République Algérienne Démocratique et Populaire

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ MENTOURI DE CONSTANTINE

FACULTÉ DES SCIENCES DE L'INGÉNIEUR

DÉPARTEMENT D'INFORMATIQUE

Laboratoire Informatique Répartie (LIRE)

N° d'ordre : ...

Série : …

M E M O I R E

Pour obtenir le diplôme de :

Magister en Informatique
Option : Information & Computation

Thème

Intégration de XML dans le cadre de la logique de réécriture

Présenté par :

M^me^ DOUIBI HALIMA

Dirigé par :

Dr. F. BELALA Maître de Conférence – Université Mentouri Constantine

Soutenu le : …/…/2006, devant le Juré composé de :

- Président : Pr. Z. BOUFAIDA Professeur – Université Mentouri Constantine
- Rapporteur : Dr. F. BELALA Maître de Conférence – Université Mentouri Constantine
- Examinateur : Dr. S. MESHOUL Maître de Conférence – Université Mentouri Constantine
- Examinateur : Dr. N. ZAROUR Maître de Conférence – Université Mentouri Constantine

Résumé

Les méthodes formelles permettent de raisonner rigoureusement sur les programmes informatiques, afin de démontrer leur correction, en se basant sur des raisonnements de logique mathématique. La logique de réécriture est une de ces méthodes,ayant plusieurs applications. En effet, différents formalismes destinés à la modélisation des systèmes concurrents (réseaux de pétri, systèmes de transitions étiquetés, etc) ont été intégrés dans cette logique unificatrice. De plus, elle a aussi une portée assez large sur les systèmes orientés objets, les langages de programmation, les générateurs de programmes, etc.

Dans ce contexte, notre projet vise la formalisation d'un langage universel, largement utilisé par la communauté se trouvant autour du Web: XML (eXtensible Markup Langage) successeur de HTML. Nous suggérons une nouvelle application de la logique de réécriture via son langage Maude, en attribuant une sémantique formelle au langage d'échange de données XML. Ce langage est considéré comme un format standard des documents pour l'écriture et l'échange de l'information sur le Web. Il est utilisé dans toutes les applications liées au Web, qui importent, exportent, manipulent, stockent ou transmettent des données. XML donne la possibilité à l'utilisateur de créer ses propres balises et de les structurer selon une syntaxe bien définie.

Cependant, le langage XML souffre de limites liées aux contraintes impliquées par le type d'applications nouvelles du Web. La limite majeure de XML est sa sémantique qui est implicitement exprimée dans les documents. D'autre part, XML n'est pas approprié pour représenter les services dans la génération suivante du Web (Web sémantique).

L'objectif de notre travail est double, en plus de la proposition d'une nouvelle application de la logique de réécriture, nous donnons une spécification formelle des documents XML ainsi que leur grammaire. Cette modélisation a défini formellement la structure générique d'une classe de documents XML, ainsi que de validation d'un document XML par rapport à son schéma XML.

Mots Clés: Document XML, Schéma XML, Logique de Réécriture, Système Maude.

Abstract

Formal methods allow arguing rigorously about computer programs, to demonstrate their correction. The rewriting logic is one of these methods, which has several applications. In fact, several formalisms intended for the modelling of concurrent systems (Petri net, labelled transitions systems, etc.) were integrated into this unifying logic. Furthermore, it has also a rather wide reach on oriented objects systems, programming languages, generators of programs, etc.

In this context, our work aims at the formalization of a universal language, widely used in the Web: XML (eXtensible Markup Language) successor of HTML. We suggest a new application of the rewriting logic and its Maude language, attributing formal semantics to the language of data exchange XML. This language is considered as a standard for information writing and exchanging on the Web. It is used in all the Web applications, which export, manipulate, store or pass data. XML gives possibility to the user to create his own tags and to structure them according to a defined syntax.

However, language XML suffers from some limits related to constraints implied by the new Web applications. The main limit of XML is its poor semantics, which is implicitly expressed in documents. On the other hand, XML is not suited to represent services in the following Web generation (Web semantic).

The objective of our work is double, in addition to the proposition of a new rewriting logic application, we give a formal specification of XML documents as well as their grammar. This model defines the generic structure of an XML documents class, and also the validation process of a XML document regarding its XML schema.

Key Words: XML Document, XML Schema, Rewriting logic, Maude system.

Introduction Générale

Contexte et motivations

Un point crucial de la démarche de spécification-développement-vérification d'un logiciel est l'élaboration d'une structure formelle qui résume sa sémantique. Le second point important de cette démarche est la validation (automatique) de la structure. Si elle est valide, elle constitue une documentation concise et précise du logiciel.
En pratique, une méthode formelle permet d'atteindre des exigences de qualité élevées pour le logiciel, en raisonnant rigoureusement sur les programmes informatiques qui lui sont associés.

Les méthodes formelles permettent d'obtenir une très forte assurance de l'absence d'erreurs dans les logiciels. Cependant, elles sont généralement coûteuses en ressources (humaines et matérielles) et actuellement réservées aux logiciels les plus critiques. Leur amélioration et l'élargissement de leurs champs d'application pratique sont la motivation de nombreuses recherches scientifiques en informatique.
Les méthodes formelles peuvent s'appliquer à différent stade du processus de développement d'un système (logiciel, électronique, mixte), de la spécification jusqu'à la réalisation finale. Elles sont appelées à jouer un rôle fondamental pour l'introduction d'une sémantique rigoureuse.
Entre autres, leurs points forts sont aussi :

- Permettre l'expression non ambiguë des besoins de l'utilisateur.
- Lorsque la spécification est transformable en code exécutable, on obtient un prototype. L'application de méthodes formelles peut guider le processus de test.
- Disposer de la spécification formelle de composants logiciels est un atout pour la réutilisation.

Différents corpus mathématiques ont été utilisés pour élaborer des raisonnements formels sur les logiciels. Cependant, il n'existe pas de méthode universelle pour modéliser formellement un système informatique.
Plusieurs «familles» de méthodes formelles sont alors engendrées. Citons notamment celles dites:

- basées sur un modèle : VDM, Z, B.
- basées algèbres de processus : CSP, CCS, μ-calcul.
- algébriques : Larch, CASL.
- mixtes : Lotos, RAISE.

Sachant qu'il existe des gradations et des mélanges possibles entre ces méthodes, *la logique de réécriture*, inventée par José Meseguer [Mes 92], est considérée comme un formalisme unifié de concurrence dans laquelle plusieurs modèles bien connus des systèmes concurrents sont intégrés (CCS, μ-calcul, réseaux de Petri, etc) [VM 00, Tha 02]. Elle constitue une conséquence des travaux de Meseguer et ses coéquipiers sur les logiques générales pour décrire les systèmes informatiques et particulièrement les systèmes concurrents.

Le calcul dans cette logique prend la forme de règles de réécriture dans une syntaxe donnée. Une réécriture d'un terme consiste à le remplacer par un terme équivalent, conformément aux lois d'algèbre de termes. Cette logique formalise donc le processus de réécriture pour calculer une relation de réécrivabilité entre les termes algébriques.

En fait, elle permet de raisonner sur les changements complexes des systèmes concurrents qui correspondent aux actions atomiques axiomatisées par les règles de réécriture. Le calcul dans ces systèmes est exprimé à travers la déduction logique qui est intrinsèquement concurrente [MM 99, MM 02]. De plus, les spécifications des systèmes dans cette logique de réécriture sont exécutables. Une implémentation d'un langage supportant les concepts de la logique de réécriture est réalisée à travers plusieurs environnements opérationnels : Maude au USA, CafeOBJ au Japon et ELAN en France [MM 02].

Maude est un système performant construit autour du langage déclaratif dit : "Maude" aussi. Les programmes Maude sont des théories de réécriture et les calculs concurrents dans Maude représentent des déductions dans la logique de réécriture.

Le but du projet de Maude est d'élargir le spectre d'utilisation de la programmation déclarative et des méthodes formelles pour spécifier et réaliser des systèmes de haute qualité dans plusieurs secteurs comme : le génie logiciel, réseaux de communication, l'informatique répartie, bioinformatique, et le développement formel d'outils.

Le système Maude [Clav 99 a, Olv 00] possède une collection d'outils formels supportant différentes formes de raisonnement logique pour vérifier des propriétés des programmes comprenant :

- Un *Model-Checker* pour vérifier les propriétés temporelles linéaires de la logique (LTL) des modules d'états finis ;
- Un *Prouveur de Théorèmes* ("Theorem Prover") (ITP) : pour vérifier des propriétés des modules fonctionnels;
- Un *Analyseur Church-Rosser* permettant d'assurer une telle propriété pour les modules fonctionnels;
- Un *Analyseur de terminaison et un outil de Complétion* ("Knuth-Bendix"): pour les modules fonctionnels aussi ;
- Un *Analyseur de Cohérence* : pour les modules systèmes.

Maude présente des caractéristiques très importantes comme la simplicité, la puissance d'expression, multi-paradigmes et bien d'autres.

La théorie et les applications de logique de réécriture sont vigoureusement développées par des chercheurs dans le monde entier. La bibliographie attachée inclut plus de trois cents papiers publiés. Plusieurs ateliers internationaux sur la logique de réécriture ont été tenus aux Etats-Unis, la France, le Japon, l'Espagne, l'Italie, et dernièrement en Autriche [MM 02]. En outre, plusieurs implémentations de langages et une variété d'outils formels ont aussi été développées et employées dans une grande collection d'applications. En effet, elle a été utilisée comme cadre unificateur de plusieurs modèles de la concurrence: systèmes de transitions étiquetés, réseaux de Petri, ECATNets, CCS, machine chimique abstraite, les structures d'évènements. Elle a aussi une portée assez large sur les systèmes orientés objets, les langages de programmation, les générateurs de programmes et d'outils, etc [Bou 98, Ram 98, MM 93, MM 95, BM 93, Mes 92, Mes 03, VM 02].

Le travail que nous réalisons se situe dans ce contexte, il vise l'utilisation de cette logique pour fournir une sémantique formelle au langage XML, tout en faisant abstraction des problèmes liés à l'implémentation des documents structurés décrits et les grammaires les générant. La logique de réécriture permet une focalisation sur les aspects logiques et

conceptuels des documents manipulés. Par ailleurs, cette méthode convient bien, de par sa nature, à un prototypage logique des différentes données définies. De plus, elle permet la construction de preuves que ce soit pour contrôler la validité des documents manipulés ou pour vérifier l'obtention des résultats escomptés.

L'utilisation d'Internet a considérablement augmenté ces dernières années. Cette croissance est associée à la naissance de deux standards :

- HyperText Markup Language (HTML) : cette spécification décrit un langage associant des données et la manière de présenter ces données
- HyperText Transfer Protocol (HTTP) : cette spécification décrit un protocole de transport d'informations sur internet.

Contrairement au HTTP, le HTML est un standard qui a beaucoup évolué depuis son apparition. A l'origine, dans un document HTML, il n'y avait pas de prédominance entre les données et la présentation de ces données, un document était constitué à part égale entre les données et la présentation. Rapidement les normes successives de HTML ont laissé la présentation prendre de plus en plus de place dans un document HTML (actuellement dans un document HTML, 95% du contenu est constitué d'éléments de présentation). Il est donc apparu nécessaire de définir un nouveau langage qui serait centré sur la description des données : eXtensible Markup Language (XML).

XML a réalisé un grand succès comme un format standard des documents pour l'écriture et l'échange de l'information sur le Web [Dub 01]. Ce langage représente une révolution réelle dans le panorama des langages de publication sur le Web. Actuellement nous estimons que plusieurs centaines de langages basés sur XML ont été décrits [GC 01], nous pouvons citer quatre catégories de ces langages :

- Les langages XML pour les relations entreprise client « Business to Customer » (B2C) : RDF, XHTML, XSLT, XSL, etc.
- Les langages XML pour les services web : SOAP, WSDL, etc.
- Les langages XML pour l'intégration d'applications d'entreprise « Enterprise Application Integration » (EAI) : XSL, SOAP, etc.
- Les langages XML pour les relations inter entreprise « Business to Business » (B2B) : cXML, cCBL, XAML, ebXML, etc.

XML est alors un langage permettant de décrire, à l'aide de balises personnalisées, la structure de données ou documents de tous types.

Toutes les applications qui importent, exportent, manipulent, stockent ou transmettent des données l'ont adopté comme standard. Cela inclut bien sur toutes les applications liées au Web (sites e-commerce, portails intranet ou extranet...), mais aussi les bases de données, les logiciels de bureautique ou de graphisme, les progiciels de gestion, les outils de CRM (gestion de la relation client) ou de « knowledge management », les applications mobiles (WAP), etc.

Problématique

La technologie XML n'est pas limitée à la génération de pages Web (HTML, WAP) ou d'autres documents plus traditionnels. Par sa capacité de représenter des données très diverses, XML a rapidement été promu comme format universel pour l'échange de données entre différentes applications informatiques.
Or, les applications actuelles du Web sémantique, qui tente à prolonger le Web actuel, utilisent des pages munies d'une représentation, non seulement structurelle sémantisée (RDF/RDFS, DAML+OIL, OWL) [Ber 00], pour répondre à des requêtes sur la signification de l'information manipulée, mais aussi de nature fonctionnelle, pour représenter les composants actifs qui y agissent (WSDL, WSMF, DAML-S).

Actuellement, le langage XML souffre de limites liées aux contraintes impliquées par le type d'applications nouvelles du Web.

D'une part, la partie sémantique de XML est très pauvre, seulement un être humain peut comprendre la signification des éléments d'un document XML [Cov 98], En fait, la sémantique XML est implicitement exprimée dans les documents XML. D'autre part, XML n'est pas approprié pour représenter les services dans la génération suivante du Web (Web sémantique [Ber 01]) permettant d'invoquer, surveiller ou composer des ressources offrant tels services et vérifiant telles propriétés.

Dans la littérature, la sémantique de XML a été considérée sous plusieurs angles. Une communauté de chercheurs exploite l'analogie existante entre un document XML et une source de chaîne de caractères produite par une grammaire BNF, elle enrichit alors XML par des attributs et des fonctions sémantiques [PC 99]. Il existe une autre communauté de chercheurs, qui proposent des langages pour exprimer la sémantique d'un document XML, en particulier nous faisons référence au travail de R. Worder qui a proposé MDL (Meaning Definition Language) [Wor 02] et le travail de S. liu, J. Mei et Z. lin qui a proposé le langage XSDL [She 04]. P. Patel-Schneider et J. Siméon proposent aussi un modèle

théorique pour la sémantique de XML [PS 03 a] et plus tard le Yin/Yang [PS 03 b], dans lequel le modèle de données de XML et de RDF sont unifiés.

Le Web sémantique d'aujourd'hui, s'est beaucoup orienté vers la description de l'information structurelle sémantisée et prend peu en compte l'étude du fonctionnement des services décrits [Por 04]. Le modèle sémantique que nous proposons permettra de décrire des documents XML à la fois structurés et dynamiques. Il intègre dans un même formalisme, la structure et le fonctionnement d'un composant actif du Web. Au contraire dans les modèles utilisés jusqu'à présent la description du service est séparée de l'information manipulée.

Contribution

L'objectif de ce travail est de répondre aux limites de XML par la spécification formelle des documents XML ainsi que les grammaires les générant. Cette modélisation a pour but de définir, d'une part la structure générique d'une classe de documents, c'est à dire les règles de composition des structures admises pour un document spécifique. D'autre part, la phase de validation qui permet de déterminer si un document XML est valide par rapport au modèle qui lui est associé. Entre autre, cette modélisation constituera une base théorique pour :

- La définition formelle des langages de requêtes pour XML ayant les caractéristiques d'expressivité importante.
- La description du fonctionnement des composants actifs programmés directement en XML.

L'avantage principal du travail suggéré sera alors double. Nous contribuerons, d'une part à la formalisation efficace d'un langage d'échange de données universelle, et d'autre part à une nouvelle proposition d'étude de cas de la logique de réécriture et son langage Maude.

Dans ce travail, nous nous sommes contentés alors de montrer comment le formalisme de logique de réécriture et sa flexibilité permettent de décrire d'une part, les différents aspects syntaxiques du langage XML, et d'autre part les règles qui permettent de produire un document XML valide à partir d'un document XML schéma.

Les spécificités du langage Maude nous ont permis d'associer une spécification simple, concise et exécutable à un sous ensemble du langage XML. Avec cette spécification, nous pouvons intégrer naturellement d'autres composants et fonctionnalités, et couvrir une grande partie du XML. En particulier, nous pouvons naturellement formaliser la

description des composants actifs (structure + fonctionnement) en XML. A travers cette étude, nous associons un modèle mathématique au langage XML, basé sur les travaux théoriques déjà faits sur la logique de réécriture. L'avantage de ce modèle XML est le raisonnement formel sur les concepts du langage XML.

Cette étude permettra aussi de traduire de manière naturelle les exemples déjà existants du monde XML dans le monde Maude.

Contenu du mémoire

Le présent mémoire est organisé comme suit :

- Dans le chapitre 1, nous présentons les concepts essentiels du formalisme de base utilisé, la logique de réécriture, que nous illustrons par des exemples simples permettant d'assimiler ces concepts théoriques.
- Le chapitre 2 est aussi consacré à la présentation de la syntaxe du langage Maude et les caractéristiques de son environnement. Nous insistons sur la façon de déclarer des modules paramétrés en Maude que nous introduisons par la suite dans notre modélisation.
- La description syntaxique d'un document XML simple, composé d'une collection d'éléments organisé par un ensemble des relations définissant une structure hiérarchique, est donnée au niveau du chapitre 3. Deux langages de définition de grammaire XML (DTD et XML Schéma) sont aussi cités dans ce chapitre, un intérêt particulier est attribué aux fichiers schémas XML qui seront utilisés dans le processus de validation par la suite.
- Dans le chapitre 4, nous proposons un support sémantique, à base de logique de réécriture, pour le langage XML et son langage de définition de grammaire XML Schéma. Une conséquence théorique importante se découle de cette étude concernant la validation d'un document XML par rapport à un fichier schéma XML. Deux approches (simple et paramétrée) distinctes pour programmer cette validation en Maude sont alors exposées dans la deuxième partie de ce chapitre.
- Le dernier chapitre (5) est réservé aux aspects d'implémentation de notre modèle. Le système Maude a servi pour la spécification et le prototypage d'un ensemble d'exemples simples traduits de XML à Maude. Une étude de cas réaliste, de grandeur naturelle est aussi envisagée dans ce chapitre, il s'agit de la traduction d'un fichier XML décrivant une partie de la structure du laboratoire LIRE de notre

département d'informatique (Université de Constantine) en Maude, ainsi que sa validation par rapport à une grammaire XML (sous forme de schéma XML) exprimée aussi en Maude.

- Le manuscrit se termine par une conclusion générale permettant de situer notre contribution par rapport aux travaux réalisés dans le même contexte et d'évoquer les différentes continuations jugées possibles pour ce travail.

Chapitre I

La Logique de Réécriture

1 Introduction

La réécriture est un paradigme général d'expression du calcul dans des diverses logiques computationnelles. Le calcul prend la forme de règles de réécriture dans une syntaxe donnée. Dans la logique de réécriture, une réécriture d'un terme consiste à le remplacer par un terme équivalent, conformément aux lois d'algèbre de termes. Cette logique a été présentée par José Meseguer [Mes 92], comme une conséquence de son travail sur les logiques générales pour décrire les systèmes concurrents.

La logique de réécriture présente un modèle unifié de concurrence dans laquelle plusieurs modèles bien connus des systèmes concurrents peuvent être représentés dans une structure commune.

La logique équationnelle, dans toutes ses variantes, est une logique de raisonnement sur des types de données statiques, en raison de la symétrie d'égalité, tous les changements sont réversibles. En enlevant la règle de symétrie de déduction dans la logique équationnelle, les équations ne sont plus symétriques et elles deviennent orientées. En laissant tomber la symétrie et l'interprétation équationnelle de règles on peut interpréter une règle t **à** t' comme une transition locale concurrente d'un système ou comme un pas d'inférence des formules de type t aux formules de type t'. Dans cette voie, nous arrivons à l'idée principale derrière la logique de réécriture [Clav 99 b].

