_1...G_n$ et pour paramètres effectifs $actual_parameter_1...actual_parameter_n$.

```
B ::= ...
    | G [(O0,...,On)] [where V]
    | ...
O ::= [!] V
  ::= ? P
```

Ce comportement désigne une communication sur la porte G dans laquelle les valeurs des données échangées $V_0...V_n$ doivent satisfaire les offres $O_0...O_n$. Une valeur satisfait "[!] V" si elle est égale au résultat de l'évaluation de l'expression V. Une valeur satisfait "`?` P" si elle satisfait le motif P. Si la clause "`where`" est présente, alors la communication ne s'effectue que si le résultat de l'évaluation de l'expression V est vrai.

B ::= ...
| `select`
  B_0
  `[] ... []`
  B_n
  `end select`
| ...

Ce comportement désigne le chox non-déterministe d'un comportement B_i parmi $B_0, ..., B_n$.

B ::= ...
| `par` $[G_0$`,...,`G_n `in`$]$
  $[G_{(0,0)}$`,...,`$G_{(0,n_0)}$ `->` $]$ B_0
  `|| ... ||`
  $[G_{(m,0)}$`,...,`$G_{(m,n_m)}$ `->` $]$ B_m
  `end par`
| ...

Ce comportement désigne l'exécution parallèle des comportements $B_0...B_m$. Pour chaque comportement B_i, on indique une liste de portes $G_{(i,0)}$`,...,`$G_{(i,n_i)}$ sur lesquelles il va se synchroniser par rendez-vous avec les autres comportements. A cette liste s'ajoutent éventuellement les portes $G_0, ..., G_n$ sur lesquelles tous les comportements B_i doivent se synchroniser. Lorsqu'un comportement veut effectuer une communication sur une porte G sur laquelle il doit se synchroniser, alors son exécution est suspendue jusqu'à ce que tous les autres comportements devant également se synchroniser sur G soient prêts à effectuer une communication sur G, auquel cas tous ces comportements se synchronisent et échangent éventuellement des données à cette occasion.

B ::= ...
| `hide` *gate_declaration*$_0$`,...,`*gate_declaration*$_n$ `in`
  B
  `end hide`
  ...

Ce comportement déclare des portes de communication (*gate_declaration*$_0$...*gate_declaration*$_n$) visibles seulement à l'intérieur du comportement B (on parle de portes *locales* à B). Toutes les communications sur les portes locales à B sont cachées et représentées par des transitions invisibles (`tau` ou `i`).

B ::= ...
| `disrupt` B_1 `by` B_2 `end disrupt`
  ...

Ce comportement exprime que l'exécution de B_1 peut être interrompue à tout moment pour laisser place à l'exécution de B_2. Si l'exécution de B_1 se termine sans être interrompue, alors B_2 n'est pas exécuté.

2.4.10 Exemple

L'exemple LOTOS NT suivant illustre le fonctionnement (volontairement simplifié) d'un ascenseur :

```
channel C is (N : Nat) end channel

process ELEVATOR [CALL, GO, UP, DOWN: C] (CURRENT, TARGET: INT) is
  loop L in
    if TARGET > CURRENT then
      CURRENT := CURRENT + 1;
      UP (CURRENT)
    elsif TARGET < CURRENT then
      CURRENT := CURRENT - 1;
      DOWN (CURRENT)
    else (* TARGET == CURRENT *)
      select
        CALL (?TARGET)
      []
        GO (?TARGET)
      end select
    end if
  end loop
end process
```

La variable "`CURRENT`" désigne l'étage auquel se trouve actuellement l'ascenseur tandis que la variable "`TARGET`" désigne l'étage qu'il doit atteindre. On distingue trois cas de fonctionnement :

- Si l'ascenseur se trouve en-dessous de l'étage demandé, il monte d'un étage (incrémentation de la variable "`CURRENT`") et il signale cette montée sur la porte "`UP`".
- Si l'ascenseur se trouve au-dessus de l'étage demandé, il descend d'un étage (décrémentation de la variable "`CURRENT`") et il signale cette descente sur la porte "`DOWN`".
- Si l'ascenseur a atteint l'étage demandé, il attend soit d'être appelé, soit que l'occupant lui signale un nouvel étage à atteindre.

Partie II

Modélisation et vérification de systèmes GALS

Chapitre 3

Motivations

La première partie de cette thèse est consacrée à l'étude de la vérification des systèmes GALS. Nous présentons dans cette section les paradigmes synchrone et asynchrone de modélisation et de vérification des systèmes critiques. Nous expliquons ensuite pourquoi et comment combiner ces deux paradigmes en vue de modéliser et vérifier des systèmes GALS. Puis, nous détaillons les travaux qui ont déjà été réalisés dans cette direction avant d'exposer nos contributions à cet axe de recherche.

3.1 Paradigme synchrone

3.1.1 Systèmes réactifs

Un système réactif [Hal93] est un système informatique dont l'exécution est cyclique et théoriquement infinie, pendant laquelle il attend que l'environnement lui transmette des entrées. Ces entrées peuvent être reçues à intervalles réguliers ou non, en fonction de l'environnement. Chaque entrée apporte au système un ensemble de valeurs à partir desquelles il va calculer une sortie (ou réaction) qui sera ensuite lue par l'environnement.

De tels systèmes sont très courants dans l'informatique embarquée et ils ont souvent un rôle critique. Par exemple, de nombreux systèmes de contrôle présents dans les avions, les trains, les voitures ou encore les centrales nucléaires, entrent dans la catégorie des systèmes réactifs.

Les systèmes réactifs sont dits synchrones [Hal93] s'ils satisfont deux propriétés fondamentales :

- **instantanéité** : la lecture de l'entrée, le calcul de la sortie et son envoi à l'environnement ont une durée d'exécution nulle. En d'autres termes, on considère que le temps de calcul (délai de réaction entre la lecture d'une entrée et l'écriture d'une sortie) est nul (ou suffisamment petit pour être négligeable), ce qui est effectivement le cas pour des systèmes réactifs critiques comme un ABS (*Anti-lock Braking System*).

- **déterminisme** : une même séquence d'entrées produira toujours la même séquence de sorties. Cette restriction provient du constat que les systèmes de contrôle critiques sont toujours déterministes (notamment à cause des contraintes liées à la certification) et elle a pour conséquence de simplifier l'analyse des systèmes réactifs décrits au moyen du modèle de calcul synchrone, que nous présentons à présent.

3.1.2 Modèle de calcul synchrone

Le modèle de calcul synchrone [Mil83] a été défini pour la programmation des systèmes réactifs. Dans ce modèle de calcul, un système réactif est vu comme un programme P qui reçoit à chaque instant t d'une horloge $t_0, t_1, t_2....$ donnant une suite infinie d'instants de temps logique, une valeur d'entrée I_t et qui produit instantanément une valeur de sortie O_t. Ce programme se conforme aux deux propriétés énoncées ci-dessus : le calcul d'une valeur de sortie est instantané et pour une séquence de valeurs d'entrée $I_{t_0}, I_{t_1}, I_{t_2}...$ donnée, il produira toujours la même séquence de valeurs de sortie $O_{t_0}, O_{t_1}, O_{t_2}....$

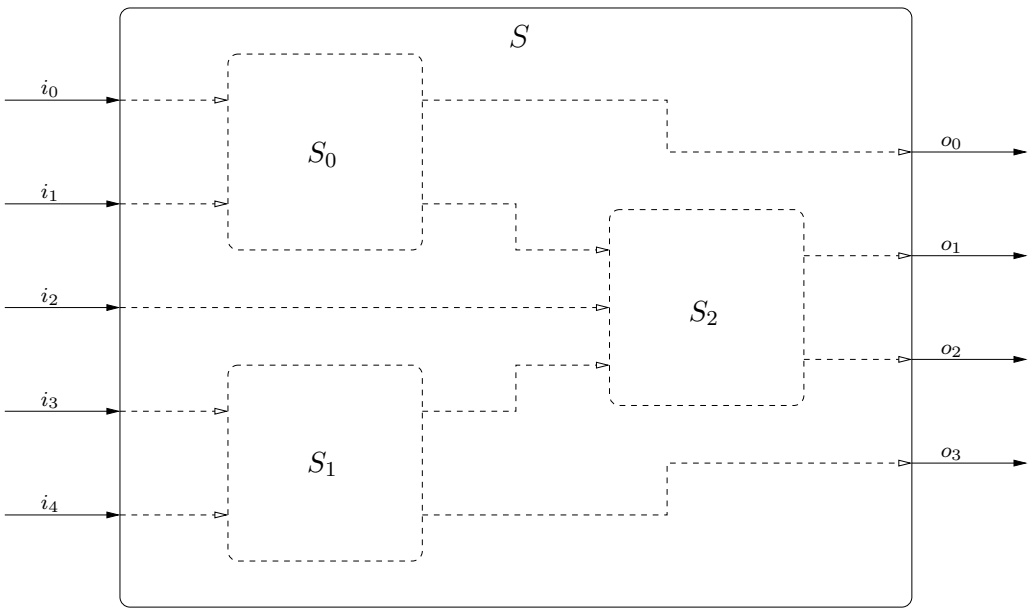

Figure 3.1: Exemple de composition synchrone de plusieurs sous-programmes synchrones

Les valeurs d'entrée I_t et de sortie O_t peuvent être des vecteurs de taille différentes de manière à couvrir le cas d'un système ayant m entrées $(i_t^0, ..., i_t^m)$ et n sorties $(o_0^t, ..., o_t^n)$. Cela permet de construire de nouveaux programmes synchrones par assemblage de programmes existants, en *propageant* les sorties de programmes vers les entrées d'un ou plusieurs autres programmes. La figure 3.1 montre la création d'un nouveau programme synchrone par connexion de trois *sous-programmes* existants. A chaque itération d'un tel programme synchrone, chaque sous-programme est exécuté exactement une fois, lorsque ses valeurs d'entrée ont été calculées. Ce mécanisme de connexion de plusieurs programmes synchrones est appelée *composition synchrone*. Une composition synchrone préserve les deux propriétés fondamentales :

- **instantanéité** : comme pour chaque sous-programme, la lecture des entrées, le calcul de la réaction et l'envoi des sorties sont des opérations instantanées, une composition synchrone est une séquence de calculs instantanés et est donc elle-même instantanée.

- **déterminisme** : il est aisé de montrer que la composition synchrone de sous-programmes déterministes est déterministe.

Un programme synchrone peut avoir un *état interne* qui est invisible pour l'environnement et dont la valeur peut changer pendant le calcul d'une réaction. Pour un programme synchrone muni d'un

état interne, la propriété de déterminisme est vérifiée si le programme garantit que pour une même valeur de l'état interne, la réaction calculée pour deux itérations identiques est la même.

3.1.3 Langages de programmation synchrone

Plusieurs langages de programmation ont été créés pour décrire, selon le paradigme synchrone, les systèmes réactifs. Ces langages sont dits de *programmation synchrone*. Ils permettent de définir :

- des programmes synchrones *simples* qui expriment les valeurs des sorties en fonction des entrées, et
- des programmes synchrones *complexes* qui sont le résultat de la composition synchrone de plusieurs programmes synchrones (simple ou complexes).

Les qualificatifs *simples* et *complexes* ne préjugent pas de la complexité d'écriture des programmes synchrones et sont seulement utilisés pour distinguer les deux types de programmes synchrones obtenus avec et sans composition. En effet, l'écriture d'un programme synchrone unique peut être complexe (s'il y a beaucoup de valeurs d'entrée et de sortie, par exemple) alors qu'il peut être plus facile de l'écrire sous la forme d'une composition synchrone de quelques programmes simples.

Nous listons ci-dessous les langages synchrones les plus connus. Tous ces langages permettent de définir la même classe de systèmes, les systèmes réactifs, mais avec des syntaxes différentes, qui reflètent les domaines d'origine de leurs concepteurs respectifs :

- Esterel [BG92] se concentre sur la description du flux de contrôle d'un système réactif et a une syntaxe impérative. Un programme Esterel consiste en un ensemble de sous-programmes concurrents (appelés *threads*) qui communiquent au moyen de signaux. Ces sous-programmes sont synchronisés par une horloge globale, de telle sorte qu'à chaque itération du programme, un sous-programme va s'exécuter et modifier la valeur de certains signaux, jusqu'à arriver à une instruction spéciale (nommée "`pause`") sur laquelle il va s'arrêter. Lors de l'itération suivante, chaque sous-programme reprendra son exécution où il l'avait précédemment suspendue. L'état interne d'un programme Esterel correspond aux portions de code sur lesquelles l'exécution de chaque sous-programme est suspendue.
- Lustre/Scade [HCRP91] utilise un modèle de flux de données similaire à ceux utilisés par les ingénieurs automaticiens, qui se servent souvent soit de systèmes d'équations soit de réseaux de flux de données. La syntaxe de Lustre est essentiellement déclarative et permet de définir les équations qui vont calculer les valeurs de sortie en fonction des valeurs d'entrée. En Lustre, au cours de la réaction de l'instant t_n, il est possible d'accéder aux valeurs de variables calculées à la réaction de l'instant t_{n-1}, voire $t_{n-2}, t_{n-3}, ...$, mais uniquement en remontant un nombre borné d'instants du passé. Ces valeurs représentent l'état interne du programme.
- Signal/Sildex [BLJ91] (devenu Polychrony), à l'instar de Lustre, permet de définir des équations qui calculent les valeurs de sortie en fonction des valeurs d'entrée. Ces valeurs sont appelées des signaux et ont des horloges propres. Cela rend le langage plus puissant que Lustre, mais complique la tâche du programmeur, car ces signaux sont moins intuitifs à manipuler que dans le modèle à flux de données de Lustre [HCRP91].

Il existe d'autres langages synchrones, notamment : Argos [MR01], qui a une syntaxe graphique pour définir les programmes synchrones simple à l'aide d'automates et les programmes synchrones complexes à l'aide de flèches connectant des sous-programmes ; Ssm [And03], qui est un langage à vocation industrielle, commercialisé dans Scade 4 et 5; Sam [CGT08], qui est un langage dont la syntaxe est similaire à celle d'Argos et dans lequel on retrouve plusieurs concepts de Ssm (priorités entre transitions, absence de dépendances cycliques...).

Ces langages ont souvent (c'est le cas pour ESTEREL, LUSTRE et SIGNAL) plusieurs sémantiques alternatives :

- une sémantique formelle (dénotationnelle pour ESTEREL, équationnelle pour LUSTRE et opérationnelle ou dénotationnelle pour SIGNAL) qui définit mathématiquement le calcul des sorties en fonction des entrées. Cette sémantique sert pour l'analyse formelle des programmes synchrones.

- plusieurs sémantiques données sous la forme de traductions depuis les langages synchrones vers des langages de programmation de bas niveau (comme le langage C ou le code assembleur) ou bien vers des formats intermédiaires (comme le format OC [PBM+93]). Ces sémantiques sont utilisées dans l'industrie pour la génération de code certifié à partir de programmes décrits à l'aide des langages de programmation synchrones.

3.1.4 Vérification des programmes synchrones

La vérification des programmes synchrones est une problématique importante, étant donné que ces langages sont utilisés dans l'industrie pour la conception de contrôleurs dans les avions, les voitures, les centrales nucléaires... La fiabilité de ces contrôleurs est critique et il est donc important de pouvoir s'assurer de leur bon comportement. Les principales méthodes de vérification sont les suivantes :

- le test automatique ;

- le *model checking* pour les configurations finies ou que l'ont sait ramener à un cas fini ;

- la preuve par SMT (*Satisfiability Modulo Theory*, une généralisation du problème SAT) à l'aide de solveurs numériques qui savent traiter des modèles infinis ;

- l'interprétation abstraite, qui sait aussi traiter des modèles infinis.

Toutes ces méthodes s'appuient sur les hypothèses d'instantanéité et de déterminisme des programmes synchrones, hypothèses qui réduisent les comportements à considérer et, par conséquent, la complexité de la tâche de vérification.

Le test automatique consiste à simuler le programme synchrone au moyen de séquences d'entrées prédéfinies ou générées aléatoirement et à vérifier que les réponses du programme sont correctes. Nous avons trouvé dans la littérature deux techniques majeures pour effectuer cette vérification :

- L'utilisation d'invariants dans le code du programme synchrone est préconisée dans [HLR92, BCE+03]. Dans cet article, les auteurs étendent le langage LUSTRE avec un nouvel opérateur `assert` qui permet de vérifier lors de la simulation que des invariants ne sont pas violés.

- L'utilisation d'observateurs [HLR93, CMSW99, RNHW02], qui sont des programmes synchrones, spécifiés par l'utilisateur, et qui encodent une propriété de bon fonctionnement vérifiant que l'évolution dans le temps de certaines entrées et sorties du programme à tester est correcte. Ces entrées et sorties sont capturées par l'observateur grâce au mécanisme de composition synchrone entre l'observateur et le programme à tester. Les mêmes auteurs, dans [HR99] proposent d'utiliser deux observateurs A et P pour générer automatiquement des cas de tests pour un programme synchrone S. Dans leur approche, A émet des contraintes sur les entrées et les sorties de S de telle sorte qu'une réaction est déclarée *d'intérêt* pour la propriété de l'observateur P si la valeur de sortie de l'observateur A est vraie.

La vérification de programmes synchrones par *model checking* consiste à représenter un programme synchrone dans un formalisme de système de transitions, puis à vérifier sur ce système de transitions des propriétés de bon fonctionnement souvent exprimées au moyen d'une logique temporelle.

Contrairement à l'approche par simulation, la vérification par *model checking* n'est pas toujours applicable, car elle est limitée par la taille du système de transitions à générer. En effet, ce système de transitions peut être trop gros pour être construit ou traité par les outils de vérification. Nous exposons ci-dessous des approches par *model checking* pour chacun des trois *grands* langages synchrones :

- [Bou98] présente une boîte à outils pour ESTEREL nommée XEVE. Les outils présentés sont capables de transformer un programme ESTEREL en une machine à état finis pour vérifier l'absence d'interblocages et de famines.
- [HR99] présente LESAR, un *model checker* qui transforme un programme LUSTRE en un système de transitions. L'expression des propriétés se fait à l'aide d'observateurs écrits en LUSTRE qui sont composés avec le système à tester. Cette composition est ensuite transformée en un système de transitions par une énumération de tous les comportements possibles. Enfin, la satisfaction ou non des propriétés est décidée par l'accessibilité (depuis les états *initiaux* du système de transitions), d'états dits *mauvais* qui représentent les cas où la propriété à tester n'est pas satisfaite.
- La vérification par *model checking* de programmes SIGNAL se fait grâce à une représentation de ces programmes sous la forme de systèmes d'équations [MRLBS01] qui sont eux-mêmes des représentations de systèmes de transitions. Les propriétés à vérifier sont exprimées à l'aide d'opérateurs mathématiques que le *model checker* va interpréter sur le système d'équations représentant le programme à tester.

La preuve par SMT des programmes synchrones est une direction de recherche relativement récente, comme en atteste [HT08]. Dans cet article, les auteurs décrivent la représentation de programmes LUSTRE et des contraintes qu'ils doivent satisfaire sous la forme d'un système d'équations SMT qu'un solveur numérique doit ensuite résoudre (en l'occurrence, les auteurs ont utilisé les outils CVC3 [BT07] et YICES [DDM06]). Il s'agit d'une avancée par rapport à ce que propose l' outil industriel LUSTRE/SCADE, dans lequel ce type de vérification est effectué à l'aide d'un solveur numérique SAT dont les équations d'entrée ont une expressivité moindre.

L'analyse statique des programmes synchrones par interprération abstraite est issue des travaux de Nicolas Halbwachs [Hal94]. Depuis, cette technique s'est répandue et est présente dans SCADE sous la forme d'une analyse statique, à l'aide de l'outil ASTRÉE [CCF+05], du code C généré par les compilateurs des langages ESTEREL et LUSTRE.

3.2 Systèmes GALS

Ces vingt dernières années, les langages synchrones ont été fréquemment utilisés pour la programmation de systèmes critiques embarqués ; les suites d'outils ESTEREL, LUSTRE/SCADE, SIGNAL/SILDEX sont utilisées pour mettre au point et vérifier formellement des contrôleurs pour avions, voitures, centrales nucléaires... Ces langages ont aussi trouvé des applications dans la conception des circuits électroniques.

Cependant, de plus en plus, les systèmes embarqués ne satisfont plus les hypothèses des systèmes réactifs. En effet, les approches récentes (*modular avionics*, *X-by-wire*...) introduisent un degré croissant d'asynchronisme et de non-déterminisme. Cette situation est connue depuis longtemps dans l'industrie des circuits électroniques où le terme GALS [Cha84] (*Globalement Asynchrone Localement Synchrone*) a été inventé pour désigner les circuits qui consistent en un ensemble de composants synchrones (gouvernés par leur propre horloge) qui communiquent de façon asynchrone. Ces évolutions remettent en cause la position établie des langages synchrones dans l'industrie. En effet,

l'asynchronisme invalide les propriétés de non déterminisme et d'instantanéité des systèmes réactifs et rend donc caduques les techniques de vérification efficaces qui existent pour ces systèmes. Il devient alors nécessaire d'adapter les techniques de vérification existantes au cas des systèmes GALS.

Nous invitons le lecteur à se référer à [MWC10] pour une liste de succès récents dans l'application des techniques de vérification formelle à des systèmes avioniques complexes synchrones et asynchrones.

3.3 Etat de l'art sur la vérification de systèmes GALS

Nous avons trouvé dans la littérature diverses tentatives visant à repousser les limites des langages synchrones pour les appliquer à l'étude des systèmes GALS. Suivant les résultats de Milner [Mil83] qui ont montré que l'asynchronisme peut être encodé dans le modèle de calcul synchrone, nombre d'auteurs [HB02, LTL03, MLT+04, HM06] se sont efforcés de décrire les systèmes GALS à l'aide de langages synchrones ; par exemple, le non-déterminisme est exprimé par l'ajout d'entrées auxiliaires (*oracles*) dont la valeur binaire est indéfinie. Le désavantage principal de ces approches est que l'asynchronisme et le non-déterminisme ne sont pas reconnus comme des concepts de première classe, donc les outils de vérification des langages synchrones ne bénéficient pas des optimisations spécifiques au parallélisme asynchrone (ordres partiels, minimisation compositionnelle...). D'autres approches étendent les langages synchrones pour permettre un certain degré d'asynchronisme, comme dans les langages CRP [BRS93], CRSM [Ram98] ou encore multiclock ESTEREL [BS01] ; cependant à notre connaissance, de telles extensions ne sont pas (encore) utilisées dans l'industrie. Enfin, nous pouvons mentionner les approches [GM02, PBC07] dans lesquelles les langages synchrones sont compilés et distribués automatiquement sur un ensemble de processeurs s'exécutant en parallèle. Bien que ces approches permettent de générer directement des implémentations de systèmes GALS, elles ne permettent pas de traiter la modélisation et de la vérification de ces systèmes.

Une approche totalement différente consiste à abandonner les langages synchrones et à adopter des langages spécifiquement conçus pour modéliser le parallélisme asynchrone et le non-déterminisme, notamment les algèbres de processus (CSP [BHR84], LOTOS [ISO89]...) ou PROMELA [Hol04], qui sont équipés de puissants outils de vérification formelle. Un tel changement de paradigme est aujourd'hui difficile pour des entreprises qui ont investi massivement dans les langages synchrones et dont les produits à cycle de vie extrêmement long demandent une certaine stabilité en termes de langages de programmation et d'environnements de développement. Un compromis serait alors de combiner les langages synchrones et les algèbres de processus de telle sorte que les programmes synchrones continuent d'être vérifiés avec leurs outils habituels avant que leur composition asynchrone ne soit exprimée dans une algèbre de processus dans le but d'être vérifiée. C'est l'approche que nous préconisons ici.

Nous avons trouvé dans la littérature deux approches qui suivent cette direction :

- Dans [RSD+04], des spécifications CRSM [Ram98] sont automatiquement traduites en PROMELA pour vérifier, grâce au *model checker* SPIN, des propriétés exprimées comme un ensemble d'observateurs. Notre approche est différente, car nous réutilisons les langages synchrones tels qu'ils sont, sans qu'il soit nécessaire d'introduire un nouveau langage synchrone/asynchrone comme CRSM.

- Dans [DMK+06], le compilateur SIGNAL est utilisé pour générer du code C à partir de programmes synchrones écrits en SIGNAL. Ce code est ensuite encapsulé dans des processus PROMELA qui communiquent par une abstraction d'un bus matériel. Enfin, le *model checker* SPIN est utilisé pour vérifier des formules de logique temporelle sur la spécification obtenue.

L'approche que nous proposons suit le principe proposé dans [DMK+06] mais présente des différences clés dans la façon d'intégrer les programmes synchrones dans un environnement asynchrone :

- Le protocole de communication qui relie les deux programmes synchrones présentés dans [DMK+06] est implémenté dans [BCG+02] en LUSTRE et SIGNAL. Ce protocole présente un degré faible d'asynchronisme et aucun non déterminisme. Il est d'ailleurs prouvé que ce protocole équivaut à un canal FIFO sans perte à un élément. Notre approche est plus générale, car, comme dans l'approche synchrone, on fait l'hypothèse que les calculs sur les données locales se font en temps zéro (c'est effectivement le cas de l'évaluation des fonctions LOTOS NT) ; en revanche, contrairement à l'approche synchrone, on ne suppose pas que les communications (envois de messages) se font en temps zéro. En effet, la communication entre le programme synchrone et son environnement peut se faire, soit directement à l'aide d'un canal de communication, soit par l'intermédiaire d'un processus asynchrone auxiliaire qui implémente un protocole donné.

- Le degré d'asynchronisme est encore limité par l'utilisation de la directive "**atomic**" de PROMELA qui assure le non-entrelacement de la séquence d'actions qu'elle englobe avec les actions de l'environnement. Dans leur approche, cette directive englobe la totalité des actions de chacun des deux processus asynchrones qui encapsulent les deux programmes synchrones (qui sont représentés par une fonction C générée par le compilateur de SIGNAL). De cette façon, la réception des entrées dans l'un des processus asynchrones, l'appel de la fonction C encodant le programme synchrone et l'envoi des sorties à l'environnement sont une séquence atomique d'actions. Les deux programmes synchrones ne peuvent donc pas s'exécuter de façon concurrente, ce qui, pour nous, ne constitue pas un vrai exemple de système GALS. Au contraire, notre approche est complètement asynchrone, car les exécutions des processus asynchrones qui encapsulent les programmes synchrones peuvent s'entrelacer. Notre approche est donc plus générale car : (1) elle permet une modélisation plus réaliste de la sémantique des systèmes GALS (elle ne requiert pas l'arrêt du système tout entier durant le calcul de la réaction de l'un des programmes synchrones) et (2) elle est applicable à beaucoup d'algèbres de processus qui, pour la plupart (contrairement à PROMELA), ne possèdent pas de directive "**atomic**" ; les seules contraintes pour ces algèbres de processus sont de permettre à l'utilisateur de définir des types et des fonctions ainsi que d'avoir les primitives classiques pour exprimer le parallélisme asynchrone.

- Dans leur approche, les processus asynchrones qui encapsulent les programmes synchrones (cette notion d'encapsulation est présentée à la section 4.3) sont "vides", au sens où ils constituent une *coquille* transparente qui ne fait que transmettre les valeurs reçues par l'environnement au programme synchrone. Dans la réalité, ce schéma est parfois trop restrictif, car il arrive que des programmes synchrones ne spécifient que la partie "contrôle" d'une application et que ce soient les programmes asynchrones qui définissent et manipulent les données et éventuellement injectent du non-déterminisme. Dans notre approche, cela est possible et le degré de complexité des processus asynchrones encapsulateurs peut varier selon le système GALS à modéliser.

3.4 Contributions

Dans cette première partie de la thèse, nous proposons une approche qui combine les langages synchrones et les algèbres de processus pour modéliser, vérifier et simuler les systèmes GALS. Notre approche essaie de retenir le meilleur des deux paradigmes :

- nous continuons à utiliser les langages synchrones et leurs outils pour spécifier et vérifier les composants synchrones d'un système GALS,

- nous introduisons une algèbre de processus pour : (1) encapsuler ces composants synchrones ; (2)

modéliser des composants additionnels dont le comportement est non déterministe, comme par exemple des canaux de communications non sûrs qui peuvent perdre, dupliquer et/ou permuter des messages ; (3) interconnecter tous ces composants d'un même système GALS grâce aux opérateurs de parallélisme asynchrone présents dans l'algèbre de processus. La spécification qui résulte est donc asynchrone et peut être analysée par les outils accompagnant l'algèbre de processus considérée.

Nous illustrons notre approche par une étude de cas industrielle fournie par Airbus dans le contexte du projet TOPCASED : un protocole de communication entre un avion et le sol qui consiste en deux entités TFTP (*Trivial File Transfer Protocol*) qui s'exécutent en parallèle et qui communiquent par un canal UDP (*User Datagram Protocol*). Comme langage synchrone, nous considérons SAM [CGT08], qui a été conçu par Airbus et qui est utilisé au sein de cette entreprise. Un éditeur pour SAM, un générateur de code et un traducteur de SILDEX vers SAM sont distribués avec la plateforme *open-source* TOPCASED basée sur Eclipse (voir figure 3.4). Nous présentons SAM en détail à la section 4.5. Comme algèbre de processus, nous considérons LOTOS NT (cf. section 2.3). Grâce à CADP, les spécifications LOTOS NT peuvent être analysées et leurs performances évaluées (en l'occurrence, nous calculons le temps moyen de transfert d'un fichier).

La suite est organisée comme suit. Le chapitre 4 expose les principes de notre approche et illustre son application aux langages SAM et LOTOS NT. Ensuite, le chapitre 5 détaille l'étude de cas d'Airbus que nous avons traitée, sa modélisation par application de notre méthode, la vérification de son bon comportement et l'évaluation de ses performances.

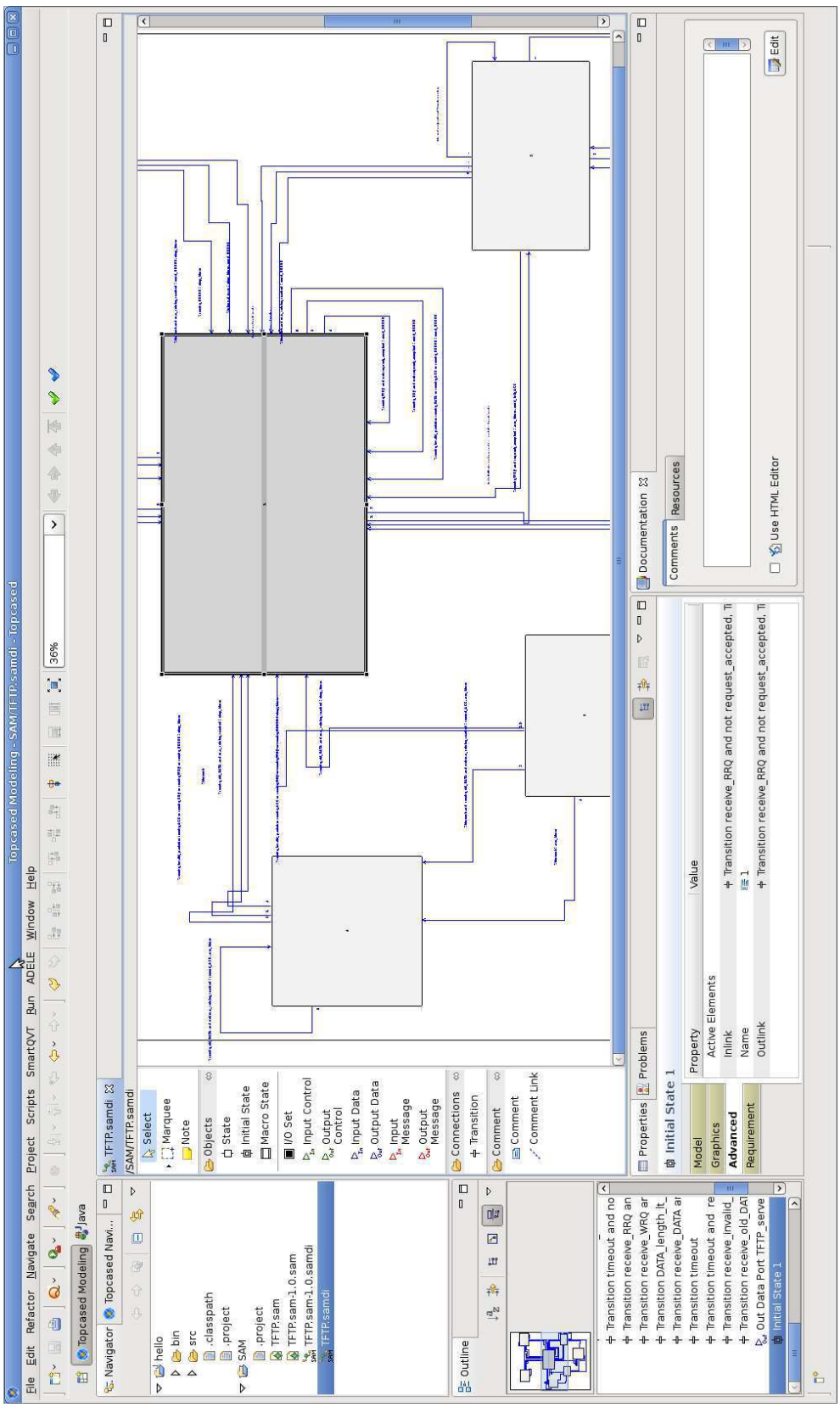

Figure 3.2: Vue d'ensemble de la plateforme TOPCASED dans l'environnement Eclipse

Chapitre 4

Approche proposée

Dans cette section, nous détaillons notre approche pour la modélisation des systèmes GALS à l'aide des langages synchrones et des algèbres de processus. Nous présentons ensuite l'application de cette méthode aux langages SAM et LOTOS NT.

4.1 Description générale de l'approche

Le but de notre approche est de rendre possible la modélisation de systèmes GALS composés d'un nombre arbitraire[32] de composants synchrones immergés au sein d'un environnement asynchrone. Notre approche se veut générique, c'est-à-dire qu'elle n'est pas liée à un langage synchrone ou une algèbre de processus en particulier. Il faut toutefois que l'algèbre de processus considérée permette la définition de types et de fonctions et soit munie d'un opérateur de composition parallèle n-aire. Avec cette approche, il est même possible de modéliser un système GALS dont les composants synchrones ont été écrits dans des langages synchrones différents.

Pour ce faire, nous encodons un programme synchrone comme une fonction dans l'algèbre de processus considérée. Nous expliquons à la section 4.2 pourquoi et comment cela est faisable. Ensuite, nous construisons un processus asynchrone (dit *coquille*) qui encapsule un programme synchrone et le fait communiquer de façon asynchrone. Autrement dit, le processus asynchrone traduit les données reçues de l'environnement en entrées pour le programme synchrone et traduit les sorties du programme synchrone en données qu'il renvoie à l'environnement. Selon les applications GALS, le degré de complexité de la *coquille* peut varier et nous revenons sur ce point dans la section 4.3. Enfin, nous présentons, à la section 4.4, différents schémas de communication entre le programme synchrone et son environnement, via la *coquille* qui l'encapsule.

4.2 Encodage des programmes synchrones

Nous partons du constat qu'un programme synchrone, qu'il soit *simple* ou *complexe* peut être représenté sous la forme d'une machine de Mealy [Mea55] qui est une sorte d'automate communiquant. Une machine de Mealy est un quintuplet ($\mathcal{S}$, s_0, $\mathcal{I}$, $\mathcal{O}$, f) où :

[32] le nombre de composants peut être arbitraire, mais il doit être fini et connu à l'avance si l'on veut faire de la vérification énumérative en utilisant les outils de CADP, et notamment les compilateurs LOTOS NT et LOTOS qui ne permettent pas la création dynamique de composants s'exécutant en parallèle

- $\mathcal{S}$ est un ensemble fini d'états ;
- s_0 est l'état initial ;
- $\mathcal{I}$ est un alphabet fini d'entrée ;
- $\mathcal{O}$ est un alphabet fini de sortie ;
- $f \in \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{S} \times \mathcal{O}$ est une fonction de transition (aussi appelée une *fonction de Mealy*) qui associe à l'état courant et un symbole de l'alphabet d'entrée, l'état pour la réaction suivante ainsi qu'un symbole de l'alphabet de sortie. En considérant qu'un symbole de l'alphabet d'entrée (*resp.* de sortie) est un ensemble de valeurs $i_1...i_m$ (*resp.* $o_1...o_n$), nous pouvons écrire : $f(s, i_1...i_m) = (s', o_1...o_n)$.

La fonction de Mealy d'un programme synchrone peut être facilement encodée en utilisant les types et fonctions d'une algèbre de processus. Les compilateurs des langages synchrones s'inspirent du même constat pour générer du code exécutable.

Nous avons identifié trois méthodes pour produire dans l'algèbre de processus considérée la fonction de Mealy d'un programme synchrone :

- Si une sémantique de transformation du langage synchrone considéré vers le formalisme des machines de Mealy existe, alors il est envisageable de réaliser un traducteur automatique qui implémente cette sémantique de transformation, avec l'algèbre de processus considérée comme langage cible. Dans certains cas, comme les langages ESTEREL, LUSTRE ou SIGNAL, cela reviendrait à re-développer des compilateurs qui ont mis des années à voir le jour. Cette méthode est donc réservée aux langages synchrones relativement simples.
- Les compilateurs de certains langages synchrones comme ESTEREL, LUSTRE ou SIGNAL utilisent (ou ont utilisé) un format intermédiaire commun nommé OC [PBM+93] (*Object Code*). Ce format encode la relation de transition de la fonction de Mealy et peut donc être utilisé comme point de départ d'une traduction vers les types et fonctions de l'algèbre de processus considérée.
- Si l'algèbre de processus considérée est capable d'appeler des fonctions définies dans un autre langage de programmation (tel que C ou C++) et qu'il existe un compilateur vers ce langage depuis le langage synchrone considéré, alors l'interface entre le code généré par le compilateur et l'algèbre de processus est directe. Par exemple, les compilateurs de la plupart des langages synchrones produisent du code C, langage avec lequel les boîtes à outils SPIN et CADP savent s'interfacer.

En combinant ces méthodes, il est parfaitement possible d'immerger dans un même environnement asynchrone des programmes décrits dans des langages synchrones différents.

4.3 Encapsulation dans un processus asynchrone

Afin de permettre les communications entre un programme synchrone et son environnement asynchrone, la fonction de Mealy qui lui correspond dans l'algèbre de processus considérée doit être encapsulée dans un processus asynchrone que l'on appelle *coquille*. Ce processus a pour tâche de recevoir des valeurs depuis l'environnement, d'appeler la fonction de Mealy du programme synchrone avec ces valeurs, et de transmettre les valeurs renvoyées par la fonction de Mealy à l'environnement. Selon la méthode utilisée pour obtenir la fonction de Mealy, l'état du programme synchrone est explicite ou non : s'il est explicite, alors cet état fait partie des entrées de la fonction de Mealy, sinon la fonction possède une variable interne dont la valeur représente l'état courant. Dans le code généré par

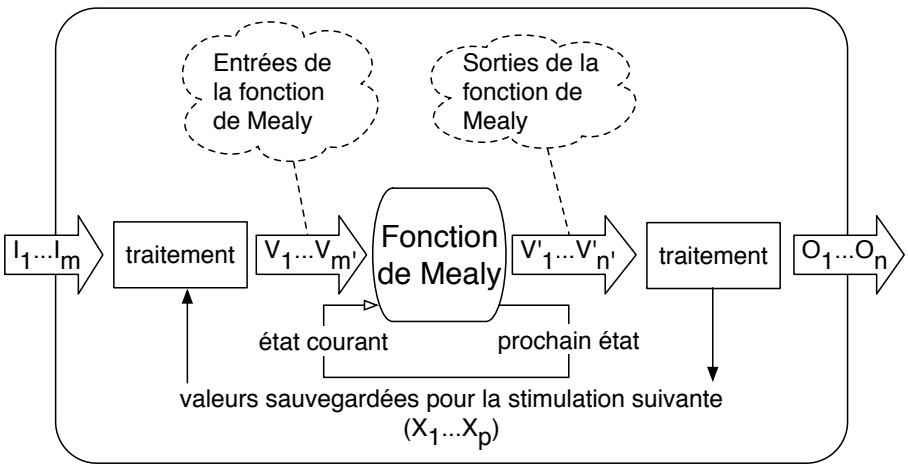

Figure 4.1: Processus asynchrone *coquille* général

les compilateurs des langages synchrones, l'état des programmes est souvent implicite. L'utilisateur doit alors fournir cette entrée. Si l'état du programme synchrone est explicite, alors la *coquille* doit mémoriser cet état après le calcul de chaque réaction afin de pouvoir le passer à la fonction de Mealy lors de l'itération suivante.

Dans les applications GALS modernes comme les systèmes de contrôle dans l'avionique et l'industrie automobile, les contrôleurs synchrones sont encapsulés dans des *coquilles* asynchrones qui font l'interface entre les contrôleurs synchrones et le réseau informatique qui les connecte. En règle générale, il est intéressant d'implémenter la partie du protocole de communication propre à chaque contrôleur dans la *coquille* associée et de laisser au contrôleur le soin de s'occuper du contrôle seulement. La *coquille* doit alors transformer un flux de données, reçues de l'environnement, en valeurs booléennes ("présent" ou "absent") qui seront traitées par le contrôleur : ceci est illustré par l'étude de cas que nous présentons au chapitre suivant. La *coquille* n'est donc pas qu'un artefact nécessaire à la modélisation des systèmes GALS : elle apparaît naturellement dans les implémentations de ces systèmes.

L'exécution d'une coquille asynchrone peut être représentée par le schéma de la figure 4.1. En LOTOS NT, cette exécution peut s'écrire comme suit, où un programme synchrone est représenté par sa fonction de Mealy F_{mealy} :

```
process W [G_in, G_out] is
  var S:{s_0, ...}, X_1:T_1^x, ..., X_p:T_p^x in
    S := s_0;
    X_1 := default;
    ...
    X_p := default;
    loop
      var I_1:T_1^i, ..., I_m:T_n^i, V_1:T_1, ..., V_m':T_m',
          V'_1:T'_1, ..., V'_n':T'_n', O_1:T_1^o, ..., O_n:T_n^o in
        G_in (?M_in (I_1, ..., I_m));
        traitement des données reçues et des valeurs sauvegardées
        F_mealy (S, V_1, ..., V_m', ?S, ?V'_1, ..., ?V'_n');
```

```
        traitement des données à renvoyer et sauvegarde de valeurs pour prochaine itération
        G_out (!M_out (O_1,...,O_n))
      end var
    end loop
  end var
end process
```

La *coquille* commence par déclarer la variable S contenant l'état interne de la fonction de Mealy, ainsi que les variables $X_1, ..., X_p$ qui vont contenir les valeurs à sauvegarder entre deux itérations. S est initialisée à l'état initial de la fonction de Mealy tandis que les variables $X_1, ..., X_p$ reçoivent des valeurs initiales qui dépendent de l'application GALS considérée. Ensuite, le processus entame une boucle infinie. Le corps de la boucle commence par déclarer les variables $I_1, ..., I_m$, $V_1, ..., V_{m'}$, $V'_1, ..., V'_{n'}$ et $O_1, ..., O_n$ destinées à stocker les valeurs d'entrée et de sortie. Puis, la *coquille* reçoit les valeurs d'entrée $I_1, ..., I_m$. Ces valeurs d'entrée, ainsi que les valeurs $X_1, ..., X_p$ sauvegardées à l'itération précédente, sont utilisées pour calculer les valeurs d'entrée $V_1, ..., V_{m'}$ de la fonction de Mealy dont le premier argument est l'état interne (noté S). La fonction de Mealy est ensuite exécutée, à la suite de quoi elle met à jour l'état interne et renvoie les valeurs $V'_1, ..., V'_{n'}$. Celles-ci sont utilisées par la *coquille* pour déterminer les valeurs $O_1, ..., O_n$ à communiquer à l'environnement. Enfin, les valeurs $X_1, ..., X_p$ sont mises à jour en fonction de $V'_1, ..., V'_{n'}$.

4.4 Communication avec l'environnement

Une fois le programme synchrone encapsulé dans un processus asynchrone, il peut s'exécuter en parallèle de son environnement et communiquer avec lui. Ce parallélisme est garanti quels que soient la méthode choisie pour générer la fonction de Mealy et le langage dans lequel celle-ci est encodée.

Les communications avec l'environnement se font au moyen de synchronisations par rendez-vous avec échanges de données, une opération fondamentale des algèbres de processus. L'environnement peut être spécifié entièrement dans l'algèbre de processus considérée ou bien contenir d'autres programmes synchrones encapsulés sans que cela ne change la modélisation des communications.

Comme indiqué à la section précédente, l'environnement et la *coquille* communiquent au moyen de deux portes G_{in} et G_{out}. La mise en parallèle de l'environnement et de la *coquille* se fait au moyen de l'opérateur de composition parallèle de LOTOS NT. Il en résulte le code LOTOS NT suivant pour la mise en parallèle de l'environnement et de la *coquille*, où W (pour *wrapper*) désigne la *coquille* et E l'environnement :

```
par
  G_in,G_out -> W [G_out,G_in]
||
  G_in,G_out -> E [G_in,G_out]
end par
```

Si les communications entre la *coquille* et l'environnement suivent un protocole donné ou bien passent par un médium aux propriétés particulières, alors, dans la modélisation, un processus tiers doit être inséré entre la *coquille* et l'environnement. Ce processus tiers (noté M) garantit le respect du protocole et implémente les propriétés du médium :

```
par
  G_in,G_out -> W [G_in,G_out]
||
  G_out,G'_in,G'_out,G_in -> M[G_out,G'_in,G'_out,G_in]
||
  G'_in,G'_out -> E [G'_in,G'_out]
end par
```

Ici, la *coquille* (*resp.* l'environnement) reçoit sur la porte G_{in} (*resp.* G'_{in}) et émet sur le porte G_{out} (*resp.* G'_{out}). Le processus tiers connecte, lorsque le protocole ou les propriétés du médium le permettent, les portes G_{out} avec G'_{in} et G'_{out} avec G_{in}. Ce mécanisme permet aussi, si le processus tiers est un médium FIFO par exemple, de simuler des communications non bloquantes et donc de désynchroniser davantage la *coquille* de l'environnement.

Lorsque le processus tiers M est symétrique, c'est-à-dire que les messages envoyés par l'environnement vers la *coquille* et par la *coquille* vers l'environnement sont traités à l'identique, alors il peut être défini comme la mise en parallèle de deux processus identiques (*half-duplex*) $M_{/\frac{1}{2}}$ (mais communiquant sur des portes différentes) et l'exécution parallèle précédente peut être redéfinie comme suit :

```
par
  G_in,G_out -> W [G_in,G_out]
||
  G_out,G'_in -> M_/1/2 [G_out,G'_in]
||
  G'_out,G_in -> M_/1/2 [G'_out,G_in]
||
  G'_in,G'_out -> E [G'_in,G'_out]
end par
```

4.5 Le langage SAM

4.5.1 Présentation

Le langage synchrone SAM a été défini en interne à Airbus en 2004. SAM s'inspire du langage SA-RT [HP88] qui était utilisé par Airbus depuis 2001 et dont il reprend l'idée de combiner modélisation fonctionnelle et automates. Entre 2001 et 2004, Airbus a aussi utilisé certaines fonctionnalités de l'outil SILDEX pour remplacer SA-RT, avant de s'orienter vers la conception de SAM, un DSL conçu pour s'adapter à la culture technique des ingénieurs Airbus. Un éditeur graphique est disponible pour SAM au sein de la plateforme TOPCASED (voir figure 3.4).

A l'heure actuelle, le langage SAM est principalement utilisé au sein d'Airbus pendant les phases de spécification des logiciels avioniques : SAM a ainsi été utilisé pour développer les spécifications logicielles de cinq applications embarquées de l'A350.

A un niveau plus détaillé, le langage SAM est parfois aussi utilisé pour l'analyse fonctionnelle, comme pour les aspects architecturaux et les logiciels du cockpit de l'A350, notamment. Lorsque les spécifications fonctionnelles sont suffisamment détaillées (cas des protocoles, notamment), il est alors possible d'effectuer des analyses comportementales et de la génération automatique de cas de tests.

Dans cette thèse, nous nous servons de SAM dans le cadre d'une étude de cas présentée au chapitre

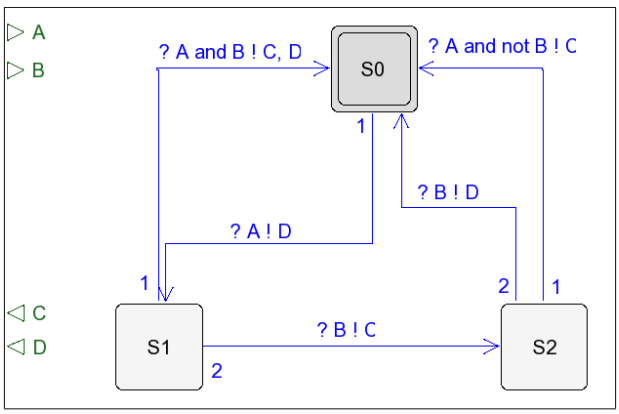

Figure 4.2: Exemple d'automate SAM

suivant et qui illustre notre approche de la modélisation des systèmes GALS. Au début de nos travaux, il n'existait pas de sémantique formelle de SAM et nous avons contribué à en établir une [CGT08]. Cette sémantique formelle est définie sur une version simplifiée du langage dans laquelle le sucre syntaxique a été éliminé.

Un programme synchrone SAM *simple* (voir figure 4.2) est un automate possédant un ensemble de ports d'entrée et un ensemble de ports de sortie. A chaque port correspond une variable booléenne. Les transitions sont étiquetées par des expressions booléennes sur les variables associées aux ports d'entrée. Une transition est franchie à la réception des valeurs d'entrées, si son état source est l'état courant et l'expression de son étiquette est évaluée à "`vrai`". Le franchissement d'une transition affecte une valeur aux variables associées aux ports de sortie de l'automate. Au cours du calcul d'une réaction, un système de priorités entre transitions garantit qu'une unique transition peut être franchie.

Un programme synchrone SAM *complexe* est la composition synchrone classique de plusieurs sous-programmes SAM. La figure 4.3 illustre un tel assemblage.

4.5.2 Sémantique

Formellement, un automate SAM est très proche d'une machine de Mealy. La principale différence réside dans le fait qu'une transition d'un automate SAM est un quintuplet (s_1, s_2, F, G, P), où :

- s_1 est l'état source de la transition ;
- s_2 est l'état destination de la transition ;
- F est une condition booléenne sur les variables d'entrée (la transition peut être franchie seulement lorsque F est vraie) ;
- G est un ensemble de variables de sortie (lorsque la transition est franchie, la valeur "`vrai`" est affectée à toutes les variables de sortie apparaissant dans G tandis que les autres variables de sortie reçoivent la valeur "`faux`"),
- P est un indice de priorité qui définit un ordre total sur les transitions sortantes d'un même état.

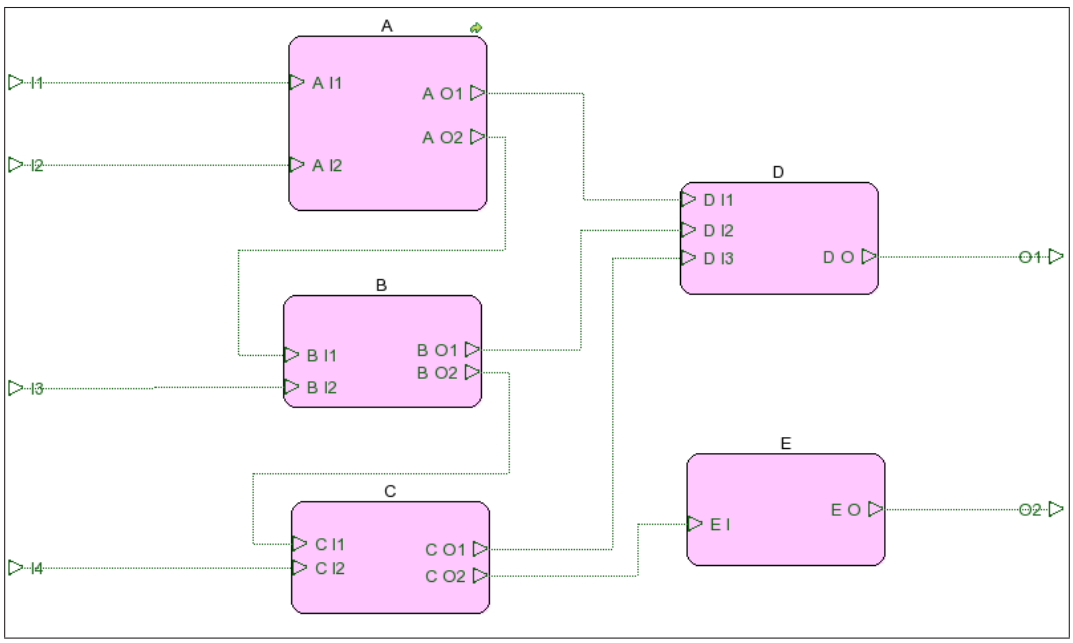

Figure 4.3: Programme Sam construit par composition de cinq sous-programmes

Les indices de priorité de transitions sortant d'un même état doivent être deux à deux distincts. Si les valeurs des variables d'entrée permettent le franchissement de plusieurs transitions, alors celle dont l'indice de priorité est le plus faible est franchie. Ce mécanisme rend l'exécution de l'automate déterministe. Les indices de priorité sont des notations pratiques qui peuvent être éliminées comme suit : chaque transition (s_1, s_2, F, G, P) peut être remplacée par (s_1, s_2, F', G) où $F' = F \wedge \neg(F_1 \vee \ldots \vee F_n)$ telle que $F_1, \ldots, F_n$ sont les conditions des transitions sortantes de l'état s_1 qui ont un indice de priorité strictement inférieur à P.

Chaque état a une transition implicite vers lui-même. Cette transition associe la valeur "`faux`" à toutes les valeurs de sortie et n'est franchie que si aucune autre transition ne peut l'être (son indice de priorité est $+\infty$). Ce genre de transition est classique dans les langages synchrones, car un programme synchrone doit fournir une réaction à chaque itération, même si les entrées reçues à cette itération ne déclenchent aucune réaction observable.

Un exemple d'automate Sam est fourni à la figure 4.2. Par convention, un point d'interrogation précède la condition F de chaque transition tandis qu'un point d'exclamation précède la liste G de variables de sortie auxquelles la valeur "`vrai`" doit être affectée. Les indices de priorité sont situés à la source des transitions.

La composition de programmes Sam suit la sémantique classique de la composition des programmes synchrones. Les communications entre les différents programmes sont exprimées par la connexion graphique de ports de sortie et de ports d'entrée, en respectant les règles suivantes :

- Les ports d'entrée d'une composition peuvent être connectés aux ports de sortie de la composition ou bien aux ports d'entrée des sous-programmes (c'est-à-dire, les programmes qui participent à la composition).
- Les ports de sortie d'un sous-programme peuvent être connectés aux ports d'entrée d'autres sous-programmes ou bien aux ports de sortie de la composition.

- Les dépendances cycliques sont interdites : il est interdit de connecter le port de sortie d'un sous-programme au port d'entrée du même sous-programme, que ce soit directement ou par transitivité (c'est-à-dire au moyen d'un ou plusieurs sous-programmes intermédiaires).

4.6 Traduction de SAM en LOTOS NT

Les automates Sam étant des machines de Mealy étendues (avec des indices de priorité dont nous avons montré qu'ils pouvaient être facilement éliminés), nous avons choisi la première méthode d'encodage (parmi celles présentées à la section 4.2), qui consiste à traduire directement les automates Sam en fonctions et types Lotos NT. Cette traduction étant naturelle, nous nous contentons de l'illustrer sur l'exemple de la figure 4.2 :

```
type State is
  S0, S1, S2
end type

function Transition (in CurrentState:State,
                     in A:Bool,
                     in B:Bool
                     out NextState:State,
                     out C:Bool,
                     out D:Bool)
is
  NextState := CurrentState;
  C := false;
  D := false;
  case CurrentState in
  S0 ->
    if A then
      NextState := S1; D := true
    end if
  | S1 ->
    if A and B then
      NextState := S0; C := true; D := true
    elsif B then
      NextState := S2; C := true
    endif
  | S2 ->
    if A and not (B) then
      NextState := S0; C := true
    elsif B then
      NextState := S0; D := true
    end if
  end case
end function
```

où `State` est un type énuméré à trois valeurs.

De même, un programme synchrone Sam *complexe* se traduit aisément en Lotos NT. Comme les dépendances cycliques sont interdites, il est possible d'effectuer un tri topologique des sous-programmes en fonction de leurs dépendances les uns aux autres. A partir de l'ordre obtenu par ce tri, un programme synchrone Sam *complexe* peut être encodé en Lotos NT comme la composition séquentielle des fonctions de Mealy de ses sous-programmes, c'est-à-dire en appelant les fonctions

de Mealy des sous-programmes dans l'ordre induit par le tri, de telle sorte que lors de l'appel de la fonction de Mealy d'un sous-programme donné, les valeurs de toutes ses variables d'entrée sont connues.

L'exemple de traduction illustre bien les raisons qui nous ont conduits à choisir LOTOS NT plutôt que LOTOS pour traiter notre étude de cas. En effet, le style fonctionnel/impératif de LOTOS NT rend la manipulation des données (et notamment la définition des fonctions) plus simple, plus concise et plus naturelle que dans le style algébrique de LOTOS. En particulier :

- LOTOS ne possède pas les constructions "`if`" et "`case`" de LOTOS NT qui doivent alors être exprimées à l'aide d'équations conditionnelles.
- Les variables LOTOS NT sont modifiables (voir l'utilisation des variables "`NextState`", "`C`" et "`D`" dans l'exemple ci-dessus) ; cela nous permet d'avoir une traduction purement syntaxique pour encoder les transitions des automates SAM en LOTOS NT. La même traduction vers LOTOS aurait été plus verbeuse, en raison de la nécessité de spécifier pour chaque transition les nouvelles valeurs de toutes les variables de sortie.
- Contrairement à LOTOS NT dont les fonctions peuvent avoir plusieurs paramètres de sortie, les fonctions de LOTOS (appelées aussi *opérations*) ne peuvent renvoyer qu'une unique valeur : cela nous aurait contraint à déclarer un type auxiliaire regroupant tous les paramètres de sortie de la fonction "`Transition`" au sein d'un unique constructeur, ainsi que des fonctions auxiliaires pour manipuler ce type et les équations algébriques définissant ces fonctions.

Le choix de LOTOS NT comme langage cible permet de programmer élégamment les automates SAM, rendant ainsi possible et facile une traduction automatique. Pour l'étude de cas industrielle présentée au chapitre suivant, la traduction de SAM vers LOTOS NT a été faite manuellement. Par la suite, cette traduction a été automatisée en utilisant le générateur de code ACCELEO[33] de la plateforme Eclipse (150 lignes de code ACCELEO et 120 lignes de code Java). De plus, nos travaux se sont déroulés conjointement avec le développement, au sein de l'équipe VASY de l'INRIA, d'un traducteur automatique de LOTOS NT vers LOTOS [CCG+10]. Les programmes LOTOS NT générés depuis les automates SAM durant nos travaux ont fortement contribué à la mise au point et à l'amélioration de ce traducteur.

[33] `http://www.acceleo.org`

Chapitre 5

Application à une étude de cas industrielle

Dans ce chapitre, nous mettons en œuvre la méthodologie proposée au chapitre 4 sur un exemple concret. A la section 5.1 nous décrivons cet exemple concret avant d'expliquer sa modélisation en LOTOS NT à la section 5.2. A la section 5.3, nous détaillons les différentes propriétés de bon fonctionnement que cet exemple doit satisfaire et donnons leur expression dans les logiques temporelles de CADP. Enfin, nous présentons les résultats obtenus, par la vérification formelle à la section 5.4 et par simulation à la section 5.5.

5.1 Description de l'étude de cas

Cette étude de cas a été distribuée par Airbus aux participants du projet TOPCASED pour illustrer un système embarqué avionique typique. Dans cette section, nous commençons par présenter les principes du protocole TFTP avant de décrire les changements effectués sur ce protocole par Airbus pour permettre la communication entre un avion et le sol.

5.1.1 Protocole TFTP

TFTP [Sol92] est l'acronyme de *Trivial File Transfer Protocol*. Il s'agit d'un protocole Internet client/serveur grâce auquel plusieurs clients peuvent écrire (*resp.* lire) un fichier sur (*resp.* depuis) un serveur. TFTP est implémenté au dessus de la couche de transport UDP (*User Datagram Protocol*) et doit donc implémenter un mécanisme de contrôle du flux des messages afin de pallier les éventuelles erreurs de transfert (perte, permutation ou duplication de messages) non traitées par UDP. Pour permettre au serveur de différencier les clients qu'il sert, chaque transfert de fichier s'effectue sur un port UDP différent.

Lors d'une session TFTP typique, un client démarre un transfert de fichier en envoyant, soit une demande de lecture au moyen d'un message `RRQ` (*Read ReQuest*), soit une demande d'écriture au moyen d'un message `WRQ` (*Write ReQuest*). Pour les besoins du transfert, le fichier est divisé en fragments de même taille, sauf dans le cas du dernier fragment qui est nécessairement plus petit. Si la taille du fichier à transférer est un multiple de la taille des fragments, il faut tout de même envoyer un dernier fragment vide pour signifier la fin du transfert. Ces fragments sont envoyés de

façon séquentielle et le protocole reconnaît le dernier fragment du fichier à sa taille qui est différente de celle des fragments le précédant. Le serveur répond à une demande de lecture en envoyant au client un message `DATA` contenant le premier fragment du fichier et l'indice 1. Le client répond en envoyant, à son tour, un message `ACK` (acquittement) contenant l'indice 1. Cet échange se poursuit jusqu'à l'envoi du dernier fragment du fichier suivi de la réception de l'acquittement correspondant. Le serveur répond à une demande d'écriture en envoyant au client un message `ACK` contenant l'indice 0. Le client répond en envoyant un message `DATA` contenant le premier fragment du fichier et l'indice 1. L'échange continue alors en suivant le même schéma que pour la lecture d'un fichier, sauf que les rôles du client et du serveur sont alors inversés.

Le protocole est robuste : un message perdu (`RRQ`, `WRQ`, `DATA` ou `ACK`) peut être retransmis après un délai (*timeout*). Les acquittements dupliqués (renvoyés à cause d'un *timeout* par exemple) doivent être ignorés afin d'éviter le problème connu sous le nom de *l'apprenti sorcier* [Bra89]. La norme Tftp suggère l'attente pour gérer le cas de l'acquittement final : l'entité (client ou serveur) qui envoie le dernier acquittement n'a aucun moyen de savoir si ce message a été reçu par l'autre entité. Dans ce cas, la norme préconise que l'entité qui envoie les acquittements attende, après l'envoi du dernier acquittement, une durée équivalente à deux *timeouts* avant de considérer le transfert comme terminé ; si durant cette période, le dernier fragment de fichier est reçu de nouveau, alors le cycle d'attente recommence.

Lorsqu'une erreur (épuisement de la mémoire disponible, erreur du système, etc.) se produit dans une entité, celle-ci doit envoyer un message `ERROR` pour annuler le transfert.

5.1.2 Variante Airbus du protocole TFTP

Lorsqu'un avion atteint sa position finale dans l'aéroport, il est connecté au réseau informatique de cet aéroport. A l'heure actuelle, les communications qui se déroulent entre l'avion et les serveurs de l'aéroport sont régies par un protocole de communication très simple et certifié correct. Airbus nous a demandé d'étudier un protocole de remplacement plus complexe, une variante du protocole Tftp, qui pourrait être d'intérêt pour de nouvelles générations d'avions. Les principales différences entre ce protocole et le protocole Tftp classique sont :

- Dans la pile de protocoles considérée par Airbus, la variante du protocole Tftp repose toujours sur le protocole Udp pour la transmission des messages. Cependant, ce ne sont plus des fichiers qui sont transportés mais les trames d'un protocole de communication de plus haut niveau dédié à l'avionique (comme Arinc 615a).

- Chaque entité communicante de cette variante du protocole Tftp a la faculté d'être à la fois client ou serveur (on parle alors de protocole *full duplex*), selon ce que requiert le protocole de communication de plus haut niveau.

- Chaque entité ne communique qu'avec une seule autre entité. En effet, pour chaque avion qui se connecte, il y a dans les serveurs de l'aéroport une entité Tftp qui lui est réservée. Cela nous permet de ne pas modéliser le fait qu'une entité puisse transférer plusieurs fichiers simultanément sur des ports Udp différents.

Dans le reste de ce chapitre, l'abréviation Tftp désigne (sauf mention contraire) la variante du protocole Tftp définie par Airbus.

Les entités Tftp ont été spécifiées par Airbus au moyen d'un automate Sam à 7 états, 39 transitions, 15 ports d'entrée et 11 ports de sortie, que nous désignons par *automate* Tftp Sam dans la suite de ce chapitre. Cet automate ne comporte que des ports d'entrée et de sortie booléens et ne traite donc

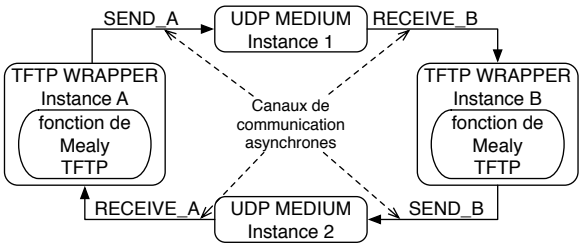

Figure 5.1: Connexion asynchrone de deux processus TFTP via deux média UDP

que de la partie contrôle du protocole TFTP. Les données sont donc abstraite : aussi, la réception d'un message `DATA` correspond à plusieurs ports d'entrée :

- `receive_DATA` : réception d'un nouveau message `DATA`,
- `receive_old_DATA` : réception du dernier message `DATA` acquitté et
- `DATA_length_lt_512` : indication que le message `DATA` reçu contient le dernier fragment du fichier en cours de transfert (512 octets était la longueur par défaut des fragments de fichiers dans la première version de la norme TFTP, cette contrainte ayant été relachée depuis).

5.2 Modélisation en LOTOS NT

Airbus était intéressé par l'étude du comportement de l'automate TFTP SAM dans un environnement fortement asynchrone. Il nous a donc été demandé de modéliser deux entités TFTP (dont le comportement est régi par l'automate TFTP SAM) communiquant par un médium non sûr comme UDP (c'est-à-dire avec des pertes, des duplications et des permutations de messages).

Nous avons donc modélisé une spécification qui comporte deux entités TFTP connectées par deux média UDP. Comme illustré à la figure 5.1, les entités TFTP sont deux instances du même processus LOTOS NT qui encapsule, comme nous l'avons présenté à la section 4.3, la fonction de Mealy de l'automate TFTP SAM. Cet automate a été traduit manuellement en 215 lignes de code LOTOS NT (ce nombre inclut la fonction de Mealy et le type énuméré qui encode les états). Les média sont deux instances du même processus LOTOS NT qui reproduit les propriétés (perte, duplication et permutation de messages) du protocole de transport UDP ; en effet, le protocole UDP étant symétrique, nous pouvons le modéliser par deux processus identiques, comme expliqué à la section 4.4.

Afin d'illustrer nos propos de la section 4.3, nous avons défini deux versions de la *coquille* LOTOS NT qui encapsule la fonction de Mealy de l'automate TFTP SAM. Le premier que nous appelons "processus TFTP *simplifié*" est très simple : il suit les recommandations d'Airbus de connecter deux automates TFTP SAM dans un environnement asynchrone. Comme nous le verrons à la section suivante, cette modélisation est trop rudimentaire et nous avons dû créer une nouvelle *coquille*, plus complexe, que nous appelons "processus TFTP *réaliste*".

5.2.1 Modélisation d'entités TFTP simplifiées

Le processus TFTP *simplifié* (voir figure 5.2) est très simple (environ 260 lignes de code en LOTOS NT). Il transmet directement les valeurs reçues de l'environnement (extraites d'un message reçu sur la

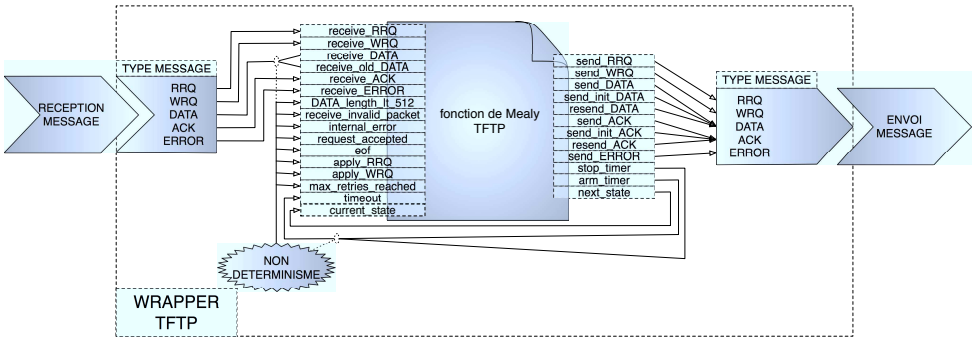

Figure 5.2: Schéma du processus Tftp *simplifié*

porte `RECEIVE`) aux entrées correspondantes de la fonction de Mealy de l'automate Tftp Sam ; d'autres entrées, telles que `internal_error`, ne peuvent être déterminées par la coquille et se voient affecter une valeur non-déterministe ; enfin la valeur de l'état courant est égale à celle calculée par la fonction de Mealy à l'itération précédente. Certaines valeurs de sortie de la fonction de Mealy sont directement renvoyées à l'environnement (sous la forme d'un message émis sur la porte `SEND`), tandis que `arm_timer` et `stop_timer` sont sauvegardées pour être utilisées à l'itération suivante pour décider si un *timeout* peut avoir lieu. Tout comme l'automate Tftp Sam, le processus Tftp *simplifié* ne se préoccupe pas des données du protocole Tftp (numéro de fragment de fichier, taille du fragment...). Les messages échangés entre le processus Tftp *simplifié* et l'environnement sont les messages Tftp standard, mais sans données associées.

Ce manque de prise en compte des données rend parfois le processus Tftp simplifié non-déterministe. Par exemple, les valeurs des deux ports d'entrée `receive_DATA` et `receive_old_DATA` dépendent de la réception d'un message `DATA`. Or, lors de la réception d'un tel message, l'absence de données ne nous permet pas de déterminer si cette réception de message correspond à la réception d'un nouveau message ou bien à la réception d'un message déjà reçu. Cela nous conduit à affecter, de façon non-déterministe la valeur "`vrai`" soit à `receive_DATA`, soit à `receive_old_DATA`. Pour résoudre ce problème, nous rajoutons dans la liste des messages Tftp un message `OLD_DATA` de telle sorte que les sorties `send_DATA` et `resend_DATA`, dans une entité Tftp, soient respectivement connectées aux entrées `receive_DATA` et `receive_old_DATA` dans l'autre entité. Pourtant, en procédant ainsi, le problème ne disparaît pas et n'est que déplacé. En effet, si l'une des entités envoie un message `DATA` (`send_DATA`) et que ce message est perdu par le médium, alors cette entité va renvoyer le message (en tant que `OLD_DATA` cette fois-ci) après un *timeout*. Dans l'autre entité, le premier message `DATA` ne sera pas reçu (car perdu) et le second sera reçu en tant que `OLD_DATA`, ce qui indique à la *coquille* qu'il s'agit d'un message précédemment reçu, alors que ce n'est pas le cas en réalité.

Malgré ces approximations inhérentes à la modélisation sans prise en compte des données du protocole Tftp, le processus Tftp *simplifié* nous a permis d'obtenir des résultats de vérification, comme nous l'expliquons à la section 5.4.

5.2.2 Modélisation d'entités TFTP réalistes

Afin d'aller plus loin que le processus Tftp simplifié, nous avons étudié une version plus élaborée : le processus Tftp *réaliste*. Ce dernier reçoit et envoie de vrais messages Tftp tels que définis dans la norme. Sa définition en Lotos NT est beaucoup plus conséquente (670 lignes de code) que celle du processus Tftp *simple*.

Afin de modéliser fidèlement les messages du protocole TFTP, nous devons modéliser les fichiers et leurs fragments. Pour ce faire, nous considérons qu'à chaque entité TFTP est associé un répertoire de fichiers et que chaque entité TFTP est instanciée avec les paramètres suivants :

- une liste de fichiers, pris parmi ceux du répertoire qui lui est associé, à écrire sur l'autre entité ; nous désignons cette liste de fichiers par "liste de fichiers à écrire",
- une liste de fichiers, pris parmi ceux du répertoire associé à l'autre entité, à lire depuis l'autre entité ; nous désignons cette liste de fichiers par "liste de fichiers à lire".

Lorsqu'il n'y a pas de transfert en cours, l'une des entités peut choisir, de façon non-déterministe, un fichier parmi sa liste de fichiers à lire ou à écrire et commencer le transfert de ce fichier.

Le type de données que nous utilisons pour modéliser les fichiers est une liste de fragments (dans notre modèle, le fichier est donc déjà fragmenté). Les noms des fichiers sont représentés par un entier naturel unique associé au fichier dans le répertoire qui le contient. Chaque fragment de fichier est représenté, de façon abstraite, par un caractère unique (c'est-à-dire que les caractères représentant les fragments d'un même fichier sont deux à deux différents).

Les données associées à chaque message TFTP, dans le protocole *réaliste*, sont alors les suivantes :

- `RRQ` : nom du fichier à lire,
- `WRQ` : nom du fichier à écrire,
- `DATA` : fragment de fichier, indice du fragment et valeur booléenne indiquant si le fragment est le dernier ; en effet, les fragments étant représentés par des caractères, ils ont une taille identique, ce qui rend nécessaire l'introduction d'une valeur supplémentaire pour indiquer la réception du dernier fragment.
- `ACK` : indice du fragment de données acquitté, ou 0 s'il s'agit de l'acquittement d'un message `WRQ`.
- `ERROR` : aucune donnée associée.

En plus de l'état courant de l'automate TFTP SAM, d'autres valeurs doivent être sauvegardées entre deux itérations, notamment le nom du fichier en cours de transfert, l'indice du dernier fragment (ou acquittement) reçu ou envoyé, le nombre de renvois du dernier message...

Les listes de fichiers et le contenu de chaque fichier sont des paramètres modifiables auxquels s'ajoute la possibilité de spécifier le nombre maximal de renvois des messages. Ces paramètres nous permettront d'explorer différents scénarios dans la section 5.4. En jouant sur les valeurs de ces paramètres, nous pourrons aussi contrôler, dans une certaine mesure, la taille de l'espace d'états de notre spécification.

5.2.3 Modélisation des liens de communication

Conformément aux principes énoncés à la section 4.4, les deux processus LOTOS NT décrivant les média UDP n'ont pas été dérivés d'une spécification SAM mais écrits directement, par nos soins, en LOTOS NT.

Ces processus reproduisent de façon précise la couche de transport UDP mise en œuvre dans le réseau informatique reliant le sol et l'avion. UDP est un protocole dit *non connecté*, c'est-à-dire que chaque message est envoyé sans que les mécanismes du protocole ne permettent de déterminer s'il a bien été reçu. Ce protocole ne détecte pas, ni ne répare, les erreurs survenant dans les communications. Ces erreurs, lorsqu'elles se produisent, doivent donc être gérées par les applications de la couche supérieure

qui utilisent le protocole UDP pour communiquer (c'est-à-dire, dans notre cas, les entités TFTP). Ces erreurs peuvent être de trois types :

- pertes de message, causées par un problème physique sur le réseau ou la congestion d'un routeur ;
- permutations de messages, qui se produisent lorsqu'un routeur envoie les messages d'un même flux sur différentes routes pour des raisons de répartition de charge ;
- duplications de messages, qui peuvent se produire dans des cas très particuliers, par exemple si les couches réseau de bas niveau sont défectueuses ou bien si un routeur dit *store-and-forward* tombe en panne et qu'il est remis en service depuis un état antérieur dans lequel plusieurs messages n'avaient pas encore été envoyés.

Nous avons choisi de modéliser le médium UDP de deux façons différentes, au moyen de deux processus LOTOS NT différents, afin de nous assurer que les entités TFTP se comportent correctement, indépendamment du médium choisi. Ces deux processus LOTOS NT peuvent perdre les messages et possèdent une mémoire tampon dans laquelle les messages reçus non perdus sont enregistrés dans l'attente de leur acheminement. Nous ne modélisons pas explicitement les duplications de messages causées par le médium UDP, car chaque entité TFTP peut déjà renvoyer un même message un nombre borné de fois (borne qui peut d'ailleurs être différente pour chaque entité TFTP).

Le premier processus modélise le cas où les permutations de messages ne se produisent pas. Il utilise une file FIFO comme mémoire tampon : les messages sont acheminés dans le même ordre que celui dans lequel ils arrivent. Le code LOTOS NT pour ce médium dit "FIFO", dont la taille de la mémoire tampon est notée `MAX`, est donné ci-dessous :

```
process FIFO [RECEIVE, SEND:any] (MAX:NAT) is
  var Q:QUEUE, M:MESSAGE in
    Q := EMPTY_QUEUE;
    loop
      select
        RECEIVE (?M);
        select
          if (SIZE (Q) < MAX) then
            Q := ENQUEUE (M, Q)
          end if
        []
          null   (* perte du message *)
        end select;
        i   (* transition "tau" *)
      []
        if (SIZE (Q) > 0) then
          SEND (!FIRST (Q));
          Q := DEQUEUE (Q)
        end if
      end select
    end loop
  end var
end process
```

Le second processus modélise le cas où les permutations de messages se produisent. Il utilise un *bag* (multi-ensemble non trié) comme mémoire tampon. Le code LOTOS NT de ce processus est obtenu à partir du code du médium "FIFO" en remplaçant :

```
if (SIZE (Q) > 0) then
  SEND (!FIRST (Q));
  Q := DEQUEUE (Q)
end if
```

par :

```
if (SIZE (Q) > 0) then
  var N:NAT in
    (* choix non deterministe d'un message *)
    N := any NAT where (N > 0) and (N <= SIZE (Q));
    SEND (!NTH (N, Q));
    Q := REMOVE (N, Q)
  end var
end if
```

Dans la suite de ce chapitre, nous notons *FIFO(n)* (*resp. BAG(n)*) un médium "Fifo" (*resp.* "bag") dont la mémoire tampon a une taille de n. Il convient de noter que *FIFO(1)* et *BAG(1)* sont identiques.