Cette logique formalise le processus de réécriture pour calculer une relation de réécrivabilité entre les termes algébriques. Elle permet de raisonner sur des changements complexes possibles correspondant aux actions atomiques axiomatisées par les règles de

réécriture. Le calcul dans un système concurrent est exprimé à travers la déduction logique qui est intrinsèquement concurrente [Clav 99 b].

La logique de réécriture est proposée comme un cadre logique dans lequel d'autres logiques peuvent être représentées, et comme un cadre sémantique pour spécifier plusieurs systèmes et langages dans des domaines variés. Elle offre des techniques d'analyse formelle permettant de prouver des propriétés du système à spécifier, et de raisonner sur ses changements.

Plusieurs modèles de la concurrence ont fait déjà l'objet d'une intégration dans la logique de réécriture, nous citons : Les systèmes de transitions étiquetés [MM 93], les réseaux de Petri [MM 95], CCS [Mes 92], la machine chimique abstraite [MM 95], les ECATNets [BM 93], les structures d'évènements [Bou 98, Ram 98]. Comme il existe d'autre applications tel que : Les systèmes orientés objets, spécification des langages et des systèmes, les systèmes concurrents et/ou parallèles, vérification formelle des spécifications exprimées en logique de réécriture, etc.

Ce chapitre présente quelques éléments de base sur l'algèbre des termes et la réécriture en général, ainsi que les différentes notions formelles nécessaires à la compréhension de la logique de réécriture, en commençant par la définition d'une théorie de réécriture et les termes de preuve. En suite, nous découvrons le modèle (sémantique) d'une théorie de réécriture. Enfin, nous introduisons les modules du système MAUDE basé sur la logique de réécriture.

2 Algèbre des termes

Les termes représentent les objets principaux de n'importe quelle logique, parce que les formules qui permettent l'établissement des démonstrations sont toujours construites à partir des termes.

Pour cette raison et dans le but d'éclaircir encore mieux les différentes notions concernant la logique de réécriture, nous allons donner une définition précise de ses termes ainsi que des définitions d'autres concepts nécessaires à la compréhension de ces derniers.

2.1 Signature

Définition 1

Une signature Σ est une paire (S,Op) où :

- S est un ensemble de noms de sortes : $s_1, s_2, \ldots, s_m$
- Op est un ensemble d'opérations f : $s_1\ s_2 \ldots s_n$ **à** s_{n+1}

Pour toute opération f

- $(s_1, s_2, \ldots, s_n, s_{n+1})$ est appelé le profil de l'opération f
- $s_1\ s_2 \ldots s_n$ domaine de f
- s_{n+1} co-domaine de f
- n est l'arité de f.

Les constantes sont des opérations d'arité nulle.

Exemple 1.1

Signature des booléens :

Sorts bool — **S**

Opns true : **à** bool
false : **à** bool
not(_) : bool **à** bool
and : bool bool **à** bool
_or_ : bool bool **à** bool

Op

å

2.2 å-Algèbre

Définition 2

Soit une signature Σ=(S, Op), une ∑-algèbre A est une paire $A = (S_A, Op_A)$ tel que :

- $S_A = (As)s \in S$ ensemble de l'univers U
- $Op_A = (f_A)_{f \in Op}$ est l'ensemble de fonctions f_A telles que pour toute opération f : $s_1, \ldots s_n \rightarrow s_{n+1}$ de Op, f_A est une fonction telle que $f_A : A_{s1} \ldots A_{sn} \rightarrow A_{sn+1}$.

Remarques

1. Une ∑-algèbre correspond à une *interprétation* de la signature∑.

2. Nous notons Alg(∑) la classe de toutes les ∑-algèbres (toutes les interprétations possibles de ∑).

3. A_s est le support de s dans A.

4. f_A est l'interprétation de f dans A.

5. l'interprétation d'une opération constante est une fonction constante qui détermine un élément de A.

2.3 Les termes

Définition 3

Soit ∑=(S, Op) une signature et V est un ensemble infini dénombrable de variables, L'ensemble des termes T est le plus petit ensemble contenant les variables et les constantes (T =C U V) stable par l'application des symboles de fonctions de Op à des termes.

Autrement dit, un terme est un mot qu'on peut obtenir en appliquant récursivement un nombre fini de fois les règles ci dessous:

- tout symbole de constante est un terme.
- toute variable est un terme.
- Si f est un symbole de fonction d'arité n et si t1, …, tn sont des termes, alors

  f(t1,…,tn) est un terme.

Remarque

On appelle terme clos un terme qui ne contient pas de variable.

2.4 å-équation

Définition 4

Soit ∑=(S, Op) une signature et V un ensemble de variable, une ∑-équation est une paire (t, t') avec t et t' deux éléments de $T_\Sigma(V)$. Sachant que la notation $T_\Sigma(V)$ est utilisé pour désigner l'ensemble des termes T formés à partir de l'ensemble des variables V vérifiant la signature ∑ .

Une ∑-équation (t, t') est notée t = t'.

2.5 Spécification algébrique

Définition 5

Une spécification algébrique est le couple Spec = (∑, E) où :

- Σ : est une signature.
- E : est un ensemble de Σ-équations.

Exemple 1.2

Spécification algébrique des booléens

Sorts bool

Opns true : **à** bool

false : **à** bool

not(_) : bool **à** bool

and : bool bool **à** bool

_or_ : bool bool **à** bool

Vars x : bool

Eqns (not(true)=false)

(not(false)=true)

(true and x = x)

(false and x = false)

(true or x = true)

(false or x = x)

3 Théorie de réécriture

Dans une théorie de la logique de réécriture, sa signature est donnée par le couple (Σ, E) (où Σ est un ensemble de sortes et d'opérateurs, et E est un ensemble d'équations) ses règles de réécriture étiquetées représentant des axiomes. Elles agissent sur les classes d'équivalence des termes $T_{\Sigma, E}$ modulo E [MM 96].

Définition 6 (Théorie de réécriture étiquetée) : Une théorie de réécriture est un quadruplet $\Re$ = (Σ , E, L, R) tel que:

- Σ est un ensemble de symboles de fonctions et de sortes,
- E est un ensemble de Σ-équations,
- L est un ensemble d'étiquettes,
- R est un ensemble de paires $R \subseteq L \times (T_{\Sigma, E}(X))^2$ tel que le premier composant est une étiquette, et le second est une paire de classes d'équivalences de termes modulo les équations E avec $X = \{x_{1,\ldots} x_n \ldots\}$ un ensemble infini et dénombrable de variables.

Les éléments de R sont appelés règles de réécriture. On utilise pour ces dernier la notation suivante r : [t] $\rightarrow$ [t'].

Pour indiquer que $\{x_{1,\ldots}\ x_n \ldots\}$ est l'ensemble de variables apparue dans t ou bien dans t' nous écrivons r : [t $(x_1,\ldots., x_n)$] $\rightarrow$ [t'$(x_1,\ldots., x_n)$] ou dans une notation abrégée r : [t($\bar{x}$)] $\rightarrow$[t'($\bar{x}$)]

Etant donné une théorie de réécriture $\Re$, nous disons que $\Re$ implique une formule [t]$\rightarrow$[t'] et nous écrivons $\Re \vdash$ [t] $\rightarrow$ [t'] si et seulement si [t]$\rightarrow$[t'] est obtenue par application finie des règles de déduction suivantes :

1- Réflexivité. Pour chaque [t] $\in T_{\sum, E}(X)$,

$$\frac{}{[t] \rightarrow [t]}$$

2- Congruence. Pour chaque symbole de fonction $f \in \sum_n$, $n \in$ N,

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

3- Remplacement. Pour chaque règle de réécriture r : [t$(x_1,\ldots,x_n)$] $\rightarrow$ [t' $(x_1,\ldots,x_n)$] dans $\Re$,

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

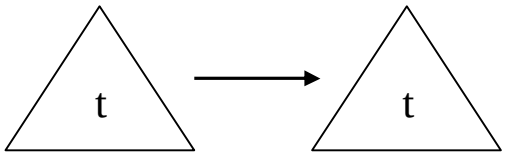
Où $\bar{w}/\bar{x}$ est une substitution de x_i par w_i ,$1 \leq i \leq n$

4- Transitivité.

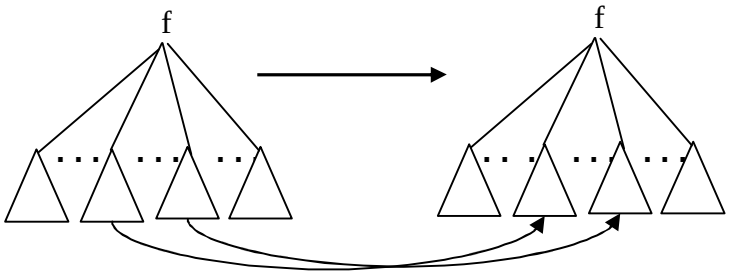
$$\frac{[t_1] \rightarrow [t_2][t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Nous pouvons visualiser graphiquement ces règles de déduction comme suit (figure 1.1) [MR 04] :

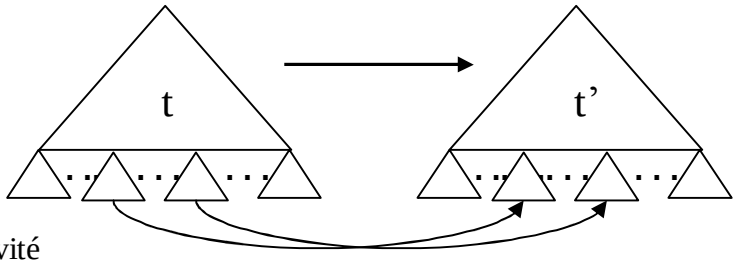
1. Réflexivité



2. Congruence



3. Remplacement



4. Transitivité

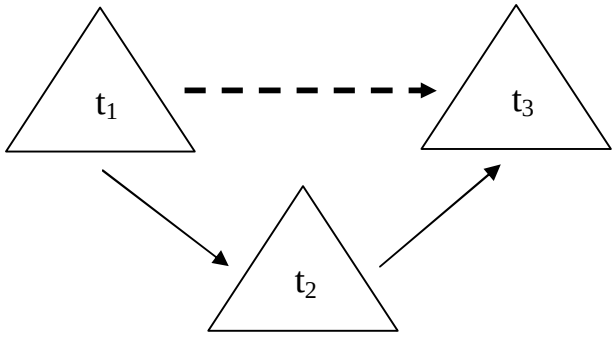


Figure 1.1 : Représentation graphique des règles de déduction

Exemple 1.3

Soit la théorie de réécriture suivante exprimant l'ensemble des booléens tel que :

$\sum$ = ({bool}, {T, F : **à** bool, _and_ : bool $\times$ bool **à** bool,})

E = {x and y = y and x, x and (y and z) = (x and y)and z}

L = {l_0, l_1}

R = {l_0 : [T] **à** [T and T]:, l_1 :: [F] **à** [F and T]}

Etant donné la formule suivante :

[F and F] **à** [F and T and F and T and T]

1) Nous pouvons prouver cette formule dans la théorie proposée en utilisant la règle de réécriture l_1 comme prémisse pour la règle de déduction congruence pour l'opération _and_ $\in \sum$:

$$\frac{[F]\rightarrow[FandT][F]\rightarrow[FandT]}{[FandF]\rightarrow[FandTandFandT]}$$ 'Congruence'

2) Nous allons prendre la règle l_0 et la réflexivité comme prémisse pour la règle de déduction congruence avec la même opération _and_ $\in \sum$:

$$\frac{[FandTandF]\rightarrow[FandTandF][T]\rightarrow[TandT]}{[FandTandFandT]\rightarrow[FandTandFandTandT]}$$ 'Congruence'

3) Ces deux dernier résultas seront utilisés comme des prémisses pour la règle de déduction transitivité pour obtenir le résultat attendu :

$$\frac{[FandF]\rightarrow[FandTandFandT][FandTandFandT]\rightarrow[FandTandFandTandT]}{[FandF]\rightarrow[FandTandFandTandT]}$$

'Transitivité'

Définition 7 (terme de preuve)

Soit $\Re$ = ($\sum$, E, L, R) une théorie de réécriture, un terme de preuve a est un terme de l'algèbre $T_R = T_{\sum \cup \{;\} \cup L, E}(X)$, tel que ' ; ' est un opérateur binaire (nous supposons qu'il n'y a aucune collision de noms parmi les différents ensembles d'opérateurs).

Donc, les formules d'une théorie de réécriture sont des triplets (a , [t],[t']) généralement écrit comme suit : a : [t] **à** [t'] tel que a est un terme de preuve et [t],[t'] ∈ $T_{\Sigma, E}$ (X).

Le terme de preuve contient la justification de la réécriture de [t] à [t'].

Nous disons que [t] se réécrit à [t'] via a si la formule a : [t] **à** [t'] peut être obtenue par un nombre d'applications fini des règles de déduction suivantes :

1- Réflexivité. Pour chaque [t] ∈ $T_{\Sigma, E}$ (X),

$$\frac{}{t:[t]\rightarrow[t']}$$

2- Congruence. Pour chaque symbole de fonction $f \in \Sigma_n$, $n \in$ N,

$$\frac{a_1:[t_1]\rightarrow[t'_1]\ldots a_n:[t_n]\rightarrow[t'_n]}{f(a_1,\ldots,a_n):[f(t_1,\ldots,t_n)]\rightarrow[f(t'_1,\ldots,t'_n)]}$$

3- Remplacement. Pour chaque règle de réécriture
r : [t(x_1,…,x_n)] → [t' (x_1,…,x_n)] dans $\Re$,

$$\frac{a_1:[w_1]\rightarrow[w'_1]\ldots a_n:[w_n]\rightarrow[w'_n]}{r(a_1,\ldots,a_n):[t(\overline{w}/\overline{x})]\rightarrow[t'(\overline{w'}/\overline{x})]}$$

Où $\overline{w}/\overline{x}$ est une substitution de x_i par w_i ,$1 \leq i \leq n$

4- Transitivité.

$$\frac{a_1:[t_1]\rightarrow[t_2]a_2:[t_2]\rightarrow[t_3]}{a_1;a_2:[t_1]\rightarrow[t_3]}$$

Exemple 1.4

Reprenons l'exemple 1.2 avec sa formule, le terme de preuve pour cette formule est le suivant :

(l_1 and l_1);((FandTandF) and l_0)

Ce terme de preuve est obtenu comme suit :

1) application de la règle de congruence à la règle l_1 pour la fonction _and_

2) application de la règle de la réflexivité pour le terme [FandTandF] et en suite nous appliquons la congruence entre ce terme et la règle l_0 pour la fonction _and_.

3) La dernière étape consiste à l'application de la règle de déduction transitivité.

Prenant maintenant par exemple la formule :

[FandF] **à** [F and T and F and T and T and T]

Deux preuves de cette formule correspondent aux termes de preuves :

$((l_1 \text{ and } l_1);l_0;l_0$

$(l_1 \text{ and } l_1); (l_0 \text{and } l_0)$

En plus de ces deux termes de preuve, nous pouvons trouver d'autres pour la même formule (nous parlons des classes d'équivalence pour les termes de preuves)

3.1 Modèle (sémantique) d'une théorie de réécriture

Les catégories sont le modèle mathématique associé aux théories de réécriture, celui ci peut supporter les termes générés par la spécification algébrique (Σ, E) et les preuves générées par la déduction. Les objets de cette catégorie sont les Σ-termes et les morphismes représentent les preuves entre eux. Nous pouvons justifier l'utilisation des catégories par l'importance des morphismes car ils peuvent faire une abstraction des détails d'implémentation des objets, et donc l'accent sera mis sur les relations qui existent entre ces objets.

De manière plus formelle, ce modèle est défini comme suit :

Définition 8 (modèle d'une théorie de réécriture)

Le modèle d'une théorie de réécriture $\mathfrak{R}$ = (Σ, E, L, R) est une catégorie $^{T}\mathfrak{R}$ (X) où les objets sont des classe d'équivalence de terme [t] $\in$ $T_{\Sigma, E}$ (X) et les morphismes sont aussi des classes d'équivalence de termes de preuves.

Les termes de preuves possèdent aussi une structure algébrique notée $^{p}\mathfrak{R}$(X) Intuitivement, cette structure est l'une des interprétations possibles de la réécriture dans une théorie $\mathfrak{R}$.

4 Applications de la logique de réécriture

Les applications de la logique de réécriture sont nombreuses et différentes, plusieurs modèles de concurrence ont été naturellement exprimés dans cette logique parmi eux nous citons le lambda calcul, les systèmes de transitions étiquetés, les réseaux de Petri, la machine chimique abstraite,CCS, LOTOS, E-LOTOS [Mes 02], et bien d'autres.

La logique de réécriture est utilisée également comme un paradigme de programmation déclaratif. Cela est montré par l'implémentation de cette logique au niveau du système Maude [Clav 00, Gad 96]. Ce système est caractérisé par sa mise en oeuvre rapide et performante et ses capacités de Réflexivité. Ces capacités permettent la spécification et l'implémentation des environnements exécutables pour des langages via une translation formelle de leurs syntaxe et sémantique à la logique de réécriture.

4.1 Les systèmes concurrents

La logique de réécriture est une logique pour le raisonnement correcte sur les systèmes concurrents ayant des états et évoluant au moyen des transitions. La signature d'une théorie de réécriture décrit une structure particulière des états d'un système qui peuvent être distribués selon cette structure [MM 96].

Les règles de réécriture décrivent les transitions élémentaires possibles. Les règles de déduction de la logique de réécriture nous permettent de raisonner correctement sur les transitions générales possibles dans un système satisfaisant une telle description. Ainsi, nous pouvons dire que chaque étape de réécriture est une transition parallèle locale dans un système concurrent.

À la différence de la plupart des autres logiques, les opérateurs logiques Σ et leurs propriétés structurelles E sont entièrement définis par l'utilisateur. Cela fournit une grande flexibilité pour considérer beaucoup de structures différentes d'état et donne une capacité plus grande à la logique de réécriture pour traiter des types différents des systèmes concurrents, et augmente aussi sa capacité pour représenter plus de logiques différentes.

Exemple 1.5

Un exemple de système concurrent intégré au niveau de la logique de réécriture est montré à travers le réseau de Petri ordinaire suivant (figure 1.2) :

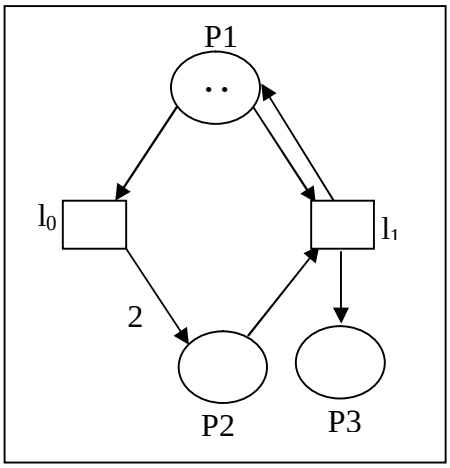


Figure 1.2 : Un exemple de réseau de Petri

Ce réseau de Petri peut être représenté par la théorie de réécriture suivante :

Rdp= (∑, E, L, R), tels que:

∑= ({place, marking}, {p_1, p_2, p_3 : **à** place, null: **à** place,_ : place **à** marking,

_._: marking × marking **à** marking})

E = {x.null = x, x.y = y.x, x.(y.z) = (x.y).z}

L = {l_0, l_1}

R = {l_0 : [p_1] **à** [p_2. p_2]:, l_1 .: [p_1.p_2] **à** [p_1.p_3]}

Dans cette théorie de réécriture, les places et le marquage du réseau de Petri sont spécifiés par le biais des sortes `place` et `marking`. Ainsi, l'ensemble des opérations suggérées sert à générer les places et le marquage du réseau. L'ensemble des règles de réécriture permet de décrire les transitions du réseau, la transition l_0 permet de consommer un jeton de la place p_1, et de générer deux jetons au niveau de la place p_2. La transition l_1 permet de consommer un jeton de la place p_1 et un autre de la place p_2, et de générer un jeton au niveau de la place p_1 et un autre dans la place p_3.

Un comportement simple de ce réseau peut être habituellement vu par son évolution de l'état [p_1.p_1], par exemple, vers un autre état [p_1. p_3. p_3] (voir figure 1.3) :

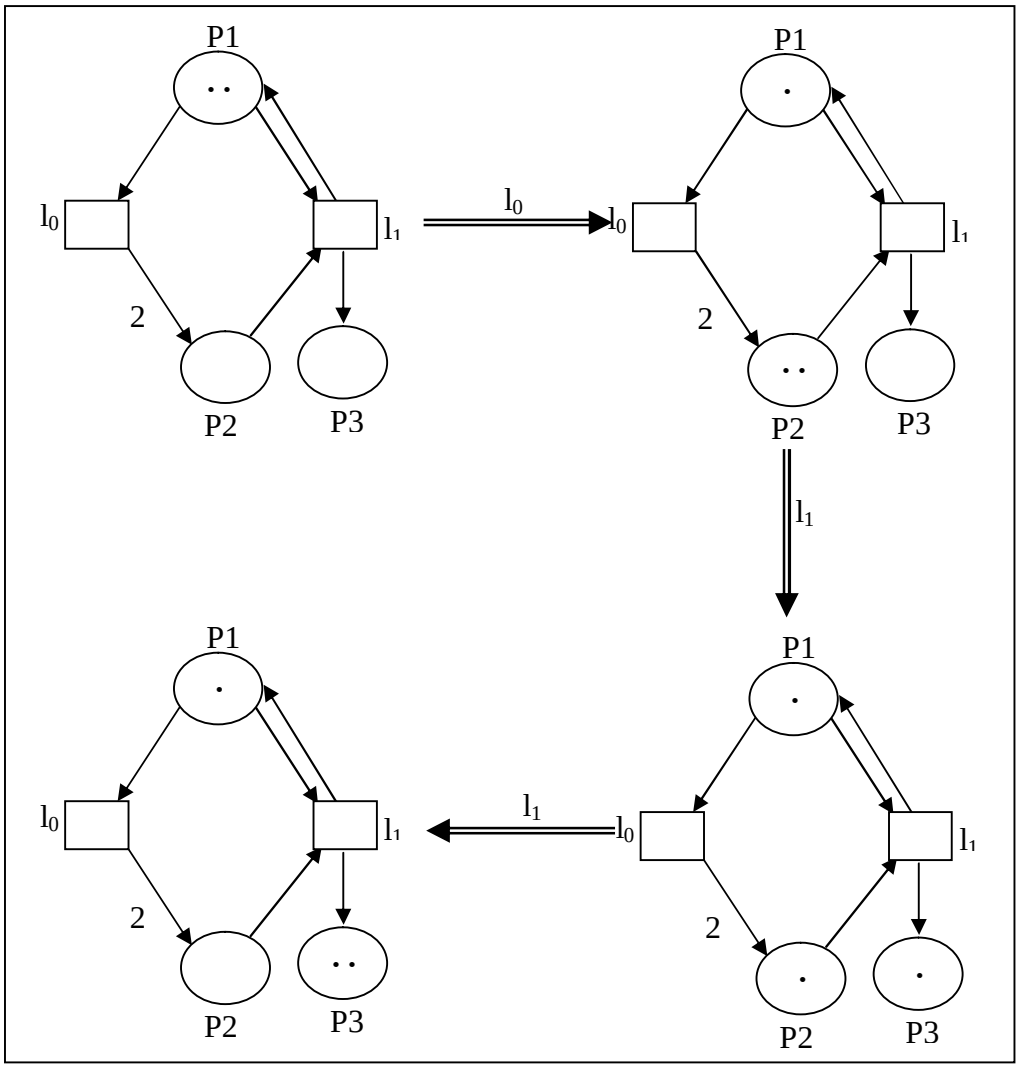


Figure 1.3 : Un exemple d'évolution de réseau de Petri

Ce comportement est déduit par la preuve : Rdp ├ [$p_1.p_1$] **à** [$p_1.\ p_3.\ p_3$]

Ayant comme terme de preuve :

($p_1.\ l_0$) ; ($l_1.\ p_2$) ; ($l_1.\ p_3$)

noté généralement :

l_0 ; l_1 ; l_1

Et obtenu ainsi :

$$\frac{p_1 : [p_1] \to [p_1] \qquad l_0 : [p_1] \to [p_2 . p_2]}{p_1 . l_0 : [p_1 . p_1] \to [p_1 . p_2 . p_2]}$$

Congruence

$$\frac{l_1 : [p_1.p_2] \to [p_1.p_3] \qquad p_2 : [p_2] \to [p_2]}{l_1.p_2 : [p_1.p_2.p_2] \to [p_1.p_3.p_2]}$$ Congruence

$$\frac{p_1.l_0 : [p_1.p_1] \to [p_1.p_2.p_2] \qquad l_1.p_2 : [p_1.p_2.p_2] \to [p_1.p_3.p_2]}{(p_1.l_0);(l_1.p_2) : [p_1.p_1] \to [p_1.p_3.p_2]}$$ Transitivité

$$\frac{p_3 : [p_3] \to [p_3] \qquad l_1 : [p_1.p_2] \to [p_1.p_3]}{l_1.p_3 : [p_1.p_2.p_3] \to [p_1.p_3.p_3]}$$ Congruence

$$\frac{(p_1.l_0);(l_1.p_2) : [p_1.p_1] \to [p_1.p_3.p_2] \qquad l_1.p_3 : [p_1.p_2.p_3] \to [p_1.p_3.p_3]}{(p_1.l_0);(l_1.p_2);(l_1.p_3) : [p_1.p_1] \to [p_1.p_3.p_3]}$$

Transitivité

Nous constatons bien qu'a ce niveau, le comportement d'un système concurrent (modéliser dans notre cas par un réseau de Petri) est décrit formellement par des déductions dans cette logique.

5 Conclusion

Nous avons présenté dans ce chapitre la logique de réécriture en définissant quelques notions de base de l'algèbre des termes, tels que les termes et les spécifications algébriques. Nous avons aussi donné la définition de la théorie de réécriture, les termes de preuves et le modèle sémantique d'une théorie de réécriture.

Nous avons évoqué la relation qui existe entre la logique de réécriture et les systèmes concurrents, cette logique présente un outil de raisonnement correct sur ces systèmes ayant des états et évoluant en termes de transitions. C'est alors un moyen rigoureux pour définir de manière formelle les systèmes concurrents.

Les différents concepts de la logique de réécriture sont implémentés à travers l'environnement MAUDE que nous allons présenter dans le chapitre suivant.

Chapitre II

Langage Maude

1 Introduction

La programmation déclarative a eu le mérite de faciliter considérablement les aspects conceptuels de la programmation. Cependant, la logique sur laquelle se base généralement les langages déclaratifs ne prend pas en charge les caractéristiques inhérentes aux systèmes réels, notamment l'action et le changement.

Pour mettre en évidence ce constat, Meseguer a défini un langage, appelé Maude [Clav 01, Clav 00], basé sur la logique de réécriture. La logique de réécriture est une logique de changements concurrents qui peut naturellement traiter les états et les calculs concurrents. Elle a de bonnes propriétés comme la flexibilité et elle offre un cadre sémantique dans lequel un grand nombre de langages et de modèles de concurrence peuvent être représentés. C'est aussi une meta logique dans la quelle beaucoup d'autres logiques peuvent être naturellement représentées et implémentées. Par conséquent, les applications les plus intéressantes de Maude sont celles de métalangage, dans lesquelles Maude est utilisé pour créer des environnements exécutables, des langages de programmation et des modèles de concurrence.

Dans cette partie nous présentons ce langage, à travers ses différents modules, l'utilisation de ses commandes est illustrée par des exemples. Nous accordons un intérêt particulier au concept de paramétrisation dans ce langage.

2 Le Système Maude

Maude est un langage déclaratif autour duquel est construit un système performant. Les programmes Maude sont des théories de réécriture et les calculs concurrents dans Maude représentent des déductions dans la logique de réécriture. Les buts du projet de Maude est

de supporter les spécifications formelles exécutables, et d'élargir le spectre d'utilisation de la programmation déclarative, spécifier et réaliser des systèmes de haute qualité dans des secteurs comme : le génie logiciel, réseaux de communication, l'informatique répartie, bioinformatique, et le développement formel d'outils [Clav 04].

Ce langage possède une collection d'outils formels supportant différentes formes de raisonnement logique pour vérifier des propriétés de programme comprenant :

• Un *Model-Checker* pour vérifier les propriétés temporelles linéaires de la logique (LTL) des modules d'états finis ;

• Un *Prouveur de Théorème* ("Theorem Prover") (ITP) : pour vérifier des propriétés des modules fonctionnels;

• Un *Analyseur Church-Rosser* permettant assurer une telle propriété pour les modules fonctionnels;

• Un *Analyseur de terminaison et un outil de Complétion* ("Knuth-Bendix") : pour les modules fonctionnels

• Un *Analyseur de Cohérence* : pour les modules systèmes.

Maude présente des caractéristiques très importantes comme la simplicité, la puissance et bien d'autres [Mcc 03] :

- *Maude est simple :* la programmation en Maude est très simple et facile à comprendre. Il existe des équations et des règles, et il a dans les deux cas une simple sémantique de réécriture dans laquelle les instances du côté gauche sont remplacées par les instances correspondants du côté droit. Maude possède un ensemble de mots clés et de symboles, et quelques orientations et conventions à maintenir, et donc, les programmes Maude sont aussi simples et aussi faciles à comprendre [Mcc 03].
- *Maude est expressif :* Il est possible d'exprimer naturellement dans Maude un très grand nombre d'applications, en commençant par les systèmes déterministes arrivant aux systèmes concurrents. Il offre aussi un du cadre sémantique, dans lequel non seulement les applications, mais des formalismes entiers (d'autres langages, d'autres logiques) peuvent être naturellement exprimées.
- *Maude est puissant :* il est possible d'utiliser ce langage non seulement pour les spécifications exécutables et le prototypage des systèmes, mais pour la programmation réelle et le développement.

- *Multi-paradigme :* Il combine la programmation déclarative et orientée objet. Dans Maude, les spécifications non exécutables sont appelées *les théories*, et leurs axiomes sont utilisés pour imposer des exigences formelles sur les modules programmes et sur les interfaces des modules paramétrés.

3 Core Maude

On appelle Core Maude l'interpréteur de Maude implémenté en C++ offrant toutes les fonctionnalités de base de Maude, intégrant les modules systèmes et fonctionnels, et n'acceptant qu'une hiérarchie des modules fonctionnels et systèmes non paramétrés.

Core Maude intègre d'autres modules tels que : les modules prédéfinis des types de données et le module *Model-Checker* ; la réflexivité et la fonction meta-langage sont aussi prises en compte.

3.1 Syntaxe de Maude

Nous présentons dans cette section les notions syntaxiques les plus importantes du langage Maude (les modules, les déclarations, etc) [Clav 04, Mcc 03].

3.1.1 Les Modules

Dans Core Maude, il existe deux types de modules : les module fonctionnels et les modules systèmes. Du point de vue programmation, un module fonctionnel est un programme fonctionnel de type équationnel, dont la syntaxe est définie par l'utilisateur. Le même module fonctionnel est considéré comme une théorie équationnelle avec la sémantique d'algèbre initiale, de point de vue spécification.

Un module système, est un programme concurrent de type déclaratif, dont la syntaxe est définie par l'utilisateur. Comme il présente une théorie de réécriture avec la sémantique du modèle initial [Clav 01, Clav 96].

a) Modules Fonctionnels

Les modules fonctionnels définissent des types de données et des opérations sur ces données par le biais des théories équationnelles.

La réécriture dans ces modules est basée sur un principe très simple, qui se résume à la simplification équationnelle.

Pour atteindre la forme réduite (finale) d'une expression, nous répétons l'application des équations comme des règles de réécriture jusqu'aucune nouvelle équation ne s'applique. Le résultat est appelé la forme canonique, qui est le même indépendamment de l'ordre de l'application des équations. Ainsi, chaque classe d'équivalence a un représentant canonique unique qui peut être calculé par la simplification équationnelle.

Un module fonctionnel est déclaré dans Maude en employant les mots-clés:

fmod (Nom du Module) is (Declaration set instructions) endfm

b) Modules Systèmes

Un module système spécifie une théorie de réécriture. Une telle théorie a des sortes, des opérateurs et peut avoir plusieurs types de déclarations : des équations, et des règles, dont toutes peuvent être conditionnelles.

Les règles spécifient les transitions locales concurrentes qui peuvent avoir lieu dans un système. Si la partie gauche de la règle correspond à un fragment de l'état de système et la condition de la règle est satisfaite, dans ce cas, la transition indiquée par la règle peut avoir lieu, et le fragment de l'état correspond est transformé à l'instance correspondante de la partie droite.

Un module système est déclaré dans Maude en employant les mots-clés:

mod *(Nom du Module)* is *(Declaration set instructions)* endm

La déclaration des sortes et de équations est identique à celle des modules fonctionnels, tout ce qui reste à savoir est la déclaration des règles.

Une règle de réécriture a la forme l : t **à** t', avec t et t' des termes de la même sorte, pouvant contenir des variables et une étiquette. Intuitivement, une règle décrit une transition locale concurrente dans un système.

3.1.2 La déclaration des sortes et sous sortes

Les types de données (Sorts) sont les premières à déclarer dans une spécification algébrique.

Une sorte est déclarée en utilisant le mot clé *sort* suivi par le nom de la sorte comme suit :

sort (le nom de la Sorte) .

Lorsque nous avons plusieurs sortes, nous pouvons utiliser le mot clé *sorts* comme suit :

sorts (sorte-1) (sorte-k) .

Le point (.) à la fin de la déclaration des sortes, comme pour les autres déclarations, est important. Et le manque du point ou de l'espace avant le point peut produire un problème ou un comportement inattendu.

Par exemple nous pouvons déclarer les sortes Nat et Zero comme suit :

sort Nat .
sort Zero .

Ou :

sorts Nat Zero .

Nous pouvons ordonner ces types selon une relation de sous sortes (subsorts).déclarée en employant le mot-clé *subsort* :

subsort (Sorte-1) < (Sorte-2) .

Cette déclaration indique que la première sorte est une sous sorte de la deuxième. Par exemple, la déclaration :

subsort Zero < Nat .

indique que la sorte Zero est une sous sorte de la sorte Nat.

3.1.3 La déclaration des opérations

Dans un module Maude, les opérations sont déclarées par le mot clé *op* suivi par le nom de l'opération, suivi par la liste des sortes formant ses arguments (arité) ; après la sorte du résultat apparaît à la fin. La forme générale de la déclaration d'une opération est la suivante :

op (OpName) : (Sorte-1) ... (Sorte-k) -> (Sorte) .

Par exemple, pour l'ensemble des naturels, nous pouvons déclarer les opérations habituelles 0, successeur, et + ainsi :

op zero : **à** Zero .
op s_ :Nat **à** Nat .
op _+_ : Nat Nat **à** Nat .

Si nous avons plusieurs opérations ayant la même déclaration nous utilisons le mot clé *ops* comme suit :

ops _+_ _*_ : Nat Nat **à** Nat .

Si la liste des arguments est vide, l'opération est une constante.

3.1.4 La surcharge des opérations

Les opérateurs dans Maude peuvent être surchargés, c'est-à-dire nous pouvons avoir plusieurs déclarations d'opérateurs pour le même opérateur avec des arités et des coarités différentes.

Par exemple l'opération _+_ peut être surchargée, en donnant trois déclarations à cette même opération pour trois sortes différentes (les naturels, les entiers, et les rationnels):

op _+_ : Nat Nat **à** Nat .

op _+_ : Int Int **à** Int .

op _+_ : Rat Rat **à** Rat .

3.1.5 Les variables

Une variable doit obligatoirement avoir une sorte particulière. Elle est déclarée dans Maude avec le mot clé *var* (ou *vars* pour plusieurs variables). Par exemple

var X : Nat .

vars X Y : Nat .

déclare une variable nommée X de type Nat.
Plusieurs variables de la même sorte peuvent être déclarées en utilisant le mot clé *vars*.

var X Y : Nat .

3.1.6 Les termes

Un terme est une constante, une variable, ou bien l'application d'une opération à une liste de termes d'argument. Le type d'une constante ou variable est sa sorte déclarée.

Dans l'application d'une opération, la liste d'arguments doit être en accord avec l'arité déclarée de l'opération. C'est-à-dire, elle doit avoir la même longueur et chaque terme doit avoir la même sorte de l'argument correspondant.

Nous donnons les deux exemples suivants :

s_(zero)

_+_(X :Nat, Y :Nat)

L'application d'une opération déclarée avec la forme mixfix a aussi une syntaxe mixfix, le nom mixfix de l'opération, avec chaque souligne est remplacé par le terme correspondant de la liste d'argument.

La forme de mixfix des exemples déjà donnés est:

s zero	:l'opération s a une notation préfixée.
X:Nat + Y:Nat	: l'opération + a une notation préfixée.
s zero	: l'opération s a ici une notation mixfixe.
X:Nat+Y:Nat	: l'opération + a ici une notation mixfixe.

Remarques

1. Étant donné un terme de la forme f (t_1,...., t_n), si t_i a la sorte s_i pour i = 1,...., n et il y a une déclaration d'opération f : s_1….s_n **à** s, alors le terme f (t_1,….,.t_n) a la sorte s et chacun de ses super sortes.
2. La forme préfixe peut être toujours utilisée pour n'importe quelle opération, même si l'opération a été déclarée avec la forme préfixe, ou bien mixfix.

Exemple 2.1 (Modules fonctionnels)

Reprenons l'exemple 1.5 des réseaux de Petri du premier chapitre. Nous pouvons maintenant lui associer d'abord un module fonctionnel:

```
fmod PN-SIGNATURE is
 sorts PLACE MARKING .
  ops p1 p2 p3: -> Place .
  op nill : -> Place .
  op _ : Place -> Marking .
  op _._ : Marking Marking -> Marking .
 vars X Y Z : Marking .
  eq X . nill = X .
  eq X . Y = Y . X .
  eq X . (Y . Z) = (X . Y) . Z .
endfm
```

Ce module fonctionnel définit la structure syntaxique (place, marquage) du réseau de Petri. Il commence par le mot clé *fmod* suivi par le nom du module, et se termine par le mot clé *endfm*.

Exemple 2.2 (Module systèmes)

Nous reprenons l'exemple 2.1, pour lequel nous ajoutons des règles de réécriture associées aux transitions de réseau de Petri pour décrire son comportement.

```
mod PETRI-NET is
Extending PN-SIGNATURE .
rl p1 ⇒ p2 . p2.
rl p1.p2 ⇒ p1.p3 .
endm
```

Le module PN-SIGNATURE de l'exemple 2.1 est importé par le module système PETRI-NET.

Remarque

Comme la plupart des langages de programmation, nous pouvons importer un module à un autre. Cela signifie que nous pouvons inclure toutes les déclarations et les définitions d'un autre module dans un nouveau pour éviter la redondance et bénéficier de la réutilisation. Nous utilisons les mots clé " Protecting, "Extending", ou "Including", selon l'utilisation du module importé. "Protecting" signifie essentiellement que les déclarations dans le module importé ne sont pas modifiées, et que toutes les opérations définies et les sortes sont employées strictement comme elles sont dans le module importé. "Including" indique que nous pouvons changer le sens dans lequel les déclarations ont été employées. Par exemple, si nous avons voulu importer le module INT (le module fourni pour des entiers) mais nous avons voulu définir son opérateur d'addition comme un opérateur de multiplication au lieu d'addition, nous devons utiliser "Including". "Extending" se trouve entre ces deux. La syntaxe est simple :

protecting MODULENAME .

or

including MODULENAME .

or

extending MODULENAME .