5.2.4 Composition parallèle des liens de communication et des entités TFTP

La composition asynchrone des entités Tftp et des média Udp est illustrée par la figure 5.1. Ce schéma de composition suit la troisième méthode que nous avons présentée à la section 4.4. Un tel schéma de composition est possible, car le comportement du médium Udp est symétrique.

Dans la suite de ce chapitre, nous différencions les deux entités Tftp en les nommant A et B (cf. figure 5.1). Les portes de communication utilisées pour les synchronisations avec les média sont donc :

- `SEND_A` : porte sur laquelle l'entité A envoie des messages,
- `SEND_B` : porte sur laquelle l'entité B envoie des messages,
- `RECEIVE_A` : porte sur laquelle l'entité A reçoit des messages,
- `RECEIVE_B` : porte sur laquelle l'entité B reçoit des messages.

Comme nous avons deux *coquilles* différentes et deux média différents, nous pouvons les combiner pour obtenir quatre spécifications différentes à vérifier :

- deux processus Tftp *simplifiés* communiquant par l'intermédiaire de deux média "Fifo",
- deux processus Tftp *simplifiés* communiquant par l'intermédiaire de deux média "bag",
- deux processus Tftp *réalistes* communiquant par l'intermédiaire de deux média "Fifo",
- deux processus Tftp *réalistes* communiquant par l'intermédiaire de deux média "bag".

5.3 Description formelle des propriétés de bon fonctionnement

Dans cette section, nous commençons par présenter succinctement les deux langages de formules de logique temporelle supportés par Cadp. Ensuite, nous détaillons les différentes propriétés de bon fonctionnement que les spécifications Tftp doivent satisfaire et leur expression en tant que formules

de logique temporelle. Enfin, nous comparons ces formules à des "motifs" de propriétés logiques couramment utilisés.

5.3.1 Logique temporelle RAFMC de CADP

Le *model checker* actuel de CADP est EVALUATOR 3.6 [MS03]. Il utilise, pour l'expression des formules de logique temporelle, RAFMC[MS03] (*Regular Alternation-Free μ-Calculus*), une logique temporelle arborescente construite sur le μ-calcul modal qu'elle étend avec des expressions régulières (ou rationnelles). Nous présentons ci-après la syntaxe et la sémantique du sous-ensemble de RAFMC dont nous nous sommes servis pour l'étude de cas du TFTP.

Les formules RAFMC sont interprétées sur un STE (*Système de Transitions Etiquetées*, cf. section 2.2.1). Il existe trois sortes de formules en RAFMC :

- α: formules sur actions, évaluées sur une étiquette du STE ;
- β: formules régulières, évaluées sur une séquence de transitions du STE ;
- φ: formules sur états, évaluées sur un état du STE ;

L'axiome principal de la grammaire du langage RAFMC est la formule sur états.

Les formules sur actions ont la syntaxe suivante :

$$
\begin{array}{lcl}
\alpha & ::= & \texttt{true} \\
 & | & \texttt{false} \\
 & | & \texttt{not } \alpha \\
 & | & \alpha_1 \texttt{ or } \alpha_2 \\
 & | & \alpha_1 \texttt{ and } \alpha_2 \\
 & | & \textit{string} \\
 & | & \textit{regexp}
\end{array}
$$

Les opérateurs booléens ont la sémantique usuelle : une étiquette du STE satisfait toujours "`true`" ; elle ne satisfait jamais "`false`" ; elle satisfait "`not` α" si et seulement si elle ne satisfait pas α ; elle satisfait "α_1 `or` α_2" (*resp.* "α_1 `and` α_2") si et seulement si elle satisfait α_1 ou (*resp.* et) elle satisfait α_2. Une chaîne de caractères (*string*) est une séquence de zéro ou plusieurs caractères délimitée par des guillemets ("`"`") qui représente une étiquette du STE. Une étiquette de transition satisfait une chaîne de caractères S si et seulement si la chaîne de caractères représentant l'étiquette est identique à S. Une expression régulière (*regexp*) suit la syntaxe des expressions régulières UNIX, est délimitée par des apostrophes ("`'`") et dénote un prédicat sur les étiquettes du STE. Une étiquette satisfait une expression régulière si la chaîne de caractères représentant cette étiquette est reconnue par l'expression régulière.

RAFMC étend le μ-calcul modal avec des constructions inspirées des expressions régulières pour décrire des séquences de transitions :

$$
\begin{array}{lcl}
\beta & ::= & \alpha \\
 & | & \beta_1.\beta_2 \\
 & | & \beta^*
\end{array}
$$

Une formule régulière β dénote une séquence de transitions consécutives du STE de telle sorte que le mot obtenu par la concaténation de leurs étiquettes respectives soit reconnu par le langage régulier défini par β. Les opérateurs réguliers ont la sémantique suivante : une séquence de transitions du STE satisfait "α" si et seulement si elle est composée d'exactement une transition dont l'étiquette

satisfait la formule sur actions α ; elle satisfait "$\beta_1.\beta_2$" si et seulement si elle est la concaténation de deux séquences de transitions qui satisfont respectivement β_1 et β_2 ; elle satisfait "β^*" si et seulement si elle est la concaténation de zéro ou plusieurs séquences de transitions qui satisfont β.

Les formules sur états ont la syntaxe suivante :

$$
\begin{array}{rcl}
\varphi & ::= & \texttt{true} \\
& | & \texttt{false} \\
& | & \texttt{not } \varphi \\
& | & \varphi_1 \texttt{ or } \varphi_2 \\
& | & \varphi_1 \texttt{ and } \varphi_2 \\
& | & \langle \beta \rangle \, \varphi \\
& | & [\beta] \, \varphi \\
& | & \texttt{mu } X \texttt{ . } \varphi \\
& | & \texttt{nu } X \texttt{ . } \varphi \\
& | & X
\end{array}
$$

Les opérateurs booléens ont la sémantique usuelle : un état satisfait toujours "`true`" ; il ne satisfait jamais "`false`" ; il satisfait "`not` φ" si et seulement s'il ne satisfait pas φ ; il satisfait "φ_1 `or` φ_2" (*resp.* "φ_1 `and` φ_2") si et seulement s'il satisfait φ_1 ou (*resp.* et) il satisfait φ_2. Un état S du STE satisfait "$\langle \beta \rangle \, \varphi$" (modalité de possibilité) si et seulement s'il existe (au moins) une séquence de transitions partant de S, satisfaisant β et menant dans un état satisfaisant φ ; S satisfait "$[\beta] \, \varphi$" (modalité de nécessité) si et seulement si toutes les séquences de transitions qui partent de S et satisfont β mènent à des états satisfaisant φ. Les opérateurs de point fixe "`mu`" et "`nu`" ont la sémantique suivante : un état satisfait "`mu` X `.` φ" s'il appartient à la solution minimale de l'équation de point fixe "$X = \varphi(X)$" et il satisfait "`nu` X `.` φ" s'il appartient à la solution maximale de l'équation de point fixe "$X = \varphi(X)$, où la variable propositionnelle X dénote un ensemble d'états du STE. Intuitivement, l'opérateur de point fixe minimal (*resp.* maximal) permet la caractérisation de motifs arborescents finis (*resp.* infinis) dans le STE. Nous utilisons ces opérateurs implicitement, par le biais d'expressions régulières à l'intérieur des modalités.

5.3.2 Logique temporelle MCL de CADP

Les formules qui manipulent des données peuvent être écrites avec RAFMC mais sont souvent très verbeuses car toutes les valeurs possibles doivent être considérées. De telles formules peuvent être écrites de façon beaucoup plus concise à l'aide du langage MCL [MT08]. Par exemple, lorsqu'un type T a n valeurs, on peut avoir besoin de n formules RAFMC, une par valeur de T ; l'expérience montre que MCL permet souvent de remplacer ces n formules par une formule MCL unique, paramétrée par une variable de type T. MCL est un sur-ensemble de RAFMC qu'il étend par l'ajout de constructions pour la manipulation de variables et de valeurs. Les formules écrites dans le langage MCL peuvent être évaluées par l'outil prototype EVALUATOR 4.0, actuellement en cours d'intégration dans CADP. Nous présentons ci-après les constructions introduites par le langage MCL que nous utilisons dans le cadre de l'étude de cas TFTP. La syntaxe du langage MCL introduit un nouveau non-terminal σ (qui représente une "offre"). Les définitions des non-terminaux α, β et φ sont étendues comme suit :

$$
\begin{array}{lcl}
\sigma & ::= & !E \\
& | & ?x : T \\
& | & ?\texttt{any} \\
\alpha & ::= & \ldots \\
& | & \{G\ \sigma_1 \ldots \sigma_n\} \\
& | & \{G\ \sigma_1 \ldots \sigma_n\ \texttt{"..."}\} \\
& | & \{G\ \sigma_1 \ldots \sigma_n\ \texttt{"..."}\ \texttt{where}\ E\} \\
\beta & ::= & \ldots \\
& | & \beta\{E\} \\
& | & \texttt{if}\ \varphi\ \texttt{then}\ \beta_1\ \texttt{else}\ \beta_2\ \texttt{end if} \\
\varphi & ::= & \ldots \\
& | & \texttt{forall}\ x : T\ \texttt{among}\ \{E_1\ \texttt{"..."}\ E_2\}\ .\ \varphi
\end{array}
$$

E dénote une expression de variables. Une offre σ est évaluée sur une valeur : l'offre "$!E$" est satisfaite par une valeur V si et seulement si le résultat de l'évaluation de E est égal à V ; l'offre "$?x : T$" est satisfaite par une valeur V si et seulement si le type de V est égal à T ; dans ce cas, la variable x prend la valeur V ; l'offre "?`any`" est satisfaite par toutes les valeurs, quel que soit leur type.

Une étiquette du STE satisfait "$\{G\ \sigma_1 \ldots \sigma_n\}$" si et seulement si elle est de la forme "$G'\ !V_1 \ldots !V_n$" (une porte G' suivie de n valeurs précédées d'un point d'exclamation) et G' est égale à G, et chaque offre $\sigma_i, i \in \{1, ..., n\}$ est satisfaite par la valeur correspondante V_i. Une étiquette du STE satisfait "$\{G\ \sigma_1 \ldots \sigma_n\ \ldots\}$" si et seulement si elle est de la forme "$G'\ !V_1 \ldots !V_m$", $m \geq n$ et G' est égale à G, et chaque offre σ_i (avec $i \in \{1, ..., n\}$ est satisfaite par la valeur correspondante V_i. Une étiquette du STE satisfait "$\{G\ \sigma_1 \ldots \sigma_n\ \ldots\ \texttt{where}\ E\}$" si et seulement si elle satisfait "$\{G\ \sigma_1 \ldots \sigma_n\ \ldots\}$" et le résultat de l'évaluation de l'expression booléenne E (qui peut contenir des variables initialisées dans les offres $\sigma_1 \ldots \sigma_n$) donne la valeur "`vrai`".

Une séquence de transitions du STE satisfait "$\beta\{E\}$" si et seulement si elle est la concaténation d'exactement E sous-séquences qui satisfont β ; elle satisfait "$\texttt{if}\ \varphi\ \texttt{then}\ \beta_1\ \texttt{else}\ \beta_2\ \texttt{end if}$" si et seulement si elle part d'un état satisfaisant φ et satisfait β_1 ou si elle part d'un état ne satisfaisant pas φ et satisfait β_2.

Un état du STE satisfait "$\texttt{forall}\ x : T\ \texttt{among}\ \{E_1\ \ldots\ E_2\}\ .\ \varphi$" si et seulement si, pour chaque valeur de la variable x dans l'intervalle $[E_1 \ldots E_2]$, l'état satisfait φ (qui peut contenir des occurrences de la variable x).

Il est intéressant de remarquer que, si les valeurs de E_1 et E_2 sont connues lors de la compilation, l'opérateur "`forall`" peut être implémenté à l'aide d'un simple traitement syntaxique, en effectuant l'expansion de la formule φ pour chaque valeur de x. Cependant, EVALUATOR 4.0 ne procède pas de cette façon car, dans le cas général, les valeurs de E_1 et E_2 ne sont connues qu'à l'évaluation. C'est pourquoi EVALUATOR 4.0 effectue une itération sur les valeurs de l'intervalle $[E_1 ... E_2]$. Cette itération se termine lorsque toutes les valeurs contenues dans l'intervalle ont été parcourues ou bien dès que φ n'est plus satisfaite.

5.3.3 Propriétés exprimées en RAFMC

Une analyse minutieuse du protocole TFTP classique [Sol92] et des discussions avec les ingénieurs d'Airbus nous ont permis de spécifier les propriétés de bon fonctionnement que le protocole TFTP doit satisfaire. Ces propriétés ont d'abord été écrites en langage naturel avant d'être traduites en formules de logique temporelle.

Pour les spécifications TFTP construites sur le processus TFTP "*simplifié*", nous avons écrit une première collection de 12 propriétés en RAFMC. Chacune de ces propriétés consiste en deux formules

Rafmc similaires, une pour l'entité Tftp A et une pour l'entité Tftp B. Bien que les deux entités Tftp soient symétriques, ce n'est pas forcément le cas des média dont les tailles des mémoires tampon peuvent différer.

Nous listons ci-dessous les 12 propriétés et donnons, pour chacune, la formule Rafmc correspondante à satisfaire par l'entité A (les étiquettes des transitions effectuées par cette entité finissent par le suffixe "`_A`"). Pour l'entité B, il existe un second jeu de formules identiques, à la différence du suffixe qui est alors "`_B`".

Certaines de ces formules sont écrites selon un motif commun. Nous proposons une discussion à ce sujet à la section 5.3.5.

- *Propriété 01*: l'automate Tftp Sam a deux ports de sortie `arm_timer` et `stop_timer` servant à contrôler un compte à rebours qui, lorsqu'il est écoulé, indique que la réponse attendue est considérée comme perdue. La formule qui suit garantit qu'entre deux actions `stop_timer`, il doit forcément y avoir action `arm_timer`. Pour un port p de l'automate Tftp Sam nous appelons action p toute réaction (appel de la fonction de Mealy de l'automate) qui affecte la valeur "`vrai`" à p. Afin de faire apparaître ces actions dans le Ste, nous ajoutons au code Lotos NT des portes de communication qui ne sont pas utilisées pour les synchronisations et qui ont le même nom que les actions que nous voulons observer. Chaque fois que l'une de ces actions se produit, nous effectuons une communication sur la porte correspondante, ce qui a pour effet de créer une transition observable dans le Ste. En Rafmc, cela se traduit par une formule qui interdit la présence de séquences de transitions partant de l'état initial du Ste qui contiendraient deux actions `stop_timer` sans action `arm_timer` intermédiaire.

  ```
  [
    true* . "STOP_TIMER_A" .
     not ("ARM_TIMER_A")* .
    "STOP_TIMER_A"
  ] false
  ```

  Par le jeu de la précédence des opérateurs, "`*`" dans "`not ("ARM_TIMER_A")*`" s'applique à "`not ("ARM_TIMER_A")`" et non pas seulement à "`ARM_TIMER_A`".

- *Propriété 02*: entre deux actions `arm_timer`, il doit forcément y avoir soit une action `stop_timer`, soit un *timeout*, soit une réception. La formule suivante garantit cela en interdisant la présence de séquences de transitions qui contiendraient deux actions `arm_timer` sans action `stop_timer`, *timeout* ou réception intermédiaire.

  ```
  [
    true* .
    "ARM_TIMER_A" .
    not ("STOP_TIMER_A" or
         "TIMEOUT_A" or
         'RECEIVE_A.*')* .
    "ARM_TIMER_A"
  ] false
  ```

- *Propriété 03*: le compte à rebours ne doit pas être actif entre deux transferts. La formule qui suit garantit cela en interdisant la présence d'actions `ACTIVE_TIMER_BETWEEN_TRANSFERS`. Cette action correspond à une porte Lotos NT spéciale que nous utilisons pour signaler que le compte à rebours est toujours actif, alors que le transfert de fichier est terminé.

  ```
  [
    true* .
    "ACTIVE_TIMER_BETWEEN_TRANSFERS_A"
  ] false
  ```

- *Propriété 04*: un *timeout* ne doit pas se produire entre une action `stop_timer` et une action `arm_timer`. La formule qui suit garantit cela en interdisant les séquences de transitions qui contiendraient une action `stop_timer` suivie d'un *timeout* sans action `arm_timer` intermédiaire.

```
[
  true* .
  "STOP_TIMER_A" .
  not ("ARM_TIMER_A")* .
  "TIMEOUT_A"
] false
```

- *Propriété 05*: un *timeout* ne doit pas se produire avant l'envoi du premier message. La formule qui suit garantit cela en interdisant la présence de séquences de transitions dans lesquelles un *timeout* se produirait avant qu'une communication n'ait eu lieu sur la porte `SEND_A`. Dans cette formule, il n'est pas nécessaire, ni correct, d'ajouter une formule régulière `true*` en début de modalité, car nous cherchons à caractériser les séquences d'actions qui ne contiennent pas de communication sur la porte `SEND_A`, tandis que `true*` caractériserait toutes les séquences de transitions partant de l'état initial.

```
[
  not ('SEND_A.*')* .
  "TIMEOUT_A"
] false
```

- *Propriété 06*: une erreur interne doit entraîner l'arrêt du transfert. La formule qui suit garantit cela en interdisant les séquences de transitions dans lesquelles une erreur interne pourrait être suivie par la réception d'un message ou l'envoi d'un message autre qu'`ERROR` (le message utilisé pour signaler à l'autre entité qu'une erreur vient de se produire) sans qu'il y ait, entre temps, une ré-initialisation (`REINIT`) ou l'envoi d'un message d'erreur.

```
[
  true* .
  "INTERNAL_ERROR_A" .
  not ("REINIT_A" or "SEND_A !ERROR")* .
  'RECEIVE_A.*' or
    ('SEND_A.*' and not "SEND_A !ERROR")
] false
```

- *Propriété 07*: la réception d'un paquet invalide doit entraîner l'arrêt du transfert en cours. La formule qui suit est construite sur le même modèle que la formule précédente.

```
[
  true* .
  "INVALID_PACKET_A" .
  not ("REINIT_A" or "SEND_A !ERROR")* .
  'RECEIVE_A.*' or
    ('SEND_A.*' and not "SEND_A !ERROR")
] false
```

- *Propriété 08*: lorsqu'une entité Tftp reçoit un message d'erreur (`ERROR`), elle doit arrêter le transfert en cours et non pas renvoyer un message d'erreur. La formule suivante interdit la présence de séquences de transitions dans lesquelles l'entité Tftp répondrait à la réception d'un message d'erreur en envoyant un message d'erreur.

```
[
  true* .
  "RECEIVE_A !ERROR" .
  not ('.*_A.*')* .
  "SEND_A !ERROR"
```

```
] false
```

En fait, nous avons utilisé une formule plus simple :

```
[
  true* .
  "RECEIVE_A !ERROR" .
  "SEND_A !ERROR"
] false
```

qui, dans le cadre de l'étude de cas du TFTP, est équivalente à la première dont la formule régulière est plus générale et inclut la séquence de transitions remarquable dans laquelle "RECEIVE_A !ERROR" est directement suivi de "SEND_A !ERROR". Réciproquement, la seconde formule implique la première, car toutes les actions qui ne sont pas effectuées par l'entité A et qui se produisent entre "SEND_A !ERROR" et "RECEIVE_A !ERROR" sont des actions concurrentes qui résultent de la sémantique d'entrelacement et qui peuvent être ignorées pour l'évaluation de cette formule en raison de considérations d'ordres partiels.

- *Propriétés 09a et 09b*: si les deux entités TFTP essaient d'initier un transfert simultanément, elles doivent abandonner après réception de la demande de transfert de l'autre entité. La propriété *09a* exprime le cas où une demande de transfert est reçue après l'envoi d'une requête de lecture (RRQ) tandis que la propriété 09b exprime le cas où une demande de transfert est reçue après l'envoi d'une requête d'écriture (WRQ). La formule qui suit garantit la propriété *09a* en interdisant la présence de séquences de transitions dans lesquelles l'envoi d'une demande de lecture (RRQ) suivi de la réception d'une demande de transfert (RRQ ou WRQ) pourrait précéder l'envoi d'un message sans une ré-initialisation intermédiaire. Il existe une formule similaire pour la propriété *09b* dans laquelle les occurrences de "RRQ" et de "WRQ" sont interverties.

```
[
  true* .
  'SEND_A !RRQ.*' .
  true . (* ARM_TIMER_A *)
  'RECEIVE_A !RRQ.*' or
    'RECEIVE_A !WRQ.*' .
  not ("REINIT_A")* .
  'SEND_A.*'
] false
```

- *Propriété 10*: une entité TFTP ne doit pas passer de l'envoi de fragments de fichiers (DATA) à l'envoi d'acquittements (ACK) sans avoir reçu une demande d'écriture (WRQ) ni envoyé une demande de lecture (RRQ). La formule qui suit garantit cela en interdisant les séquences de transitions dans lesquelles l'envoi d'un message DATA serait suivi par l'envoi d'un message ACK sans qu'il y ait eu un envoi de demande de lecture ou la réception d'une demande d'écriture entre temps.

```
[
  true* .
  'SEND_A !DATA.*' .
  not ('SEND_A !RRQ.*' or
       'RECEIVE_A !WRQ.*')* .
  'SEND_A !ACK.*'
] false
```

- *Propriété 11*: une entité TFTP ne doit pas passer de l'envoi d'acquittements (ACK) à l'envoi de fragments de fichiers (DATA) sans avoir reçu une demande de lecture (RRQ) ni envoyé une demande d'écriture (WRQ). La formule qui suit est construite sur un modèle similaire à celui de la formule précédente.

```
[
  true* .
  'SEND_A !ACK.*' .
  not ('SEND_A !WRQ.*'  or
       'RECEIVE_A !RRQ.*')* .
  'SEND_A !DATA.*'
] false
```

5.3.4 Propriétés exprimées en MCL

Pour les spécifications TFTP construites sur le processus TFTP "*réaliste*", la première collection de propriétés que nous avons écrite peut être réutilisée sans la moindre modification en utilisant l'outil EVALUATOR 3.6 pour évaluer les formules. Pour aller plus loin et afin de capturer les messages échangés entre les deux entités TFTP, nous avons écrit une seconde collection de 17 propriétés qui sont exprimées dans le langage MCL.

En ce qui concerne les considérations de symétrie, les remarques concernant la dissymétrie des média sont toujours valides. Avec les processus TFTP *réalistes*, les entités TFTP deviennent elles aussi dissymétriques, car elles peuvent être (et le sont en général) instanciées avec des paramètres différents.

Pour ces raisons, il est possible qu'une formule pour l'entité TFTP A soit vraie tandis que la formule symétrique pour l'entité TFTP B est fausse, et inversement.

Nous détaillons les 17 nouvelles propriétés ci-dessous.

- *Propriété 12*: Il doit être possible de terminer un transfert avec succès. Il est intéressant de constater que la formule MCL qui suit (et qui n'est vérifiable que sur le protocole TFTP *réaliste*) est aussi une formule RAFMC valide, c'est-à-dire qu'elle appartient au fragment RAFMC de MCL. `SUCCESS_A` est une porte LOTOS NT que nous utilisons exclusivement pour annoncer la fin d'un transfert. La formule qui suit vérifie la présence d'une communication sur cette porte dans les transitions du LTS.

  ```
  <
    true* .
    "SUCCESS_A"
  > true
  ```

- *Propriété 13*: durant la phase d'attente suivant l'envoi du dernier acquittement, il doit être possible de commencer un nouveau transfert en cas de réception d'une demande de lecture ou d'écriture. La formule qui suit s'assure que pour chaque sous-phase de la phase d'attente (attente du premier *timeout* et attente du second *timeout*), la réception d'une demande de lecture ou d'écriture conduit au démarrage d'un nouveau transfert. L'envoi du dernier acquittement est caractérisé par la réception du dernier fragment de fichier. La première sous-phase de la phase d'attente commence immédiatement après l'envoi de l'acquittement, tandis que la seconde sous-phase commence après le premier *timeout*. Afin de vérifier la propriété pour les deux sous-phases, nous utilisons l'opérateur `forall` associé à la formule régulière β{`X`} (qui exprime la répétition X fois exactement du chemin β). Nous utilisons la modalité de possibilité (`<...>`) car certaines séquences de transitions qui mènent à la phase d'attente ne vérifient pas la propriété : c'est notamment le cas des séquences qui caractérisent l'exécution des deux entités TFTP jusqu'à l'envoi du dernier acquittement du dernier transfert entre ces deux entités : comme il ne reste plus de fichiers à transférer, aucune des deux entités ne va renvoyer de demande de transfert.

  ```
  forall X:NAT among {0 ... 1} .
    <
      true* .
      {RECEIVE_A !"DATA" ?N:NAT ?any !TRUE} .
  ```

```
    not ({SEND_A !"ACK" !N})* .
    {SEND_A !"ACK" !N} .
    (
      not (TIMEOUT_A or REINIT_A)* .
      TIMEOUT_A
    ) {X} .
    not (TIMEOUT_A or REINIT_A)*
  >
    (
      <
        {RECEIVE_A !"WRQ" ?any} .
        not {RECEIVE_A ...}* .
        {SEND_A !"ACK" !0 of Nat}
      >   true
      or
      <
        {RECEIVE_A !"RRQ" ?any} .
        not {RECEIVE_A ...}* .
        {SEND_A !"DATA" !1 of Nat ...}
      > true
    )
```

Les lecteurs attentifs auront remarqué que la contrainte exprimée par cette propriété est faible. Nous aurions souhaité pouvoir écrire une propriété plus forte qui exprime que tous les envois d'un dernier acquittement sont nécessairement suivis d'un ou deux *timeouts* avant le redémarrage d'un transfert. Malheureusement, cela n'est pas possible dans notre cas. En effet, comme nous le verrons à la section 5.4, afin de lutter contre le phénomène d'explosion de l'espace d'états, nous vérifions chaque formule sur un STE bien particulier dans lequel toutes les actions de l'autre entité TFTP (celle qui n'est pas concernée par la formule) sont cachées. Sans ces actions, il est très difficile de caractériser seulement des séquences d'actions saines (sans erreur survenue auparavant et qui reste bloquée dans le médium bag) après lesquelles le comportement attendu se produit nécessairement. Ce cas de figure se reproduit pour la plupart des formules qui suivent.

- *Propriétés 14*: Afin d'éviter le bogue de l'apprenti sorcier [Bra89], tout duplicata d'un acquittement déjà reçu doit être ignoré. La formule qui suit garantit cela en interdisant les séquences de transitions dans lesquelles l'entité TFTP répondrait à deux réceptions consécutives du même acquittement.

```
[
  true* .
  {RECEIVE_A !"ACK" ?N:NAT} .
  {SEND_A !"DATA" !N+1 ...} .
  not (REINIT_A or
      {RECEIVE_A !"ACK" !N})* .
  {RECEIVE_A !"ACK" !N} .
  {SEND_A !"DATA" !N+1 ...}
] false
```

- *Propriété 15*: une entité TFTP doit acquitter chaque réception du même fragment de fichier, dans la limite fixée par la valeur du nombre maximal de renvois. La formule qui suit garantit l'existence, pour chaque fragment de fichier, d'une séquence de transitions dans laquelle ce fragment de fichier est reçu `MIN_RETRIES_AB` fois consécutivement (c'est-à-dire que l'entité TFTP A ne doit ne faire que des actions `stop_timer_A` ou `arm_timer_A` entre temps) et acquitté autant de fois. `MIN_RETRIES_AB` est la plus petite des deux valeurs `MAX_RETRIES_A` et `MAX_RETRIES_B`,

qui dénotent respectivement le nombre maximal de renvois que les entités A et B ont le droit d'effectuer ; MIN_RETRIES_AB exprime qu'un fragment de fichier ne doit être acquitté plus de fois par une entité qu'il ne peut être envoyé par l'autre. MIN_RETRIES_AB est une macro MCL dont la valeur est automatiquement générée en fonction de la taille des fichiers échangés par les deux entités TFTP.

```
forall N:NAT among {1 ... FILE_SIZE_A()} .
  <
    true* .
    {RECEIVE_A !"DATA" !N ...} .
    (not ('.*_A.*') or '.*TIMER_A.*')* .
    {SEND_A !"ACK" !N} .
    (
      (not ('.*_A.*') or '.*TIMER_A.*')* .
      {RECEIVE_A !"DATA" !N ...} .
      (not ('.*_A.*') or '.*TIMER_A.*')* .
      {SEND_A !"ACK" !N}
    ) {MIN_RETRIES_AB ()}
  > true
```

- *Propriété 16*: une entité TFTP peut répondre à chaque réception d'une même demande de lecture, dans la limite fixée par la valeur du nombre maximal de renvois. La formule qui suit garantit l'existence, pour toutes les demandes de lecture, d'une séquence de transitions dans laquelle la demande de lecture considérée est reçue MIN_RETRIES_AB fois et le premier fragment du fichier demandé est envoyé, en conséquence, autant de fois. Contrairement à la formule précédente, nous pouvons prouver la propriété 16 pour toutes les réceptions de demandes de lecture. Dans la formule de la propriété 15, des cas d'erreurs (erreur interne, *timeout*...) survenues dans l'entité TFTP B peuvent se glisser dans la séquence de transitions qui mène à la réception d'un fragment de fichier et empêcher les renvois de ce fragment de fichier par l'entité TFTP B. La formule de la propriété 16, en revanche, capture le début d'un transfert de fichier, ce qui garantit l'absence de cas d'erreur antérieurs à la demande de lecture qui débute le transfert.

```
[
  not {RECEIVE_A !"RRQ" ...}* .
  {RECEIVE_A !"RRQ" ?N:NAT} .
  not ({RECEIVE_A ...})* .
  {SEND_A !"DATA" !1 of NAT ...}
] forall X:NAT
    among {1 ... MIN_RETRIES_AB ()} .
  <
    (
      not (REINIT_A  or
           {RECEIVE_A !"RRQ" !N})* .
      {RECEIVE_A !"RRQ" !N} .
      {SEND_A !"DATA" !1 of NAT ...}
    ) {X}
  > true
```

- *Propriété 17*: une entité TFTP doit acquitter chaque réception d'une même demande d'écriture, dans la limite fixée par la valeur du nombre maximal de renvois. La formule qui suit vérifie l'existence, pour toutes les demandes d'écriture, d'une séquence de transitions dans laquelle la demande d'écriture considérée est reçue MIN_RETRIES_AB fois et acquittée autant de fois.

```
[ not {RECEIVE_A !"WRQ" ...}* .
  {RECEIVE_A !"WRQ" ?N:NAT} .
  not ({RECEIVE_A ...}* .
```

```
  {SEND_A !"ACK" !0 of NAT}
] forall X:NAT
    among {1 ... MIN_RETRIES_AB ()} .
    < (
         not (REINIT_A or
              {RECEIVE_A !"WRQ" !N})* .
         {RECEIVE_A !"WRQ" !N} .
         {SEND_A !"ACK" !0 of NAT}
    ) {X} > true
```

- *Propriété 18*: un acquittement doit être renvoyé autant de fois que permis par la valeur du nombre maximal de renvois (c'est-à-dire MAX_RETRIES_A). La formule qui suit vérifie si, à partir du premier envoi de chaque acquittement, il existe au moins une séquence de transitions, pour chaque entier n compris entre 1 et MAX_RETRIES_A, dans laquelle l'acquittement puisse être envoyé n fois consécutivement (sans réception ni envoi de demande de lecture ou d'écriture intercalé).

```
forall N:NAT among {0 ... FILE_SIZE_A ()} .
  [
    not ({SEND_A !"ACK" !N})* .
    {SEND_A !"ACK" !N}
  ] forall X:NAT
      among {1 ... MAX_RETRIES_A ()} .
    <
      (
        not ('.*[WR]RQ.*' or
             {SEND_A !"ACK" !N})* .
        {SEND_A !"ACK" !N}
      ) {X}
    > true
```

- *Propriété 19*: un acquittement ne doit être renvoyé plus de fois que permis par la valeur du nombre maximal de renvois. La formule qui suit vérifie qu'il n'existe aucune séquence de transitions dans laquelle le même acquittement puisse être envoyé MAX_RETRIES_A + 1 fois consécutivement.

```
forall N:NAT among {0 ... FILE_SIZE_A ()} .
  [
    true* .
    {SEND_A !"ACK" !N} .
    (
      not ('.*[WR]RQ.*' or
           {SEND_A !"ACK" !N})* .
      {SEND_A !"ACK" !N}
    ) {MAX_RETRIES_A () + 1}
  ] false
```

Nous pouvons aussi exprimer cette propriété ainsi :

```
[
  true* .
  {SEND_A !"ACK" ?N:NAT} .
  (
    not ('.*[WR]RQ.*' or
         {SEND_A !"ACK" !N})* .
    {SEND_A !"ACK" !N}
  ) {MAX_RETRIES_A () + 1}
] false
```

où la construction "?N:NAT" ne sélectionne que les numéros de fragments présents dans les messages plutôt que d'itérer sur les valeurs de l'intervalle [1...FILE_SIZE_A] ; en pratique, cela rend l'évaluation de la formule 1,8 fois plus rapide.

- *Propriété 20*: un fragment de fichier doit être renvoyé autant de fois que permis par la valeur du nombre maximal de renvois. La formule qui suit vérifie qu'à partir du premier envoi de chaque fragment de fichier, il existe au moins une séquence de transitions, pour chaque entier n compris entre 1 et MAX_RETRIES_A, dans laquelle le fragment de fichier puisse être envoyé n fois consécutivement.

```
forall N:NAT among {1 ... FILE_SIZE_A ()} .
  [
    not ({SEND_A !"DATA" !N ...})* .
    {SEND_A !"DATA" !N ...}
  ] forall X:NAT among {1 ... MAX_RETRIES_A ()} .
    <
      (
        not ('.*[WR]RQ.*' or
             {SEND_A !"DATA" !N ...})* .
        {SEND_A !"DATA" !N ...}
      ) {X}
    > true
```

- *Propriété 21*: un fragment de fichier ne doit être renvoyé plus de fois que permis par la valeur du nombre maximal de renvois. La formule qui suit vérifie qu'il n'existe aucune séquence de transitions dans laquelle le même fragment de fichier puisse être envoyé MAX_RETRIES_A + 1 fois consécutivement.

```
forall N:NAT among {1 ... FILE_SIZE_A ()} .
  [
    true* .
    {SEND_A !"DATA" !N ...} .
    (
      not ('.*[WR]RQ.*' or
           {SEND_A !"DATA" !N ...})* .
      {SEND_A !"DATA" !N ...}
    ) {MAX_RETRIES_A () + 1}
  ] false
```

Comme pour la propriété 19, nous pouvons également exprimer cette propriété ainsi :

```
[
  true* .
  {SEND_A !"DATA" ?N:NAT ...} .
  (
    not ('.*[WR]RQ.*' or
         {SEND_A !"DATA" !N ...})* .
    {SEND_A !"DATA" !N ...}
  ) {MAX_RETRIES_A () + 1}
] false
```

ce qui accélère l'évaluation de la formule par 17%.

- *Propriété 22*: une demande de lecture doit être renvoyée autant de fois que permis par la valeur du nombre maximal de renvois. La formule qui suit vérifie qu'à partir du premier envoi de chaque demande de lecture, il existe au moins une séquence de transitions, pour chaque nombre n compris entre 1 et MAX_RETRIES_A, dans laquelle la demande de lecture puisse être envoyée n fois consécutivement (c'est-à-dire sans ré-initialisation intermédiaire).

```
forall X:NAT among {0 ... MAX_RETRIES_A ()} .
  [
    not ({SEND_A !"RRQ" ...})* .
    {SEND_A !"RRQ" ?N:NAT}
  ]
    <
      (
        not (REINIT_A or {SEND_A !"RRQ" !N})* .
        {SEND_A !"RRQ" !N}
      ) {X}
    > true
```

- *Propriété 23*: une demande de lecture ne doit être renvoyée plus de fois que permis par la valeur du nombre maximal de renvois. La formule qui suit vérifie qu'il n'existe aucune séquence de transitions dans laquelle la même demande de lecture puisse être envoyée MAX_RETRIES_A + 1 fois consécutivement.

```
[
  true*.
  {SEND_A !"RRQ" ?N:NAT} .
  (
    not (REINIT_A or
         {SEND_A !"RRQ" !N})* .
    {SEND_A !"RRQ" !N}
  ) {MAX_RETRIES_A () + 1}
] false
```

- *Propriété 24*: une demande d'écriture doit être renvoyée autant de fois que permis par la valeur du nombre maximal de renvois. La formule qui suit vérifie qu'à partir du premier envoi de chaque demande d'écriture, il existe au moins une séquence de transitions, pour chaque nombre n compris entre 1 et MAX_RETRIES_A, dans laquelle la demande d'écriture puisse être envoyée n fois consécutivement.

```
forall X:NAT among {0 ... MAX_RETRIES_A ()} .
  [
    not ({SEND_A !"WRQ" ...})* .
    {SEND_A !"WRQ" ?N:NAT}
  ]
    <
      (
        not (REINIT_A or
            {SEND_A !"WRQ" !N})* .
        {SEND_A !"WRQ" !N}
      ) {X}
    > true
```

- *Propriété 25*: une demande d'écriture ne doit être renvoyée plus de fois que permis par la valeur du nombre maximal de renvois. La formule qui suit vérifie qu'il n'existe aucune séquence de transitions dans laquelle la même demande d'écriture puisse être envoyée MAX_RETRIES_A + 1 fois consécutivement.

```
[
  true* .
  {SEND_A !"WRQ" ?N:NAT} .
  (
    not (REINIT_A or
         {SEND_A !"WRQ" !N})* .
```

```
      {SEND_A !"WRQ" !N}
    ) {MAX_RETRIES_A () + 1}
  ] false
```

- *Propriété 26*: Les fragments de fichiers doivent être envoyés consécutivement. La formule qui suit vérifie qu'après l'envoi d'un fragment de fichier dont l'indice est `X`, il n'existe aucune séquence de transitions qui conduise (sans passer par une ré-initialisation) à l'envoi d'un fragment de fichier dont l'indice `Y` est strictement inférieur à `X`.

```
[
  true* .
  {SEND_A !"DATA" ?X:NAT ...} .
  not (REINIT_A)* .
  {SEND_A !"DATA" ?Y:NAT ... where Y < X}
] false
```

- *Propriété 27*: entre les envois de deux fragments de fichier consécutifs, l'acquittement du premier fragment de fichier envoyé doit forcément être reçu. La formule qui suit vérifie cela en interdisant la présence de séquences de transitions dans lesquelles, entre les envois de deux fragments de fichier consécutifs (d'indices respectifs `X` et `X + 1`), ne figure pas de réception de l'acquittement d'indice `X`.

```
[
  true* .
  {SEND_A !"DATA" ?X:NAT ?any !FALSE} .
  not ({RECEIVE_A !"ACK" !X})* .
  {SEND_A !"DATA" !X + 1 ...}
] false
```

- *Propriété 28*: entre les envois de deux acquittements consécutifs, le fragment de fichier acquitté par le second acquittement doit forcément être reçu. La formule qui suit vérifie cela en interdisant la présence de séquences de transitions dans lesquelles, entre les envois de deux acquittements consécutifs (d'indices respectifs `X` et `X + 1`), ne figure pas de réception du fragment de fichier d'indice `X + 1`.

```
[
  true* .
  {SEND_A !"ACK" ?X:NAT} .
  not ({RECEIVE_A !"DATA" !X + 1 ...})* .
  {SEND_A !"ACK" !X + 1}
] false
```

5.3.5 Classification des propriétés

Il est intéressant de comparer les 27 formules RAFMC et MCL présentées ci-dessus aux classifications existantes des propriétés de logique temporelle. Nous considérons en particulier la classification établie par Dwyer et al. [DAC98][34] qui propose une collection de "motifs" de propriétés pouvant être encodés dans diverses logiques temporelles, soit du temps linéaire (comme LTL [MP92]) soit du temps arborescent (comme CTL [CES86]). Un encodage de ces "motifs" en RAFMC est aussi disponible[35]. A l'heure actuelle, l'encodage de ces "motifs" dans le langage MCL n'a pas été défini, mais nous pouvons nous inspirer de l'encodage existant pour RAFMC qui est applicable dans notre cas (RAFMC étant un sous-ensemble de MCL) y compris en présence des variables MCL et des constructions de manipulation des données.

[34] `http://patterns.projects.cis.ksu.edu`
[35] `http://vasy.inria.fr/cadp/resources/evaluator/rafmc`

Ainsi :

- Les propriétés 01, 02, 04, 06, 07, 09a, 09b, 10, 11, 14, 26, 27 et 28 sont des instances du motif "*Existence de P entre Q et R*". Ces propriétés réutilisent directement l'encodage RAFMC sus-mentionné, avec une simplification justifiée par le fait que la formule `[true*.`Q`.(not (`P`))*.`R`] false` est équivalente à `[true*.`Q`.(not (`P `or` R`))*.`R`] false`.
- Les propriétés 03, 08 et 12 sont des instances du motif "*Absence globale de P*" ; plus précisément, la propriété 03 est une affirmation de ce motif ; la propriété 12 est une négation de ce motif ; la propriété 08 est un cas spécial dans lequel P dénote la succession immédiate de deux actions.
- La propriété 05 est une instance du motif "*S précède P globalement*".
- Les propriétés 19, 21, 23 et 25 sont des instances du motif "*Existence bornée de P entre Q et R*", où P apparaît au plus `MAX_RETRIES_A + 1` ou `MAX_RETRIES_B + 1` fois, selon l'entité TFTP pour laquelle est écrite la formule.
- Nous estimons que les propriétés 16, 17, 18, 20, 22 et 24 ne peuvent être encodées dans une logique du temps linéaire (à cause de leur structure de branchement) et ainsi ne peuvent être exprimées à l'aide des motifs de propriétés (dans le cas contraire, elles pourraient être encodées en LTL). Nous remarquons que les propriétés 20, 22 et 24 pourraient être les instances d'un nouveau motif arborescent qui serait décrit comme *Exactement N occurrences possibles de P entre la première occurrence de Q et l'occurrence suivante de R* ; ce motif pourrait être encodé de la façon suivante en MCL : `[not (`Q`)*.`Q`] <(not (`P `or` R`)*.`P`) {`N`}> true`.

5.4 Vérification fonctionnelle des modèles

Dans cette section, nous abordons les difficultés liées à la génération des espaces d'états des spécifications pour l'étude de cas TFTP et nous présentons les résultats de vérification obtenus à l'aide de CADP.

5.4.1 Génération de l'espace d'états

Les spécifications LOTOS NT sont automatiquement traduites en spécifications LOTOS (par le traducteur "LOTOS NT to LOTOS" [CCG+10]) qui sont, à leur tour, compilées en STE en utilisant les compilateurs CÆSAR.ADT [GT93] et CÆSAR [Gar89] de CADP.

Un problème récurrent en *model checking* est le phénomène de l'explosion de l'espace d'états. Dans notre cas, ce phénomène peut survenir, soit durant la génération de l'espace d'états (quand le STE devient trop large pour être généré dans sa totalité), soit durant la vérification des formules de logique temporelle (quand le *model checker* épuise la mémoire disponible lors de l'évaluation d'une formule sur un STE).

Pour lutter contre ce phénomène, nous restreignons la taille de la mémoire tampon des média UDP à de petites valeurs (c'est-à-dire, $n = 1, 2, 3...$). Dans le cas du processus TFTP *réaliste*, nous limitons aussi la taille de chaque fichier à deux fragments, car nous avons observé que c'était suffisant pour exercer toutes les transitions de l'automate TFTP SAM. De plus, nous avons remarqué que l'utilisation de plus de deux fragments par fichier n'entraîne pas l'invocation de la fonction de Mealy de l'automate TFTP SAM avec un ensemble de valeurs d'entrée qui n'existait pas déjà lors de l'utilisation de seulement deux fragments. Autrement dit, utiliser plus de deux valeurs n'apporte rien de plus dans l'exhaustivité de

Médium	Etats	Transitions	Temps de génération	Temps de vérification
BAG(1) / FIFO(1)	1,886,861	11,378,088	59 s	426 s
BAG(2)	82,213,578	520,362,698	3,058 s	*plus de mémoire*
FIFO(2)	76,676,294	467,009,630	2,715 s	*plus de mémoire*

Table 5.1: Temps de génération et de vérification de la spécification TFTP simplifiée en utilisant CÆSAR

la vérification. Nous contraignons aussi le nombre de fichiers échangés par les deux entités TFTP en bornant la taille des listes de fichiers à lire et à écrire. Pour couvrir toutes les possibilités d'échanges, nous considérons les cinq scénarios suivants :

- Scénario A: l'entité TFTP A écrit un fichier ;
- Scénario B: l'entité TFTP A lit un fichier ;
- Scénario C: les deux entités TFTP A et B écrivent un fichier ;
- Scénario D: l'entité TFTP A écrit un fichier et l'entité TFTP B écrit un fichier en même temps ;
- Scénario E: les deux entités TFTP A et B lisent un fichier simultanément.

Pour les deux sortes de spécifications TFTP (celles basées sur le processus TFTP *simplifié* et celles basées sur le processus TFTP *réaliste*), nous avons successivement appliqué diverses techniques, de complexité et d'efficacité croissante, afin de lutter contre l'explosion de l'espace d'états.

Toutes les expériences ont été faites sur une machine équipée d'un processeur Intel Xeon Dual Core 2 GHz et de 7 Go de mémoire vive sur laquelle était installé un système d'exploitation Linux 64 bits. Les résultats des expériences ont été obtenus avec la version bêta 2008-d de CADP (datée de juillet 2009) qui est plus récente que la version de CADP utilisée pour l'article [GT09]. Un bogue corrigé dans cette version plus récente explique les différences entre les résultats de l'article [GT09] et ceux du présent chapitre.

5.4.2 Génération directe

La première technique que nous avons utilisée a consisté à générer directement les STEs avec le compilateur CÆSAR de CADP.

La table 5.1 donne les temps de génération et de vérification, tout en montrant l'influence des média utilisés sur la taille de l'espace d'états, pour la spécification TFTP construite à partir du processus TFTP *simplifié* et que nous appellerons désormais "spécification TFTP simplifiée". La colonne *Temps de génération* donne le temps cumulé nécessaire à la traduction du code LOTOS NT en LOTOS et à la génération du STE à partir de ce code LOTOS. La colonne *Temps de vérification* donne le temps nécessaire à la vérification, sur ce STE, des 2×12 formules RAFMC de la première collection. Nous rappelons que chaque propriété a deux formules lui correspondant, une par entité TFTP.

La table 5.2 donne les temps de génération et de vérification du scénario D de la spécification TFTP construite à partir du processus TFTP *réaliste* que nous appellerons désormais "spécification TFTP réaliste". La taille des fichiers est de deux fragments et le nombre maximal de renvois est fixée à 1 pour chaque entité. Ce scénario D sera utilisé tout au long de cette section pour illustrer la génération des spécifications TFTP réalistes. Nous avons choisi le scénario D car c'est celui pour lequel nous avons observé les STEs avec le plus d'états et de transitions. La colonne *Temps de génération* a la même signification que pour la table 5.1. La colonne *Temps de vérification* additionne le temps nécessaire

Médium	Etats	Transitions	Temps de génération	Temps de vérification
BAG(1) / FIFO(1)	3,999,194	18,394,697	137 s	1,964 s

Table 5.2: Temps de génération et de vérification de la spécification TFTP réaliste en utilisant CÆSAR

à la vérification des 2×12 formules RAFMC de la première collection et des 2×17 formules MCL de la seconde collection.

Nous rappelons que *FIFO(n)* (*resp. BAG(n)*) dénote un médium "FIFO" (*resp.* "bag") dont la mémoire tampon a une taille de n, et que *FIFO(1)* et *BAG(1)* sont identiques.

Avec la génération directe, nous n'avons pas pu générer la spécification simplifiée avec des média de taille 3 ni les spécifications réalistes avec des média de taille 2 et 3.

5.4.3 Génération compositionnelle – niveau 1

La génération directe à l'aide de CÆSAR ne nous a pas permis de produire les STEs correspondant à une spécification simplifiée (*resp.* réaliste) avec un médium dont la taille de la mémoire tampon est supérieure à 2 (*resp.* à 1). De plus, les STEs, produits pour la spécification simplifiée avec *FIFO(2)* et *BAG(2)* sont trop larges pour être vérifiés sans qu'EVALUATOR ne consomme toute la mémoire disponible.

Afin de générer des STEs plus petits, nous nous sommes tournés vers la génération compositionnelle. Il s'agit d'une technique de génération par laquelle chaque composant est généré séparément, minimisé avant d'être composé avec les autres composants qui ont eux-mêmes été générés de cette façon.

Dans le cas des spécifications TFTP, nous avons quatre composants, deux média et deux entités TFTP. Pour chacun de ces composants, nous produisons un STE noté comme suit :

- T_a: entité TFTP A,
- T_b: entité TFTP B,
- M_a: médium sur lequel l'entité TFTP A envoie ses messages et
- M_b: médium sur lequel l'entité TFTP B envoie ses messages.

Nous utilisons les notations suivantes pour expliquer comment notre stratégie de génération compositionnelle fonctionne :

- Nous notons "*par* (x, y)" le STE résultant de la composition parallèle (avec synchronisation) de deux STEs x et y. Afin de simplifier les écritures, nous faisons abstraction des actions sur lesquelles x et y se synchronisent.
- Nous notons "*min* (x)"' le STE résultat de la minimisation du STE x modulo la relation d'équivalence de branchement [GV90]. Cette relation est la plus faible congruence implémentée dans CADP qui préserve les valeurs de vérité de toutes les propriétés listées en section 5.3.3 et section 5.3.4. En général, la bisimulation de branchement ne préserve pas forcément toutes les propriétés de vivacité, car elle supprime les boucles et les circuits de tau-transitions ; cependant, nous avons vérifié soigneuseument (et avec l'aide de Radu Mateescu) que les propriétés des sections 5.3.3 et 5.3.4 étaient préservées.

A l'aide de la génération compositionnelle, le STE L pour une spécification TFTP est obtenu par l'expression suivante :

Médium	Etats	Transitions	Temps de génération	Temps de vérification
BAG(1) / FIFO(1)	153,932	754,186	20 s	34 s
BAG(2)	3,395,977	17,505,916	47 s	809 s
BAG(3)	36,878,272	196,178,260	352 s	21,541 s
FIFO(2)	6,181,204	66,982,402	69 s	1,635 s
FIFO(3)	226,028,896	1,124,203,692	2,272 s	*plus de mémoire*

Table 5.3: Temps de génération et de vérification de la spécification TFTP simplifiée en utilisant le premier niveau de génération compositionnelle

Médium	Etats	Transitions	Temps de génération	Temps de vérification
BAG(1) / FIFO(1)	515,538	2,344,793	41 s	268 s
BAG(2)	16,687,096	83,289,158	182 s	17,707 s
BAG(3)	183,484,244	963,604,528	2,005 s	*plus de mémoire*
FIFO(2)	14,328,587	66,982,402	154 s	13,760 s
FIFO(3)	424,834,481	2,012,520,224	5,000 s	*plus de mémoire*

Table 5.4: Temps de génération et de vérification de la spécification TFTP réaliste en utilisant le premier niveau de génération compositionnelle

$$
\begin{array}{lcl}
L & = & par\ (\\
 & & \quad min\ (par\ (min\ (T_a), min\ (M_a))) \\
 & & \quad min\ (par\ (min\ (T_b), min\ (M_b))) \\
 & &)
\end{array}
$$

D'un point de vue pratique, la génération compositionnelle est rendue simple par SVL [GL01], le langage de *script* de CADP. SVL permet à l'utilisateur écrire des scénarios de génération compositionnelle à un haut niveau d'abstraction et se charge de toutes les tâches de bas niveau comme l'invocation des outils de CADP avec les options de ligne de commande appropriées, la gestion des fichiers temporaires, etc.

Les temps de génération et de vérification obtenus en utilisant la génération compositionnelle sont donnés pour la spécification TFTP simplifiée (*resp.* réaliste) en table 5.3 (*resp.* table 5.4). Comparée à l'approche directe, la génération compositionnelle est plus rapide et génère des STEs plus petits ce qui réduit les temps de vérification. De plus, cela permet de générer des spécifications TFTP avec des média dont la taille de la mémoire tampon est plus grande qu'avec la génération directe. Cependant, cela ne résoud pas tous les problèmes : nous avons toujours été confrontés à un manque de mémoire dans trois cas : *FIFO(3)* pour la spécification TFTP simplifiée, et *BAG(3)* et *FIFO(3)* pour la spécification TFTP réaliste. Les colonnes *Temps de génération* des tables 5.3 et 5.4 donnent le temps nécessaire pour traduire le code LOTOS NT en LOTOS, générer les STEs intermédiaires (T_a, T_b, M_a et M_b), minimiser ces STEs et composer les quatre STEs minimisés (*min* (T_a), *min* (T_b), *min* (M_a) et *min* (M_b)) de façon à produire le STE final sur lequel les formules de logique temporelle seront évaluées. Les colonnes *Temps de vérification* des tables 5.3 et 5.4 ont la même signification que pour les tables 5.1 et 5.2, respectivement.

Il se peut que l'approche par génération compositionnelle ne fonctionne pas bien, car les composants découplés ont parfois un espace d'états encore plus grand que leur produit désynchronisé[36]. Dans notre cas, cela ne se produit pas, comme l'attestent les tailles des STEs intermédiaires pour la génération de la spécification *réaliste* avec un médium FIFO de taille 3 (424 834 481 états et 2 012 520 224

[36]Nous souhaitons remercier Charles Pecheur qui a apporté cette précision lors de sa lecture d'une version préliminaire du document

Médium	Etats		Transitions		Temps de génération	Temps de vérification
	L_a	L_b	L_a	L_b		
BAG(1) / FIFO(1)	24,626	24,626	111,332	111,332	22 s	13 s
BAG(2)	682,714	682,714	3,372,898	3,372,898	32 s	120 s
BAG(3)	8,266,294	8,266,294	42,739,565	42,739,565	153 s	3,077 s
FIFO(2)	1,091,891	1,091,891	5,038,459	5,038,459	38 s	178 s
FIFO(3)	40,521,394	40,521,394	187,518,561	187,518,561	617 s	20,687 s

Table 5.5: Temps de génération et de vérification de la spécification TFTP simplifiée en utilisant le second niveau de génération compositionnelle (L_a et L_b sont des LTSs identiques à l'exception du suffixe des étiquettes de transitions qui est "_A" dans L_a et "_B" dans L_b)

transitions) :

- M_a : 436 états et 1370 transitions ;
- M_b : 733 états et 2544 transitions ;
- T_a : 209 états et 779 transitions ;
- T_b : 60 états et 186 transitions.

5.4.4 Génération compositionnelle – niveau 2

Dans le but de vérifier des spécifications générées avec des média dont la taille de la mémoire tampon est supérieure ou égale à 3, nous avons affiné notre technique de génération afin de réduire encore la taille des STEs produits.

Comme nous l'avons déjà expliqué à la section 5.3.3, les formules de logique temporelle sont divisées en deux ensembles : celles écrites pour l'entité TFTP A et celles écrites pour l'entité TFTP B. Il est possible de vérifier, sur une spécification TFTP, une formule écrite pour l'entité TFTP A (*resp.* B) en masquant toutes les actions du STE qui concernent les communications entre l'entité TFTP B (*resp.* A) et le médium vers lequel elle envoie des messages (Nous rappelons que les communications entre l'entité A et l'entité B se font à l'aide de deux média symétriques). De cette façon, nous diminuons la taille des STEs à générer puisque ces actions masquées entraînent une réduction plus importante de la taille des STEs intermédiaires.

Suivant cette idée, nous générons deux STEs pour chaque spécification TFTP (simplifiée ou réaliste), c'est-à-dire un STE pour chaque ensemble de formules à vérifier. Afin de décrire comment nous produisons ces deux STEs, nous définissons la notation "*hide* (x)" qui dénote le STE x dans lequel toutes les communications entre l'entité TFTP et le médium ont été masquées.

Ces deux STEs, que nous appelons L_a et L_b, sont obtenus comme suit :

$$\begin{aligned}
C_{a,a} &= min\ (par\ (min\ (T_a), min\ (M_a))) \\
C_{a,b} &= min\ (hide\ (par\ (min\ (T_b), min\ (M_b)))) \\
L_a &= par\ (C_{a,a}, C_{a,b}) \\
C_{b,a} &= min\ (hide\ (par\ (min\ (T_a), min\ (M_a)))) \\
C_{b,b} &= min\ (par\ (min\ (T_b), min\ (M_b))) \\
L_b &= par\ (C_{b,a}, C_{b,b})
\end{aligned}$$

Les tables 5.5 et 5.6 montrent que cette technique de génération réduit significativement la taille des STEs générés. Cependant, la table 5.6 montre que cette technique ne réussit pas à rendre possible la

Médium	Etats		Transitions		Temps de génération	Temps de vérification
	L_a	L_b	L_a	L_b		
BAG(1) / FIFO(1)	121,707	70,430	514,265	286,836	42 s	58 s
BAG(2)	5,349,051	2,238,526	25,498,095	10,060,851	298 s	2,774 s
BAG(3)	66,325,575	24,953,863	337,515,970	119,213,620	814 s	*plus de mémoire*
FIFO(2)	3,893,082	2,221,600	16,970,312	9,311,991	162 s	1,896 s
FIFO(3)	118,087,365	65,795,557	520,700,338	278,266,832	1,456 s	*plus de mémoire*

Table 5.6: Temps de génération et de vérification de la spécification TFTP réaliste en utilisant le second niveau de génération compositionnelle

vérification de la spécification TFTP réaliste avec les média *FIFO(3)* et *BAG(3)*. Les colonnes *Temps de génération* des tables 5.5 et 5.6 ont la même signification que pour les tables 5.3 et 5.4, même s'il y a, cette fois, deux STEs finaux différents (L_a et L_b). La colonne *Temps de vérification* de la table 5.5 donne le temps nécessaire à la vérification des formules RAFMC (les 12 formules écrites pour l'entité TFTP A étant vérifiées sur L_a et les 12 autres formules écrites pour l'entité TFTP B étant vérifiées sur L_b). La colonne *Temps de vérification* de la table 5.6 donne le temps nécessaire à la vérification des formules RAFMC et MCL ($12 + 17$ formules vérifiées sur L_a et $12 + 17$ autres formules vérifiées sur L_b).

5.4.5 Génération compositionnelle – niveau 3

Pour réduire davantage la taille des STEs, nous avons décidé d'introduire la notion de "groupes de formules". Cette idée consiste à regrouper les formules qui référencent le même ensemble d'actions et à générer, pour chaque groupe de formules, deux STEs (un pour chaque entité TFTP) dans lesquelles toutes les actions non nécessaires à la vérification sont masquées.

Par exemple, les formules des propriétés 01, 02 et 04 pour l'entité A (et par symétrie, les formules des mêmes propriétés pour l'entité B) sont groupées, car ces formules utilisent seulement les actions "`ARM_TIMER_A`" et "`STOP_TIMER_A`" parmi celles que l'entité A peut effectuer. En réalité, l'ensemble des actions nécessaires à la vérification des propriétés 01 et 04 pour l'entité A (`STOP_TIMER_A` et `ARM_TIMER_A`) est un sous-ensemble des actions nécessaires à la vérification de la propriété 02 pour l'entité TFTP A (`RECEIVE_A ...`, `TIMEOUT_A`, `STOP_TIMER_A` et `ARM_TIMER_A`), mais comme nous devons, quoi qu'il arrive, générer un STE pour vérifier la propriété 02 pour l'entité TFTP A, autant le réutiliser pour vérifier les propriétés 01 et 04.

La génération des deux STEs associés à un groupe de formules suit la technique décrite à la section précédente, à la différence près que, lorsque le STE $C_{a,a}$ (*resp.* $C_{b,b}$) est généré en vue de produire le STE L_a (*resp.* L_b), toutes les actions non nécessaires à la vérification du groupe de formules considéré sont masquées. Nous notons par hide_a (*resp.* hide_b) cette opération de masquage. En masquant plus d'actions que précédemment, nous obtenons des STEs $C_{a,a}$ et $C_{b,b}$ plus petits, ce qui réduit la taille des STEs L_a et L_b :

$$\begin{aligned}
C_{a,a} &= min\ (hide_a\ (par\ (min\ (T_a), min\ (M_a)))) \\
C_{a,b} &= min\ (hide\ (par\ (min\ (T_b), min\ (M_b)))) \\
L_a &= par\ (C_{a,a}, C_{a,b}) \\
C_{b,a} &= min\ (hide\ (par\ (min\ (T_a), min\ (M_a)))) \\
C_{b,b} &= min\ (hide_b\ (par\ (min\ (T_b), min\ (M_b)))) \\
L_b &= par\ (C_{b,a}, C_{b,b})
\end{aligned}$$

Une autre façon de procéder serait de composer chaque entité TFTP avec le médium par lequel elle

Médium	Moyennes		Temps de génération	Temps de vérification
	Etats	Transitions		
BAG(1) / FIFO(1)	11,447	50,295	226 s	11 s
BAG(2)	332,754	1,608,287	272 s	74 s
BAG(3)	4,215,010	21,336,829	823 s	1,542 s
FIFO(2)	506,105	2,272,416	293 s	103 s
FIFO(3)	18,791,881	84,635,823	2,698 s	11,802 s

Table 5.7: Temps de génération et de vérification de la spécification TFTP simplifiée en utilisant le troisième niveau de génération compositionnelle

Médium	Moyennes		Temps de génération	Temps de vérification
	Etats	Transitions		
BAG(1) / FIFO(1)	29,615	113,782	756 s	25 s
BAG(2)	1,035,232	4,522,643	995 s	347 s
BAG(3)	11,807,451	55,577,212	3,740 s	9,370 s
FIFO(2)	873,220	3,493,021	948 s	279 s
FIFO(3)	23,460,517	94,763,037	6,051 s	23,810 s

Table 5.8: Temps de génération et de vérification de la spécification TFTP réaliste en utilisant le troisième niveau de génération compositionnelle

reçoit des messages, plutôt qu'avec le médium vers lequel elle envoie des messages, comme nous avons fait jusqu'à présent :

$$\begin{aligned} C'_{a,a} &= min\ (hide_a\ (par\ (min\ (T_a), min\ (M_b)))) \\ C'_{a,b} &= min\ (hide\ (par\ (min\ (T_b), min\ (M_a)))) \\ L_a &= par\ (C'_{a,a}, C'_{a,b}) \\ C'_{b,a} &= min\ (hide\ (par\ (min\ (T_a), min\ (M_b)))) \\ C'_{b,b} &= min\ (hide_b\ (par\ (min\ (T_b), min\ (M_a)))) \\ L_b &= par\ (C'_{b,a}, C'_{b,b}) \end{aligned}$$

Dans certains cas, cela donne de meilleurs résultats. Par exemple, afin de vérifier la formule de la propriété 05 écrite pour l'entité TFTP A, toutes les actions effectuées par l'entité A doivent être masquées, à l'exception de `TIMEOUT_A` et `SEND_A`. Cela signifie donc que toutes les actions préfixées par `RECEIVE_A` doivent être cachées. Cependant, lorsque nous produisons le STE $C_{a,a}$ en composant T_a et M_a, les actions préfixées par `RECEIVE_A` ne peuvent être masquées, car elles sont nécessaires à la composition entre $C_{a,a}$ et $C_{a,b}$. Dans ce cas, nous utilisons alors les STEs intermédiaires $C'_{a,a}$ et $C'_{a,b}$, et $C'_{b,b}$ et $C'_{b,a}$ pour produire L_a et L_b.

Pour la mise en œuvre de la technique que nous venons de décrire, nous avons divisé les formules à vérifier en 10 groupes pour la spécification TFTP simplifiée et en 18 groupes pour la spécification TFTP réaliste. Chaque groupe de formule contient à la fois des formules écrites pour l'entité TFTP A et à vérifier sur le STE L_a, et des formules écrites pour l'entité TFTP B, à vérifier sur le STE L_b.

Les tables 5.7 et 5.8 montrent les résultats de génération et de vérification obtenus avec cette technique. Pour chaque spécification, les nombres d'états et de transitions donnés sont une valeur moyenne calculée sur tous les STEs générés pour cette spécification ($(L_a + L_b)/2$ pour chaque groupe de formules). Les colonnes *Temps de génération* des tables 5.7 et 5.8 donnent le temps qui s'est écoulé durant la génération de tous les STEs nécessaires à la vérification des spécifications (deux STEs par groupe de formules). Les colonnes *Temps de vérification* donnent le temps nécessaire pour vérifier toutes les formules de logique temporelle.

Ces tables montrent que cette dernière amélioration dans notre technique de génération des STEs permet de vérifier les spécifications TFTP réalistes avec média *FIFO(3)* et *BAG(3)* qui n'avaient pas pu l'être dans la section précédente. La consommation mémoire maximale est de 4,7 Go (sur les 7 que possède la machine de test) dans le cas de la vérification de la propriété 28 pour l'entité A avec un médium FIFO de taille 3.

En appliquant cette méthode, nous avons vérifié les cinq scénarios (A, B, C, D et E) pour la spécification TFTP réaliste. Nous avons fait varier la taille des mémoires tampons des média de 1 à 2 (*FIFO(1)*, *FIFO(2)*, *BAG(1)* et *BAG(2)*), le nombre maximal de renvois de 1 à 2, et les tailles de fichiers de 1 à 2. Sur une machine équipée d'un processeur Intel Xeon Dual Core 2 GHz et de 7 Go de mémoire vive sur laquelle était installé un système d'exploitation Linux 64 bits, il a fallu 61 heures aux outils de CADP pour générer et vérifier tous les STEs.

Un outil [CL11] a récemment vu le jour pour automatiser et améliorer le procédé que nous décrivons. Cet outil prend différents STE intermédiaires et détermine automatiquement la meilleure façon de les combiner pour obtenir un STE final minimal. De plus, il s'attache à cacher, pour chaque formule, les actions non nécessaires à sa vérification.

5.4.6 Vérification à la volée

Jusqu'à présent, nous avons effectué nos expériences en générant d'abord un STE (en appliquant éventuellement des stratégies de réduction compositionnelle) puis en vérifiant des formules de logique temporelle sur ce STE. Une approche alternative, appelée "à la volée", consiste à vérifier chaque formule en même temps que la génération du STE (pour ce faire, la génération est guidée par l'outil de vérification). La vérification à la volée est supportée dans CADP par l'environnement OPEN/CÆSAR [Gar98] et le solveur CÆSAR_SOLVE pour les systèmes d'équations booléennes [Mat06], et implémentée dans les outils BISIMULATOR, EXP.OPEN, EVALUATOR, etc.

Afin de mesurer les bénéfices potentiels de la vérification à la volée dans le contexte de l'étude de cas TFTP, nous avons refait toutes les expériences de la section 5.4.5 (troisième niveau de génération compositionnelle, c'est-à-dire les tables 5.7 et 5.8), à la fois sur la spécification TFTP simplifiée et la spécification TFTP réaliste. Il est important de noter que la vérification à la volée prend place *après* les étapes de génération compositionnelle. Cela signifie que la vérification à la volée n'inactive pas les techniques de réduction de l'espace d'états, mais au contraire bénéficie de leurs avantages. Pour toutes les expériences listées dans les tables 5.7 et 5.8, nous avons obtenu les même résultats de vérification (en termes de valeurs de vérité des formules) avec et sans vérification à la volée.

Afin de comparer le coût en temps et en mémoire de chacune des approches, nous avons légèrement modifié notre protocole expérimental. Nous avons remplacé la version bêta 2008-d de CADP (de juillet 2009) par la version bêta 2008-j (d'avril 2010), car dans cette dernière, les outils de vérification à la volée ont été considérablement améliorés : sur toutes les expériences des tables 5.7 et 5.8, la bêta-version 2008-j est 3,5% plus rapide sans activer la vérification à la volée, et 3,6 fois plus rapide en activant la vérification à la volée.

En ce qui concerne la consommation mémoire, nous obtenons les résultats suivants :

- En prenant en compte toutes les expériences, la vérification à la volée consomme 23% moins de mémoire en moyenne.
- Les gains ne sont pas uniformes : dans le pire cas, la vérification à la volée requiert 5,9 fois plus de mémoire (cela se produit dans le plus petit exemple) tandis que dans le meilleur des cas, elle divise la consommation mémoire par un facteur de 554.

En ce qui concerne le temps de calcul, nous obtenons les résultats suivants :

- En prenant en compte toutes les expériences, la vérification à la volée est 26% moins rapide (21 heures et 22 minutes contre 16 heures et 56 minutes), en comptant à la fois le temps de génération et le temps de validation.
- La plus importante perte de temps a été observée pour la vérification de la formule 11 (dans sa version écrite pour l'entité TFTP A) sur la spécification TFTP réaliste avec des média *FIFO(3)*. Dans ce cas, la vérification à la volée est 64% moins rapide (18 minutes et 43 secondes contre 11 minutes et 25 secondes).
- Pour certains exemples cependant, la vérification à la volée amène d'importants gains en temps. Le gain maximal (21 fois plus rapide) a été observé lors de la vérification de la formule 05 (dans sa version écrite pour l'entité TFTP B) sur la spécification TFTP simplifiée avec des média *FIFO(3)* (21,55 secondes contre 7 minutes et 46 secondes).
- Si l'on ne considère que les exemples sur lesquels la vérification à la volée est plus rapide (c'est-à-dire pour la spécification TFTP simplifiée : les formules 03, 05, 06, 07, 08, 09a et 09b, et pour la spécification TFTP réaliste : les formules 05, 06, 07, 08, 12, auxquelles s'ajoutent les versions pour l'entité TFTP A des formules 03, 09a, 17, 18 et 22, et les versions pour l'entité TFTP B des formules 09b, 16, 20, et 24), le gain moyen est de 3 (1 heure et 8 minutes contre 3 heures et 22 minutes).

Ces résultats peuvent être expliqués comme suit :

- La vérification à la volée est plus rapide lorsque la valeur de vérité de la formule à vérifier peut être déterminée sans explorer la totalité de l'espace d'états ; dans un tel cas, le *model checker* s'arrête dès que possible.
- La vérification à la volée est moins rapide lorsque (1) l'évaluation de la formule requiert l'exploration de la totalité de l'espace d'états, ou (2) un nombre important de formules doivent être évaluées sur le même STE ; dans de tels cas — en particulier si l'évaluation de certaines formules requiert l'exploration exhaustive de l'espace d'états — il est souvent plus judicieux de générer les deux STEs d'un groupe de formules une seule fois et de les utiliser ensuite pour vérifier toutes les formules de ce groupe.

Pour l'étude de cas TFTP, les gains apportés par la vérification à la volée ne contrebalancent pas les pertes. Pour d'autres applications, la vérification à la volée est une technique adaptée, surtout lorsque la mémoire est un facteur limitant.

5.4.7 Résultats de vérification

Plusieurs propriétés de la première collection ne sont pas satisfaites ni sur la spécification TFTP simplifiée, ni sur la spécification TFTP réaliste. Cela nous a permis de trouver 11 erreurs dans l'automate TFTP SAM. La vérification des 17 propriétés de la seconde collection a amené à la découverte de 8 nouvelles erreurs. Pour les deux types (simplifiée et réaliste) de spécifications TFTP, nous avons observé que les médias utilisés n'avaient aucune incidence sur les valeurs de vérité des formules.

Pour chaque violation de propriété, les outils EVALUATOR 3.6 et EVALUATOR 4.0 produisent un contre-exemple, qui est un fragment du STE prouvant que la propriété est fausse. Selon la formule, ce fragment peut être une séquence, un arbre ou un graphe général, y compris avec des circuits. La taille de ce fragment du STE est souvent réduite par l'utilisation d'une stratégie adaptée, par exemple un parcours en largeur [Mat06].

Entité Tftp	Propriété	Tftp simplifié	Tftp réaliste				
			A	B	C	D	E
A	03	X		X	X	X	X
	06	X	X	X	X	X	X
	07	X	X	X	X	X	X
	08	X	X	X	X	X	X
	09a	X				X	X
	09b	X			X		
	13					X	X
	15			X	X	X	X
	16						X
	17					X	X
B	03	X	X		X		X
	06	X	X		X	X	X
	07	X			X	X	X
	08	X	X	X	X	X	X
	09a	X					X
	09b	X			X	X	
	13						X
	15		X		X		X
	16			X		X	X
	17		X		X		

Table 5.9: Situations dans lesquelles les propriétés ont été prouvées fausses dans les divers scénarios

Cette information de diagnostic a été utile pour éliminer les erreurs dues à des artefacts de programmation du code Lotos NT (traduction de l'automate Tftp Sam, spécification des *coquilles* ou des média Udp), ou d'écriture des formules de logique temporelle.

Au final, nous avons identifié 19 erreurs qui existent réellement dans l'automate Sam original (c'est-à-dire, transitions erronées ou manquantes). Ces erreurs ont été confirmées par Airbus comme étant de vraies erreurs dans leur variante du Tftp.

Comme indiqué à la section 5.1.2, il est important de noter que ces erreurs ne concernent qu'un prototype d'une variante du protocole Tftp et non pas les protocoles de communications effectivement utilisés dans les avions et les aéroports.

La table 5.9 montre, pour chaque propriété prouvée fausse (certaines propriétés ont entraîné la découverte de plusieurs erreurs), quels scénarios ne la satisfont pas. Afin de résoudre les erreurs découvertes, nous avons suggéré des correctifs à appliquer à l'automate Tftp Sam.

Alors que certaines de ces erreurs auraient pu être découvertes par un ingénieur, après une étude attentive de l'automate, d'autres sont plus subtiles et auraient été difficiles à détecter sans l'aide d'outils de vérification : par exemple, le fait que, si deux entités Tftp envoient une demande de transfert (`RRQ` ou `WRQ`) en même temps, les deux requêtes vont être simplement ignorées.

5.5 Evaluation des performances par simulation

En dépit des erreurs détectées, l'automate Tftp Sam peut toujours se rétablir à l'aide de *timeouts*, c'est-à-dire, en attendant suffisamment longtemps que le compte à rebours expire. Cependant, ces *timeouts* additionnels (et les messages supplémentaires qu'ils provoquent) entraînent une dégradation des performances qui doit être quantifiée.

Il existe plusieurs techniques d'évaluation des performances : la théorie des files d'attente, les chaînes de Markov (la boîte à outils CADP fournit des outils pour les *Chaînes de Markov Interactives* [GH02]) et les méthodes basées sur la simulation. Pour l'étude de cas TFTP, nous avons choisi ces dernières, car CADP nous permet de simuler nos spécifications sans qu'il soit nécessaire de les modifier.

5.5.1 Méthodologie de simulation avec CADP

Afin de quantifier la dégradation de performances causée par les erreurs, un modèle "optimal" doit servir de référence. Dans ce but, nous avons écrit en LOTOS NT une fonction de Mealy correspondant à l'automate TFTP SAM, mais dans laquelle toutes les erreurs ont été corrigées. De plus, pour chaque erreur e, nous avons défini une nouvelle fonction de Mealy dans laquelle toutes les erreurs sauf e ont été corrigées. Ces fonctions nous servent à quantifier l'impact individuel de chaque erreur sur la dégradation globale des performances.

L'explosion de l'espace d'états est un phénomène qui ne se produit pas lors de la simulation (car nous générons une trace d'exécution parmi toutes les traces possibles). Cette propriété nous permet d'accroître la complexité de nos modèles :

- Le nombre de fichiers à échanger entre les deux entités TFTP est de 10 000. Avant chaque simulation, ces fichiers sont aléatoirement générés et distribués dans les listes de fichiers à lire et les listes de fichiers à écrire de chaque entité TFTP. La simulation s'arrête lorsque tous les fichiers ont été transférés.
- La taille des fichiers a été augmentée pour se situer entre 4 et 10 fragments. Nous considérons que les fragments ont une taille "réelle" de 32 ko chacun.
- Nous avons utilisés des média *bag* avec des mémoire tampon de taille 6.

Nous avons considéré deux scénarios de simulation qui utilisent la spécification TFTP réaliste :

1. Une entité TFTP se comporte comme un serveur et ne démarre aucun transfert, tandis que l'autre se comporte comme un client (tous les fichiers générés sont distribués sur les listes de fichiers à lire et à écrire d'une seule entité). Il s'agit d'un modèle réaliste des communications entre le sol et l'avion.
2. Les deux entités TFTP peuvent lire et écrire des fichiers. Il s'agit du plus complexe scénario possible, car les deux entités sont en compétition pour l'accès au médium. Cette situation peut se produire en cas de forte charge réseau et les ingénieurs d'Airbus ont confirmé qu'elle devait être testée.

Pour effectuer les simulations, nous avons adapté l'outil EXECUTOR de CADP à nos besoins. Cet outil est capable d'exécuter une spécification LOTOS en tirant une séquence de transitions aléatoire. Par défaut, lorsque l'outil arrive sur un état, toutes les transitions sortant de cet état ont la même probabilité d'être franchie. Afin d'avoir un modèle réaliste, dans lequel, par exemple, les *timeouts* ont une probabilité plus faible de se produire qu'un comportement normal, nous avons modifié l'outil pour affecter des probabilités aux différentes actions du STE. Nous avons donné aux *timeouts* et aux pertes de messages (*resp.* aux erreurs internes) une probabilité qui est 100 (*resp.* 10 000) fois inférieure à la probabilité des autres transitions. De plus, dans le médium *bag*, les messages en attente dans le tampon ont une probabilité d'être choisis plus grande que les messages nouvellement arrivés.

En considérant que le médium UDP a une vitesse de transfert de 1 Mo/s et une latence de 8 ms, nous avons associé à chaque transition un temps d'exécution que nous avons calculé comme suit :

- La réception ou l'envoi d'un message `RRQ`, `WRQ`, `ACK` ou `ERROR` prend 2 ms (c'est-à-dire un quart de la latence).