4 Full Maude

Full Maude est une extension de Core Maude écrite en langage Maude (notion de méta-level), et qui dote Maude d'un environnement puissant et extensible dans lequel les modules Maude peuvent être combinés pour donner d'autres modules complexes. Les modules peuvent aussi être paramétrés et instantiés avec des traces appelées *vues* (*views*). Les paramètres dans ces modules sont des théories définissant les conditions sémantiques pour l'instanciation correcte, et les théories peuvent être elles mêmes paramétrées.

Les modules orientés objet (peuvent être paramétrées) sont aussi utilisés dans cette version de Maude. Ils supportent les concepts de la programmation orientée objet : les objets, les messages, les classes et l'héritage [Clav 04, DM 99, Mcc 03].

4.1 Les modules orientés objets

Dans Full Maude, les systèmes concurrents orientés objet peuvent être définis par le biais des modules orientés objets présentés par le mot-clé *omod... Endom* utilisant une syntaxe plus pratique que celle des modules systèmes. Particulièrement tous les modules orientés objet incluent implicitement le module *CONFIGURATION +*.

Bien que les modules système de Maude soient suffisants pour la spécification des systèmes orientés objets, il y a des avantages conceptuels importants fournis par la syntaxe de modules orientés objets. Ces derniers permettent à l'utilisateur de penser et exprimer ses idées en orientés l'objet. Les modules orientés objets sont transformés en des modules systèmes pour des buts d'exécution.

Exemple 2.3 (Modules Orienté Objet)

Soit le module orienté objet suivant :

```
(omod CPT is
protecting QID .
protecting INT .
subsort Qid < Oid .
class Cpt | som : Int .
msgs credit debit : Oid Int -> Msg .
msg de_à_transferer_ : Oid Oid Int -> Msg .
vars A B : Oid .
vars M N N' : Int .
rl [credit] :
```

```
credit(A, M)
< A : Cpt | som : N >
=> < A : Cpt | som : N + M > .
crl [debit] :
debit(A, M)
< A : Cpt | som : N >
=> < A : Cpt | som : N - M >
if N >= M .
crl [transferer] :
(de A à B transferer M)
< A : Cpt | som : N >
< B : Cpt | som : N' >
=> < A : Cpt | som : N - M >
< B : Cpt | som : N' + M >
if N >= M .
endom)
```

Cet exemple décrit un compte bancaire, avec les changements qui peuvent surgir sur ce compte. Nous avons une classe appelée *Cpt* (spécifiant le compte) qui possède un attribut appelé *som* (présentant la somme actuelle du compte) de type entier. Cette classe possède trois méthodes (messages), la première appelée *credit* sert à ajouter une autre somme au compte, la deuxième est appelée *debit* sert à retirer une somme du compte, la dernière est appelée *de_à_transferer_* qui sert à transférer une somme d'un compte à un autre compte. Le déroulement de ces méthodes est exprimé par les règles de réécriture données au niveau du module *CPT*.

Pour déclarer les classes nous utilisons le mot clé *class* suivi par le nom de la classe, et puis nous trouvons une barre verticale après laquelle nous pouvons trouver les noms des attributs avec ses types.

La déclaration des messages est similaire à celle des opérations, mais nous utilisons *msg* ou *msgs* au lieu de *op* et *ops*. La syntaxe d'un objet est la suivante :

<nom de l'objet : sa classe ïnom de attribut$_1$: sa valeur,…, nom de attribut$_n$: sa valeur>

4.2 La programmation paramétrée

Dans Full Maude, nous pouvons utiliser en plus des mots-clés protecting, extending et including (ou pr, ex et inc dans la forme abrégée), pour définir des spécifications structurées, le concept puissant de la programmation paramétrée.

Les théories, les modules paramétrés et les vues forment les composantes de base de la programmation paramétrée dans Maude. Une théorie définit l'interface d'un module paramétré, c'est-à-dire la structure et les propriétés exigées d'un paramètre réel. L'instanciation des paramètres formels d'un module paramétré avec des modules de paramètre réels ou des théories exige une vue de la théorie d'interface au module réel correspondant [DM 99].

4.2.1 Les théories

Les théories sont utilisées pour déclarer les interfaces des modules paramétrés. Comme pour les modules, Full Maude supporte trois types différents de théories : théories fonctionnelles, théories systèmes et théories orientées objets. Leurs structure est la même que celle des modules correspondants. Les théories sont déclarées respectivement avec les mots-clés *fth... endfth*, *th... endth* et *oth... endoth*. Chaque type de ces théories peut avoir des sortes, des relations de sous sorte, des opérations, des variables, des équations, des règles, des classes, des relations de sous-classe et des messages. Elles peuvent aussi importer d'autres théories ou modules.

Les théories sont utilisées pour déclarer les exigences d'interface pour les modules paramétrés.

Exemple 2.4

Nous donnons un exemple de théories fonctionnelle la théorie *TRIV* qui est utilisée fréquemment, Et est prédéfinie dans Full Maude.

```
(fth TRIV is
sort Elt .
endfth)
```

Et une autre théorie système, la théorie *CHOIX*. Dans celle-ci nous spécifions une opération de choix défini sur les sacs selon une règle de réécriture non déterministe pour

prendre un des éléments dans le sac. Nous déclarons la sorte *Sac* comme une super sorte de la sorte *Elt* (élément).

```
(th CHOIX is
sorts Sac Elt .
subsort Elt < Sac .
op empty : -> Sac .
op _ _ : Sac Sac -> Sac [assoc comm id: empty] .
op choice : Sac -> Elt .
var E : Elt .
var B : Sac .
rl [choice] : choice(E B) => E .
endth)
```

4.2.2 Les modules paramétrés

Les modules peuvent être paramétrés avec une ou plus théories. Toutes les théories apparaissant dans l'interface doivent être étiquetées pour que leurs sortes puissent être identifiés. La forme syntaxique d'un module paramétré est : M (X1 :: T1 | . . . |Xn :: Tn), où M est le nom du module paramétr, X1 . . . Xn sont les étiquettes et T1 . . . Tn sont respectivement les théories paramètre, et (X1 :: T1 | . . . |Xn :: Tn) st son interface.

Dans Full Maude toutes les sortes parvenant des théories de l'interface doivent être qualifiés par leurs étiquettes, même s'il n'y a pas d'ambiguïté. Si Z est l'étiquette d'une théorie de paramètre T, donc chaque sorte S dans T doit être qualifié comme Z@S. il ne peut pas y avoir la surcharge des sous sortes entre un opérateur déclaré dans une théorie étant utilisée comme le paramètre d'un module paramétré et un opérateur déclaré dans le corps du module paramétré, ou entre des opérateurs déclarés dans deux théories paramètre du même module.

Exemple 2.5

Un module paramétré SIMPLE-SET, avec TRIV comme interface peut être défini comme suit :

```
(fmod SIMPLE-SET(X :: TRIV) is
sorts Set .
subsorts X@Elt < Set .
op mt : -> Set .
```

```
op _ _ : Set Set -> Set [assoc comm id: mt] .
op _in_ : X@Elt Set -> Bool .
var E : X@Elt .
var S : Set .
eq E E = E .
eq E in E S = true .
eq E in S = false .
endfm)
```

Dans Full Maude, nous supposons que toutes les vues sont nommées et que ces noms sont ceux utilisés dans la qualification.

En général, les modules paramétrés peuvent avoir plusieurs paramètres. Il peut en outre arriver que plusieurs paramètres sont déclarés avec la même théorie comme paramètre, c'est-à-dire nous pouvons avoir par exemple une interface de la forme (X :: TRIV | Y :: TRIV). Par conséquent, les paramètres ne peuvent pas être traités comme des sous modules. Nous ne considérons pas la relation entre le corps d'un module paramétré et l'interface de ses paramètres comme une inclusion. Le module paramétré est plutôt évalué en produisant des copies renommées des paramètres, qui y sont alors incluses. Pour l'interface précédente, deux copies de la théorie TRIV sont alors produites, avec des noms X :: TRIV et Y :: TRIV.

Exemple 2.6

Nous considérons comme un exemple de module à deux paramètres ; le module *PAIR*, dans lequel nous désignons l'utilisation et la qualification des sortes parvenant de chacun des deux paramètres.

```
(fmod PAIR(X :: TRIV | Y :: TRIV) is
sort Pair(X | Y) .
op <_;_> : X@Elt Y@Elt -> Pair(X | Y) .
op 1st : Pair(X | Y) -> X@Elt .
op 2nd : Pair(X | Y) -> Y@Elt .
var A : X@Elt .
var B : Y@Elt .
eq 1st(< A ; B >) = A .
eq 2nd(< A ; B >) = B .
endfm)
```

4.2.3 Les vues

Une vue établit une correspondance entre les sortes et les opérations définies dans la théorie à celles du module paramétré réel donné. Autrement dit, les vues fournissent une interprétation des paramètres réels. Nous créons une vue en utilisant la syntaxe : *NomdeVue from THÉORIE to MODULE is... endv*. Théorie et module sont, évidemment, les noms de la théorie et celui du module réel. La transformation réelle est réaliser dans la vue.

Exemple 2.7

Reprenant l'exemple 2.5, concernant le module paramétré SIMPLE-SET. Nous pouvons spécifier un ensemble des entiers par l'instanciation de ce module des ensembles avec la vue `Int`, qui définie une correspondance entre la sorte `Elt` de la théorie source `TRIV` et la sorte `Int` du module cible `INT`. La vue `Int` est définie comme suit :

```
(view Int from TRIV to INT is sort Elt to Int . endv)
```

Et le module ensemble des entiers `INT-SET` peut être défini de façon paramétrée en utilisant l'instantiation `SIMPLE-SET [Int]`.

5 Les commandes les plus utilisées dans Maude

Nous considérons le module prédéfini NAT pour illustrer l'utilisation des différentes commandes de Maude (tableau 2.1) [Clav 04, Mcc 03]. Dans ce module proposé nous avons, en plus des opérations habituelles des naturels (zero, s, +), l'opération notée !, qui sert à effectuer un choix quelconque entre deux naturels.

```
mod NAT is
        sort Nat .
         op zero : -> Nat .
         op s_ : Nat -> Nat .
         op _+_ : Nat Nat -> Nat [comm].
         Op _!_ : Nat Nat -> Nat .
         vars N M : Nat .
         eq zero + N = N .
         eq s N + s M = s s(N + M) .
         rl N ! M => N .
         rl N ! M => M .
       endfm
```

Syntaxe	Exemple	commentaire
parse *[in (Module) :] (Terme)*	`Maude> parse in Nat` s zero +s zero`.` `Nat: s zero + s zero`	analyse syntaxique des termes

Set trace *on*	`Maude> set trace on .`
	`Maude>red zero + zero .`
	`reduce in NAT: zero + zero .`
	`********** equation`
	`eq zero + N = N .`
	`N --> zero`
	`zero + zero`
	`--->`
	`zero`
	`rewrites: 1 in -152546541049ms cpu (0ms real) (~ rewrites/second)`
	`result Nat: zero`
		afficher les équations et les règles appliquées à chaque opération de réécriture

reduce *{*in *module}* : *term* .	`Maude> red zero + (zero + zero ).`
	`reduce in NAT : zero + ( zero + zero ) .`
	`rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)`
	`result Nat: zero`
		évaluer des expressions

rewrite {[*bound*]} *{*in *module*:*}* *term* .	`Maude> rew [1] zero + s zero .`
	`rewrite [1] in NAT : zero + s zero .`
	`rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)`
	`result Nat: s zero`
	`Maude> rew [2] s zero + ( zero+ s s s zero ) .`
	`rewrite [2] in NAT : s zero + (zero + s s s zero) .`
	`rewrites: 3 in 9999999000ms cpu (0ms real) (0 rewrites/second)`
	`result Nat: s s s s zero`
		le terme spécifié sera réécrit en utilisant les règles, et les équations du module donné. Le processus arrête quand le nombre d'application des règles atteint la limite *bound*

show module *{module}*	Maude> show module NAT. mod NAT is sort Nat . op zero : -> Nat . op s_ : Nat -> Nat . op _+_ : Nat Nat -> Nat . op _!_ : Nat Nat -> Nat . vars N M : Nat . eq zero + N = N . eq s N + s M = s s (N + M). rl N ! M => N . rl N ! M => M . endm	visualiser le module spécifié
show sorts *{module}*	Maude> show sorts . sort Nat .	Visualise une représentation des informations des sortes et des sous sortes
show ops *{module}*	Maude> show ops . op zero : -> Nat . op s_ : Nat -> Nat [prec 15 gather (E)] . op _+_ : Nat Nat -> Nat [comm prec 41 gather (E E)] . op _!_ : Nat Nat -> Nat [prec 41 gather (E E)] .	visualise la liste des opérations
show vars *{module}*	**Maude> show vars .** **var N : Nat .** **var M : Nat .**	visualise la liste des variables
show eqs *{module}*	**Maude> show eqs .** **eq zero + N = N .** **eq s N + s M = s s (N + M) .**	visualise la liste des équations
show rls *{module}*	**Maude> show rls .** **rl N ! M => N .** **rl N ! M => M .**	visualise la liste des règles

Tableau 2.1 : Les commandes de Maude

6 Conclusion

Nous avons donné dans ce chapitre les différents concepts de base concernant le langage Maude. Nous avons présenté le système Maude, son implémentation (Core et full Maude) avec les notions les plus importantes concernant sa syntaxe, ces différents modules, et ses commandes d'exécution. Nous avons présenté beaucoup plus en détail la programmation paramétrée en Maude du moment que nous allons l'adopter dans l'implémentation de notre approche.

Maude est un langage avec un large spectre supportant la spécification formelle, le prototypage rapide et la programmation parallèle. Maude inclut différents paradigmes : Fonctionnel, l'orienté objet. Le fait que la logique de réécriture est réflexive mène à des nouvelles capacités concerne la meta programmation qui peuvent augmenter la réutilisabilité et l'adaptabilité des logiciels développés .

Dans la suite nous allons procéder à l'intégration du langage XML dans ce formalisme, tout en bénéficiant des capacités de Maude.

Chapitre III

Langage XML

1 Introduction

Depuis son élaboration, en 1996, dans le but d'adapter le SGML au Web et de transformer le HTML (Hyper Texte Markup Language) en un langage porteur de sémantique, XML (eXtensible Markup Language) a connu un très grand succès, car il offre une technique de modélisation normalisée, qui met à la disposition des développeurs un langage unique de structuration de données et d'écriture de modèles.

- Il enlève les frontières entre les formats de données : XML permet d'avoir un format unique tant pour les documents que les graphiques ou les données.
- Il permet aussi la diminution du nombre d'interfaces spécifiques à développer. XML abaisse le coût de rédaction des spécifications techniques car un modèle XML suffit pour spécifier complètement le format des données échangées entre deux applications. XML garantit une diminution des coûts de développement.
- Il donne une syntaxe qui tient compte de l'hétérogénéité des systèmes et des langages humains et l'indépendance entre les documents XML et leur modèle, ne rend plus nécessaire le recours systématique au modèle pour lire et traiter les données.

Ce langage possède une puissance dans la transmission des données qui a fait la rapide notoriété de XML. Une donnée transmise au format XML est ouverte à toutes les formes de traitements, comme l'affichage de formulaires, l'enregistrement dans une base de données relationnelle, la transformation en document imprimable, l'intégration dans des applications multimédias ou la reprise par une application tierce.

La grammaire des fichiers XML est définie par le bais des modèles spécifiques, comme les DTD et les schémas XML. Un modèle permet de définir le vocabulaire d'une classe de fichiers XML.

Dans ce chapitre nous présentons les aspects syntaxiques de base du langage XML, nous donnons en premier lieu la structure générale d'un document XML, ainsi que celle d'un fichier schéma XML.

Nous abordons par la suite l'aspect validation des documents XML par rapport soit à une DTD soit à un schéma XML.

2 La syntaxe du langage XML

2.1 Structure d'un document XML

La figure 3.1 représente la structure générale d'un document XML, qui se compose de trois parties [All 00, Sil 03, Ber 00] :

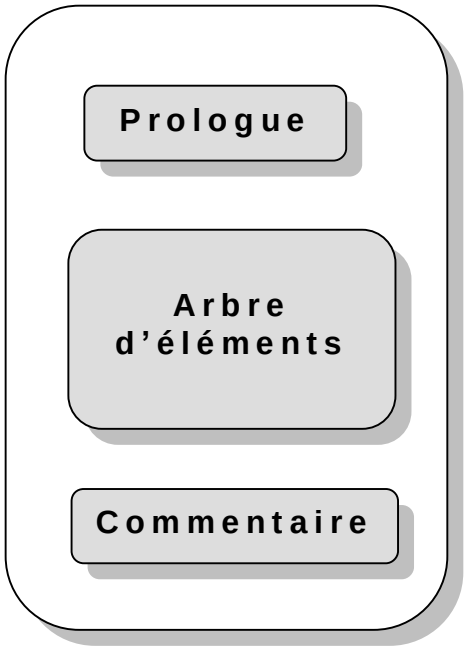


Figure 3.1 : Structure d'un document XML

- Le prologue, sa présence est facultatif mais conseillée. Il contiendra une déclaration de la version du XML, et le codage de caractères utilisé. Comme il peut contenir une référence vers le modèle de validation utilisé (DTD ou schéma XML).

- L'arbre d'éléments, est la partie essentielle d'un document XML, elle est formée d'une hiérarchie d'éléments. L'élément père (l'élément racine) qui contient tous les autres éléments est unique dans un document.
- Les commentaires, dont la présence est facultative. Ils sont encadrés par les marques de début <!-- et de fin --> de commentaire.

Exemple 3.1

Cet exemple illustre les composants les plus importants d'un document XML :

<?xml version="1.0" encoding="UTF-8"?>	Le prologue
<parent>	Début de l'arbre d'éléments (balise d'ouverture de l'élément racine)
<garcon >Ali</garcon>	Le premier élément fils
<fille >Lylia</fille>	Le deuxième élément fils
</parent>	La balise de fermeture de l'élément racine

2.2 Les éléments d'un document XML

Chaque élément d'un document XML se compose d'une balise d'ouverture, d'un contenu d'élément et d'une balise de fermeture [All 00, Bra 00].

```
<nom>contenu-élément</nom>
```

Le nom des éléments est composé de caractères alphanumériques, de l'underscore _, du signe moins - ou du point. Le caractère deux-points : est autorisé mais il a une signification particulière. De plus le premier caractère doit obligatoirement être alphabétique ou un underscore. Ce nom ne doit pas contenir d'espacement ou de fin de ligne. Aucun nom ne peut commencer par la chaîne `"xml"`.

XML fait une distinction entre majuscule et minuscule, ce qui fait que les balises `<nom>` et `<NOM>`, ne sont pas équivalentes.

Le contenu d'un élément XML peut être un contenu texte, ou encore un ensemble d'éléments fils. Comme il peut être vide (juste le nom de l'élément avec ou sans un ensemble d'attributs).

2.3 Les attributs des éléments XML

Un élément peut inclure des attributs dans sa balise d'ouverture sous la forme de paires [All 00] :

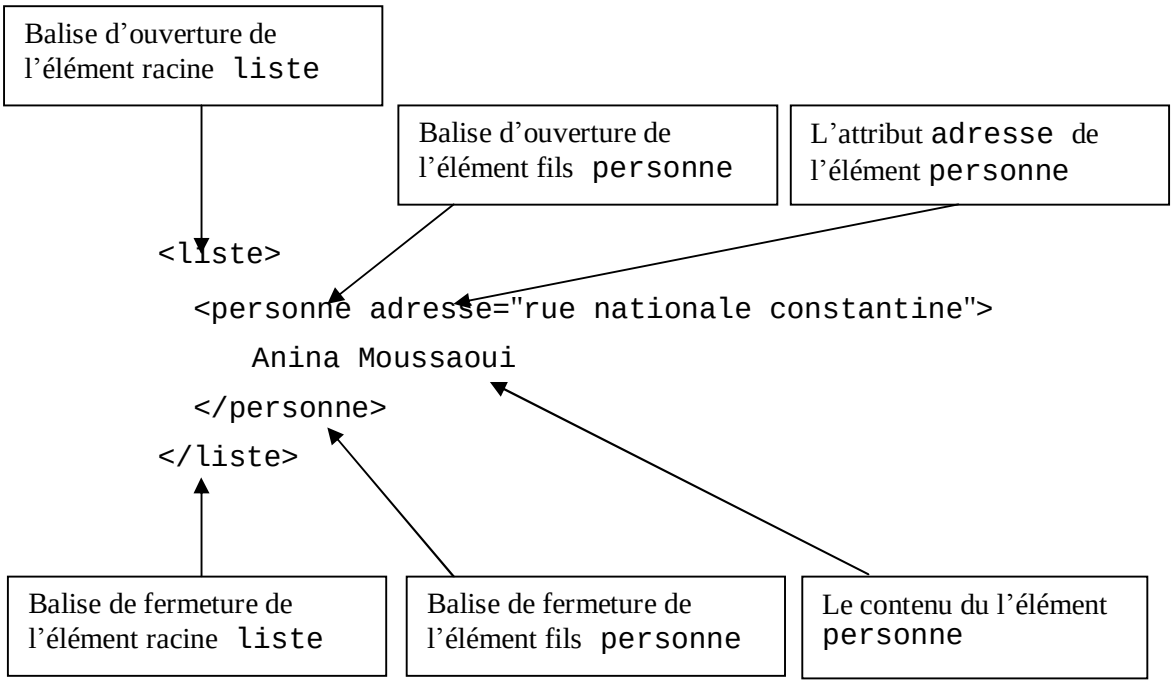
```
nom="valeur"
```

On peut avoir des documents avec des éléments sans attributs et d'autres utilisant des éléments avec attributs car on n'a pas des règles d'utilisation de ces derniers.

L'avantage d'utiliser les attributs réside dans le fait que le document pourra, avec la feuille de style adéquate, être visualisé sans problème dans un navigateur.

Par contre un document qui n'utilise pas d'attributs sera peut être plus concis mais ne pourra plus forcement être correctement affiché dans un navigateur.

Exemple 3.2



3 Définition des grammaires XML

La grammaire en général est l'ensemble des règles d'assemblage et d'accord des mots permettant à une langue d'exprimer des idées compréhensibles. Les seuls mots du vocabulaire ne suffisent pas, et il faut des règles pour les arranger les uns par rapport aux autres. Les règles de la grammaire XML sont données par modèles bien spécifiques.

Historiquement, la DTD (Document Type Definition) présente le premier modèle proposé, ensuite les schémas XML, qui sont développés par la société Microsoft, il s'agit en fait d'un langage XML dédié à la définition de grammaire [All 00, Ber 00].

Un schéma XML permet de définir une grammaire XML selon une approche orienté objets. Il permet de définir ses propres types, de les réutiliser et les spécialiser.

Un document qui a été vérifié par rapport à une DTD ou un schéma, est considéré comme un document "valide" (voir figure 3.2) [Tho 03]. Certains outils, plus ou moins simples, permettent de valider les fichiers de données. Il s'agit d'ailleurs plus d'environnements de travail, que de simples outils. Sur plate-forme Windows, la société Microsoft en fournie une multitude comme XML Spy, Oxygène, Xmllint, Schematron etc [Oba 04].

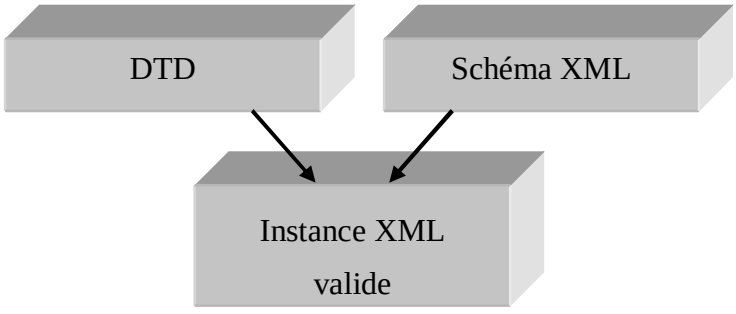


Figure 3.2 : Une instance XML valide par rapport à une DTD ou à un schéma XML

3.1 Définition des documents type (DTD)

Le but d'une DTD est de définir les composantes légales de n'importe quel document basé SGML. La DTD définit la structure du document avec une liste d'éléments légaux (c'est-à-dire les éléments qui doivent apparaître dans le document). Il existe deux possibilités pour définir une DTD. Soit embarquer la DTD au sein d'un fichier de données, soit la définir dans un fichier externe. Les deux techniques peuvent même être mixées. Notons, dans les deux cas, que la DTD va devoir être nommée [All 00, Sil03].

3.1.1 Déclaration d'éléments

La déclaration d'un type d'élément peut se faire en associant un nom de type à un modèle de contenu. Tous les éléments instances d'un type doivent être conformes au modèle défini au niveau de la déclaration. La syntaxe de cette déclaration est la suivante :

```
<!ELEMENT nom modèle>
```

Le modèle de contenu permet la création d'éléments contenant des éléments fils ou des données représentées par un flot de caractères ou un mélange de données et d'éléments fils. Il est aussi possible de spécifier qu'un élément reste vide.

Le nombre d'occurrences des éléments fils est spécifié dans le tableau 3.1 :

Occurrence du symbole	Signification
?	Zéro ou une fois
*	Zéro, une ou plusieurs fois
+	Une ou plusieurs fois
< Rien >	Une seule fois

Tableau 3.1 : La spécification de nombre d'occurrences des éléments fils

3.1.2 Déclaration de liste d'attributs

La déclaration d'attribut dans une DTD permet de spécifier quels sont les attributs qui pourront ou devront être associés à une instance d'élément, et indiquer éventuellement quelle sera la valeur par défaut d'un attribut:

```
<!ATTLIST nom_élément nom_attribut type_attribut déclaration de-
défaut>
```

La déclaration peut prendre quatre formes (voir tableau 3.2):

Type d'attribut	Signification
Une valeur	la valeur par défaut de l'attribut
#FIXED	la valeur est fixée
#REQUIRED	l'attribut est obligatoire
#IMPLIED	L'attribut est optionnel

Tableau 3.2 : Les types d'attributs

Exemple 3.3

Nous allons donner la DTD correspondante au document XML du l'exemple 3.1 :

```
<?xml version="1.0" standalone="yes" ?>
< ! DOCTYPE parent [< ! ELEMENT parent (garcon, fille)>
< ! ELEMENT garcon (#PCDATA)>
< ! ELEMENT fille (#PCDATA)>]>
```

3.1.3 Les limites des DTD

Les DTD souffrent de nombreuses limites [CA 01]:

1. Les DTD ne sont pas au format XML. Cela signifie qu'il est nécessaire d'utiliser un outil spécial pour analyser un tel fichier, différent de celui utilisé pour les fichiers XML.
2. Les DTD ne supportent pas les "espaces de nom". En pratique, cela implique qu'il n'est pas possible, dans un fichier XML défini par une DTD, d'importer des définitions de balises définies ailleurs.
3. Le "typage" des données est extrêmement limité.

3.2 Les schémas XML

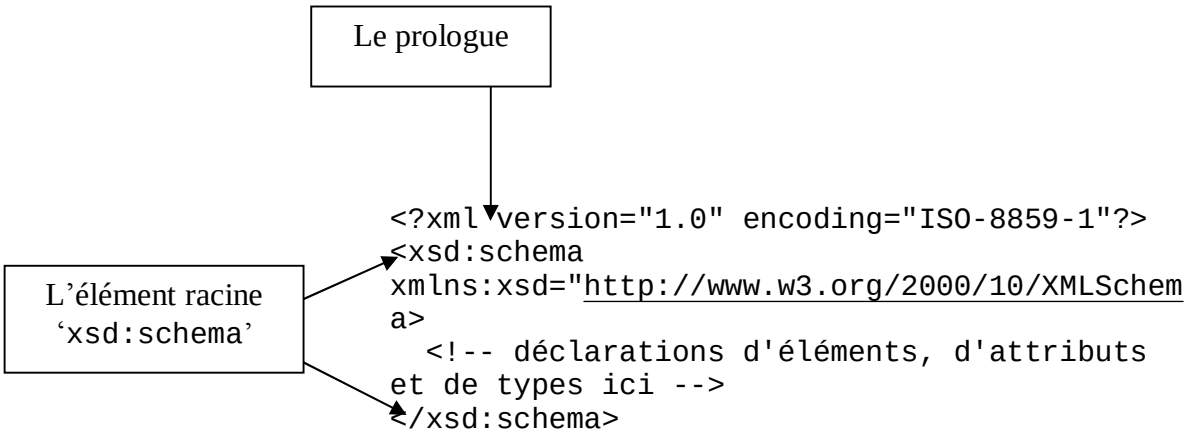
Pour pallier aux manques précités des DTD, les schémas XML ont été conçus. Ces derniers proposent, en plus des fonctionnalités fournies par les DTD, des nouveautés [CA 01]:

1. Le typage des données est introduit, ce qui permet la gestion de booléens, d'entiers, d'intervalles de temps, etc. Il est même possible de créer de nouveaux types à partir de types existants.
2. Le support des espaces de nom.
3. Les indicateurs d'occurrences des éléments peuvent être tout nombre non négatif (rappel : dans une DTD, on était limité à 0, 1 ou un nombre infini d'occurrences pour un élément).
4. Les schémas sont très facilement concevables par modules.
5. La séparation entre le modèle de validation et l'instance XML.

3.2.1 Structure de base

Les schémas XML utilisent une syntaxe proche du XML. Chaque document schéma XML commence par un prologue, et possède un élément racine, et les noms des éléments doivent commencer par un même préfixe `xsd` [Cha 05, Bro 01, CA 01].

Exemple 3.4



3.2.2 Déclaration d'éléments

Un élément, dans un schéma, se déclare avec la balise `<xsd:element>`. Les éléments sont typés via l'attribut '`type`'.

Exemple 3.5

Cet exemple présente comment définir les deux éléments : l'élément `contacts` et l'élément `remarque`. L'élément `contacts` du type `typeContacts`, qui est un type complexe défini par l'utilisateur. L'élément `remarque` qui est du type `xsd:string` qui est un type simple prédéfini de Schéma XML.

Les éléments pouvant contenir des éléments fils ou des attributs sont dits de type complexe, tandis que les éléments n'en contenant pas sont dits de type simple. Pour les éléments de type complexe on peut déclarer des séquences d'éléments, des choix d'éléments, comme on peut définir des contraintes d'occurrence.

a. Séquences d'éléments

Nous pouvons déclarer un élément contenant un ou plusieurs éléments fils. En utilisant l'élément `xsd:sequence`

Exemple 3.6

```
<xsd:complexType name="typePersonne">
  <xsd:sequence>
      <xsd:element name="nom" type="xsd:string"/>
      <xsd:element name="prénom" type="xsd:string"/>
      <xsd:element name="dateDeNaissance" type="xsd:date"/>
      <xsd:element name="adresse" type="xsd:string"/>
      <xsd:element name="adresseElectronique" type="xsd:string"/>
      <xsd:element name="téléphone" type="numéroDeTéléphone"/>
  </xsd:sequence>
</xsd:complexType>
```

Nous avons donné ici une définition d'un type complexe appelé `typePersonne`. Chaque élément dans un document XML ayant ce type doit avoir six élément fils : `nom`, `prénom`, `dateDeNaissance`, `adresse`, `adresseElectronique`, `téléphone`.

b. Choix d'éléments

On peut définir un choix entre les éléments fils. Dans l'exemple 3.7 nous pouvons faire le choix entre adresse d'une personne, et adresse électronique, en indiquant l'élément `xsd:choice`.

Exemple 3.7

```
<xsd:complexType name="typePersonne">
  <xsd:sequence>
   <xsd:element name="nom" type="xsd:string"/>
   <xsd:element name="prénom" type="xsd:string"/>
   <xsd:element name="dateDeNaissance" type="xsd:date"/>
   <xsd:choice>
    <xsd:element name="adresse" type="xsd:string"/>
    <xsd:element name="adresseElectronique" type="xsd:string"/>
   </xsd:choice>
   <xsd:element name="téléphone" type="numéroDeTéléphone"/>
  </xsd:sequence>
  </xsd:complexType>
```

c. L'élément all

Cet élément constitue une nouveauté par rapport aux DTD. Il indique que les éléments fils doivent apparaître une fois (ou pas du tout), et dans n'importe quel ordre. Cet élément `xsd:all` doit être un enfant direct de l'élément `xsd:complexType`.

Exemple 3.8

```
<xsd:complexType name="typePersonne">
<xsd:all>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prénom" type="xsd:string"/>
    <xsd:element name="dateDeNaissance" type="xsd:date"/>
    <xsd:element name="adresse" type="xsd:string"/>
    <xsd:element name="téléphone" type="numéroDeTéléphone"/>
</all>
</xsd:complexType>
```

d. Contraintes d'occurrence

Dans une `DTD`, un indicateur d'occurrence ne peut prendre que les valeurs 0, 1 ou l'infini. On peut forcer un élément `sselt` à être présent 378 fois, mais il faut pour cela écrire `(sselt, sselt..., sselt, sselt)` 378 fois. `Schéma XML` permet de déclarer directement une telle occurence, car tout *nombre entier non négatif* peut être utilisé. Pour déclarer qu'un élément peut être présent un nombre illimité de fois, on utilise la valeur `unbounded`. Les attributs utiles sont `minOccurs` et `maxOccurs`, qui indiquent respectivement les nombres minimal et maximal de fois où un élément peut apparaître. Le tableau 3.3 récapitule les possibilités :

Dans une DTD	valeur de minoccurs	valeur de maxoccurs
*	0	Unbounded
+	1	Unbounded
?	0	1
Rien	1	1

Tableau 3.3 : Liste des indicateurs d'occurrence

3.2.3 Déclaration d'attributs

Un attribut ne peut être que de type simple. Cela signifie que les attributs ne peuvent contenir ni d'autres éléments, ni attributs.

Un attribut se déclare via la balise `<xsd:attribute>` Cette balise doit être placée après les éléments `<xsd:sequence>`, `<xsd:choix>` et `<xsd:all>`.

Exemple 3.9

L'exemple montre la déclaration d'un attribut `maj` de type `xsd:date` (un type simple) qui indique la date de dernière mise à jour de la liste des contacts.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
<xsd:element name="contacts" type="typeContacts"/>
<xsd:element name="remarque" type="xsd:string"/>
<!-- déclarations de types ici -->
<xsd:complexType name="typeContacts">
  <!-- déclarations du modèle de contenu ici -->
  <xsd:attribute name="maj" type="xsd:date">
</xsd:complexType>
</xsd:schema>
```

L'élément `attribute` dans un Schéma XML peut avoir trois attributs optionnels : `use`, `default` et `fixed`. Des combinaisons de ces trois attributs permettent de paramétrer ce qui est acceptable ou non dans le fichier `XML` final (attribut obligatoire, optionnel, possédant une valeur par défaut, etc.). Par exemple, la ligne suivante permet de rendre l'attribut `maj` optionnel, avec une valeur par défaut au 11 octobre 2003 s'il n'apparaît pas

```
<xsd:attribute name="maj" type="xsd:date" use="optional"
default="2003-10-11"/>
```

Quand l'attribut `fixed` est utilisé, la seule valeur que peut prendre l'attribut déclaré est celle de l'attribut `fixed`.

Exemple 3.10

Nous donnons le schéma correspondant de l'exemple 3.1 :

```
<xsd:element name =”parent ” type =”parent-type”>
```

```
<xsd:complextype name =”parent-type”>
<xsd:sequence>
<xsd:element name =”garcon ” type =”xsd:string”>
<xsd:element name =”fille ” type =”xsd:string”>
</xsd:sequence>
</xsd:complextype>
</xsd:element>
```

Le tableau 3.4 présente une comparaison entre le format DTD et le Schéma XML

DTD	use	default	Commentaire
# REQUIRED	required	-	L’attribut est obligatoire
”blabla”#REQUIRED	required	blabla	L’attribut est obligatoire avec une valeur par défaut ”blabla”
#IMPLIED	optional	-	L’attribut est optionnel
”blabla”#IMPLIED	optional	blabla	L’attribut est optionnel avec une valeur par défaut ”blabla”

Tableau 3.4 : Contraintes d'occurrences fixables par les attributs use et default

4 validation d’un document XML

Le XML impose des règles de syntaxe très spécifiques. Un document est bien formé s’il obéit à ces règles. Un tel document sera correctement traité par un programme adéquat, s’il n’est pas bien formé son traitement provoquera un message d’erreur ou un arrêt de l’application qui le traite.

Un document valide est forcément un document bien formé mais il obéit en plus à une structure type définie dans une DTD, ou dans un schéma XML. Donc nous pouvons dire qu’une même instance XML peut être valide par rapport à ces deux modèles

Pour fournir la validation, nous avons besoin de :

1. Un schéma.
2. Une référence dans le document au fichier de définition de schéma.

Exemple 3.11

Nous avons le fichier XML suivant :

```
<?xml version = "1.0" encoding = "UTF-8"?>

<voitures  xmlns:xsi  =  http://www.w3.org/2001/XMLSchema-instance
xsi:noNamespaceSchemaLocation ="exemple.xsd">

       <marque> Renault </marque>
       <marque> Honda </marque>
       <marque> Mercedes </marque>
</voitures>
```

Ce fichier est validé par rapport à un fichier schéma XML appelé exemple.xsd, définit comme suit :

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:exemple
    xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
    <xsd:element name = "voitures">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name = "marque"
                    type = "xsd:string"
                    maxOccurs = "unbounded"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
</xsd:exemple>
```

Nous trouvons dans l'élément racine `<xsd:exemple>` du schéma, l'attribut `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`. Cela signifie que dans le document, tous les éléments commençant par `xsd` sont référencés à cette URL. On fait référence au schéma dans le fichier XML en utilisant l'attribut : `xsi:noNamespaceSchemaLocation`. C'est-à-dire que le schéma se trouve dans exemple.xsd.

Chaque élément du fichier XML trouve sa définition dans le document schéma XML (voir figure 3.3), l'élément `<voitures>` est définie au niveau du schéma par le bais de

l'élément schéma XML <xsd:element>, ainsi que les balises ayant le nom marque sont définie dans le schéma XML en utilisant le deuxième élément <xsd:element>, le nombre d'occurrence de l'élément marque est spécifié par l'attribut maxOccurs du deuxième élément <xsd:element>.

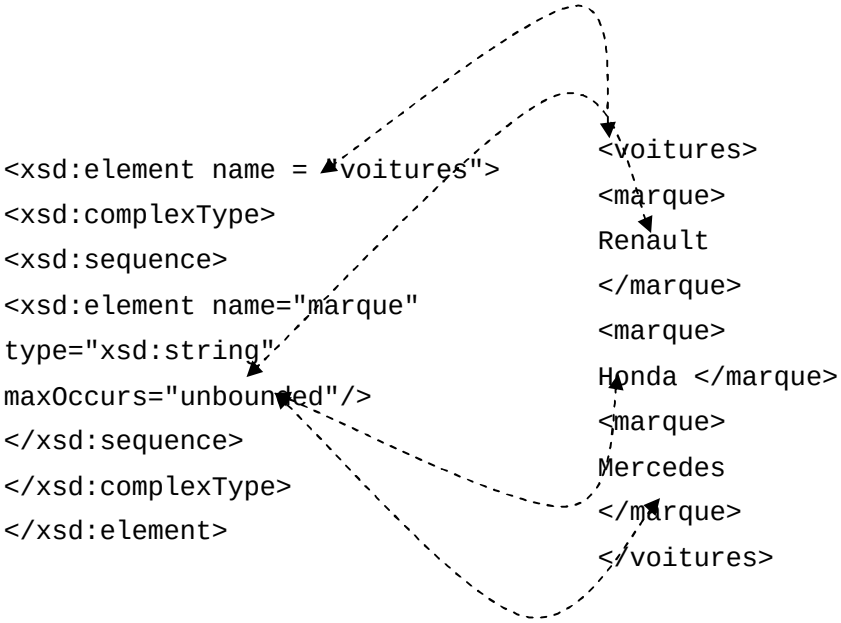


Figure 3.3 : Un exemple de validation d'un fichier XML par rapport à son schéma XML

La figure 3.4 donne une vue plus adaptée de document schéma XML sous forme d'arborescence, fournie par XML Spy, un éditeur de schéma XML. Elle fait apparaître les attributs portés par les différents éléments de l'arborescence.