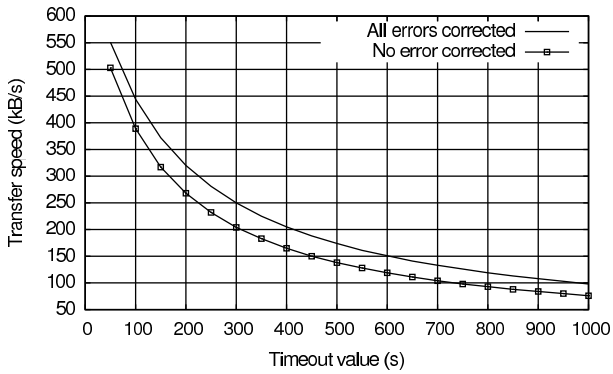

Figure 5.3: Résultats de simulation pour le scénario 1

- la réception ou l'envoi d'un fragment de fichier (`DATA`) prend 18 ms, c'est-à-dire un quart de la latence (2 ms) plus la moitié du temps nécessaire au transfert de 32 ko à la vitesse de 1 Mo/s (16 ms).
- Nous avons fait varier les valeurs de *timeouts* par pas de 50 ms entre 50 ms et 1 seconde. L'idée est de s'assurer que, dans tous les cas, il est plus rapide de corriger une erreur de transmission en respectant le protocole Tftp plutôt que d'attendre l'écoulement du compte à rebours avant de recommencer le transfert.
- Toutes les autres transitions ont un temps d'exécution estimé à 0 ms.

Pour toutes les fonctions de Mealy que nous avons définies (celle correspondant à l'automate Tftp Sam, celle dans laquelle toutes les erreurs ont été corrigées et celles dans lesquelles une seule erreur a été volontairement gardée), pour les deux scénarios de simulations et pour chaque valeur de *timeout*, nous avons lancé dix simulations. Nous avons ensuite analysé les séquences d'exécutions produites par ces simulations pour calculer :

- un temps d'exécution, c'est-à-dire, la somme des temps d'exécutions individuels des séquences ;
- un nombre d'octets transférés, c'est-à-dire, la somme des nombres d'octets transférés par chaque séquence (32 768 (32 ko) fois le nombre de fragments de fichier contenus dans les 10 000 fichiers générés).

En divisant le nombre d'octets transférés par le temps d'exécution, nous avons obtenu une vitesse de transfert que nous associons à la fonction de Mealy et à la valeur de *timeout* choisie.

5.5.2 Résultats de simulation

Pour le premier scénario de simulation, nous avons observé (voir figure 5.3) que la correction de toutes les erreurs entraîne une amélioration d'environ 10% des performances.

Pour le second scénario de simulation (voir figure 5.4), l'utilisation de la fonction de Mealy de l'automate Tftp Sam entraîne une vitesse de transfert quasiment nulle, pour toutes les valeurs de *timeout*. Cela confirme nos intuitions que les erreurs que nous avons détectées empêchent le prototype de la variante du protocole Tftp de fonctionner correctement en cas de forte charge réseau. Lorsque toutes les erreurs sont corrigées, la vitesse de transfert observée pour le second scénario est la même que pour le premier scénario. Les erreurs détectées durant la vérification n'ont pas toutes le même impact sur la dégradation des performances. Par exemple, les erreurs 13a et 13c qui caractérisent

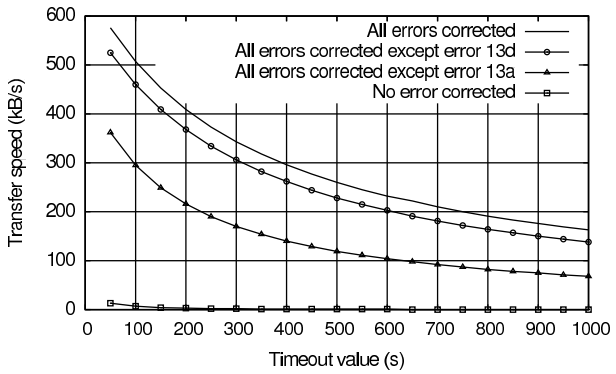

Figure 5.4: Résultats de simulation pour le scénario 2

une absence de réponse aux demandes de transfert (13a pour les demandes de lecture et 13c pour les demandes d'écriture) reçues durant la phase d'attente suivant l'envoi du dernier acquittement, ont une importance bien différente.

Partie III

Modélisation et vérification de services Web

Chapitre 6

Les services Web

La seconde partie de cete thèse est consacrée à l'étude de la vérification des services Web Bpel. Nous présentons dans ce chapitre les services Web et un état de l'art des travaux portant sur leur vérification. Dans les chapitres qui suivent, nous détaillons les différentes phases de l'algorithme de traduction de Bpel vers Lotos NT, grâce auquel des services Web Bpel peuvent être formellement vérifiés à l'aide de Cadp.

6.1 Les origines des services Web

La démocratisation des réseaux d'ordinateurs a conduit à l'interaction, de plus en plus fréquente, entre applications *distantes*. Les systèmes sur lesquels ces applications sont exécutées et les langages avec lesquels elles sont écrites ne sont pas nécessairement identiques. Par conséquent, la conception de ces applications distantes peut inclure, dans le but de les rendre interopérables :

- la définition et l'implémentation d'un protocole de communication,
- l'utilisation explicite de directives réseau pour l'échange de données avec d'autres applications.

Afin de faciliter la conception d'applications distantes, de nombreuses normes, décrivant des mécanismes d'échange de données et accompagnées d'outils, ont vu le jour. La première d'entre elles, Edi [KP96] (*Electronic Data Interchange*), est apparue en 1975, avant l'adoption massive des réseaux d'ordinateurs et a défini un format textuel pour le codage des documents électroniques. Jusqu'à la fin des années 1980, de nombreuses normes dérivées d'Edi ont été définies pour diverses branches de l'industrie : Wins (*Warehouse Information Network Standard*, 1981) pour le stockage, Ucs (*Uniform Communication Standard*, 1981) pour l'alimentation, Odette[37] pour l'industrie automobile européenne (1984).

L'arrivée des réseaux d'ordinateurs au début des années 1980 a coïncidé avec celle de la nouvelle technologie Rpc [Whi76] (première implémentation en 1981 sous Unix par Xerox), qui traitait non plus seulement du format d'échange des données mais aussi de leur acheminement. Par la suite, l'apparition d'Internet a déclenché la définition de nouvelles normes comme Corba [OMG06a] (proposé par l'Omg en 1991) et Dcom [Mic08] (proposé par Microsoft en 1996).

Toutes ces normes et technologies sont encore utilisées de nos jours, mais aucune d'entre elles n'a réussi à s'imposer comme la solution idéale auprès des concepteurs d'applications distantes. En

[37] `http://www.odette.org`

effet, leur trop forte association avec un système d'exploitation ou un langage de programmation particulier a ralenti leur diffusion. Par exemple, la technologie RPC, dont les implémentations se sont trop longtemps limitées aux systèmes UNIX et au langage C, a fini par être supplantée par CORBA. Ce dernier n'a pourtant jamais eu le succès escompté, notamment à cause du manque de soutien de Microsoft, qui a privilégié sa propre solution DCOM.

6.2 Les solutions W3C : l'approche par les services Web

L'avènement d'Internet et du *World Wide Web*, au milieu des années 1990, a permis une très large acceptation des normes HTTP [W3C99] (*HyperText Transfer Protocol*, 1997), comme protocole de transport des données à échanger, et XML [Gro04b] (*eXtensible Markup Language*, 1998), comme langage de description de ces données. Devant le succès de ces normes et l'explosion du nombre d'applications en ligne (achat, réservation, enchères, abonnements, bourse...) à la fin des années 1990, le W3C (*World Wide Web Consortium*) a défini trois nouvelles normes : WSDL [W3C01] (*Web Services Description Language*), SOAP [Gro00] (*Simple Object Access Protocole*) et UDDI [Com07] (*Universal Description Discovery and Integration*) afin d'améliorer l'interopérabilité entre les applications en ligne. Est alors apparu le terme *services Web* que le W3C définit ainsi :

> *Un "service Web" est un système logiciel conçu pour permettre des interactions entre machines via un réseau informatique. Un "service Web" a une interface décrite dans un format compréhensible par un ordinateur (spécifiquement* WSDL*). D'autres systèmes peuvent interagir avec le "service Web" en se conformant à sa description, au moyen de messages* SOAP*, typiquement acheminés par le protocole* HTTP *en utilisant un encodage* XML *des données, en conjonction avec d'autres normes relatives aux "services Web".*

WSDL permet la définition de l'interface d'un service Web, c'est-à-dire les différents liens de communications du service, les fonctions qu'il fournit sur chacun de ces liens, les messages que ces fonctions reçoivent et envoient et les structures de données (décrites à l'aide de la norme XML Schema [Gro04d, Gro04e] qui sera présentée au chapitre 7) qui composent les messages. Le chapitre 9 présentera WSDL de manière détaillée. SOAP permet de configurer la couche réseau de transport (HTTP) ainsi que de spécifier l'encodage des messages WSDL. Pour transformer une application communicante en service Web, il suffit de l'équiper avec une interface WSDL/SOAP, c'est-à-dire de remplacer ses directives de communication réseau par des appels aux bibliothèques WSDL/SOAP adaptées au langage de programmation dans lequel est écrite l'application. WSDL et SOAP sont complétées par une troisième norme, UDDI. UDDI décrit la définition et le protocole d'interrogation d'annuaires de services Web.

L'acceptation de WSDL et SOAP par les grands acteurs d'Internet (Microsoft, IBM, Oracle, Amazon, Google, etc.) repose sur deux facteurs :

- HTTP s'est imposé comme le principal protocole de transport de données et bénéficie donc d'implémentations sur tous les systèmes d'exploitation modernes. De plus, l'utilisation de HTTP comme protocole de transport permet de traverser aisément les pare-feux (au moyen des proxies HTTP) et de créer des connexions sécurisées grâce à HTTPS (HTTP *Secure*).

- XML, WSDL et SOAP ne sont liés à aucun langage de programmation particulier. Il suffit qu'une bibliothèque logicielle pour XML, WSDL et SOAP soit définie dans un langage de programmation pour que toute application communicante écrite dans ce langage puisse être transformée en service Web.

Cette acceptation massive a eu plusieurs conséquences sur les outils de développement d'applications en ligne :

- A l'heure actuelle, les principaux langages de programmation tels que C, Java, Ruby, Perl, Python, etc. possèdent tous des bibliothèques logicielles qui implémentent les normes XML, WSDL et SOAP.
- Les plateformes de développement de sites Web (JSP, Zend, ASP.NET, Ruby on Rails, etc.) proposent toutes des solutions pour encapsuler, dans des services Web, des applications écrites avec le langage qui leur est associé (Java pour JSP, PHP pour Zend, ASP pour ASP.NET, Ruby pour Ruby on Rails...).
- Les serveurs Web Apache et IIS de Microsoft, qui représentent 85% de parts de marché à eux deux[38], fournissent des implémentations pour WSDL et SOAP et peuvent, par conséquent, héberger des services Web.

Comme exemple de services Web, nous pouvons citer AWS[39], qui est une collection de services proposés par Amazon et accessibles par une interface WSDL/SOAP. Ils permettent, entre autres, d'accéder à des bases de données hébergées par Amazon, d'administrer à distance des serveurs de calculs loués chez Amazon ou encore d'utiliser l'infrastructure d'Amazon pour les paiements en ligne.

6.3 Les langages de description de services Web

Avec la multiplication des services Web est apparu un nouveau besoin : créer des services Web, non plus en équipant une application communicante de définitions WSDL/SOAP, mais en faisant interagir des services Web existants. Des services Web définis ainsi sont dit *composites*. A cet effet, de nouveaux langages dits d'*orchestration* ont vu le jour ; ils sont aux services Web ce que les interpréteurs de commandes (*shells*) sont aux programmes binaires exécutables qu'ils font interagir. Par rapport aux langages de programmation, les langages d'orchestration ont les avantages suivants :

- ils permettent la manipulation directe des structures de données et des messages définis par l'interface WSDL d'un service et
- ils possèdent des opérateurs dédiés à l'interaction de services Web existants (invocation de fonctions distantes, exécution parallèle de plusieurs branches...).

Deux précurseurs des langages d'orchestration ont été WSFL [Ley01] d'IBM et XLANG [Tha01] de Microsoft. WSFL est issu de la théorie de la gestion des *workflows* [VdAVH04] et XLANG du π-calcul [Mil99], un calcul de processus mobiles proposé par Robin Milner. Ces deux langages ont par la suite fusionné pour donner BPEL 1.1 (aussi appelé BPEL4WS) qui a ensuite évolué vers BPEL 2.0 [Com01] (aussi appelé WS-BPEL). BPEL est aujourd'hui le principal langage d'orchestration. Parmi les langages d'orchestration alternatifs, dont l'utilisation reste strictement académique, nous pouvons citer ORC [KQCM09] et JOLIE [MGLZ07]. Il est intéressant de noter que le W3C n'a pas défini son propre langage d'orchestration. Nous présenterons BPEL 2.0 en détail au chapitre 10.

Les nouveautés introduites par BPEL 2.0 ont enrichi le langage de telle sorte qu'il est devenu possible de créer des services Web non composites. En particulier, BPEL 2.0 propose désormais un mécanisme normalisé pour que les auteurs d'interpréteurs (BPEL est en général interprété) BPEL puissent ajouter de nouvelles constructions (*extensions*) au langage (accès au système de fichiers, interrogation d'une base de données...). De plus, BPEL 2.0 permet à l'utilisateur d'écrire les expressions (dans les affectations, conditions d'arrêt des boucles, condition de l'opérateur "`if`"...) dans le langage de son

[38] `http://news.netcraft.com/archives/category/web-server-survey/`
[39] `http://aws.amazon.com/`

choix (à condition que l'interpréteur Bpel accepte ce langage). Auparavant, dans Bpel 1.1, les expressions étaient écrites dans le langage XPath, que nous présentons au chapitre 8 et qui reste le langage par défaut pour les expressions dans Bpel 2.0. Dans le cadre de notre étude, comme expliqué au chapitre 10, nous ne considérons pas les extensions ni la possibilité d'utiliser d'autres langages d'expressions que XPath. Comme ces deux nouveautés de Bpel 2.0 sont les principaux changements de cette nouvelle norme, nous ne faisons pas de distinction, dans ce document, entre Bpel 1.1 et Bpel 2.0 et comparons nos travaux à ceux déjà effectués pour Bpel 1.1 et Bpel 2.0.

Au final, la norme Bpel repose sur cinq autres normes du W3C que nous nous proposons de lister :

- Http : protocole de transport des données,
- Wsdl : interface du service (structures de données et messages),
- Soap : lien entre Http et Wsdl,
- Xml Schema : définition des structures de données utilisées dans les messages Wsdl,
- XPath : langage d'expressions.

La position dominante de Bpel par rapport à ses concurrents est mise en évidence par son intégration au sein d'outils commerciaux. Nous pouvons notamment citer Oracle Bpel Manager [40] et ActiveVos[41] qui sont des suites logicielles pour, entre autres, le développement, le déploiement et le test de services Web Bpel. Les environnement de développement Ibm WebSphere et Microsoft BizTalk ne sont pas exclusivement centrés autour de Bpel mais proposent un éditeur et un interpréteur pour ce langage.

En ce qui concerne l'utilisation en vraie grandeur de Bpel, elle peut être illustrée par les quelques exemples significatifs suivants :

- La ville de Las Vegas[42], a choisi Bpel, en 2005, pour construire des logiciels pour faire fonctionner et superviser le système de traitement des eaux usées. Depuis, la ville a continué d'utiliser des processus Bpel pour d'autres usages, comme par exemple, la gestion de ses effectifs (édition des fiches de paie, affectation de postes...).
- L'Agence Spatiale Européenne[43] a lancé, en 2005, un portail européen d'intégration des ressources d'observation de la Terre qui fonctionne sur des solutions Bpel fournies par Oracle.
- L'équipe des moyens informatiques du Cern[44] a réécrit, en 2007, le système de gestion de documents du centre en utilisant Bpel et ActiveVos.

6.4 Vérification des services Web

L'utilisation croissante des services Web dans des applications à risques, telles que des solutions de paiement en ligne, soulève le problème de la vérification de ces services. La vérification des services Web définis au moyen de langages de programmation classiques puis équipés d'interfaces Wsdl/Soap se restreint habituellement au test fonctionnel, afin de s'assurer que les services répondent correctement aux requêtes qui leur sont envoyées. La vérification des services Web composites, qui font interagir différents services existants, requiert d'aller au-delà du simple test fonctionnel afin de détecter

[40] http://www.oracle.com/technetwork/middleware/bpel/overview/index.html
[41] http://www.activevos.com
[42] http://www.oracle.com/us/corporate/press/015582_EN
[43] http://earth.eo.esa.int/rtd/Articles/BPEL_ESA_01_Reseaux.pdf
[44] http://cnl.web.cern.ch/cnl/2007-04/04.htm

des erreurs de programmation propres aux services Web composites. Les techniques de vérification de ces services reposent sur les formalismes mathématiques à partir desquels les langages d'orchestration ont été définis. Par exemple, la vérification par *model checking* des services Web Bpel est rendue possible par le fait que ce langage est inspiré du π-calcul.

Nous séparons les approches de vérification des services Web composites en trois catégories, selon que ces approches traitent des données manipulées par les services Web, de leur comportement ou bien de ces deux aspects simultanément.

6.5 Vérification des données uniquement

La mise en interaction de services Web demande d'accorder un soin particulier à la façon dont les données échangées sont représentées. Des services Web qui utilisent des structures de données dont les champs portent les mêmes noms et ont le même type peuvent néanmoins donner à ces champs des interprétations différentes. Par exemple, un champ "prix" avec une valeur de type réel pourra avoir une valeur en euros pour un premier service, et en dollars pour un second.

La vérification de la sémantique des structures de données dans les compositions de services Web se ramène à un problème de vérification du typage. La difficulté de cette vérification réside dans la façon de spécifier les informations manquantes en étant le moins intrusif possible. En effet, l'idée est d'instrumenter les interfaces de services Web, sans modifier les fichiers existants (c'est-à-dire sans créer une nouvelle interface), afin d'ajouter des informations sur les types de données, avec des formalismes *adhoc* comme les ontologies. Parmi les travaux dans ce domaine, nous pouvons citer :

- Li et al. [LMZF09] qui représentent le système de typage d'un service composite au moyen d'une ontologie et équipent les interfaces Wsdl, de ce service et des services qu'il fait interagir, d'annotations qui se rapportent à des concepts de l'ontologie. Par exemple, les concepts "prix en euros" et "prix en dollars" de l'ontologie descendent du même concept "prix" et sont utilisés pour annoter déclarations de types "prix" dans les interfaces Wsdl. Grâce à un algorithme *ad-hoc*, ces annotations servent ensuite à insérer des fonctions de conversion dans le corps du service Web Bpel.

- Nagarajan et al. [NVS+06] suivent une approche similaire et annote les descriptions Wsdl des services Web au moyen de Wsdl-S [eldG07] (*Web Services Semantics*), une recommandation W3C qui définit un mécanisme normalisé pour annoter des descriptions Wsdl.

- Paolucci et al. [PKPS02] étudient la possibilité d'équiper les services Web de descriptions Daml-S [PS03] (Darpa *Agent Markup Langage for Services*) afin de publier une interface dite *sémantique* de ces services. Des services Web cherchant un service (au moyen d'Uddi par exemple) pour remplir une certaine fonctionnalité peuvent utiliser ces interfaces sémantiques afin de vérifier si un service donné implémente la fonctionnalité recherchée.

6.6 Vérification du comportement uniquement

Les travaux de vérification du bon comportement des services Web se concentrent sur le langage d'orchestration Bpel (et donc sur les services composites). Ce type de vérification a pour but de détecter des erreurs dues à une mauvaise compréhension/implémentation des protocoles de certains services, ce qui peut, dans un service qui les fait interagir, entraîner des interblocages (attente d'un message qui ne viendra jamais, par exemple).

La vérification du bon comportement des services Web composites repose principalement sur des outils existants pour la vérification des systèmes concurrents et/ou parallèles (*model checkers* par exemple). Il s'agit alors de définir une traduction depuis le langage d'orchestration considéré vers le langage d'entrée de l'outil de vérification utilisé.

Les travaux de vérification de bon comportement des services composites se concentrent sur le langage d'orchestration Bpel. En effet, Bpel a été défini par la fusion de deux langages d'orchestration : Wsfl, issu des langages de description de *workflows* et, Xlang, issu des algèbres de processus. Par conséquent, les approches pour la vérification comportementale de services Web décrits en Bpel se déclinent en deux familles :

- traduction vers des réseaux de Petri (ou une variante de ce formalisme) pour vérification avec des outils de gestion de *workflows* et
- traduction vers des algèbres de processus pour vérification par *model checking*.

On remarque aussi que tous les travaux s'abstraient de la couche de transport Http/Soap. En effet, la vérification de cette couche réseau est orthogonale à la vérification de services Web composites.

Les traductions vers les réseaux de Petri ont l'avantage de modéliser naturellement les constructions de Bpel issues de la théorie des *workflows* [VdAVH04] dans laquelle un processus est constitué d'activités parallèles dont l'ordre d'exécution est contraint par des liens de causalité entre activités. Il existe des outils, tels que Tina [BRV04], pour composer et vérifier formellement le bon comportement des réseaux de Petri. En revanche, la représentation en réseau de Petri de certaines constructions Bpel issues de Xlang est délicate. Par exemple, une séquence d'activités Bpel (c'est-à-dire une collection d'activités s'exécutant les unes après les autres) doit être traduite comme une parallélisation dans laquelle chaque activité est reliée à la suivante par un lien de causalité afin de respecter l'ordre d'exécution souhaité.

Les traductions vers des algèbres de processus, quant à elles, permettent de modéliser les données ainsi que les constructions de Bpel issues de Xlang (exécution séquentielle ou concurrente, communication asynchrone entre services). En revanche, elles rendent difficile la représentation des liens de causalité entre activités et celle des mécanismes de gestion des exceptions. La difficulté varie selon l'algèbre de processus considérée. Le grand avantage des algèbres de processus provient des techniques de vérification qui leur sont associées : vérification énumérative (*model checking*) ou vérification par équivalence (*equivalence checking*). Ces techniques permettent à leurs utilisateurs d'écrire et de vérifier la plupart des propriétés de bon comportement. De plus, un service Web Bpel se traduit par un processus communicant qui peut être vérifié soit isolément, soit en interaction avec un environnement.

Nous présentons ci-dessous, par ordre chronologique, les principaux articles traitant de la vérification du bon comportement des services Web, sans considération pour les données, qui sont abstraites. A chaque fois qu'un choix dépend de l'évaluation d'une expression ou de la valeur d'une variable (condition de l'opérateur `if`, condition d'arrêt d'une boucle...), toutes les possibilités sont considérées. Dans la section suivante, la table 6.8 compare ces articles aux approches (dont la nôtre fait partie) qui prennent en compte à la fois le comportement et les données.

- Salaün et al. [SBS04] décrivent les avantages (large panel de solutions de vérification) apportés par la définition d'une sémantique formelle, par traduction vers une algèbre de processus, pour les langages d'orchestration. Ils illustrent leur propos par une traduction de Bpel vers Ccs [Mil80].
- Koshkina et Breugel [KvB04] proposent un nouveau langage nommé Bpe-calculus qui est un sous-ensemble de Bpel. Les auteurs ont utilisé le *Process Algebra Compiler* (Pac) pour ajouter Bpe-calculus comme langage d'entrée de la boîte à outils Cwb [CLS00] (*Concurrency Work-*

bench).

- Yeung [Yeu06] propose de vérifier, à l'aide du langage Csp [Hoa78] et de la boîte à outils Fdr [Ltd05], qu'un service Bpel implémente un schéma d'interactions (ordonnancements valides pour les messages) décrit dans le langage Ws-Cdl [Gro04a] (*Web Services Choregraphy Description Language*). Le service Bpel et le schéma d'interactions sont traduits en deux processus Csp. Le service implémente le schéma d'interactions si le processus Csp qui lui correspond est inclus (par la relation d'équivalence *traces-refinement* de Fdr) dans le processus qui correspond au schéma.

- Ouyang et al. [OVvdA+07] définissent une sémantique de Bpel au moyen de réseaux de Petri et présentent l'outil Bpel2Pnml [45]) qui traduit des services Bpel dans le langage Pnml. La spécification Pnml qui résulte est alors analysée par l'outil WofBpel (présenté dans le même article) pour vérifier des propriétés statiques comme la détection des activités non atteignables.

- Qian et al. [QXW+07] améliorent le moteur d'exécution ActiveBpel de la suite logicielle ActiveVos en lui ajoutant un traducteur de Bpel vers des automates temporisés ainsi que la possibilité de faire vérifier par le *model checker* Uppaal [PL00], sur les automates produits, des formules de logique temporelle.

- [MR08] connectent Bpel à la boîte à outils Cadp en compilant les services Web Bpel (en prenant en compte les constructions liées à la gestion du temps) directement vers un système de transitions appelé : dtLts (*Discrete Time Labelled Transition System*). Ils présentent des résultats de vérification obtenus sur un service Web pour la navigation Gps.

6.7 Vérification du comportement en tenant compte des données

Dans cette section nous présentons, dans l'ordre chronologique, des travaux qui sont non pas un mélange des approches présentées dans les deux sections précédentes, mais des variantes des travaux de la section précédente dans lesquelles la vérification du bon comportement des services Web tient compte des données manipulées dans le corps des services.

- Foster et al. [FUMK03, Fos06] présentent une traduction de Bpel vers Fsp [MK06] pour vérifier formellement des services Bpel avec l'outil Ltsa. Ces travaux ont par la suite donné lieu à l'outil WS Engineer[46] qui permet d'utiliser Ltsa au sein d'Eclipse. La trop grande simplicité du système de types du langage Fsp a forcé les auteurs à se restreindre à la traduction de quelques types prédéfinis. La traduction des expressions n'est pas détaillée dans ces travaux.

- Fu et al. [FBS04a, FBS04b] définissent une traduction de Bpel vers Promela [Hol91] afin de vérifier des services Bpel avec Spin [Hol91]. La traduction est automatisée par l'outil Wsat[47]. La traduction des types est incomplète tandis que celle des des expressions quasiment complète.

- Le groupe *Theory of Programming* à Humbolt-Universität zu Berlin a publié plusieurs articles dont [HSS05, Loh08] dans lesquels est décrite (puis améliorée) une sémantique de Bpel à l'aide de réseaux de Petri dits de *haut-niveau* grâce auxquels ils peuvent transformer un service Bpel en un *open workflow net* grâce à l'outil Bpel2oWFN. Des propriétés prédéfinies sont ensuite

[45] *Petri Nets Markup Language* : `http://www.pnml.org`
[46] `http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/`
[47] `http://www.cs.ucsb.edu/~su/WSAT/`

vérifiées sur les réseaux de Petri obtenus. Les auteurs utilisent des réseaux de Petri étendus avec des variables booléennes aléatoires afin de modéliser les conditions de branchement (dans l'opérateur de branchement et les boucles). Ces auteurs et ceux de Ouyang et al. [OVvdA+07] ont comparé leurs approches dans [LVO+07].

- Fisteus et al. [FFK05] ont mis au point une plateforme de travail nommée Verbus[48] qui utilise un langage pivot nommé Cfm (*Common Formal Model*) pour la représentation des services Web. Le fonctionnement de cette plateforme comprend deux traductions depuis Bpel vers Promela et Smv [CCGR00]. Dans cette approche, les types définis dans les services Bpel sont traduits vers quatre types différents (c'est très peu) dans le langage cible : boolean, enumerated, integer et abstract (pour les types que les auteurs ne savent pas comment traduire). La traduction des expressions n'est pas mentionnée.

- Nakajima [Nak06] présente une traduction de Bpel vers un formalisme nommé Efa (*Extended Finite-state Automata*) qui étend les automates à états finis avec des variables booléennes. Ces variables sont justement utilisées pour encoder les différentes conditions qui peuvent apparaître dans un service Bpel (conditions des opérateurs `if` et conditions d'arrêt des boucles). La valeur de ces variables est décidée de façon non déterministe pour modéliser tous les cas de branchement possibles.

- Bianculli et al. [BGS07] traduisent des services Bpel vers le langage d'entrée Bir du *model checker* Bogor [DH03]. Les propriétés de bon fonctionnement sont spécifiées soit à l'aide de formules de Ltl soit dans le langage Ws-Col [BG05], un langage d'invariants pour les services Web. Une propriété exprimée à l'aide d'un invariant est injectée, sous la forme d'un appel à l'opérateur `assert`, dans le code Bir produit à partir du service Bpel. Ces auteurs prétendent traiter certains types mais ne donne pas la moindre indication quant à la façon dont ils procèdent.

- Moser et al. [MMG+07] proposent d'améliorer les approches existantes de traduction vers les réseaux de Petri au moyen de techniques d'analyses Cssa [LMP98] (*Concurrent Single Static Assignment*) des relations de dépendance entre les variables. Ces techniques sont inspirées de la représentation Ssa [CFR+91] (*Static Single Assignment*) couramment utilisée pour permettre des optimisations dans les compilateurs modernes.

6.8 Comparaison des approches de vérification des services Web

Toutes les approches de vérification que nous avons présentées dans les deux sections précédentes sont intéressantes mais elles ont des limitations que nous détaillons dans cette section en les comparant les unes aux autres, ainsi qu'à notre approche, dans la table 6.8. La signification des entêtes de colonnes sont les suivants :

- T : traduction des types,
- E : traduction des expressions,
- V : traduction des variables,
- C : traduction des constantes Bpel, ces constantes sont différentes des expressions comme nous le verrons à la section 7.5,

[48] `http://www.it.uc3m.es/jaf/verbus/`

- AS : traduction des activités simples, c'est-à-dire les activités qui sont normalement très faciles à traduire vers des algèbres de processus ou des réseaux de Petri,
- Ar : traduction de l'opérateur d'arrêt (`exit`) dont la sémantique (voir section 10.2.10) n'est pas aussi évidente qu'il n'y parait,
- GE : traduction des gestionnaires d'exceptions,
- Ev : traduction des événements,
- At : prise en compte de l'atomicité de certaines activités dont l'exécution ne peut être entrelacée avec des activités parallèles,
- LdC : traduction des liens de contrôle,
- Tps : traduction des activités liées au temps et
- Env : possibilité de faire interagir le service BPEL traduit avec un environnement.

Dans chaque colonne, nous utilisons les notations suivantes :

- $--$: non traduit,
- $-$: traduit de façon très succincte,
- $+$: traduit de façon incomplète,
- $++$: traduit complètement ou quasi-complètement.

Approche	Données				Comportements							
	T	E	V	C	AS	Ar	GE	Ev	At	LdC	Tps	Env
Salaün et al	$--$	$--$	$--$	$--$	$+$	$--$	$--$	$-$	$--$	$--$	$--$	oui
Koshkina et Breugel	$--$	$--$	$--$	$--$	$+$	$--$	$--$	$-$	$--$	$-$	$--$	non
Yeung	$--$	$--$	$--$	$--$	$+$	$-$	$--$	$--$	$--$	$--$	$--$	non
Ouyang et al.	$--$	$--$	$--$	$--$	$++$	$++$	$++$	$+$	$--$	$++$	$--$	non
Qian et al.	$--$	$--$	$--$	$--$	$+$	$--$	$-$	$--$	$--$	$-$	$-$	oui
Mateescu et Rampacek	$--$	$--$	$--$	$--$	$++$	$--$	$-$	$--$	$--$	$--$	$++$	oui
Foster et al.	$-$	$--$	$+$	$--$	$++$	$--$	$--$	$-$	$--$	$--$	$--$	oui
Fu et al.	$-$	$+$	$+$	$--$	$++$	$--$	$-$	$--$	$--$	$-$	$--$	oui
Humbolt-Universität	$--$	$--$	$-$	$--$	$++$	$++$	$++$	$+$	$--$	$++$	$--$	non
Fisteus et al.	$-$	$--$	$-$	$--$	$++$	$--$	$-$	$--$	$--$	$--$	$--$	non
Nakajima	$--$	$--$	$-$	$--$	$++$	$--$	$--$	$--$	$--$	$-$	$--$	non
Bianculli	$-$	$--$	$-$	$--$	$+$	$-$	$-$	$-$	$--$	$-$	$--$	non
Moser et al.	$--$	$-$	$-$	$--$	$++$	$--$	$-$	$-$	$--$	$-$	$--$	non
Notre approche	$++$	$+$	$+$	$+$	$++$	$++$	$+$	$+$	$+$	$++$	$-$	oui

Table 6.1: Comparaison détaillée des approches de vérification des services Web

Ce tableau de comparaison montre bien que les meilleures approches, à l'heure actuelle, sont celles qui procèdent par traduction vers des réseaux de Petri : Ouyang et al [OVvdA+07] et celle des chercheurs de la Humbolt-Universität zu Berlin [HSS05, Loh08]. Les approches basées sur les algèbres de processus sont en principe plus prometteuses car elles proposent plus de possibilités de vérification et permettent de prendre en compte plus facilement les données. Toutefois, aucune n'a jusqu'à présent réussi à traiter en même temps les points difficiles : les données, les liens de causalités, les mécanismes de gestions d'erreurs et les constructions liées au temps. Aussi, les différentes approches que nous avons présentées ici ne détaillent, en général, pas les raisons pour lesquelles les traductions sont incomplètement définies.

6.9 Considérations pragmatiques

Les langages Xml Schema, XPath, Wsdl et Bpel sont vastes et comportent un grand nombre de constructions différentes. Avant de commencer nos travaux, nous avons voulu savoir quelles constructions sont utilisées en pratique afin de pouvoir cibler notre approche sur les vrais besoins et d'éviter de perdre du temps ou de compliquer la présentation de nos travaux en traitant des constructions dont personne ne se sert.

A cette fin, nous avons constitué une base d'exemples Xml Schema, XPath, Wsdl et Bpel récupérés sur Internet en 2009 grâce au moteur de recherche Google. Ces exemples proviennent à la fois de sources académiques, comme les travaux mentionnés aux sections 6.5, 6.6 et 6.7 et de sources industrielles, notamment la documentation et les exemples livrés avec ActiveBpel (l'interpréteur Bpel de la suite ActiveVos) et Oracle Bpel Manager.

Nous avons rassemblé ainsi 312 services Web Bpel. Ils contiennent 1450 expressions XPath dont 753 uniques. Ils incluent 431 fichiers Wsdl, dont 306 uniques, qui incluent à leur tour 261 fichiers Xml Schema dont 64 uniques. Le faible nombre de fichiers Xml Schema comparé au nombre de fichiers Wsdl s'explique par le fait qu'il est permis de déclarer des types Xml Schema directement au sein d'un fichier Wsdl.

6.10 Contributions

Dans la suite de cette partie, nous détaillons une traduction des services Bpel en Lotos NT. Nous présentons nos contributions, langage par langage, dans l'ordre suivant : Xml Schema au chapitre 7, XPath au chapitre 8, Wsdl au chapitre 9 et Bpel au chapitre 10. La figure 6.1 montre une vue d'ensemble de la traduction. Un service Bpel est traduit par un module Lotos NT dans lequel :

- les déclarations de types Xml Schema sont traduites par des déclarations de types Lotos NT ainsi que des fonctions Lotos NT pour manipuler ces types,
- les expressions XPath figurant au sein de ce service sont traduites en expressions Lotos NT,
- l'interface Wsdl du service est traduite par des déclarations de canaux de communications et des déclarations de types et
- un processus Lotos NT encode le comportement du service Bpel.

Nous avons choisi le langage Lotos NT comme langage cible car c'est un langage nouveau, qui se prête particulièrement bien à la manipulation de types de données et de fonctions, à l'opposé de Lotos Ccs, ou Fsp. De plus, ce langage est intégré à la boîte à outils Cadp, de telle sorte que traduire des processus Bpel dans ce langage permet, par la suite, de les vérifier à l'aide des outils de Cadp. Enfin, nous présentons les premiers travaux visant à encoder des services Web en Lotos NT.

Le résultat de notre étude est une traduction détaillée d'un service Bpel en processus Lotos NT. Ce processus Lotos NT permet alors de vérifier formellement le bon comportement du service Bpel d'origine par le biais de formules de logique temporelle que l'utilisateur fournit. Le processus Lotos NT peut être vérifié isolément, ou bien en interaction avec un environnement qui reproduit le comportement du service Bpel d'origine. De plus, par cette traduction nous donnons une sémantique formelle au langage Bpel, alors que la norme ne donne qu'une sémantique floue, à base de longues explications informelles. Au final, notre étude apporte trois contributions à la vérification formelle des services Web Bpel :

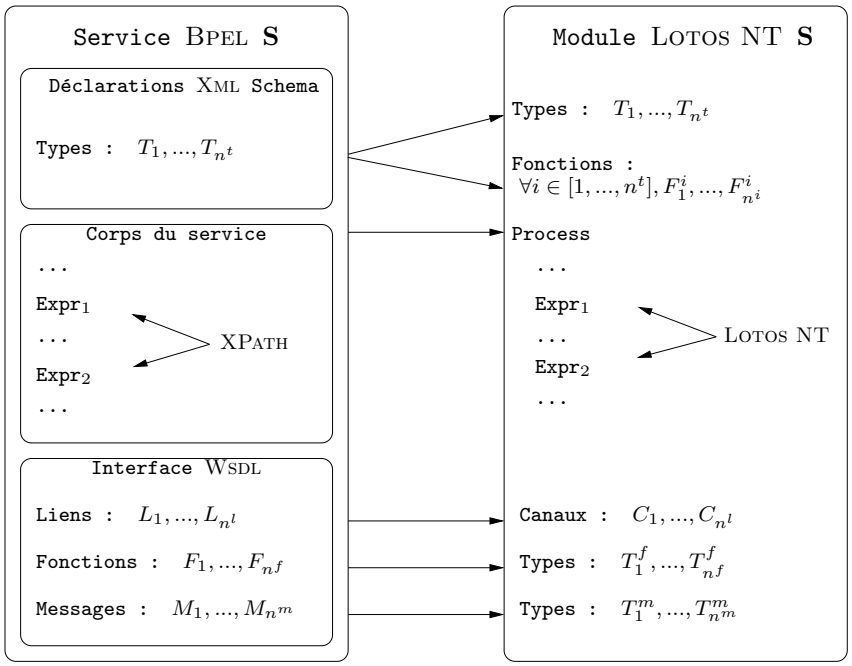

Figure 6.1: Vue d'ensemble de la traduction de BPEL vers LOTOS NT

1. elle est pragmatique, c'est-à-dire que nous ne traduisons que les constructions utiles et nous expliquons clairement pourquoi nous ne traitons pas certaines constructions,
2. elle est plus complète que les approches présentées dans les travaux existants, nous traitons notamment les quatre langages qui servent à définir un service Web BPEL,
3. elle est entièrement spécifiée dans ce document, au moyen de notations mathématiques, afin que quiconque puisse reproduire la traduction que nous proposons.

Chapitre 7

Traduction des types XML Schema

Dans ce chapitre, nous commençons par présenter, à la section 7.1, le langage XML (*eXtensible Markup Language*) qui est une syntaxe générique pour la représentation d'informations. Officiellement, un fichier écrit dans cette syntaxe est appelé un *document* XML et contient un arbre d'*éléments* qui descendent d'un élément *racine*. Dans le cas de BPEL, les documents (que nous appèlerons désormais *termes*) XML désignent les *valeurs* qui sont manipulées dans les services.

Il existe de nombreux langages de description ou de programmation avec une syntaxe XML comme par exemple : BPEL, XML Schema, WSDL, SOAP, XHTML, MATHML, RSS, XSLT... De nombreux outils utilisent des langages à syntaxe XML pour la sauvegardes de documents : Microsoft Office, Open Office, iWorks... Ces langages sont définis à l'aide de schémas XML qui n'acceptent qu'un sous-ensemble de XML.

Il existe différents langages pour définir des schémas XML. Le précurseur de ces langages a été DTD [Gro04b] (*Document Type Definition*) que le W3C a défini en même temps que la première version de la norme XML. Mais, DTD n'était pas un langage à syntaxe XML et ne permettait pas de distinguer entre des valeurs entières, booléennes... ni de traiter les espaces de noms XML. Pour remédier à ces lacunes et permettre une définition plus précise de sous-ensembles du langage XML, de nouveaux langages de schémas (comme Relax NG) sont apparus par la suite. XML Schema est le langage de schémas officiel du W3C et nous le présentons à la section 7.2. Sa syntaxe est elle-même définie en XML. XML Schema sert aussi à définir les types des données manipulées dans les services BPEL et c'est ce qui nous intéresse dans le cadre de cette étude.

Une fois cette introduction terminée, nous proposons, à la section 7.3 une grammaire pour le langage XML Schema. Nous utilisons ensuite cette grammaire pour définir la traduction de XML Schema en LOTOS NT, à la section 7.4. Enfin, à la section 7.5, nous présentons les constantes du langage BPEL et détaillons leur traduction en LOTOS NT. Ces constantes sont des termes XML qui satisfont les types XML Schema qu'un service BPEL définit. C'est pour cette raison que nous avons choisi de présenter ces constantes dans ce chapitre plutôt que dans le chapitre sur BPEL.

7.1 Les termes de XML

Un terme XML est constitué d'un élément XML racine qui contient une arborescence de sous-éléments. Chaque élément est nommé (par une chaîne `x`) et son contenu est textuellement encadré par deux balises (ouvrante `<x>` et fermante `</x>`). Ce contenu peut être :

- soit vide, on parle alors d'*élément vide*,
- soit une chaîne de caractères, on parle alors d'*élément à contenu simple*,
- soit une liste d'éléments, on parle alors d'*élément à contenu complexe*,
- soit un mélange de texte et d'éléments, on parle alors d'*élément à contenu mixte*.

Afin d'illustrer ces différents types d'éléments, voici quelques exemples écrits en XHTML, un langage à syntaxe XML promu par le W3C pour l'écriture de pages Web :

- élément vide : `<br></br>`, aussi abrégé en `<br />`
- élément à contenu simple :

```
<p>
  Ceci est un paragraphe.
</p>
```

- élément à contenu complexe :

```
<ol>
  <li> Premier point,</li>
  <li> Second point,</li>
  <li> ... </li>
</ol>
```

- élément à contenu mixte :

```
<p>
  Le site Internet de l'<b>INRIA</b> est le suivant :
    http://www.inria.fr
</p>
```

La balise ouvrante d'un élément peut contenir des *attributs* dont les valeurs sont des chaînes de caractères. En général, les attributs donnent des informations supplémentaires sur l'élément, indépendamment de son contenu, Par exemple, l'attribut `href` de l'élément `a` de XHTML est utilisé pour spécifier l'adresse de la page Internet qui doit s'ouvrir lorsque l'utilisateur, consultant une page Internet, clique sur la zone de l'écran correspondant à un élément `a`. L'attribut `style` du même élément permet de spécifier comment le contenu de l'élément doit être affiché à l'écran. Par exemple :

```
<a style="color:ref" href="http://www.inria.fr">http://www.inria.fr</a>
```

Nous formalisons à présent une grammaire concrète exacte des termes XML :

$$
\begin{array}{rcl}
\mathit{Term} & ::= & \mathit{Element} \\
\mathit{Element} & ::= & \texttt{<}E\ A_1\texttt{=}V_1\ \ldots\ A_n\texttt{=}V_n\ \texttt{>}\mathit{Content}\texttt{</}E\texttt{>} \\
 & | & \texttt{<}\ E\texttt{/>} \\
\mathit{Content} & ::= & \mathit{Child}_0\ \ldots\ \mathit{Child}_n \\
\mathit{Child} & ::= & V \\
 & | & \mathit{Element}
\end{array}
$$

Nous utilisons les notations suivantes pour les identificateurs XML :

- E est un identificateur d'élément et
- A est un identificateur d'attribut.

V est une expression constante qui peut être une chaîne de caractères, un booléen ou un nombre.

Dans ce document, les grammaires sont données au moyen d'une notation Bnf. Nous utilisons des indices (N_1, N_2, ...) pour distinguer les différentes occurrences du même non-terminal au sein d'une même règle. $X_1...X_n$ désigne une liste potentiellement vide tandis que $X_0...X_n$ désigne une liste qui contient au moins un élément.

Cette grammaire pour Xml est très lâche et donne tous les termes Xml possibles. En pratique, on s'intéresse à des sous-ensembles "bien typés" que l'on définit à l'aide de schémas Xml, comme nous allons le voir à la section suivante.

Nous n'avons, jusqu'ici, pas mentionné deux constructions qui peuvent apparaître librement au sein d'un terme Xml et ne peuvent être spécifiés par les langages de schémas :

- les "instructions de traitement" (*processing instruction*) servent à indiquer, à l'application qui va traiter le terme Xml, une action particulière à exécuter. Elle a pour balise ouvrante `<?` et pour balise fermante `?>`.
- les "commentaires" sont délimités par la balise ouvrante `<!--` et la balise fermante `-->`.

7.2 Présentation de XML Schema

Xml Schema [Gro04e, Gro04d] est une grammaire ou un système de types qui décrit précisément un sous-ensemble de Xml, c'est-à-dire, le nom et le contenu de chaque élément pouvant apparaître dans un terme Xml. Pour qu'un terme soit valide (bien typé) par rapport à un langage ou un format donné, il doit appartenir au sous-ensemble de Xml décrit par le schéma Xml associé à ce langage ou format.

Nous présentons maintenant la syntaxe de Xml Schema 1.0, la version de la norme à laquelle la norme Bpel fait référence, en omettant toutes les constructions qui relèvent du sucre syntaxique et qui peuvent être éliminées automatiquement à l'aide de transformation syntaxiques décrites dans la norme Xml Schema [Gro04d, Sec. 3.4–3.8]. Aussi, nous ne considérons pas les éléments à contenu mixte, ni la restriction **whiteSpace** [Gro04e, Sec 2.5.1], car ces constructions ne sont pas utilisées par le langage Bpel.

La figure 7.1 donne une vue synthétique du système de types de Xml Schema que nous détaillons ci-après.

7.2.1 Types spéciaux

Xml Schema compte deux types spéciaux qui sont définis par union des domaines d'autres types :

- anySimpleType dont le domaine est l'union du domaine de anyAtomicType et des domaine des *types simples* (pour les éléments à contenu simple) définis par l'utilisateur et
- anyType dont le domaine est l'union du domaine de anySimpleType et des domaines des *types complexes* (pour les éléments à contenu complexe) définis par l'utilisateur.

Ces types ne peuvent être redéfinis par l'utilisateur.

7.2.2 Types prédéfinis

Xml Schema possède 44 types prédéfinis qui ne peuvent être redéfinis par l'utilisateur. Nous

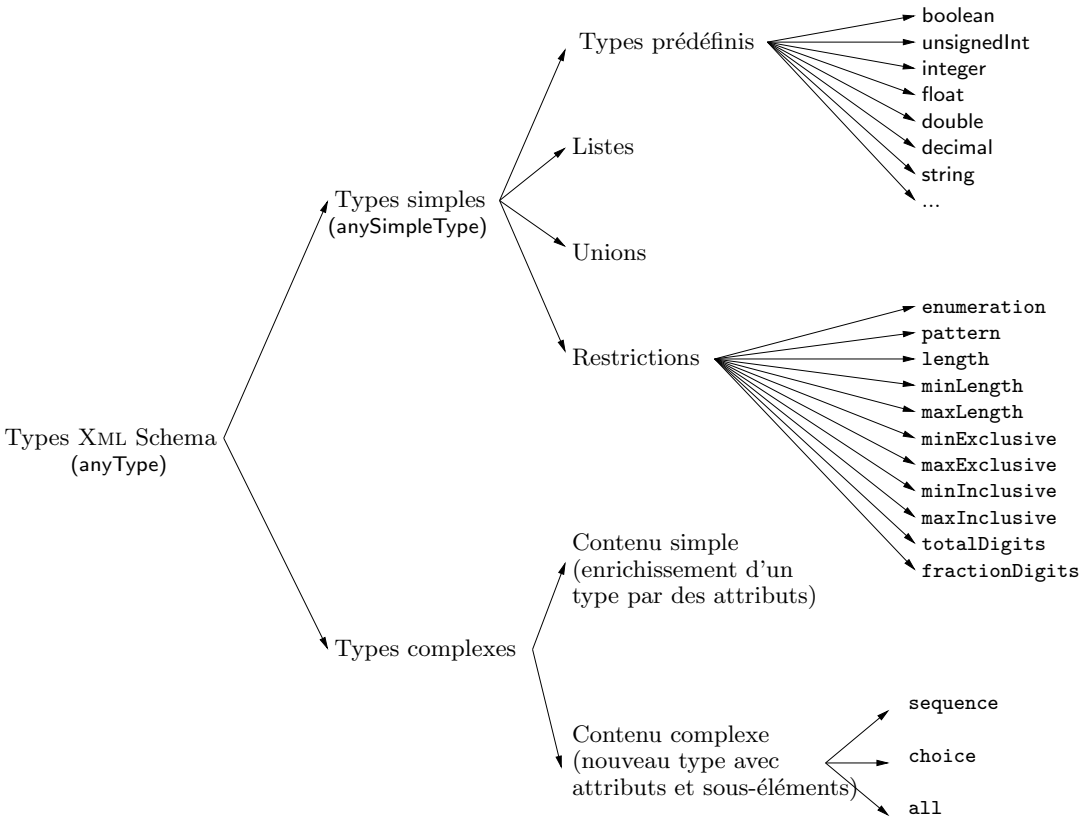

Figure 7.1: Système de types de Xml Schema

présentons ces types en les classant par catégories :

- booléens : boolean,
- entiers naturels : unsignedShort, unsignedByte, unsignedInt, unsignedLong, positiveInteger et nonNegativeInteger,
- entiers relatifs : short, byte, int, long, negativeInteger, nonPositiveInteger et integer,
- nombres réels : float, double et decimal,
- chaînes de caractères : string, hexBinary, base64Binary, anyURI, QName, NOTATION, normalizedString, token, language, NMTOKEN, NMTOKENS, Name, NCName, ID, IDREF, IDREFS, ENTITY et ENTITIES,
- durées : duration,
- dates : dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay et gMonth.

En Xml, string est le type le plus général et les domaines des autres types sont définis par sous-typage des chaînes de caractères au moyen d'expressions régulières et éventuellement de bornes inférieures et supérieures pour certains types numériques. Par exemple :

- le domaine de boolean est constitué de toutes les chaînes de caractères qui satisfont l'expression régulière `(true|false)`,

- le domaine de integer est constitué de toutes les valeurs qui satisfont l'expression régulière `(\+|-)?[0-9]+`,
- le domaine de unsignedByte est constitué de toutes les valeurs du type integer qui sont comprises entre 0 et 255 (0 et 255 inclus) et
- le domaine de hexBinary est constitué de toutes les chaînes de caractères qui satisfont l'expression régulière `([0-9a-fA-F]{2})*`.

7.2.3 Types simples

Les types simples comprennent les types prédéfinis, le type spécial anySimpleType et les types *dérivés*, définis par l'utilisateur. La définition de types simples dérivés se fait au moyen de trois constructions : *liste*, *union* et *restriction* :

- La construction *liste* a la syntaxe suivante :

  ```
  <simpleType name="T">
    <list itemType="T'" />
  </simpleType>
  ```

  où T' est un type simple. Elle définit un nouveau type simple, nommé T, dont le domaine comprend toutes les chaînes de caractères correspondant à une liste de valeurs du type T' séparées par des espaces.

  Par exemple :

  ```
  <simpleType name="IntegerList">
    <list itemType="integer" />
  </simpleType>
  ```

  définit un nouveau type qui possède, entre autres, les valeurs suivantes : `"42"`, `"-2 -1 0 1 2"` et `"2 4 8 15 16 23 42"`.

- La construction *union* a la syntaxe suivante :

  ```
  <simpleType name="T">
    <union memberTypes="T0 ... Tn" />
  </simpleType>
  ```

  où $T_0 ... T_n$ sont des types simples. Elle définit un nouveau type simple nommé T dont le domaine est l'union des domaines des types T_0, ..., T_n. Il s'agit d'une union sans discriminant, similaire à celle du langage C. Il est important de noter que les domaines des types $T_0 ... T_n$ n'ont pas nécessairement une intersection vide. Une valeur de T peut donc appartenir à plusieurs types parmi $T_0 ... T_n$.

  Par exemple :

  ```
  <simpleType name="BooleanIntegerUnion">
    <union memberTypes="boolean integer" />
  </simpleType>
  ```

définit un nouveau type dont le domaine comprend à la fois les valeurs entières et les valeurs booléennes.

- La construction *restriction* a la syntaxe suivante :

```
<simpleType name="T">
  <restriction base="T'" />
    Rest_0
    ...
    Rest_n
  </restriction>
</simpleType>
```

Elle définit un nouveau type simple nommé T, qui est un sous-type du type simple T' : le domaine de T comprend toutes les valeurs du domaine de T' (dit *type de base*) qui satisfont les restrictions $Rest_0$ ... $Rest_n$. Les restrictions permises sont les suivantes :

- `<enumeration="`V_i`" />` : cette restriction peut apparaître à n reprises ($i \in [1...n]$), avec des valeurs V_i deux à deux distinctes. Une valeur V du type de base satisfait cette restriction si $\exists\ i \in [1...n] \mid V_i = V$.
- `<pattern="`V_i`">` : cette restriction peut apparaître à n reprises ($i \in [1...n]$), avec des valeurs V_i deux à deux distinctes. Une valeur V du type de base satisfait cette restriction si $\forall\ i \in [1...n]$, la représentation sous forme de chaîne de caractères de V est acceptée par l'expression régulière V_i.
- `<length="`V'`" />` : une valeur V du type de base valide cette restriction si la longueur de la chaîne de caractères représentant la valeur V est égale à V'. Pour certains types, la longueur a une signification différente : ainsi, pour hexBinary, il s'agit du nombre d'octets que la valeur V contient tandis que pour les types simples `list` il s'agit du nombre d'éléments contenus dans la liste V.
- `<minLength="`V'`" />` : une valeur V du type de base satisfait cette restriction si la longueur de la chaîne de caractères représentant la valeur V est supérieure ou égale à V'.
- `<maxLength="`V'`" />` : une valeur V du type de base satisfait cette restriction si la longueur de la chaîne de caractères représentant la valeur V est inférieure ou égale à V'.

Les restrictions suivantes ne s'appliquent qu'aux types *numériques* (c'est-à-dire, les types entiers et réels, ainsi que les types de date et de durée, tels que date et duration notamment) :

- `<minExclusive="`V'`" />` : une valeur V du type de base satisfait cette restriction si $V > V'$.
- `<maxExclusive="`V'`" />` : une valeur V du type de base satisfait cette restriction si $V < V'$.
- `<minInclusive="`V'`" />` : une valeur V du type de base satisfait cette restriction si $V \geq V'$.
- `<maxInclusive="`V'`" />` : une valeur V du type de base satisfait cette restriction si $V \leq V'$.
- `<totalDigits="`V'`" />` : une valeur V du type de base satisfait cette restriction si sa représentation sous forme de chaîne de caractères contient au plus V' chiffres.

– `<fractionDigits="`V'`" />` : une valeur V du type de base satisfait cette restriction si sa représentation sous forme de chaîne de caractères contient au plus V' chiffres après la virgule.

Par exemple :

```
<simpleType name="IntegerRestriction">
  <restriction base="integer">
    <pattern value="^3.*2$" />
    <minLength value="2" />
    <maxLength value="3" />
  </restriction>
</simpleType>
```

définit un nouveau type dont le domaine comprend les valeurs suivantes : 32, 302, 312, 322, 332, 342, 352, 362, 372, 382 et 392.

7.2.4 Types complexes

Les types complexes se déclinent en deux sortes :

- types complexes *à contenu simple* et
- types complexes *à contenu complexe*.

Nous présentons ces deux sortes successivement.

7.2.5 Types complexes à contenu simple

La syntaxe de définition d'un type complexe à contenu simple est la suivante :

```
<complexType name="T">
  <simpleContent>
    <extension base="T'" />
      Attr_0
      ...
      Attr_n
    </extension>
  </simpleContent>
</complexType>
```

Elle définit un nouveau type complexe nommé T dont le contenu est une valeur du domaine du type T' (T' doit être un identificateur de type simple). Le nouveau type est muni des attributs $Attr_0, ..., Attr_n$ auxquels la section 7.2.7 est consacrée.

7.2.6 Types complexes à contenu complexe

La syntaxe de définition d'un type complexe à contenu complexe est la suivante :

```
<complexType name="T">
  <Order = (sequence|choice|all)>
    Elem_0
    ...
    Elem_m
  </Order>
  Attr_1
  ...
  Attr_n
</complexType>
```

L'ordre et le nombre d'occurrence des éléments $Elem_0 ... Elem_m$ dans le type complexe est gouverné par la valeur de la balise *Order* qui entoure leur déclaration :

- `sequence` : les m éléments apparaissent dans l'ordre dans lequel ils ont été déclarés ;
- `choice` : un élément seulement parmi les m déclarés apparaît ;
- `all` : les m éléments apparaissent tous dans un ordre arbitraire.

7.2.7 Attributs

La syntaxe de définition d'un attribut A est la suivante :

```
<attribute name="A"
     type="T"
   [ use="V^use" ]
   [ fixed="V^fixed" ]
   [ default="V^default" ] />
```

où :

- T désigne le type de la valeur de l'attribut (T doit être un type simple).
- La clause `use`, si présente et égale à "`required`", indique que la présence de l'attribut est obligatoire, sinon, il est optionnel (même si la clause est absente).
- La clause `fixed`, si présente, rend l'attribut constant et spécifie sa valeur.
- La clause `default`, si présente, spécifie la valeur par défaut de l'attribut. Cette valeur sera prise en compte lorsque l'attribut est optionnel et que l'utilisateur ne spécifie pas sa valeur.

L'exemple suivant illustre un cas typique d'utilisation des attributs :

```
<simpleType name="DistanceUnit">
  <restriction base="string">
    <enumeration value="cm" />
    <enumeration value="dm" />
    <enumeration value="m" />
    <enumeration value="dam" />
    <enumeration value="hm" />
    <enumeration value="km" />
  </restriction>
</simpleType>
<complexType name="DistanceValue">
  <simpleContent>
    <extension base="decimal">
      <attribute name="unit" type="DistanceUnit" default="m" />
    </extension>
  </simpleContent>
</complexType>
```

Le type `DistanceUnit` spécifie une énumération d'unités de distance. Le type `DistanceValue` est un type complexe à contenu simple dont la valeur est de type `decimal` et qui a un attribut nommé `unit` dont la valeur est de type `DistanceUnit`. Par défaut, une distance dont l'unité est non spécifiée sera évaluée en mètres. En déclarant un élément nommé `distance` dont le type est `DistanceValue`, il devient possible d'écrire :

```
<distance unit="km">384467</distance>
```

7.2.8 Eléments

La syntaxe de définition d'un élément E est la suivante :

<element name="E"
type="T"
fixed="V^{fixed}"?
default="$V^{default}$"?
minOccurs="$V^{minOccurs}$"?
maxOccurs="$V^{maxOccurs}$"? />

où :

- T est le type du contenu de l'élément.
- La clause `fixed` rend l'élément constant et spécifie sa valeur, seulement si $V^{maxOccurs} = 1$ et que T est un type simple.
- La clause `default` désigne la valeur par défaut de l'élément lorsqu'il est omis. La présence de $V^{default}$ est prise en compte uniquement si $V^{minOccurs} = 0$, $V^{maxOccurs} = 1$ et que T est un type simple.
- La clause `minOccurs`, si présente, définit le nombre minimum d'occurrences de l'élément.
- La clause `maxOccurs`, si présente, définit le nombre maximum d'occurrences de l'élément ; la valeur "`unbounded`" dénote un nombre d'occurrences non borné.

La syntaxe de définition d'un élément racine c'est-à-dire l'élément au sommet de l'arborescence d'éléments dans un document Xml ou une valeur de variable Bpel, est la même que celle d'un élément au sein d'une définition de type complexe à contenu complexe (cf. section 7.2.6). Les valeurs $V^{minOccurs}$ et $V^{maxOccurs}$ sont fixées à 1, tandis que $V^{default}$ est interdite.

7.2.9 Récursivité

La récursivité dans les déclarations de types Xml Schema n'est pas mentionnée dans la norme. Nous considérons donc qu'elle est permise, à défaut d'être explicitement interdite.

7.2.10 Exemple d'utilisation de XML Schema

Les déclarations Xml Schema suivantes :

```
<complexType name="MusicalPiece">
  <all>
    <element name="name" type="string" />
    <element name="author" type="string" />
    <element name="type" type="string" />
    <element name="year" type="gYear" />
  </all>
</complexType>

<complexType name="DiscographyType">
  <sequence>
    <element name="entry" type="MusicalPiece" maxOccurs="unbounded" />
  </sequence>
</complexType>

<element name="discography" type="DiscographyType" />
```

permet de définir une bibliothèque d'œuvres musicales, dont voici un exemple :

```
<discography>
  <entry>
    <name>L'estate</name>
    <author>Antonio Vivaldi</author>
    <type>Violin Concerto</type>
    <year>1723</year>
  </entry>
  <entry>
    <year>1893</year>
    <type>Symphony</type>
    <name>New World Symphony</name>
    <author>Anton Dvorak</author>
  </entry>
  <entry>
    <author>Maurice Ravel</author>
    <name>Bolero</name>
    <year>1928</year>
    <type>Ballet</type>
  </entry>
</discography>
```

7.3 Grammaire abstraite pour XML Schema

Nous présentons dans cette section une grammaire abstraite de XML Schema dont nous nous servirons pour formaliser la traduction de ce langage vers LOTOS NT.

Afin de simplifier la présentation de notre grammaire abstraite XML Schema, nous ne considérons pas les espaces de noms XML qui ne sont qu'un mécanisme pour garantir l'unicité des identificateurs XML, ce dont nous faisons l'hypothèse ici. Nous utilisons les notations suivantes pour les identificateurs XML Schema :

- T^p: identificateur de type prédéfini (à l'exception de anyType qui est un type complexe),
- T^s: identificateur de type simple,
- T^c: identificateur de type complexe et
- T^e: identificateur d'élément racine ; XML Schema ne considère pas les éléments racines comme des types, à l'inverse de BPEL qui permet la déclaration de variables avec pour type un élément racine XML Schema (`var:`T^e).

Afin de faire référence à une sorte de types parmi plusieurs possibles, nous utilisons les abréviations suivantes :

- $T^{ps} ::= T^p \mid T^s$
- $T^{psc} ::= T^p \mid T^s \mid T^c$

Notre grammaire abstraite pour XML Schema est la suivante :

$$
\begin{array}{rcl}
Spec & ::= & Def_1 \ ... \ Def_n \\
Def & ::= & \textbf{type } T^s \ = \ Simp \\
 & | & \textbf{type } T^e \ = \ Elem \\
 & | & \textbf{type } T^c \ = \ Comp \\
Simp & ::= & \textbf{restriction } (T^{ps}, Rest_0, ..., Rest_n) \\
 & | & \textbf{list } (T^{ps}) \\
 & | & \textbf{union } (T_0^{ps}, ..., T_n^{ps}) \\
Elem & ::= & \textbf{element } (T^{psc}, V^{fixed}, V^{default}, V^{minOccurs}, V^{maxOccurs}) \\
Comp & ::= & \textbf{extension } (T^s, A_0 : Attr_0, ..., A_n : Attr_n) \\
 & | & \textbf{sequence } (E_0 : Elem_0, ..., E_m : Elem_m, A_1 : Attr_1, ..., A_n : Attr_n) \\
 & | & \textbf{all } (E_0 : Elem_0, ..., E_m : Elem_m, A_1 : Attr_1, ..., A_n : Attr_n) \\
 & | & \textbf{choice } (E_0 : Elem_0, ..., E_m : Elem_m, A_1 : Attr_1, ..., A_n : Attr_n) \\
Attr & ::= & \textbf{attribute } (T^{ps}, V^{optional}, V^{fixed}, V^{default}) \\
Rest & ::= & \textbf{enumeration } (V_1, ..., V_n) \mid \textbf{pattern } (V_1, ..., V_n) \mid \textbf{length } (V) \\
 & | & \textbf{minLength } (V) \mid \textbf{maxLength } (V) \mid \textbf{minExclusive } (V) \\
 & | & \textbf{maxExclusive } (V) \mid \textbf{minInclusive } (V) \mid \textbf{maxInclusive } (V) \\
 & | & \textbf{fractionDigits } (V) \mid \textbf{totalDigits } (V) \\
 & | & \textbf{assertions } (Expr_1, ..., Expr_n)
\end{array}
$$

où A, E et V sont définis à la section 7.1.

Dans la définition d'un attribut, nous remplaçons la notation V^{use} par une notation $V^{optional}$, où $V^{optional}$ est une valeur booléenne qui vaut "faux" si V^{use} est égale à "`required`", et "vrai" sinon.

7.4 Traduction en LOTOS NT

Dans cette section, nous définissons un algorithme de traduction de XML Schema vers LOTOS NT. Cette traduction génère du code LOTOS NT efficace afin de répondre aux exigences spécifiques de la vérification formelle : la traduction des types XML Schema en types LOTOS NT optimisés réduit la quantité de mémoire et le temps nécessaires à la génération de l'espace d'états correspondant à un service BPEL.

Nous commençons par examiner les travaux antérieurs qui traitent de la traduction de XML Schema vers des structures de données, notamment ceux pour lesquels cette traduction s'inscrit dans des approches plus générales de vérification formelle de services BPEL. Ensuite, nous présentons notre traduction des types XML Schema en LOTOS NT, d'abord de manière générale, puis de manière détaillée, construction par construction.

Nous avons utilisé la version du compilateur LOTOS NT disponible en janvier 2011. Les versions plus récentes comportent la notion de types intervalles (sur des types entiers, signés ou non, et caractères) et les notions de sous-types définis par un prédicat. Il est donc possible de mettre au point une traduction plus fine que celle proposée ici, en utilisant ces deux nouveaux concepts de LOTOS NT.

7.4.1 Etat de l'art

Le W3C tient à jour une liste d'outils[49] pour XML Schema. On y trouve notamment des outils de traduction de XML Schema vers des langages de programmation. LMX[50] et Code Synthesis XSD[51]

[49] `http://www.w3.org/XML/Schema`
[50] `http://www.codalogic.com/lmx/`
[51] `http://codesynthesis.com/products/xsd/`

transforment des types XML Schema en classes C++, tout comme XBINDER[52] qui sait aussi générer des classes C# et Java ainsi que des structures de données du langage C. Comme LOTOS NT peut s'interfacer avec le langage C, XBINDER aurait pu être un outil intéressant dans le cadre de notre traduction. Malheureusement, il s'agit d'un outil commercial dont la documentation n'illustre la traduction qu'à l'aide d'exemples. Sans une compréhension pointue de l'algorithme de traduction, il n'est pas possible de savoir si XBINDER répond aux exigences spécifiques de la vérification formelle par *model checking*. Il n'est donc pas envisageable d'utiliser cet outil au sein de notre propre traduction.

Dans le cadre de la vérification formelle de services BPEL, seuls Fu, Bultan et Su [FBS04b] décrivent précisément comment sont traduits les types de données XML Schema. Toutefois, leur approche ne nous donne pas entièrement satisfaction, pour les raisons suivantes :

- XML Schema est traduit au moyen d'un langage intermédiaire : MSL [BFRW01]. Ce langage se veut une notation concise pour XML Schema, mais ne modélise pas les types simples, qui sont un point clé dans le contexte de la vérification formelle de services BPEL.
- Il n'y a aucune réflexion (itérateurs, restrictions sur le domaine...) portant sur la traduction en PROMELA des types XML Schema (qu'ils s'agissent de types prédéfinis, simples ou complexes),
- Les auteurs simplifient XML Schema en omettant la construction **all**.

7.4.2 Principe général de traduction

En vérification formelle, les facteurs limitants ne comprennent pas seulement la taille de l'espace d'états d'une spécification. Le temps et la mémoire nécessaires à la génération de cet espace d'états sont aussi primordiaux. Ces deux facteurs peuvent être diminués de deux manières :

- en améliorant les outils de traduction/compilation et
- en optimisant la spécification à traiter.

En règle générale, nous essayons d'écrire un maximum de code LOTOS NT qui est éventuellement traduit en LOTOS par LNT2LOTOS puis en C par CÆSAR.ADT et CÆSAR. Puisque LOTOS NT permet d'importer directement du code C, nous écrivons parfois du code C, lorsque nous pensons que cela induit un gain significatif de performances (comme pour les itérateurs).

La norme BPEL [Com01, Sec. 8] précise que les valeurs des variables doivent être encodées sous la forme d'un *Information Set* (ou *InfoSet*) XML [Gro04c]. Il s'agit d'une structure de données qui représente un terme XML sous la forme d'une arborescence s'il s'agit d'une valeur de type complexe ou d'une chaîne de caractères s'il s'agit d'une valeur de type simple. L'approche de traduction la plus simple consisterait alors à s'inspirer des *InfoSets* pour représenter les termes XML par un unique type LOTOS NT qui serait défini ainsi :

[52] `http://www.obj-sys.com/xbinder.shtml`

```
type ValueType is
  SimpleValue (value:String)
  ComplexValueSimpleContent (attributes:AttributeList, value:String),
  ComplexValueComplexContent (attributes:AttributeList, elements:ElementList)
end type
type Attribute is
  Attribute (name:String, value:String)
end type

type AttributeList is
  list of Attribute
end type
type Element is
  Element (name:String, value:ValueType)
end type

type ElementList is
  list of Element
end type
```

Dans le cadre de la vérification formelle, une telle traduction serait inefficace :

- **en termes de temps génération**, car elle obligerait à faire constamment des conversions entre chaînes de caractères et nombres entiers, nombres réels ou booléens dans l'évaluation d'expressions XPath (cf. chapitre 8) et
- **en termes d'utilisation de la mémoire**, à cause de la représentation de valeurs numériques à l'aide de chaînes de caractères.

Par conséquent, nous avons choisi une approche différente, plus complexe, qui consiste à traduire chaque type Xml Schema par un type Lotos NT spécifique et un ensemble de fonctions Lotos NT. Par exemple :

- le type prédéfini Xml Schema string est traduit par le type Lotos NT String qui est équipé des fonctions *length* pour calculer la longueur d'une chaîne et *match* pour vérifier que le langage défini par une expression régulière accepte une chaîne de caractères donnée. Ces fonctions sont nécessaires pour la traduction des types définis par restriction (cf. section 7.4.10).
- un type simple défini, comme suit, par restriction sur le type prédéfini string :

  ```
  <simpleType name="identifier">
    <restriction base="string">
      <pattern value="[a-zA-Z][a-zA-Z0-9]*" />
    </restriction>
  </simpleType>
  ```

  est traduit en un type Lotos NT et une fonction Lotos NT qui va vérifier si une valeur de type string est compatible avec une valeur de ce type défini par restriction :

  ```
  type identifier is
    identifier (value:String)
  end type

  function valid_identifier (input:String) : Bool is
    return match (input, "[a-zA-Z][a-zA-Z0-9]*")
  ```

```
end function
```

- un type complexe **sequence** est traduit par un type LOTOS NT dont le constructeur prend comme argument les différents éléments constituant la séquence ainsi que les attributs du type. Pour le type `DiscographyType` de la section 7.2.10, cela donne :

  ```
  type DiscographyType is
    DiscographyType (entry:MusicalPieceList)
  end type
  ```

  Dans ce cas, il est nécessaire de traduire le type de l'élément `entry` par une liste car le nombre d'occurrences de cet élément n'est pas limité.

Avant de nous lancer dans les détails de la traduction, nous introduisons, dans les trois prochaines sections, les notions, essentielles à la mise en œuvre de la vérification formelle, de domaines et types de base (nécessaires à la définition des conversions et des itérateurs), de conversions entre types (dues au passage de BPEL, un langage très faiblement typé, à LOTOS NT, un langage fortement typé) et d'itérateurs (pour énumérer l'ensemble des valeurs d'un type donné).

7.4.3 Domaines et types de base

Les notions de domaine d'un type et de type de base d'un type sont nécessaires à la définition des conversions et des itérateurs.

Le domaine d'un type T, que nous notons *domain* (T), désigne l'ensemble des valeurs de T. Le domaine de certains types est un ensemble de valeurs fini, comme par exemple $\{\texttt{true}, \texttt{false}\}$ pour le type boolean ou $\{0, 1, 2...255\}$ pour le type unsignedByte. Pour d'autres types, en revanche, tels que string et decimal ainsi que certains types qui en sont dérivés par restriction, le domaine est infini.

Lors de la présentation des types prédéfinis à la section 7.2.3, nous avons vu que les types atomiques peuvent être répartis en plusieurs groupes. Dans chaque groupe, il existe un type dont le domaine inclut les domaines des autres types du groupe : string pour les chaînes de caractères, integer pour les nombres entiers relatifs ou encore decimal pour les nombres réels. En suivant ce principe, et en s'inspirant de ce qui se fait dans un langage comme ADA, nous introduisons pour chaque type XML Schema, la notion de *type de base* afin de faciliter la définition des conversions dans le reste de ce chapitre. Il s'agit de classifier les types XML Schema selon des groupes et de choisir, dans chaque groupe, un type de base vers lequel tous les autres types du même groupe peuvent être facilement convertis. En général (sauf, pour les attributs et les éléments, cf. sections 7.4.12 et 7.4.16), pour un type T, le type de base de T, que nous notons *base* (T) est un type dont le domaine contient toutes les valeurs du domaine de T :

$$T' = \mathit{base}\ (T)\ \Rightarrow\ \mathit{domain}\ (T) \subset \mathit{domain}\ (T')$$

La conversion d'une valeur du domaine de T en une valeur du domaine de *base* (T) est alors directe et la conversion inverse nécessite la vérification de l'appartenance, au domaine de T, de la valeur à convertir (qui appartient au domaine de *base* (T)).

Par exemple :

- les types de base des types prédéfinis sont les éléments maximaux du treillis de types XML Schema ; notamment, le type integer est le type de base de integer (il est son propre type de base), nonPositiveInteger, negativeInteger, long, int, short et byte,
- le type de base d'un type simple défini par restriction de T est le type de base de T ; notamment, integer est le type de base de tous les types simples dérivés par restriction d'un type ayant integer pour type de base,

- les types complexes représentent le pire des cas car l'analyse nécessaire à leur classification en différents groupes nous semble trop compliquée par rapport aux bénéfices qu'elle pourrait apporter (car chaque service de notre base d'exemples ne déclare en moyenne que 5 types complexes XML Schema) ; nous avons donc décidé que les types complexes seraient leur propre type de base.

Le type de base de chaque type XML Schema est défini dans la section de ce chapitre lui correspondant : section 7.4.7 pour les types prédéfinis, section 7.4.10 pour les types simples, section 7.4.12 pour les attributs de types complexes, section 7.4.14 pour les éléments racine, section 7.4.16 pour les éléments de types complexes et section 7.4.17 pour les types complexes.

7.4.4 Les conversions

La norme BPEL [Com01, Sec 8.4.1] spécifie qu'une variable BPEL peut recevoir une valeur de n'importe quel type. En réalité, la norme BPEL considère trois sortes de valeurs possibles : les valeurs de types complexes, les valeurs de types simples et les attributs, et explique comment convertir chaque sorte de valeur vers les deux autres. Ces conversions, en BPEL, sont effectuées lors de l'exécution du service Web. Il s'agit d'un premier type de conversions implicites. Il existe un second type de conversions implicites, introduites par le langage BPEL, qui surviennent lorsque, par exemple, au sein de l'opérateur d'affectation de BPEL (`assign`, présenté à la section 10.1.24), une variable d'un type complexe T reçoit une valeur d'un type complexe T'. Un interpréteur BPEL ne vérifie que la valeur d'une variable appartient bien au domaine du type (type XML Schema, élément racine ou message WSDL) avec lequel cette variable a été déclarée seulement lorsque l'utilisateur le demande expressément à l'aide de la construction `validate` (présentée à la section 10.1.25). Dans ce cas, si la valeur de la variable n'appartient pas au domaine du type de la variable, alors l'interpréteur doit lever une exception. De plus, le langage XPATH, qui est utilisé dans BPEL pour les expressions, introduit aussi des conversions implicites car il autorise les comparaisons entre valeurs de types différents. Par conséquent, en plus de l'opérateur d'affectation, toutes les constructions BPEL qui manipulent des expressions XPATH sont susceptibles d'introduire des conversions implicites. Dans notre base d'exemples de 312 services BPEL, nous avons répertorié 310 conversions implicites (dont, par exemple, 14 conversions de string vers int et 27 conversions d'un type simple T vers un type simple T' défini par restriction sur T), soit presque une conversion par fichier.

Comme LOTOS NT est un langage fortement typé, notre approche de traduction, qui consiste à définir pour chaque type XML Schema un type LOTOS NT correspondant, nous oblige a rendre explicites les conversions implicites des langages BPEL et XPATH. Pour cela, nous devons définir, pour chaque pair de type T_1 et T_2, un moyen de convertir une valeur de T_1 en une valeur de T_2 et inversement.

Chaque service BPEL de notre base d'exemples ne déclare que 7 types XML Schema en moyenne, auxquels il faut ajouter les 30 types prédéfinis que nous avons choisi de traiter (cf. section 7.4.6). Par une approche naïve, il faudrait, pour un service BPEL déclarant n types :

- définir, dans une bibliothèque LOTOS NT, 30×29 fonctions pour assurer les conversions entre types prédéfinis et
- générer, pour chaque type t défini dans le service, $2 \times (30 + n - 1)$ fonctions pour assurer les conversions entre t et les autres types (prédéfinis ou non).

Cela donne au final, $2n^2 + 58n + 870$ fonctions de conversion.

Etant donné qu'il n'y a, en moyenne, qu'une conversion par service BPEL, un nombre si important de fonctions de conversion nous paraît inapproprié.

Afin de réduire le nombre de fonctions de conversions nécessaires, nous définissons, pour chaque type

T, les deux fonctions de conversion suivantes :

- $up_T : T \rightarrow base\ (T)$ qui convertit une valeur de T en une valeur de son type de base et
- $down_T : base\ (T) \rightarrow T$ qui convertit une valeur du type de base de T en une valeur de T.

Dans le cas général (à l'exception de certains cas relevant des éléments et des attributs, cf. sections 7.4.12 et 7.4.16), up_T effectue une conversion "naturelle" du type T vers son type de base : elle s'apparente donc à une fonction identité, et $down_T$, à la différence de up_T, peut lever une exception si la valeur de $domain\ (base\ (T))$ à convertir n'est pas comprise dans $domain\ (T)$. Par exemple, $down_{\mathsf{byte}} : \mathsf{long} \rightarrow \mathsf{byte}$ lèvera une exception si son argument n'est pas compris entre -128 et 127. De façon similaire, $down_{\mathsf{hexBinary}} : \mathsf{string} \rightarrow \mathsf{hexBinary}$ lèvera une exception si son argument contient au moins un caractère autre que `'0'..'9'` et `'A'..'F'`. Dans le cas général, nous définissons, pour un type T, une fonction $valid_T$ qui vérifie qu'une valeur du domaine de $base\ (T)$ appartient au domaine de T. Cette fonction nous permet d'écrire la plupart des fonctions de conversion du domaine de $base\ (T)$ vers T en s'inspirant du modèle générique présenté ci-dessous :

$down_T\ (V)\ =$
`si` $valid_T\ (V)$ `alors`
  V
`sinon`
  `lever` *erreur de domaine*

Si un type T est son propre type de base, alors la définition des fonctions up_T, $down_T$ et $valid_T$ n'est pas nécessaire.

Nous définissons aussi des fonctions de conversion entre types de base différents. Si T_1 et T_2 sont deux types de base distincts, alors nous notons $base_conv_{T_1 \rightarrow T_2}$ la fonction de conversion de T_1 vers T_2. Si la conversion n'est pas autorisée (par exemple, d'une valeur de type string vers un type complexe), nous choisissons de signaler une erreur fatale dès la traduction (et non pas lors de la compilation du code LOTOS NT généré) afin de prévenir l'utilisateur le plus rapidement possible. Les conversions permises entre types de base sont les suivantes :

- entre types de base des types simples et prédéfinis et
- entre types complexes et string

Lorsque la conversion entre T_1 et T_2 est permise, nous pouvons définir la conversion d'une valeur V de type T_1 vers le type T_2 comme suit :

$$conv_{T_1 \rightarrow T_2}\ (V)\ =\ down_{T_2}\ \left(base_conv_{T_1' \rightarrow T_2'}\ (up_{T_1}\ (V))\right)$$

où $T_1' = base\ (T_1)$ et $T_2' = base\ (T_2)$.

Au final, cette approche réduit considérablement le nombre de fonctions de conversion nécessaires. Si un service BPEL déclare n types (types prédéfinis inclus) et qu'il y a parmi ces types m types de base, alors, il faut $2n + m(m-1)$ fonctions de conversion. En pratique, le nombre de types de base (5 pour les 30 types prédéfinis, par exemple) est bien inférieur au nombre de types total, ce qui justifie le choix de cette approche.

7.4.5 Les itérateurs

Lors de la génération de l'espace d'états d'un service BPEL, il faut pouvoir énumérer, pour chaque réception de message, un ensemble de messages pouvant être reçus (en supposant que ce nombre reste fini, sans quoi l'espace d'états est infini). Les messages (qui sont définis dans le langage WSDL, cf. chapitre 9) sont des enregistrements construits à partir de types XML Schema. Il est donc nécessaire

de générer en LOTOS NT, pour chaque type XML Schema dont le domaine est fini, un itérateur qui va énumérer toutes les valeurs du domaine de ce type. Notez que l'itérateur d'un message se construit naturellement en réutilisant les itérateurs des types qui le composent.

En LOTOS NT, l'itérateur d'un type T à domaine fini est défini par un couple de fonctions (*premier* et *suivant*) :

- *premier* renvoie le premier élément du domaine de T et
- *suivant* reçoit une valeur du domaine de T et renvoie la valeur suivante dans l'énumération du domaine de T ainsi qu'une valeur booléenne indiquant si l'énumération est terminée.

Ce couple de fonctions induit implicitement une relation d'ordre sur le domaine de T. En pratique, *premier* et *suivant* sont choisies pour refléter l'ordre naturel sur T, lorsqu'il en existe un : ordre croissant pour les nombres, ordre lexicographique pour les types complexes (que l'on peut considérer comme des n-uplets)...

Le langage LOTOS NT peut s'interfacer avec le langage C. Cela nous permet de produire automatiquement du code C implantant les itérateurs et donc d'atteindre de bien meilleures performances qu'avec des itérateurs écrits en LOTOS NT.

Dans notre traduction, la plupart des types XML Schema sont équipés d'un itérateur. Les itérateurs des types définis par l'utilisateur sont générés automatiquement, tandis que les itérateurs des types prédéfinis ont été écrits à la main, en C, et sont fournis sous la forme d'une bibliothèque.

Le cas des types, tels que string, les types simples définis par restriction sur string ou encore les types complexes dont l'un des éléments ou attributs est de type string, dont le domaine est infini peut être traité de deux manières :

1. En définissant des itérateurs qui énumèrent seulement un sous-ensemble de valeurs du domaine d'un tel type, ce qui a pour conséquence de garantir la terminaison de la compilation du code LOTOS NT mais d'empêcher l'utilisateur d'accéder à des valeurs qui pourraient l'intéresser ;
2. En ne définissant pas d'itérateurs pour un tel type, cela oblige l'utilisateur, s'il souhaite manipuler ces types, à écrire lui-même des itérateurs ou bien, dans le cas d'un type simple T, à utiliser un type simple XML Schema défini par restriction sur T.

Nous avons choisi la seconde solution car nous pensons qu'il est impossible que le traducteur fasse automatiquement des choix judicieux quant aux sous-ensembles de valeurs à énumérer. L'analyse statique des services BPEL dans le but de déterminer des sous-ensembles de valeurs intéressantes serait une piste prometteuse, mais elle sort du cadre de cette étude.

7.4.6 Traduction des types prédéfinis

Nous ne traitons pas tous les types prédéfinis. Tout d'abord, les deux types spéciaux ne sont pas traduits car ce sont des types "génériques" : au demeurant, anySimpleType n'apparaît pas dans notre base d'exemples, tandis que anyType figure dans seulement deux exemples : le premier n'utilise pas le type faisant référence à anyType et le second ne peut être traité car il fait appel à une extension propriétaire de BPEL, définie par Oracle pour l'opérateur d'affectation.
Ensuite, les types NOTATION, ID, IDREF, IDREFS, ENTITY et ENTITIES sont ignorés car leur domaine de valeurs dépend du contenu du document XML dans lequel ils apparaissent. Or, cette notion de document XML n'existe pas en BPEL.
Enfin, nous ne traitons pas les types relatifs aux dates et aux durées, car LOTOS NT, dans sa version actuelle, ne possède pas de constructions liées au temps.

Pour les types prédéfinis de XML Schema que nous pouvons traduire, nous avons écrit une bibliothèque LOTOS NT dans laquelle, à chaque type prédéfini XML Schema correspond exactement un type LOTOS NT. Cette bibliothèque (co-développée avec Rémi Hérilier) comprend trois fichiers :

- `XML_V1.tnt` est un fichier C qui contient les implémentations des types prédéfinis XML Schema pour lesquels aucun type correspondant n'a pu être défini en LOTOS NT (short et byte, par exemple, car LOTOS NT ne contient que des types entiers non bornés).
- `XML_V1.fnt` est un fichier C qui contient les fonctions de conversion que nous avons préféré ne pas définir en LOTOS NT (par exemple, il est bien plus aisé d'écrire la conversion d'un nombre entier vers un nombre réel dans le langage C qu'en LOTOS NT) et les définitions d'itérateurs (en réalité, si le couple de fonctions *premier* et *second* est défini à l'aide de macros C, elles seront contenues dans `XML_V1.tnt`).
- `XML_V1.lnt` est un fichier LOTOS NT qui contient l'implémentation des types que nous avons définis en LOTOS NT, ainsi que les liaisons pour rendre les types et fonctions définis en C accessibles depuis les programmes LOTOS NT.

Voici quelques exemples d'implantation des types prédéfinis (sachant que tous les choix d'implémentation que nous avons faits pour cette bibliothèque sont modifiables par l'utilisateur) :

- le type boolean de XML Schema est implémenté par le type Bool de LOTOS NT.
- unsignedByte est implémenté par le type unsigned char du langage C, pour lequel nous avons défini un itérateur qui énumère toutes les valeurs entières comprises entre 0 et 255.
- Nous limitons les domaines de types entiers à des valeurs d'une taille maximale de 32 bits ce qui signifie que *domain* (long) = *domain* (int) et *domain* (unsignedLong) = *domain* (unsignedInt).
- Les types numériques XML Schema ayant un domaine infini sont implémentés par des types C bornés, ainsi, les types integer, nonNegativeInteger et decimal sont respectivement implémentés par int, unsigned int et float en C.
- Les types XML Schema dérivés des chaînes de caractères (normalizedString, hexBinary...) peuvent être implémentés de deux façons : soit par des types C construits sur des chaînes de caractères (traduction actuelle), soit par un indice dans une table de chaînes de caractères (fonctionnalité fournie par CADP) pouvant contenir des chaînes de caractères de longueur arbitraire, mais en nombre limité.

7.4.7 Conversion des types prédéfinis

La relation "type de base" pour les types prédéfinis XML Schema, qui est utile pour définir les fonctions de conversion, est donnée par la table 7.1.

Comme expliqué à la section 7.4.4, chaque type prédéfini T^p est équipé des fonctions de conversion suivantes :

- $up_{T^p} : T^p \rightarrow base\ (T^p)$
- $down_{T^p} : base\ (T^p) \rightarrow T^p$

Afin de vérifier que l'argument de $down_{T^p}$ est valide, nous utilisons la forme générique de la fonction de validation : $valid_{T^p} : base\ (T^p) \rightarrow \{\texttt{true}, \texttt{false}\}$ (telle que définie à la section 7.4.4). La table 7.2 détaille la définition de $valid_{T^p}$ pour chacun des types prédéfinis que nous traduisons. Cette fonction

Types prédéfinis Xml Schema	Type de base
boolean	boolean
nonNegativeInteger, positiveInteger, unsignedLong, unsignedInt, unsignedShort, unsignedByte	nonNegativeInteger
integer, nonPositiveInteger, negativeInteger, long, int, short, byte	integer
decimal, double, float	decimal
string, hexBinary, base64Binary, QName, anyURI, normalizedString, token, language, NMTOKEN, NMTOKENS, Name, NCName	string

Table 7.1: Types de base pour les types prédéfinis de Xml Schema

peut être réutilisée par la suite pour définir la fonction de validation de tout type obtenu par restriction sur T^p.

De plus, comme il est possible de définir un nouveau type T^s par restriction d'un type prédéfini T^p, il est nécessaire, comme nous le verrons à la section 7.4.9 de munir T^p de fonctions pour vérifier qu'une valeur du domaine de T^p satisfait les restrictions émises par T^s. Nous définissons donc :

- *match* (V, R_e) : prend en argument une valeur V de type T^p et une expression régulière R_e. Elle est utilisée pour valider les restrictions **pattern**.
- *length* (V) : prend en argument une valeur V de type T^p. Elle est utilisée pour valider les restrictions **length**, **minLength** et **maxLength**.
- *totalDigits* (V) : est définie uniquement pour les types numériques entiers et prend en argument une valeur V de type T^p. Elle est utilisée pour valider la restriction **totalDigits**.

Nous ne considérons pas la validation de la restriction **fractionDigits** car nous implémentons les nombres réels Xml Schema par des nombres à virgule flottante en C. Nous sommes donc incapable de dire combien de chiffres après la virgule un nombre réel possède, à la différence de Xml Schema qui supporte une précision infinie, grâce à la représentation des nombres à virgule flottante par des chaînes de caractères,

7.4.8 Itérateurs pour les types prédéfinis

Nous ne définissons pas d'itérateurs pour les types prédéfinis ayant un domaine infini. Cela laisse le choix à l'utilisateur de fournir son propre itérateur ou bien de créer un nouveau type par restriction afin d'isoler les valeurs qu'il souhaite manipuler. Pour les types prédéfinis dont le domaine est fini, nous définissons des itérateurs qui peuvent être modifiés par l'utilisateur. Les ensembles de valeurs énumérés par les itérateurs des types préféfinis sont listés dans la table 7.3.

7.4.9 Traduction des types simples

Nous avons choisi de ne traiter ni les listes, ni les unions. En effet, les listes sont rendues inutiles par le fait que le langage XPath (le langage d'expressions utilisé par Bpel et détaillé au chapitre 8) ne fournit aucune construction pour les manipuler. Les unions sont représentées par des chaînes de caractères et, comme le langage XPath permet la conversion de chaînes de caractères vers n'importe quel autre type de base, l'intérêt d'utiliser des unions s'en trouve très limité. Aussi, aucun type liste ni union n'apparaît dans notre base d'exemples.

Types prédéfinis XML Schema T	Définition de la fonction $valid_T\ (V)$
boolean, nonNegativeInteger, unsignedLong, unsignedInt, integer, long, int, decimal, double, float, string, anyURI	`true`
unsignedShort	$0 \leq V \leq 255$
unsignedByte	$0 \leq V \leq 65535$
positiveInteger	$V \geq 1$
byte	$-128 \leq V \leq 127$
short	$-32768 \leq V \leq 32767$
nonPositiveInteger	$V \leq 0$
negativeInteger	$V \leq -1$
hexBinary	$match\ (V, \texttt{"([0-9a-fA-F]2)*"})$
base64Binary	$match\ (V, \texttt{"((([A-Za-z0-9+/] ?){4})*}$ $\texttt{(([A-Za-z0-9+/] ?){3}[A-Za-z0-9+/]}$ $\texttt{|([A-Za-z0-9+/] ?){2} [AEIMQUYcgkosw048]?}$ $\texttt{=|[A-Za-z0-9+/] ?[AQgw] ?= ?=))?"})$
QName	$match\ (V, \texttt{"([a-zA-Z_][a-zA-Z0-9_\-\.]*:)?}$ $\texttt{[a-zA-Z_][a-zA-Z0-9_\-\.]*"})$
normalizedString	$\neg match\ (V, \texttt{"[\r\n\t]*"})$
token	$\neg(match\ (V, \texttt{"[\r\n\t]*"}) \vee$ $match\ (V, \texttt{"^ .*"}) \vee$ $match\ (V, \texttt{".* \$"}) \vee$ $match\ (V, \texttt{".* .*"}))$
language	$match\ (V, \texttt{"[a-zA-Z]\{1,8\}(-[a-zA-Z0-9]\{1,8\})*"})$
NMTOKEN	$match\ (V, \texttt{"[a-zA-Z0-9_\-:\.]+"})$
NMTOKENS	$match\ (V, \texttt{"[a-zA-Z0-9_\-:\.]+([a-zA-Z0-9_\-:\.]+)*"}))$
Name	$match\ (V, \texttt{"[a-zA-Z_:][a-zA-Z0-9_\-:\.]*"})$
NCName	$match\ (V, \texttt{"[a-zA-Z_][a-zA-Z0-9_\-\.]*"})$

Table 7.2: Validation des types prédéfinis

Le problème de la traduction des types simples XML Schema se ramène donc à la traduction des types simples obtenus par restriction. Un tel type, défini par :

$$\textbf{type}\ T^s\ =\ \textbf{restriction}\ (T^{ps}, Rest_0, ..., Rest_n)$$

est traduit par un type LOTOS NT. Ce type LOTOS NT est différent du type LOTOS NT correspondant à T^{ps}, cela permet d'écrire des itérateurs LOTOS NT différents pour T^s et T^{ps}.

Avant de traduire T^s, nous vérifions si une récursion se produit (c'est-à-dire si T^{ps} est dérivé, directement ou indirectement, de T^s). Si c'est le cas, nous signalons une erreur à la compilation, en conformité avec la norme XML Schema.

Dans le but de réduire la quantité de mémoire consommée lors de la génération d'une spécification BPEL, nous introduisons la notation *unrestricted* (T^{ps}), spécifique aux types simples, qui désigne le type prédéfini qui se trouve au sommet de la hiérarchie de types induite par la relation de restriction, c'est-à-dire le premier type sans restriction duquel dérive T^{ps} :

Types prédéfinis XML Schema	Valeurs énumérées par l'itérateur
boolean	`true`, `false`
unsignedByte	0, 1, ..., 254, 255
unsignedShort	0, 1, ..., 65534, 65535
positiveInteger	1, ..., 4294967294, 4294967295
nonNegativeInteger, unsignedLong, unsignedInt	0, 1, ..., 4294967294, 4294967295
byte	-128, -127, ..., 126, 127
short	-32768, -32767, ..., 32766, 32767
nonPositiveInteger	-2147483648, -2147483647, ..., -1, 0
negativeInteger	-2147483648, -2147483647, ..., -2, -1
integer, long, int	-2147483648, -2147483647, ..., 2147483646, 2147483647
decimal, double, float, string, hexBinary, base64Binary, QName, anyURI, normalizedString, token, language, NMTOKEN, NMTOKENS, Name, NCName	pas d'itérateur

Table 7.3: Itérateurs pour les types prédéfinis

unrestricted (T^{ps}) =
  `si` $T^{ps} = T^{p}$ `alors` T^{p}
  `sinon si` $T^{ps} =$ **restriction** $(T^{ps'}, Rest_0, ..., Rest_n)$ `alors` *unrestricted* $(T^{ps'})$
  `sinon` *cas non traité : liste ou union*

Par exemple, si l'on considère les types prédéfinis integer, short et le type smallShort défini ainsi :

```
<simpleType name="smallShort">
  <restriction base="short">
    <maxInclusive value="50" />
    <minInclusive value="-50" />
  </restriction>
</simpleType>
```

nous obtenons les égalités suivantes :

- *unrestricted* (integer) = integer
- *unrestricted* (short) = short
- *unrestricted* (smallShort) = short
- *base* (smallShort) = *base* (short) = *base* (integer) = integer

Cela nous permet d'économiser de la mémoire, par la traduction présentée ci-dessous, en représentant une valeur de type smallShort par une valeur de type *unrestricted* (smallShort), c'est-à-dire le type short dont chaque valeur tient sur un octet, plutôt que par une valeur de type *base* (smallShort) dont chaque valeur tient sur quatre octets. La traduction LOTOS NT d'un type simple est donc un enregistrement avec un champ unique f de type *unrestricted* (T^{ps}) :

`type` T^{s} `is`
  T^{s} `(`f`:` *unrestricted* (T^{ps})`)`
`end type`

Le type de base d'un type simple T^s défini par restriction est le type de base de *unrestricted* (T^s) :

$$base\ (T^s)\ =\ base\ (unrestricted\ (T^s))$$

7.4.10 Conversion des types simples

Les relations *unrestricted* (T) et *base* (T) nous permettent de définir les fonctions LOTOS NT dont nous équipons un type simple T^s défini par restriction de T^{ps} (**type** T^s = **restriction** ($T^{ps}, Rest_0, ..., Rest_n$)) :

- $valid_{T^s}$: *domain* (*unrestricted* (T^s)) $\rightarrow$ {`vrai`, `faux`} vérifie qu'une valeur du domaine de *unrestricted* (T^s) satisfait toutes les restrictions sur le chemin de types entre *unrestricted* (T^s) et T^s (conjonction de toutes les restrictions sur ce chemin). Cette validation fonctionne par appels récursifs, $valid_{T^s}$ vérifie les restrictions déclarées par le type T^s puis appel la fonction $valid_{T^{ps}}$ qui, à son tour, va vérifier les restrictions éventuellement déclarées par T^{ps} avant de passer la main à la fonction de validation du type duquel T^{ps} dérive et ainsi de suite. Si nous désignons la vérification d'une restriction *Rest* pour une valeur V (appartenant au domaine du type racine) par la notation : *Rest* (V), alors nous pouvons écrire :

  $$valid_{T^s}\ (V)\ =\ Rest_0\ (V) \wedge ... \wedge Rest_n\ (V) \wedge valid_{T^{ps}}\ (V)$$

- $down_{T^s}$: *domain* (*base* (T^s)) $\rightarrow$ *domain* (T^s) convertit une valeur du domaine du type de base en une valeur du domaine du type simple créé. Cette conversion se fait en deux étapes successives, tout d'abord, la valeur du type *base* (T^s) est convertie vers une valeur de type *unrestricted* (T^s) (par définition de *base* (T^s), cette fonction de conversion existe et peut lever une exception), ensuite, la valeur de *unrestricted* (T^s) obtenue est validée au moyen de $valid_{T^s}$ avant d'être finalement utilisée pour instancier une valeur de type T^s :

  $down_{T^s}\ (V)\ =$
  $\quad V'\ =\ down_{unrestricted\ (T^s)}\ (V)$
  $\quad$ `si` $valid_{T^s}\ (V')$ `alors`
  $\quad\quad T^s\ (V')$
  $\quad$ `sinon`
  $\quad\quad$ `lever` *erreur de domaine*

- up_{T^s} : *domain* (T^s) $\rightarrow$ *domain* (*base* (T^s)) qui convertit une valeur du domaine du type simple créé en une valeur du domaine du type de base. Cette conversion s'effectue en deux étapes successives, tout d'abord, la valeur de T^s est convertie vers une valeur de type *unrestricted* (T^s), pour cela, il suffit d'accéder au champ f de la valeur V de type T^s ($V\,.\,f$), ensuite, la valeur de type *unrestricted* (T^s) est convertie vers une valeur de type *base* (T^s) par un appel à la fonction $up_{unrestricted\ (T^s)}$:

  $$up_{T^s}\ (V)\ =\ up_{unrestricted\ (T^s)}\ (V.f)$$

7.4.11 Itérateurs pour les types simples

Afin d'obtenir des itérateurs efficaces, nous fusionnons en une seule liste de restrictions les différentes restrictions exprimées sur le chemin entre *unrestricted* (T^{ps}) et T^s. Par exemple, si un premier type T_1 est défini comme une restriction de `integer` avec **minInclusive** (V_1) et si un second type T_2 est défini comme une restriction de T_1 avec **maxInclusive** (V_2), alors l'itérateur de T_2 énumère toutes les valeurs énumérées par l'itérateur de T_1 qui sont inférieures à V_2. En applatissant la relation de

restriction, il est possible de définir un itérateur qui énumère directement les valeurs comprises entre V_1 et V_2, ce qui est plus efficace.

Nous définissons l'applatissement de la relation de restriction à l'aide des règles de réduction suivantes :

1. **enumeration** $(V_1, ..., V_n)$, **enumeration** $(V'_1, ..., V'_{n'}) \rightarrow$
   enumeration $(V''_1, ..., V''_{n''})$, $\{V''_1, ..., V''_{n''}\} = \{V_1, ..., V_n\} \cap \{V'_1, ..., V'_{n'}\}$
2. **pattern** $(V_1, ..., V_n)$, **pattern** $(V'_1, ..., V'_{n'}) \rightarrow$ **pattern** $(V_1, ..., V_n, V'_1, ..., V'_{n'})$
3. **length** (V_1), **length** $(V_2) \rightarrow$ **length** (V_1) : notez que si $V_1 \neq V_2$, c'est une erreur car il est interdit de redéfinir la longueur une fois qu'elle a été fixée.
4. **maxLength** (V_1), **maxLength** $(V_2) \rightarrow$ **maxLength** $(\min(V_1, V_2))$
5. **minLength** (V_1), **minLength** $(V_2) \rightarrow$ **minLength** $(\max(V_1, V_2))$
6. **maxExclusive** (V_1), **maxExclusive** $(V_2) \rightarrow$ **maxExclusive** $(\min(V_1, V_2))$
7. **maxInclusive** (V_1), **maxInclusive** $(V_2) \rightarrow$ **maxInclusive** $(\min(V_1, V_2))$
8. **minExclusive** (V_1), **minExclusive** $(V_2) \rightarrow$ **minExclusive** $(\max(V_1, V_2))$
9. **minInclusive** (V_1), **minInclusive** $(V_2) \rightarrow$ **minInclusive** $(\max(V_1, V_2))$
10. **maxExclusive** (V_1), **maxInclusive** $(V_2) \rightarrow$
    `si` $V_1 > V_2$ `alors` **maxExclusive** (V_1) `sinon` **maxInclusive** (V_2)
11. **minExclusive** (V_1), **minInclusive** $(V_2) \rightarrow$
    `si` $V_1 < V_2$ `alors` **minInclusive** (V_2) `sinon` **minExclusive** (V_1)
12. **fractionDigits** (V_1), **fractionDigits** $(V_2) \rightarrow$ **fractionDigits** $(\min(V_1, V_2))$: moins il y a de chiffres après la virgule, plus la restriction est forte, c'est pourquoi nous choisissons la valeur V la plus petite qui est contenue dans une restriction **fractionDigits** (V).
13. **totalDigits** (V_1), **totalDigits** $(V_2) \rightarrow$ **totalDigits** $(\min(V_1, V_2))$: moins il y a de chiffres dans un nombre, plus la restriction est forte, c'est pourquoi nous choisissons la valeur V la plus petite qui est contenue dans une restriction **totalDigits** (V).

Soient $Rest'_0, ..., Rest'_{n'}$, les restrictions obtenues apres réduction. Cette liste permet de construire l'itérateur de T^s comme suit :

- Si *base* (T^s) est égal à boolean, alors l'itérateur de T^s énumère les valeurs parmi `true` et `false` qui valident les restrictions *Rest'*$_0, ...,$ *Rest'*$_{n'}$.
- Sinon, s'il existe *Rest'* = **enumeration** $(V_0, ..., V_m)$ parmi *Rest'*$_0, ...$*Rest'*$_{n'}$, alors l'itérateur de T^s énumère toutes les valeurs de $\{V_0, ..., V_m\}$ qui valident *Rest'*$_0, ...$*Rest'*$_{n'}$ (*Rest'* est automatiquement validée).
- Sinon, si *base* (T^s) est égal à nonNegativeInteger ou integer, alors un itérateur peut être automatiquement construit pour énumérer toutes les valeurs, comprises entre la borne inférieure V^{inf} et la borne supérieure V^{sup} du domaine de T^s, qui valident les restrictions émises par T^s. S'il existe *Rest'* = **minExclusive** (V) parmi *Rest'*$_0, ...,$ *Rest'*$_{n'}$, alors $V^{\text{inf}} = V + 1$; sinon s'il existe *Rest'* = **minInclusive** (V) parmi *Rest'*$_0, ...,$ *Rest'*$_{n'}$, alors $V^{\text{inf}} = V$; sinon V^{inf} est égal à la borne inférieure du domaine de *unrestricted* (T^s). S'il existe *Rest'* = **maxExclusive** (V) parmi *Rest'*$_0, ...,$ *Rest'*$_{n'}$, alors $V^{\text{sup}} = V - 1$; sinon s'il existe *Rest'* = **maxInclusive** (V) parmi *Rest'*$_0, ...,$ *Rest'*$_{n'}$, alors $V^{\text{sup}} = V$; sinon V^{sup} est égal à la borne supérieure du domaine de *unrestricted* (T^s).

- Sinon, si *base* (T^s) est égal à decimal et si les restrictions **fractionDigits**, **minExclusive** ou **minInclusive** et **maxExclusive** ou **maxInclusive**, sont présentes, alors un itérateur pour T^s peut être automatiquement construit pour énumérer toutes les valeurs, comprises entre la borne inférieure $V^{\inf}$ et la borne supérieure $V^{\sup}$ du domaine de T^s, qui valident les restrictions émises par T^s. Le pas de l'itération est de 10^{-V^f}, où V^f est la valeur présente dans **fractionDigits** (V^f). S'il existe *Rest'* = **minExclusive** (V) parmi $Rest'_0, ..., Rest'_{n'}$, alors $V^{\inf} = V + 10^{-V^f}$; sinon, il existe *Rest'* = **minInclusive** (V) parmi $Rest'_0, ..., Rest'_{n'}$ et $V^{\inf} = V$. S'il existe *Rest'* = **maxExclusive** (V) parmi $Rest'_0, ..., Rest'_{n'}$, alors $V^{\sup} = V - 10^{-V^f}$; sinon il existe *Rest'* = **maxInclusive** (V) parmi $Rest'_0, ..., Rest'_{n'}$ et $V^{\sup} = V$.
- Pour tous les autres cas (c'est-à-dire ceux pour lesquels le domaine du type est infini), l'itérateur du type T^s n'est pas généré automatiquement et devra être écrit par l'utilisateur. En règle générale, nous conseillons d'éviter d'utiliser ce genre de types de données pour la vérification formelle et d'utiliser exclusivement des types avec un domaine borné, obtenus par restriction si nécessaire.

7.4.12 Traduction des attributs des types complexes

La définition d'un attribut A dans une définition de type complexe XML Schema est de la forme suivante :

$$A \ : \ Attr$$

où :

$$Attr \ = \ \textbf{attribute} \ (T^{ps}, V^{optional}, V^{fixed}, V^{default})$$

Soit T_A le type de l'attribut **attribute** ($T^{ps}, V^{optional}, V^{fixed}, V^{default}$). Le type de base de T_A est toujours le type de base de T^{ps} :

$$base \ (T_A) \ = \ base \ (T^{ps})$$

Chaque définition d'un attribut de type T_A dans la définition d'un type complexe T^c engendre, dans la définition de type LOTOS NT correspondant à T^c, un champ dont le type LOTOS NT est la traduction de T_A. Ce type dépend des valeurs de $V^{optional}$, V^{fixed} et $V^{default}$ et est équipé des fonctions de conversions :

- up_{T_A} : $domain \ (T_A) \rightarrow domain \ (base \ (T_A))$
- $down_{T_A}$: $domain \ (base \ (T_A)) \rightarrow domain \ (T_A)$

V^{fixed} et $V^{default}$ ne pouvant apparaître en même temps, et $V^{default}$ ne pouvant apparaître qu'en présence de $V^{optional}$, il n'y a que 5 cas possibles (et non 8) :

1. Si $V^{optional}$ est faux et que ni V^{fixed} ni $V^{default}$ n'est présent, alors $T_A \ = \ T^{ps}$, c'est-à-dire que le type du champ LOTOS NT, correspondant à A dans la traduction LOTOS NT d'un type complexe, est le type LOTOS NT correspondant à T^{ps}. Les fonctions de conversion de T_A sont, dans ce cas, celles de T^{ps}.
2. Si $V^{optional}$ est vrai et que ni V^{fixed} ni $V^{default}$ n'est présent, alors T_A est un nouveau type construit LOTOS NT ayant deux constructeurs : T_A^{absent} (), qui représente le cas où l'attribut n'est pas présent, et $T_A^{present}$ ($f : T^{ps}$), qui représente le cas où l'attribut est présent :

```
type T_A is
  T_A^absent (),
  T_A^present (f : T^ps)
end type
```

Les fonctions de conversion de ce nouveau type sont définies ainsi :

$up_{T_A}\ (V)\ =$
  si $V = T_A^{absent}\ ()$ **alors**
    lever *valeur non présente*
  sinon
    $up_{T}ps\ (V.f)$

$down_{T_A}\ (V)\ = T_A^{present}\ (down_{T}ps\ (V))$

3. Si $V^{optional}$ est vrai et que $V^{default}$ est présent, alors, T_A est un nouveau type LOTOS NT dont la définition est la même qu'au point précédent, à la différence près que up_{T_A} ne lève pas d'erreur si la valeur n'est pas présente mais retourne $V^{default}$.

4. Si $V^{optional}$ est faux et que V^{fixed} est présent, alors il est nécessaire d'introduire un nouveau type XML Schema. Soit $T^{ps'} = \textbf{restriction}\ (T^{ps}, \textbf{enumeration}\ (V^{fixed}))$ (la traduction en LOTOS NT d'un tel type a été traitée à la section 7.4.9), alors $T_A = T^{ps'}$, c'est-à-dire que le type du champ LOTOS NT, correspondant à A dans la traduction LOTOS NT d'un type complexe, est le type LOTOS NT correspondant à $T^{ps'}$. Les fonctions de conversion de T_A sont, dans ce cas, celles de $T^{ps'}$.

5. Si $V^{optional}$ est vrai et que V^{fixed} est présent, alors, il est nécessaire d'introduire un nouveau type XML Schema $T^{ps'}$, défini de la même façon qu'au point précédent. T_A est alors défini ainsi :

```
type T_A is
  T_A^absent (),
  T_A^present (f : T^ps')
end type
```

Les fonctions de conversion de T_A sont :

$up_{T_A}\ (V)\ =$
  si $V = T_A^{absent}\ ()$ **alors**
    lever *valeur non présente*
  sinon
    $up_{T}ps',\ (V.f)$

$down_{T_A}\ (V)\ = T_A^{present}\ (down_{T}ps',\ (V))$

7.4.13 Traduction des éléments racine

Dans une déclaration d'élément racine :

$$\textbf{type}\ T^e\ =\ Elem$$

où :

$$Elem \ = \ \textbf{element} \ (T^{psc}, V^{fixed}, V^{default}, V^{minOccurs}, V^{maxOccurs})$$

les valeurs booléennes de $V^{minOccurs}$ et $V^{maxOccurs}$ sont égales à 1 et V^{fixed} et $V^{default}$ absentes (notation $\perp$). La syntaxe de XML Schema interdit l'utilisation de ces attributs dans la définition d'un élément racine qui se ramène alors à la définition suivante :

$$Elem \ = \ \textbf{Element} \ (T^{psc}, \perp, \perp, 1, 1)$$

Comme XML Schema considère que les types T^e et T^{psc} sont différents, nous traduisons T^e par un type LOTOS NT distinct du type LOTOS NT correspondant à T^{psc} :

```
type T^e is
  T^e (f : T^psc)
end type
```

7.4.14 Conversion des éléments racine

Le type de base de T^e est le type de base de T^{psc} :

$$base \ (T^e) \ = \ base \ (T^{psc})$$

Les fonctions de conversion sont définies comme suit :

- $up_{T^e} \ (V) \ = \ up_{T^{psc}} \ (V.f)$, conversion en deux étapes d'une valeur du type T^e vers une valeur du type T^{psc} qui est ensuite convertie vers une valeur du type $base \ (T^e)$.
- $down_{T^e} \ (V) \ = \ T^e \ (down_{T^{psc}} \ (V))$, cette fonction de conversion ne lève des exceptions que si $down_{T^{psc}}$ en lève.

7.4.15 Itérateurs pour les éléments racine

L'itérateur d'un type élément est généré automatiquement par LNT2LOTOS (en réalité par CÆSAR.ADT auquel LNT2LOTOS délègue cette tâche) à partir de l'itérateur de T^{psc}. Si le domaine de T^{psc} est constitué des valeurs $V_0...V_n$, alors le domaine de T^e est constitué des valeurs $T^e \ (V_0)...T^e \ (V_n)$.

7.4.16 Traduction des éléments des types complexes

La définition d'un élément E au sein d'une définition de type complexe XML Schema est de la forme suivante :

$$E \ : \ Elem$$

où :

$$Elem \ = \ \textbf{element} \ (T^{psc}, V^{fixed}, V^{default}, V^{minOccurs}, V^{maxOccurs})$$

La différence entre un élément déclaré au sein d'un type complexe et un élément racine réside dans les valeurs que peuvent prendre V^{fixed}, $V^{default}$, $V^{minOccurs}$ et $V^{maxOccurs}$. Dans le cas d'un élément de type complexe, V^{fixed} et $V^{default}$ peuvent être présents tandis que $0 \leq V^{minOccurs} \leq V^{maxOccurs} \leq +\infty$.

Soit T_E le type de l'élément **element** $(T^{psc}, V^{fixed}, V^{default}, V^{minOccurs}, V^{maxOccurs})$. La traduction de T_E en LOTOS NT est similaire à la traduction du type d'un attribut (cf. section 7.4.12). En effet, comme les valeurs de $V^{minOccurs}$ et $V^{maxOccurs}$ peuvent rendre un élément optionel ou

obligatoire, nous pouvons distinguer les cinq mêmes cas que pour les attributs. L'unique différence vient du cas où $V^{maxOccurs}$ est strictement supérieur à 1. Il faut alors distinguer un sixiéme cas dans lequel, T_E est défini au moyen d'un type liste LOTOS NT. Pour cette raison, le type de base de T_E dépend, en premier lieu, de la valeur de $V^{maxOccurs}$:

base (T_E) =
  `si` $V^{maxOccurs} > 1$ `alors`
    T_E
  `sinon`
    base (T^{psc})

Chaque définition d'élément dans la définition d'un type complexe T^c engendre, dans la définition de type LOTOS NT correspondant à T^c, un champ dont le type LOTOS NT est la traduction de T_E. Ce type dépend des valeurs de V^{fixed}, $V^{default}$, $V^{minOccurs}$ et $V^{maxOccurs}$ et sera équipé des fonctions de conversion :

- up_{T_E} : *domain* $(T_E) \rightarrow$ *domain* (*base* (T_E))
- $down_{T_E}$: *domain* (*base* $(T_E)) \rightarrow$ *domain* (T_E)

Nous distinguons 6 cas selon les valeurs que prennent V^{fixed}, $V^{default}$, $V^{minOccurs}$ et $V^{maxOccurs}$ (V^{fixed} et $V^{default}$ sont mutuellement exclusives et ne peuvent apparaître si $V^{maxOccurs}$ est supérieur à 1) :

1. Si $V^{minOccurs}$ et $V^{maxOccurs}$ sont égaux à 1, et que ni V^{fixed}, ni $V^{default}$ n'est présent, alors $T_E = T^{psc}$. Les fonctions de conversions pour T_E sont celles de T^{psc}.
2. Si $V^{minOccurs}$ est égal à 0, $V^{maxOccurs}$ égal à 1 et que ni V^{fixed}, ni $V^{default}$ n'est présent, alors nous procédons comme au point 2 (pour les attributs) de la section 7.4.12 pour créer un type LOTOS NT qui encapsule une valeur optionnelle de type T^{psc}.
3. Si $V^{minOccurs}$ est égal à 0, $V^{maxOccurs}$ égal à 1 et que $V^{default}$ est présent, alors nous procédons comme au point 3 (pour les attributs) de la section 7.4.12 pour créer un type LOTOS NT qui encapsule une valeur optionnelle de type T^{psc} et dont la fonction de conversion up_{T_E} utilise $V^{default}$ lorsque la valeur n'est pas présente.
4. Si $V^{minOccurs}$ est égal à 1, $V^{maxOccurs}$ égal à 1 et que V^{fixed} est présent, alors nous procédons comme au point 4 (pour les attributs) de la section 7.4.12 pour créer un type LOTOS NT dont les valeurs sont égales à V^{fixed}.
5. Si $V^{minOccurs}$ est égal à 0, $V^{maxOccurs}$ égal à 1 et que V^{fixed} est présent, alors nous procédons comme au point 5 (pour les attributs) de la section 7.4.12 pour créer un type LOTOS NT qui encapsule une valeur constante optionnelle qui vaut V^{fixed}.
6. Si $V^{maxOccurs} > 1$ alors, T_E est un type LOTOS NT qui encode une liste d'éléments de type T^{psc} :

   `type` T_E `is`
     `list of` T^{psc}
   `end type`

   Dans ce cas, T_E est son propre type de base, donc nous ne définissons pas les fonctions de conversion up_{T_E} et $down_{T_E}$. En revanche, T_E est équipé des fonctions suivantes :

- **setnth** : *domain* (T_E), *domain* (nonNegativeInteger), *domain* $(T^{psc}) \rightarrow$ *domain* (T_E) permet de modifier un élément de la liste,
- **getnth** : *domain* (T_E), *domain* (nonNegativeInteger) $\rightarrow$ *domain* (T_E) permet d'accéder à un élément de la liste,
- **count** : *domain* $(T_E) \rightarrow$ *domain* (nonNegativeInteger) compte le nombre d'éléments dans la liste,
- **sum** : *domain* $(T_E) \rightarrow$ *domain* $(base\ (T^{psc}))$ calcule la somme des éléments contenus dans la liste, si *base* (T^{psc}) est decimal, integer ou nonNegativeInteger.
- $base_conv_{T_E \rightarrow \mathsf{string}}$: $T_E \rightarrow$ string retourne la concaténation des conversions en chaîne de caractères des éléments contenus dans la liste.

En BPEL, la longueur d'une liste d'éléments est fixe. En effet, une fois déclarée, une valeur de type liste ne peut se voir ajouter ou retirer des élément, seule la modification d'un élément est permise. Pour cette raison, nous ne définissons pas de fonctions pour ajouter ou retirer des éléments.

L'itérateur de T_E énumère toutes les listes qui satisfont les conditions suivantes :

- longueur supérieure ou égale à $V^{minOccurs}$,
- longueur inférieure ou égale à $V^{maxOccurs}$ et
- les valeurs des éléments de la liste appartiennent à *domain* (T^{psc}).

Si $V^{maxOccurs}$ est égale à "unbounded" (c'est-à-dire si le domaine de T^{psc} n'est pas borné), nous ne définissons pas d'itérateur.

7.4.17 Traduction des types complexes

Chaque définition de type complexe T^c est traduite par un type LOTOS NT pour lequel nous définissons une fonction de conversion vers les chaînes de caractères. Cette fonction est nécessaire pour effectuer les comparaisons entre valeurs dans le langage XPATH (cf. section 8.4). Nous distinguons quatre cas dans notre traduction, selon que T^c a un contenu simple (**extension**) ou un contenu complexe, auquel cas il y a trois agencements possibles pour les éléments que définit T^e : **sequence**, **all** ou **choice**. Nous détaillons ces différentes traductions ci-dessous :

- **type** T^c = **extension** $(T^s, A_0 : Attr_0, ..., A_n : Attr_n)$ est traduit en LOTOS NT par le type construit suivant :

```
type T^c is
  T^c (f : T^s, A_0 : T_{A_0}, ..., A_n : T_{A_n})
end type
```

  où T_{A_i} est le type LOTOS NT résultant de la traduction de $A_i : Attr_i$.

  $base_conv_{T^c \rightarrow \mathsf{string}}$ consiste alors en la conversion en chaîne de caractères du contenu simple f.

- **type** T^c = **sequence** $(E_0 : Elem_0, ..., E_m : Elem_m, A_1 : Attr_1, ..., A_n : Attr_n)$ est traduit en LOTOS NT par le type construit suivant :

```
type T^c is
  T^c (E_0 : T_{E_0}, ...E_m : T_{E_m}, A_0 : T_{A_0}, ..., A_n : T_{A_n})
end type
```

où T_{E_i} est le type LOTOS NT résultant de la traduction de $E_i : Elem_i$.

$base_conv_{T^c \to \mathsf{string}}$ consiste alors en la concaténation des conversions en chaîne de caractères des éléments $E_0, ..., E_n$.

- **type** T^c = **all** $(E_0 : Elem_0, ..., E_m : Elem_m, A_1 : Attr_1, ..., A_n : Attr_n)$ est traduit en LOTOS NT par le type construit suivant :

```
type T^c is
  T^c (E_0 : T_{E_0}, ...E_m : T_{E_m}, l : NatList, A_0 : T_{A_0}, ..., A_n : T_{A_n})
end type
```

où l est une liste d'entiers naturels qui donne l'ordre dans lequel les éléments apparaissent. Par exemple, la liste $0, 3, 2, 1, 4, 2$ indique que le premier élément est un E_0, le second un E_3, le troisième un E_2. ... Cette liste d'entiers naturels sert au calcul de la conversion de T^c en chaîne de caractères ($base_conv_{T^c \to \mathsf{string}}$). En effet, la norme XPATH [XG99, Sec. 3.4] précise que lors de la conversion d'une valeur V de type complexe en chaîne de caractères, il faut impérativement effectuer la concaténation des sous-éléments de V en préservant l'ordre dans lequel ces sous-éléments apparaissent dans la représentation XML de V.

- **type** T^c = **choice** $(E_0 : Elem_0, ..., E_m : Elem_m, A_1 : Attr_1, ..., A_n : Attr_n)$ est traduit en LOTOS NT par le type construit suivant :

```
type T^c is
  T_0^c (E_0 : T_{E_0}, A_0 : T_{A_0}, ..., A_n : T_{A_n}),
  ...
  T_m^c (E_m : T_{E_m}, A_0 : T_{A_0}, ..., A_n : T_{A_n}),
end type
```

$base_conv_{T^c \to \mathsf{string}}$ consiste alors en la conversion en chaîne de caractères de l'élément, parmi $E_0, ..., E_n$, contenu dans la valeur de type T^c.

Les types complexes sont leur propre type de base :

$$base\ (T^c) \ = \ T^c$$

Nous ne définissons donc pas de fonctions de conversion up_{T^c} et $down_{T^c}$. Les itérateurs des types complexes sont automatiquement générés par CÆSAR.ADT à l'aide des itérateurs existants pour les types de leurs éléments et attributs.

7.5 Traduction des constantes BPEL

Dans cette section, nous définissons un algorithme de traduction pour les constantes BPEL. Une constante BPEL est un terme XML qui satisfait le sous-ensemble de XML Schema que nous considérons et qui apparaît entre les bases `<literal>` et `</literal>` en partie droite de l'opérateur d'affection "`assign`" de BPEL (cf. section 10.1.24).

La traduction des constantes est fastidieuse et le lecteur pressé pourra ignorer la fin de cette section pour se rendre directement au chapitre 8.

A notre connaissance, aucune des approches de vérification de services Web BPEL ne prend en compte la traduction des constantes.

A partir de chaque constante Bpel, nous générons une expression Lotos NT constituée de constructeurs de types. Ces constructeurs sont issus de la traduction de Xml Schema vers Lotos NT et permettent d'instancier les types Xml Schema utilisés pour définir la constante.

Voici une constante Bpel notée L (*Literal*) extraite de notre base d'exemples, dont la valeur est le terme Xml contenu dans l'élément "`literal`", et avec laquelle nous donnons d'abord une idée intuitive de la traduction des constantes, avant de présenter le principe général de cette traduction.

```
<literal>
  <LoanApproverResponse>
    <responseToLoanRequest>
      approved
    </responseToLoanRequest>
  </LoanApproverResponse>
</literal>
```

Supposons maintenant que l'élément racine `LoanApproverResponse` soit défini en Xml Schema de la façon suivante :

```
<element name="LoanApproverResponse" type="LoanApprovalResponseType" />
<complexType name="LoanApprovalResponseType">
  <sequence>
    <element name="responseToLoanRequest" type="ResponseValue" />
  </sequence>
</complexType>
<simpleType name="ResponseValue">
  <restriction base="string">
    <enumeration value="approved" />
    <enumeration value="declined" />
    <enumeration value="underReview" />
  </restriction>
</simpleType>
```

Selon l'algorithme présenté à la section 7.4, la traduction en Lotos NT de cette définition Xml Schema est la suivante :

```
type LoanApproverResponse_Element is
  LoanApproverResponse_Element (f:LoanApprovalResponse_Type)
end type
type LoanApprovalResponseType_Type is
  LoanApprovalResponseType_Type (responseToLoanRequest:ResponseValue_Type)
end type
type ResponseValue_Type is
  ResponseValue_Type (f:String)
end type
```

La constante Bpel L déclarée précédemment (dont la valeur est le terme Xml contenu dans l'élément "`literal`") est traduite en Lotos NT par l'expression Lotos NT suivante :

```
LoanApproverResponse_Element (
  LoanApprovalResponseType_Type (
    down_ResponseValue ("approved")
  )
)
```

où `down_ResponseValue` est la fonction de conversion qui transforme une valeur de type "`string`" (le type de base de "`ResponseValue`") en une valeur de type `ResponseValue`.

7.5.1 Grammaire des constantes BPEL

Nous définissons à présent une grammaire abstraite des constantes Bpel. La norme Bpel [Com01, Sec. 8.4] accepte, entre les balises `<literal>` et `</literal>`, soit une valeur V de type simple représentée par une chaîne de caractères, soit un élément E. Comme une constante Bpel est un terme Xml, cette valeur ou cet élément désigne le nœud racine de la représentation arborescente du terme. Le type d'un élément E (cf. section 7.3) peut être choisi parmi :

- les types simples (qui comprennent les types prédéfinis), auquel cas E a la forme :
  <E> V </E>
- les types complexes à contenu simple, auquel cas E a la forme :
  <E A_0="V_0" ... A_n="V_n">" V </E>
- les types complexes à contenu complexe, auquel cas E a la forme :
  <E A_1="V_1" ... A_n="V_n">"
  $Element_0$
  ...
  $Element_m$
  </E>

La grammaire abstraite qui suit ramène ces trois cas à seulement deux, en remarquant que le premier peut devenir un cas particulier du second si on remplace A_0 par A_1. *Literal* est l'axiome principal de la grammaire :

$$
\begin{array}{rcl}
\mathit{Literal} & ::= & \textbf{SimpleConstant}\ (V) \\
 & | & \textbf{ElementConstant}\ (E, \mathit{Content}) \\
\mathit{Content} & ::= & \textbf{SimpleContent}\ (A_1 = V_1, ..., A_n = V_n, V) \\
 & | & \textbf{ComplexContent}\ (A_1 = V_1, ..., A_n = V_n, E_0 : \mathit{Content}_0, ..., E_n : \mathit{Content}_m)
\end{array}
$$

7.5.2 Principe général de traduction des constantes

La traduction d'une constante Bpel L s'effectue par un parcours en profondeur infixe, de gauche à droite, de l'arbre Xml correspondant à la constante. La traduction de chaque nœud n de cet arbre nécessite la propagation d'un type cible T (un attribut hérité). Initialement (pour le nœud racine), il s'agit du type T de la variable ou du champ de variable destiné à recevoir la valeur de la constante (dans l'opérateur d'affectation de Bpel). Ensuite, la traduction de chaque champ de la constante s'effectue en propageant le type du champ correspondant dans T. Si, lors d'une étape de la traduction, un champ de la constante Bpel est d'un type T' différent du type T propagé mais qu'il est existe une fonction de conversion entre T' et T, alors ce champ est converti vers T. Sinon, une erreur signalant l'incompatibilité des types est signalée.

Nous allons à présent définir formellement la traduction en Lotos NT, notée $[\![\mathit{Literal}]\!]^{\mathit{Const}}\ (T)$, d'une constante L ayant T pour type cible.

Les règles de la grammaire présentée à la section précédente étant mutuellement récursives, le plan de ce chapitre ne peut éviter certaines références en avant :

- A la section 7.5.3, nous détaillons la traduction d'une constante de type simple (**SimpleConstant**).
- A la section 7.5.4, nous détaillons la traduction d'une constante de type élément (**ElementConstant**).
- A la section 7.5.5, nous détaillons la traduction d'un attribut dans le contenu d'une constante ($A = V$).
- A la section 7.5.6, nous détaillons la traduction des attributs dans le contenu d'une constante ($A_1 = V_1, ..., A_n = V_n$).
- A la section 7.5.7, nous détaillons la traduction d'un sous-élément dans le contenu d'une constante (E : *Content*).
- A la section 7.5.8, nous détaillons la traduction des sous-éléments dans le contenu d'une constante (E_0 : $Content_0, ..., E_m$: $Content_m$).
- A la section 7.5.9, nous détaillons la traduction du contenu simple d'une constante (*SimpleContent*).
- A la section 7.5.10, nous détaillons la traduction du contenu complexe d'une constante (*ComplexContent*).

7.5.3 Traduction des constantes de type simple

Dans cette section, nous anticipons sur la traduction des expressions XPath vue au chapitre 8 afin de faciliter la traduction des constantes de type simple. En effet, la traduction d'une constante V de type simple consiste à créer une expression XPath constituée de la constante V uniquement pour ensuite utiliser l'algorithme de traduction des expressions XPath que nous définissons plus loin, à la section 8.6 :

$[\![Literal]\!]^{Const}\ (T)$ = -- premier cas : type simple
`si` *Literal* est de la forme **SimpleConstant** (V) `alors`
`faire` *décideConversion* (V, T) `dans` $[\![V]\!]^{D}$
...

Ici, T désigne le type cible pour la traduction de la constante V. *décideConversion* (V, T) est une procédure (définieà la section 8.6.3) qui calcule un booléen destiné à déclencher une éventuelle conversion dans la traduction de l'expression en Lotos NT. La notation $[\![V]\!]^{D}$ exprime la traduction de l'expression XPath V et est définie à la section 8.6.4.

7.5.4 Traduction des constantes de type élément racine

La traduction d'une constante de type élément dont le contenu est *Content* ne peut s'effectuer que si le type cible T est un type élément racine :

$[\![Literal]\!]^{Const}\ (T)\ =$ ...-- deuxième cas : type élément
  sinon -- *Literal* est forcément de la forme **ElementConstant** (E, *Content*)
    si T ne désigne pas un type élément racine T^e alors
      erreur
    sinon si E n'a pas le même nom que T^e alors
      erreur
    sinon
      soit T^{psc} tel que
        def $(T^e)\ =$ **element** $(T^{psc}, ...)$
      dans
        T^e $([\![Content]\!]^{Const}\ (T^{psc}))$

Nous commençons par nous assurer que T désigne un type élément racine que nous notons T^e. Ensuite, nous vérifions que l'identificateur de l'élément E est égal à l'identificateur du type élément racine T^e. Ensuite, nous utilisons le constructeur LOTOS NT T^e pour instancier une nouvelle valeur de type T^e. Les arguments de cet appel au constructeur T^e seront donnés par la traduction en LOTOS NT du contenu de l'élément avec T^{psc} pour type cible. T^{psc} désigne le type spécifié dans la définition du type élément racine T^e (notée *def* (T^e)) pour décrire le contenu de tout élément racine de type T^e. Enfin, nous appelons la traduction du contenu de l'élément qui est notée $[\![Content]\!]^{Const}\ (T^{psc})$ et définie aux sections 7.5.9 et 7.5.10.

7.5.5 Traduction d'un attribut dans le contenu d'une constante

En XML Schema, un type complexe peut contenir des déclarations d'attributs (*Attr*) dont nous rappelons ici la syntaxe :

$$Attr = \textbf{attribute}\ (T^{ps}, V^{optional}, V^{fixed}, V^{default})$$

Nous avons vu à la section 7.4.12 que, selon les valeurs de $V^{optional}$, V^{fixed} et $V^{default}$, il existe cinq traductions possibles pour produire en LOTOS NT le type T_A qui va encoder la valeur d'un attribut défini par *Attr*.

Dans une constante BPEL de type complexe, la clause "*A*=*V*" indique que l'attribut se voit affecter la valeur V. Il s'agit donc de générer l'expression LOTOS NT qui va instancier le constructeur T_A du type T_A (homonyme) en donnant à son argument unique la valeur V.

Nous notons $[\![A]\!]^{Attr}\ (Attr, V)$ la traduction de l'attribut A, déclaré en XML Schema avec *Attr* et associé dans une constante BPEL à la valeur V. Ci-dessous, nous énumérons, dans le même ordre qu'à la section 7.4.12, les différentes traductions possibles selon les valeurs de $V^{optional}$, V^{fixed} et $V^{default}$ dans *Attr* et la valeur V associée à A :

1. $[\![A]\!]^{Attr}\ (\textbf{attribute}\ (T^{ps}, \texttt{faux}, \bot, \bot), V)\ =$
   si $V\ = \bot$ alors
     erreur
   sinon
     T^{ps} $([\![V]\!]^{Const}\ (T^{ps}))$
2. $[\![A]\!]^{Attr}\ (\textbf{attribute}\ (T^{ps}, \texttt{vrai}, \bot, \bot), V)\ =$

```
si V = ⊥ alors
  T_A^absent ()
sinon
  T_A^present ([[V]]^Const (T^ps))
```

3. $[\![A]\!]^{Attr}$ (**attribute** $(T^{ps}, \texttt{vrai}, \bot, V^{default}), V)$ =

```
si V = ⊥ alors
  T_A^present ([[V^default]]^Const (T^ps))
sinon
  T_A^present ([[V]]^Const (T^ps))
```

4. $[\![A]\!]^{Attr}$ (**attribute** $(T^{ps}, \texttt{faux}, V^{fixed}, \bot), V)$ =

```
si V = ⊥ alors
  erreur
sinon si V ≠ V^fixed alors
  erreur
sinon
  T_A ([[V]]^Const (T^ps))
```

5. $[\![A]\!]^{Attr}$ (**attribute** $(T^{ps}, \texttt{vrai}, V^{fixed}, \bot), V)$ =

```
si V = ⊥ alors
  T_A^absent ()
sinon
  soit T^ps' = restriction (T^ps, enumeration (V^fixed)) dans
    T_A^present (T^ps' ([[V]]^Const (T^ps)))
```

7.5.6 Traduction des attributs dans le contenu d'une constante

Un type complexe T^c (cf. section 7.3) peut déclarer une liste d'attributs et spécifier les types des valeurs de ces attributs. Ces attributs peuvent ensuite être utilisés dans des constantes, ayant un contenu de type T^c, qui leur associent des valeurs.

Soient $A'_1 = V_1, ..., A'_{n'} = V_{n'}$ la liste des attributs utilisés dans une constante L de type complexe T^c et $A_1 : Attr_1, ..., A_n : Attr_n$ les déclarations de ces attributs dans T^c.

L'ordre d'apparition des attributs dans L peut être différent de celui dans lequel ils sont déclarés dans T^c. De plus, si certains attributs sont optionnels, ils peuvent être omis dans L. Il faut donc s'assurer que la liste d'attributs $A'_1 = V_1, ..., A'_{n'} = V_{n'}$ utilisée dans L est valide par rapport à la liste d'attributs $A_1 : Attr_1, ..., A_n : Attr_n$ déclarée par T^c :

1. Le nombre d'attributs utilisés par L ne peut être strictement supérieur au nombre d'attributs déclarés dans T^c, donc $'$ doit être supérieur ou égal à n'.

2. Les attributs $A'_1, ..., A'_{n'}$ de L doivent être deux à deux distincts (les occurrences multiples sont interdites).

3. A chaque utilisation d'un attribut A'_i, avec $i \in [1...n']$, dans L doit correspondre une déclaration d'attribut $A_j : Attr_j$, avec $j \in [1...n]$ dans T^c (les identificateurs A'_i et A_j devant être identiques).

En LOTOS NT, un type complexe T^c est traduit par un type construit (cf. section 7.4.17) qui possède un ou plusieurs constructeurs. Les n derniers arguments de chaque constructeur correspondent aux

attributs $A_1, ..., A_n$ déclarés dans T^c. La traduction des attributs $A'_1 = V_1, ..., A'_{n'} = V_{n'}$ utilisés dans une constante ayant un contenu complexe de type T^c consiste donc à produire, pour chaque attribut A_j, avec $j \in [1...n]$, une expression LOTOS NT représentant la valeur associée à A_j par la constante. Les différentes expressions LOTOS NT ainsi obtenues sont alors assemblées comme une liste d'arguments à fournir au constructeur LOTOS NT de T^c afin de spécifier la valeur des arguments de la constante de type T^c en cours de traduction.

Nous notons $[\![A'_1 = V_1, ..., A'_{n'} = V_{n'}]\!]^{Attr}\ (A_1 : Attr_1, ..., A_n : Attr_n)$ la traduction en LOTOS NT des attributs $A'_1 = V_1, ..., A'_{n'} = V_{n'}$ utilisés dans L et déclarés dans T^c par $A_1 : Attr_1, ..., A_n : Attr_n$. Cette traduction est définie ainsi :

$[\![A'_1 = V_1 ... A'_{n'} = V_{n'}]\!]^{Attr}\ (A_1 : Attr_1 ... A_n : Attr_n)\ =$
  `si` $n' > n$ `alors`
    erreur `--` condition 1
  `sinon si` $\exists\ i, j \in [1...n'] \mid A'_i = A'_j$ `alors`
    erreur `--` condition 2
  `sinon si` $\exists\ i \in [1...n'], \nexists\ j \in [1...n] \mid A'_i = A_j$ `alors`
    erreur `--` condition 3
  `sinon`
    `soit` $valeur : \{A_1 ... A_n\} \rightarrow \{V_1 ... V_n\} \cup \{\bot\} \mid$
      $\forall i \in [1...n], valeur\ (A_i)\ =$
        `si` $\exists\ j \in [1...n'] \mid A'_j = A_i$ `alors`
          V_j
        `sinon`
          $\bot$
    `dans`
      $[\![A_1]\!]^{Attr}\ (Attr_1, valeur\ (A_1))$`,...,`$[\![A_n]\!]^{Attr}\ (Attr_{n'}, valeur\ (A_n))$

Les trois conditions précédemment énoncées sont d'abord vérifiées. Puis, une fonction *valeur* est définie, qui associe à chaque attribut A_j, avec $j \in [1..n]$, une valeur qui est, soit indéfinie si $A_j \notin \{A'_1, ..., A'_{n'}\}$, soit égale à la valeur V_i s'il existe $i \in [1...n']$ tel que $A_j = A'_i$. L'attribut A_j est alors traduit en une expression LOTOS NT selon le schéma de traduction présenté à la section précédente. Enfin, les expressions LOTOS NT ainsi obtenues sont assemblées en une liste d'arguments.

7.5.7 Traduction d'un sous-élément dans le contenu d'une constante

En XML Schema, un type complexe peut contenir des déclarations de sous-éléments (*Elem*) dont nous rappelons ici la syntaxe :

$$Elem = \textbf{element}\ (T^{psc}, V^{fixed}, V^{default}, V^{minOccurs}, V^{default})$$

Nous avons vu à la section 7.4.16 que, selon les valeurs de V^{fixed}, $V^{default}$, $V^{minOccurs}$ et $V^{maxOccurs}$, il existe six traductions possibles pour produire en LOTOS NT le type T^e qui encode le contenu d'un élément E.

Dans une constante BPEL ayant un contenu type complexe, un même élément peut avoir de multiples occurrences (selon les valeurs de $V^{minOccurs}$ et $V^{maxOccurs}$) et, par conséquent, se voir attribuer plusieurs contenus $Content_1, ..., Content_n$ (un contenu par occurrence de l'élément). Il s'agit donc de générer l'expression LOTOS NT qui instancie le constructeur T_E du type T_E, auquel on passe comme liste d'arguments les traductions des contenus $Content_1, ..., Content_n$.

Nous notons $[\![E]\!]^{Elem}\ (Elem, Content_1, ..., Content_n)$ la traduction du sous-élément E, déclaré en XML Schema avec *Elem* et associé, au sein une constante BPEL, aux contenus $Content_1, ..., Content_n$.

Ci-dessous, nous énumérons les différentes traductions possibles selon les valeurs de V^{fixed}, $V^{default}$, $V^{minOccurs}$ et $V^{maxOccurs}$ dans *Elem*, dans le même ordre qu'à la section 7.4.12 :

1. $[\![E]\!]^{Elem}$ (**element** $(T^{psc}, \bot, \bot, 1, 1), Content_1 ... Content_n)$ =

   `si` $n \neq 1$ `alors`
   erreur `--` n forcément compris entre $V^{minOccurs}$ et $V^{maxOccurs}$
   `sinon`
   $[\![Content_1]\!]^{Const}$ (T^{psc})

2. $[\![E]\!]^{Elem}$ (**element** $(T^{psc}, \bot, \bot, 0, 1), Content_1 ... Content_n)$

   `si` $n = 0$ `alors`
   T_E^{absent}
   `sinon si` $n = 1$ `alors`
   $T_E^{present}$ `(`$[\![Content_1]\!]^{Const}$ (T^{psc})`)`
   `sinon`
   erreur `--` n forcément compris entre $V^{minOccurs}$ et $V^{maxOccurs}$

3. $[\![E]\!]^{Elem}$ (**element** $(T^{psc}, \bot, V^{default}, 0, 1), Content_1 ... Content_n)$ =

   `si` $n = 0$ `alors`
   $[\![V^{default}]\!]^{Const}$ (T^{psc})
   `sinon si` $n = 1$ `alors`
   $T_E^{present}$ `(`$[\![Content_1]\!]^{Const}$ (T^{psc})`)`
   `sinon`
   erreur `--` n forcément compris entre $V^{minOccurs}$ et $V^{maxOccurs}$

4. $[\![E]\!]^{Elem}$ (**element** $(T^{psc}, V^{fixed}, \bot, 1, 1), Content_1 ... Content_n)$ =

   T^{psc} désigne forcément un type simple ou prédéfini T^s (cf. section 7.4.16)
   `si` $n \neq 1$ `alors`
   erreur `--` n forcément compris entre $V^{minOccurs}$ et $V^{maxOccurs}$
   `sinon si` $Content_1$ n'est pas de la forme **SimpleContent** $(A_1 = V_1, ..., A_n = V_n, V)$ `alors`
   erreur `--` car V^{fixed} impose un contenu simple pour l'élément
   `sinon si` $V \neq V^{fixed}$ `alors`
   erreur `--` la valeur du contenu est fixée
   `sinon`
   `soit` $T^{ps'}$ = **restriction** $(T^{ps}$, **enumeration** $(V^{fixed}))$ `dans`
   T_E `(`$T^{ps'}$ `(`$[\![V]\!]^{Const}$ (T^{ps})`))`

5. $[\![E]\!]^{Elem}$ (**element** $(T^{psc}, V^{fixed}, \bot, 0, 1), Content_1 ... Content_n)$ =

-- T^{psc} désigne forcément un type simple ou prédéfini T^{ps} (cf. section 7.4.16)
si $n = 0$ alors
  T_E^{absent}
sinon si $n \neq 1$ alors
  erreur -- n forcément compris entre $V^{minOccurs}$ et $V^{maxOccurs}$
sinon si $Content_1$ n'est pas de la forme **SimpleContent** $(A_1 = V_1, ..., A_n = V_n, V)$ alors
  erreur -- car V^{fixed} force un contenu simple pour l'élément
sinon si $V \neq V^{fixed}$ alors
  erreur -- la valeur du contenu est fixée
sinon
  soit $T^{ps'} =$ **restriction** $(T^{ps}, \textbf{enumeration}\ (V^{fixed}))$ dans
    $T_E^{present}\ (T^{ps'}\ (\llbracket V \rrbracket^{Const}\ (T^{ps})))$

6. $\llbracket E \rrbracket^{Elem}$ (**element** $(T^{psc}, \perp, \perp, V^{minOccurs}, V^{maxOccurs}), Content_1 ... Content_n) =$
   si $V^{minOccurs} \leq n \leq V^{maxOccurs}$ alors
     $\{\llbracket Content_1 \rrbracket^{Const}\ (T^{psc}), ..., \llbracket Content_n \rrbracket^{Const}\ (T^{psc})\}$
   sinon
     erreur

7.5.8 Traduction des sous-éléments dans le contenu d'une constante

Dans cette section, nous abordons la traduction des sous-éléments d'une constante L dont le contenu est défini au moyen d'un type complexe T^c à contenu complexe (définis à la section 7.2.6).

Soient $E_0 : Content_0, ..., E_m : Content_m$ les occurrences des sous-éléments dans le contenu de la constante L, et $E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'}$ la déclaration de ces sous-éléments dans T^c.

La traduction des occurrences des sous-éléments $E_0 : Content_0, ..., E_m : Content_m$ est constituée de trois étapes successives :

1. Selon que les sous-éléments $E'_0, ..., E'_{m'}$ ont déclarés à l'intérieur d'une balise "`sequence`", "`all`" ou "`choice`", différentes conditions doivent être satisfaites pour valider la correspondance des éléments $E_0, ..., E_m$ présents par rapport à leurs déclarations $E'_0, ..., E'_{m'}$. Ces différentes conditions sont détaillées plus loin dans la présente section.

2. A chaque sous-élément E'_j, avec $j \in [0...m']$, sont associés les contenus $Content_{j,0}, ..., Content_{j,m^j}$ des occurrences de E'_j parmi $E_0, ..., E_m$.

3. Chaque sous-élément E'_j, avec $j \in [0...m']$, engendre une expression LOTOS NT selon la traduction définie à la section 7.5.7. Ces expressions sont alors assemblées afin de produire la séquence d'arguments à passer au constructeur de T^c utilisé lors de la traduction du contenu de la constante L.

Avant de la définir formellement, nous illustrons la traduction au moyen d'un type "`libraryType`" très simplifié et décrit par les types XML Schema suivants :

```
<element name="library" type="libraryType" />
<complexType name="libraryType">
  <sequence>
    <element name="book" type="string" minOccurs="0" maxOccurs="unbounded" />
    <element name="journal" type="string" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

Ici, l'ordre d'apparition des sous-éléments est figé par l'utilisation de la balise "`sequence`" et le nombre d'occurrences de chaque sous-élément est contrôlé par les attributs "`minOccurs`" ($V^{minOccurs}$) et "`maxOccurs`" ($V^{maxOccurs}$).

Voici une constante de type "`libraryType`" :

```
<library>
  <book>Oliver Twist</book>
  <book>Le Comte de Monte Cristo</book>
  <journal>New York Times</journal>
  <journal>Le Monde</journal>
</library>
```

L'ordre et le nombre d'occurrences des sous-éléments respectent la définition du type "`libraryType`". Nous pouvons maintenant associer aux sous-éléments "`book`" et "`journal`" les contenus apparaissant dans leurs occurrences :

- `book` : `Oliver Twist`, `Le Comte de Monte Cristo`
- `journal` : `New York Times`, `Le Monde`

Ensuite, chaque sous-élément est ensuite traduit en une expression LOTOS NT selon le schéma de traduction présenté à la section 7.5.7 :

- pour "`book`" : {`"Oliver Twist", "Le Comte de Monte Cristo"`}
- pour "`journal`" : {`"New York Times", "Le Monde"`}

où {...} désigne le constructeur de liste du langage LOTOS NT.

Enfin, la liste de valeurs destinée à être insérée dans le constructeur du type "`libraryType`" en LOTOS NT est produite par l'assemblage suivant :

{`"Oliver Twist", "Le Comte de Monte Cristo"`}, {`"New York Times", "Le Monde"`}

Nous allons, à présent, définir formellement la traduction des sous-éléments du contenu complexe d'une constante.

Les conditions à satisfaire par le nombre et l'ordre d'occurrence de chaque sous-élément $E'_j, j \in [0...m']$ dans $E_0, ..., E_m$ dépendent de la définition de E'_j dans le type complexe. Nous distinguons trois cas :

- Si les sous-éléments sont définis dans une balise "`sequence`", alors les conditions à satisfaire sont les suivantes :
  1. Pour chaque occurrence d'un sous-élément E_i, avec $i \in [0...m]$, il doit y avoir un sous-élément parmi $E'_0, ..., E'_{m'}$ dont l'identificateur soit égal à celui de E_i.
  2. Chaque sous-elément E'_i, avec $i \in [0...m']$, déclaré par T^c peut apparaître à plusieurs reprises au sein de $E_0, ..., E_m$ de E. Dans ce cas, toutes les occurrences de E'_i dans $E_0, ..., E_m$ doivent être consécutives, c'est-à-dire groupées les unes à la suite des autres.

3. L'ordre d'occurrence des sous-éléments dans $E_0, ..., E_m$ doit respecter l'ordre de déclaration des sous-éléments $E'_0, ..., E'_{m'}$ dans le type complexe.

Soit *conditionsSequence* $(E_0, ..., E_m, E'_0, ..., E'_{m'})$ la fonction qui prend en entrée les sous-éléments d'un type complexe T^c et leurs occurrences dans une constante de type T^c. Cette fonction renvoie la valeur "`vrai`" si les occurrences des sous-éléments satisfont la déclaration des sous-éléments au sein d'une balise "`sequence`", et la valeur "`faux`" sinon. Cette fonction est définie comme suit :

conditionsSequence $(E_0, ..., E_m, E'_0, ..., E'_{m'})$ $=$
```
-- condition 1
si ∀ i ∈ [0...m], ∃ j ∈ [0...m'] | E'_j = E_i
-- condition 2
et ∀ i, j ∈ [0...m], i ≤ j + 1 | E_i = E_j ⇒ ∄ k ∈ [i + 1...j − 1] | E_k ≠ E_i
-- condition 3
et ∀ i ∈ [0...m], i' ∈ [0...m'] E_i = E'_i' ⇒
     ∄ j ∈ [0...m], j' ∈ [0...m'] | E_j = E_j' ∧ j < i ∧ j' > i' alors
  renvoie vrai
sinon
  renvoie faux
```

- Si les sous-éléments sont définis dans une balise "`all`", il suffit de s'assurer que chaque pour chaque occurrence d'un sous-élément E_i, avec $i \in [0...m]$, il existe parmi $E'_0, ...E'_{m'}$ un élément ayant le même identificateur que E_i. Soit *conditionsAll* $(E_0, ..., E_m, E'_0, ..., E'_{m'})$ la fonction qui vérifie cette condition :

conditionsAll $(E_0, ..., E_m, E'_0, ..., E'_{m'})$ $=$
```
si ∀ i ∈ [0...m], ∃ j ∈ [0...m'] | E'_j = E_i alors
  renvoie vrai
sinon
  renvoie faux
```

- Si les sous-éléments sont définis dans une balise "`choice`", les deux conditions suivantes doivent être satisfaites :

1. Tous les identificateurs des sous-éléments $E_0, ..., E_m$ doivent être égaux.
2. Il existe un sous-élément E' parmi $E'_0, ..., E'_{m'}$ dont l'identificateur soit égal à l'identificateur des sous-éléments $E_0, ..., E_m$.

Soit *conditionsChoice* $(E_0, ..., E_m, E'_0, ..., E'_{m'})$ la fonction qui vérifie ces deux conditions :

conditionsChoice $(E_0, ..., E_m, E'_0, ..., E'_{m'})$ $=$
```
-- condition 1
si ∀ i, j ∈ [0...m], E_i = E_j
-- condition 2
et ∃ i ∈ [0...m'] | E'_i = E_0 alors
  renvoie vrai
sinon
  renvoie faux
```

Une fois vérifiées les conditions sur les occurrences $E_0, ..., E_m$ des sous-éléments $E'_0, ..., E'_{m'}$, nous pouvons associer à chaque sous-élément E'_j, avec $j \in [0...m']$, les contenus $Content_0, ..., Content_m$ de ses occurrences, qui figurent parmi $E_0, ..., E_m$. A cette fin, nous définissons une fonction *contenu* $(E, E_0 : Content_0, ..., E_m : Content_m)$ qui établit cette correspondance :

$$
\begin{array}{l}
contenu\ (E, E_0 : Content_0, ..., E_m : Content_m)\ = \\
\quad Content_{i^1} ... Content_{i^n}\ \texttt{où} \\
\quad\quad i^1, ..., i^n \in [0...m]\ |\ (i^1 < ... < i^n) \wedge (Ei' = E_{i^1} = ... = E_{i^n})
\end{array}
$$

Nous pouvons maintenant traduire les sous-éléments $E'_0, ..., E'_{m'}$ et produire les expressions LOTOS NT représentant leurs occurrences au sein du contenu complexe d'une constante.

- Nous notons $[\![E_0 : Content_0, ..., E_m : Content_m]\!]^{Sequence}\ (E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'})$ la traduction des sous-éléments $E'_0, ..., E'_{m'}$ déclarés dans une balise "`sequence`" à l'aide de $Elem_0, ..., Elem_{m'}$ et dont les occurrences dans une constante sont $E_0 : Content_0, ..., E_m : Content_m$. Nous définissons cette traduction ainsi :

$$
\begin{array}{l}
[\![E_0 : Content_0, ..., E_m : Content_m]\!]^{Sequence}\ (E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'})\ = \\
\quad \texttt{si}\ conditionsSequence\ (E_0, ..., E_m, E'_0, ..., E'_{m'})\ \neq\ \texttt{vrai alors} \\
\quad\quad erreur \\
\quad \texttt{sinon} \\
\quad\quad [\![E'_0]\!]^{Elem}\ (Elem_0, contenu\ (E'_0, E_0 : Content_0, ..., E_m : Content_m)) \\
\quad\quad \texttt{,...,} \\
\quad\quad [\![E'_{m'}]\!]^{Elem}\ (Elem_{m'}, contenu\ (E'_0, E_0 : Content_0, ..., E_m : Content_m))
\end{array}
$$

- Le cas des sous-éléments déclarés dans une balise "`all`" est plus complexe car en plus des expressions LOTOS NT représentant les contenus des différents sous-éléments, il faut aussi fournir une liste d'entiers qui indique l'ordre d'occurrence de chaque sous-élément dans le contenu de la constante (cf. section 7.4.17). Pour cela, nous introduisons une fonction *indice* qui associe à une occurrence E_j, avec $j \in [0...m]$, la position $i \in [0...m']$ de sa déclaration E'_i :

$$
\begin{array}{l}
[\![E_0 : Content_0, ..., E_m : Content_m]\!]^{All}\ (E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'})\ = \\
\quad \texttt{si}\ conditionsAll\ (E_0, ..., E_m, E'_0, ..., E'_{m'})\ \neq\ \texttt{vrai alors} \\
\quad\quad erreur \\
\quad \texttt{sinon} \\
\quad\quad \texttt{soit}\ indice : \{E_0, ...E_m\} \rightarrow \{0...m'\}, \\
\quad\quad\quad indice\ (E)\ = i \in [0...m']|E'_i = E \\
\quad\quad \texttt{dans} \\
\quad\quad\quad [\![E'_0]\!]^{Elem}\ (Elem_0, contenu\ (E'_0, E_0 : Content_0, ..., E_m : Content_m)) \\
\quad\quad\quad \texttt{,...,} \\
\quad\quad\quad [\![E'_{m'}]\!]^{Elem}\ (Elem_n, contenu\ (E'_0, E_0 : Content_0, ..., E_m : Content_m))\texttt{,} \\
\quad\quad\quad \{indice\ (E_0)\texttt{,...,}indice\ (E_m)\}
\end{array}
$$

- Le cas des sous-éléments déclarés dans une balise "`choice`" est le plus simple :

$[\![E_0 : Content_0, ..., E_m : Content_m]\!]^{Choice}\ (E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'})\ =$
  `si` *conditionsChoice* $(E_0, ..., E_m, E'_0, ..., E'_{m'})\ \neq$ `vrai alors`
    erreur
  `sinon`
    `soit` $i \in [0...m'] | E'_i = E_0$ `dans`
      $[\![E_i]\!]^{Elem}\ (Elem_i, Content_0, ..., Content_m)$

7.5.9 Traduction du contenu simple d'une constante

La traduction en LOTOS NT du contenu simple "**SimpleContent** $(A_1 = V_1, ..., A_n = V_n, V)$" d'une constante L dépend du type cible T vers lequel la constante doit être traduite :

- Si T désigne un type simple ou un type prédéfini T^{ps}, alors nous vérifions que la liste d'attributs de la constante est vide (car un type simple ou prédéfini ne définit pas d'attributs) avant d'appeler récursivement l'algorithme de traduction sur V avec T^{ps} pour type cible. C'est dans ce nouvel appel que sera vérifiée la compatibilité entre le type de la valeur V et T^{ps}.
- Si T désigne un type complexe T^c à contenu simple, c'est-à-dire l'extension d'un type simple T^{ps} avec des déclarations d'attributs $A'_0 : Attr_0, ..., A'_{n'} : Attr_{n'}$ (cf. section 7.2.5), alors nous commençons par traduire les attributs $A'_0, ..., A'_{n'}$ en fonction des valeurs qui leur sont associées dans la constante (par la liste d'associations $A_1 = V_1, ..., A_n = V_n$). Ensuite, nous traduisons la valeur V en appelant récursivement l'algorithme de traduction avec T^{ps} pour type cible.

Formellement, la traduction du contenu simple d'une constante est la suivante :

$[\![$**SimpleContent** $(A_1 = V_1, ..., A_n = V_n, V)]\!]^{Const}\ (T)\ =$
  `si` T désigne un type simple ou prédéfini T^{ps} `alors`
    `si` $n \neq 0$ `alors`
      erreur
    `sinon`
      $[\![V]\!]^{Const}\ (T^{ps})$
  `sinon si` T désigne un type complexe T^c `alors`
    `si` *def* $(T^c)\ =$ **extension** $(T^{ps}, A_0 : Attr_0 ... A_n : Attr_n)$ `alors`
      $T^c\ ([\![V]\!]^{Const}\ (T^{ps}), [\![A_1 = V_1 ... A_n = V_n]\!]^{Attr}\ (A'_0 : Attr_0 ... A'_{n'} : Attr_{n'}))$
    `sinon`
      erreur
  `sinon`
    erreur

7.5.10 Traduction du contenu complexe d'une constante

Le contenu complexe d'une constante est noté "**ComplexContent** $(A_1 = V_1, ..., A_n = V_n, E_0 : Content_0, ..., E_n : Content_m)$" et est défini par un type complexe T^c (cf. section 7.2.4).

La traduction d'un tel contenu réutilise les différentes notations introduites au long de cette section pour la traduction des attributs et des sous-éléments. Selon la définition de T^c, nous distinguons trois cas : **sequence**, **all** ou **choice**.

pour lesquels nous réutilisons les résultats des sections 7.5.6 et 7.5.8.

Formellement, la traduction du contenu complexe d'une constante est la suivante :

$[\![$**ComplexContent** $(E_0 : Content_0, ..., E_m : Content_m, A_1 : V_1, ..., A_n : V_n)]\!]^{Const}$ (T) $=$
`si` T ne désigne pas un type complexe T^c `alors`
erreur
`sinon si` *def* (T^c) $=$ **extension** $(V, A'_0 : Attr_0, ..., A'_{n'} : Attr_{n'})$ `alors`
erreur
`sinon si` *def* (T^c) $=$ **sequence** $(E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'}, A'_1 : Attr_1, ..., A'_{n'} : Attr_{n'})$ `alors`
T^c `(`$[\![E_0 : Content_0, ..., E_m : Content_m]\!]^{Sequence}$ $(E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'})$`,`
$[\![A_1 : V_1, ..., A_n : V_n]\!]^{Attr}$ $(A'_1 : Attr_1, ..., A'_{n'} : Attr_{n'})$`)`
`sinon si` *def* (T^c) $=$ **all** $(E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'}, A'_1 : Attr_1, ..., A'_{n'} : Attr_{n'})$ `alors`
T^c `(`$[\![E_0 : Content_0, ..., E_m : Content_m]\!]^{All}$ $(E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'})$`,`
$[\![A_1 : V_1, ..., A_n : V_n]\!]^{Attr}$ $(A'_1 : Attr_1, ..., A'_{n'} : Attr_{n'})$`)`
`sinon`
`--` T^c a forcément la forme
`--` **choice** $(E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'}, A'_1 : Attr_1, ..., A'_{n'} : Attr_{n'})$
`si` $\exists\ i \in [0...m'] \mid \forall\ j \in [0...m], E'_i = E_j$ `alors`
T_i^c `(`$[\![E_0 : Content_0, ..., E_m : Content_m]\!]^{Choice}$ $(E'_0 : Elem_0, ..., E'_{m'} : Elem_{m'})$`,`
$[\![A_1 : V_1, ..., A_n : V_n]\!]^{Attr}$ $(A'_1 : Attr_1, ..., A'_{n'} : Attr_{n'})$`)`
`sinon`
erreur

Chapitre 8

Traduction des expressions XPath

XPath [XG99] est un langage d'expressions recommandé par le W3C pour l'extraction d'informations dans des documents Xml. Contrairement à Xml Schema, Wsdl et Bpel, la syntaxe de XPath n'est pas basée sur Xml. En effet, les concepteurs de XPath ont voulu une syntaxe concise, de façon à pouvoir faire figurer des expressions XPath comme valeurs d'attributs dans des langages à syntaxe Xml, tels que XPointer [Gro03a] et Xslt [Gro03b], pour lesquels XPath avait à l'origine été défini, ou encore Bpel qui est l'objet de notre étude.

Bien qu'une version 2.0 [XG07] (2007) de la norme XPath ait été publiée, nous nous intéressons ici à la version 1.0 (1999) car il s'agit de celle utilisée par Bpel. La norme XPath 2.0 est une extension de la norme XPath 1.0 (une expression XPath 1.0 est donc valide dans la norme 2.0) qui amène les nouveautés principales suivantes : support des types Xml Schema, structures de contrôles (`if`, `while`, `for`...) et quantificateurs (`some` et `every`).

Dans ce chapitre, nous commençons par définir le modèle de données utilisé par un interpréteur XPath pour représenter le document Xml sur lequel une expression XPath est évaluée (section 8.1). Puis, nous expliquons comment XPath partitionne un document Xml en vue d'en extraire des valeurs (section 8.2). Ensuite, nous présentons, de façon concise et formelle et dans un ordre rigoureux, la syntaxe (section 8.3), la sémantique statique normalisée (section 8.4) et la sémantique dynamique normalisée (section 8.5) de la norme XPath qui compte 32 pages dans lesquelles la syntaxe et la sémantique du langage sont mélangées. Enfin, nous dressons un état de l'art des travaux visant à prendre en compte les expressions XPath dans le cadre de la vérification formelle de service Web Bpel (section 8.6.1), avant d'isoler un sous-ensemble simplifié de XPath qui regroupe les constructions qui sont réellement utiles à la définition de services Web (section 8.6.2), de proposer un nouvel algorithme de typage pour ces constructions (section 8.6.3), en tenant compte des déclarations de types Xml Schema, et de donner la traduction de ces constructions en Lotos NT (section 8.6.4).

8.1 Modèle de données

Une expression XPath est interprétée sur un arbre Xml représentant un terme ou un document Xml. Cet arbre est enraciné et formellement défini par le quadruplet $(\Sigma, n_0, \rightarrow, <_{\texttt{doc}})$ où :

- Σ est un ensemble de nœuds,
- n_0 est le nœud racine,

- $\rightarrow$: $\Sigma \times \Sigma$ est une relation de transition dans laquelle il existe un chemin simple unique entre chaque nœud de Σ et n_0.
- $<_{\texttt{doc}}$ est une relation d'ordre total strict sur Σ qui correspond intuitivement au parcours infixe de l'arbre, de gauche à droite.

Chaque nœud n de l'arbre appartient à une sorte de nœud, que l'on dénote par *sorte* (n), selon le terme XML (cf. section 7.1) que ce nœud représente. XPATH distingue 7 sortes de nœuds :

- *élément* pour les nœuds représentant des éléments XML,
- *attribut* pour les nœuds représentant des attributs XML
- *texte* pour les nœuds représentant des valeurs XML de type simple d'éléments à contenu simple,
- *commentaire* pour les nœuds représentant des commentaires XML,
- *instruction de traitement* pour les nœuds représentant des instructions de traitement XML et
- *espace de noms* pour les noeuds représentant des espaces de noms XML.
- *racine* pour le nœud spécial ayant comme fils, l'élément racine du document ou terme XML ainsi que les instructions de traitement et les commentaires figurant avant ou après l'élément racine.

Seuls les nœuds appartenant aux sortes *élément* et *élément racine* peuvent avoir des sous-nœuds.

L'exemple qui suit illustre les différents types de nœuds qui peuvent figurer dans un arbre XML. Une représentation graphique de cet arbre est donné par la figure 8.1.