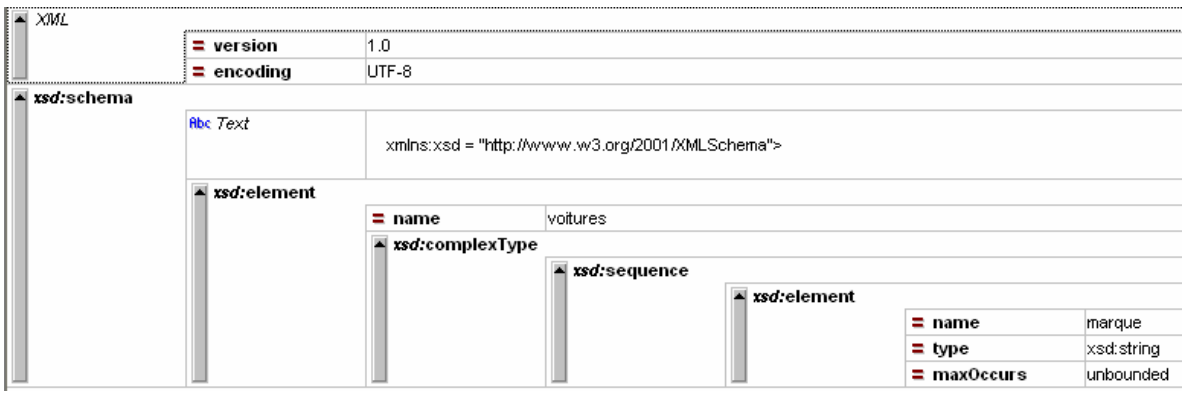


Figure 3.4 : Représentation graphique de notre schéma exemple dans l'éditeur XML Spy

5 Conclusion

Le XML est une véritable révolution dans le panorama des langages de publication sur le Web. Il apparaît comme incontournable car il est déjà à la base de toute une série de nouveaux langages qui sont ou qui seront utilisés dans la conception des pages Internet comme le XHTML, le successeur désigné du Html, le WML pour le Wap des téléphones mobiles, le MathML pour les mathématiques, le SOAP et bien d'autres [Lun 01].

Nous avons donné dans ce chapitre la structure générale d'un fichier XML. Nous avons défini les composants essentiels d'un document XML.

Nous avons aussi indiqué les deux types de grammaires permettant de générer un document XML (DTD et schéma XML). La validation d'un document XML par rapport à son schéma est aussi évoquée dans ce chapitre du moment que nous allons la formaliser dans le prochain chapitre.

Chapitre IV

Un Support Sémantique pour XML

1 Introduction

La logique de réécriture a été introduite comme un modèle unifié de concurrence dans laquelle plusieurs modèles des systèmes concurrents sont représentés. Ce fait a été étendu par Marti-Oliet et Meseguer [MM 93] à l'idée de considérer la logique de réécriture comme un cadre unificateur logique aussi. Beaucoup d'autres logiques largement différentes dans la nature, comme la logique propositionnelle, la logique linéaire, et la logique équationnelle, sont représentées dans cette logique de façon naturelle et directe. De plus, la logique de réécriture a une portée assez large sur la définition de la sémantique des langages de programmation et les générateurs de programmes [Bos 94, Bru 01].

Dans ce contexte, nous suggérons une nouvelle application de la logique de réécriture via son langage Maude. Il s'agit d'associer, au langage d'échange de données XML, un modèle sémantique permettant de prendre en compte les aspects logiques et conceptuels des documents XML manipulés.

En fait, la sémantique de XML est implicitement exprimée dans les documents XML. L'inconvénient de cette sémantique implicite est quelle est ambiguë [Usc 03].

De plus, actuellement, le langage XML souffre de limites liées aux contraintes impliquées par le type d'application nouvelles du Web.

Le Web sémantique d'aujourd'hui, s'est beaucoup orienté vers la description de l'information structurelle sémantisée et prend peu en compte l'étude du fonctionnement des services décrits [Por 04]. Le modèle sémantique que nous proposons permettra de décrire des documents XML à la fois structurés et dynamiques. Il intègre dans un même

formalisme, la structure et le fonctionnement d'un composant actif du web. Au contraire dans les modèles utilisés jusqu'à présent la description du service est séparée de l'information manipulée.
L'objectif de ce travail est de répondre aux limites de XML par la spécification formelle des documents XML ainsi que les grammaires les générant. Cette modélisation a pour but de définir, d'une part la structure générique d'une classe de documents, c'est à dire les règles de composition des structures admises pour un document spécifique. D'autre part, la phase de validation permet de déterminer si oui ou non un document XML est valide par rapport au modèle qui lui est associé.

Ce chapitre comporte deux parties, la première présente la sémantique de XML à travers les différentes tentatives de formalisation du ce langage. En suite, nous présentons l'intégration de XML dans la logique de réécriture.

La deuxième partie concerne la validation des documents XML. Nous présentons premièrement les différents processus de validation qui existent dans la littérature, par la suite les deux approches (simples et paramétrées) concernant la programmation de la validation en Maude sont alors exposées.

2 Sémantique de XML

2.1 Travaux antérieurs

Dans la littérature, la sémantique de XML est considérée sous plusieurs angles. Un travail inspiré de l'analogie existante entre les documents XML et les chaînes de caractères générées par une grammaire BNF, a permis d'ajouter des attributs et des fonctions sémantiques aux éléments XML [PC 99].

P. Patel-Schneider et J. Siméon ont proposé l'idée de Yin/Yang Web, basée sur le développement d'un modèle théorique pour XML, XQuery1.0 et XPath 2.0. Il fournit un modèle unifié pour XML et RDF à la fois [PS 03 b].

Il existe d'autres travaux qui proposent des langages pour exprimer la sémantique des documents XML, particulièrement nous faisons référence au travail du R. Worder dont lequel il propose le langage MDL (Meaning Definition Language) qui définit la sémantique XML par des modèles conceptuels, utilisant UML [Wor 02].

Nous citons aussi le langage XSDL (XML Semantics Definition Langage) donné par S. liu, J. Mei et Z. lin, qui sert à exprimer les intentions des auteurs en proposant un modèle théorique et sémantique pour XML, en utilisant le OWL DL. XSDL définit la sémantique de XML en faisant un mapping de XML aux ontologies [She 04]. Nous pouvons trouver d'autres efforts dans ce sens : M. Erdmann et R. Studer [ES 01] présentent un outil pour générer les DTD à partir d'une ontologie, et les éléments d'un document XML qui respectent cette DTD peuvent être transformés en des concepts et propriétés dans l'ontologie. B. Amann proposent des règles pour transformer des fragments XML en une ontologie générale [Ama 02].

Dans ces différents travaux évoqués, nous constatons que toutes ces tentatives de formalisation de XML n'utilisent pas des modèles formels. De plus, elles tiennent compte de la description de l'information structurelle sémantisée uniquement, en ajoutant des attributs ou des fonctions pour exprimer le sens des éléments XML.

Par contre, le modèle sémantique, basé sur la logique de réécriture, que nous proposons permettra de décrire des documents XML à la fois structurés et dynamiques.

2.2 Intégration de XML dans la logique de réécriture

Actuellement, le langage XML souffre de limites liées aux contraintes impliquées par le type d'applications nouvelles du Web. D'une part, la partie sémantique de XML est très pauvre, d'autre part, XML n'est pas approprié pour représenter les informations (services, agents, etc) dans la génération du Web sémantique.

Nous montrons, dans cette section, comment la puissance d'expression de la logique de réécriture et sa flexibilité permettent de définir :

- Les aspects structurels des différents composants du langage XML. Donc, nous formalisons l'aspect syntaxique des documents XML et des documents schéma XML en termes de théories équationnelles. A la base de ce travail, nous pouvons étendre le sous langage XML considéré par d'autres fonctionnalités et d'autres puissances. Nous pouvons facilement généraliser cette spécification à n'importe quel langage décrivant des informations structurées.

- Les aspects dynamiques des entités décrites par XML et qui ne sont pas, jusqu'à présent, considérées par ce langage. Dans ce modèle sémantique de XML, les pages Web munies de représentation, non seulement structurelle sémantisée mais aussi de

nature fonctionnelle, peuvent être naturellement décrites en termes de modules systèmes dotés de règles de réécriture qui formalisent le fonctionnement possible de ces entités.

Les spécificités du langage Maude nous laissent associer aux différents composants syntaxiques de XML, et de son langage de grammaire XML schéma, une spécification simple, concise, lisible, extensible et exécutable [DB 06].

2.2.1 La formalisation de documents XML

Nous présentons dans cette section, la définition formelle de tous les composants d'un document XML sous forme de théories fonctionnelles. Pour plus de clarté, nous avons jugé important de donner ces théories en utilisant le code Maude.

Pour les documents XML, nous avons déjà noté, dans le chapitre précédent, que le principal constituant syntaxique de ces documents est "l'élément". Par conséquent, nous donnons un module Maude qui présente la spécification des éléments XML. Ce module est appelé `ELEMENT` et donné par la figure 4.1.

```
fmod ELEMENT is

including STRING BOOL ATTRIBUTS .

sorts Element Sub-elements.
subsort Element < Sub-elements.

Ops
e-vide : -> Element .                                          (01)
cst(_) : String  -> Element .                                  (02)
<_,_,_> : String  Attributs Sub-elements -> Element .          (03)
_/_:Sub-elements Sub-elements -> Sub-elements .[ACI:e-vide]    (04)
Name(_): Element -> String .                                   (05)
Att(_) : Element -> Attributs .                                (06)
Sub(_) : Element -> Sub-elements .                             (07)
Acces-elem (_,_) : String Sub-elements -> Sub-element .        (08)
_is-const : Element -> Bool .                                  (09)

var x: String .
var y: Attributs .
```

```
var z: Sub-elements .
var e: Sub-element .


eq Name(<x , y , z>)= x .
eq Att (<x , y , z>)= y .
eq Sub (<x , y , z>)= z .
eq Acces-elem (x ,e-vide) = e-vide .
ceq Acces-elem (x , e / z) = e if x == name (e)
                                else Acces-elem(x,z).
eq cst(x) is-const = true .
eq e is-const = false .
endfm
```

Figure 4.1: Le module `ELEMENT`.

Rappelons qu'un fichier XML peut contenir des éléments avec ou sans attributs, et chaque élément peut avoir des élément fils ou un contenu texte, on peut suggérer l'opération notée `<_,_,_>` pour générer un élément, cette opération a trois arguments. Le nom d'un élément représente le premier argument, il est de type *String* (prédéfini dans Maude), le deuxième argument représente les attributs, (de type *Attributs)*, importés à partir du module `ATTRIBUTS` (figure 4.2), le dernier est de type *Sub-elements* qui représente le contenu d'un élément. Cette opération va retourner la sorte *Element*. L'élément racine est considéré comme un élément simple. L'élément vide est donné par l'opération `e-vide`.

Le contenu d'un élément peut être un texte ou des élément fils. Si le contenu est un texte, nous offrons l'opération `cst(_)`, qui présente un seul argument de type *String*. L'opération `_/_` est utilisée si le contenu d'un élément contient un ou plusieurs éléments fils.

Pour accéder aux diverses structures d'un élément XML, nous proposons un ensemble d'opérations, une opération appelée `Name(_)` permet d'extraire le nom d'un élément, une autre opération notée `Att(_)` pour trouver ses attributs, et une autre nommée `Sub(_)` donne le contenu de cet élément.

L'opération `elem-Acces(_,_)` a deux arguments, le premier de type *String* représente le nom de l'élément cherché, le deuxième de type *Sub-elements* représente l'ensemble des éléments dans lequel la recherche est effectuée.

L'opération `_is-const` permet de vérifier si le contenu d'un élément est un texte ou non.

Nous proposons un module Maude séparé appelé ATTRIBUTS pour spécifier les attributs des éléments XML (Figure 4.2).

```
fmod ATTRIBUTS is

including STRING .
sorts Attribut  Attributs .
subsort Attribut < Attributs .

ops
∅ : -> Attribut .                                    (10)
_ [ _ ] : String String   -> Attribut .              (11)
_ _ : Attributs Attributs -> Attributs . [ACI:∅]      (12)
att-name(_):Attribut -> String .                     (13)
att-val(_) :Attribut -> String .                     (14)
Acces-att (_,_) : String Attributs -> Attribut .     (15)

vars x y: String .
var e : Attribut .
var S : Attributs .

eq att-name(x [y]) = x .
eq att-val(x [y]) = y .
eq Acces-att(x, ∅)= ∅ .
ceq Acces-att (x, e S) = e if x == att-name(e)
                         else Acces-att(x,S).

endfm
```

Figure 4.2: Le module ATTRIBUTS

L'opération `_[_]` sert à la génération un attribut. Cette opération possède deux arguments le premier représente le nom de l'attribut, le deuxième représente sa valeur. Ces deux

arguments sont de type *String*. L'opération notée _ _ génère plusieurs attributs. Et finalement l'ensemble d'attributs vide est présenté par l'opération ∅.

Pour trouver le nom et la valeur d'un attribut nous suggérons respectivement les opérations `att-name(_)` et `att-val(_)`.

L'opération `Acces-att(_,_)` permet d'accéder à un attribut dont le nom est spécifié dans le premier argument de l'opération, parmi un ensemble d'attributs (deuxième argument).

Exemple 4.1

Nous considérons le document XML suivant, qui comporte un élément racine appelé `parent`, et un élément fils appelé `children`.

```
<Parent>
  <children > Ali </children>
</Parent>
```

Nous associons à ce document un terme algébrique respectant la signature proposée précédemment :

`<parent, ∅,<children , ∅, cst(Ali)>>.`

Les opérations principales intervenant dans la génération de ce terme algébrique sont illustrées dans le tableau 4.1:

`<parent,_,_>`	L'application de l'opération numéro 03
`<parent,∅, _ >`	L'application de l'opération numéro 10
`<parent,∅, <children ,_,_>>`	L'application de l'opération numéro 3
`<parent,∅, <children , ∅ ,_>>`	L'application de l'opération numéro 10
`<parent,∅,< children , ∅, cst(Ali)>>`	L'application de l'opération numéro 02

Tableau 4.1: Les opérations principales intervenant dans la génération du terme algébrique de l'exemple.

Si nous voulons exprimer quelques requêtes XML simples concernant le nom d'un élément, ses attributs, ou ses éléments fils par exemple, nous appliquons essentiellement les opérations : `Name(_)`, `Att(_)`, et `Sub(_)` comme suit :

```
Name(<parent,∅,<children, ∅,cst(Ali)>>) = "parent".

Att(<parent,∅, < children , ∅, cst(Ali)>>) = ∅ .

sub(<parent,∅,<children, ∅ ,cst(Ali)>>) = <children,
∅,cst(Ali)>).
```

Si nous voulons, par exemple, savoir le nom du l'élément fils, nous appliquons la requête suivante :

```
Name(sub(<parent,∅,<children, ∅,cst(Ali)>>)) = " children ".
```

Et si nous voulons raffiner un peu plus, et accéder aux attributs de l'élément fils nous appliquons :

```
Att(sub(<parent,∅,<children, ∅,cst(Ali)>>)) = ∅.
```

Pour appliquer l'opération `Acces-elem`, nous nous donons la requête suivante:

```
Acces-elem ( children , Sub ( <parent, ∅,<children, ∅,cst(Ali)>>))
```

Le résultat de cette opération sera `<children, ∅,cst(Ali)>` (c'est dans ce cas le seul élément qui constitue la liste des sous éléments).

2.2.2 La formalisation des fichiers schéma XML

La syntaxe d'un fichier schéma XML est identique à celle d'un fichier XML à l'exception de quelques différences. Par conséquent les modules spécifiant la syntaxe d'un fichier schéma XML sont quasi similaires a ceux d'un fichier XML.

Le module spécifiant un élément d'un document schéma XML est présenté par la Figure 4.3:

```
fmod SH-ELEMENT is

including IDENTIFIERS ATTRIBUTS .

sorts Shelement Sub-shelements .
subsort Shelement < Sub-shelements .
ops
sh-vide : -> Shelement .                                    (16)
<_,_,_> : Id Attributs Sub-shelements -> Shelement .        (17)
_+_ :Sub-shelements Sub-shelements -> Sub-shelements.       (18)
```

```
[ACI :she-vide]                                                    (19)
Name-sh(_):Shelement -> Id .                                       (20)
Att-sh(_):Shelement-> Attributs .                                  (21)
Sub-sh(_):Shelement ->Sub-shelements .                             (22)
Acces-shelem(_,_):Id Sub-shelements -> Sub-shelement.              (23)
reste(_,_):Sub-shelements Sub-shelement -> Sub-shelements.         (24)

var x : Id .
var y    : Attributs .
var Z    : Sub-shelements .
var H S  : Sub-shelement .

eq Name-sh(< x , y , z >) =  x .
eq Att-sh(< x , y , z >)  =  y .
eq Acces-shelem (x, sh-vide) = sh-vide .
eq Sub-sh(< x , y , z >)  =  z .
ceq Acces-shelem (x, H + Z) = H if name-sh(H)== x
                                else Acces-shelem (x, Z).
eq reste ( H ,sh-vide) = sh-vide .
eq reste (S + Z , H) = S + reste (Z , H) .
eq reste( H + Z , H ) = Z .

Endfm
```

Figure 4.3: Le module SH-ELEMENT .

Comme les balises des documents schéma XML sont prédéfinies, leurs noms sont prédéfinis aussi, c'est pourquoi nous remplaçons la sorte *String* dans la description des éléments schéma XML par une nouvelle sorte appelée *Id* importé du module IDENTIFIERS (Figure 4.4). Cette dernière est utilisée pour nommer de façon particulière les éléments d'un shéma XML.

Le contenu d'un élément dans les documents schéma XML ne peut pas être un texte, mais seulement des éléments fils.

Notons que nous réutilisons le même module utilisé pour spécifier les attributs dans le cas des documents XML.

```
fmod IDENTIFIERS is

including STRING BOOL.

sorts Id .

ops
 xsd:element : -> Id .                    (25)
 xsd:complextype :->Id .                  (26)
 xsd:sequence : -> Id .                   (27)
 xsd:attribute : -> Id .                  (28)
 _ is-Id-type : Striing -> Bool .         (29)

var X : String .

eq "xsd:string" is-Id-type = true .
eq "xsd:float" is-Id-type = true .
eq "xsd:binary" is-Id-type = true .
eq "xsd:decimal" is-Id-type = true .
eq "xsd:integer" is-Id-type = true .
eq "xsd:nonpositiveInteger" is-Id-type = true .
eq "xsd:nonnegativeInteger" is-Id-type = true .
eq "xsd:positiveInteger" is-Id-type = true .
eq "xsd:negativeInteger" is-Id-type = true .
eq "xsd:long" is-Id-type = true .
eq "xsd:int" is-Id-type = true .
eq "xsd:short" is-Id-type = true .
eq "xsd:byte" is-Id-type = true .
eq "xsd:date" is-Id-type = true .
eq "xsd:month" is-Id-type = true .
eq "xsd:year" is-Id-type = true .
eq "xsd:century" is-Id-type = true .
eq X is-Id-type = false .
endfm
```

Figure 4.4: Le module IDENTIFIERS.

Dans ce module nous proposons aussi une opération vérifiant si une chaîne de caractères donnée correspond à un type prédéfini (`xsd:string, xsd:float`, etc) déclaré comme valeur des attributs dans le schéma XML.