```
<E1 A1="V1">
  <E2 xmlns:p="Ns">
    <E3>Va</E3>
    <?I?>
    <E4>Vb</E4>
  </E2>
  <!--C1-->
  <E5 A2="V2">
    <E6>Vc</E6>
    <!--C2-->
    <E7 A3="V3">Vd</E7>
  </E5>
</E1>
```

XPATH traite les attributs de la forme `xmlns:`p`="`N_s`"` comme des déclarations d'espaces de noms [Gro09]. Les nœuds représentant de tels attributs n'appartiennent pas à la sorte *attribut*, mais à la sorte *espace de noms*. En XML, si un élément E possède un attribut de la forme `xmlns:`p`="`N_s`"`, alors dans E, ainsi que dans les éléments $E_1 ... E_n$ contenus (directement ou indirectement) dans E, les noms (noms d'éléments, noms d'attributs...) appartenant à l'espace de noms N_s vont être préfixés par p`:`. Une déclaration d'espace de noms dans un élément XML E est donc propagée dans les sous-éléments $E_1 ... E_n$ de E. Dans le modèle de données considéré par XPATH, cette propagation se traduit par la présence, dans le nœud représentant E ainsi que dans ceux représentant $E_1 ... E_n$, d'un sous-nœud représentant la déclaration d'espace de noms `xmlns:`p`="`N_s`"`, sauf dans le cas où une nouvelle déclaration d'espace de noms vient associer N_s à un nouveau préfixe p'.

Tous les nœuds, à l'exception de ceux dont la sorte est *texte* ou *commentaire*, possèdent un identificateur. L'identificateur N d'un nœud n est donné par la notation $id\ (n)$. Chaque identificateur est

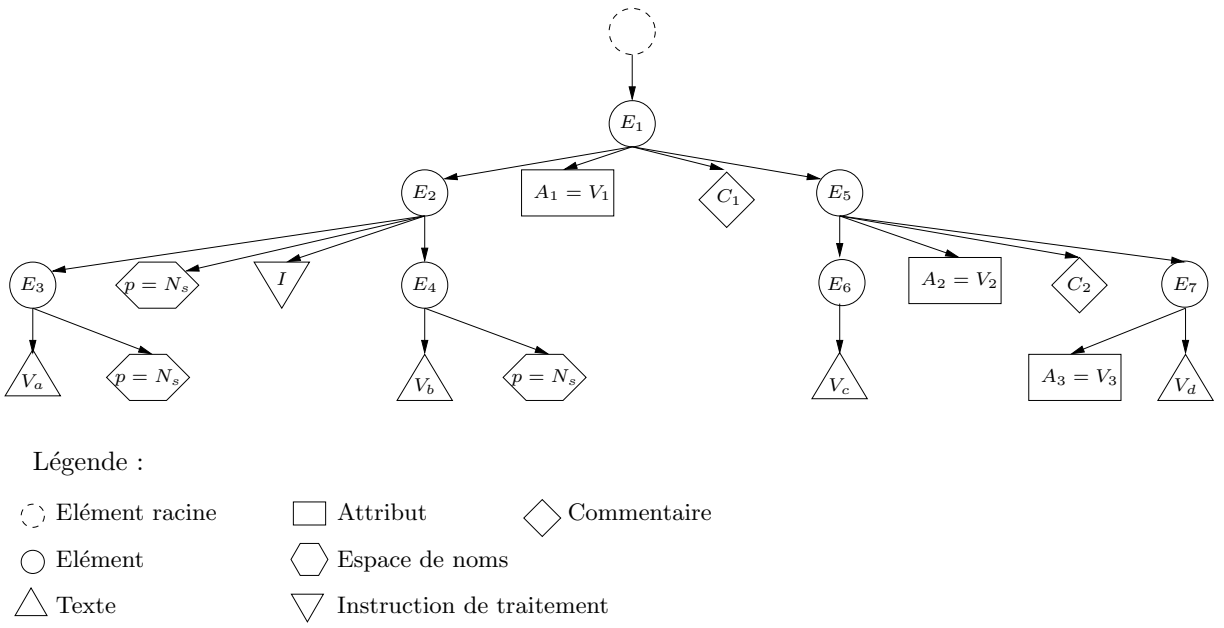

Figure 8.1: Exemple d'arbre Xml

composé de deux parties : un préfixe qui permet d'associer le nœud à un espace de noms, et un nom. Nous définissons les fonctions suivantes pour accéder aux parties d'un identificateur N :

- *prefixe* (N)
- *nom* (N)

Deux identificateurs sont égaux s'ils ont le même préfixe et le même nom.

8.2 Partitionnement d'un arbre Xml autour d'un nœud

XPath effectue, autour d'un nœud donné, une partition d'un arbre Xml en plusieurs ensembles dont l'intersection n'est pas vide. Soit n un nœud d'arbre Xml, la partition de l'arbre autour de n est la suivante :

- n.`child` : contient les nœuds fils de n (à l'exception des nœuds de type *attribut* ou *espace de noms*[53]),
- n.`descendant` : fermeture transitive de n.`child`,
- n.`parent` : contient le nœud parent de n, s'il existe, est égal à $\emptyset$ sinon,
- n.`ancestor` : fermeture transitive de n.`parent`,
- n.`following-sibling` : contient les nœuds de l'arbre qui ont le même nœud parent que n et qui sont situés après n (selon $<_{\texttt{doc}}$) ; les nœuds de type *attribut* et *espace de noms* n'ont pas de nœuds frères et ne sont les nœuds frères d'aucun nœud,

[53]Le document normatif de XPath 1.0 ne donnant pas cette précision, nous nous appuyons sur la définition de l'axe *child* donnée dans le document normatif de XPath 2.0 : `http://www.w3.org/TR/xpath20`

- n.`preceding-sibling` : contient les nœuds de l'arbre qui ont le même nœud parent que n et qui sont situés avant n (selon $<_{\texttt{doc}}$),
- n.`following` : contient les nœuds de l'arbre qui sont situés après n (selon $<_{\texttt{doc}}$), à l'exception des descendants du nœud courant et des nœuds de type *attribut* ou *espace de noms*.
- n.`preceding` : contient les nœuds de l'arbre qui sont situés avant n (selon $<_{\texttt{doc}}$), à l'exception des ancêtres de n et des nœuds de type *attribut* ou *espace de noms*,
- n.`attribute` : contient les sous-nœuds de type *attribut* de n,
- n.`namespace` : contient les sous-nœuds de type *espace de noms* de n,
- n.`self` : contient n,
- n.`descendant-or-self` : contient n et ses descendants,
- n.`ancestor-or-self` : contient n et ses ancêtres.

Dans le vocabulaire de XPath, ces ensembles de nœuds sont appelés axes [XG99, Sec. 2.2] (*axis* en anglais). La définition formelle des axes (dont nous notons l'identificateur a) est donnée par la table 8.1.

a	$n.a$
`child`	$\{x \in \Sigma \mid n \rightarrow x \ \wedge \ \textit{sorte}\ (x) \neq \textit{attribut} \ \wedge \ \textit{sorte}\ (x) \neq \textit{espace de noms}\}$
`descendant`	$\{x \in \Sigma \mid \exists\, x_1 ... x_n \in \Sigma \ \wedge \ n \rightarrow x_1 \rightarrow ... \rightarrow x_n \rightarrow x$ $\wedge \ \textit{sorte}\ (x) \neq \textit{attribut} \ \wedge \ \textit{sorte}\ (x) \neq \textit{espace de noms}\}$
`parent`	$\{x \in \Sigma \mid x \rightarrow n\}$
`ancestor`	$\{x \in \Sigma \mid \exists\, x_1 ... x_n \in \Sigma \ \wedge \ x \rightarrow x_1 \rightarrow ... \rightarrow x_n \rightarrow n$ $\wedge \ \textit{sorte}\ (x) \neq \textit{attribut} \ \wedge \ \textit{sorte}\ (x) \neq \textit{espace de noms}\}$
`following`	$\{x \in \Sigma \mid n <_{\texttt{doc}} x \ \wedge \ x \neq n \ \wedge \ x \notin n.\texttt{descendant}$ $\wedge \ \textit{sorte}\ (x) \neq \textit{attribut} \ \wedge \ \textit{sorte}\ (x) \neq \textit{espace de noms}\}$
`preceding`	$\{x \in \Sigma \mid x <_{\texttt{doc}} n \ \wedge \ x \neq n \ \wedge \ x \notin n.\texttt{ancestor}$ $\wedge \ \textit{sorte}\ (x) \neq \textit{attribut} \ \wedge \ \textit{sorte}\ (x) \neq \textit{espace de noms}\}$
`following-sibling`	`si` $\textit{sorte}\ (n) = \textit{attribut} \ \vee \ \textit{sorte}\ (n) = \textit{espace de noms}$ `alors` $\emptyset$ `sinon` $\{x \in \Sigma \mid x.\texttt{parent} = n.\texttt{parent} \ \wedge \ x \in n.\texttt{following}\}$
`preceding-sibling`	`si` $\textit{sorte}\ (n) = \textit{attribut} \ \vee \ \textit{sorte}\ (n) = \textit{espace de noms}$ `alors` $\emptyset$ `sinon` $\{x \in \Sigma \mid x.\texttt{parent} = n.\texttt{parent} \ \wedge \ x \in n.\texttt{preceding}\}$
`attribute`	$\{x \in \Sigma \mid n \rightarrow x \ \wedge \ \textit{sorte}\ (x) = \textit{attribut}\}$
`namespace`	$\{x \in \Sigma \mid n \rightarrow x \ \wedge \ \textit{sorte}\ (x) = \textit{espace de noms}\}$
`self`	$\{n\}$
`descendant-or-self`	$n.\texttt{self} \cup n.\texttt{descendant}$
`ancestor-or-self`	$n.\texttt{self} \cup n.\texttt{ancestor}$

Table 8.1: Définition formelle des axes

Pour illustrer ce partitionnement, nous donnons ci-dessous les valeurs des axes pour le nœud E_5 de l'arbre Xml de la section 8.1 :

Axe	Ensemble de nœuds
`child` (E_5)	$\{E_6, C_2, E_7\}$
`descendant` (E_5)	$\{E_6, V_c, C_2, E_7, V_d\}$
`parent` (E_5)	$\{E_1\}$
`ancestor` (E_5)	$\{E_1\}$
`following-sibling` (E_5)	$\emptyset$
`preceding-sibling` (E_5)	$\{E_2, C_1\}$
`following` (E_5)	$\emptyset$
`preceding` (E_5)	$\{C_1, V_b, E_4, I, V_a, E_3, E_2\}$
`attribute` (E_5)	$\{A_2 = V_2\}$
`namespace` (E_5)	$\emptyset$
`self` (E_5)	$\{E_5\}$
`descendant-or-self` (E_5)	$\{E_5, E_6, V_c, C_2, E_7, V_d\}$
`ancestor-or-self` (E_5)	$\{E_5, E_1\}$

XPath introduit deux notions relatives aux axes [XG99, Sec. 2.3 et 2.4] qui sont nécessaires à la définition de la sémantique du langage à la section 8.5.2 :

- A chaque axe a est associée une sorte de nœud principale que nous notons *principal* (a). Pour l'axe `attribut`, la sorte de nœud principale est *attribut*, pour l'axe `namespace`, la sorte de nœud principale est *espace de noms*, tandis que pour tous les autres axes, la sorte de nœud principale est *élément*.

- A chaque axe a est associé un ordre que nous notons *ordre* (a). L'ordre des axes `child`, `descendant`, `parent`, `following`, `following-sbling`, `attribute`, `namespace`, `self` et `descendant-or-self` est *normal*, c'est-à-dire que dans les ensembles de nœuds sélectionnés par ces axes, les nœuds sont rangés selon l'ordre induit par la relation d'ordre total strict $<_{\texttt{doc}}$. L'ordre des axes `ancestor`, `preceding`, `preceding-sibling` et `ancestor-or-self` est *inversé*, c'est-à-dire que dans les ensembles de nœuds sélectionnés par ces axes, les nœuds sont rangés selon l'ordre induit par la relation d'ordre total strict inverse de $<_{\texttt{doc}}$.

8.3 Syntaxe normalisée

Nous présentons dans cette section une syntaxe abstraite normative pour le langage XPath, c'est-à-dire une syntaxe minimale, sans sucre syntaxique. Les conventions définies au chapitre 7 pour les notations restent valables et sont enrichies par les notations qui suivent :

- a est un identificateur d'axe (cf. section 8.2),
- X est un identificateur de variable,
- P est un identificateur de partie de message Wsdl (cf. section 9.1.1),
- I est un identificateur d'instruction de traitement,
- N_s est un identificateur d'espace de nom,
- F désigne une fonction prédéfinie ou un opérateur parmi : l'opérateur unaire **-**, les opérateurs binaires **-**, +, *****, **div**, **mod**, **and**, **or**, >, <, >=, <=, =, **!=**, | (union de deux ensembles de nœuds) et les fonctions **id**, **local-name**, **namespace-uri**, **name**, **count**, **string-length**, **number**, **sum**, **floor**, **ceiling**, **round**, **starts-with**, **contains**, **boolean**, **not**, **true**, **false**,

lang, **string**, **concat**, **substring-before**, **substring-after**, **substring**, **normalize-space**, **translate**.

- N est un identificateur d'élément, un identificateur d'attribut, un identificateur d'instruction de traitement ou bien un identificateur d'espace de noms :

$$N ::= E \mid A \mid I \mid N_s$$

Dans la grammaire XPath que nous présentons ci-dessous, certaines constructions sont marquées d'un astérisque (*). Cela signifie que ces constructions sont des abbréviations et qu'elles peuvent être réécrites grâce à d'autres constructions. Le point d'entrée de la grammaire est *Expr*.

Var	::=	X	accès à une variable
	|	X `.` P	accès à une partie de message Wsdl
Test	::=	N	validation des nœuds d'identificateur N
	|	N^s`:*`	validation des nœuds dont le nom appartient à N^s
	|	`*`	validation de tous les nœuds
	|	`text()`	validation des nœuds *texte*
	|	`comment()`	validation des nœuds *commentaire*
	|	`processing-instruction()`	validation des nœuds *instruction de traitement*
	|	`processing-instruction(`I`)`	validation de l'instruction de traitement I
	|	`node()`	validation de nœuds selon le contexte
	|	*Test* `[`*Expr*`]`	validation des nœuds pour lesquels *Expr* vaut vrai
	|	*Test* `[`V`]`$*$	
Step	::=	a`::`*Test*	pas de chemin d'accès
	|	N^*	
	|	`.`*	
	|	`..`*	
	|	`@`A^*	
Steps	::=	$Step_0$ `/` ... `/` $Step_n$	chemin d'accès
Expr	::=	V	constante booléenne, numérique ou chaîne de caractères
	|	*Var*	variable
	|	$F(Expr_1, ..., Expr_n)$	appel de fonction
	|	`position()`	fonction spéciale
	|	`last()`	fonction spéciale
	|	`/` *Steps*	chemin d'accès depuis le nœud racine
	|	*Steps*	chemin d'accès depuis le nœud courant du contexte
	|	`//`*	
	|	*Expr* `/` *Steps*	chemin d'accès depuis *Expr*

L'utilisateur peut référencer des variables dans une expression XPath, mais il ne lui est pas possible d'en déclarer de nouvelles. En effet, l'ensemble des variables est statique et est passé à l'interpréteur lors de l'invocation. Dans le cas de l'utilisation d'expressions XPath en Bpel, l'interpréteur XPath reçoit la liste des variables déclarées dans le service Bpel. Le type d'une variable Bpel peut être soit un type Xml Schema, soit un élément racine Xml Schema, soit un message Wsdl. Les messages Wsdl, que nous présentons à la section 9.1.1, sont composés de parties $P_0..P_n$. Le type de chaque partie est soit un type Xml Schema, soit un élément racine Xml Schema. Pour accéder aux parties d'une variable dont le type est un message Wsdl, Bpel prévoit une extension au langage XPath [Com01, Sec. 8.2.2]. Il s'agit de la construction X`.`P qui permet, si X est une variable dont le type est un message Wsdl, d'accéder à la partie P du message contenu dans X.

La norme XPath définit un ensemble de fonctions prédéfinies que chaque interpréteur XPath se doit d'implémenter. Cet ensemble n'est pas extensible : aucune construction ne permet à l'utilisateur de

créer ses propres fonctions. `position()` et `last()` sont des fonctions spéciales dont la valeur dépend uniquement du contexte d'évaluation. Afin de simplifier l'écriture de la sémantique dynamique de XPath, nous avons choisi de les sortir de la règle $F(Expr_1, ..., Expr_n)$.

Les constructions appartenant au sucre syntaxique de XPath peuvent être réécrites ainsi :

- *Test* `[`V`]` devient *Test* `[position() =` V`]`, si V est une constante numérique
- N devient `child::`N
- . devient `self::node()`
- .. devient `parent::node()`
- @ a devient `attribute::` a
- `//` devient `/descendant-or-self::node()/`

8.4 Sémantique statique normalisée

Dans cette section, nous introduisons la sémantique statique normalisée de XPath, c'est-à-dire l'algorithme de typage de XPath, tel que présenté dans la norme (d'où le terme "normalisée"). En réalité, la norme XPath, ne préconise pas d'effectuer une analyse statique des expressions XPath avant de les interpréter car les types des variables ne peuvent être statiquement connu. Les vérifications de types sont faites durant l'exécution, à la volée. Nous avons tout de même choisi d'extraire un algorithme de typage de la norme XPath, en considérant connus les types des variables. Notre but est d'adapter, à la section 8.6.3, cet algorithme à Lotos NT, un langage dans lequel la vérification des types est effectuée statiquement.

Le système de types de XPath ignore les déclarations de types de Xml Schema et repose sur quatre types prédéfinis : node-set, boolean, number et string. Ces quatre types peuvent se convertir les uns vers les autres, avec la restriction qu'aucune conversion n'est possible vers node-set.

Des conversions implicites ont lieu lors des appels de fonctions, si les valeurs données en argument à une fonction n'ont pas les types attendus. Ces conversions sont effectuées lors de l'évaluation. Si une conversion n'est pas permise, alors une erreur est levée. En outre, certaines fonctions ont des prototypes variables, par exemple **concat** prend en arguments, un nombre arbitraire de chaînes de caractères, ou bien, certaines fonctions comme **substring** ont un argument optionnel.

La prise en compte de ces deux aspects (conversions et prototypes variables) complexifie la présentation de la sémantique statique normalisée de XPath. Nous avons décidé de les ignorer afin de présenter, dans cette section, un algorithme de typage simple. Nous revenons plus longuement sur ces aspects à la section 8.6.3 car ils sont importants dans le cadre de la traduction en Lotos NT. Nous considérons donc que la vérification du nombre d'arguments d'une fonction ainsi que la vérification du type de ces arguments est effectué dynamiquement, par la fonction, lors de son exécution.

Nous introduisons les quatre notations suivantes :

- *V.type* qui donne le type XPath de la valeur constante V,
- *P.type* qui donne le type XPath d'une partie P de message Wsdl,
- *X.type* qui donne le type XPath de la variable X ; dans le cas général, ce type n'est pas statiquement connu et peut varier entre deux évaluations de la même expression, dans notre

cas, au contraire, nous pouvons le déterminer statiquement car les expressions XPath sont contenues dans des services Bpel dont les variables sont statiquement typées,

- *F.type* qui donne le type XPath de retour de la fonction F, en rappelant que les fonctions de XPath sont prédéfinies et que par conséquent leur type de retour est connu.

Ce typage est défini à l'aide de la grammaire attribuée suivante. Elle synthétise un attribut $\mathcal{T}$ qui représente le type d'une expression :

$Expr \uparrow \mathcal{T}$	::=	V	$\mathcal{T} := V.type$
	|	$Var \uparrow \mathcal{T}'$	$\mathcal{T} := \mathcal{T}'$
	|	$F(Expr_1, ..., Expr_n)$	$\mathcal{T} := f.type$
	|	`position()`	$\mathcal{T} :=$ `number`
	|	`last()`	$\mathcal{T} :=$ `number`
	|	`/` *Steps*	$\mathcal{T} :=$ `node-set`
	|	*Steps*	$\mathcal{T} :=$ `node-set`
	|	$Expr \uparrow \mathcal{T}'$/*Steps*	`si` $\mathcal{T}' =$ `node-set` `alors` $\mathcal{T} :=$ `node-set` `sinon` *erreur*
$Var \uparrow \mathcal{T}$	::=	X	$\mathcal{T} := X.type$
	|	$X.P$	$\mathcal{T} := T \mid X.type =$ **message** $(P_0 : T_0 ... P_n : T_n) \wedge \exists\ i \in [0...n], P_i = P\ \wedge\ T = T_i$

où **message** $(P_0 : T_0 ... P_n : T_n)$ désigne un message Wsdl formé des parties $P_0 ... P_n$ dont les types respectifs sont $T_0 ... T_n$.

Nous définissons la notation *Expr.type* pour accéder au type $\mathcal{T}$ d'une expression après qu'il a été calculé par la grammaire attribuée ci-dessus.

8.5 Sémantique dynamique normalisée

Dans cette section, nous décrivons la sémantique dynamique de XPath, c'est-à-dire le calcul du résultat de l'évaluation d'une expression XPath. Nous présentons une sémantique dynamique fidèle à la norme, au moyen d'une sémantique dénotationnelle qui a le mérite d'être plus succincte et précise que la sémantique de la norme qui consiste en de longues explications textuelles.

Nous commençons par présenter le contexte d'évaluation d'une expression XPath, c'est-à-dire les paramètres qui sont passés à l'évaluateur et qui peuvent influer sur le résultat de l'évaluation. Ensuite, nous détaillons la sémantique de XPath, construction par construction.

8.5.1 Contexte d'évaluation

Le contexte d'évaluation d'une expression XPath, tel que défini dans la norme, est un 6-uplet (c, p, s, $\mathcal{X}$, $\mathcal{F}$, $\mathcal{N}$) où :

- c est le nœud courant de l'arbre Xml sur lequel l'expression XPath est évaluée,
- p est la position du nœud courant parmi l'ensemble de nœuds dans lequel il a été choisi,
- s est la taille de l'ensemble de nœuds dans lequel le nœud courant a été choisi,
- $\mathcal{X}$ est un environnement de variable qui associe à chaque variable une valeur,
- $\mathcal{F}$ est l'environnement des fonctions qui sont implémentées par l'interpréteur,

- $\mathcal{N}$ est un environnement d'espaces de noms qui associe un préfixe à chaque espace de nom déclaré dans le document XML.

L'arbre XML représentant le document (ou terme) XML sur lequel l'expression est évaluée est inclus indirectement dans le contexte. En effet, le nœud racine de cet arbre est accessible depuis le nœud courant. Pour ce faire, il suffit de sélectionner, parmi les ancêtres du nœud courant, celui qui ne possède pas d'ancêtre.

c, p et s peuvent changer de valeurs au cours de l'évaluation d'une expression. p et s ne sont utilisées que dans les prédicats, pour définir les valeurs retournées par les appels aux fonctions `position()` et `last()`. $\mathcal{X}$, $\mathcal{F}$ et $\mathcal{N}$, au contraire, sont constants tout au long de l'évaluation.

8.5.2 Sémantique dénotationnelle

La sémantique dynamique des expressions XPATH est donnée, dans cette section, construction par construction, au moyen d'une sémantique dénotationnelle. Les valeurs propagées sont :

- Σ : l'ensemble de nœuds courant,
- p : un entier naturel qui représente la position du nœud courant dans Σ,
- a : pour définir la sémantique des pas de chemin d'accès ($Step :: a :: Test$), il est nécessaire de propager l'axe a qui a servi à sélectionner l'ensemble de nœuds sur lequel *Test* est évalué. En effet, comme nous le verrons, la sorte de nœud principale et l'ordre de a ont un effet sur l'évaluation de *Test*.

Nous préférons propager l'ensemble des nœuds courant (Σ) et la position (p) du nœud courant dans cet ensemble et non pas un unique nœud courant (c), ainsi que sa position (p) dans l'ensemble de nœuds courant et le nombre (s) de nœuds dans l'ensemble de nœuds courant, comme préconisé dans la norme XPATH. Nous trouvons que cela rend l'écriture de la sémantique dynamique plus intuitive car l'évaluation d'une expression XPATH consiste, en général, à sélectionner, dans l'arbre XML, un ensemble de nœuds à partir duquel un nouvel ensemble de nœuds est calculé et ainsi de suite. Nous pensons donc qu'il est important de faire figurer cet ensemble de nœuds dans la sémantique. De plus, cela nous permet de réduire le nombre de valeurs propagées car nous pouvons dériver de Σ et p le nœud courant et le nombre de nœuds dans Σ.

Les environnements $\mathcal{X}$, $\mathcal{F}$ et $\mathcal{N}$ n'apparaissent pas dans les valeurs propagées car ils restent constants pendant l'évaluation de l'expression. $\mathcal{N}$ est utilisé par certaines fonctions, telles que **namespace-uri**, ainsi que pour vérifier que les préfixes apparaissant dans les tests de l'expression XPATH sont valides. Il n'apparaît donc pas explicitement dans la sémantique que nous donnons.

Afin de faciliter la lecture, nous utilisons les notations suivantes :

- $\mathcal{X}[X]$: accès à la valeur de la variable X contenue dans l'environnement $\mathcal{X}$.
- $\mathcal{X}[X].P$: si X est une variable contenue dans l'environnement $\mathcal{X}$ et que son type est un message WSDL (composé donc des parties $P_0 ... P_n$), alors cette écriture permet d'accéder à la partie P du message WSDL contenu dans X.
- $\mathcal{F}[F]$ $(Expr_1, ..., Expr_n)$: appel de la fonction F contenue dans l'environnement de fonctions $\mathcal{F}$ avec les arguments $Expr_1, ..., Expr_n$. Nous ne détaillons pas les conversions ni la vérification du nombre d'argument car nous reviendrons longuement sur ces points dans la section 8.6.3.
- n_0 désigne le nœud racine de l'arbre XML.
- *taille* (Σ) renvoie la taille de l'ensemble de nœuds Σ.

Les valeurs initiales de Σ et p sont indéfinies (notées $\perp$). a, lorsqu'elle est présente, est forcément initialisée.

Afin de présenter la sémantique de XPath en suivant une approche *top-down*, nous détaillons la sémantique des constructions dans un ordre différent de celui de la grammaire de la section 8.3 : nous commençons par le non-terminal *Expr*, le point d'entrée de la grammaire.

Expr :

$$
\begin{array}{lcl}
[\![V]\!]\ (\Sigma, p) & = & V \\
[\![\mathit{Var}]\!]\ (\Sigma, p) & = & [\![\mathit{Var}]\!] \\
[\![F(\mathit{Expr}_1, ..., \mathit{Expr}_n)]\!]\ (\Sigma, p) & = & \mathcal{F}[F]([\![\mathit{Expr}_1]\!]\ (\Sigma, p), ..., [\![\mathit{Expr}_n]\!]\ (\Sigma, p)) \\
[\![\texttt{position()}]\!]\ (\Sigma, p) & = & \texttt{si}\ p = \perp\ \texttt{alors}\ \mathit{erreur}\ \texttt{sinon}\ p \\
[\![\texttt{last()}]\!]\ (\Sigma, p) & = & \texttt{si}\ \Sigma = \perp\ \texttt{alors}\ \mathit{erreur}\ \texttt{sinon}\ \mathit{taille}\ (\Sigma) \\
[\![\texttt{/}\ \mathit{Steps}]\!]\ (\Sigma, p) & = & [\![\mathit{Steps}]\!]\ (\{n_0\}) \\
[\![\mathit{Steps}]\!]\ (\Sigma, p) & = & [\![\mathit{Steps}]\!]\ (\{x \in \Sigma \mid \mathit{taille}\ (\{n \in \Sigma \mid n <_{\texttt{doc}} x\}) = p - 1\}) \\
[\![\mathit{Expr}\ \texttt{/}\ \mathit{Steps}]\!]\ (\Sigma, p) & = & [\![\mathit{Steps}]\!]\ ([\![\mathit{Expr}]\!]\ (\Sigma, p))
\end{array}
$$

Les valeurs Σ et p servent uniquement à l'évaluation des fonctions spéciales `position` et `last`. Il est important de remarquer que `position` (*resp.* `last`) ne peut être appelée tant que p (*resp.* Σ) n'a pas été initialisée, sans quoi une erreur se produit. Comme Σ et p ont des valeurs indéfinies au début de l'évaluation d'une expression, les fonctions `position()` et `last()` des expressions XPath valides ; elles ne peuvent être utilisées qu'à l'intérieur d'un test. La construction `/` *Steps* évalue le chemin d'accès, dénoté par *Steps*, sur le nœud racine. La construction *Steps* évalue le chemin d'accès, dénoté par *Steps* sur le nœud courant. La construction *Expr* `/` *Steps* évalue le chemin d'accès, dénoté par *Steps*, sur l'ensemble de nœuds renvoyé par l'évaluation de l'expression *Expr*.

Les accès aux variables ne dépendent d'aucune valeur propagée :

Var :

$$
\begin{array}{lcl}
[\![X]\!] & = & \mathcal{X}[X] \\
[\![X.P]\!] & = & \mathcal{X}[X].P
\end{array}
$$

L'évaluation d'un chemin d'accès Step_0 `/` ... `/` Step_n se fait de façon séquentielle. Chaque pas du chemin, en commençant par Step_0, est évalué sur l'ensemble de nœuds courant. Le nouvel ensemble de nœuds ainsi créé devient l'ensemble de nœuds courant pour l'évaluation du pas suivant. $\mathit{Step}_1 ... \mathit{Step}_n$.

Steps :

$$[\![\mathit{Step}_0\ \texttt{/}\ ...\ \texttt{/}\ \mathit{Step}_n]\!]\ (\Sigma) \quad = \quad [\![\mathit{Step}_1\ \texttt{/}\ ...\ \texttt{/}\ \mathit{Step}_n]\!]\ ([\![\mathit{Step}_0]\!]\ (\Sigma))$$

Un pas de chemin d'accès *Step* consiste en un axe et un test. L'évaluation d'un axe a sur l'ensemble de nœuds courant crée un nouvel ensemble de nœuds courant qui consiste en l'union des évaluations, pour chaque nœud n de Σ, de $n.a$. Le nouvel ensemble de nœuds ainsi créé est ensuite soumis à un test (*Test*) : tous les nœuds de l'ensemble ne satisfaisant pas le test sont supprimés.

Step :

$$[\![a :: \mathit{Test}]\!]\ (\Sigma) \quad = \quad [\![\mathit{Test}]\!]\ (\bigcup_{n}^{\Sigma}\ n.a, a)$$

La sémantique des tests est présentée ci-dessous. Pour les tests N, N_s`:*` et `*`, les nœuds sélectionnés appartiennent forcément à la sorte de nœuds principale du dernier axe rencontré. L'évaluation du test *Test* `[`*Expr*`]` commence par l'évaluation de *Test* sur l'ensemble de nœuds courant. Pour chaque nœud n du nouvel ensemble de nœuds ainsi créé, le résultat de l'évaluation de *Expr* avec n comme

nœud courant (il faut impérativement tenir compte de l'ordre du dernier axe rencontré pour le calcul de l'indice p qui donne la position de n dans l'ensemble de nœuds), est converti en une valeur booléenne : si ce résultat est vrai, alors n est ajouté à l'ensemble de nœuds résultant de l'évaluation de *Test* `[`*Expr*`]`.

Test :

$$
\begin{array}{lcl}
[\![N]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma \ \wedge \ \mathit{nom}\ (\mathit{id}\ (n)) = \mathit{nom}\ (N) \\
 & & \quad \wedge \ \mathit{prefixe}\ (\mathit{id}\ (n)) = \mathit{prefixe}\ (N)\} \\
 & & \quad \wedge \ \mathit{sorte}\ (n) = \mathit{principal}\ (a)\} \\
[\![N_s\texttt{:*}]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma \ \wedge \ \mathit{prefixe}\ (\mathit{id}\ (n)) = N_s \\
 & & \quad \wedge \ \mathit{sorte}\ (n) = \mathit{principal}\ (a)\} \\
[\![\texttt{*}]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma \ \wedge \ \mathit{sorte}\ (n) = \mathit{principal}\ (a)\} \\
[\![\texttt{text()}]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma \ \wedge \ \mathit{sorte}\ (n) = \mathit{texte}\} \\
[\![\texttt{comment()}]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma \ \wedge \ \mathit{sorte}\ (n) = \mathit{commentaire}\} \\
[\![\texttt{processing-instruction()}]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma \\
 & & \quad \wedge \ \mathit{sorte}\ (n) = \mathit{instruction\ de\ traitement}\} \\
[\![\texttt{processing-instruction(}I\texttt{)}]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma \\
 & & \quad \wedge \ \mathit{sorte}\ (n) = \mathit{instruction\ de\ traitement} \\
 & & \quad \wedge \ \mathit{nom}\ (\mathit{id}\ (n)) = I\} \\
[\![\texttt{node()}]\!]\ (\Sigma, a) & = & \Sigma \\
[\![\mathit{Test}\ \texttt{[}\mathit{Expr}\texttt{]}]\!]\ (\Sigma, a) & = & \{n \mid n \in \Sigma' \ \wedge \ \mathcal{F}[\mathbf{boolean}]\ ([\![\mathit{Expr}]\!]\ (\Sigma', k))\} \\
 & & \texttt{où} \\
 & & \quad \Sigma' = [\![\mathit{Test}]\!]\ (\Sigma, a) \\
 & & \quad k = \texttt{si}\ \mathit{ordre}\ (a) = \text{“inversé”} \\
 & & \qquad \texttt{alors}\ \mathit{taille}\ (\Sigma') + 1 - j \\
 & & \qquad \texttt{sinon}\ j \\
 & & \quad j = \mathit{taille}\ (\{n' \mid n' \in \Sigma' \ \wedge \ n' <_{\texttt{doc}} n\})
\end{array}
$$

Comme nous pensons que les pas de chemin d'accès sont les constructions les plus complexes du langage XPath, nous présentons ci-dessous quelques exemples d'utilisation (Σ dénote l'ensemble de nœuds d'entrée du pas) :

- `child::*` sélectionne tous les nœuds de sorte *élément* dont le parent fait partie de Σ,
- `attribute::*` sélectionne tous les nœuds de sorte *attribut* dont le parent fait partie de Σ,
- `child::*[attribute::titre = "Introduction"]` sélectionne tous les nœuds de sorte *élément* dont le parent fait partie de Σ et qui possèdent un attribut `titre` ayant pour valeur `"Introduction"`,
- `child::*[attribute::titre = "Introduction"][position() = 5]` sélectionne le cinquième nœud (selon l'ordre des nœuds dans l'axe `attribute`) parmi ceux qui sont de sorte *élément*, ont leur nœud parent dans Σ et possèdent un attribut `titre` ayant pour valeur `"Introduction"`,
- `child::*[position() = 5][attribute::titre = "Introduction"]` commence par extraire la liste des nœuds fils des nœuds contenus dans Σ avant de ne garder que le cinquième élément de cette liste, s'il possède un attribut `titre` ayant pour valeur `"Introduction"`,
- `attribute::*[position() = last()]` sélectionne le dernier attribut (selon l'ordre de l'axe `attribute`) parmi les attributs des nœuds de Σ.

8.6 Traduction en LOTOS NT

8.6.1 Etat de l'art

XPath est fréquemment associée, dans la littérature, aux logiques temporelles. Ces travaux ne sont pas liés à la vérification formelle. Ils utilisent les formalismes de logiques temporelles pour représenter des expressions XPath afin de raisonner sur ces expressions. Par exemple, Layaïda et Geneves[NP06], ainsi que Jurdzinski et Lazic[JL07] encodent expressions XPath et définitions (Xml Schema ou Dtd) de documents en μ-calcul pour ensuite décider si :

- pour une expression XPath donnée et un document Xml, conforme à une définition Dtd ou Xml Schema, le résultat de l'évaluation de l'expression sur le document est toujours différent de l'ensemble vide.
- pour deux expressions XPath données et un document Xml, conforme à une définition Dtd ou Xml Schema, le résultat de l'évaluation d'une des deux expressions sur le document est toujours inclus dans le résultat de l'évaluation de l'autre expression sur le même document.

D'autres auteurs emploient des outils de vérification formelle pour évaluer des expressions XPath sur des documents Xml. Afanasiev et al.[AFMDR04], par exemple, traduisent les expressions XPath en formules de Ctl[CES86] (*Computational Tree Logic*) et les documents Xml dans le formalisme d'arbres binaires de décision[Bry86] (qui encode symboliquement un système de transitions) utilisé par le *model checker* NuSMV[CCGR00]. L'évaluation de l'expression XPath se fait par l'évaluation de la formule de Ctl par NuSMV sur l'arbre binaire de décision obtenu. Les auteurs ont modifé l'outil NuSMV afin de calculer tous les états (et non pas seulement le premier) du modèle pour lesquels la formule de Ctl est satisfaite.

A notre connaissance, dans le cadre de la vérification formelle de services Web, seuls Fu, Bultan et Su [FBS04b] décrivent explicitement la traduction des expressions XPath, qu'ils ont implémentée dans un interpréteur XPath écrit en Promela. Toutefois, la grammaire XPath considérée est inadaptée à la vérification de services Web Bpel :

- Les axes (à l'exception de `child`), les constantes, les variables et les fonctions (sauf `last` et `position`) sont absents de la grammaire, alors que la plupart des expressions XPath comportent au moins l'une de ces quatre constructions. Il est intéressant de constater que les deux fonctions XPath traitées n'apparaissent dans aucun des exemples de notre base. En revanche, des fonctions fréquemment utilisées comme `concat`, ou `length` ne sont pas traitées.
- Les suites de prédicats (règle *Test* ::= *Test* [*Expr*] de la grammaire dans laquelle *Expr* est le prédicat) sont traitées n'apparaissent dans notre base d'exemples qu'avec un seul prédicat.
- Le test `*` est traité alors qu'il n'apparait qu'une seule fois dans notre base d'exemples. Cette unique occurrence peut être réécrite plus simplement, sans le `*`.

Ces trois points confirment la pertinence de notre base d'exemples qui nous permet, à l'inverse de Fu, Bultan et Su[FBS04b], de nous concentrer sur les constructions importantes (notamment l'axe `attribute`, les constantes, les variables, les fonctions), dans le cadre de la vérification de services Web Bpel, tout en évitant de compliquer la traduction par la prise en compte de constructions inutiles comme les prédicats multiples ou le test `*`,

En outre, dans l'agorithme de traduction, le typage des expressions XPath n'est pas mentionné, tout comme le lien entre le système de type de Xml Schema et son utilisation dans le typage des expressions XPath : en particulier, rien n'est dit sur les conversions de types qui sont un aspect essentiel de XPath (*convert* apparaît plus de 50 fois dans les 32 pages de la norme XPath).

8.6.2 Sous-ensemble utile de XPath

Grâce à notre base d'exemples, nous avons pu identifier les constructions du langage XPath qui sont réellement utilisées dans les services Web. Dans ce contexte, les expressions XPath servent principalement à accéder aux champs des variables Bpel. De ce fait, de nombreuses constructions deviennent inutiles :

- Sur les 1450 expressions XPath de notre base d'exemples, seulement 5 (dont 4 que nous ne pouvons pas traiter car il nous manque les définitions Xml Schema des variable manipulées) renvoient une valeur de type `node-set` avec plus d'un élément. Par conséquent, nous choisissons d'ignorer les tests XPath "N_s`:*`", "`*`" et "`node()`" qui sélectionnent des ensembles de nœuds qui comptent plus d'un élément, ainsi que l'opérateur | qui fait l'union de deux ensembles de nœuds.

- Pour la même raison, les fonctions qui prennent des ensembles de nœuds en arguments (**id**, **name** et **local-name**) sont ignorées, à l'exception de **sum** et **count** que nous continuons d'utiliser dans le cas des éléments qui peuvent apparaître plus d'une fois.

- Nous ignorons aussi les fonctions **namespace-uri** et **lang** qui ne sont jamais utilisées dans nos exemples.

- En Bpel, les expressions XPath sont évaluées seulement sur des variables dont le contenu est exploré au moyen de pas successifs, et non pas sur un arbre Xml. L'expression `/` *Steps* qui permet d'accéder au nœud racine de l'arbre Xml n'est donc pas nécessaire et est ignorée.

- Comme l'atteste notre base d'exemples, l'exploration du contenu d'une variable se fait exclusivement au moyen des axes `child` et `attribute`. Tous les autres axes sont donc ignorés.

- Les tests `comment()`, `processing-instruction()` et `processing-instruction(`I`)` permettent de sélectionner des nœuds de sorte *commentaire* et *instruction de traitement*. Ces nœuds, qui ne peuvent être décrits en Xml Schema, ont peu d'intérêt dans le cadre de notre étude. Par conséquent, nous avons choisi d'ignorer ces tests. Notre choix est aussi justifié par le fait que ces tests n'apparaissent pas dans notre base d'exemples.

- Dans notre base d'exemples, les expressions qui peuvent apparaître dans la règle *Test* ::= *Test* [*Expr*] ont toujours pour type `number`. Ces expressions servent à accéder au *Expr*-ième élément d'une liste (lorsqu'un élément peut apparaître plusieurs fois). Par conséquent, dans cette règle, seules les expressions dont le type est `number` sont acceptées. En l'occurrence, la fonction `last()` dont le type est `boolean` est ignorée.

- La fonction `position()` n'est pas utilisée dans notre base d'exemples. Elle est donc ignorée.

En plus de ces simplifications, nous avons opéré divers changements dans la présentation de la syntaxe de XPath. Les règles *Step*, *Test* et *Predicates* ont été expansées dans la règle *Steps*. Les pas ne s'appliquant plus que sur des variables Bpel, les règles *Expr* et *Var* ont été modifiées en conséquence (nous n'avons pas expansé la règle *Var* car nous en avons besoin à la section 8.6.4).

Nous rappelons les conventions précédemment définies pour les identificateurs :

- V est une constante XPath,

- X est un identificateur de variable,

- P est un identificateur de partie de message Wsdl,

- A est un identificateur d'attribut et

- E est un identificateur d'élément.

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & V \mid \mathit{Var} \mid F(\mathit{Expr}_1, ..., \mathit{Expr}_n) \\
\mathit{Var} & ::= & X \mid X\texttt{.}P \mid X\ \texttt{/}\ \mathit{Steps} \mid X\texttt{.}P\ \texttt{/}\ \mathit{Steps} \\
\mathit{Steps} & ::= & \texttt{child::text()} \\
& \mid & \texttt{attribute::}\ A \\
& \mid & \texttt{child::}\ E\ \texttt{[}\mathit{Expr}\texttt{]?} \\
& \mid & \texttt{child::}\ E\ \texttt{[}\mathit{Expr}\texttt{]?}\ \texttt{/}\ \mathit{Steps}
\end{array}
$$

8.6.3 Sémantique statique simplifiée

Comme expliqué à la section 8.4, durant l'évaluation d'une expression XPath, des conversions peuvent se produire lorsque les types des arguments d'un appel de fonction ne correspondent pas aux types attendus. Afin de déterminer statiquement les conversions à effectuer, nous avons conçu un algorithme de calcul des types pour notre version simplifiée de XPath.

Auparavant, nous avions tenté une approche sans calcul de types, dans laquelle nous avions défini des fonctions de conversions surchargées (portant toutes le même nom) entre tous les types Xml Schema. Dès qu'une conversion pouvait avoir lieu (c'est-à-dire à chaque passage d'arguments à une fonction XPath), la fonction de conversion était insérée dans le code Lotos NT. Le compilateur Lotos NT était alors chargé d'identifier la "bonne" conversion en utilisant le mécanisme de résolution des surcharges de fonctions qui existe en Lotos NT.

Nous nous sommes rapidement rendu compte que cette approche simple soulevait des ambiguïtés qui empêchaient le compilateur Lotos NT de traiter les fichiers. Par exemple, soient deux variables Bpel X_1 et X_2, dont les types respectifs sont **decimal** et **integer** : l'expression XPath X_1+X_2 était alors traduite en Lotos NT par `convertir` (X_1) `+ convertir` (X_2). Or, les types Xml Schema **decimal** et **integer** sont traduits en Lotos NT par les types **Real** et **Int**, lesquels sont justement munis de l'opérateur $+$. Par conséquent, le compilateur Lotos NT ne savait pas s'il devait choisir la fonction de conversion de **Real** vers **Int** sur X_1 ou bien la fonction de conversion de **Int** vers **Real** sur X_2.

Ayant compris les limitations de cette approche, nous avons conçu un algorithme de calcul des types pour XPath qui prend en compte les déclarations de types Xml Schema incluses par le service Bpel, afin d'insérer convenablement les fonctions de conversion, partout où cela est nécessaire.

Cet algorithme de calcul des types suit une approche *top-down* en calculant le type de chaque expression lors de la descente et en déterminant les conversions à effectuer à la remontée. Une telle approche est envisageable car il est possible, en XPath, de connaître le type de chaque expression sans avoir à connaître le contexte dans lequel est plongée l'expression. Lors de la remontée, le type de chaque expression est comparé au type attendu : s'ils sont différents, mais compatibles, une conversion est insérée ; s'ils sont différents mais incompatibles, une erreur est signalée. Il s'agit d'une technique classique de résolution des opérateurs surchargés.

La principale différence entre cet algorithme de calcul des types et celui, très simple, présenté à la section 8.4 est que nous tenons compte des déclarations de types Xml Schema qui diffèrent des types prédéfinis de XPath. Les deux types prédéfinis Xml Schema : string et boolean, prennent naturellement la place des types XPath de même nom. Xml Schema compte trois types numériques : decimal, integer et nonNegativeInteger, qui remplacent l'unique type numérique de XPath (number). Il est donc nécessaire, pour les fonctions XPath définies sur le type number, de considérer trois variantes, une pour decimal, une pour integer et une pour nonNegativeInteger. Les expressions XPath ayant pour type soit un type prédéfini Xml Schema autre que les cinq types sus-mentionnés, soit un type simple Xml Schema, devront être converties vers leur type de base (qui figure forcément parmi

string, boolean, decimal, integer et nonNegativeInteger). En XPath, une valeur de type complexe est représentée par un nœud de l'arbre Xml. Ce nœud est contenu dans une valeur XPath de type node-set à un élément. Dans notre traduction, le type node-set de XPath disparaît car nous ignorons toutes les constructions qui peuvent engendrer des valeurs de ce type. Nous faisons donc apparaître les types complexes Xml Schema dans notre algorithme de typage.

Nous introduisons les notations suivantes :

- *V.type* : renvoie le type de la constante V qui est forcément **boolean**, **nonNegativeInteger**, **integer**, **decimal** ou **string**.
- *X.type* : renvoie le type de la variable X.
- *A.type* : renvoie le type de l'attribut A.
- *E.type* : renvoie le type de l'élément E.
- *Expr.conversion* : représente l'attribut de *Expr* qui désigne le type compatible vers lequel il va être nécessaire de convertir *Expr*. La valeur indéfinie, notée $\perp$ pour *Expr.conversion* signifie que *Expr* ne doit pas être convertie.

Nous avons découpé la description de l'algorithme de calcul des types en 11 parties :

1. Nous introduisons une fonction pour la manipulation de l'attribut de conversion.
2. Nous présentons la procédure de calcul des types pour les expressions, mais sans détailler le calcul des types des fonctions.
3. Nous détaillons le calcul des types des fonctions **true**, **false**, **not**, **or**, **and**, **floor**, **ceiling**, **round**, **string-length**, **starts-with**, **contains**, **substring-before**, **substring-after**, **translate**, **substring** et **concat**.
4. Nous détaillons le calcul des types des fonctions **boolean**, **number** et **string**.
5. Nous détaillons le calcul des types des fonctions **count** et **sum**.
6. Nous détaillons le calcul du type de l'opérateur unaire **-**.
7. Nous détaillons le calcul des types des opérateurs binaires $+$, **-**, ***, **div** et **mod**.
8. Nous détaillons le calcul des types des opérateurs binaires $<$, $>$, $<=$ et $>=$.
9. Nous détaillons le calcul des types des opérateurs binaires $=$ et **!=**.
10. Nous définissons la fonction qui renvoie le type d'un accès à une variable.
11. Nous définissons la fonction qui renvoie le type d'un chemin d'accès.

1) Afin de faciliter la manipulation de l'attribut de conversion dans la suite de cette section, nous définissons une procédure *décideConversion* qui choisi la conversion à effectuer pour transformer l'expression *Expr* en une valeur de type T :

```
procédure décideConversion (Expr, T)
  soit T' = Expr.type dans
    si T' = T alors Expr.conversion ←⊥
    sinon si ∃ base_conv_{T'→T} alors Expr.conversion ← T
    sinon erreur de typage
```

2) Le calcul du type d'une expression est réalisé par une procédure *typage* qui prend une expression *Expr* en entrée et initialise la valeur de l'attribut *Expr.type* :

```
procédure typage (Expr)
  si Expr est de la forme V alors
    Expr.type ← V.type
  sinon si Expr est de la forme Var alors
    Expr.type ← typage_X (Var)
  sinon si Expr est de la forme F (Expr_1, ..., Expr_n) alors
    ...
```

où $typage_X$ (*Var*) désigne un appel à la fonction $typage_X$ qui renvoie le type de *Var*. Le reste de la procédure *typage*, détaillant le calcul des types des fonctions est défini dans les points qui suivent.

3) Le calcul des types des fonctions F_f appartenant à la liste : **true**, **false**, **not**, **or**, **and**, **floor**, **ceiling**, **round**, **string-length**, **starts-with**, **contains**, **substring-before**, **substring-after**, **translate**, **substring** est usuel et consiste à calculer le type de chaque argument de la fonction, convertir le type ainsi déterminé pour chaque argument vers le type attendu avant d'affecter le type de retour de la fonction comme type de l'expression courante :

```
procédure typage (Expr)
  ...
    si F ∈ F_f alors
      soit T_1, ..., T_m → T = signature (F) dans
        si m ≠ n alors
          erreur de typage
        sinon
          typage (Expr_1)
          décideConversion (Expr_1, T_1)
          ...
          typage (Expr_n)
          décideConversion (Expr_n, T_m)
          Expr.type := T
  ...
```

où *signature* (*F*) dénote la signature de la fonction *F*. Les signatures des fonctions F_f sont listées dans la table 8.2.

Note 1 : Le troisième argument de la fonction **substring** est optionnel et, s'il n'est pas spécifié, prend pour valeur la longueur du premier argument. Nous réécrivons donc chaque occurrence de **substring** ($Expr_1$, $Expr_2$) par **substring** ($Expr_1$, $Expr_2$, **string-length** ($Expr_1$)).

Note 2 : **concat** prend un argument un nombre arbitraire de chaînes de caractères. Pour simplifier le typage et la traduction de cette fonction, nous forçons le nombre d'arguments à 2 (ce qui explique la signature présentée pour cette fonction dans la table 8.2) et réécrivons les occurrences de **concat** ($Expr_0, ..., Expr_n$) par **concat** ($Expr_0$, **concat** ($Expr_1$, **concat** (..., $Expr_n$)...)).

4) Les fonctions **boolean**, **number** et **string** sont des fonctions XPath qui convertissent explicitement des valeurs de type quelconque vers une valeur de type boolean, number et string. Le calcul des types de ces fonctions consiste à convertir l'argument vers le type de sortie attendu puis à affecter ce type de sortie attendu comme type de l'expression. Ce procédé nous permet de traduire ces trois fonctions en encodant la conversion explicite qu'elles représentent dans l'attribut $Expr_1.conversion$.

Fonctions	Signature
true et **false**	→ boolean
not	boolean → boolean
or et **and**	boolean, boolean → boolean
normalize-space	string → string
floor, **ceiling** et **round**	decimal → integer
string-length	string → nonNegativeInteger
starts-with et **contains**	string, string → boolean
substring-before et **substring-after**	string, string → string
translate	string, string, string → string
substring	string, nonNegativeInteger, nonNegativeInteger → string
concat	string, string → string

Table 8.2: Signature des fonctions XPath

```
...
  sinon si F ∈ {boolean, number, string} alors
    si n ≠ 1 alors
      erreur de typage
    sinon
      typage (Expr_1)
      si F = boolean alors
        décideConversion (Expr_1, boolean)
        Expr.type ← boolean
      si F = number alors
        décideConversion (Expr_1, decimal)
        Expr.type ← decimal
      si F = string alors
        décideConversion (Expr_1, string)
        Expr.type ← string
...
```

5) Les deux fonctions **count** et **sum** prennent un argument de type quelconque et renvoient une valeur de type nonNegativeInteger. Pour être exact, ces fonctions ne sont définies que sur les types LOTOS NT représentant des définitions d'éléments pour lesquelles la valeur $V^{maxOccurs}$ est strictement supérieure à 1. Un appel à l'une ces fonctions sur une valeur d'un type pour lequel elle n'est pas définie entraînera une erreur lors de la compilation des fichiers LOTOS NT. Le calcul des types de ces fonctions est donc le suivant :

```
...
  sinon si F ∈ {count, sum} alors
    si n ≠ 1 alors
      erreur de typage
    sinon
      typage (Expr_1)
      si Expr_1.type ≠ element (T^psc, V^fixed, V^default, V^minOccurs, V^maxOccurs) alors
        erreur de typage
      sinon
        si V^maxOccurs ≤ 1 alors
          erreur de typage
        sinon
          Expr.type ← nonNegativeInteger
...
```

6) L'opérateur unaire **-** a plusieurs signatures qui viennent de la séparation du type de base XPath number en trois types de base Xml Schema : decimal, integer et nonNegativeInteger. Il est donc nécessaire d'analyser le type des arguments et de choisir la signature qui entraîne le moins de conversions. Le calcul des types de cet opérateur est le suivant :

```
...
  sinon si F = - et n = 1 alors
    typage (Expr_1)
    si base (Expr_1.type) = nonNegativeInteger alors
      décideConversion (Expr_1, nonNegativeInteger)
      Expr.type ← integer
    sinon si base (Expr_1.type) = integer alors
      décideConversion (Expr_1, integer)
      Expr.type ← integer
    sinon
      décideConversion (Expr_1, decimal)
      Expr.type ← decimal
...
```

7) Les opérateurs binaires +, **-**, ***, **div** et **mod** sont traités d'une façon similaire à l'opérateur unaire **-**. Comme ces opérateurs ont deux arguments, il faut toutefois faire attention à ne pas faire de conversions qui feraient perdre des informations, en convertissant, par exemple, une valeur de type decimal en integer parce que l'un des arguments est de type integer. Pour cette raison, les tests sur les types des arguments dans le calcul des types de ces opérateurs binaires sont différents de ceux du calcul des types de l'opérateur unaire **-** :

```
...
  sinon si F ∈ {+, -, *, div, mod} alors
    si n ≠ 2 alors
      erreur de typage
    sinon
      typage (Expr1)
      typage (Expr2)
      si base (Expr1.type) = decimal ou base (Expr2.type) = decimal alors
        décideConversion (Expr1, decimal)
        décideConversion (Expr2, decimal)
        Expr.type ← decimal
      sinon si base (Expr1.type) = integer ou base (Expr2.type) = integer alors
        décideConversion (Expr1, integer)
        décideConversion (Expr2, integer)
        Expr.type ← integer
      sinon si base (Expr1.type) = nonNegativeInteger ou
                  base (Expr2.type) = nonNegativeInteger alors
        décideConversion (Expr1, nonNegativeInteger)
        décideConversion (Expr2, nonNegativeInteger)
        Expr.type ← nonNegativeInteger
      sinon
        décideConversion (Expr1, decimal)
        décideConversion (Expr2, decimal)
        Expr.type ← decimal
...
```

8) Les opérateurs de comparaison $<$, $>$, $<=$ et $>=$ ont un calcul des types particulier. En effet, la norme XPath, définit précisément les conversions à effectuer selon le type des arguments des opérateurs. De plus, il faut adapter cette définition à la séparation du type number de XPath en decimal, integer et nonNegativeInteger. Le calcul des types des opérateurs de comparaison est décrit ci-dessous :

```
...
  sinon si F ∈ {<, >, <=, >=} alors
    si n ≠ 2 alors
      erreur de typage
    sinon
      typage (Expr1)
      typage (Expr2)
      Expr.type ← boolean
      si base (Expr1.type) = boolean ou base (Expr2.type) = boolean alors
        décideConversion (Expr1, nonNegativeInteger)
        décideConversion (Expr2, nonNegativeInteger)
      sinon si base (Expr1.type) = decimal ou base (Expr2.type) = decimal alors
        décideConversion (Expr1, decimal)
        décideConversion (Expr2, decimal)
      sinon si base (Expr1.type) = integer ou base (Expr2.type) = integer alors
        décideConversion (Expr1, integer)
        décideConversion (Expr2, integer)
      sinon si base (Expr1.type) = nonNegativeInteger ou
            base (Expr2.type) = nonNegativeInteger alors
        décideConversion (Expr1, nonNegativeInteger)
        décideConversion (Expr2, nonNegativeInteger)
      sinon
        décideConversion (Expr1, decimal)
        décideConversion (Expr2, decimal)
...
```

9) Les opérateurs d'égalité = et **!=** ont un calcul des types similaire à celui des opérateurs de comparaison. Les conversions à expliciter sont décrites dans la norme XPath et nous devons aussi prendre en compte la séparation du type number. Le calcul des types des opérateurs d'égalité est décrit ci-dessous :

```
...
  sinon -- F appartient forcément à {=, !=}
    si n ≠ 2 alors
      erreur de typage
    sinon
      typage (Expr1)
      typage (Expr2)
      Expr.type ← boolean
      si base (Expr1.type) = boolean ou base (Expr2.type) = boolean alors
        décideConversion (Expr1, boolean)
        décideConversion (Expr2, boolean)
      sinon si base (Expr1.type) = decimal ou base (Expr2.type) = decimal alors
        décideConversion (Expr1, decimal)
        décideConversion (Expr2, decimal)
      sinon si base (Expr1.type) = integer ou base (Expr2.type) = integer alors
        décideConversion (Expr1, integer)
        décideConversion (Expr2, integer)
      sinon si base (Expr1.type) = nonNegativeInteger ou
            base (Expr2.type) = nonNegativeInteger alors
        décideConversion (Expr1, nonNegativeInteger)
        décideConversion (Expr2, nonNegativeInteger)
      sinon
        décideConversion (Expr1, string)
        décideConversion (Expr2, string)
```

10) Le calcul des types des accès aux variables (*Var*) est une fonction nommée $typage_X$ qui renvoie le type d'un accès à une variable :

```
typage_V (Var) =
  si Var est de la forme X alors
    renvoie X.type
  sinon si Var est de la forme X.P alors
    renvoie P_i.type | X.type = message  (P_0, ..., P_n) ∧ ∃ i ∈ [0...n], P_i = P
  sinon si Var est de la forme X / Steps alors
    renvoie typage_S (Steps, typage_X (X))
  sinon -- Var est forcément de la forme X . P / Steps
    renvoie typage_S (Steps, typage_X (X.P))
```

où $typage_S$ est la fonction qui calcule le type d'un chemin d'accès.

11) Le calcul des types des chemins d'accès est effectué au moyen d'une fonction récursive $typage_S$ qui prend en argument un chemin d'accès et le type de la valeur sur laquelle est "évaluée" ce chemin d'accès, et qui renvoie le type de la valeur sélectionnée par le chemin d'accès. Nous définissons cette fonction étape par étape, comme nous avons défini la procédure *typage*.

Pour la sélection de la valeur textuelle (par `child::text()`) contenue dans une valeur de type T, il faut vérifier que T est soit un type simple soit un type complexe à contenu simple, auquel cas le type renvoyé est string ; sinon, nous signalons une erreur :

$typage_S$ (*Steps*, T) =
  `si` *Steps* est de la forme `child::test() alors`
    `si` $T \neq T^s$ `et` $T \neq$ **extension** $(T^s, A_0, ...A_n)$ `alors`
      erreur de type
    `sinon`
      `renvoie` string
...

Pour la sélection de l'attribut A (par `attribute::`A) contenu dans une valeur de type T, il faut vérifier que T est un type complexe et que A figure parmi les attributs de T, auquel cas le type de A est renvoyé ; sinon, nous signalons une erreur :

...
  `sinon si` *Steps* est de la forme `attribute::`A `alors`
    `si` $T \neq$ **extension** $(T^{ps}, A_0, ...A_n)$ `et` $T \neq$ **sequence** $(E_1, ..., E_m, A_1, ...A_n)$
        `et` $T \neq$ **choice** $(E_1, ..., E_m, A_1, ...A_n)$ `et` $T \neq$ **all** $(E_1, ..., E_m, A_1, ...A_n)$ `alors`
      erreur de type
    `sinon`
      `si` $\nexists\ i \in [0..n] \mid A = A_i$ `alors`
        erreur de type
      `sinon`
        `renvoie` $A.type$
...

Pour la sélection de l'élément E (par `child::`E) contenu dans une valeur de type T, il faut vérifier que T est un type complexe à contenu complexe et que E figure parmi les éléments de T. auquel cas le type de E est renvoyé ; sinon, nous signalons une erreur :

...
  `sinon si` *Steps* est de la forme `child::`E `alors`
    `si` $T \neq$ **sequence** $(E_1, ..., E_m, A_1, ...A_n)$ `et` $T \neq$ **choice** $(E_1, ..., E_m, A_1, ...A_n)$
        `et` $T \neq$ **all** $(E_1, ..., E_m, A_1, ...A_n)$ `alors`
      erreur de type
    `sinon`
      `si` $\nexists\ i \in [1..n] \mid E = E_i$ `alors`
        erreur de type
      `sinon`
        `renvoie` $E_i.type$
...

Dans le cas où un prédicat (`[`*Expr*`]`) est présent, il faut d'abord s'assurer que le type de *Expr* est compatible avec nonNegativeInteger (pour accéder au *Expr*-ième élément). Ensuite, il faut s'assurer que l'élément E sélectionné peut apparaître plus d'une fois en vérifiant que dans la déclaration du type de E, $V^{maxOccurs}$ est strictement supérieure à 1 :

...
`sinon si` *Steps* est de la forme `child::`E `[`*Expr*`]`$alors$
typage (*Expr*)
décideConversion (*Expr.type*, nonNegativeInteger)
`soit` **element** $(T^{psc}, V^{fixed}, V^{default}, V^{minOccurs}, V^{maxOccurs}) = typage_S$ (`child::`E, T)
`dans`
`si` $V^{maxOccurs} \leq 1$ `alors`
erreur de type
`sinon`
`renvoie` T^{psc}
...

Pour les chemins d'accès récursifs (`child::`E `/` *Steps*$'$ et `child::`E `/` *Steps*$'$ `[`*Expr*`]`, il suffit d'appeler récursivement la fonction $typage_S$:

...
`sinon si` *Steps* est de la forme `child::`E `/` *Steps*$'$ `alors`
`renvoie` $typage_S$ (*Steps*$'$, $typage_S$ (`child::`E, T))
`sinon --` *Steps* est forcément de la forme `child::`E `[`*Expr*`] /` *Steps*$'$
`renvoie` $typage_S$ (*Steps*$'$, $typage_S$ (`child::`E `[`*Expr*`]`, T))

8.6.4 Sémantique opérationnelle simplifiée

Nous présentons dans cette section une sémantique dynamique des expressions XPath par traduction vers Lotos NT. Il convient de distinguer deux cas, selon que l'expression considérée figure en partie droite ou en partie gauche d'une affectation en Bpel. Dans la grammaire simplifiée de XPath que nous avons donnée, les expressions en partie droite sont dérivées du non-terminal *Expr* tandis que les expressions en partie gauche sont dérivées du non-terminal *Var*.

La traduction des expressions en partie droite est aisée, car les constructions XPath de la grammaire simplifiée ont des équivalents directs en Lotos NT. Nous notons $[\![Expr]\!]^D$, la traduction des expressions en partie droite. Il faut d'abord insérer les fonctions de conversion lorsque l'attribut *conversion* est présent :

$[\![Expr]\!]^D$ =
`si` *Expr.conversion* $= T$ `alors` $conv_{Expr.type \to T}$ $([\![Expr]\!]^{D'})$
`sinon` $[\![Expr]\!]^{D'}$

où $[\![Expr]\!]^{D'}$ désigne la traduction d'une expression en partie droite sans prise en compte de l'attribut *Expr.conversion*. $[\![Expr]\!]^{D'}$ définit le code Lotos NT qui traduit l'expression *Expr* :

$[\![Expr]\!]^{D'}$ =
  `si` *Expr* est de la forme V `alors`
    V
  `sinon si` *Expr* est de la forme F $(Expr_1, ..., Expr_n)$ `alors`
    `si` $F \in \{\textbf{boolean}, \textbf{number}, \textbf{string}\}$ `alors`
      $[\![Expr_1]\!]^D$
    `sinon`
      F $([\![Expr_1]\!]^D, ..., [\![Expr_n]\!]^D)$
  `sinon --` *Expr* est forcément de la forme *Var*
    $[\![Var]\!]^D$

Les constantes XPath sont traduites directement en constantes Lotos NT. Pour chaque fonction ou opérateur XPath qui ne possède pas d'équivalent parmi les fonctions prédéfinies où les opérateurs de Lotos NT, nous avons écrit une fonction ou un opérateur correspondant dans une bibliothèque Lotos NT. Pour les fonctions ou opérateurs XPath polymorphes, il existe un ensemble de fonctions ou opérateurs polymorphes équivalents en Lotos NT. Les fonctions **boolean**, **number** et **string** sont traduites par des fonctions de conversions insérées automatiquement en fonction de la valeur de l'attribut *Expr.conversion*. Les accès aux variables XPath sont traduits par $[\![Var]\!]^D$ qui prend en entrée un accès à une variable (*Var*) et renvoie le code Lotos NT correspondant :

$[\![Var]\!]^D$ =
  `si` *Var* est de la forme X `alors`
    X
  `sinon si` *Var* est de la forme X`.`P `alors`
    X`.`P
  `sinon si` *Var* est de la forme X `/` *Steps* `alors`
    $[\![Steps, X]\!]^D$
  `sinon --` *Var* est forcément de la forme X`.`P `/` *Steps*
    $[\![Steps, X.P]\!]^D$

La traduction de X et $X.P$ est directe car, dans le but de simplifier la présentation de la traduction, nous ne tenons pas compte du problème de la concordance des identificateurs et considérons que tous les identificateurs de variables et de parties de messages sont des identificateurs Lotos NT valides. La traduction des chemins d'accès ($[\![Steps, Var]\!]^D$) propage l'accès à une variable *Var* figurant à "gauche" de *Steps* dans l'expression XPath considérée. Par exemple pour l'expression X `/` *Steps*, X est propagée pour la traduction de *Steps*. Cette propagation est nécessaire pour deux raisons :

1. Afin de traduire le pas de chemin `child::text()`, il faut connaître le type de la valeur sélectionnée (c'est-à-dire le type de la variable ou du champ de variable *Var* que nous propageons) par le pas précédent. S'il s'agit d'une valeur de type complexe à contenu simple, alors il faut se conformer à la traduction Xml Schema que nous avons présentée à la section 7.4.17 et accéder au champ `f` de la valeur. S'il s'agit d'une valeur de type simple, alors il suffit de traduire l'accès de variable propagé. Il est possible de ne propager que le type de la valeur sélectionnée par le pas précédent, mais cela reviendrait à rédéfinir le calcul des types que nous avons présenté à la section 8.6.3.

2. Cette propagation est nécessaire à la traduction du pas `child::`E `[`*Expr*`]`. Ce pas permet de sélectionner le *Expr*-ième élément E d'une valeur de type complexe. Cette sélection, en Lotos NT, se fait au moyen de la fonction $getNth_T$ (cf. section 7.4.16), où T désigne le type de l'élément E. Le premier argument de cette fonction est la valeur sélectionnée par le pas précédent, c'est-à-dire la traduction en Lotos NT de l'accès de variable (*Var*) propagé.

La définition de la traduction des chemins d'accès figurant dans une expression en partie droite est donc la suivante :

$[\![Steps, Var]\!]^D$ =
  `si` *Steps* est de la forme `attribute::`A `alors`
    $[\![Var]\!]^D$`.`A
  `sinon si` *Steps* est de la forme `child::text() alors`
    `si` $typage_X$ (*Var*) = **extension** $(T, A_1, ..., A_n)$ `alors`
      $[\![Var]\!]^D$`.f`
    `sinon`
      $[\![Var]\!]^D$
  `sinon si` *Steps* est de la forme `child::`E `alors`
    $[\![Var]\!]^D$`.`E
  `sinon si` *Steps* est de la forme `child::`E `[`*Expr*`] alors`
    $getNth_T$ `(`$[\![$`child::`E, *Var*$]\!]^D$`,` $[\![$*Expr*$]\!]^D$`)`
    `où` $T = typage_X$ (*Var* `/ child::`E)
  `sinon si` *Steps* est de la forme `child::`E `/` *Steps*$'$ `alors`
    $[\![$*Steps*$'$, *Var* `/ child::`$E]\!]^D$
  `sinon --` *Steps* est forcément de la forme `child::`E `[`*Expr*`] /` *Steps*$'$
    $[\![$*Steps*$'$, *Var* `/ child::`E `[`*Expr*`]`$]\!]^D$

Les expressions en partie gauche (*Var*) ne sont pas sujettes aux conversions, à l'exception des prédicats qu'elles peuvent contenir et que l'on traduit comme des expressions en partie droite. Afin de comprendre comment fonctionne la procédure de traduction des expressions en partie gauche, il faut considérer l'opérateur d'affectation de BPEL :

```
<assign>
  <copy>
    <from> Expr </from>
    <to> Var </to>
  </copy>
</assign>
```

On serait tenté de traduire cet opérateur par une simple affectation *Var* `:=` *Expr* en LOTOS NT. Ce n'est malheureusement pas aussi simple, car pour affecter la valeur V au champ f d'une variable X en LOTOS NT, il faut écrire X `:=` X`.{`f`=>`V`}` ; l'écriture $X.f$ `:=` V n'est pas permise. Il est donc nécessaire de propager l'expression à affecter (V dans l'exemple) dans la procédure de traduction afin de pouvoir l'utiliser lors de l'"affectation finale" (f`=>`V dans notre exemple). Nous notons $[\![Var \leftarrow Expr]\!]^G$ la traduction de l'affectation du résultat de l'évaluation de l'expression XPATH *Expr* à la variable ou au champ de variable sélectionné par *Var*. $[\![Var \leftarrow Expr]\!]^G$ est définie ainsi :

$[\![Var \leftarrow Expr]\!]^G$ =
  `si` *Var* est de la forme X `alors`
    X `:=` $[\![Expr]\!]^D$
  `sinon si` *Var* est de la forme $X.P$ `alors`
    X `:=` X`.{`P`=>`$[\![Expr]\!]^D$`}`
  `sinon si` *Var* est de la forme X `/` *Steps* `alors`
    X `:=` $[\![Steps, X \leftarrow Expr]\!]^G$
  `sinon --` *Var* est forcément de la forme X`.`P `/` *Steps*
    X `:=` X`.{`P`=>`$[\![Steps, X.P \leftarrow Expr]\!]^G$`}`

où $[\![Steps, Var \leftarrow Expr]\!]^G$ désigne la traduction de l'affectation du résultat de l'évaluation de l'expression XPath *Expr* dans la variable ou le champ de variable sélectionné par l'évaluation du chemin d'accès *Steps* sur *Var*. La définition de $[\![Steps, Var \leftarrow Expr]\!]^G$ est similaire à celle de $[\![Steps, Var]\!]^D$, il faut simplement prendre en compte la valeur (*Expr*) à affecter :

$[\![Steps, Var \leftarrow Expr]\!]^G$ =
  si *Steps* est de la forme `attribute::`A alors
    $[\![Var]\!]^D$.{A=>$[\![Expr]\!]^D$}
  sinon si *Steps* est de la forme `child::text()` alors
    si $typage_X$ (*Var*) = **extension** $(T, A_1, ..., A_n)$ alors
      $[\![Var]\!]^D$.{f=>$[\![Expr]\!]^D$}
    sinon
      $[\![Expr]\!]^D$
  sinon si *Steps* est de la forme `child::`E alors
    $[\![Var]\!]^D$.{E=>$[\![Expr]\!]^D$}
  sinon si *Steps* est de la forme `child::`E `[`*Expr′*`]` alors
    $setNth_T$ (`child::`$E]\!]^D$ *Var*, $[\![Expr']\!]^D$, $[\![Expr]\!]^D$)
    où $T = typage_X$ (*Var* / `child::`E)
  sinon si *Steps* est de la forme `child::`E / *Steps′* alors
    $[\![Var]\!]^D$.{E=>$[\![Steps', Var$ / `child::`$E \leftarrow Expr]\!]^G$
  sinon -- *Steps* est forcément de la forme `child::` E `[`*Expr′*`]` / *Steps′*
    $[\![Var]\!]^D$.{E=>
      $setNth_T$ (
        $[\![Var$ / `child::`$E]\!]^D$,
        $[\![Expr']\!]^D$,
        $[\![Steps', Var$ / `child::`E `[`*Expr′*`]` $\leftarrow Expr]\!]^G$
)
    }

Le dernier cas (`child::` E `[`*Expr′*`]` / *Steps′*) ne peut se traduire directement par récursion comme pour la traduction des expressions en partie droite. Ceci est du à l'utilisation de la fonction *setnth* que nous appelons avec des arguments différents selon que le chemin d'accès à traiter est `child::`E `[`*Expr′*`]` ou `child::` E `[`*Expr′*`]` / *Steps′*. Pour la traduction des expressions en partie droite, la fonction *getnth* était appelée avec les mêmes arguments dans les deux cas, ce qui permettait de simplifier la traduction du second cas par un appel à la traduction du premier.

Chapitre 9

Traduction des interfaces WSDL

9.1 Présentation du langage WSDL

Un service Web communique avec un ou plusieurs entités que l'on appelle *son ou ses partenaire(s)*. Il est connecté à ses partenaires au moyen de liens de communication, chaque lien le connectant à un et un seul partenaire. Sur un lien de communication, le service et son partenaire échangent des données au moyen d'appels de fonctions distantes. A cet effet, le service ainsi que son partenaire peuvent fournir un ensemble de fonctions distantes (cf. section 9.1.2) que l'autre peut appeler (on parle alors de *fonctionnalité de service*, cf. section 9.1.3). Si le partenaire du service fournit un ensemble de fonctions au service, alors ce partenaire est forcément un service Web, sinon il peut s'agir soit d'une simple application communicante soit d'un service Web. Chaque fonction prend comme argument un message WSDL (cf. section 9.1.1) et renvoie un message WSDL ou, en cas d'erreur, une exception (dont les données associées sont contenues dans un message WSDL). On désigne par *type de lien* (cf. section 9.1.4) la paire constituée des fonctionnalités des deux services que le lien connecte. WSDL (*Web Services Description Language*) est un langage à syntaxe XML pour la description des types des liens de communication par lesquels un service communique avec ses partenaires.

La plupart du temps, la définition WSDL d'un service Web est donnée a posteriori, lorsque l'on cherche à transformer une application distante en service Web. Mais, dans le cas de BPEL, les fonctionnalités définies en WSDL sont directement utilisées en BPEL lors des communications entre services, comme nous le verrons au chapitre suivant.

Nous utilisons les conventions ci-dessous pour les identificateurs WSDL :

- P est un identificateur de partie de message,
- M est un identificateur de message,
- F est un identificateur de fonction,
- χ est un identificateur d'exception,
- ϕ est un identificateur de fonctionnalité de service,
- R est un identificateur de rôle, et
- θ est un identificateur de type de lien de communication.

La spécification d'une interface WSDL suit la syntaxe suivante :

```
<definitions targetNamespace="Ns">
  <types>
    Types
  </types
  Message0 ... Messagenm
  PortType0 ... PortTypenp
  PartnerLinkType0 ... PartnerLinkTypenl
</definitions>
```

où :

- N_s désigne l'espace de nom qui va recevoir les identificateurs déclarés dans le fichier.
- *Types* désigne un ensemble de définitions de types XML Schema. Ces types, qui servent à décrire les valeurs échangées lors des appels de fonctions distantes entre services, peuvent aussi figurer dans les définitions de variables dans le langage BPEL, auquel cas, chaque type décrit l'ensemble des valeurs qu'une variable de ce type peut prendre.
- Chaque $Message_i$ désigne une définition d'un message WSDL qui est un enregistrement de valeurs typées.
- Chaque $PortType_i$ désigne une définition de fonctionnalité de service.
- Chaque $PartnerLinkType_i$ désigne une définition de type de lien de communication. Cette construction est un ajout de la norme BPEL [Com01, Sec. 6] au langage WSDL.

Nous ne traitons pas ici les constructions WSDL : *ports* et *services*, car elle ne sont pas nécessaires à la définition de services Web BPEL. Nous ne traitons pas non plus les liaisons SOAP pour la spécification du transport des messages, car nous avons choisi de ne pas modéliser, en LOTOS NT, le protocole de transport des messages ni la façon dont ces messages sont encodés pour le transport.

9.1.1 Messages

Un message WSDL M est un enregistrement de zéro ou plusieurs valeurs appelées *parties* : $Part_1$ ... $Part_n$. La syntaxe de déclaration d'un tel message est la suivante :

```
Message ::= <message name="M">
              Part1
              ...
              Partn
            </message>
```

Chaque partie $Part_i$ est nommée P_i et typée, soit par un élément racine XML Schema (T_i^e, cf section 7.3), soit par un type XML Schema (T_i^{psc}, cf section 7.3) :

```
Part ::= <part name="Pi" element="Tie" />
       | <part name="Pi" type="Tipsc" />
```

9.1.2 Fonctions distantes

Une fonction distante WSDL F, est fournie par un service pour être invoquée par un service partenaire. Elle prend comme argument un message WSDL M_I (*message d'entrée*) et peut renvoyer comme

résultat un message WSDL M_O (*message de sortie*). Si une erreur se produit durant l'exécution de la fonction, alors elle renvoie une exception χ_i (*fault* en anglais) à laquelle un message M_i est associé. Ce message contient les données relatives à l'exception. Il existe quatre sortes de fonctions distantes en WSDL, selon la présence et l'ordre des messages d'entrée et de sortie. BPEL [Com01, Sec. 3] n'en accepte que deux, selon que le message de sortie est présent ou non. C'est pourquoi dans la syntaxe que nous présentons ci-dessous, le message de sortie est optionnel :

Function ::= `<operation name="`F`">`
`  <input message="`M_I`" />`
`  <output message="`M_O`" />`?
`  <fault name="`χ_1`" message="`M_1`" />`
`  ...`
`  <fault name="`χ_n`" message="`M_n`" />`
`</operation>`

9.1.3 Fonctionnalité d'un service

Une fonctionnalité de service (*port type* en anglais) énumère l'ensemble des fonctions F_0 ... F_n (dont nous désignons les déclarations par *Function*$_0$ ... *Function*$_n$) qu'un service fournit à son partenaire sur un lien. La syntaxe de définition d'une fonctionnalité ϕ de service est la suivante :

PortType ::= `<portType name="`ϕ`">`
Function$_0$
...
Function$_n$
`</portType>`

9.1.4 Types de liens de communication

La déclaration d'un type de lien de communication est un ajout que fait BPEL au langage WSDL. Un type de lien de communication θ définit les deux fonctionnalités respectives ϕ_1 et ϕ_2 que les deux services connectés par un lien de type θ doivent fournir. A ϕ_1 et ϕ_2 sont respectivement associés les rôles R_1 et R_2. Ces rôles sont des identificateurs, locaux à θ, qui servent, dans les déclarations de liens de communication en BPEL (cf. section 10.1.5), à indiquer la fonctionnalité du service en cours de définition et la fonctionnalité de son service partenaire. Comme l'un des deux services connectés par un lien peut ne proposer aucune fonction, la déclaration de la seconde fonctionnalité est optionnelle :

PartnerLinkType ::= `partnerLinkType name="`θ`" >`
`  <role name="`R_1`" portType="`ϕ_1`" />`
`  <role name="`R_2`" portType="`ϕ_2`" />`?
`</partnerLinkType>`

9.1.5 Grammaire WSDL

Voici la grammaire WSDL qui résume les constructions que nous avons présentées dans les sous-sections précédentes :

$$
\begin{array}{rcl}
\mathit{Spec} & ::= & \mathit{Def}_0 \ ... \ \mathit{Def}_n \\
\\
\mathit{Def} & ::= & M \ = \ \textbf{message} \ (P_1 : T_1, ..., P_n : T_n) \\
& | & \phi \ = \ \textbf{portType} \ (F_0 = \mathit{Function}_0, ..., F_n = \mathit{Function}_n) \\
& | & \theta \ = \ \mathit{PartnerLinkType} \\
\\
\mathit{Function} & ::= & \textbf{oneWayFunction} \ (M_I, \chi_1 : M_1, ...\chi_n : M_n) \\
& | & \textbf{twoWayFunction} \ (M_I, M_O, \chi_1 : M_1, ...\chi_n : M_n) \\
\\
\mathit{PartnerLinkType} & ::= & \textbf{oneWayPartnerLinkType} \ (R : \phi) \\
& | & \textbf{twoWayPartnerLinkType} \ (R_1 : \phi_1, R_2 : \phi_2)
\end{array}
$$

9.2 Traduction en LOTOS NT

9.3 Etat de l'art des traductions d'interfaces WSDL

Nous n'avons trouvé dans la littérature scientifique aucun article qui détaille la traduction des déclarations WSDL dans le cadre d'une traduction de BPEL vers un autre formalisme en vue de vérifier formellement des services BPEL. Pourtant, nous considérons qu'il est important de traduire ces déclarations.

Premièrement, les messages WSDL sont des structures de données primordiales pour la communication des services BPEL. Malheureusement, comme nous l'avons déjà vu, peu d'approches modélisent les communications des services BPEL et encore moins traitent les données.

Deuxièmement, la traduction des types de liens de communication permet d'émettre des contraintes de types sur les communications dans le langage cible considéré. Ces contraintes sont utiles pour effectuer des vérifications de types et pour énumérer l'ensemble des messages attendus par un service lors d'une réception.

Troisièmement, en traduisant toutes les constructions WSDL qui relèvent de l'appel de fonctions distantes, le code généré pour ces appels devient facilement lisible. Par exemple, grâce à cette traduction, la réception, sur le lien de communication λ, du message d'entrée X de la fonction distante F donne le code LOTOS NT suivant :

λ (?F_Input (X))

9.4 Définition de la traduction

Afin de simplifier la présentation de la traduction du langage WSDL vers LOTOS NT, nous faisons ici abstraction des espaces de noms ainsi que de la concordance des identificateurs entre les deux langages. Néanmoins, ces deux aspects sont pris en compte dans notre traduction automatique (cf. chapitre 11).

Les règles de traduction de WSDL en LOTOS NT sont fournies dans la table 9.1. Chaque règle produit du code LOTOS NT à partir d'une construction WSDL. Les sections suivantes donnent des explications détaillées sur chaque règle de traduction.

Spec :		
$[\![Spec]\!]$	=	$[\![Def_1]\!]...[\![Def_n]\!]$
Def :		

$[\![M$ = **message** $(P_1 : T_1, ..., P_n : T_n)]\!]$	=	`type` M `is`
		M `(`P_1`:`T_1`,...,`P_n`:`T_n`)`
		`end type`

$[\![\phi$ = **portType** $(F_0 = Function_0, ..., F_n = Function_n)]\!]$	=	`type` ϕ `is`
		$[\![F_0 = Function_0]\!]$ `,`
		... `,`
		$[\![F_n = Function_n]\!]$
		`end type`

$[\![\theta$ = **oneWayPartnerLinkType** $(R : \phi)]\!]$	=	`channel` θ `(`ϕ`) end channel`
$[\![\theta$ = **twoWayPartnerLinkType** $(R_1 : \phi_1, R_2 : \phi_2)]\!]$	=	`channel` θ `(`ϕ_1`,` ϕ_2`) end channel`
Function :		
$[\![F =$ **oneWayFunction** $(M_I, M_O, \chi_1 : M_1, ...\chi_n : M_n)]\!]$	=	F`_Input (`F_m_I`:`M_I`),` F`_Fault_`χ_1 `(`χ_1`:`M_1`),` ... F`_Fault_`χ_n `(`χ_n`:`M_n`)`
$[\![F =$ **twoWayFunction** $(M_I, M_O, \chi_1 : M_1, ...\chi_n : M_n)]\!]$	=	F`_Input (`F_m_I`:`M_I`),` F`_Output (`F_m_O`:`M_O`),` F`_Fault_`χ_1 `(`χ_1`:`M_1`),` ... F`_Fault_`χ_n `(`χ_n`:`M_n`)`

Table 9.1: Sémantique dénotationnelle de WSDL

9.5 Traduction d'une interface WSDL

Une interface WSDL *Spec* comprend un ensemble de définitions $Def_0...Def_n$, donc sa traduction en LOTOS NT $[\![Spec]\!]$ correspond à la concaténation des traductions respectives en LOTOS NT de ses définitions.

9.6 Traduction des messages

Chaque message WSDL M est traduit en LOTOS NT par un enregistrement (type avec un unique constructeur) M dont les champs $P_1...P_n$ correspondent aux parties $P_1...P_n$ du message.

9.7 Traduction des fonctionnalités de services

La traduction en LOTOS NT des fonctionnalités de services WSDL illustre une différence fondamentale entre les communications de services Web et les communications de processus LOTOS NT. Soient S_1 et S_2 deux services Web connectés par le lien λ. S_2 fournit à S_1 une fonction distante F prenant comme message d'entrée M_I, comme message de sortie M_O. En BPEL, l'invocation de F par S_1 s'écrit de la façon suivante :

```
<invoke partnerLink="λ"
  operation="F"
  inputVariable="X_I"
  outputVariable="X_O" />
```

où X_I est la variable contenant la valeur du message d'entrée à envoyer à S_2 et X_O est la variable dans laquelle est enregistré le message de sortie renvoyé par S_2.

L'exécution de cette fonction par S_2 débute par la réception du message d'entrée, continue par le traitement du corps de la fonction et se termine par l'envoi du message de sortie :

```
<receive partnerLink="λ"
  operation="F"
  variable="X'_I" />
...
<reply partnerLink="λ"
  operation="F"
  variable="X'_O" />
```

où X'_I est la variable dans laquelle est enregistré le message d'entrée reçu de S_1 et X'_O est la variable contenant la valeur du message de sortie à renvoyer à S_1.

Pour S_1, l'invocation de l'opération F est un événement atomique tandis que pour S_2, l'invocation de F représente plusieurs événements donc deux événements de communication (la réception du message d'entrée et l'envoi du message de sortie). Cette asymétrie n'est pas reproductible en LOTOS NT. Dans ce langage, les communications sont effectuées par synchronisation sur une porte de communication. Pour chaque processus engagé dans cette synchronisation, il s'agit d'un événement atomique qui ne peut être scindé en plusieurs étapes.

Une invocation de fonction distante se traduit donc en LOTOS NT par deux événements de communication, l'un pour envoyer le message d'entrée et l'autre pour recevoir le message de sortie. Par

exemple, en LOTOS NT, l'invocation de F faite par S_1 serait la suivante :

λ (!F_Input (X_I));
λ (!F_Output (X_O))

Nous montrons à la section 10.2.5 comment garantir l'exécution atomique de ces deux étapes successives.

En ce qui concerne les exceptions, lorsqu'une fonction lève une exception χ, elle est renvoyée en lieu et place du message de sortie :

λ (!F_χ (X))

Nous traduisons donc une déclaration de fonction distante F par une liste de constructeurs constituée de F_Input pour l'échange du message d'entrée, F_Output pour l'échange du message de sortie et F_χ_1...F_χ_n pour l'échange des exceptions. Ces constructeurs sont insérés dans la traduction de la fonctionnalité de service qui déclare F.

9.7.1 Traduction des fonctionnalités de service

Chaque fonctionnalité de service ϕ est traduite dans un type LOTOS NT qui contient tous les constructeurs LOTOS NT associés aux fonctions distantes F_0...F_n qu'elle déclare.

9.7.2 Traduction des types de liens de communication

Un type de lien de communication θ est traduit par un canal LOTOS NT de même nom. Ce canal LOTOS NT a pour but d'interdire les échanges de valeurs qui ne correspondent pas aux appels des fonctions définies par la ou les fonctionnalités de services déclarées par θ.

Les rôles associés par les types de liens de communication aux fonctionnalités de service servent en BPEL à définir les fonctions fournies, sur un lien, par le service en cours de définition à son partenaire sur ce lien, et inversement. Ces rôles servent également à s'assurer que, sur chaque lien, le service en cours de définition invoque seulement les fonctions fournies par son partenaire sur ce lien, et inversement. Nous ne pouvons reproduire ce mécanisme en LOTOS NT donc nous ne prenons pas les rôles en compte dans notre traduction. En revanche, nous vérifions statiquement, avant la traduction que les fonctions distantes appelées, sur chaque lien, par le service en cours de définition correspondent bien aux fonctions fournies par son partenaire sur ce lien, et inversement.

9.7.3 Traduction des exceptions

BPEL possède une construction nommée `catchAll` (cf. section 10.1.13) qui permet la capture d'exceptions de façon générique. Comme LOTOS NT est un langage fortement typé, nous devons, pour reproduire ce mécanisme, regrouper dans un même type LOTOS NT toutes les exceptions $\chi_1 : M_1, ...\chi_m : M_m$ déclarées dans les définitions de fonctions distantes WSDL. Dans ce type, à chaque exception χ correspond un constructeur de même nom qui prend comme argument une valeur de type M, le message contenant les données associées à χ. Comme BPEL permet la levée et la capture d'exceptions sans prise en compte des données associées, il faut aussi prévoir, pour chaque exception χ, un second constructeur, également nommé χ, qui ne prend pas d'argument. LOTOS NT permet la déclaration de plusieurs constructeurs aux identificateurs identiques au sein d'un même type si les arguments de ces constructeurs sont de types différents.

La norme BPEL considère un ensemble d'exceptions prédéfinies (cf. section 10.2.7). Ces exceptions correspondent à des erreurs qui se produisent à l'exécution, telles que l'accès à une variable non initialisée, et qui n'ont pas de données associées. Elles n'ont donc aucun rapport avec les exceptions des fonctions distantes. Néanmoins, comme elles peuvent être levées et capturées (à l'aide d'opérateurs BPEL) au sein d'un même service, il est nécessaire de leur associer un constructeur dans le type LOTOS NT pour les exceptions. Nous notons ces exceptions $\chi_{m+1} \cdots \chi_n$.

Nous appelons `Exceptions` le type LOTOS NT regroupant les exceptions. Sa définition est la suivante :

```
type Exceptions is
  χ1 (f_1:M1),
  χ1,
  ...
  χm (f_m:Mm),
  χm,
  χm+1,
  ...
  χn
end type
```

Chapitre 10

Traduction des services BPEL

10.1 Présentation du langage BPEL

Bpel [Com01] (*Business Process Execution Language*) est un langage à syntaxe Xml principalement destiné à la définition de services Web composites qui sont construits par assemblage de services Web existants. Ces services Web composites peuvent être de deux sortes : concrets ou abstraits. Nous nous intéressons dans le cadre de cette étude aux services Web composites concrets, c'est-à-dire ceux dont le comportement est complètement spécifié. Dans les services Web composites abstraits, il est possible (mais pas obligatoire [Com01, Sec. 13.1]) de cacher certaines instructions pour ne laisser apparaître que les réceptions et envois de messages. Les services Web composites abstraits ne sont donc pas exécutables et n'ont qu'une fonction documentaire. A ce propos, certains auteurs [VdADH+05] critiquent la possibilité de mélanger dans le même langage des services concrets et abstraits.

Un service Web Bpel concret est, selon les outils, interprété par un moteur d'exécution (ActiveBpel par exemple), ou traduit en Java, compilé puis exécuté (comme c'est le cas dans Oracle Bpel Manager et Ibm WebSphere ainsi que dans le projet Bpel pour Eclipse[54]).

10.1.1 Modèle de communication

Comme expliqué à la section 9.1, un service Web est exécuté en parallèle de différentes entités avec lesquelles il communique au moyen de liens de communication, chaque lien connectant le service à exactement une entité. Dans le vocabulaire des services Web, ces entités sont appelées des partenaires. Le service et son partenaire sur un lien échangent des données au moyen d'appels de fonctions distantes. Les ensembles de fonctions fournies sur chaque lien par le service et son partenaire sont décrits par le type Wsdl (cf. section 9.1.4) du lien de communication qui les connecte. Si le partenaire fournit un ensemble de fonctions au service, alors ce partenaire est un service Web, sinon il s'agit soit d'une simple application distante, soit d'un service Web.

Un service Web Bpel peut être amené à appeler des fonctions distantes fournies par ses partenaires et à exécuter des fonctions pour ses partenaires. Il n'est pas possible en Bpel de déclarer et d'appeler des fonctions locales. Une fonction distante reçoit comme entrée un message Wsdl (cf. section 9.1.1) et renvoie soit un message Wsdl, soit une exception, si une erreur s'est produite durant l'exécution du corps de la fonction. Par exemple, le service Web Bpel le plus simple fonctionne comme une

[54] `http://www.eclipse.org/bpel`

boucle infinie comportant les étapes suivantes :

- réception du message d'entrée d'une fonction fournie par le service,
- exécution d'instructions (appelées *activités* dans le vocabulaire de Bpel) en vue de produire un message de sortie,
- envoi du message de sortie ou d'une exception, si une erreur a été levée par l'exécution des instructions.

En Bpel, les liens de communication sont dynamiques, c'est-à-dire que l'adresse d'un lien de communication peut être modifiée de façon à changer l'entité avec laquelle le service communique sur ce lien. L'adresse d'un nouveau partenaire peut être envoyée au service par le biais d'un appel de fonction distante. Ce mécanisme de mobilité par passage de liens de communication (et non pas de mobilité des services eux-mêmes) est inspiré directement du π-calcul [Mil99].

10.1.2 Vue d'ensemble d'un service Web BPEL

Le comportement d'un service Web Bpel est défini par une activité. Les activités sont des constructions issues des deux langages, Wsfl et Xlang, qui ont fusionné pour donner Bpel. Elles sont choisies parmi :

- "`empty`" (cf. section 10.1.16) pour ne rien faire,
- "`exit`" (cf. section 10.1.17) pour terminer l'exécution du service,
- "`wait`" (cf. section 10.1.18) pour attendre un certain laps de temps avant de poursuivre l'exécution du service,
- "`receive`" (cf. section 10.1.19) pour la réception du message d'entrée d'une fonction fournie pour le service,
- "`reply`" (cf. section 10.1.20) pour l'envoi du résultat (message de sortie ou exception) d'une fonction fournie par le service,
- "`invoke`" (cf. section 10.1.21) pour invoquer une fonction distante fournie par un service partenaire,
- "`throw`" (cf. section 10.1.22) pour lever une exception,
- "`rethrow`" (cf. section 10.1.23) pour relancer une exception capturée par un gestionnaire d'exceptions,
- "`assign`" (cf. section 10.1.24) pour affecter une valeur à une variable,
- "`validate`" (cf. section 10.1.25) pour vérifier que le contenu d'une variable est conforme à son type,
- "`sequence`" (cf. section 10.1.26) pour exécuter des activités les unes après les autres,
- "`if`" (cf. section 10.1.27) pour l'exécution conditionnelle d'activités,
- "`while`" (cf. section 10.1.28), "`repeatUntil`" (cf. section 10.1.29) et "`forEach`" (cf. section 10.1.30) pour l'exécution répétée d'une activité,
- "`pick`" (cf. section 10.1.31) pour l'attente du déclenchement d'un événement parmi plusieurs,

- "`flow`" (cf. section 10.1.32) pour la mise en parallèle de plusieurs activités,

A une réception d'un message d'entrée d'une fonction peut correspondre plusieurs envois de réponses, issus de différentes branches d'exécution (dues aux activités "`if`", "`flow`" et "`pick`", par exemple), selon le déroulement du calcul du message de sortie de la fonction et la gestion des erreurs se produisant durant ce calcul. Afin de garantir que la réception du message d'entrée d'une fonction est suivie de l'envoi d'un unique résultat, Bpel fournit un mécanisme optionnel appelé "échange de message" (*message exchange*, cf. section 10.1.11).

Pour des raisons de qualité de service, un service Web Bpel peut se répliquer (mécanisme strictement équivalent à l'appel système "`fork`" dans les systèmes Unix), de façon à servir plusieurs requêtes simultanément, à la manière des serveurs Web qui utilisent plusieurs fils (*threads*) d'exécution. Lorsque plusieurs réplications du même service sont en cours d'exécution, il est nécessaire d'utiliser le mécanisme d'ensembles de corrélations (cf. section 10.1.12) afin d'identifier à quelle réplication sont destinés les messages reçus.

Bpel propose pour la gestion des erreurs un mécanisme complexe à trois facettes :

- Les gestionnaires d'exceptions (cf. section 10.1.13) capturent et traitent les exceptions levées par les activités d'un service. La levée d'une exception se fait soit de façon explicite, par l'utilisation de l'activité "`throw`" (cf. section 10.1.22), soit de façon implicite, lors de l'invocation d'une fonction distante dont le résultat peut être une exception. Le traitement d'une exception par un gestionnaire d'exception se fait à l'aide d'une activité définie par le gestionnaire.

- Les gestionnaires de compensation (cf. section 10.1.35) sont des activités, au nombre d'une par service, qui tentent d'annuler les changements que l'exécution du service a entraînés : invocation de fonctions distantes et modifications des valeurs des variables internes au service, principalement. Le gestionnaire de compensation d'un service ne peut être déclenché que par un gestionnaire d'exception, au moyen de l'activité `compensate` (cf. section 10.1.35).

- Les gestionnaires de terminaison (cf. section 10.1.34) sont des activités au nombre d'une par service. Le gestionnaire de terminaison d'un service est exécuté lorsque le service doit être terminé avant d'avoir fini son exécution normale, lorsqu'une exception n'est pas capturée.

En parallèle d'un service, peuvent être exécutées, à la suite d'événements (cf. section 10.1.14), des activités. Bpel distingue deux types d'événements (il s'agit de constructions différentes des exceptions), les *réceptions de messages* ("`onEvent`") et les *alarmes* ("`onAlarm`"). Un gestionnaire d'événement déclare un événement et le lie avec une activité à exécuter lorsque l'événement se produit.

Un service Web Bpel déclare un ensemble de variables qui peuvent être lues et écrites par les activités qui définissent son comportement ainsi que par ses gestionnaires d'exceptions, de compensation, de terminaison et d'événements. Les variables sont typées mais ce typage est dynamique et optionnel. C'est-à-dire qu'en Bpel, les valeurs affectées aux variables ne sont pas validées par rapport au type de la variable. Cette validation ne se fait que sur ordre explicite de l'utilisateur, par l'intermédiaire de l'activité "`validate`" (cf. section 10.1.25).

Afin de structurer le comportement d'un service Web Bpel, il est possible de le décomposer en une hiérarchie de sous-services. Un sous-service est une activité spéciale ("`scope`", cf. section 10.1.34) qui encapsule une autre activité et déclare ses propres variables, liens de communication et gestionnaires d'exceptions, de compensation, de terminaison et d'événements. Une variable ne peut être lue et écrite que par l'activité du service ou sous-service qui la déclare. Un lien de communication ne peut être utilisé qu'au sein de l'activité du service ou sous-service qui le déclare. Un gestionnaire d'exception ne peut capturer que des exceptions levées par l'activité du service ou sous-service qui le déclare. Un gestionnaire de compensation ne peut annuler que les changements effectués par le sous-service qui le déclare. Un gestionnaire de terminaison ne s'exécute que lorsque l'exécution du

sous-service qui le déclare est terminée inopinément. L'événement d'un gestionnaire d'événement ne peut se produire que si l'exécution de l'activité du sous-service qui le déclare n'est pas terminée. Les gestionnaires de compensation et de terminaison ne s'appliquent en réalité qu'aux sous-services et ne peuvent être déclarés par le service principal.

10.1.3 Relations entre activités, services et sous-services

Nous définissons ici le vocabulaire que nous employons pour décrire les relations entre activités, services et sous-services.

Activité parente de et activité fille de : si une activité a définit les activités $a_0, ...a_n$ (par exemple, l'activité "`if`" définit deux activités, une pour le cas où la condition est vraie, une autre pour le cas où la condition est fausse), alors a est *l'activité parente de* $a_0, ..., a_n$ et $a_0, ..., a_n$ sont les *activités filles de* a.

Activité comprise dans : la relation *activité comprise dans* est la fermeture transitive de la relation *activité fille de*.

Sous-service contenant : un sous-service s contient une activité a si a est l'activité de s ou si a est *comprise dans* l'activité de s.

Service contenant : un service s contient une activité a si a est l'activité de s ou si a est *comprise dans* l'activité de s.

Sous-service englobant : un sous-service s englobe une activité a si s contient a et qu'il n'existe pas de sous-service s' tel que s contient s' (en BPEL un sous-service est défini au moyen d'une activité) et s' contient a.

Service englobant : un service s englobe une activité a si s contient a et qu'il n'existe pas de sous-service s' tel que s contient s' et s' contient a.

10.1.4 Abréviations syntaxiques

Nous présentons dans les sections qui viennent une syntaxe canonique de BPEL, c'est-à-dire sans les constructions énumérées ci-dessous, qui relèvent du sucre syntaxique :

- Nous ne considérons pas les initialisations de variables lors de leur déclaration (construction *from-spec* [Com01, Sec. 8.1]) car elles peuvent être remplacées par des affectations ultérieures.
- Dans les opérations de communication, nous ne considérons pas les constructions *from-parts* et *to-parts* [Com01, Sec. 10.4] qui permettent respectivement, à l'aide d'expressions XPATH, de construire à la volée le message d'entrée d'une fonction et d'affecter les parties du message de sortie à des champs de variables. Ces mécanismes peuvent être remplacés par des affectations antérieures ou ultérieures à l'invocation de la fonction.
- Dans les affectations, nous ne considérons pas les sélections de valeurs par les constructions suivantes :

  `<from variable="`X`" part="`P`">`
  `  <query>` *Steps* `</query>`
  `/from>`

  et

```
<to variable="X" part="P">
  <query> Steps </query>
/to>
```

car elles peuvent être remplacées par une sélection de valeur à l'aide de l'expression XPath : $X.P$ / *Steps*.

- Dans l'activité de branchement conditionnel "`if`", nous ne considérons pas l'existence de la notation abrégée "`elsif`".
- Les activités de communication attendent normalement une variable dont le type est un message Wsdl M (cf. section 9.1.1, mais peuvent accepter à la place une variable dont le type est un élément racine Xml Schema T^e (cf. section 7.3), dans le cas où M est composé d'exactement une partie dont le type est T^e. Pour ces activités, nous ne considérons pas la syntaxe alternative traitant l'élément Xml Schema.

10.1.5 Syntaxe concrète d'un service BPEL

Un fichier Bpel contient la définition d'un (et d'un seul) service Web. Cette définition spécifie le comportement du service en composant des activités. La syntaxe concrète de définition d'un service Web Bpel est la suivante :

```
<process name="S" targetNamespace="V^tN"
  queryLanguage="V^qL"?
  expressionLanguage="V^eL"?
  suppressJoinFailure="V^s"?
  exitOnStandardFault="V^e"?
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
    (<extensions>
      Extension_0 ... Extension_m
     </extensions>)?
    Import_1 ... Import_n
    <partnerLinks>
      PartnerLink_0 ... PartnerLink_p
    </partnerLinks>
    (<messageExchanges>
      MessageExchange_0 ... MessageExchange_q
     </messageExchanges>)?
    (<variables>
      Variable_0 ... Variable_r
     </variables>)?
    (<correlationSets>
      CorrelationSet_0 ... CorrelationSet_s
     </correlationSets>)?
```

```
        (<faultHandlers>
           FaultHandler_0 ... FaultHandler_t
         </faultHandlers>)?
        (<eventHandlers>
           EventHandler_0 ... EventHandler_u
         </eventHandlers>)?
        Activity
    </process>
```

où :

- S désigne le nom du service.
- V^{tN} (*targetNamespace*) désigne l'espace de noms dans lequel le service est déclaré.
- V^{qL} (*queryLanguage*) désigne le langage utilisé pour la sélection des champs de variables figurant en partie gauche des affectations. L'attribut "`queryLanguage`" est optionnel et s'il est omis, la norme Bpel considère que le langage XPath est utilisé. Si une activité d'affectation redéfinit la valeur de l'attribut "`queryLanguage`", alors pour, cette activité seulement, le langage utilisé est celui spécifié par la nouvelle valeur de l'attribut.
- V^{eL} (*expressionLanguage*) désigne le langage utilisé dans les expressions (dans les structures de contrôle et en partie droite des affectations). L'attribut "`expressionLanguage`" est optionnel et s'il est omis, la norme Bpel considère que le langage XPath est utilisé. Si une activité d'affectation redéfinit la valeur de l'attribut "`expressionLanguage`", alors, pour cette activité seulement, le langage utilisé est celui spécifié par la nouvelle valeur de l'attribut.
- V^{s} (*suppressJoinFailure*) est une valeur booléenne qui indique si l'exception *joinFailure* (sur laquelle nous revenons à la section 10.1.32) doit être prise en compte.
- V^{e} (*exitOnStandardFault*) est une valeur booléenne qui indique si la levée d'une exception prédéfinie doit entraîner l'arrêt du service.

Dans la suite, nous présentons tour à tour les différentes constructions et activités utilisées dans la définition d'un service Bpel et nous désignons le service S par *le service principal*.

10.1.6 Attribut spécial "suppressJoinFailure"

En Bpel, les différentes activités parallèles d'un service peuvent être liées par des contraintes de séquencement, de telle sorte que le début de l'exécution d'une activité dépende de la fin de l'exécution d'autres activités. Nous expliquons précisément ce mécanisme complexe à la section 10.1.32.

Il peut arriver qu'une activité ne soit pas autorisée à débuter son exécution (si l'une des contraites de séquencement n'est jamais satisfaite). Dans ce cas, si la valeur de l'attribut "`suppressJoinFailure`" est fausse, alors l'exception *joinFailure* est levée ; sinon, l'exception n'est pas levée. Cela permet au développeur de décider si le non-démarrage d'une activité est une erreur critique qui doit entraîner un traitement spécial (capture de l'exception *joinFailure*), provoquer un arrêt du service (pas de capture de l'exception), ou être ignorée (lorsque la valeur de l'attribut est vraie).

L'attribut "`suppressJoinFailure`" d'une déclaration de service peut être redéfini par chaque activité contenue dans ce service. Pour chaque activité, la valeur V^s de cet attribut est alors soit définie par l'activité, soit héritée de l'activité parente.

Chaque définition de l'attribut "`suppressJoinFailure`", dans un service s (resp. une activité a), a donc une portée qui correspond à l'activité du service s (resp. l'activité a) et les activités $a_0, ...a_n$

contenues dans s (resp. comprises dans a), pour lesquelles il n'existe pas d'activité a' telle que a' est contenue dans s (resp. comprise dans a), les activités $a_0, ...a_n$ sont comprises dans a' et a' définit pour l'attribut "`suppressJoinFailure`" une valeur différente de celle définie par s (resp. a).

10.1.7 Variables

La syntaxe de déclaration d'une variable en Bpel est la suivante :

Variable ::= `<variable name="`X`" messageType="`M`" />`
| `<variable name="`X`" type="`T^{psc}`" />`
| `<variable name="`X`" element="`T^{e}`" />`

Le type d'une variable Bpel est donc soit un message Wsdl, soit un type Xml Schema, soit un élément Xml Schema.

10.1.8 Inclusion de fichiers XML Schema et WSDL

L'inclusion d'un fichier Xml Schema ou Wsdl se fait par l'intermédiaire d'une construction appelée *Import* qui a la syntaxe suivante :

Import ::= `<import (namespace="`V^n`")? location="`V^l`" importType="`V^i`" />`

où V^n, V^l et V^i sont des chaînes de caractères. Si l'attribut "`namespace`" est spécifié, alors V^n représente l'espace de noms qui va contenir les noms déclarés par le fichier importé. Si ce fichier déclare un espace de noms cible, alors V^n doit lui être égal. La valeur V^i de l'attribut "`importType`" détermine le type du fichier inclus :

- `http://www.w3.org/2001/XMLSchema` pour un fichier Xml Schema
- `http://schemas.xmlsoap.org/wsdl` pour un fichier Wsdl

V^l désigne l'emplacement du fichier inclus.

Etrangement, les inclusions de fichiers ne sont pas délimitées par une paire de balises `<imports> </imports>`.

10.1.9 Extensions du langage Bpel

Un service Bpel peut faire usage d'extensions du langage : connexion à un système de gestion de base de données, accès au système de fichiers, appels de fonctions externes définies en Java, envoi d'e-mails... La sémantique de ces extensions n'est pas définie dans la norme et dépend de l'interprète Bpel utilisé. La syntaxe de déclaration d'une extension (*Extension*) est la suivante :

Extensions ::= `<extension namespace="`V^n`" mustUnderstand="`V^m`" />`

où V^n est une chaîne de caractères et V^m un booléen. La valeur de V^n a deux rôles : elle identifie l'extension et sert d'espace de noms pour les constructions introduites par l'extension. Si un interprète ne reconnaît pas une extension dont le nom est V^n, alors si V^m est vraie, il doit terminer son exécution et afficher un message d'erreur, sinon il peut se contenter de l'ignorer. Les activités définies par une extension doivent forcément apparaître à l'intérieur d'une activité "`extensionActivity`" (cf. section 10.1.33).

10.1.10 Liens de communication

La syntaxe de déclaration d'un lien de communication (*PartnerLink*) est la suivante :

PartnerLink ::= `<partnerLink name="`λ`"`
`partnerLinkType="`θ`"`
(`myRole="`R^m`"`)?
(`partnerRole="`R^p`"`)?
(`initializePartnerRole="`V^{iPR}`"`)? `/>`

où λ désigne le nom du lien de communication, tandis que θ désigne le type de ce lien. Si θ associe les rôles R_1 et R_2 aux fonctionnalités de services ϕ_1 et ϕ_2 respectivement (cf. section 9.1.3), alors les valeurs de R^m et R^n sont choisies parmi $\{R_1, R_2\}$, de telle sorte que $R_m \neq R_n$. Les valeurs de R^m et R^n déterminent, sur λ, la fonctionnalité du service principal et celle du service partenaire.

L'initialisation d'un lien peut se faire de deux façons :

- en affectant une adresse de service Web au lien (dans ce cas, le lien connectera le service en cours de définition au service se trouvant à l'adresse mentionnée)
- à la réception du premier message sur ce lien (dans ce cas, le lien connectera le service en cours de définition au service qui a envoyé le message)

V^{iPR} est une valeur booléenne qui indique si le lien doit être initialisé avant la première utilisation. Si cette valeur vaut "`true`", alors une exception est levée lorsqu'une fonction est invoquée sur ce lien avant qu'il ait été initialisé.

10.1.11 Mécanisme d'échanges de messages

La syntaxe de déclaration d'un échange de message est la suivante :

MessageExchange ::= `<messageExchange name="`M^e`" />`

où M^e désigne le nom de l'échange de message.

La réception d'un message dans un service BPEL par l'événement "`onEvent`" (cf. section 10.1.14), l'événement "`onMessage`" (cf. section 10.1.31), l'activité "`receive`" (cf. section 10.1.19), ou l'activité "`pick`" (cf. section 10.1.31) correspond à la réception du message d'entrée d'une fonction fournie par ce service. Si la fonction doit renvoyer un résultat, alors BPEL permet (c'est optionnel) d'associer un échange de message à la réception du message d'entrée et à l'envoi, par l'activité "`reply`" (cf. section 10.1.20), du résultat (le message de sortie de la fonction ou, si une erreur s'est produite, une exception), afin de s'assurer qu'un résultat et un seul a été renvoyé.

L'association d'un échange de message à un événement "`onEvent`" ou "`onMessage`", ou à une activité "`receive`" ou "`pick`", se fait à l'aide de l'attribut "`messageExchange`" que ces constructions fournissent. Le même attribut est fourni par l'activité "`reply`" pour associer un échange de message à l'envoi du résultat d'une fonction.

Le fonctionnement de cette liaison entre réception du message d'entrée d'une fonction et envoi du résultat est le suivant :

1 Au début de l'exécution d'un service ou d'un sous-service, tous les échanges de messages qu'il déclare sont *indéfinis*.

2 Lors de la réception d'un message par un événement "`onEvent`" ou "`onMessage`", ou par une

activité "`receive`" ou "`pick`", l'échange de message M^e auquel cette réception est éventuellement associée devient *ouvert*.

3 Lors de l'envoi d'une réponse par une activité "`reply`" associée à M^e, M^e devient *fermé*.

Durant l'exécution d'un service, l'interprète Bpel doit procéder à plusieurs vérifications relatives aux échanges de messages (pour garantir qu'à une réception de message correspond un unique envoi de réponse) :

- Lors de la réception d'un message par une activité ou un événement, l'échange de message M^e associé doit être *indéfini*.
- Lors de l'envoi d'une réponse par l'activité "`reply`", l'échange de message associé M^e doit être *ouvert*.
- A la fin de l'exécution d'un service ou d'un sous-service, tous les échanges de messages qu'il déclare doivent être soit *indéfinis*, soit *fermés*.

10.1.12 Mécanisme de corrélation

Un service Web Bpel peut se dupliquer en plusieurs exécutions parallèles (cf. section 10.1.19) pour traiter plusieurs requêtes simultanément. Lorsque plusieurs réplications du même service sont exécutées, le problème se pose d'identifier la réplication vers laquelle un message reçu doit être redirigé. Le mécanisme de corrélation proposé dans Bpel consiste à identifier la bonne réplication à partir des données contenues dans le message. L'idée repose sur le principe que parmi ces données, certaines (par exemple, un numéro de client) identifient de manière unique le dialogue entre la réplication et son partenaire. Une telle donnée est appelée *corrélation*. Parfois, une donnée unique n'est pas suffisante et plusieurs valeurs doivent être extraites du message reçu afin d'établir avec certitude la réplication à laquelle le message est destiné. On parle alors d'*ensemble de corrélations*.

Pour l'extraction de données à partir d'un message, Bpel permet la définition de propriétés. Une propriété est une expression nommée, à la façon des macros du langage C, dont l'intérêt est d'être réutilisable dans les définitions de différents ensembles de corrélations. Ces propriétés sont déclarés dans des fichiers Wsdl. Il s'agit d'un ajout de Bpel à la syntaxe de Wsdl. Nous avons volontairement omis ces déclarations lors de la présentation du langage Wsdl (cf. chapitre 9), car nous ne les considérons pas dans notre traduction (cf. section 10.2.1). Nous avons donc trouvé plus judicieux de les introduire dans la présente section.

Dans la définition suivante :

```
<property propertyName="clientID" type="integer" />
<propertyAlias propertyName="clientID" message="LogInMessage" part="id" />
<propertyAlias propertyName="clientID" message="OrderMessage" part="client">
  <query>id</query>
</propertyAlias>
```

la première ligne indique le nom et le type renvoyé par la propriété, il s'agit d'une déclaration abstraite. Les deux lignes qui suivent définissent concrètement la propriété pour deux messages différents. Pour le message "`LogInMessage`", il s'agit de renvoyer la valeur de la partie "`id`" du message. Pour le message `OrderMessage`, il s'agit de renvoyer l'élément "`id`" de la partie "`client`" du message.

Chaque service ou sous-service peut déclarer un nombre arbitraire d'ensembles de corrélations. La syntaxe de déclaration d'un ensemble de corrélations C avec $P_1^r, ..., P_n^r$ comme propriétés est la suivante :

CorrelationSet ::= `<correlationSet name="`C`" properties="`P_1^r ... P_n^r`" />`

Un ensemble de corrélations est utilisable dans n'importe quelle activité de communication ("`receive`", "`reply`", "`onMessage`", "`onEvent`" et "`invoke`"). Dans chaque réplication du service, un tuple de valeurs (une valeur par propriété) est associé à l'ensemble de corrélations C. Si les propriétés avec lesquelles il a été déclaré sont judicieusement choisies, alors le tuple de valeurs associé à C sera différent dans chacune des réplications du service, permettant ainsi une redirection correcte des messages.

Initialement, le tuple de valeurs associé à un ensemble de corrélation est indéfini. Lors d'une activité de communication, il est possible d'initialiser un (ou plusieurs) ensemble de corrélations C avec le message reçu dans cette activité. Cette initialisation est déclenchée par la présence de l'élément "`correlations`" au sein d'une activité de communication [Com01, Sec. 9.2]. Par la suite, toutes les activités de réception de messages qui utilisent l'ensemble de corrélations C se verront délivrer les messages reçus par le service seulement si pour ces messages, l'évaluation des propriétés de C donnent un tuple de valeurs égal à celui associé à C. Similairement, lors de l'émission d'un message dans une activité utilisant l'ensemble de corrélations C, l'évaluation des propriétés de C pour ce message doit donner un tuple égal à celui associé à C.

10.1.13 Gestionnaires d'exceptions

La gestion des erreurs dans Bpel se fait au moyen d'exceptions qui sont levées, puis capturées. Chaque exception est nommée et peut se voir associer des informations relatives à l'erreur qui s'est produite.

Les exceptions sont levées par l'activité "`throw`" (cf. section 10.1.22). Elles sont capturées et traitées par des gestionnaires d'exceptions (*fault handlers*) qui sont déclarés dans les définitions de services ou de sous-services et les activités "`invoke`" (pour traiter les exceptions renvoyées lors d'un appel de fonction distante, cf. section 10.1.21). Il existe quatre syntaxes différentes pour les gestionnaires d'exceptions :

FaultHandler ::= Capture d'une exception par son nom, sans donnée associée :
`<catch faultName="`χ`">`
Activity
`</catch>`

| Capture d'une exception par son nom, avec données associées :
`<catch faultName="`χ`" faultVariable="`X`" faultMessageType="`M`">`
Activity
`</catch>`

| Capture d'une exception anonyme, avec données associées :
`<catch faultVariable="`X`" faultMessageType="`M`">`
Activity
`</catch>`

| Capture de n'importe quelle exception :
`<catchAll>`
Activity
`</catchAll>`

Dans les déclarations de services ou de sous-service, ainsi que dans les activités "`invoke`", il arrive que plusieurs gestionnaires d'exceptions soient définis : $\mathit{FaultHandler}_0, ..., \mathit{FaultHandler}_n$. La norme Bpel [Com01, 12.5] explique en détail comment choisir le bon gestionnaire d'exception $\mathit{FaultHandler}_i$

pour capturer une exception donnée.

Les gestionnaires d'exceptions d'un service, d'un sous-service et d'une activité "`invoke`" doivent capturer toutes les exceptions qui peuvent être levées. La présence d'une clause "`catchAll`" est donc obligatoire et, si l'utilisateur n'en a pas donné, alors la clause suivante est automatiquement insérée :

```
<catchAll>
  <sequence>
    <compensate />
    <rethrow />
  </sequence>
</catchAll>
```

L'activité "`compensate`" est détaillée à la section 10.1.35 tandis que l'activité "`rethrow`" est détaillée à la section 10.1.23.

10.1.14 Gestionnaires d'événements

BPEL considère deux types d'événements. Les événements de type *réception de message* ("`onEvent`") sont déclenchés par la réception du message qui leur est associé. Les événements de type *alarme* ("`onAlarm`") sont déclenchés par l'alarme qui leur est associée. Un gestionnaire d'événement est une paire constituée d'un événement et d'une activité. Lorsque l'événement se déclenche, l'activité est exécutée. Cette activité est toujours une activité "`scope`" (cf. section 10.1.34), c'est-à-dire une définition de sous-service. L'exécution d'un gestionnaire d'événement se déroule en parallèle de l'exécution du service (ou sous-service) qui le déclare. Par conséquent, l'exécution des gestionnaires d'événements d'un service prend fin lorsque l'exécution du service se termine.

1) La syntaxe de définition d'un gestionnaire d'événement "`onEvent`" est la suivante :

EventHandler ::= `<onEvent partnerLink="`λ`"`
`portType="`ϕ`"?`
`operation="`F`"`
`messageType="`M`"`
`variable="`X`"`
`messageExchange="`M^e`"? />`
Activity
`</onEvent>`
| ...

Ce gestionnaire d'événement désigne la réception du message d'entrée de la fonction distante F (cf. section 9.1.2), fournie à son partenaire sur le lien λ par le service principal. La présence de la fonctionnalité de service ϕ ("`portType`", cf section 9.1.3) est optionnelle et a une vocation purement documentaire, car sa valeur peut être déduite de la déclaration du lien λ. Néanmoins, si ϕ est présente, sa valeur doit être égale à la fonctionnalité, sur le lien λ, du service principal. Le gestionnaire d'événement déclare une variable X dont le type est le message WSDL M (M doit correspondre au type du message d'entrée de la fonction F). La valeur de cette variable est initialisée par le message reçu. Si l'attribut "`messageExchange`" est présent, alors M^e indique l'échange de message auquel la réception du message est associée. Après réception du message, l'activité *Activity* est exécutée. L'activité d'un gestionnaire d'événement "`onEvent`" peut se répéter aussi souvent que la fonction F est invoquée. Si la fonction est invoquée plusieurs fois simultanément, alors le gestionnaire d'événement va se dupliquer pour traiter toutes les invocations en parallèle.

2) La syntaxe d'un gestionnaire d'événement "`onAlarm`" est la suivante :

EventHandler ::= ...
| `<onAlarm>`
`(`
`<for>`*Expr*$_1$`</for>`
`|`
`<until>`*Expr*$_2$`</until>`
`)?`
`<repeatEvery>`*Expr*$_3$`</repeatEvery>?`
Activity
`</onAlarm>`

L'exécution temporisée de l'activité *Activity* dépend des valeurs de *Expr*$_1$, *Expr*$_2$ et *Expr*$_3$. *Expr*$_1$ et *Expr*$_3$ représente des valeurs de temps relatifs (type XML Schema duration) tandis que *Expr*$_2$ représente une valeur de temps absolu (type XML Schema dateTime). Cinq cas sont possibles :

1. Si la clause contenant *Expr*$_1$ est présente mais que la clause contenant *Expr*$_3$ ne l'est pas, alors l'exécution de l'activité démarre après une attente dont la durée est spécifiée par l'expression *Expr*$_1$.

2. Si la clause content *Expr*$_2$ est présente et que la clause contenant *Expr*$_3$ ne l'est pas, alors l'exécution de l'activité démarre lorsque la date (en temps absolu) spécifiée par *Expr*$_2$ est atteinte.

3. Si les clauses contenant *Expr*$_1$ et *Expr*$_3$ sont présentes, alors l'exécution de l'activité démarre lorsque la durée spécifiée par l'expression *Expr*$_1$ s'est écoulée. L'exécution de l'activité se répète tant que l'activité du service contenant le gestionnaire d'événement n'est pas terminée. Entre deux exécutions consécutives de l'activité, s'écoule la durée spécifiée par *Expr*$_3$.

4. Si les clauses contenant *Expr*$_2$ et *Expr*$_3$ sont présentes, alors l'exécution de l'activité démarre lorsque la date spécifiée par l'expression *Expr*$_2$ est atteinte. L'exécution de l'activité se répète tant que l'activité du service contenant le gestionnaire d'événement n'est pas terminée. Entre deux exécutions consécutives de l'activité, s'écoule la durée spécifiée par *Expr*$_3$.

5. Si seule la clause contenant *Expr*$_3$ est présente, alors l'activité va s'exécuter de nouveau, chaque fois que la durée spécifiée par *Expr*$_3$ sera écoulée.

10.1.15 Activités

A présent, nous passons en revue les différents types d'activités du langage BPEL. Comme nous allons le voir dans les sections suivantes, il existe deux sortes d'activités :

- les activités *atomiques* dont l'exécution ne peut être ni interrompue, ni entrelacée avec d'autres activités parallèles, et
- les activités *structurées* dont l'exécution n'est pas atomique et qui sont définies par composition de plusieurs activités.

10.1.16 Activité "empty"

L'activité atomique "`empty`", dont la syntaxe est :

Activity ::= `<empty />`
| ...

désigne l'activité qui ne fait rien. Elle sert, par exemple, à traiter une exception en ne faisant rien dans le gestionnaire de l'exception ou à réaliser une barrière de synchronisation entre différentes exécutions parallèles (cf. section 10.1.32).

10.1.17 Activité "exit"

L'activité atomique "`exit`", dont la syntaxe est :

Activity ::= ...
| `<exit />`
| ...

entraîne la terminaison du service Web. Si cette activité est exécutée dans une réplication du service (cf. section 10.1.19), alors l'exécution de la réplication se termine.

10.1.18 Activité "wait"

L'activité atomique "`wait`" propose deux syntaxes d'utilisation :

Activity ::= ...
| `<wait>`
  `<for>`$Expr_1$`</for>`
  `</wait>`
| `<wait>`
  `<until>`$Expr_2$`</until>`
  `</wait>`
| ...

qui réalisent, pour la première, l'attente pendant une durée $Expr_1$ (temps relatif) et, pour la seconde, l'attente jusqu'à une date fixée $Expr_2$ (temps absolu).

10.1.19 Activité "receive"

L'activité atomique "`receive`" dont la syntaxe est :

Activity ::= ...
| `<receive partnerLink="`λ`"`
  (`portType="`ϕ`"`)?
  `operation="`F`"`
  `variable="`X`"`
  (`createInstance="`V^c`"`)?
  (`messageExchange="`M^e`"`)?`/>`
| ...

désigne la réception du message d'entrée d'une fonction fournie par le service en cours de définition. λ, ϕ, F et M^e ont la même signification que pour l'événement "`onEvent`" (cf. section 10.1.14). Si l'attribut "`createInstance`" est présent et que sa valeur V^c est vraie, alors à chaque réception du message, le service va se répliquer (à l'instar d'un programme Unix invoquant l'appel système "`fork`") ; la copie du service traitera le message qui vient d'être reçu tandis que le service original

continuera d'attendre l'arrivée du prochain message. Si l'attribut "`createInstance`" est absent, ou que sa valeur V^c est *fausse*, alors le service traite le message sans se répliquer.

V^c ne peut être vraie que dans la première activité de réception de message du service. L'exemple qui suit, dans lequel le protocole de communication entre un service et son partenaire veut que le partenaire invoque deux fonctions distantes successives "`placeOrder`" et "`checkOut`", est invalide :

```
<receive partnerLink="λ"
  portType="φ"
  operation="placeOrder"
  variable="X_I"
  createInstance="yes" />
...
<receive partnerLink="λ"
  portType="φ"
  operation="checkOut"
  variable="X'_I"
  createInstance="yes" />
```

10.1.20 Activité "reply"

L'activité atomique "`reply`" dont la syntaxe est la suivante :

```
Activity ::= ...
           | <reply partnerLink="λ"
               (portType="φ")?
                operation="F"
                variable="X"?
               (faultName="χ"?
               (messageExchange="M^e")? />
```

désigne l'envoi du message de sortie d'une fonction distante fournie par le service en cours de définition. λ, ϕ, F et M^e ont la même signification que pour l'activité "`receive`" (cf. section 10.1.19). La variable X, si présente, contient le message de sortie de la fonction F. Si l'attribut "`faultName`" est présent, alors la chaîne de caractères χ doit correspondre à l'une des exceptions déclarées dans la définition de F et, dans ce cas, la variable X doit être présente et contient les données associées à l'exception.

10.1.21 Activité "invoke"

L'activité atomique "`invoke`" dont la syntaxe est la suivante :

```
Activity ::= ...
             <invoke partnerLink="λ"
               portType="φ"?
               operation="F"
               inputVariable="X_I"
               outputVariable="X_O"? />
               FaultHandler_1 ... FaultHandler_n
             </invoke>
           | ...
```

représente l'appel de la fonction distante F fournie par le partenaire, dont la fonctionnalité de service est ϕ, sur le lien λ. La variable X_I contient la valeur du message passé en entrée de la fonction. La présence de la variable X_O doit concorder avec la présence d'un message de sortie dans la définition de la fonction F. Si la variable X_O est présente, alors elle reçoit le résultat (message de sortie) renvoyé par le service partenaire. Les éventuels gestionnaires d'exceptions $\mathit{FaultHandler}_1$ ... $\mathit{FaultHandler}_n$ traitent, le cas échéant, les exceptions levées par le service partenaire lors de l'exécution de la fonction.

10.1.22 Activité "throw"

L'activité atomique "`throw`" dont la syntaxe est :

Activity ::= ...
| `<throw` (`faultName="`χ`"`)? (`faultVariable="`X`"`)? `/>`
| ...

permet la levée d'une exception χ dont les données associées sont, si elles existent, contenues dans la variable X. Si le nom χ de l'exception n'est pas précisé, alors il s'agit d'une levée de faute anonyme qui sera capturée soit en fonction de ces données associées si elle en transporte, soit par un gestionnaire d'exception "`catchAll`" sinon.

La levée d'une exception χ dans une activité a entraîne sa terminaison immédiate ainsi que la terminaison (la norme BPEL ne précise pas la rapidité de cette terminaison) de toutes les activités contenues dans le service ou sous-service s englobant a, dont les gestionnaires d'exceptions vont capturer χ.

10.1.23 Activité "rethrow"

L'activité atomique `<rethrow />` dont la syntaxe est la suivante :

Activity ::= ...
| `<rethrow />`
| ...

ne peut être comprise que dans l'activité d'un gestionnaire d'exception. Elle propage l'exception traitée par un gestionnaire d'exception aux gestionnaires d'exceptions du service ou sous-service englobant (cf. section 10.1.3).

10.1.24 Activité "assign"

L'activité "`assign`" dont la syntaxe est la suivante :

Activity ::= ...
| `<assign validate="`V^v`"? >`
Copy$_1$ ... *Copy*$_n$
`</assign>`
| ...

Copy ::= `<copy keepSrcElementName="`V^k`"?`
`ignoreMissingFromData="`V^i`"? >`
From
To
`</copy>`

From ::= `<from partnerLink="`λ`" endpointReference="(myRole|partnerRole)" />`
| `<from>` *Expr* `</from>`
| `<from><literal>` *Literal* `</literal></from>`

To ::= `<to partnerLink="`λ`" />`
| `<to>` *Expr* `</to>`

permet l'affectation de variables et de liens de communication. Cette activité effectue les copies $Copy_1, ..., Copy_n$. Chaque copie affecte la valeur d'une source vers une destination. Si l'attribut "`validate`" est présent et que la valeur booléenne V^v vaut "`vrai`", alors chaque affectation donne lieu à un contrôle de type à l'exécution (on parle de validation) : l'interprète BPEL doit vérifier que le type de la destination est compatible avec le type de la source. Si V^v est absente ou bien qu'elle vaut "`faux`", alors aucune validation n'est effectuée et la valeur de la source est affectée à la destination, quel que soit son type. Par conséquent, les situations dans lesquelles des variables BPEL contiennent des valeurs qui ne correspondent pas à leur type sont possibles.

La première syntaxe du non-terminal *From* désigne la sélection de l'adresse de l'une des entités connectées par le lien λ : le service en cours de définition si l'attribut "`endpointReference`" vaut "`myRole`", le partenaire si l'attribut "`endpointReference`" vaut "`partnerRole`". La seconde syntaxe désigne la sélection d'une valeur égale au résultat de l'évaluation de l'expression XPATH *Expr* (cf. section 8.3), à la façon d'une *R-value* dans les langages de programmation classiques. La troisième syntaxe sélectionne comme source le terme XML défini par *Literal* (cf. section 7.5.1).

La première syntaxe du non-terminal *To* désigne l'adresse du service partenaire sur le lien λ. Dans ce cas, la source doit être une adresse de lien de communication. Une telle valeur est sélectionnée par 1) la lecture d'une valeur constante décrivant une adresse de service, au format spécifié dans la norme BPEL [Com01, Sec. 6.3], 2) la lecture de la valeur d'une variable ou d'un champ de variable dont le type XML Schema est le format d'adresse décrit dans la norme BPEL ou 3) la sélection, pour un lien donné, de l'adresse du service partenaire ou bien du service en cours de définition. L'affectation d'une adresse à un lien de communication a pour effet de l'initialiser, s'il ne l'était pas déjà. Il s'agit du mécanisme de mobilité par passage de liens de communication mentionné à la section 10.1.1. La seconde syntaxe désigne une variable ou un champ de variable auquel on accède par l'évaluation de l'expression *Expr*, à la façon d'une *L-value* dans les langages de programmation classiques.

La valeur boolénne V^k n'est à prendre en compte que si la valeur de la source et la valeur de la destination sont respectivement de la forme :

- `<`E A_1 ... A_m`>` *Content* `<`E`/>` et
- `<`E' A'_1 ... A'_n`>` *Content*$'$ `<`E'`/>`.

où E désigne un élément XML Schema, A un attribut XML Schema et *Content*, le contenu d'un élément XML Schema, tels que définis à la section 7.1. Si V^k vaut "`vrai`", alors :

- la destination devient `<`E A_1 ... A_m`>` *Content* `<`E`/>`,

- sinon <E' A_1 ... A_m> *Content* <E'/>.

La valeur booléenne V^i n'est à prendre en compte que lorsque la sélection de la source n'aboutit pas (requête d'un attribut optionnel qui n'a pas été initialisé, par exemple). Dans ce cas, si la valeur de V^i est égale à "`vrai`", alors la copie n'a pas lieu et une exception est levée, sinon la copie est ignorée et aucune exception n'est levée.

10.1.25 Activité "validate"

L'activité atomique "`validate`" dont la syntaxe est la suivante :

Activity ::= ...
| <validate variables="X_0 ... X_n" />
| ...

permet de forcer explicitement le contrôle de la correspondance entre le type des variables $X_0, ..., X_n$ et leur valeur courante. En BPEL, à moins de faire appel à cette activité, ou bien de spécifier l'attribut "`validate`" lors d'une affectation (cf. section 10.1.24), cette correspondance n'est pas contrôlée.

10.1.26 Activité "sequence"

L'activité structurée "`sequence`" dont la syntaxe est la suivante :

Activity ::= ...
| <sequence>
Activity$_0$ ... *Activity*$_n$
</sequence>
| ...

désigne l'exécution séquentielle des activités *Activity*$_0$...*Activity*$_n$.

10.1.27 Activité "if"

L'activité structurée "`if`" dont la syntaxe est la suivante :

Activity ::= ...
| <if>
<condition> *Expr* </condition>
Activity$_1$
(<else>
Activity$_2$
</else>)?
</if>
| ...

désigne l'exécution conditionnelle de l'activité *Activity*$_1$ si le résultat de l'évaluation de l'expression *Expr* est vraie, de l'activité *Activity*$_2$ sinon. La clause "`else`" est optionnelle. Si cette clause est absente, alors la clause "`<else><empty/></else>`" est implicitement insérée.

10.1.28 Activité “while”

L'activité structurée “`while`” dont la syntaxe est la suivante :

Activity ::= ...
| `<while>`
  `<condition>` *Expr* `</condition>`
  Activity
  `</while>`
| ...

désigne l'exécution répétée de l'activité *Activity* aussi longtemps que le résultat de l'évaluation de *Expr* est vrai.

10.1.29 Activité “repeatUntil”

L'activité structurée “`repeatUntil`” dont la syntaxe est la suivante :

Activity ::= ...
| `<repeatUntil>`
  Activity
  `<condition>` *Expr* `</condition>`
  `</repeatUntil>`
| ...

désigne l'exécution répétée de l'activité *Activity*, qui est exécutée au moins une fois et jusqu'à ce que le résultat de l'évaluation de *Expr* devienne “`faux`”.

10.1.30 Activité “forEach”

L'activité structurée “`forEach`” dont la syntaxe est la suivante :

Activity ::= ...
| `<forEach counterName="`X`" parallel="`V^p`"`
  `<startCounterValue>` $Expr_1$ `</startCounterValue>`
  `<finalCounterValue>` $Expr_2$ `</finalCounterValue>`
  (`<completionCondition>`
    `<branches successfulBranchesOnly="`V^b`"? >`
      $Expr_3$
    `</branches>`
   `</completionCondition>`)?
  Activity
  `</forEach>`
| ...

désigne l'exécution répétée de l'activité *Activity* (qui, ici, est forcément l'activité “`scope`”, cf. section 10.1.34). Entre chaque exécution, la valeur de la variable entière X, dont le résultat de l'évaluation de $Expr_1$ donne la valeur initiale, est incrémentée (le pas d'incrémentation est forcément égal à 1). La dernière exécution de l'activité survient lorsque la valeur de la variable X est égale au résultat de l'évaluation de $Expr_2$. La portée de la variable X se limite à l'activité *Activity*.

Si la clause "`completionCondition`", dont la syntaxe est inutilement compliquée, est présente, alors l'activité *Activity* est exécutée au plus $Expr_3$ fois. Le résultat de l'évaluation de $Expr_3$ doit toujours être inférieur ou égal au résultat de l'évaluation de $Expr_2 - Expr_1 + 1$. Lorsque l'attribut "`successfulBranchesOnly`" est spécifié, si sa valeur booléenne V^b vaut "`vrai`", alors l'activité "`forEach`" se termine après que l'activité *Activity* a été exécutée au plus $Expr_3$ avec succès (c'est-à-dire, sans lever d'exception). Si l'attribut "`successfulBranchesOnly`" n'est pas spécifié, alors sa valeur V^b vaut "`faux`".

Si la valeur V^p de l'attribut "`parallel`" de l'activité "`forEach`" vaut "`vrai`", alors au lieu d'effectuer n exécutions de l'activités *Activity* en séquence, on lance n exécutions de *Activity* en parallèle.

10.1.31 Activité "pick"

L'activité "`pick`", dont la syntaxe est la suivante :

Activity ::= ...
| `<pick createInstance="`V^c`"? >`
  PickEventHandler$_0$
  ...
  PickEventHandler$_n$
  `</pick>`
| ...

PickEventHandler ::= `<onMessage partnerLink="`λ`"`
  `portType="`ϕ`"?`
  `operation="`F`"`
  `messageType="`M`"`
  `variable="`X`"`
  `messageExchange="`M^e`"? />`
  Activity
  `</onMessage>`
| `<onAlarm>`
  `<for>`*Expr*`</for>`
  Activity
  `</onAlarm>`
| `<onAlarm>`
  `<until>`*Expr*`</until>`
  Activity
  `</onAlarm>`

déclare les gestionnaires d'événements *PickEventHandler*$_0$, ..., *PickEventHandler*$_n$. Ces gestionnaires d'événements sont très semblables aux gestionnaires d'événements des services (cf. section 10.1.14). Chaque gestionnaire d'événement associe une activité à un événement. Les événements "`onMessage`" sont identiques aux messages "`onEvent`" des gestionnaires d'événements des services. Les événements "`onAlarm`" sont définis comme une restriction syntaxique des événements "`onAlarm`" des gestionnaires d'événements des services (pas de clause "`repeatEvery`" et présence obligatoire de la clause "`for`" ou de la clause "`until`"). Une activité "`pick`" doit comprendre au moins un gestionnaire d'événement avec un 'événement de type "`onMessage`".

Une activité "`pick`" attend que le premier événement parmi ceux définis dans ses gestionnaires d'événements se produise puis exécute l'activité qui lui est associée. Comme son nom l'indique,

une activité "`pick`" n'exécute, au final, qu'un seul de ces gestionnaires d'événements, celui dont l'événement se produit le premier. La fin de l'exécution de l'activité "`pick`" correspond à la fin de l'exécution de l'activité du gestionnaire d'événement *choisi*.

Un événement "`onMessage`" se produit lorsque le message qu'il attend est reçu par le service. Un événement "`onAlarm`" se produit lorsque l'alarme qu'il définit est atteinte. Si l'événement "`onAlarm`" utilise une construction "`for`", alors l'événement se produit lorsque la durée définie par *Expr* est écoulée. Si l'événement "`onAlarm`" utilise une construction "`until`", alors l'événement se produit lorsque la date définie par *Expr* est atteinte.

Lorsque l'attribut "`createInstance`" est absent, sa valeur booléenne V^c vaut "`faux`". Lorsque V^c vaut "`vrai`", l'interprète BPEL doit vérifier statiquement que tous les événements des gestionnaires d'événements de l'activité "`pick`" sont de type "`onMessage`" et que l'activité "`pick`" est la première activité de réception de message du service principal. Si ces deux conditions sont satisfaites, alors le service principal se réplique, à chaque fois qu'un événement se produit, pour traiter l'activité associée.

10.1.32 Activité "flow" et liens de contrôle

L'activité "`flow`" est certainement la plus complexe de toutes. Sa syntaxe est la suivante :

Activity ::= ...
| `<flow>`
  `<links>`
  `<link name="`L_1`" />`
  ...
  `<link name="`L_m`" />`
  `</links>`
  Activity$_0$
  ...
  Activity$_n$
  `</flow>`
| ...

Elle introduit du parallélisme asynchrone entre activités, qui peut être contraint par l'utilisation d'un mécanisme de liens de contrôle dont le but est de fixer l'ordre d'exécution de certaines activités parallèles, en retardant ou en bloquant leur démarrage. Ces activités parallèles partagent les variables déclarées par le service et les sous-services (cf. section 10.1.34) contenant l'activité "`flow`".

Les activités *Activity*$_0$ ... *Activity*$_n$ sont exécutées en parallèle et peuvent être liées les unes aux autres grâce aux liens de contrôle L_1 ... L_m. A chaque lien sont associées une activité source et une activité cible. Les restrictions syntaxiques suivantes s'appliquent :

- Chaque lien de contrôle doit avoir exactement une activité source et une activité cible.
- Deux liens de contrôle différents ne peuvent avoir à la fois la même activité source et la même activité cible.
- Un lien de contrôle ne doit pas créer de dépendance cyclique entre activités.
- Un lien de contrôle qui a pour cible une activité contenue dans un gestionnaire d'exception ou de terminaison (cf. section 10.1.34) doit avoir pour source une activité contenue dans le même gestionnaire d'exception ou de terminaison.

- Une activité itérative ("`while`", "`repeatUntil`" et "`forEach`"), un gestionnaire d'événement ou un gestionnaire de compensation ne peut pas utiliser de liens de contrôle définis dans une activité "`flow`" englobante.

Pour déclarer une activité a comme source des liens de contrôle $L_0, ..., L_n$, il suffit d'ajouter la déclaration suivante à la définition de l'activité (cette déclaration doit figurer entre "<*a*>" et "</*a*>") :

```
<sources>
  <source linkName="L0" >
    <transitionCondition>Expr0</transitionCondition>?
  </source>
  ...
  <source linkName="Ln" >
    <transitionCondition>Exprn</transitionCondition>?
  </source>
</sources>
```

où $Expr_0, ..., Expr_n$ sont des expressions booléennes appelées "conditions de transition" dans lesquelles peuvent apparaître les variables déclarées par le service et les sous-services contenant l'activité "`flow`". Si pour un lien L_i, $Expr_i$ n'est pas explicitement spécifiée, alors la norme BPEL considère que $Expr_i$ est définie implicitement comme étant la constante XPATH "`true`". A chaque lien est donc associée exactement une condition de transition dans son activité source.

Pour déclarer une activité a comme cible des liens de contrôle $L_0, ..., L_p$, il suffit d'ajouter la déclaration suivante à la définition de l'activité (cette déclaration doit figurer entre "<*a*>" et "</*a*>") :

```
<targets>
  <joinCondition>Expr</joinCondition>?
  <target linkName="L0" />
  ...
  <target linkName="Lp" />
</targets>
```

où *Expr* est une expression booléenne appelée "condition de jonction". *Expr* est formée uniquement des opérateurs booléens de négation, de conjonction et de disjonction, et des variables $L_0, ..., L_p$ contenant les valeurs des liens. Si la condition de jonction est omise, alors la norme BPEL considère qu'elle est égale à la disjonction des valeurs des liens $L_0, ..., L_p$, c'est-à-dire l'expression XPATH :

$$L_0 \texttt{ or } ... \texttt{ or } L_p$$

Un lien de contrôle peut prendre trois valeurs possibles : "`vrai`", "`faux`" et $\perp$. $\perp$ désigne la valeur indéfinie qui est initialement associée à tous les liens de contrôle. Lorsque l'exécution de l'activité source des liens $L_0, ..., L_n$ prend fin, les conditions de transition des liens de contrôle $L_0, ..., L_n$ sont évaluées et le résultat de l'évaluation de chaque $Expr_i$ est affecté comme valeur du lien de contrôle L_i correspondant.

L'exécution de l'activité cible des liens $L_0, ..., L_p$ ne peut démarrer que si les valeurs des liens $L_0, ..., L_p$ sont différentes de $\perp$. Ensuite, la condition de jonction de cette activité est évaluée. Si le résultat de cette évaluation est vrai, alors l'activité est exécutée. Sinon l'activité n'est pas exécutée et il faut distinguer les deux cas suivants :

- Si l'activité se trouve dans la portée d'une définition de l'attribut "`suppressJoinFailure`" (cf. section 10.1.6) dont la valeur V^s est égale à "`vrai`", alors la valeur "`faux`" est affectée à tous les liens dont cette activité est la source.

- Si l'activité se trouve dans la portée d'une définition de l'attribut "`suppressJoinFailure`" dont la valeur V^s est égale à "`faux`", alors l'exception `joinFailure` est levée et la valeur des liens, dont cette activité est la source, reste indéfinie ($\perp$).

Nous faisons à présent trois remarques importantes sur la sémantique non intuitive définie dans la norme Bpel [Com01, Sec. 11.6.2] pour la gestion des valeurs des liens de contrôles dans les activités non exécutées. Ce mécanisme est appelé *Dead Path Elimination*.

Première remarque : si une activité a déclenche une exception (autre que "`joinFailure`"), alors tous les liens dont elle est la source et qui ont pour cible une activité a' telle que a et a' sont englobées par le même service ou sous-service (cf. définitions de la section 10.1.3), sont laissés à $\perp$, tandis que la valeur "`faux`" est affectée aux autres liens dont elle est la source.

Seconde remarque : si une activité a est comprise dans une activité a' et que, dans l'exécution de a, a' ne sera pas exécutée, alors la valeur "`faux`" est affectée à tous les liens dont a est la source, lorsque les valeurs des liens, dont a' est la cible, sont différentes de $\perp$. Cette situation peut se produire dans les cas suivants :

- a' est une activité "`if`" et a est l'activité de la branche qui n'a pas été sélectionnée.
- a' est une activité "`pick`" et a est l'activité de l'un des gestionnaires d'événements qui n'est pas exécuté.
- a est l'activité d'un gestionnaire d'exception d'un service ou d'un sous-service (cf. section 10.1.34) qui n'est pas déclenché (soit parce qu'aucune exception n'est levée, soit parce que l'exception levée est traitée par un autre gestionnaire).
- a est l'activité d'un gestionnaire de terminaison (cf. section 10.1.35) qui n'est pas exécuté.

Troisième remarque : si une activité a est interrompue par la levée d'une exception dans une activité parallèle a', alors les liens, dont a est la source et qui ont pour cible une activité englobée par le sous-service ou service englobant l'activité "`flow`" dans laquelle a et a' sont comprises, sont laissés à $\perp$, tandis que la valeur "`faux`" est affectée aux autres liens dont a est la source.

10.1.33 Activité "extensionActivity"

La norme Bpel permet à un interprète d'ajouter de nouvelles activités au langage [Com01, Sec. 10.9]. Une telle activité a doit être utilisée au sein de l'activité "`extensionActivity`" :

Activity ::= ...
| `<extensionActivity>`
  a
  `</extensionActivity>`
| ...

Par exemple, la norme Bpel ne permet pas à un service d'accéder au système de fichiers du serveur qui l'héberge. Imaginons qu'un interprète Bpel munisse le langage d'une collection d'activités pour manipuler des fichiers ("`open`", "`close`", "`read`", "`write`", "`readLine`", ...). Ces activités sont regroupées au sein d'une extension à laquelle est associé un espace de noms que nous nommons "`file`". Afin d'utiliser ces nouvelles activités, un service Bpel doit tout d'abord activer l'extension dans l'en-tête de sa définition au moyen de la construction "`extensions`" :

```
<extensions>
  <extension namespace="file" />
</extensions>
```

Dans le corps du service, une occurrence de l'une des nouvelles activités ne peut apparaître qu'à l'intérieur de l'activité "`extensionActivity`" :