Exemple 4.2

Nous illustrons notre spécification à travers le document schéma XML suivant :

```
<xsd:element name="parent">
 <xsd:complextype>
  <xsd:sequence>
   <xsd:element name="children" type="xsd :string"/>
  </xsd:sequence>
 </xsd:complex type>
</xsd:element>
```

Ce document présente le modèle schéma XML du document XML donné au niveau de l'exemple 4.1. En utilisant les modules Maude `ATTRIBUTS`, `SH-ELEMENT`, et `IDENTIFIERS` nous pouvons obtenir une expression concise et une notation claire de notre document schéma XML exprimée par le terme algébrique suivant :

```
<xsd:element , name[parent], <xsd:complextype , ∅,
<xsd:sequence,∅,<xsd:element, name[children]type[xsd:string],
e_vide>>>>
```

2.2.3 Résultats Sémantiques

À travers les différents modules présentés dans cette section, nous constatons que nous avons donné une spécification modulaire des documents XML, et des documents schéma XML. Donc, nous pouvons facilement enrichir cette spécification, et ajouter d'autres opérations, sortes, équations ou même des modules pour spécifier les différents aspects syntaxiques non considérés dans notre spécification.

Par l'implémentation de ces modules dans le langage Maude, nous obtenons des spécifications exécutables, à travers lesquelles nous réalisons le prototypage des documents XML, et des documents schéma XML.

1. En se basant sur les travaux théoriques de la logique de réécriture [Mes 92], XML peut bénéficier d'un fondement mathématique bien défini, en décrivant son aspect syntaxique comme étant des théories équationnelles. Chaque théorie de réécriture en générale, possède

un modèle mathématique appelé catégorie, et comme les théories équationnelles sont un cas particulier des théories de réécritures, donc elles possèdent également un modèle mathématique qui est une catégorie particulière appelée algèbre.

Par exemple à la théorie équationnelle ELEMENT, décrivant les principales structures syntaxiques du langage XML, nous associons un modèle mathématique sur lequel nous pouvons raisonnr de façon logique.

Chaque document XML qui obéit à la syntaxe décrite par la théorie ELEMENT est un terme algébrique appartenant à l'algèbre : $\sum_{ELEMENT}$ -Algèbre.

Les équations de cette théorie permettent de former des classes d'équivalences dans cette algèbre. Deux documents XML D1 et D2 appartenant à une même classe d'équivalence peuvent être différents dans leurs structures (syntaxiquement différents) mais sémantiquement ils ont le même sens (voir figure 4.5).

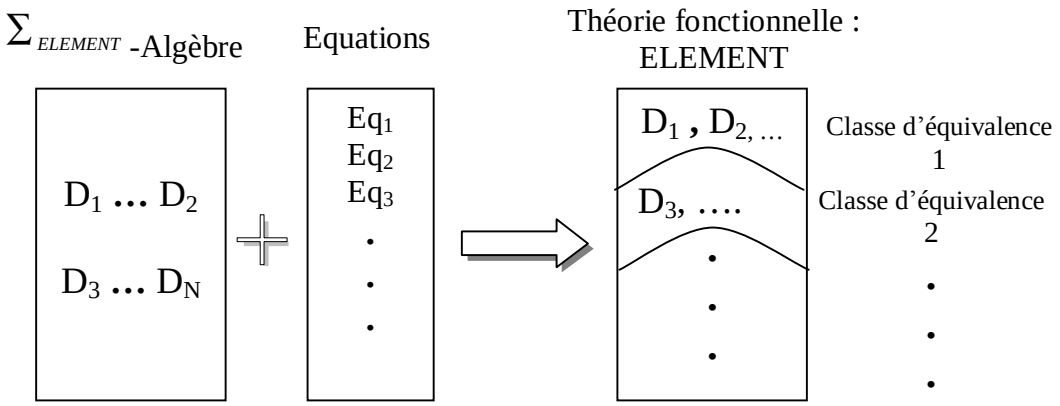


Figure 4.5 : Le modèle mathématique de `ELEMENT`

Touts les résultats mathématiques (fusion, composition, etc) concernant les algèbres et les opérations qui y sont définies peuvent être appliqués à cette algèbre particulière.
Nous pouvons facilement déduire, par exemple que :

Si : D_1 = <personne , age[45] ,<nom, ø ,cst (Ali)>/<prenom , ø ,cst (Moussaoui)> >

Et

D_2= < personne , age[45] , <prenom , ø ,cst (Moussaoui)> / <nom, ø ,cst (Ali)> >

Alors selon la théorie équationnelle ELEMENT, D_1 et D_2 sont équivalents (l'opérateur / est commutatif) sémantiquement, mais syntaxiquement différents.

Si D_1 et D_2 sont des documents XML, alors la fusion de D_1 et D_2 (notée D_1 o D_2), peut être naturellement démontrée, c'est aussi un document XML, ayant un nom, un ensemble d'attributs vide, et le contenu sera D_1 / D_2. Dans l'exemple précédent:

D_1 o D_2 = <liste, ø , <personne , age[45] ,<nom, ø ,cst (Ali)>/<prenom , ø ,cst (Moussaoui)> >/< personne , age[45] , <prenom , ø ,cst (Moussaoui)> / <nom, ø ,cst (Ali)> >>

Cette opération hérite toutes les proriétées sémantiques de l'opération fusion dans uns algèbre.

Ce même résultat est applicable pour les documents schéma XML.

2. A la base de cette description formelle de la syntaxe du XML, nous pouvons réaliser plusieurs applications, nous pouvons, comme nous l'avons déjà mentionné, réaliser une extension du XML pour décrire le comportement dynamique à travers les règles de réécriture. A chaque document XML, nous pouvons rajouter une composante dynamique décrivant l'ensemble des transitions locales pouvant être exécutées par l'élément décrit dans ce document ainsi que l'état actuel dans lequel se trouve cet élément.

Les éléments fils d'un fichier XML sont aussi des documents XML ayant la même caractéristique. Leurs états actuels réunis constituent l'état du document racine. Les règles de fonctionnement de ce dernier sont alors déduites à partir des règles de fonctionnement de chacun de ces sous éléments.

La logique de réécriture est un formalisme qui s'adapte bien à la spécification du comportement d'un système (élément XML) qui évolue d'un état à un autre en exécutant des actions (règles de réécriture) propre à ce système.

3. De plus, en formalisant ainsi XML, nous pouvons bénéficier aussi des outils formels existants dans Maude pour la vérification et la preuve. Un document XML possède un modèle mathématique sur lequel nous pouvons exprimer et vérifier formellement des propriétés. Si nous nous donnons un fichier XML décrivant les employeurs d'une entreprise avec leurs grades respectifs et nous formulons la propriété par exemple, "existe-t-il un employé de grade Directeur régional dans cette entreprise?"

Ou bien encore, "il n'existe jamais d'employés de grades différents et percevant le même salaire".

L'expréssion de la première propriété est généralement:

$\exists E_i \in$ Sub (Entreprise) / Att (E_i) contient un attribut "grade" dont sa valeur est "Directeur régional". Sachant que l'élément Entreprise a la forme d'un terme algébrique.

L'outil model-checker de Maude peut naturellement prouver ce type de propriété (codifiées en logique temporelle). Des résultats plus significatifs peuvent être établis en considérant l'aspect dynamique des éléments XML décrits.

Un certain ensemble de propriétés concernant ce type de fichiers pouvait être aussi déduit en répondant à des requêtes telle que:

- Déterminer l'ensemble des employés qui occupent le poste "chef chantier".
- Déterminer l'ensemble des employés qui percoivent un salaire supérieur à 10.000 DA.

4. Une autre application possible sur cette description est bien la validation des documents XML par rapport aux fichiers schéma XML, qui est présentée en détail dans la section suivante.

3 Validation de fichiers XML

3.1 Travaux existants

Les documents incluant les différents modèles de validation, sont simplement des fichiers texte, donc ils sont faciles à comprendre. Il existe plusieurs classes de travaux qui aident ces fichiers dans le processus de validation, parmi ces classes nous citons les deux parseurs spécifiques de Java : JAXP, et Xerces Java.

Un parseur doit interpréter n'importe quel document XML avant qu'une application ne puisse l'utiliser, il prend chaque caractère du document et se décide si c'est un élément, attribut, ou une chaîne de caractères, etc. La validation commence quand le parseur vérifie aussi la structure du document par rapport à une DTD ou un schéma. Cependant, le parseur fera seulement cela s'il a été configuré pour le faire.

Il existe une autre approche dans laquelle les documents XML sont vus comme un arbre d'arité non bornée qui doit respecter un ensemble de contraintes. Cette approche propose une méthode de validation incrémentale des documents XML par rapport aux contraintes d'intégrité [Abr 05 b]. La vérification est faite par un parcours dans l'arbre XML, en visitant chaque noeud exactement une fois. Les contraintes d'intégrités sont des clés représentées par une grammaire d'attributs dont les règles de production sont définies par le schéma. Les attributs représentent des informations sur les clés. Le processus de validation

revient à calculer les valeurs de ces attributs pendant le parcours de l'arbre. Un arbre XML est valide lorsque le processus termine (la racine de l'arbre XML est atteinte) et que les attributs définis pour la racine ont tous la valeur vraie [Abr 05 a].

Une dernière approche, considère la validation XML comme un problème de correspondance d'expression régulière. Étant donné une expression régulière e et une chaîne de caractère s, le problème à résoudre sera : s dans L (e) ? (Ici L (e) dénote le langage accepté par e). La technique la plus utilisée pour résoudre ce problème est basée sur des automates finis. Il y a un autre algorithme, basé sur les dérivations d'expressions régulières, implémenté par le programme Haskell. [Eng 99]

Il existe alors plusieurs méthodes de validation de documents XML par rapport à leur grammaire spécifiée dans un fichier schéma XML. D'où, une multitude d'outils de validation tels que : XMLSpy, Oxygène, Xmllint, et XML EditorAdd, et bien d'autres [Oba 04, WP 04].

L'approche de validation que nous proposons se base sur le modèle sémantique que nous avons associé aux fichiers XML et schéma XML. Ainsi la validation est réalisée indépendamment des problèmes liés à l'implémentation des documents manipulés et leurs grammaires correspondantes. Elle se résume à l'ajout de quelques règles de réécriture décrivant son processus. En effet, l'avantage majeur de notre approche est l'intégration dans un même formalisme, logique de réécriture, la structure sémantisée des documents XML ainsi que leur validation par rapport à une grammaire donnée.

3.2 Approche de validation adoptée

Les documents XML et les documents schéma XML sont naturellement traduits aux modules Maude, et donc ils peuvent être des données pour des applications Maude. Ainsi, il devient beaucoup plus simple de réaliser, d'une façon rigoureuse, l'axiomatisation de la correspondance entre les deux type des documents XML.

3.2.1 Processus de validation

Le processus de validation suit la règle suivante :

« *Chaque élément d'un document XML doit correspondre à une définition présentée dans le schéma XML par des balises prédéfinies* »

Nous considérons l'exemple 4.1, et son schéma XML présenté dans l'exemple 4.2. Nous pouvons schématiser la correspondance de ce document XML par rapport à son schéma XML, en donnant la figure 4.6. Dans cette figure, le schéma XML est représenté à gauche et le document XML est représenté à droite.

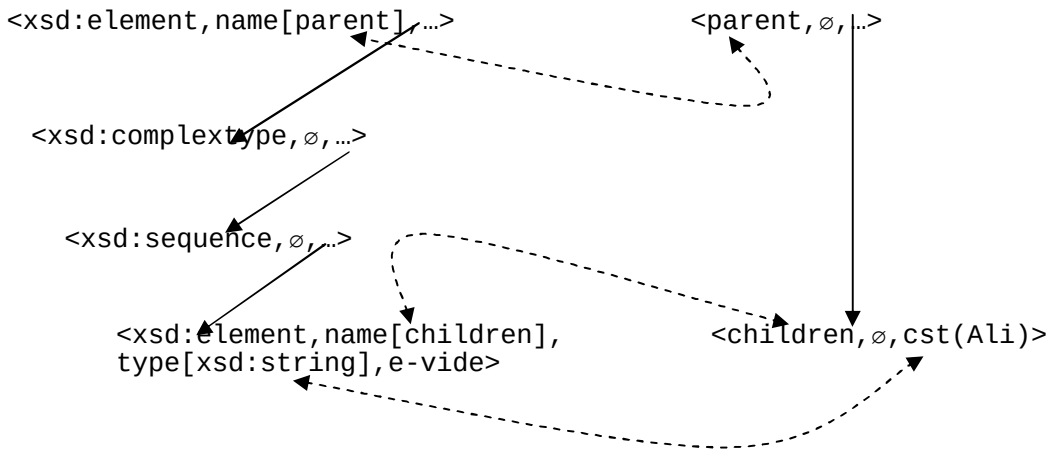


Figure 4.6:Un exemple de correspondance entre un document XML et un schéma XML.

Selon cet exemple simple la racine `parent` correspond à l'attribut de l'élément nommé xsd:element, l'élément fils `children` correspond à l'attribut du deuxième élément xsd:element. Nous notons ici aussi qu'il y a une correspondance entre les types des éléments (xsd:string et cst(Ali)).

Nous pouvons déduire que le processus de la validation commence par le document XML et essaye de vérifier la correspondance de ses composants par rapport aux différents définitions et déclarations dans le schéma XML. Par exemple, la valeur du nom d'attribut du premier élément <xsd:element> dans le schéma XML représentera le nom de l'élément racine dans le document XML.

D'une manière générale, nous pouvons écrire un algorithme qui énumére les différentes phases de validation d'un élément XML par rapport à sa définition donnée dans le schéma XML :

Algorithme *validation*

DONNEES

P Un fichier schéma XML S de type : <Nom (S), Att (S), Contenu (S)> sachant que :

- Att (S) st une liste d'attributs élémentaires Att_i (S) chaque Att_i (S) est constitué d'un nom et d'une valeur : Att_i (S)-nom et Att_i (S)-val.
- Contenu (S) est aussi une liste d'éléments élémentaires $Contenu_i$ (S), chacun ayant aussi la forme d'un élément XML c à d :
  <Nom ($Contenu_i$ (S)), Att ($Contenu_i$ (S)), Contenu ($Contenu_i$ (S))>
- Nom (S) est une chaîne de caractère prédéfinie dans XML.

P Un document XML D de type : <Nom (D), Att (D), Contenu (D)> sachant que :

- Att (D) est une liste d'attributs élémentaires Att_i (D), chaque Att_i (D) est constitué d'un nom et d'une valeur : Att_i (D)-nom et Att_i (D)-val.
- Contenu (D) est aussi une liste d'éléments élémentaires $Contenu_i$ (D), chacun ayant aussi la forme d'un élément XML c à d :
  <Nom ($Contenu_i$ (D)), Att ($Contenu_i$ (D)), Contenu ($Contenu_i$ (D))>
- Nom (D) est une chaîne de caractère non prédéfinie dans XML.

SORTIE

Le document XML D est valide ou non par rapport au fichier S.

Etapes de l'algorithme

1. Vérifier que Nom (S) = xsd:element.
2. Vérifier que Nom (D) est la valeur d'un attribut trouvé parmi la liste Att (S) dont son nom est 'name', c à d si l'attribut trouvé est Att_k (S) dont : Att_k (S)-nom = 'name' et Att_k (S)-val = Nom (D)
3. Pour chaque Att_i (D) de la liste Att (D)

   Vérifier que Att_i (D)-nom est la valeur d'un attribut A, c à d

   Att_i (D)-nom=A-val, sachant que l'attribut A est trouvé parmi la liste d'attributs

Att (contenu$_k$ (S)) où contenu$_k$ (S) est un élément de contenu (S) ayant comme nom xsd:attribute.

autrement dit Contenu$_k$ (S) = <xsd:attribute, Att (Contenu$_k$ (S)), Contenu (Contenu$_k$ (S))>

4. Pour chaque Contenu$_i$ (D) de la liste Contenu (D).

   - Remplacer D par :
     Contenu$_i$ (D) = <Nom (Contenu$_i$ (D)), Att (Contenu$_i$ (D)), Contenu (Contenu$_i$ (D))>
   - Remplacer S par Contenu$_l$ (S).
     Contenu$_l$ (S) n'est pas arbitraire, c'est un élément de la liste d'éléments Contenu (S) dont le nom est xsd:element.
     Et refaire les étapes 2, 3, et 4 de l'algorithme jusqu'à épuisement de la liste des élément Contenu (D).

Nous remarquons que la réalisation de cet algorithme peut être envisagée selon deux vues différentes :

Soit en formalisant les différents étapes de l'algorithme sous forme de règles de réécriture que nous réunissant dans une seule théorie de réécriture, (voir figure 4.7), cette dernière utilise les théories équationnelles décrivant le document XML à valider ainsi que le fichier schéma XML (grammaire de référence).

Soit en exploitant la programmation paramétrée dans Maude pour définir les différents étapes de l'algorithme dans des théories de réécritures modulaires, concises et claires. Ces théories peuvent facilement être réutilisables par d'autres applications, autre que la validation.

3.2.2 Première forme de validation

Le processus de validation présenté dans la section précédente, peut être formalisé sous forme de règles de réécriture, ces règles sont regroupées dans un module système de Maude (figure 4.7) :

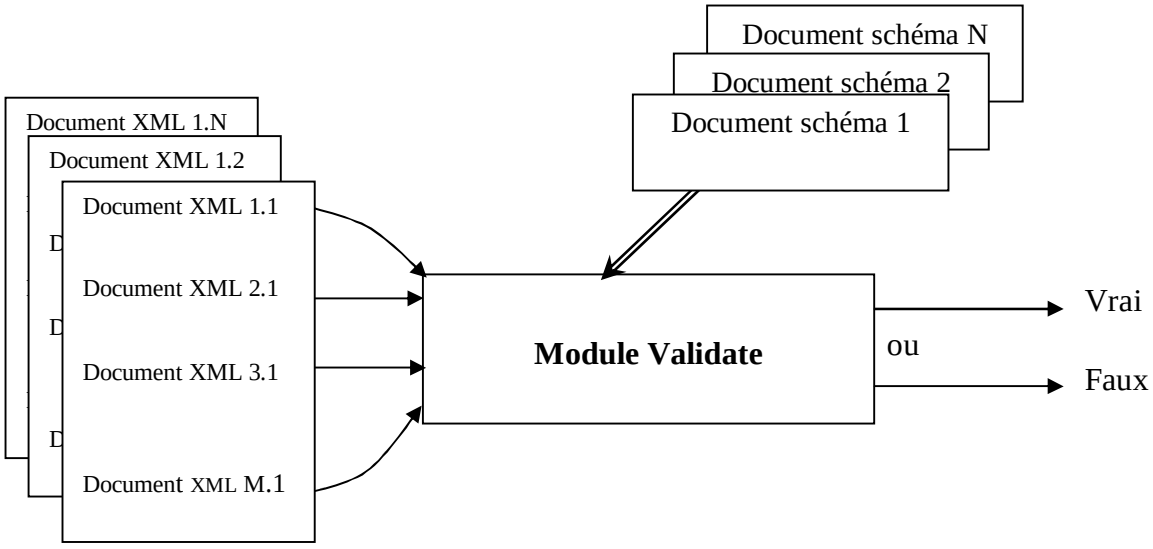


Figure 4.7:Première approche de validation

Le module `validate` contient les différentes opérations, et règles qui participent à la validation d'un élément XML par rapport à un document schéma XML. Il importe principalement les modules `Element`, et `Sh-element`.

L'opération principale dans ce module est appelée `validate`, elle est utilisée pour valider un élément XML par rapport à un élément schéma XML. Cette opération présente deux arguments, le premier est de type `Sh-element` et le deuxième est de type `Element`, et elle retourne un booléen :

```
validate(-,-) : Sh-element Element -> Bool .
```

La spécification de cette opération est définie par la règle de réécriture suivante, qui exprime qu'un document de type élément est valide par rapport à un fichier schéma. `(validate(e,e')=true)` si et seulement si trois conditions sont vérifiées :

Correspondance entre les noms (étape 1 et 2 de l'algorithme), correspondance au niveau des attributs (étape 3 de l'algorithme) et enfin une correspondance des contenus (étape 4 de l'algorithme).

```
crl1 : validate (e,e') à true
     if Name-sh(e) == xsd:element
     and Name-corr(Att-sh(e),Name(e'))
     and Att-corr(e ,Att(e'))
     and Sub-corr(e, Sub(e'))
```

La première condition de cette règle, vérifie d'une part si le nom de l'élément du document schéma XML correspond à `xsd:element`, car la balise prédéfinie de schéma XML responsable à la définition des élément porte le nom `xsd:element`.