```
<sequence>
  <extensionActivity>
    <open filename="name.ext" mode="r" variable="f" />
  </extensionActivity>
  <extensionActivity>
    <readLine fileVariable="f" intoVariable="str" />
  </extensionActivity>
  ...
</sequence>
```

10.1.34 Activité "scope" et notion de sous-service

Un sous-service est une activité structurée au sein d'un service qui possède ses propres gestionnaires (exceptions, compensation, terminaison) et qui peut définir de nouveaux liens de communications, variables, échanges de messages et ensembles de corrélations. Il ne s'agit pas d'un service exécuté en parallèle du service en cours de définition, mais plutôt d'une unité logique permettant d'encapsuler une activité a afin de faciliter la gestion des erreurs se produisant dans a. La définition d'un sous-service ressemble beaucoup à la définition d'un service (cf. section 10.1.5) et se fait au moyen de l'activité "`scope`" dont la syntaxe est la suivante :

Activity ::= ...
| `<scope name="`S`" isolated="`V^i`" exitOnStandardFault="`V^e`" >`
  (`<variables>`
  Variable$_0$ ... *Variable*$_m$
  `</variables>`)?
  (`<partnerLinks>`
  PartnerLink$_0$ ... *PartnerLink*$_n$
  `</partnerLinks>`)?
  (`<messageExchanges>`
  MessageExchange$_0$ ... *MessageExchange*$_p$
  `</messageExchanges>`)?
  (`<correlationSets>`
  CorrelationSet$_0$ ... *CorrelationSet*$_r$
  `</correlationSets>`)?
  (`<eventHandlers>`
  EventHandler$_0$ ... *EventHandler*$_s$
  `</eventHandlers>`)?
  (`<faultHandlers>`
  FaultHandler$_0$ ... *FaultHandler*$_t$
  `</faultHandlers>`)?
  CompensationHandler?
  (`<terminationHandler>`
  Activity′
  `</terminationHandler>`)?
  Activity
  `</scope>`
| ...

La valeur booléenne V^i, qui n'est pas présente lors de la définition d'un service, indique, lorsqu'elle est égale à "`vrai`", que le sous-service est *isolé*. La norme BPEL indique [Com01, Sec. 12.8] qu'un sous-service isolé est exécuté de façon atomique pour éviter les accès concurrents par d'autres activités parallèles aux variables qu'il utilise. La norme est toutefois très peu explicite quant à la sémantique exacte des sous-services isolés. C'est peut être la raison pour laquelle ces sous-services ne sont jamais mentionnés dans les travaux de vérification formelle de services BPEL.

Un gestionnaire de terminaison est une activité qui peut être exécutée lorsque le sous-service est arrêté inopinément à cause de la levée d'une exception dans une activité parallèle.

Le gestionnaire de compensation est optionnel et son fonctionnement est décrit à la section 10.1.35.

Un sous-service est une activité comme une autre. Un service BPEL peut donc définir l'activité qu'il encapsule comme étant, par exemple, un sous-service, ou un "`flow`" de sous-services.

10.1.35 Gestionnaires de compensation

A chaque sous-service peut être associé un gestionnaire de compensation (*compensation handler*) dont la syntaxe de définition est la suivante :

CompensationHandler ::= `<compensationHandler>`
Activity
`</compensationHandler>`

La norme BPEL ne permet pas d'associer un gestionnaire de compensation à un service.

Si une définition de sous-service ne déclare pas explicitement de gestionnaire de compensation, alors le gestionnaire suivant est automatiquement inséré :

```
<compensationHandler>
  <compensate />
</compensationHandler>
```

L'activité spécifiée par un gestionnaire de compensation a pour but, dans la mesure du possible, d'annuler les changements effectués (communications et modifications de variables) par l'exécution d'un sous-service. Cette activité est exécutée lorsque le gestionnaire de compensation est invoqué. Cette construction a été étudiée en détail par Eisentraut et Spieler [ES09].

Un gestionnaire de compensation est dit *activé* si le sous-service qu'il compense a terminé son exécution sans lever d'exception. Dans un cas contraire, le gestionnaire de compensation d'un sous-service est dit *désactivé*. A chaque gestionnaire de compensation activé est associé un *état* qui contient une sauvegarde des valeurs des variables, des liens de communication et des ensembles de corrélations. Cette sauvegarde est faite au moment où le sous-service termine son exécution.

L'exécution d'un gestionnaire de compensation activité commence par restaurer les anciennes valeurs sauvegardées dans l'état qui lui est associé, avant d'exécuter l'activité que le gestionnaire définit. La norme BPEL conseille de définir cette activité comme un sous-service isolé afin d'éviter l'exécution parallèle d'autres activités qui pourraient modifier les variables.

L'invocation d'un gestionnaire de compensation ne peut se faire que depuis un autre gestionnaire de compensation, un gestionnaire de terminaison ou bien un gestionnaire d'exception. L'invocation d'un gestionnaire de compensation désactivé est équivalente à l'exécution de l'activité "`empty`". Il existe deux syntaxes pour invoquer un gestionnaire de compensation :

Activity ::= ...
| <compensateScope target="S" />
| <compensate>

"compensateScope" invoque le gestionnaire de compensation du sous-service S qui est englobé dans S', le service ou sous-service qui englobe l'activité "compensateScope".

"compensate" invoque le gestionnaire de compensation de chaque sous-service qui est englobé dans S', le service ou sous-service qui englobe l'activité "compensate". Ces invocations se font dans l'ordre défini par la norme Bpel [Com01, Sec. 12.5.2], c'est-à-dire l'ordre dans lequel les sous-services se sont terminés.

10.2 Traduction en LOTOS NT

Dans cette section, nous présentons un algorithme original de traduction de Bpel vers Lotos NT, dans le but de générer par la suite le Ste correspondant à un service Bpel. Ainsi, ce service Bpel pourra être formellement vérifié, par exemple en évaluant des formules de logique temporelle sur le Ste généré.

Comme nous ne traitons pas la totalité du langage Bpel, nous commençons par expliquer pourquoi certaines constructions ont été omises. Ensuite, nous présentons une grammaire abstraite pour Bpel avant de détailler notre algorithme de traduction, construction par construction, en le comparant, pour les constructions les plus délicates à traduire (telles que les liens de contrôle ou la gestion des exceptions), aux traductions proposées par d'autres auteurs.

10.2.1 Restrictions

(a) Nous ne traduisons que les spécifications Bpel correctement typées, afin de produire une spécification Lotos NT correctement qui sera acceptée par le compilateur Lnt2Lotos. Nous rejetons donc à la traduction toute spécification Bpel mal typée. Cela implique que les attributs "keepSrcElementName" et "ignoreMissingFromData" dans l'activité d'affectation "assign" sont ignorés car ils relèvent du fait que les valeurs en Bpel sont non typées. De plus, dans notre base d'exemples, ces deux attributs n'apparaissent que dans trois fichiers.

(b) En ce qui les communications, nous ne traitons pas la mobilité des services Bpel. Il s'agit d'une fonctionnalité, héritée de la théorie du π-calcul [Mil99] qui consiste, pour un service, à manipuler les canaux de communication comme des objets de premier ordre, c'est-à-dire que les canaux de communication (*liens* en Bpel) sont des valeurs comme les autres qui peuvent être modifiées et échangées entre services partenaires. Concrètement, cela se traduit par la possibilité de recevoir, sur un lien de communication, l'adresse d'un nouveau service partenaire puis d'utiliser cette adresse pour modifier un lien existant ou créer un nouveau lien.

En Lotos et en Lotos NT, il n'est pas permis de modifier ou créer dynamiquement un nouveau lien de communication pendant l'exécution d'un processus, ce qui rend difficile la traduction de la mobilité des services Bpel. Difficile ne veut toutefois pas dire impossible. Par exemple, Mateescu et Salaun [MG10], lorsqu'ils traduisent le π-calcul [Mil99] en Lotos NT, représentent les noms du π-calcul (c'est-à-dire les canaux de communication) au moyen d'un type Lotos NT "Chan". Chaque nouvelle création de nom entraîne l'instantiation d'une nouvelle variable de type "Chan", avec un identificateur unique, qui est alors propagée, par une porte de communication spéciale, vers d'autres processus pour leur signaler la création de ce nouveau nom. L'extension d'une algèbre de processus comme Csp [BHR84] avec un opérateur dédié au recensement des noms du π-calcul est

aussi envisageable, comme l'a montré Roscoe [Ros08].

En pratique, les deux seuls services Bpel de notre base d'exemples qui utilisent des constructions relatives à la mobilité (à savoir, l'affectation d'une valeur à un lien de communication) sont tirés de la base de tests du projet Orchestra[55] et ont pour seule vocation de tester cette fonctionnalité. Nous avons donc choisi de ne traiter aucune construction relative à la mobilité des services Bpel. Concrètement :

- l'attribut "`initializePartnerRole`" des déclarations de liens de communication (cf. section 10.1.10) est ignoré et
- les clauses "`from`" et "`to`" de l'activité "`assign`" (cf. section 10.1.24) ne peuvent contenir des références à des liens de communication ou bien des adresses de services partenaires.

(c) Les échanges de message (décrits à la section 10.1.11) ne sont pas pris en compte. Ces constructions permettent de s'assurer qu'à une réception de message correspond un unique envoi de réponse. Nous considérons qu'il est plus aisé de vérifier cette liaison à l'aide de formules de logique temporelle, après avoir généré l'espace d'états du service Bpel, plutôt que de la vérifier statiquement, lors de la traduction de Bpel vers Lotos NT. En outre, le seul exemple de notre base qui fait usage des échanges de message est un test du projet Orchestra.

(d) Dans un service, la réception du premier message peut contenir un attribut "`createInstance`" de valeur "`vrai`" dont le but est d'améliorer les performances du service en créant une nouvelle instance, à chaque réception de ce message, ce qui permet de traiter plusieurs requêtes simultanément. Cette construction de Bpel, qui s'apparente à la *réplication* du π-calcul, n'est pas adaptée à la vérification formelle par génération d'espaces d'états car elle entraîne des espaces d'états infinis. C'est pourquoi toutes les approches de vérification de services Bpel, la nôtre y compris, ignorent cet attribut et ne considèrent que les services possédant un unique fil d'exécution.

(e) Par conséquent, nous ne traitons pas non plus les ensembles de corrélations, dont l'existence n'est justifiée que par la présence simultanée de plusieurs instances d'un même service.

(f) L'attribut "`parallel`" de l'activité "`forEach`" indique si les différentes itérations de la boucle doivent être exécutées en parallèle. Dans le cas général, le nombre d'itérations est déterminé dynamiquement, ce qui ne permet pas une traduction simple vers une algèbre de processus comme Lotos NT, dans laquelle le nombre de comportements parallèles doit être déterminé statiquement. Lohmann [Loh08] aborde ce problème en donnant une traduction de l'activité "`forEach`" parallèle dans le cas où le nombre d'itérations peut être statiquement déterminé. Parmi les 11 exemples de notre base qui utilisent l'activité "`forEach`" parallèle, 10 sont des tests des projets Orchestra et ActiveBpel qui ont été écrits dans le seul but de tester cette activité ; au contraire, le dernier exemple est un service réel dans lequel le nombre d'itérations de l'activité "`forEach`" est déterminé dynamiquement.

La présence d'un unique exemple de cette fonctionnalité dans notre base nous a décidé à ne pas traiter la version parallèle de l'activité "`forEach`" dans notre traduction, d'autant plus que les mêmes obstacles qu'au point (d) se dressent.

(g) Au niveau des exceptions, nous ne traitons qu'une seule exception prédéfinie : "`joinFailure`". Les autres exceptions prédéfinies relèvent de constructions Bpel que nous avons choisies de ne pas traiter (les exceptions "`missingReply`", "`missingRequest`", "`conflictingReceive`", "`conflictingRequest`", "`correlationViolation`", par exemple), ou sont levées en fonction de contraintes statiques (les exceptions "`invalidExpressionValue`" et "`uninitializedVariable`" par exemple). Seule l'exception "`invalidVariables`" entraîne parfois la levée d'une erreur dans la compilation du code Lotos NT, lorsqu'une conversion d'un type de base vers un type simple défini par

[55] `http://orchestra.ow2.org`

restriction ne réussit pas. Dans ce cas, nous avons décidé de ne pas traiter cette exception comme une exception Bpel qui peut être capturée par un sous-service, mais plutôt comme une erreur de compilation, afin d'être cohérent avec le point (a) où nous indiquions que seules les spécifications Bpel bien typées étaient acceptées.

(h) L'attribut "`exitOnStandardFault`" d'une déclaration de service (cf. section 10.1.5) est ignoré car les deux seuls exemples de notre base qui l'utilisent déclarent un service qui ne lève aucune exception prédéfinie. Si la condition de démarrage d'une activité est évaluée à "`faux`", alors cet attribut spécifie si l'exécution du sous-service englobant cette activité continue ou si l'exception "`joinFailure`" est levée.

(i) Nous ne traduisons pas la levée d'exceptions par l'activité "`throw`" lorsque l'exception n'est pas explicitement nommée (cf. section 10.1.22). Dans notre base d'exemples, toutes les exceptions levées sont nommés. Nous ne traitons donc pas les exceptions anonymes. Nous ne traitons pas non plus la levée d'exceptions non déclarées, c'est-à-dire celles qui n'ont pas été explicitement déclarées en Wsdl.

(j) En ce qui concerne les constructions liées à la gestion des exceptions, nous ne traitons ni les gestionnaires de terminaison, ni les gestionnaires de compensation. En effet,

- les gestionnaires de terminaison, qui sont une nouveauté de la version 2.0 de la norme Bpel, ne semblent pas être réellement utilisés en pratique ; tous les exemples de notre base qui les utilisent sont des tests dont le seul but est de tester cette construction. Parmi les différentes approches de vérification de services Web Bpel, seules celles reposant sur des traductions vers des formalismes de réseaux de Petri [Loh08, OVvdA+07] ont choisi de traiter cette construction. La difficulté du traitement des gestionnaires de terminaison réside dans la détection de l'arrêt forcé d'un sous-service puis dans l'exécution de son gestionnaire de terminaison qui peut accéder aux variables déclarées par le sous-service. Le premier point est facile à traiter dans les formalismes de réseaux de Petri car il suffit de tracer des transitions depuis l'ensemble de places symbolisant l'exécution du sous-service vers la place qui symbolise le début de l'exécution du gestionnaire de terminaison. Le second point, le plus difficile à notre avis, est ignoré par ces approches car elles ne considèrent pas les valeurs des variables.

- la traduction des gestionnaires de compensation nécessite de sauvegarder, après l'exécution d'un sous-service, les valeurs finales de chaque variable visible dans ce sous-service (afin d'être en mesure de les restaurer le cas échéant, si une compensation doit avoir lieu). Traiter cette construction contribuerait donc fortement au phénomène de l'explosion de l'espace d'états. C'est pourquoi cette construction n'est traitée correctement que par les approches par traduction vers des formalismes de réseaux de Petri [Loh08, OVvdA+07] (dans lesquelles les données ne sont pas traitées). Cependant, il y a peu de raisons qui justifient la prise en compte du mécanisme de compensation alors que les données ne sont pas prises en compte. En effet, un gestionnaire de compensation continue l'exécution d'un sous-service pour tenter d'annuler les modifications faites par ce sous-service. Avant d'effectuer cette continuation, un gestionnaire de compensation restaure les variables aux valeurs qui étaient les leurs au moment où le sous-service s'est terminé. L'intérêt de ce mécanisme réside donc en grande partie dans la sauvegarde des valeurs des variables à la fin de l'exécution d'un sous-service. Bianculli et al. [BGS07] ont tenté de traduire cette construction dans une algèbre de processus, mais ne précisent pas comment sont traduits les appels aux gestionnaires de compensation ; de plus, ils ont fait une erreur en sauvegardant les valeurs des variables au début de l'exécution d'un sous-processus, et non pas à la fin, comme le veut la norme.

(k) Bien évidemment, nous ne pouvons nous permettre de traiter les extensions du langage Bpel. En effet, la sémantique d'une extension n'est pas définie par la norme Bpel et dépend de l'implémentation

de chaque interprète Bpel qui comprend cette extension. La construction "`extensions`" est donc ignorée.

10.2.2 Syntaxe abstraite du sous-ensemble de BPEL considéré

Nous présentons à la table 10.1 la grammaire de Bpel sur laquelle nous définissons notre algorithme de traduction vers Lotos NT.

Cette grammaire ne modélise pas la construction "`import`". Les définitions Xml Schema et Wsdl importées par le service ont été, au préalable, traduites en Lotos NT. Les identificateurs qu'elles déclarent ont été liés et peuvent apparaître, en Bpel, dans les définitions de variables, les définitions de liens de communication et certaines activités ("`receive`" ou "`reply`", par exemple).

Les événements "`onEvent`" des gestionnaires d'événements (cf. section 10.1.14) et les événements "`onMessage`" de l'activité "`pick`" (cf. section 10.1.31) ont la même syntaxe et ont été fusionnés en un unique terme de la grammaire (**OnEvent** $(\lambda, F, M, X, \mathit{Activity})$).

Les notations utilisées dans cette grammaire sont les suivantes :

- S est un identificateur de service,
- λ est un identificateur de lien de communication,
- θ est un identificateur de type de lien de communication,
- X est un identificateur de variable,
- T est un identificateur de type (type Xml Schema, élément Xml Schema, ou message Wsdl),
- F est un identificateur de fonction,
- M est un identificateur de message, et
- χ est un identificateur d'exception.

Le non-terminal *Expr* a été défini à la section 8.3 tandis que le non-terminal *Literal* a été défini à la section 7.5.

Scope (*Def*) désigne une définition de sous-service. La norme Bpel considérant les sous-services comme des activités, il est légitime que **Scope** (*Def*) figure parmi les productions du non-terminal *Activity* dans notre grammaire.

10.2.3 Paramètre de la fonction de traduction

La fonction de traduction que nous définissons dans cette section propage un paramètre booléen nommé f. Ce paramètre indique si la construction à traduire se trouve ou non à l'intérieur d'une activité "`flow`". En effet, certaines activités ont une traduction qui diffère en fonction de la valeur de f, comme nous le verrons à la section 10.2.26.

Initialement, pour la traduction du service principal, la valeur de f est fausse.

10.2.4 Traduction des liens de communications

Un service Bpel (non-terminal *Def*) est traduit par un processus Lotos NT sans paramètres. Chaque déclaration de lien de communication λ avec un type de lien θ résulte en une porte Lotos NT

Def	::=	**Service** $(S, \lambda_1 : \theta_1 ... \lambda_{n^p} : \theta_{n^p},$ $X_1 : T_1 ... X_{n^x} : T_{n^x},$ $FaultHandler_0 ... FaultHandler_{n^f},$ $EventHandler_1 ... EventHandler_{n^e}, Activity)$
FaultHandler	::=	**CatchFault** $(\chi, Activity)$
	|	**CatchFaultData** $(\chi, X : M, Activity)$
	|	**CatchAll** $(Activity)$
EventHandler	::=	**OnEvent** $(\lambda, F, M, X, Activity)$
	|	**OnAlarmFor** $(Expr, Activity)$
	|	**OnAlarmForRepeat** $(Expr_1, Expr_2, Activity)$
	|	**OnAlarmUntil** $(Expr, Activity)$
	|	**OnAlarmUntilRepeat** $(Expr_1, Expr_2, Activity)$
	|	**OnAlarmRepeat** $(Expr, Activity)$
Activity	::=	**Scope** (Def)
	|	**Empty**
	|	**Exit**
	|	**WaitFor** $(Expr)$
	|	**WaitUntil** $(Expr)$
	|	**Receive** (λ, F, X)
	|	**Reply** (λ, F, X)
	|	**ReplyFault** (λ, F, χ)
	|	**ReplyFaultData** (λ, F, χ, X)
	|	**InvokeOneWay** (λ, F, X_I)
	|	**InvokeTwoWays** $(\lambda, F, X_I, X_O, FaultHandler_1 ... FaultHandler_n)$
	|	**Assign** $(Var_0 := From_0, ..., Var_n := From_n)$
	|	**Throw** (χ)
	|	**ThrowData** (χ, X)
	|	**Rethrow**
	|	**Validate** $(X_0 ... X_n)$
	|	**Sequence** $(Activity_0 ... Activity_n)$
	|	**If** $(Expr, Activity_1, Activity_2)$
	|	**While** $(Expr, Activity)$
	|	**RepeatUntil** $(Expr, Activity)$
	|	**ForEach** $(X, Expr_1, Expr_2, Activity)$
	|	**ForEachCond** $(X, Expr_1, Expr_2, Expr_3, V^b, Activity)$
	|	**Pick** $(EventHandler_0 ... EventHandler_n)$
	|	**Flow** $(L_1 ... L_m, Activity_0 ... Activity_n)$
	|	**LinkedActivity** $(L_1 ... L_m, Expr, Activity, L'_1, Expr_1 ... L'_n, Expr_n)$
From	::=	*Expr*
	|	*Literal*

Table 10.1: Grammaire abstraite du langage BPEL

λ dont le canal est θ, le canal LOTOS NT défini lors de la traduction du type de lien de communication θ. Les informations relatives aux rôles ("`myRole`" et "`partnerRole`") des services sur un lien de communication ont déjà été encodées dans la traduction du type de ce lien (cf. chapitre 9) et ne sont pas explicitement mentionnées ici.

Nous désignons la traduction d'une déclaration de service *Def* par $[\![Def]\!]\ (f)$. Selon que *Def* est la déclaration du service principal ou bien une définition de sous-services, nous distinguons deux traductions différentes:

- S'il s'agit de la déclaration du service principal du fichier BPEL, la traduction de *Def*, qui contient au moins un lien de communication, est la suivante :

  $[\![Def]\!]\ (f)\ =$
  `process` S `[`λ_1`:`θ_1`,...,`λ_{n^p}`:`θ_{n^p}`] is`
  $\quad [\![Def]\!]^A\ (f)$
  `end process`

  où $[\![Def]\!]^A\ (f)$ désigne la seconde partie de la traduction des déclaration de service, que nous définissons à la section suivante.

- S'il s'agit de la déclaration d'un sous-service, alors il faut déclarer les éventuels liens au moyen de l'opérateur "`hide`", qui est le seul opérateur de LOTOS NT permettant la définition de nouvelles portes de communication à l'intérieur d'un processus (LOTOS NT ne permet pas la définition de processus imbriqués). La traduction de *Def* est alors la suivante :

  $[\![Def]\!]\ (f)\ =$
  `hide` λ_1`:`θ_1`,...,`λ_{n^p}`:`θ_{n^p} `in`
  $\quad [\![Def]\!]^A\ (f)$
  `end hide`

  Par ce schéma de traduction, les liens de communication $\lambda'_1 ... \lambda'_m$ déclarés par le sous-service n'engendreront aucune action observable dans le système de transitions final. Un schéma de traduction différent, qui rendrait observables les communications sur les liens définis par les sous-services, consiste à déclarer pour chaque $\lambda'_i, i \in [1...m]$ une porte correspondante dans la déclaration du processus LOTOS NT. Le traducteur offrira à l'utilisateur de le choix entre ces deux schémas de traduction.

Si le sous-service ne déclare aucun nouveau lien de communication, alors $[\![Def]\!]\ (f) = [\![Def]\!]^A\ (f)$.

10.2.5 Préservation de l'atomicité des activités atomiques

Afin de préserver l'atomicité des activités "atomiques" de BPEL, nous introduisons deux portes "`Lock`" et "`Release`" qui définissent un verrou que les activités atomiques doivent acquérir afin de s'exécuter de manière atomique. Un comportement, s'exécutant en parallèle de la traduction des activités contenues dans le service BPEL, gère l'ordre des synchronisations sur ces portes, en forçant une alternance entre synchronisations sur la porte "`Lock`" et synchronisations sur la porte "`Release`". Au début (*resp.* à la fin) de la traduction de chaque activité atomique de BPEL nous effectuons une synchronisation sur la porte "`Lock`" (*resp.* "`Release`"), ce qui permet d'éviter que ne s'entrelacent les exécutions de deux comportements LOTOS NT correspondants à deux activités atomiques de BPEL. Il s'agit du concept classique de moniteur.

En BPEL, une activité atomique ne peut être interrompue par la levée d'une exception. En LOTOS NT, cela ne peut être garanti comme nous le verrons à la section 10.2.7. Dans ce langage donc, l'exécution d'une activité atomique BPEL peut être interrompue[56] entre le moment où elle acquiert le verrou (synchronisation sur la porte "`Lock`") et le moment où elle le libère (synchronisation sur la porte "`Release`"). Le verrou s'en trouverait irrémédiablement bloqué car la synchronisation sur la porte "`Release`" n'aurait alors jamais lieu. Afin d'éviter ce cas de figure, nous associons à chaque activité atomique BPEL un numéro unique au moyen de la fonction :

$$[\![Activity]\!]^{atomic}$$

Ce numéro unique est communiqué au gestionnaire d'atomicité par la porte "`Lock`" lors de la prise du verrou. Le gestionnaire d'atomicité ne libère ensuite le verrou que si un numéro identique lui est envoyé sur la porte "`Release`". Comme nous le verrons à la section 10.2.7, cela permet à un gestionnaire d'exceptions de tenter de libérer le verrou de chaque sous-activité atomique qu'il englobe.

Nous désignons par $[\![Def\,]\!]^A\ (f)$ le code LOTOS NT correspondant au mécanisme de préservation de l'atomicité des activités atomiques au sein de services ou sous-services définis par *Def*. La définition de $[\![Def\,]\!]^A\ (f)$ est la suivante :

$[\![Def\,]\!]^A\ (f)\ =$

```
  loop
    hide Lock,Release:NatChannel in
      par Lock,Release in
        var N:Nat in
          Lock (?N);
          loop Atomic in
            var M:Nat in
              Release (?M);
              if (M == N) then
                break Atomic
              end if
            end var
          end loop
        end var
      ||
        [[Def ]]^X (f)
      end par
    end hide
  end loop
```

où $[\![Def\,]\!]^X\ (f)$ désigne la traduction des déclarations de variables du service que nous présentons à la section suivante.

Ce mécanisme est global, c'est-à-dire qu'il n'est pas nécessaire de le redéfinir pour un sous-service ; autrement dit, lorsque *Def* désigne une définition de sous-service, on a $[\![Def\,]\!]^A\ (f) = [\![Def\,]\!]^X\ (f)$.

Il est important de noter que la traduction du reste du processus est insérée dans une boucle ("`loop`") afin de préserver la sémantique de BPEL qui veut que l'exécution d'un service se répète indéfiniment (afin de servir une nouvelle requête immédiatement après en avoir traité une). Si cette boucle était placée à la racine du processus LOTOS NT (avant la déclaration des portes "`Lock`" et "`Release`"), alors il faudrait faire en sorte que la boucle de contrôle des portes "`Lock`" et "`Release`" se termine (en

[56] Nous souhaitons remercier chaleureusement Charles Pecheur pour avoir remarqué ce cas d'erreur dans une version préliminaire du manuscrit

effectuant une synchronisation entre la boucle contenant "`Lock`" et "`Release`" et la fin de l'exécution du service) avant chaque redémarrage du service BPEL. Une telle solution ajouterait des transitions invisibles inutiles dans l'espace d'états final.

10.2.6 Gestion des variables partagées

Une variable partagée est une variable à laquelle au moins deux activités parallèles essaient d'accéder (en lecture ou en écriture).

A notre connaissance, aucune approche pour la vérification formelle de services Web BPEL ne mentionne le problème de la traduction de l'accès aux variables partagées.

$[\![Def]\!]^X\ (f)$ désigne la portion de la traduction des définitions de services ou de sous-services BPEL qui traite des déclarations de variables. Nous trions les variables $X_1...X_{n^x}$ pour obtenir une nouvelle liste $X'_1...X'_{m^x}, X'_{m^x+1}...X'_{n^x}$ où $m < n$, de telle sorte que la liste $X'_1...X'_{m^x}$ représente les variables partagées tandis que la liste $X'_{m^x+1}...X'_{n^x}$ représente les variables non partagées.

La traduction des variables non partagées est directe. A chaque déclaration de variable non partagée en BPEL correspond une déclaration de variable en LOTOS NT. Si une telle variable est utilisée avant d'avoir été initialisée, le compilateur LOTOS NT refusera de compiler le processus et affichera une erreur.

Pour chaque variable partagée X de type T, nous définissons un type construit LOTOS NT "`Shared_`T" qui a deux constructeurs : "`undef`", qui désigne la valeur indéfinie et "`init`", dont l'argument contient la valeur de la variable après qu'elle a été initialisée. La définition en LOTOS NT de ce type est la suivante :

```
type Shared_T is
  undef,
  init (v:T)
end type
```

Nous associons à ce type un canal de communication LOTOS NT "`Channel_`T" :

```
channel Channel_T is (T) end channel
```

Nous utilisons ces types et canaux pour construire un gestionnaire unique et central [Hoa74] (chaque sous-service définit son propre gestionnaire pour les nouvelles variables qu'il déclare) de variables partagées qui va contrôler l'accès aux variables partagées au moyen de deux portes associées à chaque variable partagée : une porte pour lire la valeur de la variable et une porte pour écrire une nouvelle valeur dans la variable. La lecture de la valeur d'une variable ne peut se faire qu'après son initialisation. Une tentative d'accès à une variable partagée non initialisée entraîne la levée d'une exception par le compilateur LOTOS NT. L'écriture de la valeur d'une variable entraîne son initialisation si elle ne l'était pas déjà. Afin que ce gestionnaire de variables ait une durée de vie égale au service ou sous-service qui le déclare, nous le définissons en parallèle de l'exécution du service ou du sous-service, au moyen d'une boucle ("`loop`"). Une porte spéciale nommée "`exitVariableManager_`S", où S est le nom du service ou du sous-service englobant, permet d'arrêter le gestionnaire de variables lorsque l'exécution de S doit redémarrer.

Au final, la traduction des déclarations de variables $X'_1{:}T'_1...X'_{n^x}{:}T'_{n^x}$ est la suivante:

```
⟦Def⟧^X (f) =
  var X'_{m^x+1}:T'_{m^x+1},...,X'_{n^x}:T'_{n^x} in
    hide Read_X'_1,Write_X'_1:Channel_T'_1,...,Read_X'_{m^x},Write_X'_{m^x}:Channel_T'_{m^x},
         exitVariableManager_S:none in
      par Read_X'_1,Write_X'_1,...,Read_X'_{m^x},Write_X'_{m^x} in
        var X'_1:Shared_T'_1,...,X'_{m^x}:Shared_T'_{m^x} in
          X'_1 := undef;...;X'_{m^x} := undef;
          loop variableManager_S in
            select
              Read_X'_1 (!X'_1.v)
            []
              var Tmp_X'_1:T'_1 in
                Write_X'_1 (?Tmp_X'_1);
                X'_1 := init (Tmp_X'_1)
              end var
            [] ... []
              if (X'_{m^x} != undef) then
                Read_X'_{m^x} (!X'_{m^x}.v)
              end if
            []
              var Tmp_X'_{m^x}:T'_{m^x} in
                Write_X'_{m^x} (?Tmp_X'_{m^x});
                X'_{m^x} := init (Tmp_X'_{m^x})
              end var
            []
              exitVariableManager_S; break variableManager_S
            end select
          end loop
        end var
      ||
        ⟦Def⟧^F (f); exitVariableManager_S
      end par
    end hide
  end var
```

où $[\![Def]\!]^F\ (f)$ désigne la traduction des gestionnaires d'exceptions que nous présentons à la section suivante.

Si le service ou sous-service ne déclare aucune variable non partagée, alors il suffit de supprimer la portion de code LOTOS NT déclarant les variables $X'_{m^x+1}...X'_{n^x}$. Si le service ou sous-service ne déclare aucune variable partagée, alors il suffit de remplacer la portion de code LOTOS NT se trouvant entre "`hide`" et "`end hide`" par "$[\![Def]\!]^F\ (f)$".

10.2.7 Gestion des exceptions

Nous détaillons ici le cas des services et sous-services qui ne sont pas définis au sein d'une activité "`flow`" (valeur de l'argument f égale à "`faux`"). La traduction des gestionnaires d'exceptions d'un sous-service déclaré au sein d'une activité "`flow`" est détaillée à la section 10.2.26.

Selon Lohmann et al. [LVO+07], il existe deux façons d'interpréter la norme BPEL au sujet de la

gestion des exceptions. La première consiste à arrêter toutes les activités d'un service aussitôt qu'une de ces activités a levé une exception. La seconde consiste à arrêter toutes les activités d'un service, à la suite de la levée d'une exception, mais sans imposer de contrainte de temps sur la rapidité avec laquelle cet arrêt est effectué.

Les approches suivantes implémentent (au moins partiellement) la première interprétation :

- Ouyang et al. [OVvdA+07] capturent les exceptions, sans les données qui peuvent leur être éventuellement associées, puis exécutent les gestionnaires d'exceptions correspondants.
- Foster [Fos06] capture les exceptions et les données qui leur sont éventuellement associées (sans toutefois prendre en compte les types de données XML Schema), mais ne traite pas l'exécution des gestionnaires d'exceptions.
- Bianculli et al. [BGS07] traite les exceptions au moyen de la construction "`try`"/"`catch`" du langage BIR mais ne précise pas ce qu'il advient des données associées à l'exception.

A notre connaissance, la seule approche qui implémente la seconde interprétation est celle du groupe *Theory of Programming* de l'université Humbolt-Universität zu Berlin [HSS05, Loh08]. Les autres approches donnent peu de détails sur la façon dont sont traités les exceptions et leurs gestionnaires.

Nous avons du choisir la seconde interprétation car le mécanisme de gestion des exceptions de LOTOS NT n'est que partiellement implémenté dans le compilateur LNT2LOTOS à l'heure actuelle. Il permet de lever des exceptions, mais pas de les capturer. Pour cette raison, nous avons du simuler les gestionnaires d'exceptions à l'aide de portes LOTOS NT et de l'opérateur "`disrupt`" (aussi noté `[>` en LOTOS) qui peut interrompre, de manière non-déterministe, une activité au profit d'une autre. Cet aspect non-déterministe ne permet pas d'implémenter la première interprétation de la gestion des exceptions. En première approche, notre idée consiste à écrire :

`disrupt` A `by` F `end disrupt`

où A est la traduction de l'activité principale du service ou sous-service et F la traduction des gestionnaires d'exceptions. Dans cette première approche, F peut interrompre A à tout moment, ce qui n'est pas satisfaisant. Pour atteindre une relation de causalité entre la levée d'une exception dans A et l'exécution de l'un des gestionnaires d'exceptions de F, nous devons affiner notre approche en introduisant deux portes de synchronisation G et G' sur le modèle suivant :

```
hide G,G':none in
  par G,G' in
    disrupt
      A; G; stop
    by
      G';
      F
    end disrupt
  ||
    G;
    G'
  end par
end hide
```

Comme LOTOS NT ne permet pas d'effectuer directement une synchronisation entre un comportement de A et un comportement de F, nous avons recours à un comportement parallèle qui se synchronise avec la levée d'exception dans A, sur la porte G et qui déclenche l'exécution de F par une

synchronisation sur la porte G'. Ce mécanisme respecte scrupuleusement la sémantique des gestionnaires d'exceptions de BPEL qui précise que l'activité qui lève une exception doit être immédiatement interrompue (ici, ne pas confondre l'activité qui lève l'exception et les activités parallèles que nous ne pouvons interrompre immédiatement en LOTOS NT). Pour les activités exécutées en parallèle de l'activité levant l'exception, la norme BPEL précise seulement qu'elles peuvent être interrompues, ce qui est exactement ce qui se produit avec la traduction que nous proposons. Cette traduction des gestionnaires d'exceptions d'un service ou sous-service suit le schéma présenté ci-dessus, mais avec des portes typées pour transmettre les données associées aux exceptions. Le type de ces portes est défini ainsi :

```
channel Channel_Exceptions is (Exceptions) end channel
```

où le type "`Exceptions`" a été défini à la section 9.7.3 pour regrouper toutes les exceptions définies par l'utilisateur ainsi que les exceptions prédéfinies de BPEL que nous considérons.

Dans notre traduction, nous introduisons les portes "`faultHandle_`S" et "`faultRaise_`S" (en lieu et place des portes G et G'), de type "`Channel_Exception`", pour effectuer la synchronisation entre la levée des exceptions dans l'activité principale du service ou sous-service S et leur capture dans la traduction des gestionnaires d'exceptions. Lorsqu'une exception est reçue sur la porte de synchronisation "`faultHandle_`S", le gestionnaire d'exception correspondant est choisi à l'aide de l'opérateur LOTOS NT "`case`".

Nous introduisons la porte "`exitFaultRaise_`S" pour forcer la terminaison du comportement auxiliaire (G `;` G') si aucune exception n'a été levée.

Afin de simplifier la traduction, nous trions la liste des gestionnaires d'exceptions $FaultHandler_1 ... FaultHandler_{n^f}$ d'une définition de service, afin de la partitionner en trois :

- $FaultHandler_0 ... FaultHandler_{m^f}$ sont des gestionnaires d'exceptions qui capturent des exceptions seules, sans traiter leur données associées. Nous avons donc, $\forall i \in [0...m^f]$, $FaultHandler_i = CatchFault\ (\chi_i, Activity_i)$.

- $FaultHandler_{m^f+1} ... FaultHandler_{n^f-1}$ sont des gestionnaires d'exceptions qui capture des exceptions et leurs données associées. Nous avons donc, $\forall i \in [m^f + 1...n^f - 1]$, $FaultHandler_i = CatchFault\ (\chi_i, X_i : M_i, Activity_i)$.

- $FaultHandler_{n^f}$ est le gestionnaire d'exception *CatchAll* qui est toujours présent et qui correspond à la clause "`catchall`" de BPEL qui peut figurer librement parmi les gestionnaires d'exceptions. Nous avons donc $FaultHandler_{n^f} = CatchAll\ (Activity_{n^f})$. Nous plaçons *CatchAll* en dernière position afin de capturer tous les cas que les autres gestionnaires d'exceptions n'auraient pas traités.

Avant de traiter la traduction du gestionnaire d'exception "*CatchAll*", nous définissons la gestion de l'exception "`exitFault`". Il s'agit d'une exception spéciale que nous introduisons afin de traduire correctement la construction "`<exit />`" de BPEL (cf. section 10.2.10). L'idée est de propager cette exception sans la traiter jusqu'à ce qu'elle entraîne l'arrêt du service principal.

La traduction des gestionnaires d'exceptions d'une définition de service (non contenue dans un opérateur "`flow`") est la suivante :

```
⟦Def⟧^F (faux) =
  hide faultHandle_S,faultRaise_S:Channel_Exceptions,exitFaultRaise_S:any in
    par exitFaultRaise,faultHandle_S,faultRaise_S in
      disrupt
        ⟦Def⟧^E (faux); exitFaultRaise_S
      by
        Release (!i_1);
        ...
        Release (!i_p);
        var F:Exceptions in
          faultHandle_S (?F);
          case F in
          var X_{m^f+1}:M_{m^f+1},...,X_{n^f-1}:M_{n^f-1} in
            χ_0 -> ⟦Activity_0⟧ (faux)
          | ...
          | χ_{m^f} -> ⟦Activity_{m^f}⟧ (faux)
          | χ_{m^f+1} (X_{m^f+1}) -> ⟦Activity_{m^f+1}⟧ (faux)
          | ...
          | χ_{n^f-1} (X_{n^f-1}) -> ⟦Activity_{n^f-1}⟧ (faux)
          | exitFault -> ⟦exitFault⟧
          | any -> ⟦Activity_{n^f}⟧ (faux)
          end case
        end var
      end disrupt
    ||
      select
        exitFaultRaise
      []
        var F:Exceptions in
          faultRaise_S (?F); faultHandle_S (!F)
        end var
      end select
    end par
  end hide
```

où $[\![Def]\!]^E$ (faux) désigne la traduction des gestionnaires d'événements que nous étudierons à la section suivante. Si un service ne déclare aucun gestionnaire d'exception, alors $[\![Def]\!]^F(f) = [\![Def]\!]^E(F)$.

$i_1 ... i_p$ désignent les numéros uniques associés aux activités atomiques contenues dans le service ou sous-service en cours de traduction (cf. section 10.2.5).

$[\![Activity]\!]$ (faux) désigne la traduction de l'activité *Activity* qui n'est pas contenue dans un opérateur "flow".

La définition de la traduction des gestionnaires d'exceptions d'un sous-service contenu dans un opérateur "flow" est donnée à la section 10.2.26.

Ici, le fragment de code qui devrait correspondre à A; G; stop est produit par la traduction d'éventuelles occurrences de l'activité "throw" dans l'activité du service (voir la traduction de cet opérateur à la section 10.2.16), qui est traitée dans $[\![Def]\!]^E$ (faux).

L'exception "`exitFault`" est traitée différemment selon qu'elle est levée dans un service ou un sous-service :

- si l'exception est levée dans un service, alors $[\![exitFault]\!]$ = `null`, l'exception n'est pas traitée et le service va se terminer,
- si l'exception est levée dans un sous-service, alors
  $[\![exitFault]\!]$ = `faultRaise_`S' `(!exitFault)`
  , où S' est le service ou sous-service englobant.

10.2.8 Gestion des événements

Il y a deux aspects de la traduction des événements que nous traitons de façon approchée.

- Premièrement, les valeurs des alarmes ("`onAlarm`") ne sont pas prises en compte, car nous ne pouvons pas modéliser l'écoulement du temps en LOTOS NT. Nous remplaçons donc le déclenchement d'une alarme en BPEL par une transition invisible ("`i`") en LOTOS NT. Toutes les alarmes des gestionnaires d'événements peuvent alors se déclencher de manière non-déterministe.
- Deuxièmement, en BPEL, plusieurs instances d'une même réception de message peuvent être traitées simultanément, ce que nous interdisons pour éviter une explosion de la taille de l'espace d'états. Autrement dit, les réceptions consécutives d'un même message dans un gestionnaire d'événements sont exécutées séquentiellement (et non pas en parallèle) dans le code LOTOS NT généré.

Ces choix sont les mêmes que ceux faits par Lohmann [Loh08] et Ouyang et al. [OVvdA+07], qui, en revanche, ne prennent pas en compte les données. Bianculli et al. [BGS07] se contentent d'une fonction auxiliaire qui génère des constantes correspondant soit à une alarme, soit à la réception d'un message (mais sans modéliser le contenu du message).

$[\![Def]\!]^E\ (f)$ désigne la traduction des gestionnaires d'événements $EventHandler_1 ... EventHandler_{n^e}$ du service en cours de traduction. Nous traduisons ces gestionnaires par une boucle qui, à chaque itération, va choisir l'un des gestionnaires disponibles et l'exécuter. Certains gestionnaires (*onAlarmFor* et *onAlarmUntil*) ne doivent pas se répéter. Pour garantir cela, nous leur associons une valeur booléenne qui indique si le gestionnaire a déjà été exécuté. En parallèle de ces gestionnaires, nous exécutons l'activité principale du service ou sous-service.

Afin de simplifier la traduction, nous trions la liste des gestionnaires d'événements de telle sorte que :

- $EventHandler'_1 ... EventHandler'_{m^e}$ sont des gestionnaires d'événements qui ne peuvent être répétés et
- $EventHandler'_{m^e+1} ... EventHandler'_{n^e}$ sont des gestionnaires d'événements qui peuvent être répétés.

Finalement, la traduction des gestionnaires d'événements est la suivante :

$[\![Def]\!]^E\ (f)$ =
```
hide exitEventHandler_S in
  par exitEventHandler_S in
    var B1,...,Bme:Bool in
    B1 := true;...;Bme := true;
      loop eventHandlers_S in
```

```
          select
            ⟦EventHandler'_1 ⟧^Eh (B_1)
          []
            ...
          []
            ⟦EventHandler'_{m^e} ⟧^Eh (B_{m^e})
          []
            ⟦EventHandler'_{m^e+1} ⟧^Eh
          []
            ...
          []
            ⟦EventHandler'_{n^e} ⟧^Eh
          []
            exitEventHandler_S; break eventHandler_S
          end select
        end var
      end loop
    ||
      ⟦Activity ⟧ (f); exitEventHandler
      end par
    end par
  end hide
```

où $[\![Activity]\!]\ (f)$ désigne la traduction en LOTOS NT de l'activité *Activity*. Si la liste de gestionnaires d'événements est vide, alors $[\![Def]\!]^E\ (f) = [\![Activity]\!]\ (f)$.

La traduction $[\![\alpha]\!]^{Eh}$ de chaque gestionnaire d'événement α est détaillée ci-dessous :

$[\![$**OnAlarmFor** $(Expr, Activity)]\!]^{Eh}\ (B) =$
```
  if B then
    i; ⟦Activity ⟧ (faux);
    B := false
  end if
```
$[\![$**OnAlarmUntil** $(Expr, Activity)]\!]^{Eh}\ (B) =$
```
  if B then
    i; ⟦Activity ⟧ (faux);
    B := false
  end if
```
$[\![$**OnEvent** $(\lambda, F, M, X, Activity)]\!]^{Eh} =$
```
  var X:M in
    λ (?F (X));
    ⟦Activity ⟧ (faux)
  end var
```
$[\![$**OnAlarmForRepeat** $(Expr_1, Expr_2, Activity)]\!]^{Eh}$ = `i;`$[\![Activity]\!]$ (`faux`)
$[\![$**OnAlarmRepeat** $(Expr, Activity)]\!]^{Eh}$ = `i;`$[\![Activity]\!]$ (`faux`)
$[\![$**OnAlarmUntilRepeat** $(Expr_1, Expr_2, Activity)]\!]^{Eh}$ = `i;`$[\![Activity]\!]$ (`faux`)

La norme BPEL [Com01, Sec. 11.6.1] spécifie que les activités associées aux gestionnaires d'événements sont des constructions itératives et que, par conséquent, elles ne peuvent contenir de liens déclarés

dans une activité "`flow`" parente. Pour cette raison, ces activités ne sont jamais considérées comme étant dans une activité "`flow`", même si le sous-service dans lequel elles sont déclarés est compris dans une activité "`flow`".

10.2.9 Traduction de l'activité "empty"

L'activité "`empty`" est sémantiquement équivalente à l'opérateur "`null`" de LOTOS NT, donc la traduction de cette activité est la suivante ;

$[\![\textbf{Empty}]\!]\ (f)\ =\ \texttt{null}$

En dépit de la simplicité de cette activité, la plupart des auteurs (Ouyang et al. [OVvdA+07], Xiang Fu [Fu04], Hinz, Schmidt et Stahl [HSS05], Nakajima [Nak06], Salaün et al. [SBS06], Qian et al. [QXW+07] et Fisteus et al. [FFK05]) ne la mentionnent pas.

10.2.10 Traduction de l'activité "exit"

L'activité "`exit`" entraîne l'arrêt de l'instance courante du service. La norme BPEL n'est pas suffisamment explicite au sujet de la sémantique de cette activité. Comme pour la levée des exceptions, deux interprétations existent :

- l'arrêt immédiat de toutes les activités du service [OVvdA+07] ou
- l'arrêt, sans contrainte de délai, des activités du service [HSS05, Loh08].

Comme pour les gestionnaires d'activité, nous avons choisi la seconde solution, car la première ne peut pas être mise en œuvre dans LOTOS NT tant que la capture d'exceptions n'est pas supportée par le traducteur LOTOS NT vers LOTOS.

La traduction de l'activité "`exit`" n'est pas aussi facile qu'elle peut le paraître, car il faut pouvoir arrêter les branches s'exécutant en parallèle de l'activité qui déclenche la terminaison. Certains auteurs donnent des traductions fausses ou incomplètes. Par exemple, Foster [Fos06] propose une traduction qui se limite au cas extrêmement simple dans lequel l'activité "`exit`" est une activité fille d'une activité "`flow`" et les autres activités filles de cette activité "`flow`" sont des réceptions de messages. Il s'avère que cette traduction, dans le cas particulier qu'elle traite, est correcte et respecte la première interprétation de la norme. Yeung [Yeu06], en revanche, choisit de traduire l'activité "`exit`" de BPEL par l'opérateur "`stop`" de CSP, ce qui est une erreur. En effet, cet opérateur a pour effet d'arrêter la branche parallèle dans laquelle il survient, mais n'affecte pas les autres branches parallèles.

Après l'arrêt d'une instance d'un service par l'activité "`exit`", une nouvelle instance doit pouvoir démarrer. Pour simuler ce comportement, nous traduisons l'activité "`exit`" par la levée de l'exception "`exitFault`", qui est capturée puis renvoyée par les gestionnaires d'exceptions de chaque sous-service, jusqu'à arriver aux gestionnaires d'exceptions du service principal, ce qui entraîne la terminaison de ce service. En considérant que l'activité "`exit`" est englobée par le service ou sous-service S, sa traduction est la suivante :

$[\![\textbf{Exit}\ ]\!]\ (f)\ =\ \texttt{faultRaise_}S\ \texttt{(!exitFault)}$

10.2.11 Traduction de l'activité "wait"

L'activité "`wait`" ne peut être fidèlement représentée dans l'implémentation actuelle de LOTOS NT car les constructions de ce langage liées au temps ne sont pas traitées par le compilateur LNT2LOTOS. Dans notre traduction, l'écoulement du temps est non-déterministe :

$[\![\textbf{WaitFor}\ (\mathit{Expr})]\!]\ (f)\ =$ `i`
$[\![\textbf{WaitUntil}\ (\mathit{Expr})]\!]\ (f)\ =$ `i`

Cela a des conséquences sur l'ordonnancement des activités au sein d'une activité "`flow`", car des comportements qui ne sont pas permis par la norme BPEL sont autorisés en LOTOS NT. Par exemple :

- A, B, C et D sont les quatre activités parallèles d'une activité "`flow`".
- un lien a pour source A et pour cible C,
- un lien a pour source B et pour cible D,
- $A = \mathit{WaitFor}\ (\mathit{Expr})$ et $B = \mathit{WaitFor}\ (\mathit{Expr}')$ et que $\mathit{Expr} < \mathit{Expr}'$,

alors dans l'exécution de ces activités, D ne pourra jamais démarrer avant l'activité C tandis que notre traduction le permet.

10.2.12 Traduction de l'activité "receive"

L'activité "`receive`" est traduite de deux façons, selon que la variable, dans laquelle est enregistrée le message reçu, est partagée ou non. Si la variable n'est pas partagée, alors la traduction de l'activité "`receive`" est la suivante :

$[\![\textbf{Receive}\ (\lambda, F, X)]\!]\ (f)\ =\ \lambda$ `(?`F`_Input (`X`))`

Si la variable est partagée, il faut communiquer avec le gestionnaire d'atomicité pour s'assurer que, en LOTOS NT, la réception du message et son affectation dans la variable partagée ne sont pas interrompues par d'autres activités parallèles. La traduction de l'activité "`receive`" est alors la suivante :

$[\![\textbf{Receive}\ (\lambda, F, X)]\!]\ (f)\ =$
`var` X`:`$\mathit{type}\ (X)$ `in`
  `Lock (!`$[\![\textbf{Receive}\ (\lambda, F, X)]\!]^{\mathit{atomic}}$`);`
  λ `(?`F`_Input (`X`));`
  `write_`X `(!`X`);`
  `Release (!`$[\![\textbf{Receive}\ (\lambda, F, X)]\!]^{\mathit{atomic}}$`)`
`end var`

10.2.13 Traduction de l'activité "reply"

L'activité "`reply`" présente deux cas de figure, selon que l'activité envoie un message (*Reply*) ou une exception (*ReplyFaultData*).

Dans le cas où un message est envoyé, la traduction de l'activité "`reply`" suit un schéma analogue à celui de l'activité "`receive`". Si la variable n'est pas partagée, alors la traduction de l'activité "`reply`" est la suivante :

$[\![\textbf{Reply}\ (\lambda, F, X)]\!]\ (f)\ =\ \lambda\ (!F_\texttt{Output}\ (X))$

Si la variable est partagée, alors la traduction de l'activité "`reply`" est la suivante :

$[\![\textbf{Reply}\ (\lambda, F, X)]\!]\ (f)\ =$
  `var` X `:` *type* (X) `in`
    `Lock (!`$[\![\textbf{Reply}\ (\lambda, F, X)]\!]^{atomic}$`);`
    `read_`X `(?`X`);`
    λ `(!`F`_Output (`X`));`
    `Release (!`$[\![\textbf{Reply}\ (\lambda, F, X)]\!]^{atomic}$`)`
  `end var`

Dans le cas où une exception est envoyée, la traduction de l'activité "`reply`" suit un schéma similaire. Si la variable qui comporte les données associées à l'exception n'est pas partagée, alors la traduction de l'activité "`reply`" est la suivante :

$[\![\textbf{ReplyFaultData}\ (\lambda, F, \chi, X)]\!]\ (f)\ =\ \lambda\ (!F_\texttt{Fault}_\chi\ (\chi(X)))$

Si la variable est partagée, alors la traduction de l'activité "`reply`", dans le cas où une exception est envoyée, est la suivante :

$[\![\textbf{ReplyFaultData}\ (\lambda, F, \chi, X)]\!]\ (f)\ =$
  `var` X `:` *type* (X) `in`
    `Lock (!`$[\![\textbf{ReplyFaultData}\ (\lambda, F, \chi, X)]\!]^{atomic}$`);`
    `read_`X `(?`X`);`
    λ `(!`F`_Fault_`χ `(`X`));`
    `Release (!`$[\![\textbf{ReplyFaultData}\ (\lambda, F, \chi, X)]\!]^{atomic}$`)`
  `end var`

10.2.14 Traduction de l'activité "invoke"

L'activité "`invoke`" présente deux cas de figure, selon que l'invocation est à sens unique (*InvokeOneWay*, sans réponse attendue) ou à double sens (*InvokeTwoWays*, attente d'une réponse et gestionnaires d'exceptions). La traduction de ces deux cas de figure dépend du partage des variables utilisées comme conteneurs des messages d'entrée et de sortie de la fonction invoquée.

La traduction de *InvokeOneWay*, si la variable contenant le message d'entrée n'est pas partagée, est la suivante :

$[\![\textbf{InvokeOneWay}\ (\lambda, F, X_I)]\!]\ (f)\ =\ \lambda\ (!F_\texttt{Input}\ (X_I))$

La traduction de *InvokeOneWay*, si la variable contenant le message d'entrée est partagée, est la suivante :

$[\![\textbf{InvokeOneWay}\ (\lambda, F, X_I)]\!]\ (f)\ =$
  `var` X_I `:` *type* (X_I) `in`
    `Lock (!`$[\![\textbf{InvokeOneWay}\ (\lambda, F, X_I)]\!]^{atomic}$`);`
    `read_`X_I `(?`X_I`);`
    λ `(!`F`_Input (`X_I`));`
    `Release (!`$[\![\textbf{InvokeOneWay}\ (\lambda, F, X_I)]\!]^{atomic}$`)`
  `end var`

La traduction de l'invocation d'une fonction à double sens

$$\textbf{InvokeTwoWays}\ \ (\lambda, F, X_I, X_O, \mathit{FaultHandler}_1 ... \mathit{FaultHandler}_n)$$

peut être simplifiée car la norme BPEL [Com01, Sec. 10.3] définit l'équivalence suivante :

$$\textbf{InvokeTwoWays}\ (\lambda, F, X_I, X_O, \mathit{FaultHandler}_1 ... \mathit{FaultHandler}_n) = \textbf{Service}\ (\texttt{"invoke"}, \mathit{FaultHandler}_1 ... \mathit{FaultHandler}_n, \textbf{InvokeTwoWays}\ (\lambda, F, X_I, X_O))$$

Grâce à cette identité, nous n'avons à nous préoccuper, dans la traduction des invocations de fonctions, que de la levée des exceptions, mais pas des gestionnaires d'exceptions. Si la définition WSDL de la fonction ne déclare aucune exception, et que les variables contenant les messages d'entrée et de sortie de la fonction ne sont pas partagées, la traduction de l'invocation de la fonction consiste à envoyer le message d'entrée et à attendre le message de sortie :

$[\![\textbf{InvokeTwoWays}\ (\lambda, F, X_I, X_O)]\!]\ (f)$
Lock;
$\mathit{PreCode}^{X_I}$
λ (!F_Input (X_I))
$\mathit{PostCode}^{X_I}$
;
$\mathit{PreCode}^{X_O}$
λ (?F_Output (X_O))
$\mathit{PostCode}^{X_O}$
Release

où $\mathit{PreCode}^{X_I}$, $\mathit{PostCode}^{X_I}$, $\mathit{PreCode}^{X_O}$ et $\mathit{PostCode}^{X_O}$ sont des portions de code LOTOS NT qui changent selon que les variables X_I et X_O sont partagées ou non :

- Si X_I n'est pas une variable partagée, alors $\mathit{PreCode}^{X_I}$ et $\mathit{PostCode}^{X_I}$ sont vides.
- Si X_I est une variable partagée, alors nous devons déclarer une nouvelle variable X_I (de type M_I) qui va masquer la première. Nous utilisons le gestionnaires de variables pour récupérer la valeur de la première variable X_I, puis nous passons cette valeur comme message d'entrée de la fonction :
  $\mathit{PreCode}^{X_I}$ =
  "var X_I:M_I in
  Read_X_I (?X_I);"
  et :
  $\mathit{PostCode}^{X_I}$ = "end var"
- Si X_O n'est pas une variable partagée, alors $\mathit{PreCode}^{X_O}$ et $\mathit{PostCode}^{X_O}$ sont vides.
- Si X_O est une variable partagée, alors nous devons déclarer une nouvelle variable X_O (de type M_O) qui va masquer la première. Nous récupérons la valeur du message de sortie de la fonction dans cette nouvelle variable, puis nous l'utilisons pour mettre à jour la valeur de la première variable X_O :
  $\mathit{PreCode}^{X_O}$ = "var X_O:M_O in"
  et :
  $\mathit{PostCode}^{X_O}$ =
  ;
  Write_X_O (!X_O)
  end var

Si la définition WSDL de la fonction F, utilisée dans *InvokeTwoWays* déclare les exceptions $\chi_1 : M_1 ... \chi_n : M_n$, alors, après l'envoi du message d'entrée de la fonction, il faut attendre, soit la réception du message de sortie, soit la réception d'une des exceptions $\chi_1 ... \chi_n$. Après la réception de l'une des exceptions, l'exécution de l'activité s'arrête, ce qui est traduit par un "`stop`" en LOTOS NT (voir la traduction des exceptions à la section 10.2.7). Ce "`stop`" est précédé d'une synchronisation sur la porte "`Release`" afin de libérer le gestionnaire d'atomicité :

$[\![$**InvokeTwoWays** $(\lambda, F, X_I, X_O)]\!]$ (f)
  `Lock (!`$[\![$**InvokeTwoWays** $(\lambda, F, X_I, X_O)]\!]^{atomic}$`);`
  $PreCode^{X_I}$
  λ `(!`F`_Input(`X_I`))`
  $PostCode^{X_I}$
  `;`
  `select`
    $Precode^{X_O}$
    λ `(?`F`_Output(`X_O`))`
    $Postcode^{X_O}$
  `[]`
    `var F:`M_1 `in`
      λ `(?`F`_Fault_`χ_1 `(F));`
      `faultRaise_`S `(!F);`
      `Release (!`$[\![$**InvokeTwoWays** $(\lambda, F, X_I, X_O)]\!]^{atomic}$`); stop`
    `end var`
  `[]`
  ...
  `[]`
    `var F:`M_n `in`
      λ `(?`F`_Fault_`χ_n `(F));`
      `faultRaise_`S `(!F);`
      `Release (!`$[\![$**InvokeTwoWays** $(\lambda, F, X_I, X_O)]\!]^{atomic}$`); stop`
    `end var`
  `end select;`
  `Release (!`$[\![$**InvokeTwoWays** $(\lambda, F, X_I, X_O)]\!]^{atomic}$`)`

10.2.15 Traduction de l'activité "assign"

A notre connaissance, aucun des travaux précédemment réalisés au sujet de la vérification formelle de services Web BPEL ne mentionne les variables partagées (cf. section 10.2.6) et les constantes BPEL (cf. section 7.5). Par conséquent, aucune approche ne traite correctement l'activité d'affectation "`assign`". Par exemple, Foster [Fos06] et Fu [Fu04] ne considèrent que le cas de l'affectation d'une constante, représentée par une chaîne de caractères, dans une variable. Bianculli et al. [BGS07] traduisent l'activité "`assign`" de BPEL par l'opérateur d'affectation "`:=`" du langage BIR, mais sans donner le moindre détail.

Concernant la sémantique de l'activité "`assign`", la norme BPEL [Com01] ne précise pas si les multiples copies déclarées par cette activité sont effectuées simultanément ou séquentiellement. Dans notre traduction, nous considérons des copies séquentielles. Chaque copie remplace la valeur d'une variable (ou d'un champ de variable) déterminée par l'évaluation d'une expression XPATH, en lui donnant une nouvelle valeur qui résulte, soit de l'évaluation d'une seconde expression XPATH, soit d'une constante

BPEL.

La traduction de l'activité "`assign`" consiste en la concaténation séquentielle des traductions des affectations (Var_0`:=`$From_0$, ..., Var_n`:=`$From_n$) entre deux synchronisations sur les portes "`Lock`" et "`Release`" pour garantir l'atomicité de l'exécution de l'activité :

$[\![$**Assign** (Var_0`:=`$From_0$, ..., Var_n`:=`$From_n$)$]\!]$ (f) =
  `Lock (!`$[\![$**Assign** (Var_0`:=`$From_0$, ..., Var_n`:=`$From_n$)$]\!]^{atomic}$`);`
  $[\![Var_0$`:=`$From_0]\!]$ `;`
  ...`;`
  $[\![Var_n$`:=`$From_n]\!]$ `;`
  `Release (!`$[\![$**Assign** (Var_0`:=`$From_0$, ..., Var_n`:=`$From_n$)$]\!]^{atomic}$`)`

La traduction d'une affectation unique diffère selon que *From* représente une expression XPATH ou une constante BPEL. Soient $X_1...X_n$ les variables partagées figurant dans la variable en partie gauche *Var* (qui peut être une expression complexe accédant aux champs d'une variable) et $T_1...T_n$ leurs types respectifs.

- Dans le cas d'une expression XPATH, nous notons $X'_1...X'_{n'}$ les variables partagées (de types respectifs $T'_1...T'_{n'}$) figurant dans l'expression en partie droite *Expr*, qui n'appartiennent pas à $\{X_1...X_n\}$. La traduction de l'affectation en LOTOS NT est alors :

  $[\![Var$ `:=` $Expr]\!]$ =
    `var` X_1`:`T_1`,...,`X_n`:`T_n`,`X'_1`:`T'_1`,...`$X'_{n'}$`:`$T'_{n'}$ `in`
      `Read_`X_1 `(?`X_1`);`
      ...`;`
      `Read_`X_n `(?`X_n`);`
      `Read_`X'_1 `(?`X'_1`);`
      ...`;`
      `Read_`$X'_{n'}$ `(?`$X'_{n'}$`);`
      $[\![Var \leftarrow Expr]\!]^G$
    `end var`

  où $[\![Var \leftarrow Expr]\!]^G$ a été défini à la section 8.6.4.

- Si la partie droite de l'affectation est une constante BPEL, alors nous déclarons une nouvelle variable X_c dont le type est celui de *Var*. Cette variable reçoit, comme valeur, l'expression LOTOS NT résultant de la traduction de la constante *Literal* (détaillée à la section 7.5) avec pour type cible le type de l'expression *Var* :

  $[\![Var$ `:=` $Literal]\!]$ =
    `var` X_1`:`T_1`,...,`X_n`:`T_n`,`X_c`:` *Var.type* `in`
      `Read_`X_1 `(?`X_1`);`
      ...`;`
      `Read_`X_n `(?`X_n`);`
      X_c `:=` $[\![Literal]\!]^{Const}$ (*Var.type*)`;`
      $[\![Var \leftarrow X_c]\!]^G$
    `end var`

  où $[\![Literal]\!]^{Const}$ (*Var.type*) a été défini à la section 7.5 et $[\![Var \leftarrow Expr]\!]^G$ à la section 8.6.4.

Dans les deux cas, *Var* peut être une expression XPath complexe qui extrait une valeur parmi les champs d'une variable X. Si X est une variable partagée, alors il faut rajouter la ligne suivante avant "`end var`" :

```
; Write_X (!X)
```

10.2.16 Traduction de l'activité "throw"

L'activité "`throw`" est traduite en suivant le schéma mis en place pour la traduction des gestionnaires d'exceptions (cf. section 10.2.7). Soit S le service ou sous-service qui englobe une activité "`throw`".

La traduction de la levée d'une exception sans données associées est la suivante :

$[\![\textbf{Throw } (\chi)]\!]\ (f)\ =$ `faultRaise_`S `(!`χ`); stop`

La traduction de la levée d'une exception avec données associées, si la variable X qui contient ces données n'est pas partagée, est la suivante :

$[\![\textbf{ThrowData } (\chi, X)]\!]\ (f)\ =$ `faultRaise_`S `(!`χ `(`X`)); stop`

La traduction de la levée d'une exception avec données associées, si la variable X qui contient ces données est partagée, est la suivante :

$[\![\textbf{ThrowData } (\chi, X)]\!]\ (f)\ =$

```
  var X:X.type in
    Read_X (?X);
    faultRaise_S (!χ (X)); stop
  end var
```

Dans les trois cas, la levée de l'exception en LOTOS NT est suivie de l'opérateur "`stop`". Cela permet, lorsque cette levée d'exception se produit dans une activité parallèle au sein d'une activité "`flow`", d'empêcher que l'activité qui suit le "`flow`" (si elle existe) soit exécutée. Dans le même temps, cela n'empêche pas le service de continuer de s'exécuter (voire de boucler indéfiniment). Afin d'illustrer notre propos, considérons l'exemple suivant :

```
par
  A ; stop
||
  B
end par ;
C
```

En raison de la présence d'un "`stop`" après "`A`", "`C`" ne peut jamais être exécutée, contrairement à "`B`".

10.2.17 Traduction de l'activité "rethrow"

L'activité "`rethrow`" ne peut apparaître que dans un gestionnaire d'exception. Sa traduction doit donc suivre le schéma mis en place pour la traduction des gestionnaires d'exceptions (cf. section 10.2.7), dans lequel chaque gestionnaire d'exception traite une exception représentée par "`F`", une variable LOTOS NT. Soit S le service ou sous-service qui déclare le gestionnaire d'exception dans lequel l'activité "`rethrow`" apparaît. Soit S' le service ou sous-service qui englobe S. La traduction de

cette activité consiste alors à lever de nouveau l'exception "F" qui sera capturée par les gestionnaires d'exceptions de S'.

$[\![\textbf{Rethrow}\ ]\!]\ (f)\ =$ `faultRaise_`S' `(!F)`

10.2.18 Traduction de l'activité "validate"

L'activité "`validate`" est traduite par "`null`" en LOTOS NT :

$[\![\textbf{Validate}\ (X_0...X_n)]\!]\ (f)\ =$ `null`

En effet, en LOTOS NT, les valeurs sont toujours conformes à leur type, il n'est donc pas nécessaire de prévoir un traitement particulier pour l'activité "`validate`".

10.2.19 Traduction de l'activité "sequence"

L'activité "sequence" de BPEL qui consiste à exécuter n activités de façon séquentielle est directement traduite au moyen de l'opérateur ";" de LOTOS NT :

$[\![\textbf{Sequence}\ (\mathit{Activity}_0...\mathit{Activity}_n)]\!]\ (f)\ =$
$[\![\mathit{Activity}_0]\!]\ (f)$;
...;
$[\![\mathit{Activity}_n]\!]\ (f)$;

10.2.20 Traduction de l'activité "if"

L'activité "`if`" de BPEL est traduite au moyen de l'opérateur LOTOS NT de même nom. Soient $X_1...X_n$ les variables partagées (de types respectifs $T_1...T_n$) figurant dans l'expression booléenne *Expr* qui sert de condition à l'activité "`if`". Alors :

$[\![\textbf{If}\ (\mathit{Expr}, \mathit{Activity}_1, \mathit{Activity}_2)]\!]$ (`faux`) =
`var` X_1`:`T_1`,...,`X_n`:`T_n `in`
  `Read_`X_1 `(?`X_1`);`
  ...;
  `Read_`X_n `(?`X_n`);`
  `if` $[\![\mathit{Expr}]\!]^D$ `then`
    $[\![\mathit{Activity}_1]\!]$ (`faux`)
  `else`
    $[\![\mathit{Activity}_2]\!]$ (`faux`)
  `end if`
`end var`

L'attribut de conversion de l'expression *Expr* doit, au préalable, être initialisé par un appel à *décideConversion* (*Expr*, boolean).

La traduction de l'activité "`if`" à l'intérieur d'une activité "`flow`" (lorsque l'argument de la fonction de traduction vaut "`vrai`") est détaillée à la section 10.2.26.

10.2.21 Traduction de l'activité "while"

L'activité "`while`" de BPEL est traduite au moyen de l'opérateur LOTOS NT de même nom. Soient $X_1...X_n$ les variables partagées (de types respectifs $T_1...T_n$) figurant dans l'expression booléenne *Expr* qui sert de condition de continuation à l'activité "`while`". Alors :

$[\![$**While** (*Expr*, *Activity*)$]\!]$ (f) =
  `var` X_1`:`T_1`,...,`X_n`:`T_n `in`
    `Read_`X_1 `(?`X_1`);`
    `...;`
    `Read_`X_n `(?`X_n`);`
    `while` $[\![$*Expr*$]\!]^D$ `loop`
      $[\![$*Activity*$]\!]$ (`faux`);
      `Read_`X_1 `(?`X_1`);`
      `...;`
      `Read_`X_n `(?`X_n`)`
    `end loop`
  `end var`

L'attribut de conversion de l'expression *Expr* doit au préalable être initialisé par un appel à *décideConversion* (*Expr*, boolean).

L'activité "`while`" est une construction itérative. Comme expliqué à la section 10.2.8, une activité "`while`" est toujours considérée, dans le cadre de la traduction, comme ne figurant pas au sein d'une activité "`flow`". Par conséquent, la valeur de l'argument f pour la traduction de la sous-activité est "`faux`".

10.2.22 Traduction de l'activité "repeatUntil"

L'activité "`repeatUntil`" de BPEL est traduite en LOTOS NT au moyen de l'opérateur "`while`" qui itère tant que la condition d'arrêt est fausse. Cette condition est évaluée après chaque exécution de la sous-activité. Soient $X_1...X_n$ les variables partagées (de types respectifs $T_1...T_n$) figurant dans l'expression booléenne *Expr* qui sert de condition d'arrêt à l'activité "`repeatUntil`" :

$[\![$**RepeatUntil** (*Expr*, *Activity*)$]\!]$ (f) =
  `var` X_1`:`T_1`,...,`X_n`:`T_n `in`
    $[\![$*Activity*$]\!]$ (`faux`);
    `Read_`X_1 `(?`X_1`);`
    `...;`
    `Read_`X_n `(?`X_n`);`
    `while not(`$[\![$*Expr*$]\!]^D$`) loop`
      $[\![$*Activity*$]\!]$ (`faux`);
      `Read_`X_1 `(?`X_1`);`
      `...;`
      `Read_`X_n `(?`X_n`)`
    `end loop`
  `end var`

L'attribut de conversion de l'expression *Expr* doit au préalable être initialisé par un appel à *décideConversion* (*Expr*, boolean).

L'activité "`repeatUntil`" est une construction itérative. Comme expliqué à la section 10.2.8, une activité "`repeatUntil`" est toujours considérée, dans le cadre de la traduction, comme ne figurant pas au sein d'une activité "`flow`". Par conséquent, la valeur de l'argument f pour la traduction de la sous-activité est "`faux`".

10.2.23 Traduction de l'activité "forEach"

L'activité "`forEach`" de BPEL possède deux variantes. La variante sans condition d'arrêt est directement traduite vers l'opérateur équivalent "`for`" de LOTOS NT. Soient $X_1^1...X_{n^1}^1$ les variables partagées (de types respectifs $T_1^1...T_{n^1}^1$) figurant dans l'expression entière $Expr_1$ qui initialise la variable d'itération X et $X_1^2...X_{n^2}^2$ les variables partagées (de types respectifs $T_1^2...T_{n^2}^2$) qui n'appartiennent pas à $\{X_1^1...X_{n^1}^1\}$ et qui figurent dans l'expression entière $Expr_2$ qui indique la valeur maximale que peut prendre cette variable d'itération X. Ainsi :

$[\![$**ForEach** $(X, Expr_1, Expr_2, Activity)]\!]$ (f) =
  `var` X_1^1`:`T_1^1`,...,`$X_{n^1}^1$`:`$T_{n^1}^1$`,`X_1^2`:`T_1^2`,...,`$X_{n^2}^2$`:`$T_{n^2}^2$`,`X`,Max:Nat in`
    `Read_`X_1^1 `(?`X_1^1`);`
    `...;`
    `Read_`$X_{n^1}^1$ `(?`$X_{n^1}^1$`);`
    `Read_`X_1^2 `(?`X_1^2`);`
    `...;`
    `Read_`$X_{n^2}^2$ `(?`$X_{n^2}^2$`);`
    `Max :=` $[\![Expr_2]\!]^D$`;`
    `for` X `:=` $[\![Expr_1]\!]^D$ `while` X `<= Max by 1 loop`
      $[\![Activity]\!]$ `(faux);`
      `Read_`X_1^1 `(?`X_1^1`);`
      `...;`
      `Read_`$X_{n^1}^1$ `(?`$X_{n^1}^1$`)`
    `end loop`
  `end var`

L'activité "`forEach`" est une construction itérative. Comme expliqué à la section 10.2.8, une activité "`forEach`" est toujours considérée, dans le cadre de la traduction, comme ne figurant pas au sein d'une activité "`flow`". Par conséquent, la valeur de l'argument f pour la traduction de la sous-activité est "`faux`".

La variante de l'activité "`forEach`" avec condition d'arrêt, que nous notons :

$$\textbf{ForEachCond}\ (X, Expr_1, Expr_2, Expr_3, V^b, Activity)$$

se termine après que la boucle a été exécutée autant de fois que spécifié par l'expression $Expr_3$. V^b détermine si toutes les exécutions de l'activité, ou bien seulement celles qui ont terminé avec succès, sont comptabilisées.

Si la valeur de V^b est fausse, le seul changement par rapport à la traduction de la variante sans condition d'arrêt est l'ajout d'un compteur d'itérations et d'un test de sa valeur à chaque début de boucle. Ce test vérifie que le nombre d'itérations reste bien inférieur à $Expr_3$. Soient $X_1^3, ..., X_{n^3}^3$ les variables partagées (de types respectifs $T_1^3, ..., T_{n^3}^3$) figurant dans l'expression entière $Expr_3$, mais n'appartenant ni à $\{X_1^1, ..., X_{n^1}^1\}$, ni à $\{X_1^2, ..., X_{n^2}^2\}$. La traduction de la variante avec condition d'arrêt dans le cas où V^b vaut "`faux`" est alors la suivante :

$[\![$**ForEachCond** $(X, Expr_1, Expr_2, Expr_3, V^b = \texttt{faux}, Activity)]\!]\ (f)\ =$

```
var X¹₁:T¹₁,...,X¹_{n¹}:T¹_{n¹},X²₁:T²₁,...,X²_{n²}:T²_{n²},X³₁:T³₁,...,X³_{n³}:T³_{n³},X,Max,Counter:Nat in
  Read_X¹₁ (?X¹₁);
  ...;
  Read_X¹_{n¹} (?X¹_{n¹});
  Read_X²₁ (?X²₁);
  ...;
  Read_X²_{n²} (?X²_{n²});
  Read_X³₁ (?X³₁);
  ...;
  Read_X³_{n³} (?X³_{n³});
  Max := [[Expr2]]^D;
  StopCond := [[Expr3]]^D;
  for X := [[Expr1]]^D; Counter:= 0 while X <= Max and Counter <= StopCond by 1 loop
    [[Activity]] (faux);
    Read_X¹₁ (?X¹₁);
    ...;
    Read_X¹_{n¹} (?X¹_{n¹});
    Counter := Counter + 1
  end loop
end var
```

Comme l'activité contenue dans une activité "`forEach`" est forcément un sous-service, il suffit, pour traduire le cas où V^s vaut "`vrai`", de déplacer l'incrémentation du compteur après l'exécution de *Activity*, qui, nous le rappelons, est forcément la déclaration d'un sous-service S. Autrement dit, dans la traduction de S, la ligne :

$[\![Def]\!]^E\ (f)$; `exitFaultRaise_`S

est remplacée par :

$[\![Def]\!]^E\ (f)$; `Counter := Counter + 1; exitFaultRaise_`S

Les attributs de conversion des expressions $Expr_1$, $Expr_2$ et $Expr_3$ doivent au préalable être initialisés par des appels à :

- *décideConversion* ($Expr_1$, nonNegativeInteger),
- *décideConversion* ($Expr_2$, nonNegativeInteger) et
- *décideConversion* ($Expr_3$, nonNegativeInteger).

10.2.24 Traduction de l'activité "pick"

L'activité "`pick`" permet de choisir parmi plusieurs activités, la première qui devient disponible, soit après la réception d'un message, soit après le déclenchement d'une alarme.

Pour certaines approches [Loh08, OVvdA+07] qui ne représentent pas les données, la traduction de l'activité "`pick`" se fait par le choix non-déterministe de l'une des branches. Bianculli et al. [BGS07] propose une solution similaire, quand bien même leur traduction prend en compte les données. Koshkina et Breugel [KvB04] ainsi que Foster [Fos06] ne traduisent que les réceptions de messages, mais sans prendre en compte le contenu des messages.

Dans notre traduction, nous prenons en compte les réception de messages ainsi que leur contenu et les alarmes dont le déclenchement est non-déterministe. L'activité "`pick`" est traduite au moyen de l'opérateur "`select`" de LOTOS NT :

$[\![$**Pick** $(EventHandler_0 ... EventHandler_n)]\!]$ (`faux`) =
`select`
$[\![EventHandler_0]\!]^{Pick}$ (`faux`)
`[]`
...
`[]`
$[\![EventHandler_n]\!]^{Pick}$ (`faux`)
`end select`

Cette traduction respecte la sémantique donnée par la norme BPEL et qui veut que, parmi les différents événements considérés par une activité "`pick`", le premier qui survient soit choisi. En effet, notre approche modélise un service BPEL en vue de le vérifier formellement. Il est donc nécessaire de considérer, pour l'activité "`pick`", que n'importe quel événement puisse survenir en premier (afin de générer tous les cas possibles). C'est précisément ce que fait l'opérateur "`select`" de LOTOS NT.

Si l'un des événements est une alarme, alors il pourra être choisi de façon non-déterministe (approximation nécessaire, car les constructions liées au temps ne sont pas prises en compte dans l'implémentation actuelle du traducteur LNT2LOTOS).

La traduction de l'activité "`pick`" au sein d'une activité "`flow`" est détaillé à la section 10.2.26.

La traduction des gestionnaires d'événements compris dans une activité "`pick`" est la suivante :

$[\![$**OnAlarmFor** $(Expr, Activity)]\!]^{Pick}$ (f) = "`i; `$[\![Activity]\!]$ (f)"
$[\![$**OnAlarmUntil** $(Expr, Activity)]\!]^{Pick}$ (f) (B) = "`i; `$[\![Activity]\!]$ (f)"
$[\![$**OnEvent** $(\lambda, F, M, X, Activity)]\!]^{Pick}$ (f) = "`var` X`:`M `in`
λ `(?`F `(`X`));`
$[\![Activity]\!]$ (f)
`end var`"
$[\![$**OnAlarmForRepeat** $(Expr_1, Expr_2, Activity)]\!]^{Pick}$ (f) = *cas interdit*
$[\![$**OnAlarmRepeat** $(Expr, Activity)]\!]^{Pick}$ (f) = *cas interdit*
$[\![$**OnAlarmUntilRepeat** $(Expr_1, Expr_2, Activity)]\!]^{Pick}$ (f) = *cas interdit*

10.2.25 Traduction de l'activité "flow" et des liens de contrôle

La complexité de la traduction de l'activité "`flow`" réside dans le mécanisme de liens de contrôle entre les sous-activités parallèles. Nous reproduisons ce mécanisme en LOTOS NT à l'aide de :

1. une structure de données représentant les trois valeurs possibles ("`vrai`", "`faux`", "`indéfini`") d'un lien, et

1. un gestionnaire de liens, similaire au gestionnaire de variables, qui permet de lire et d'écrire la valeur d'un lien.

De toutes les approches que nous avons rencontrées, aucune n'implémente complètement la sémantique des liens de contrôle. Les approches les plus complètes sont celles de Lohmann [Loh08] et Ouyang et al. [OVvdA+07] ; toute la sémantique des liens de contrôle est traduite, sauf les aspects relevant des données comme les conditions de démarrage et de transitions. Foster [Fos06], Fu [Fu04],

Koshkina et Breugel [KvB04] et Bianculli et al. [BGS07] ne prennent en compte que l'aspect de la sémantique des liens qui relèvent de la synchronisation entre branches parallèles au sein de l'activité "`flow`", en revanche, les approches de ces différents auteurs occultent les aspects relatifs à la propagation de valeurs sur les liens en cas de levée d'exceptions.

Comme défini dans la norme BPEL, nous représentons, dans notre traduction, le domaine de valeurs d'un lien de contrôle par un type LOTOS NT pouvant prendre trois valeurs. Le constructeur "`unset`" représente la valeur indéfinie tandis que "`set (value:Bool)`" représente la valeur booléenne du lien après qu'il a été initialisé :

```
type LinkType is
  unset,
  set (value:Bool)
end type
```

Le gestionnaire de liens de contrôle, à l'instar du gestionnaire de variables, doit pouvoir recevoir et envoyer les valeurs des liens de contrôle par le biais de portes de communication. Il est donc nécessaire de définir un canal pour typer ces portes :

```
channel LinkTypeChannel is (LinkType) end channel
```

Afin de détailler la traduction de l'activité "`flow`" et des liens de contrôle, nous avons besoin de connaître, pour chaque activité *Activity* :

- l'ensemble des liens entrants de *Activity* que nous notons $Activity.\mathcal{L}^I$, c'est-à-dire les liens ayant pour cible *Activity* ou une activité comprise dans *Activity* et qui ne sont pas définis dans une activité "`flow`" comprise dans *Activity*, et
- l'ensemble des liens sortants de *Activity* que nous notons $Activity.\mathcal{L}^O$, c'est-à-dire les liens ayant pour source *Activity* ou une activité comprise dans *Activity* et qui ne sont pas définis dans une activité "`flow`" comprise dans *Activity*.

Le calcul de $Activity.\mathcal{L}^I$ est défini au moyen de la grammaire attribuée présentée ci-dessous. L'attribut hérité $\mathcal{L}$ désigne l'ensemble des liens de contrôle définis dans l'activité. Cet ensemble est enrichi au fur et à mesure du parcours de l'arbre de syntaxe, lorsque des activités "`flow`" sont rencontrées. L'attribut synthétisé $\mathcal{L}'$ désigne l'ensemble des liens de contrôle sortants de l'activité.

$$
\begin{array}{lcl}
Activity \downarrow \mathcal{L} \uparrow \mathcal{L}' & ::= & \textbf{Scope}\ (Def) \downarrow \mathcal{L} \uparrow \mathcal{L}' \quad | \quad \textbf{Empty}\ \uparrow \emptyset \\
& | & \textbf{Exit}\ \uparrow \emptyset \quad | \quad \textbf{WaitFor}\ (Expr) \uparrow \emptyset \\
& | & \textbf{WaitUntil}\ (Expr) \uparrow \emptyset \quad | \quad \textbf{Receive}\ (\lambda, F, X) \uparrow \emptyset \\
& | & \textbf{Reply}\ (\lambda, F, X) \uparrow \emptyset \quad | \quad \textbf{ReplyFault}\ (\lambda, F, \chi) \uparrow \emptyset \\
& | & \textbf{ReplyFaultData}\ (\lambda, F, \chi, X) \uparrow \emptyset \quad | \quad \textbf{InvokeOneWay}\ (\lambda, F, X_I) \uparrow \emptyset \\
& | & \textbf{InvokeTwoWays}\ (\lambda, F, X_I, X_O, \\
& | & \qquad FaultHandler_1 \downarrow \mathcal{L} \uparrow \mathcal{L}_1 ... FaultHandler_n \downarrow \mathcal{L} \uparrow \mathcal{L}_n) \\
& & \left\{ \mathcal{L}' \ = \ \mathcal{L}_1 \cup ... \cup \mathcal{L}_n \right. \\
& | & \textbf{Assign}\ (Var_0 \texttt{:=} From_0 ... Var_n \texttt{:=} From_n) \uparrow \emptyset \\
& | & \textbf{Throw}\ (\chi) \uparrow \emptyset \quad | \quad \textbf{ThrowData}\ (\chi, X) \uparrow \emptyset \quad | \quad \textbf{Rethrow}\ \uparrow \emptyset \\
& | & \textbf{Validate}\ (X_0 ... X_n) \uparrow \emptyset \\
& | & \textbf{Sequence}\ (Activity_0 \downarrow \mathcal{L} \uparrow \mathcal{L}_0 ... Activity_n \downarrow \mathcal{L} \uparrow \mathcal{L}_n) \\
& & \left\{ \mathcal{L}' \ = \ \mathcal{L}_0 \cup ... \cup \mathcal{L}_n \right.
\end{array}
$$

$$
\begin{array}{rcl}
 & | & \textbf{If}\ (Expr, Activity_1 \downarrow \mathcal{L} \uparrow \mathcal{L}_1, Activity_2 \downarrow \mathcal{L} \uparrow \mathcal{L}_2) \\
 & & \left\{ \mathcal{L}' \ = \ \mathcal{L}_1 \cup \mathcal{L}_2 \right. \\
 & | & \textbf{While}\ (Expr, Activity) \uparrow \emptyset \quad | \quad \textbf{RepeatUntil}\ (Expr, Activity) \uparrow \emptyset \\
 & | & \textbf{ForEach}\ (X, Expr_1, Expr_2, Activity) \uparrow \emptyset \\
 & | & \textbf{ForEachCond}\ (X, Expr_1, Expr_2, Expr_3, V^s, Activity) \uparrow \emptyset \\
 & | & \textbf{Pick}\ (EventHandler_0 \downarrow \mathcal{L} \uparrow \mathcal{L}_0 ... EventHandler_n \downarrow \mathcal{L} \uparrow \mathcal{L}_n) \\
 & & \left\{ \mathcal{L}' \ = \ \mathcal{L}_0 \cup ... \cup \mathcal{L}_n \right. \\
 & | & \textbf{Flow}\ (L_1...L_m, Activity_0 \downarrow \mathcal{L} \cup \{L_1...L_m\} \uparrow \mathcal{L}_0\ ... \\
 & | & \qquad Activity_n \downarrow \mathcal{L} \cup \{L_1...L_m\} \uparrow \mathcal{L}_n) \\
 & & \left\{ \mathcal{L}' \ = \ \mathcal{L}_0 \cup ... \cup \mathcal{L}_n \right. \\
 & | & \textbf{LinkedActivity}\ (L_1...L_m, Expr, Activity \downarrow \mathcal{L} \uparrow \mathcal{L}^a, L'_1, Expr_1 ... L'_n, Expr_n) \\
 & & \left\{ \mathcal{L}' \ = \ (\{L_1...L_m\} \backslash \mathcal{L}) \cup \mathcal{L}^a \right. \\
Def \downarrow \mathcal{L} \uparrow \mathcal{L}' & ::= & \textbf{Service}\ (S, \lambda_0 : \theta_0 ... \lambda_{n^p} : \theta_{n^p}, X_1 : T_1 ... X_{n^x} : T_{n^x}, \\
 & & \qquad FaultHandler_0 \downarrow \mathcal{L} \uparrow \mathcal{L}^f_0 ... FaultHandler_{n^f} \downarrow \mathcal{L} \uparrow \mathcal{L}^f_{n^f}, \\
 & & \qquad EventHandler_1 \downarrow \mathcal{L} \uparrow \mathcal{L}^e_1 ... EventHandler_{n^e} \downarrow \mathcal{L} \uparrow \mathcal{L}^e_{n^e}, \\
 & & \qquad Activity \downarrow \mathcal{L} \uparrow \mathcal{L}^a) \\
 & & \left\{ \mathcal{L}' \ = \ \mathcal{L}^f_0 \cup ... \cup \mathcal{L}^f_{n^f} \cup \mathcal{L}^e_1 \cup ... \cup \mathcal{L}^e_{n^e} \cup \mathcal{L}^a \right. \\
FaultHandler \downarrow \mathcal{L} \uparrow \mathcal{L}' & ::= & \textbf{CatchFault}\ (\chi, Activity \downarrow \mathcal{L} \uparrow \mathcal{L}') \\
 & & ... \\
EventHandler \downarrow \mathcal{L} \uparrow \mathcal{L}' & ::= & \textbf{OnEvent}\ (\lambda, F, M, X, Activity \downarrow \mathcal{L} \uparrow \mathcal{L}') \\
 & & ...
\end{array}
$$

Le calcul de $Activity.\mathcal{L}^O$ est défini au moyen d'une grammaire attribuée identique à celle présentée ci-dessus pour le calcul de $Activity.\mathcal{L}^I$, à l'exception de la règle suivante :

$$
\begin{array}{rcl}
 & | & \textbf{LinkedActivity}\ (L_1...L_m, Expr, Activity \downarrow \mathcal{L} \uparrow \mathcal{L}^a, L'_1, Expr_1 ... L'_n, Expr_n) \\
 & & \left\{ \mathcal{L}' \ = \ (\{L_1...L_m\} \backslash \mathcal{L}) \cup \mathcal{L}^a \right.
\end{array}
$$

qui devient :

$$
\begin{array}{rcl}
 & | & \textbf{LinkedActivity}\ (L_1...L_m, Expr, Activity \downarrow \mathcal{L} \uparrow \mathcal{L}^a, L'_1, Expr_1 ... L'_n, Expr_n) \\
 & & \left\{ \mathcal{L}' \ = \ (\{L'_1...L'_n\} \backslash \mathcal{L}) \cup \mathcal{L}^a \right.
\end{array}
$$

La traduction de l'activité "`flow`" est alors la suivante :

$[\![\textbf{Flow}\ (L_1...L_m, Activity_0...Activity_n)]\!]\ (f)\ =$

```
  hide Read_L1,Write_L1,...,Read_Lm,Write_Lm:LinkTypeChannel,exitLinkManager:Any in
    par exitLinkManager in
      Read_L1,Write_L1,...,Read_Lm,Write_Lm ->
        var B1,...,Bm:LinkType,N:Nat in
          B1 := unset; ...; Bm := unset;N := 0;
```

```
            loop L in
              select
                Read_L1 (!L1)
              []
                Write_L1 (?L1)
              []
                ...
              []
                Read_Lm (!Lm)
              []
                Write_Lm (?Lm)
              []
                exitLinkManager;
                N := N + 1;
                if N == n + 1 then
                  break L
                end if
              end select
            end loop
          end var
        [[Activity0]]^L -> [[Activity0]] (vrai); exitLinkManager
      ||
        ...
      ||
        [[Activityn]]^L -> [[Activityn]] (vrai); exitLinkManager
      end par
    end hide
```

Le comportement LOTOS NT résultant de la traduction de l'activité "`flow`" commence par déclarer deux portes par lien de contrôle L ("`Read_`L" pour lire la valeur du lien et "`Write_`L" pour l'écrire). Ces portes sont utilisées dans la déclaration d'un opérateur LOTOS NT "`par`" qui met en parallèle un gestionnaire de liens de contrôle et les traductions des différentes sous-activités de l'activité "`flow`". Le gestionnaire de liens de contrôle, d'une façon similaire au gestionnaire de variables présenté à la section 10.2.6, régule les accès aux valeurs des liens de contrôles grâce aux deux portes définies pour chaque lien. L'exécution d'un gestionnaire de liens de contrôle se termine lorsque tous les comportements LOTOS NT issus de la traduction des sous-activités de l'activité "`flow`" ont signalé leur terminaison au gestionnaire de liens au moyen de la porte "`exitLinkManager`". Le comportement LOTOS NT issu de la traduction d'une sous-activité de l'activité "`flow`" est une branche parallèle au gestionnaire de liens de contrôle et aux traductions des autres sous-activités de l'activité `flow`. Cette branche comporte deux parties :

1. la sélection des portes ($[\![Activity_i]\!]^L$) nécessaires à l'accès aux valeurs des liens de contrôle utilisés dans la sous-activité et sur lesquelles le comportement LOTOS NT traduisant cette sous-activité doit donc pouvoir se synchroniser,et

1. la traduction de la sous-activité suivie d'une synchronisation sur la porte "`exitLinkManager`".

La définition de $[\![Activity]\!]^L$ consiste à construire les ensembles de liens entrants et sortants de l'activité. Pour chaque lien entrant L^I (*resp.* lien sortant L^O), la porte "`Read_`L^I" (*resp.* "`Write_`L^O") est ajoutée à la liste des portes sur lesquelles le comportement LOTOS NT traduisant *Activity* doit se synchroniser avec le gestionnaire de liens de contrôle :

$[\![Activity]\!]^L$ =
  `Read_`L_1^I`,...,Read_`$L_{n^I}^I$`,Write_`L_1^O`,...,Write_`$L^O n^O$
  où
    $\{L_1^I ... L_{n^I}^I\}$ = $Activity.\mathcal{L}^I L_1 ... L_m$
    $\{L_1^O ... L_{n^O}^O\}$ = $Activity.\mathcal{L}^O L_1 ... L_m$

La traduction d'une activité munie de liens entrants et sortants *LinkedActivity* est la suivante :

$[\![$**LinkedActivity** $(L_1 ... L_m, Expr, Activity, L'_1, Expr_1 ... L'_n, Expr_n)]\!]$ (f) =
  `var` B_1`,...,`B_m`:LinkType,`L_1`,...,`L_m`:Bool in`
    B_1 `:= unset; ...;` B_m `:= unset;`
    `while` B_1 `== unset or ... or` B_m `== unset loop`
      `if` B_1 `== unset then`
        `Read_`L_1 `(?`B_1`)`
      `end if;`
      ...
      `if` B_m `== unset then`
        `Read_`L_m `(?`B_m`)`
      `end if;`
    `end while;`
    L_1 `:=` B_1`.value; ...;` L_m `:=` B_m`.value;`
    `if` $[\![Expr]\!]^D$ `then`
      $[\![Activity]\!]$ (f)`;`
      `var` X_1`:`T_1`,...,`X_{n^x}`:`T_{n^x} `in`
        `Read_`X_1 `(?`X_1`); ...; Read_`X_{n^x} `(?`X_{n^x}`);`
        `Write_`L'_1 `(!set (`$[\![Expr_1]\!]^D$`));`
        ...
        `Write_`L'_n `(!set (`$[\![Expr_n]\!]^D$`));`
      `end var`
    `else`
      ElseCode
    `end if`

Cette traduction consiste en trois parties distinctes. Premièrement, il s'agit d'évaluer la condition de démarrage (*Expr*) de l'activité. Comme cette condition est une expression booléenne sur les valeurs des liens entrants ($L_1 ... L_m$), il faut attendre que la valeur de chaque lien soit différente de la valeur indéfinie ("`unset`"). Ensuite, seulement, la condition de démarrage, traduite en LOTOS NT en suivant la traduction des expressions XPATH présentée à la section 8.6.4. Dans *Expr*, la valeur associée à chaque lien L est contenue dans une variable booléenne nommée L. Dans notre traduction des activités *LinkedActivity*, la valeur de chaque lien L est contenue dans une variable B de type "`LinkType`". Afin de rendre disponible, en LOTOS NT, la valeur de chaque lien L dans une variable booléenne nommée L, nous déclarons une telle variable L`:Bool` et lui affectons la valeur B`.value`.

Deuxièmement, il s'agit de traduire la sous-activité. Si la condition de démarrage est satisfaite, alors le comportement LOTOS NT issu de la traduction de la sous-activité ($[\![Activity]\!]$ (f)) est exécuté. Sinon, le comportement LOTOS NT à exécuter dépend de la valeur de l'attribut "`suppressJoinFailure`" dont la valeur courante pour chaque activité est dénotée $Activity.V^s$. Si la valeur de cet attribut est vraie, alors l'exception "`joinFailure`" doit être levée, sinon, les liens de contrôle sortant $L'_1 ... L'_n$ se voit associer la valeur "`faux`" :

```
ElseCode =
  si Activity.V^s = faux alors
    Write_L'_1 (!set (false));
    ...
    Write_L'_n (!set (false))
  sinon
    faultRaise_S (!joinFailure)
```

Troisièmement, après l'exécution du comportement LOTOS NT issu de la traduction de la sous-activité, les résultats des expressions $Expr_1...Expr_n$ à associer aux liens de sortie $L'_1...L'_n$ sont calculées. En LOTOS NT, chaque expression XPATH $Expr_i$ est traduite en une expression LOTOS NT "`set (`$[\![Expr_i]\!]^D$`)`" qui est passée en paramètre d'une communication sur la porte "`Write_`L'_i" pour modifier la valeur du lien L'_i. Soient $X_1...X_{n^x}$ les variables partagées figurant dans les expressions $Expr_1...Expr_n$.

10.2.26 Traduction des activités au sein de l'activité "flow"

La traduction en LOTOS NT des activités BPEL contenues au sein d'une activité "`flow`" reste inchangée, sauf pour les activités "`if`" et "`pick`" ainsi que pour les gestionnaires d'exceptions. En effet, la norme BPEL veut que si une sous-activité d'une activité "`if`" ou "`pick`" n'est pas exécutée (car il ne s'agit pas de la sous-activité choisie), alors chaque lien de contrôle sortant de cette activité se voit associer la valeur "`faux`", mais seulement après que les valeurs des liens entrants de cette sous-activité ont été déterminées. Ceci est aussi valable pour les gestionnaires d'exceptions qui ne sont pas exécutés (soit par ce qu'un autre gestionnaire d'exception du même service ou sous-service a été exécuté, soit parce qu'aucune exception n'a été levée) : les liens sortant des gestionnaires d'exceptions qui n'ont pas été sollicités se voient associer la valeur "`faux`".

Soient $L^I_1...L^I_m$ les liens entrants d'une activité et $L^O_1...L^O_n$ ses liens sortants, on dit que $L^O_1...L^O_n$ dépendent de $L^I_1...L^I_m$. Afin de donner formellement les traductions des activités "`if`" et "`pick`" ainsi que des gestionnaires d'exceptions au sein d'une activité "`flow`", nous avons besoin de définir le calcul des dépendances entres liens entrants et liens sortants au sein d'une activité. Il s'agit de rechercher, pour une activité a, les activités *LinkedActivity* comprises dans a, car ce sont les seules activités susceptibles de créer des dépendances entre liens d'entrée et liens de sortie de l'activité. Ces relations de dépendances sont calculées au moyen de la grammaire attribuée donnée ci-dessous. L'attribut $\mathcal{L}$ hérité contient l'ensemble des liens déclarés au sein de l'activité par des sous-activités "`flow`". En effet, ces liens doivent être ignorés car ils ne font partie ni des liens entrants, ni des liens sortants de l'activité. L'attribut $\mathcal{D}$ synthétisé contient l'ensemble des relations de dépendances entre liens entrants et liens sortants de l'activité. S'il existe $\mathcal{L}^I \rightarrow \mathcal{L}^O$ dans $\mathcal{D}$, alors les liens sortants contenus dans l'ensemble $\mathcal{L}^O$ dépendent des liens entrants contenus dans l'ensemble $\mathcal{L}^I$.

$$
\begin{array}{lcl}
\mathit{Activity} \downarrow \mathcal{L} \uparrow \mathcal{D} & ::= & \textbf{Scope}\ (\mathit{Def}) \downarrow \mathcal{L} \uparrow \mathcal{D} \quad | \quad \textbf{Empty}\ \uparrow \emptyset \\
& | & \textbf{Exit}\ \uparrow \emptyset \quad | \quad \textbf{WaitFor}\ (\mathit{Expr}) \uparrow \emptyset \\
& | & \textbf{WaitUntil}\ (\mathit{Expr}) \uparrow \emptyset \quad | \quad \textbf{Receive}\ (\lambda, F, X) \uparrow \emptyset \\
& | & \textbf{Reply}\ (\lambda, F, X) \uparrow \emptyset \quad | \quad \textbf{ReplyFault}\ (\lambda, F, \chi) \uparrow \emptyset \\
& | & \textbf{ReplyFaultData}\ (\lambda, F, \chi, X) \uparrow \emptyset \\
& | & \textbf{InvokeOneWay}\ (\lambda, F, X_I) \uparrow \emptyset \\
& | & \textbf{InvokeTwoWays}\ (\lambda, F, X_I, X_O, \\
& & \qquad \mathit{FaultHandler}_1 \downarrow \mathcal{L} \uparrow \mathcal{D} ... \mathit{FaultHandler}_n \downarrow \mathcal{L} \uparrow \mathcal{D}) \\
& & \Big\{ \mathcal{D} \ = \ \mathcal{D}_1 \cup ... \cup \mathcal{D}_n
\end{array}
$$

$$
\begin{array}{rcl}
 & | & \textbf{Assign}\ (\mathit{Var}_0\texttt{:=}\mathit{From}_0, ..., \mathit{Var}_n\texttt{:=}\mathit{From}_n) \uparrow \emptyset \\
 & | & \textbf{Throw}\ (\chi) \uparrow \emptyset \quad | \quad \textbf{ThrowData}\ (\chi, X) \uparrow \emptyset \quad | \quad \mathit{wRethrow} \uparrow \emptyset \\
 & | & \textbf{Validate}\ (X_0...X_n) \uparrow \emptyset \\
 & | & \textbf{Sequence}\ (\mathit{Activity}_0 \downarrow \mathcal{L} \uparrow \mathcal{D}_0 ... \mathit{Activity}_n \downarrow \mathcal{L} \uparrow \mathcal{D}_n) \\
 & & \left\{ \mathcal{D} \ = \ \mathcal{D}_0 \cup ... \cup \mathcal{D}_n \right. \\
 & | & \textbf{If}\ (\mathit{Expr}, \mathit{Activity}_1 \downarrow \mathcal{L} \uparrow \mathcal{D}_1, \mathit{Activity}_2 \downarrow \mathcal{L} \uparrow \mathcal{D}_2) \\
 & & \left\{ \mathcal{D} \ = \ \mathcal{D}_1 \cup \mathcal{D}_2 \right. \\
 & | & \textbf{While}\ (\mathit{Expr}, \mathit{Activity}) \uparrow \emptyset \\
 & | & \textbf{RepeatUntil}\ (\mathit{Expr}, \mathit{Activity}) \uparrow \emptyset \\
 & | & \textbf{ForEach}\ (X, \mathit{Expr}_1, \mathit{Expr}_2, \mathit{Activity}) \uparrow \emptyset \\
 & | & \textbf{ForEachCond}\ (X, \mathit{Expr}_1, \mathit{Expr}_2, \mathit{Expr}_3, V^s, \mathit{Activity}) \uparrow \emptyset \\
 & | & \textbf{Pick}\ (\mathit{EventHandler}_0 \downarrow \mathcal{L} \uparrow \mathcal{D}_0 ... \mathit{EventHandler}_n \downarrow \mathcal{L} \uparrow \mathcal{D}_n) \\
 & & \left\{ \mathcal{D} \ = \ \mathcal{D}_0 \cup ... \cup \mathcal{D}_n \right. \\
 & | & \textbf{Flow}\ (L_1...L_m, \mathit{Activity}_0 \downarrow \mathcal{L}' \uparrow \mathcal{D}_0 ... \mathit{Activity}_n \downarrow \mathcal{L}' \uparrow \mathcal{D}_n) \\
 & & \begin{cases} \mathcal{L}' \ = \ \mathcal{L} \cup \{L_1...L_m\} \\ \mathcal{D}' \ = \ \mathcal{D}_0 \cup ... \cup \mathcal{D}_n \end{cases} \\
 & | & \textbf{LinkedActivity}\ (L_1...L_m, \mathit{Expr}, \mathit{Activity} \downarrow \mathcal{L} \uparrow \mathcal{D}^a, L'_1, \mathit{Expr}_1 ... L'_n, \mathit{Expr}_n) \\
 & & \begin{cases} \mathcal{D} \ = \ \{L \mid L \in \{L_1...L_m\} \ \wedge \ L \notin \mathcal{L}\} \\ \qquad\qquad \rightarrow \ \{L \mid L \in \{L'_1...L'_n\} \ \wedge \ L \notin \mathcal{L}\} \ \cup \ \mathcal{D}^a \end{cases} \\
\mathit{Def} \downarrow \mathcal{L} \uparrow \mathcal{D} & ::= & \textbf{Service}\ (S, \lambda_0\texttt{:}\theta_0 ... \lambda_{n^p}\texttt{:}\theta_{n^p}, X_1\texttt{:}T_1 ... X_{n^x}\texttt{:}T_{n^x} \\
 & & \quad \mathit{FaultHandler}_0 \downarrow \mathcal{L} \uparrow \mathcal{D}^f_0 ... \mathit{FaultHandler}_{n^f} \downarrow \mathcal{L} \uparrow \mathcal{D}^f_{n^f}, \\
 & & \quad \mathit{EventHandler}_1 ... \mathit{EventHandler}_{n^e}, \mathit{Activity} \downarrow \mathcal{L} \uparrow \mathcal{D}^a) \\
 & & \left\{ \mathcal{D} \ = \ \mathcal{D}^f_0 \cup ... \cup \mathcal{D}^f_{n^f} \cup \mathcal{D}^a \right. \\
\mathit{FaultHandler} \downarrow \mathcal{L} \uparrow \mathcal{D} & ::= & \textbf{CatchFault}\ (\chi, \mathit{Activity} \downarrow \mathcal{L} \uparrow \mathcal{D}) \\
 & & ... \\
\mathit{EventHandler} \downarrow \mathcal{L} \uparrow \mathcal{D} & ::= & \textbf{OnEvent}\ (\lambda, F, M, X, \mathit{Activity} \downarrow \mathcal{L} \uparrow \mathcal{D}) \\
 & & ...
\end{array}
$$

Soit $\mathit{Activity}.\mathcal{D}$ l'ensemble des relations de dépendance entre liens d'une activité BPEL.

En plus du calcul des dépendances d'une activité, il est nécessaire d'introduire le code LOTOS NT généré pour traiter les dépendances ($\mathit{Activity}.\mathcal{D}$) au sein d'une activité ($\mathit{Activity}$), avant de donner la traduction des activités "`if`" et "`pick`", ainsi que des gestionnaires d'exceptions. $\mathit{Activity}.\mathcal{D}$ comprend un ensemble de relations de dépendances entre liens entrants et sortants de l'activité $\mathit{Activity}$: $\mathcal{L}^I_1 \rightarrow \mathcal{L}^O_1 ... \mathcal{L}^I_n \rightarrow \mathcal{L}^O_n$. Pour chaque relation de dépendance $\mathcal{L}^I \rightarrow \mathcal{L}^O$, il faut attendre que les valeurs des liens dans $\mathcal{L}^I$ soient déterminées avant de mettre les valeurs de tous les liens de $\mathcal{L}^O$ à "`faux`". Nous notons $[\![\mathit{Activity}]\!]^{Dep}$ le code LOTOS NT généré pour traiter les relations de dépendances de $\mathit{Activity}$. $[\![\mathit{Activity}]\!]^{Dep}$ est formellement définie ainsi :

$$
\begin{array}{l}
[\![\mathit{Activity}]\!]^{Dep} \ = \\
\quad [\![\mathcal{L}^I_1 \rightarrow \mathcal{L}^O_1]\!]^{Dep} \\
\quad ... \\
\quad [\![\mathcal{L}^I_n \rightarrow \mathcal{L}^O_n]\!]^{Dep} \\
\quad \texttt{où} \\
\qquad \mathcal{L}^I_1 \rightarrow \mathcal{L}^O_1 ... \mathcal{L}^I_n \rightarrow \mathcal{L}^O_n \ = \ \mathit{Activity}.\mathcal{D}
\end{array}
$$

$[\![\{L_1^I ... L_m^I\} \rightarrow \{L_1^O ... L_n^O\}]\!]^{Dep}$ =
  si $m \neq 0 \ \wedge \ n \neq 0$ alors
    var B_1,...,B_m:Bool in
      B_1 := unset; ...; B_m := unset;
      while B_1 == unset and ... and B_m==unset loop
        Read_L_1^I (?B_1); ...; Read_L_m^I (?B_m)
      end loop
    end var;
  fin si
  si $n \neq 0$ alors
    Write_L_1^O (!set (false));
    ...
    Write_L_n^O (!set (false));
  fin si

La traduction d'une activité "`if`", au sein d'une activité "`flow`", doit associer la valeur "`faux`" aux liens sortants de la sous-activité qui n'est pas exécutée (selon la condition de branchement). Cette association ne doit se faire qu'après que les valeurs des liens entrants de l'activité, dont les liens sortants dépendent, ont été déterminées. Cette traduction est donc très proche de celle présentée à la section 10.2.20 et diffère simplement par l'ajout, après la traduction d'une sous-activité, de la traduction des relations de dépendances de l'autre sous-activité :

$[\![$**If** $(Expr, Activity_1, Activity_2)]\!]$ (vrai) =
  var X_1:T_1,...,X_n:T_n in
    Read_X_1 (?X_1);
    ...;
    Read_X_n (?X_n);
    if $[\![Expr]\!]^D$ then
      $[\![Activity_1]\!]$ (vrai);
      $[\![Activity_2]\!]^{Dep}$
    else
      $[\![Activity_2]\!]$ (vrai);
      $[\![Activity_1]\!]^{Dep}$
    end if
  end var

La traduction d'une activité "`pick`", au sein d'une activité "`flow`", doit associer la valeur "`faux`" aux liens sortants des sous-activités qui ne sont pas exécutées (selon la branche choisie). Cette association ne doit se faire qu'après que les valeurs des liens entrants de l'activité, dont les liens sortants dépendent, ont été déterminées. Comme pour l'activité "`if`", cette traduction est très proche de celle présentée à la section 10.2.24 et diffère simplement par l'ajout, après la traduction d'une sous-activité, des traductions des relations de dépendances des autres sous-activités ($Activity_i$ désigne la sous-activité contenue à l'intérieur du gestionnaire d'événement $EventHandler_i$) :

$[\![\textbf{Pick}\ (\mathit{EventHandler}_0 ... \mathit{EventHandler}_n]\!]\ (\texttt{vrai})\ =$
```
  select
    [[EventHandler_0]]^Pick (vrai);
    [[Activity_1]]^Dep;
    ...
    [[Activity_n]]^Dep;
  []
    ...
  []
    [[EventHandler_n]]^Pick (vrai);
    [[Activity_0]]^Dep;
    ...
    [[Activity_n-1]]^Dep;
  end select
```

Un gestionnaire d'exception ne peut figurer qu'au sein d'une déclaration de service :

$$\begin{array}{rcl} \mathit{Def} & ::= & \textbf{Service}\ (S, \lambda_1 : \theta_1 ... \lambda_{n^p} : \theta_{n^p}, X_1 : T_1 ... X_{n^x} : T_{n^x} \\ & & \quad \mathit{FaultHandler}_0 ... \mathit{FaultHandler}_{n^f}, \\ & & \quad \mathit{EventHandler}_1 ... \mathit{EventHandler}_{n^e}, \mathit{Activity}) \end{array}$$

La traduction des gestionnaires d'exceptions d'un sous-service, au sein d'une activité "`flow`", diffère en deux points de la traduction présentée à la section 10.2.7 :

- Chaque gestionnaire d'exception doit associer la valeur "`faux`" aux liens sortants des autres gestionnaires d'exceptions qui ne sont pas exécutés.
- Si l'activité du service (*Activity*) lève une exception, alors il faut associer la valeur "`faux`" à tous les liens sortants de l'activité par un appel à $[\![\mathit{Activity}]\!]^{Dep}$. "`faux`".

L'association d'une valeur avec un lien sortant L de l'activité ou d'un gestionnaire d'exception ne peut se faire qu'après que les valeurs des liens entrants de l'activité, dont L dépend, ont été déterminées.

Finalement, la traduction des gestionnaires d'exceptions, au sein d'une activité "`flow`", est formellement définie comme suit ($\mathit{Activity}_i$ désigne la sous-activité du gestionnaire d'exception $\mathit{FaultHandler}_i$) :

$[\![\mathit{Def}]\!]^F\ (\texttt{vrai})\ =$
```
  hide faultHandle_S,faultRaise_S:Channel_Exceptions,exitFaultRaise_S:any in
    par exitFaultRaise,faultHandle_S,faultRaise_S in
      disrupt
        [[Def]]^E (faux); exitFaultRaise_S
      by
        var F:Exceptions in
          faultHandle_S (?F);
          case F in
```

```
          var X_{m^f+1}:M_{m^f+1},...,X_{n^f-1}:M_{n^f-1} in
            χ_0 ->
              [[Activity_0]] (vrai);
              [[Activity_1]]^Dep
              ...
              [[Activity_{n^f}]]^Dep
          | ...
          | exitFault -> [[exitFault]]
          | any ->
            [[Activity_{n^f}]];
            [[Activity_0]]^Dep
            ...
            [[Activity_{n^f-1}]]^Dep
          end case
        end var;
        [[Activity]]^Dep
      end disrupt
    ||
      select
        exitFaultRaise
      []
        var F:Exceptions in
          faultRaise_S (?F); faultHandle_S (!F)
        end var
      end select
    end par
  end hide
```

Chapitre 11

Conception d'un traducteur automatique

Afin de mettre en œuvre la traduction que nous avons définie dans les chapitre précédents, nous avons commencé le développement d'un traducteur automatique dans le but de générer une spécification LOTOS NT à partir d'une spécification BPEL. Ce traducteur doit pouvoir :

- lire un fichier contenant la définition d'un service BPEL ;
- construire un arbre de syntaxe abstrait représentant le service BPEL, les définitions WSDL et XML Schema, ainsi que les expressions XPATH ;
- générer le code LOTOS NT correspondant à ces définitions.

11.1 Choix du langage de programmation

A l'origine, nous avions l'intention de construire ce traducteur sur les méta-modèles EMF des langages XML Schema, WSDL et BPEL et de créer un méta-modèle pour XPATH. Mais, nous avons été confrontés à plusieurs difficultés :

- Le méta-modèle "officiel" de XML Schema[57] est extrêmement complexe par rapport à la syntaxe abstraite que nous avons présentée. Cela vient du fait que ce méta-modèle comprend toutes les constructions relevant du sucre syntaxique. De plus, la liaison d'éléments d'un méta-modèle dans un autre, qui est nécessaire, notamment pour représenter les types des variables BPEL, est une fonctionnalité très peu documentée de EMF.
- Les outils de transformation de méta-modèles (comme ATL et Kermeta), ainsi que les outils de génération de code (comme XTEXT et ACCELEO), ne sont pas suffisamment évolués, comme nous l'avons vu à la section 1.2, pour traiter une traduction aussi complexe que celle que nous avons définie entre BPEL et LOTOS NT. De plus, la production d'exécutables binaires à partir des transformations définies dans ces outils est peu ou pas documentée.
- L'utilisation des méta-modèles EMF pour XML Schema, WSDL, et BPEL obligerait à inclure dans l'exécutable binaire du traducteur, toutes les bibliothèques nécessaires au fonctionnement

[57] `http://www.eclipse.org/xsd`

d'Eclipse et d'Emf, afin d'ouvrir et de manipuler les méta-modèles. En pratique, cela entraînerait la production d'un binaire exécutable de plusieurs dizaines de méga-octets, qui est long à compiler et difficile à distribuer.

- Les technologies de méta-modélisation sont récentes et très spécifiques. Cela nuit à la pérennité du traducteur car les outils associés à ces technologies peuvent disparaître soudainement (comme cela s'est produit pour Smart Qvt).

Par conséquent, nous nous sommes tournés vers le développement d'un traducteur écrit uniquement en Java. Nous avons vu, dans ce langage, les avantages suivants :

- Il existe des bibliothèques Java pour traiter les fichiers au format Xml, ce qui permet de créer des analyseurs lexicaux et syntaxiques pour Bpel, Xml Schema et Wsdl à moindre coût, s'ils n'existent pas déjà.
- La bibliothèque Jaxen[58] pour Java fournit un analyseur syntaxique et un arbre de syntaxe abstrait sous la forme de classes Java pour XPath.
- Java est un langage mature qui est largement diffusé, ce qui permet d'envisager un cycle de vie long pour tous les programmes écrits dans ce langage ; de plus, il est plus aisé de trouver de bons développeurs spécialisés en Java qu'en techniques de méta-modélisation.
- Un programme Java peut être distribué comme une application seule, ce qui nous permet de nous affranchir de la lourde infrastructure Eclipse.

11.2 Réalisation

Pour XPath, qui n'est pas un langage Xml, nous avons utilisé Jaxen, un analyseur syntaxique écrit en Java pour le langage XPath. Nous avons directement réutilisé l'arbre de syntaxe abstrait fournit par Jaxen.

Pour Xml Schema, Wsdl et Bpel, qui sont des langages à syntaxe Xml, nous avons écrit, à l'aide de classes Java, les arbres de syntaxe abstraits. Ces arbres de syntaxe correspondent aux grammaires que nous avons présentées dans les chapitres précédents. La construction des analyseurs syntaxiques s'est faite à l'aide de Sax (*Simple* Api Xml), l'analyseur syntaxique Xml de Java. Cet outil fournit un analyseur syntaxique abstrait, c'est-à-dire pour lequel certaines portions du code généré pour l'analyseur sont laissées vides et doivent être complétées manuellement. Grâce à Sax, nous avons implémenté très rapidement, pour le langage Xml, un analyseur syntaxique qui nous permet de créer un arbre de syntaxe abstrait contenant une arborescence d'éléments et d'attributs correspondant à un fichier Xml Schema, Wsdl ou Bpel. Ensuite, nous avons créé un algorithme pour transformer un arbre de syntaxe Xml en arbre de syntaxe abstrait Xml Schema, Wsdl ou Bpel, selon le fichier analysé. Cet algorithme est très simple car il suffit, pour chaque élément Xml, d'instancier la construction correspondant au nom de l'élément, dans le langage visé.

Une fois l'analyse syntaxique du service Bpel, des définitions Xml Schema et Wsdl, et des expressions XPath, effectuée, le traducteur lie les identificateurs, avant de lancer les différents algorithmes de traduction.

A l'heure actuelle, la traduction des types Xml Schema et des définitions Wsdl est entièrement implémentée dans le traducteur. Cela représente environ 4 000 lignes de code Java. Il nous reste à finir d'implémenter la traduction des langages Bpel et XPath.

[58] `http://www.jaxen.org`

11.3 Espaces de noms

Une difficulté, pour les langages à syntaxe Xml, réside dans la gestion des espaces de noms. Ces espaces de noms sont des conteneurs d'identificateurs et sont eux-mêmes nommés de façon à faciliter l'attribution de noms uniques aux entités (variables, types...) déclarées. Chaque espace de nom est associé à un préfixe. Un préfixe est une chaîne de caractères qui est souvent très courte et qui sert à identifier localement un espace de nom (dont le nom est souvent très long).

Ainsi, l'espace de noms du langage Xml Schema est nommé "`http://www.w3.org/2001/XMLSchema`" et est souvent associé au préfixe "`xsd`". C'est pourquoi les constructions Xml Schema sont souvent précédées de "`xsd:`", comme par exemple dans `<xsd:element>`. La déclaration d'une entité avec pour nom "p`:`N" a pour effet d'insérer, dans l'espace de noms N_s associé au préfixe p, le nom N. Par la suite, toutes les références à cette entité doivent avoir la forme suivante : "p'`:`N" où p' est le préfixe localement associé à N.

Cette notion de "localité" est importante car, comme nous l'avons vu à la section 8.1, chaque élément Xml peut redéfinir l'association entre un espace de noms et son préfixe. Nous devons donc constamment maintenant à jour une table de correspondance entre un espace de noms et son préfixe, afin d'associer les identificateurs des entités déclarées au bon espace de noms.

Partie IV

Conclusion

Bilan et perspectives

Ma thèse s'est effectuée en co-tutelle entre l'équipe-projet VASY du centre INRIA de Grenoble Rhône-Alpes, sous la direction de Hubert Garavel et en connexion avec le projet TOPCASED, et l'équipe de Valentin Cristea à l'Université Polytechnique de Bucarest.

L'objectif de la thèse était d'explorer les possibilités de connexion entre de nouvelles techniques de modélisation pour les systèmes asynchrones et les boîtes à outils de vérification formelle. Nous nous sommes intéressés, en particulier, aux langages issus de, ou compatibles avec l'approche MDE.

Dans les sections suivantes, nous détaillons les conclusions auxquelles nous sommes parvenus sur les quatre thématiques abordées (qui figurent parmi les centres d'intérêts des deux équipes dans lesquelles la thèse s'est déroulée) :

- la vérification de systèmes GALS,
- la vérification de services Web BPEL,
- l'ingénierie dirigée par les modèles et
- l'algèbre de processus LOTOS NT.

Vérification de systèmes GALS

Dans la première partie de cette thèse, nous avons proposé une approche simple et élégante pour la modélisation et l'analyse des systèmes comprenant des composants synchrones qui interagissent de façon asynchrone et que l'on appelle couramment GALS (*Globally Asynchronous Locally Synchronous*). Cette approche consiste à encoder des programmes synchrones dans des fonctions et types d'algèbres de processus, à encapsuler ces fonctions et types dans des processus asynchrones et à faire communiquer les processus asynchrones ainsi obtenus. Selon les langages synchrones et les algèbres de processus considérés, plusieurs encodages sont possibles :

- transformer les programmes synchrones en machines de Mealy et encoder ces machines de Mealy en types et fonctions dans l'algèbre de processus considérée,
- transformer la représentation intermédiaire du compilateur du langage synchrone visé, afin d'obtenir rapidement un encodage du programme synchrone sous forme de types et fonctions dans l'algèbre de processus considérée, ou bien
- interfacer les fonctions C générées par les compilateurs de certains langages synchrones (ESTEREL par exemple) avec l'algèbre de processus considérée, si cela est possible.

Contrairement aux autres approches qui étendent le paradigme synchrone pour modéliser l'asynchronisme, notre approche préserve la sémantique originale des langages synchrones, ainsi que la

sémantique asynchrone des algèbres de processus. Notre approche permet la réutilisation, sans modification, des outils existants pour les langages synchrones considérés. En particulier, les différents composants synchrones peuvent être vérifiés séparément, à l'aide des outils de vérification des langages synchrones avant que soit entreprise la vérification de leurs interactions asynchrones.

Nous avons démontré la faisabilité de cette approche sur une étude de cas industrielle, une variante du protocole TFTP/UDP, fournie par Airbus, dont le bon comportement a été vérifié et les performances évaluées, au moyen de la plateforme TOPCASED et des outils de CADP. Dans cette étude de cas, deux entités de protocole communicantes étaient spécifiées dans le langage synchrone SAM d'Airbus. Ces deux entités ont été traduites en fonctions et types de données LOTOS NT avant d'être connectées l'une à l'autre par l'intermédiaire de processus LOTOS NT communiquant de façon asynchrone. Notre étude a révélé 19 erreurs qui ont été reconnues comme telles par les ingénieurs d'Airbus. Pour ces 19 erreurs, nous avons pu montré que si certaines entraînent une dégradation négligeable des performances, d'autres, en revanche, détériorent très sensiblement la vitesse de transfert des données entre les deux entités. Les retours industriels ont été encourageants et notre étude de cas est depuis considérée comme une contribution importante au projet TOPCASED[59].

Concernant les perspectives, nous pensons que notre approche n'est pas limitée au langage synchrone SAM et aux algèbres de processus LOTOS et LOTOS NT, et qu'elle peut être généralisée à d'autres langages synchrones dont le compilateur est capable de traduire des programmes synchrones en machines de Mealy (ce qui est normalement toujours le cas) et à toute algèbre de processus qui permet le parallélisme asynchrone et la définition de types et fonctions. En particulier, il faudrait tenter de mélanger, au sein de la même spécification asynchrone, plusieurs composants synchrones définis avec des langages différents. Il serait également intéressant de comparer les résultats de simulation obtenus avec CADP et les résultats fournis par des outils de simulation plus traditionnels, tels que Möbius [Wil98].

Depuis la publication de ces travaux, l'équipe VASY a continué de collaborer avec Airbus sur la vérification d'autres systèmes embarqués avioniques, tels qu'un protocole de gestion des erreurs des équipements (*equipment-failure management protocol*) utilisé dans les avions Airbus. Nous espérons également trouver d'autres partenaires industriels intéressés par l'application de notre approche à leurs problèmes.

Vérification de services Web BPEL

Dans la seconde partie de cette thèse, nous avons présenté un algorithme complet pour la transformation de modèles de services Web BPEL en processus LOTOS NT qui peuvent alors être formellement vérifiés à l'aide des outils de CADP. En comparaison avec les approches de transformation existantes, la nôtre se démarque par les spécificités suivantes :

- Nous avons constitué une base d'exemples de services BPEL afin de déterminer les constructions réellement utilisées des langages BPEL, WSDL, XPATH et XML Schema que notre algorithme de traduction doit traiter. Les exemples de la base ont été obtenus de plusieurs manières : en contactant les auteurs de certains travaux que nous avons mentionnés à la section 6.7, en téléchargeant les outils BPEL librement accessibles sur Internet, dans l'espoir de trouver des fichiers BPEL, WSDL et XML Schema parmi la documentation ou les bases de tests de ces outils, et enfin en effectuant des recherches à l'aide de moteurs de recherche (Google, Yahoo!, Bing...) avec des mots-clés extraits des langages BPEL et WSDL, tels que `partnerLink`, `flow`, `faultHandler`, `portType`...

[59] `http://gforge.enseeiht.fr/docman/view.php/52/3627/TOPCASED-presentation-2h.pdf`

- Nous avons défini un algorithme de transformation des types XML Schema (utilisés pour encoder le contenu des variables et des messages d'un service BPEL) vers LOTOS NT. Cet algorithme est efficace : il produit des types LOTOS NT et C optimisés afin de réduire les quantités de mémoire et de temps nécessaires à la génération de l'espace d'états de la spécification. En outre, cet algorithme est quasiment complet puisque, à l'exception des constructions `list` et `union`, il traite l'ensemble du langage XML Schema.

- Nous avons défini un algorithme de transformation des expressions XPATH (utilisées par BPEL dans les constructions manipulant des données) en expressions LOTOS NT équivalentes. Cet algorithme ne prend en compte que les constructions de XPATH qui apparaissent dans notre base d'exemples. Cette démarche nous a permis d'éviter l'erreur faite dans [Fu04], où le sous-ensemble de XPATH considéré comprend des constructions non utilisées dans les services Web mais ignore, dans le même temps, des constructions utiles, telles que les constantes, les variables, de nombreux axes et les fonctions prédéfinies. De plus, grâce au sous-ensemble limité (mais pertinent) de XPATH que nous traitons, nous avons pu mettre en œuvre une traduction vers LOTOS NT optimisée, du point de vue du temps d'évaluation des expressions LOTOS NT produites.

- Nous avons défini un algorithme de transformation des déclarations WSDL en LOTOS NT (ce qu'aucune autre approche ne propose) afin que les actions de communication présentes dans l'espace d'états d'un service Web BPEL puissent être facilement mises en correspondance avec les actions de communication qui se produisent au sein du service.

- Nous avons défini un algorithme de transformation des services BPEL qui prend en compte une grande partie du langage. A la différence des approches existantes, nous avons soigneusement justifié les raisons qui nous ont poussé à ne pas traiter certaines constructions. Pour les constructions que nous avons choisi de traiter, la sémantique de leur transformation en LOTOS NT est formellement définie.

Notre algorithme de traduction est plus complet que ceux existant dans la littérature scientifique. Il sert actuellement de fondation à un prototype de traducteur automatisé dans lequel les traductions de XML Schema et WSDL vers LOTOS NT sont entièrement implémentées. Les traductions de XPATH et BPEL ont été commencées.

Concernant les perspectives, le développement du prototype de traducteur pourrait être continué par un ingénieur ou un étudiant stagiaire en implémentant les règles de traduction définies dans la thèse, jusqu'à l'obtention d'un traducteur complet. Un traducteur opérationnel permettrait de déterminer la taille des STEs correspondant à nos exemples, ce qui indiquerait s'il est nécessaire ou non de formuler des abstractions sur ces exemples afin de les vérifier formellement.

Si l'on souhaitait traiter un sous-ensemble plus large de BPEL, voici quelques indications sur la manière de traiter certaines constructions que nous avons choisi d'omettre :

- On pourrait mieux modéliser les activités liées au temps dans l'opérateur de choix (`pick`). Par exemple, si cet opérateur donne le choix entre deux *alarmes* : T_1 et T_2 suivies, respectivement, des activités a_1 et a_2 et s'il peut être établi statiquement que T_1 se produit forcément avant T_2, alors il serait intéressant de préserver cette relation d'ordre en LOTOS NT, de façon à ne jamais exécuter a_2 avant a_1. Pour ce faire, nous pensons qu'il serait judicieux d'utiliser un gestionnaire de temps, qui signalerait les événements T_1 et T_2 sur une porte de communication dédiée, en s'assurant que T_1 est toujours précédée par T_2.

- La traduction de l'opérateur `forEach` pourrait être améliorée pour prendre en compte l'attribut `parallel` lorsque les bornes de l'itération peuvent être statiquement déterminées. Il suffirait

alors d'écrire en LOTOS NT un comportement parallèle comportant autant de branches qu'il y a d'itérations dans la boucle.

- Un gestionnaire de terminaison permet en BPEL d'associer à chaque sous-service S une activité de secours qui sera exécutée si S est soudainement interrompu (typiquement, lorsqu'une activité parallèle à S et définie dans le sous-service S', parent de S, a déclenché une exception). La traduction en LOTOS NT de cet opérateur est difficile, car il faut pouvoir détecter qu'un sous-service a été interrompu. Une façon de traiter cela en LOTOS NT serait d'avoir un gestionnaire "global" de terminaison qui s'exécuterait en parallèle du service BPEL. Lorsqu'un sous-service se terminerait normalement, il notifierait cette terminaison au gestionnaire de terminaison global qui "marquerait" le sous-service comme "terminé". Lorsqu'une exception serait capturée dans le gestionnaire d'exceptions d'un service ou sous-service S, le gestionnaire de terminaison en serait informé et ordonnerait à chaque sous-service S' de S qui n'est pas marqué comme "terminé" d'exécuter les instructions LOTOS NT correspondants à l'activité BPEL définie par le gestionnaire de terminaison de S'. En LOTOS NT, un tel mécanisme devrait être implémenté à l'aide de synchronisation sur des portes de communication dédiées.
- Les gestionnaires de compensation de BPEL permettent d'annuler (dans la mesure du possible) l'exécution d'un sous-service. Nous pourrions modéliser cette construction avec un gestionnaire "global" de compensation en LOTOS NT, qui regrouperait les gestionnaires de compensation de tous les sous-services. Lorsqu'un sous-service terminerait son exécution, il se signalerait auprès du gestionnaire de compensation (qui activerait donc le gestionnaire de compensation particulier de ce sous-service) et lui communiquerait, au moyen de synchronisation sur une porte de communication, les valeurs de toutes les variables non locales au sous-service. Ensuite, lorsqu'un gestionnaire d'exception souhaiterait exécuter le gestionnaire de compensation, il lui suffirait de le demander auprès du gestionnaire global de compensation.

Ingénierie dirigée par les modèles

L'ingénirie dirigée par les modèles séduit le monde industriel car elle propose des outils pour définir facilement de nouveaux langages en produisant automatiquement des éditeurs et des traducteurs pour ces langages. Malheureusement, cette approche est encore trop récente et ne permet pas le développement rapide d'outils de vérification formelle pour les nouveaux langages. La solution actuelle à ce problème est la création de passerelles entre les langages de modélisation (comme SAM ou BPEL) et les langages formels (comme LOTOS NT). Des environnements de développement tels que Eclipse fournissent toute la mécanique logicielle pour construire ces passerelles.

En étudiant les deux langages dédiés BPEL et SAM, qui sont tous deux intégrés à Eclipse, nous avons intensivement utilisé cet environnement de développement ainsi que les outils d'ingénierie dirigée par les modèles qu'il fournit. Nos expériences nous ont permis de tirer une conclusion qui nous paraît importante. A l'heure actuelle, les techniques de méta-modélisation se prêtent particulièrement bien à des langages simples, comme SAM, pour lesquels la création d'un éditeur graphique ou d'un générateur de code peut s'effectuer en quelques minutes. En revanche, pour des langages plus complexes, comme BPEL, ces techniques montrent leurs limites et l'utilisation d'un langage de programmation complet, comme Java, est préférable, comme nous l'avons éprouvé en développant un traducteur de BPEL en LOTOS NT.

Pourtant, il ne s'agit pas du seul du problème. En effet, les environnements d'ingénierie dirigée par les modèles, tels que EMF dans Eclipse, ne sont pas du tout adaptés aux contraintes de la vérification formelle, un domaine dans lequel l'optimisation des outils (*temps d'exécution* et *consommation mémoire*) est primordiale. En particulier, ces environnements ne permettent pas de

passer à l'échelle. Si nous avions voulu obtenir une chaîne de compilation complète dans Eclipse des spécifications Bpel jusqu'aux Ltss, il aurait fallu transposer les outils de Cadp dans Eclipse. Or, si le service Bpel à transformer produit un espace d'états de plusieurs millions, voire plusieurs milliards, d'états, alors la quantité de mémoire disponible est primordiale et les centaines (voire des milliers) de méga-octets de mémoire vive consommés par Eclipse risquent de faire cruellement défaut. Même si Eclipse et Emf permettent l'invocation de transformations par le biais d'un interpréteur de commandes (et donc de transférer ces transformations sur un serveur de calcul dédié), il n'est pas envisageable d'effectuer des calculs intensifs dans un environnement si gourmand en ressources matérielles (capacité de calcul du processeur et consommation mémoire). Il est donc nécessaire d'effectuer la génération de l'espace d'états dans un environnement minimal, c'est-à-dire en dehors d'Eclipse. Mais, quitte à effectuer une partie du travail dans Eclipse (les aspects graphiques et interactifs), et une autre partie du travail dans un interpréteur de commande (vérification formelle), il nous parait plus judicieux d'effectuer tout le travail dans l'environnement adapté à la vérification formelle, la partie la plus cruciale.

La complexité des environnements d'ingénierie dirigée par les modèles est donc à la fois un avantage, car ces environnements permettent de rapidement mettre en place des outils pour des langages dédiés, mais aussi un inconvénient car cette complexité se traduit par un gaspillage de la mémoire et des capacités de calcul qui vont à l'encontre des contraintes de la vérification formelle pour laquelle l'utilisation de chaque ressource disponible doit être optimisée.

Concernant les perspectives, nous pensons qu'il serait interessant de mieux intégrer les outils de Cadp à Eclipse afin de construire une interface graphique pour Cadp. Malgré l'incompatibilité entre Eclipse et le calcul intensif, nous pensons que cela permettrait de rendre Cadp plus accessible et attrayant à un grand nombre d'utilisateurs qui sont, jusqu'à présent, rebutés par le lancement des outils en mode console ou bien par l'interface graphique (Xeuca) vieillissante de Cadp.

LOTOS NT

Au début de cette thèse, l'utilisation de Lotos NT était réduite à la construction de compilateurs [GLM02a] au sein de l'équipe Vasy de l'Inria et le traducteur Lnt2Lotos ne traitait qu'un sous-ensemble des types et fonctions de Lotos NT. L'utilisation intensive de Lotos NT durant cette thèse et les nombreuses suggestions et rapports d'erreurs effectués (environ 50) ont fortement contribué à amener le traducteur Lnt2Lotos dans son état de stabilité actuel.

Cette thèse a démontré la pertinence du langage Lotos NT pour le traitement d'études de cas complexes ou bien comme langage cible de traductions depuis d'autres formalismes. Ce langage est maintenant suffisamment mûr pour être utilisé en dehors de l'équipe Vasy et de ses partenaires tels que Bull. Dans ce but, Lotos NT a été ajouté à Cadp en janvier 2010.

Bibliographie

[AFMDR04] Loredana. Afanasiev, Massimo Franceschet, Maarten Marx, and Maarten De Rijke. CTL model checking for processing simple XPath queries. In *11th International Symposium on Temporal Representation and Reasoning, 2004. TIME 2004. Proceedings*, pages 117–124, 2004.

[AH76] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. *American Mathematical Society*, 82(5), 1976.

[And03] Charles André. Semantics of SSM (Safe State Machine). *Esterel Technologies*, 2003.

[BBC+08] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The Coq proof assistant reference manual. *INRIA, version*, 8, 2008.

[BBDV03] Jean Bézivin, Erwan Breton, Grégoire Dupé, and Patrick Valduriez. ATL: a Transformation-based Model Management Framework. Technical Report RR.03.08, Institut de Recherche en Informatique de Nantes, Nantes, France, September 2003.

[BBF+08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frédéric Lang, and François Vernadat. FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In Jean-Claude Laprie, editor, *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08 (Toulouse, France)*. SIA (the French Society of Automobile Engineers), AAAF (the French Society of Aeronautic and Aerospace), and SEE (the French Society for Electricity, Electronics, and Information & Communication Technologies), January 2008.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BCG+02] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A Protocol for Loosely Time-Triggered Architectures. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 252–265, London, UK, 2002. Springer-Verlag.

[BFRW01] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL — a model for W3C XML schema. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, New York, NY, USA, 2001. ACM.

[BG92] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BG05] Luciano Baresi and Sam Guinea. Towards dynamic monitoring of WS-BPEL processes. *Service-Oriented Computing-ICSOC 2005*, pages 269–282, 2005.

[BGS07] Domenico Bianculli, Carlo Ghezzi, and Paola Spoletini. A Model Checking Approach to Verify BPEL4WS Workflows. In *SOCA '07: Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications*, pages 13–20, Washington, DC, USA, 2007. IEEE Computer Society.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BLJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous Programming with Events and Relations: The SIGNAL Language and Its Semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[BMPS08] Frank Budinsky, Ed Merks, Marcelo Paternostro, and Dave Steinberg. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley Professional, second edition, February 2008.

[BMR06] Loic Besnard, Hervé Marchand, and Eric Rutten. The Sigali tool box environment. In *2006 8th International Workshop on Discrete Event Systems*, pages 465–466, 2006.

[Bou98] Amar Bouali. XEVE, an ESTEREL verification environment. In *Computer Aided Verification*, pages 500–504. Springer, 1998.

[Bra89] R. Braden. Requirements for Internet Hosts - Application and Support. RFC 1123, Internet Engineering Task Force, October 1989.

[BRS93] Gérard Berry, S. Ramesh, and R. K. Shyamasundar. Communicating Reactive Processes. In *POPL'93*, pages 85–98, New York, NY, USA, 1993. ACM.

[BRV04] Bernard Berthomieu, Pierre-Olivier Ribet, and François Vernadat. The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[Bry86] Randy E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BS01] Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *CHARME'01*, pages 110–125, London, UK, 2001. Springer-Verlag.

[BT07] Clark Barrett and Cesare Tinelli. Cvc3. In *Proceedings of the 19th international conference on Computer aided verification*, pages 298–302. Springer-Verlag, 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. *Programming Languages and Systems*, pages 21–30, 2005.

[CCG+10] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.0). INRIA/VASY, 107 pages, March 2010.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: a New Symbolic Model Checker. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, April 2000.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. In *10th Annual Symposium on Principles of Programming Languages*. ACM, 1983.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CFR+91] Ron Cytron, Jeanna Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):490, 1991.

[CGT08] Xavier Clerc, Hubert Garavel, and Damien Thivolle. Présentation du langage SAM d'Airbus. Internal Report, INRIA/VASY, 2008. Available from TOPCASED forge: `http://gforge.enseeiht.fr/docman/view.php/33/2745/SAM.pdf`.

[Cha84] Daniel Marcos Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. Doctoral Thesis, Stanford University, Department of Computer Science, October 2984.

[CL11] Pepijn Crouzen and Frédéric Lang. Smart Reduction. *Fundamental Approaches to Software Engineering*, pages 111–126, 2011.

[CLS00] Rance Cleaveland, Tan Li, and Steve Sims. The Concurrency Workbench of the New Century (Version 1.2). User's manual, July 2000.

[CMSW99] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal Design of Distributed Control Systems with Lustre. *Computer Safety, Reliability and Security*, pages 687–687, 1999.

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[Com01] OASIS Web Services Business Process Execution Language Technical Commitee. Web Services Business Process Execution Language. International Standards, Advancing open standards for the information society (OASIS), 2001.

[Com07] OASIS UDDI Specifications Technical Commitee. Universal Description, Discovery and Integration. International Standards, Advancing open standards for the information society (OASIS), 2007.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[DDM06] Bruno Dutertre and Leonardo De Moura. *The YICES SMT solver*, 2006.

[DFFV06] Zo Drey, Cyril Faucher, Franck Fleurey, and Didier Vojtisek. *Kermeta language reference manual*, 2006.

[DH03] Matthew B. Dwyer and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European Software Engineering Conference*, pages 267–276. ACM, 2003.

[DMK+06] Frederic Doucet, Massimiliano Menarini, Ingolf H. Krüger, Rajesh K. Gupta, and Jean-Pierre Talpin. A Verification Approach for GALS Integration of Synchronous Components. *Electr. Notes Theor. Comput. Sci.*, 146(2):105–131, 2006.

[DPC+09] Philippe Dhaussy, Pierres-Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation. *Model Driven Engineering Languages and Systems*, pages 438–452, 2009.

[eldG07] IBM et l'Université de Géorgie. Web Service Semantics. W3C Submission, World Wide Web Consortium (W3C), 2007.

[ES09] C. Eisentraut and D. Spieler. Fault, Compensation and Termination in WS-BPEL 2.0?A Comparative Analysis. *Web Services and Formal Methods*, pages 107–126, 2009.

[Eur05] Tni Europe. Sildex: a formal approach to real-time applications development. Technical Report, 2005.

[FBS04a] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM.

[FBS04b] Xiang Fu, Tevfik Bultan, and Jianwen Su. Model checking XML manipulating software. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 252–262, New York, NY, USA, 2004. ACM.

[FFK05] Jesús Arias Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Applying model checking to BPEL4WS business collaborations. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 826–830, New York, NY, USA, 2005. ACM.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[Fos06] Howard Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, Imperial College London, 2006.

[Fu04] Xiang Fu. *Formal Specification and Verification of Asynchronously Communicating Web Services*. PhD thesis, University of California, Santa Barbara, 2004.

[FUMK03] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *Proc. ASE*, volume 3, pages 152–161, 2003.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar98] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[Gar03] Hubert Garavel. Défense et illustration des algèbres de processus. In Zoubir Mammeri, editor, *Actes de l'Ecole d'été Temps Réel ETR 2003 (Toulouse, France)*. Institut de Recherche en Informatique de Toulouse, September 2003.

[Gar08] Hubert Garavel. Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular. In Catuscia Palamidessi and Frank D. Valencia, editors, *Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory (Ecole Polytechnique de Paris, France), November 13–15, 2006*, volume 209 of *Electronic Notes in Theoretical Computer Science*, pages 149–164. Elsevier Science Publishers, April 2008. Also available as INRIA Research Report RR-6368.

[GH02] Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In *FME'02*, volume 2391 of *LNCS*, pages 410–429, Copenhagen, Denmark, July 2002. Springer-Verlag.

[GJM+97] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. CADP'97 – Status, Applications and Perspectives. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997.

[GL01] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[GLM02a] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler Construction using LOTOS NT. In Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction CC 2002 (Grenoble, France)*, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer Verlag, April 2002.

[GLM02b] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

[GLMS07] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, July 2007.

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In Parosh A. Abdulla and

K. Rustan M. Leino, editors, *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2011 (Saarbrücken, Germany)*, Lecture Notes in Computer Science. Springer Verlag, March 2011.

[GM02] Alain Girault and Clément Ménier. Automatic Production of Globally Asynchronous Locally Synchronous Systems. In *EMSOFT '02*, pages 266–281, London, UK, 2002. Springer-Verlag.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[Gro00] XML Protocol Working Group. Simple Object Access Protocol. International Standards, World Wide Web Consortium (W3C), 2000.

[Gro03a] W3C XML Linking Working Group. XPointer Framework. International Standards, World Wide Web Consortium (W3C), 2003.

[Gro03b] XSL Working Group. XSL Transformations. International Standards, World Wide Web Consortium (W3C), 2003.

[Gro04a] Web Services Choregraphy Working Group. Web Services Choregraphy Description Language Version 1.0. International Standards, World Wide Web Consortium (W3C), 2004.

[Gro04b] XML Core Working Group. Extensible Markup Language 1.1 – second edition. International Standards, World Wide Web Consortium (W3C), 2004.

[Gro04c] XML Core Working Group. XML Information Set. International Standards, World Wide Web Consortium (W3C), 2004.

[Gro04d] XML Schema Working Group. XML Schema Definition Language 1.0 Part 1: Structures. International Standards, World Wide Web Consortium (W3C), 2004.

[Gro04e] XML Schema Working Group. XML Schema Definition Language 1.0 Part 2: Datatypes. International Standards, World Wide Web Consortium (W3C), 2004.

[Gro07] W3C XML Query Working Group. XQuery 1.0: An XML Query Language. International Standards, World Wide Web Consortium (W3C), 2007.

[Gro09] XML Core Working Group. Namespaces in XML – third edition. International Standards, World Wide Web Consortium (W3C), 2009.

[GT93] Hubert Garavel and Philippe Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'93 (Montréal, Canada)*, 1993.

[GT09] Hubert Garavel and Damien Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In Corina Pasareanu, editor, *Model Checking Software, Proceedings of the 16th International SPIN Workshop on Model Checking of Software SPIN'2009 (Grenoble, France)*, Lecture Notes in Computer Science. Springer Verlag, June 2009.

[GV90] Jan Friso Groote and Frits Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In M. S. Patterson, editor, *Proceedings of the 17th ICALP (Warwick)*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer Verlag, 1990.

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[Hal94] Nicolas Halbwachs. About synchronous programming and abstract interpretation. *Static Analysis*, pages 179–192, 1994.

[HB02] Nicolas Halbwachs and Siwar Baghdadi. Synchronous Modelling of Asynchronous Systems. In *EMSOFT '02*, pages 240–251, London, UK, 2002. Springer-Verlag.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The Synchronous Dataflow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):793, 1992.

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HM06] Nicolas Halbwachs and Louis Mandel. Simulation and Verification of Asynchronous Systems by Means of a Synchronous Model. In *ACSD '06*, pages 3–14, Washington, DC, USA, 2006. IEEE Computer Society.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.

[Hol04] Gerard J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.

[HP88] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing Company, Incorporated, 1988.

[HR99] Nicolas Halbwachs and Pascal Raymond. Validation of Synchronous Reactive Systems: from Formal Verification to Automatic Testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), December 1999. LNCS 1742, Springer Verlag.

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri nets. *Lecture Notes in Computer Science*, 3649:220, 2005.

[HT08] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, page 15. IEEE Press, 2008.

[ISO89] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[ISO01] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[JK06] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1188–1195, New York, NY, USA, 2006. ACM.

[JL07] Marcin Jurdzinski and Ranko Lazic. Alternation-free modal mu-calculus for data trees. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 131–140. IEEE, 2007.

[Joh75] Stephen C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[Ken02] Stuart Kent. Model driven engineering. In *Integrated Formal Methods*, pages 286–298. Springer, 2002.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[KP96] Michael Kantor and Arati Prabhakar. ELECTRONIC DATA INTERCHANGE (EDI). International Standards, National Institute of Standards and Technology, 1996.

[KQCM09] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc Programming Language. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *Proceedings of FMOODS/FORTE 2009*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009.

[KvB04] Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[Ley01] Frank Leymann. Web Services Flow Language (WSFL 1.0). Technical Report, IBM, 2001.

[LMP98] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. *Languages and Compilers for Parallel Computing*, pages 114–130, 1998.

[LMZF09] Xitong Li, Stuart Madnick, Hongwei Zhu, and Yushun Fan. Reconciling Semantic Heterogeneity in Web Services Composition. In *The 30th International Conference on Information Systems ICIS '09 (Phoenix, Arizona, USA)*, 2009.

[Loh08] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. *Lecture Notes in Computer Science*, 4937:77, 2008.

[Ltd05] Formal Systems (Europe) Ltd. Failure-Divergence Refinement. FRD2 User Manual, June 2005.

[LTL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for System Design. *Journal of Circuits, Systems and Computers. World Scientific*, 12, 2003.

[LVO+07] N. Lohmann, E. Verbeek, C. Ouyang, C. Stahl, and W.M.P. van der Aalst. Comparing and evaluating Petri net semantics for BPEL. *Computer Science Report*, 7:23, 2007.

[Mar04] James Margetson. Proving the Completeness Theorem within Isabelle/HOL. 2004.

[Mat06] Radu Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 8(1):37–56, February 2006. Full version available as INRIA Research Report RR-5948, July 2006.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MG10] Radu Mateescu and Salaün Gwen. Translating π-calculus into LOTOS NT. In *IFM'08: Proceedings of the 8th International Conference on Integrated Formal Methods*, 2010.

[MGLZ07] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. JOLIE: a Java Orchestration Language Intepreter Engine. *Electronic Notes in Theoretical Computer Science*, 181:19–33, 2007.

[Mic08] Microsoft. Distributed Component Object Model (DCOM) Remote Protocol Specification. International Standards, Microsoft, October 2008.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil83] Robin Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006 edition, April 2006.

[MLT+04] Mohammad Reza Mousavi, Paul Le Guernic, Jean-Pierre Talpin, Sandeep Kumar Shukla, and Twan Basten. Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks. In *DATE '04*, page 10384, Washington, DC, USA, 2004. IEEE Computer Society.

[MMG+07] Simon Moser, Axel Martens, Katharina Görlach, Wolfram Amme, and Artur Godlinski. Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In *IEEE International Conference on Services Computing, 2007. SCC 2007*, pages 98–105, 2007.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume I (Specification). Springer Verlag, 1992.

[MR01] Florence Maraninchi and Yann Rémond. Argos: an Automaton-Based Synchronous Language. *Computer Languages*, 27(1–3):61–92, October 2001.

[MR08] Radu Mateescu and Sylvain Rampacek. Formal Modeling and Discrete-Time Analysis of BPEL Web Services. In Joseph Barjis, Murali Mohan Narasipuram, and Peter Rittgen, editors, *Proceedings of the 4th International Workshop on Enterprise and Organizational Modeling and Simulation EOMAS'08 (Montpellier, France)*, volume 10 of *Lecture Notes in Business Information Processing*, pages 179–193. Springer Verlag, June 2008.

[MRLBS01] Hervé Marchand, Eric Rutten, Michel Le Borgne, and Mazen Samaan. Formal verification of programs specified with signal: Application to a power transformer station controller. *Science of Computer Programming*, 41(1):85–104, 2001.

[MS03] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

[MT08] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland)*, number 5014 in Lecture Notes in Computer Science, pages 148–164. Springer Verlag, May 2008.

[MWC10] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[Nak06] Shin Nakajima. Model-checking behavioral specification of BPEL applications. *Electronic Notes in Theoretical Computer Science*, 151(2):89–105, 2006.

[NH84] Rocco De Nicola and Matthew C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[NP06] Layaïda Nabil and Genevès Pierre. Mu-Calculus Based Resolution of XPath Decision Problems. Technical Report, INRIA, 2006.

[NVS+06] Meenakshi Nagarajan, Kunal Verma, Amit P. Sheth, John Miller, and Jon Lathem. Semantic Interoperability of Web Services - Challenges and Experiences. In *Proceedings of the Fourth IEEE International Conference on Web Services (ICWS 2006)*, pages 373–382. IEEE CS Press, 2006.

[OMG97] OMG. Unified Modeling Language, version 1.1. International Standards, Object Management Group, November 1997.

[OMG01] OMG. Model Driven Architecture - A Technical Perspective. International Standards, Object Management Group, July 2001.

[OMG05] OMG. XMI Mapping Specification, version 2.1. International Standards, Object Management Group, September 2005.

[OMG06a] OMG. CORBA Component Model, v4.0. International Standards, Object Management Group, April 2006.

[OMG06b] OMG. Meta Object Facility Core Specification, version 2.0. International Standards, Object Management Group, January 2006.

[OMG07a] OMG. MOF Query / Views / Transformations. International Standards, Object Management Group, July 2007.

[OMG07b] OMG. Unified Modeling Language, version 2.1.1. International Standards, Object Management Group, February 2007.

[OMG08a] OMG. MOF Model to Text Transformation Language, version 1.0. International Standards, Object Management Group, January 2008.

[OMG08b] OMG. Systems Modeling Language, version 1.1. International Standards, Object Management Group, November 2008.

[OMG10] OMG. Object Query Language, version 2.2. International Standards, Object Management Group, January 2010.

[OVvdA+07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[PBC07] Dumitru Potop-Butucaru and Benoît Caillaud. Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. *Fundam. Inf.*, 78(1):131–159, 2007.

[PBM+93] Jean-Pierre Paris, Gérard Berry, Frédéric Mignard, Philippe Couronné, Paul Caspi, Nicolas Halbwachs, Yves Sorel, Albert Benveniste, Thierry Gautier, Paul Le Guernic, François Dupont, and Claude Le Maire. Projet SYNCHRONE: les formats communs des langages synchrones. Technical Report 157, IRISA, 1993.

[PKPS02] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. *The Semantic Web?ISWC 2002*, pages 333–347, 2002.

[PL00] Paul Pettersson and Kim G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70(40-44):2, 2000.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[PS03] Massimo Paolucci and Katia Sycara. Autonomous semantic web services. *IEEE Internet Computing*, 7(5):34–41, 2003.

[QXW+07] Yi Qian, Yuming Xu, Zheng Wang, Geguang Pu, Huibiao Zhu, and Chao Cai. Tool support for BPEL verification in activebpel engine. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 90–100, 2007.

[Ram98] S. Ramesh. Communicating Reactive State Machines: Design, Model and Implementation. *IFAC Workshop on Distributed Computer Control Systems*, September 1998.

[RNHW02] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 200–209. IEEE, 2002.

[Ros08] A.W. Roscoe. On the expressiveness of Csp. 2008. Draft of October 23, 2008.

[RSD+04] S. Ramesh, Sampada Sonalkar, Vijay D'Silva, Naveen Chandra, and B. Vijayalakshmi. A Toolset for Modelling and Verification of GALS Systems. In *CAV '04*, volume 3114 of *LNCS*, pages 506–509. Springer-Verlag, 2004.

[SAE09] SAE. Architecture Analysis and Design Language. International Standards, Society of Automotive Engineers, January 2009.

[SBS04] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 43, Washington, DC, USA, 2004. IEEE Computer Society.

[SBS06] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.

[SCG+08] Mihaela Sighireanu, Champelovier David Catry, Alban, Hubert Garavel, Frédéric Lang, Guillaume Schaefferm, Wendelin Serwe, and Jan Stöcker. LOTOS NT User's Manual (Version 2.6). INRIA projet VASY. `ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, February 2008.

[Sig99] Mihaela Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS"*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1999.

[Sig00] Mihaela Sighireanu. LOTOS NT User's Manual (Version 2.1). INRIA projet VASY. `ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, November 2000.

[Sol92] K. Sollins. The TFTP Protocol (Revision 2). RFC 1350, Internet Engineering Task Force, July 1992.

[Tha01] Satish Thatte. XLANG: Web Services for Business Process Design. Technical Report, Microsoft, 2001.

[TNP+] Andres Toom, Tonu Naks, Marc Pantel, Marcel Gandriau, and Indra Wati. GeneAuto: An Automatic Code Generator for a safe subset of SimuLink/StateFlow. In *Dans: European Congress on Embedded Real-Time Software (ERTS 2008).–Toulouse*, pages 08–01.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.

[VdADH+05] Wil M.P. Van der Aalst, Marlon Dumas, Artur H.M. Hofstede, Nick Russell, Eric Verbeek, and Petia Wohed. Life After BPEL? *Formal Techniques for Computer Systems and Business Processes*, pages 35–50, 2005.

[VdAVH04] Wil M.P. Van der Aalst and Kees M. Van Hee. *Workflow management: models, methods, and systems*. The MIT press, 2004.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):36, 2000.

[W3C99] W3C/MIT. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, 1999.

[W3C01] W3C. Web Services Description Language (WSDL) 1.1. International Standards, World Wide Web Consortium (W3C), 2001.

[Whi76] James E. White. A High-Level Framework for Network-Based Resource Sharing. RFC 707, Internet Engineering Task Force, January 1976.

[Wil98] Alex Lee Williamson. *Discrete event simulation in the Möbius modeling framework*. PhD thesis, 1998.

[XG99] XSL and XML Linking Working Groups. XML Path Language (XPath). International Standards, World Wide Web Consortium (W3C), 1999.

[XG07] XSL and XML Query Working Groups. XML Path Language (XPath) 2.0. International Standards, World Wide Web Consortium (W3C), 2007.

[Yeu06] Wing-lok Yeung. Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services. In *ECOWS '06: Proceedings of the European Conference on Web Services*, pages 297–305, Washington, DC, USA, 2006. IEEE Computer Society.