D'autre part, nous vérifions dans cette condition que le nom du l'élément XML `"X"` correspond à la valeur d'un attribut appelé `name`, se trouvant parmi l'ensemble d'attributs `"A"` de la balise `xsd:element`, nous exprimons cette condition par l'opération:

```
Name-corr(-,-) : Attributs String -> Bool .
```

```
crl2 : Name-corr(A,X) à true
     if att-val(Acces-att ("name", A)) == X
```

La deuxième condition, exprime la correspondance entre les attributs du l'élément XML, et les attributs déclarés dans le schéma XML. L'opération correspondante accède à la valeur de l'attribut `name` de la balise prédéfinie `xsd:attribute`, et vérifie que cette valeur correspond au nom du premier attribut de l'élément XML `(b)`, après nous continuons de la même manière la vérification des autres attributs restants `(B)` de l'élément XML. Le profil de cette opération est :

```
Att-corr(-,-):Sh-element Attributs à Bool .
```

```
crl3: Att-corr'(e1 , ∅)à true if
                    Acces-shelem (xsd:attribute, e1) == sh-vide
                    else false .
```

Cette règle exprime le cas où l'élément XML n'a pas d'attributs et le document schéma ne contient pas de définition d'attributs `Att-corr` va prendre la valeur `true` si non elle va prendre la valeur `false`.

Les autres cas sont considérés par la règle suivante:

```
crl4 : Att-corr(e, b B) à Type-corr(Att(e), Att(Sub-sh(e)))
                     and Att-corr'(Sub-sh(Sub-sh(e)),b B)
```

```
crl5 : Att-corr'(e1, b B) à true if
      att-name (b)== att-val(Acces-att ("name", Att(Acces-shelem
      (xsd:attribute, e1))).
      and att-corr(reste(e2,Acces-shelem(xsd:attribute,e1) ) ,B ).
      else false .
```

Avant la vérification des attributs, nous devons vérifier le type de l'élément XML, c à d vérifier si la valeur de l'attribut `type` de l'élément schéma `(e)` correspond à la valeur de l'attribut `name` de l'élément `complexType` dans lequel les attributs sont définis. Cette vérification est exprimée par l'opération `Type-corr` :

```
Type-corr(_,_) : Attributs Attributs à Bool .
```

```
crl6 : Type-corr (A,B)à true if Acces-att("type",A)==∅
                          and Acces-att("name",B)== ∅
```

```
crl7 : Type-corr (A,B)à true if att-val(Acces-att ("type",A))==
att-val(Acces-att ("name",B))
```

La dernière condition, consiste à vérifier le contenu de l'élément XML. Et elle est exprimée par l'opération `Sub-corr` :

```
Sub-corr(-,-):Sh-element Sub-elements -> Bool .
```

Dans la règle de réécriture associée, nous vérifions la validation de chaque sous élément de l'élément XML par rapport au sous élément le définissant dans l'élément schéma.

```
crl8 : Sub-corr( e , e2/e3) à Type-corr(Att(e), Att(Sub-sh(e)))
                          and Sub-corr'(Sub-sh(Sub-sh(e)),e2/e3)
```

Comme dans le cas des attributs, nous vérifions premièrement le type de l'élément XML de la même manière.

Si le contenu de l'élément XML `(e4)` est de type prédéfini dans XML, l'attribut `"type"` de l'élément schéma doit être un type prédéfini (`Id-type`)

```
crl9 : Sub-corr'( e1 , e3) à true if e1== Sh-vide and
                    e3 is const and
                    att-val(Acces-att ("type",Att(e1)))is Id-type
                    else false
```

Mais si la valeur de l'attribut `type` correspond à un type défini par l'utilisateur (un autre type défini par la balise `xsd:complexetype`), nous validons dans ce cas les éléments fils de l'élément XML par rapport aux éléments fils de la balise `xsd:sequence`, en faisant un appel récursif à l'opération `validate`.

```
crl10 : Sub-corr' (e1,e2/e3)= true
     if validate (Acces-shelem (xsd:element,Sub-sh(Acces-
     shelem(xsd:sequence,e1))), e2)
     and Sub-corr'(reste(Sub-sh(Acces-shelem(xsd:sequence,e1)),
     Acces-shelem(xsd:element,Sub-sh(Acces-shelem
     (xsd:sequence,e1)))), e3)
```

Le critère d'arrêt de cette opération est exprimé comme suit:

```
crl11: Sub-corr'(e1,∅) = true if Acces-shelem(xsd:element, e1)==
sh-vide    else false
```

L'objectif principal de cette section est d'explorer ce qui pourrait être la base d'une nouvelle approche pour produire un document XML valide à partir d'un document schéma XML en utilisant les règles de réécriture.

Le module Maude `Validate` est le suivant (figure 4.8):

```
mod Validate is
Including Element Sh-element .
ops
validate(-,-) : Sh-element Element -> Bool .                    (30)
Name-corr(-,-) : Attributs String -> Bool .                     (31)
Att-corr(-,-) : Sh-element Attributs à Bool .                   (32)
Sub-corr(-,-) : Sh-element Sub-elements -> Bool .               (33)
Type-corr(_,_) : Attributs Attributs à Bool .                   (34)
Att-corr'(_,_) : Sub-shelements Attributs à Bool .              (35)
Sub-corr'(_,_) : Sub-shelements Sub-elements à Bool .           (36)

var e :Sh-element .
var e':Element .
var e1:Sub-shelements .
var e2:Sub-element .
var e3:Sub-elements .
var b : Attribut .
vars A B : Attributs .
var x : String .

crl1 : validate (e,e') à true
       if  Name-sh(e) == xsd:element  and Name-corr(Att-sh(e),Name(e'))
       and Att-corr(e,Att(e')) and Sub-corr(e, Sub(e')) .

crl2 : Name-corr(A,x)à true if att-val(Acces-att ("name", A)) == x else false.

crl3: Att-corr'(e1 , ∅)à true if
               Acces-shelem (xsd:attribute, e1) == sh-vide
               else false .

crl4 : Att-corr(e, b B) à Type-corr(Att(e), Att(Sub-sh(e)))
and Att-corr'(Sub-sh(Sub-sh(e)),b B).

crl5 : Att-corr'(e1, b B) à true if att-name (b)== att-val(Acces-att ("name",
Att(Acces-shelem (xsd:attribute, e1))).
and att-corr(reste(e1,Acces-shelem(xsd:attribute,e1) ) ,B ).
else false .

crl6 : Type-corr (A,B)à true if Acces-att("type",A)==∅
and Acces-att("name",B)== ∅.

crl7 : Type-corr (A,B)à true if att-val(Acces-att ("type",A))==
att-val(Acces-att ("name",B)).
```

```
crl8 : Sub-corr( e , e2/e3) à Type-corr(Att(e), Att(Sub-sh(e)))
and Sub-corr'(Sub-sh(Sub-sh(e)),e2/e3)

crl9 : Sub-corr'( e1 , e3) à true if e1== sh-vide and
e3 is-const and
att-val(Acces-att ("type",Att(e1)))is-Id-type
else false.

crl10 : Sub-corr' (e1,e2/e3)= true
if validate (Acces-shelem (xsd:element,Sub-sh(Acces-shelem(xsd:sequence,e1))),
e2)
and Sub-corr'(reste(Sub-sh(Acces-shelem(xsd:sequence,e1)), Acces-
shelem(xsd:element,Sub-sh(Acces-shelem (xsd:sequence,e1)))), e3)

crl11: Sub-corr'(e1, ∅) = true if Acces-shelem(xsd:element, e1)== sh-vide    else
false

endm.
```

Figure 4.8: Module validate

3.2.3 Deuxième forme de validation

La paramétrisation ajoute beaucoup d'intérêts à la programmation. Avec ce concept nous pouvons traiter un problème d'une manière plus générale que le problème posé. Donc nous pouvons améliorer la résistance de la solution aux changements, et favoriser la réutilisation, par conséquent, nous allons minimiser l'effort de la programmation.
Dans ce but, et après avoir testé la faisabilité de la spécification formelle du langage XML, nous avons jugé utile de mettre un peu plus d'éffort pour réaliser une spécification paramétrée réutilisable par d'autres applications.

A) Les fichiers XML

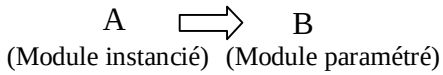
Le module paramétré simple `TRIPLET` (voir figure 4.9), avec trois paramètres de type `TRIV` (la théorie prédéfinie dans Maude) est à la base de la définition formelle d'un élément XML ou XML schéma. Quatre opérations sont suffisantes pour spécifier un triplet d'éléments la première sert à construire une donnée triplet, la deuième, troisième et quatrième opérations retournent respectivement le premier, deuxième et le dernier élément d'un tiplet.

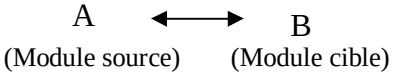
```
fmod TRIPLET (X :: TRIV | Y :: TRIV | Z :: TRIV ) is
sort Triplet .
ops
< _, _, _ > : X@Elt Y@Elt Z@Elt à Triplet .
1er : Triplet à X@Elt .
2eme : Triplet à Y@Elt .
3eme : Triplet à Z@Elt .
var A : X@Elt .
var B : Y@Elt .
var C : Z@Elt .
eqs
1er  (<A,B,C>) = A .
2eme (<A,B,C>) = B .
3eme (<A,B,C>) = C .

endfm
```

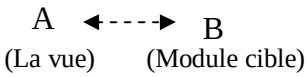
Figure 4.9: Module TRIPLET

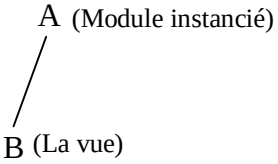
Nous allons instancier le module TRIPLET par un ensemble de vues, pour obtenir un module spécifiant le contenu d'un élément XML (voir figure 4.10).

Notations:









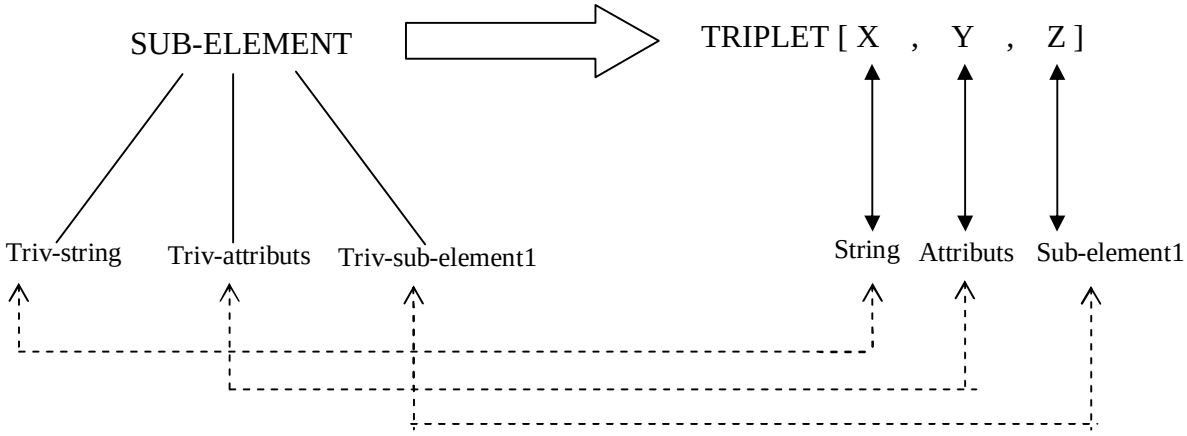


Figure 4.10: Modules et vues nécessaires à la génération de SUB-ELEMENT

D'après la figure ci-dessus, nous constatons que l'instanciation du module `TRIPLET` utilise le module prédéfini `STRING`, le module `ATTRIBUTS` (voir figure 4.12) et le module `SUB-ELEMENT1` comme modules cibles dans les trois vues nécessaires des trois paramètres formelles du module `TRIPLET`.

```
( view Triv-string from TRIV to STRING is
        sort Elt to String .                          endv)
( view Triv-attribut from TRIV to ATTRIBUT is
        sort Elt to Attribut .                        endv)
( view Triv-sub-element1 from TRIV to SUB-ELEMENT1 is
        sort Elt to Sub-element1 .                    endv)
```

Le module résultat est alors (voir figure4.11):

```
fmod SUB-ELEMENT is

protecting TRIPLET ( Triv-string | Triv-attribut | Triv-sub-
element1 ) .

endfm
```

Figure 4.11: Module `SUB-ELEMENT`

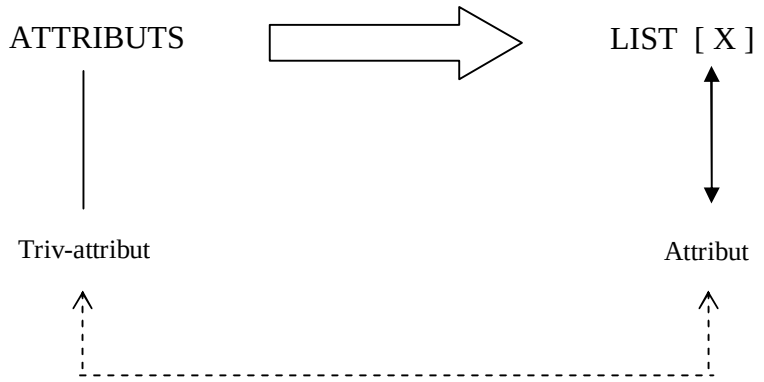


Figure 4.12: Modules et vues nécessaires à la génération de ATTRIBUTS

Etant donné le module Maude appelé `ATTRIBUT` pour spécifier un attribut donné d'un élément XML ou XML schéma (voir figure 4.13).

```
fmod ATTRIBUT is

including STRING .
sort Attribut .
ops
_ [ _ ] : String String à Attribut .
att-name ( _ ) : Attribut à String .
att-val ( _ ) : Attribut à String .
vars E S : String .
eqs
att-name ( E [S] ) = E .
att-val  ( E [S] ) = S .
endfm
```

Figure 4.13: Module `ATTRIBUT`

Pour compléter la spécification des attributs, nous suggérons un module paramétré appelé `LIST` (voir figure 4.14), Pour générer une liste d'attributs. Ce module possède un seul paramètre de type `TRIV`, et un ensemble d'opérations. L'opération ⌀ sert à générer une liste vide, l'opération _ _ génère une liste, `premier` est l'opération qui sert à trouver le

premier élément d'une liste, l'opération `reste'` retire le premier élément d'une liste, et retourne la liste résultante.

```
fmod LIST ( X :: TRIV ) is
sort List .
subsort X@Elt < List .
ops
∅ : à List .
_ _ : List List à List . [ACI: ∅]
premier : List à X@Elt .
reste' : List à List .
vars e1 e2 : List .
var : e : X@Elt .
eqs
premier (e1 e2) = premier (e1) .
premier (e) = e .
reste' (e e1) = e1 .
reste' (e) = ∅ .
endfm
```

Figure 4.14: Module `ATTRIBUT`

Pour avoir le module spécifiant tous les attributs, nous proposons l'instanciation du module `LIST` avec la vue donnée précédemment `Triv-attribut`, et nous rajoutons un ensemble d'opérations permettent de mieux manipuler les listes des attributs XML. Nous donnons l'opération `Access-att (_,_)` pour accéder à un attribut parmi un ensemble d'attributs. Le module estillustré au viveau de la figure 4.15 :

```
fmod ATTRIBUTS is
pr LIST ( Triv-attribut ) .
renaming sort List by Attributs.
ops
Acces-att (_,_) : String Attributs à Attribut .
var x : String .
var e : Attribut .
var s : Attributs .
```

```
eqs
Acces-att (x, ∅)= ∅ .
Acces-att (x, e s)= e if x == att-name(e)
                        else Acces-att(x,s)
Endfm
```

Figure 4.15: Module ATTRIBUTS

Nous présentons aussi, un module Maude appelé SUB-ELEMENT1, (voir figure 4.16) pour spécifier un contenu simple d'un élément XML qui peut être vide ou une chaine de caractère. Nous avons ajouté l'opération –is-const qui sert à tester si le contenu d'un élément est une constante ou non.

```
fmod SUB-ELEMENT1 is
Including STRING BOOL .
Sort Sub-element .
ops
e-vide : à Sub-element .
cst(_) : String à Sub-element.
-is-const: Sub-element à Bool .
var x : String . var e : Sub-element .
eqs
cst(x)is-const = true .
e is-const = false .
endfm
```

Figure 4.16: Module SUB-ELEMENT1

Nous généralisons la spécification de la liste des contenus des éléments XML, en donnant un nouveau module Maude appelé SUB-ELEMENTS (voir figure 4.17), dans lequel nous rajoutons les opérations suivantes: _/_ sert à générer un ensemble d'éléments fils, et l'opération

Access-elem(_,_) sert à accéder à un élément de nom spécifique parmi l'ensemble des éléments fils.

```
fmod SUB-ELEMENTS is
extending SUB-ELEMENT BOOL .
sort Sub-elements .
subsorts Sub-element < Sub-elements .
ops
_/_ : Sub-elements Sub-elements à Sub-elements . [ACI: e-vide]
Access-elem (_,_) : String Sub-elements à Sub-elements .
var x : String . var e : Sub-element . var z : Sub-elements .
eq Access-elem (x,vide) = vide .
eq Access-elem (x,e / z) = e if  1er(e) == x
                          else Acces-elem(x, z).
endfm
```

Figure 4.17: Module SUB-ELEMENTS

Finalement, pour trouver le module Maude spécifiant les éléments XML, nous procédons à l'instanciation du module paramétré TRIPLET (voir figure 4.18), en utilisant les vues suivantes:

```
( view Triv-string from TRIV to STRING is
        sort Elt to String .                              endv)
( view Triv-attributs from TRIV to ATTRIBUTS is
        sort Elt to Attributs .                            endv)
( view Triv-sub-elements from TRIV to SUB-ELEMENTS is
        sort Elt to Sub-elements .                              endv)
```

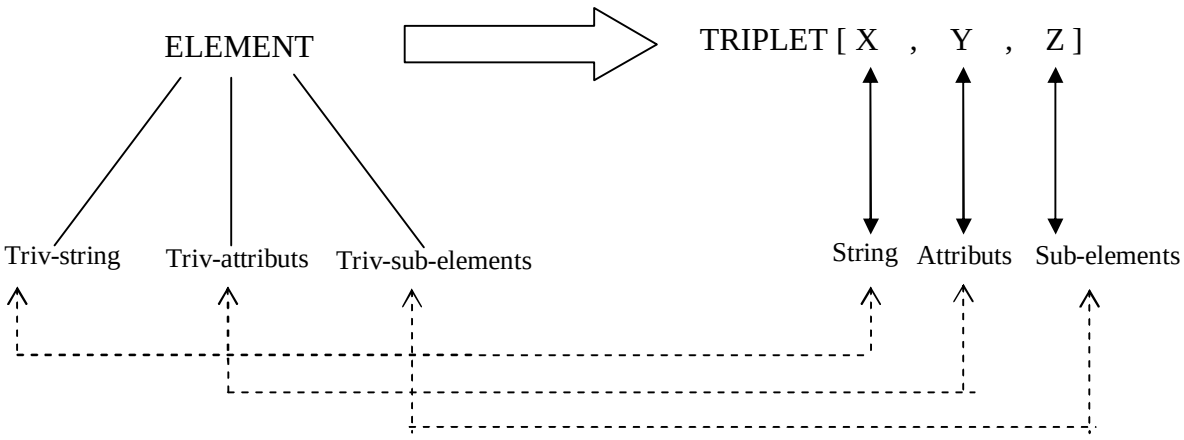


Figure 4.18: Modules et vues nécessaires à la génération de ELEMENT

Donc, le module Maude spécifiant les éléments XML est présenté à la figure 4.19:

```
Fmod ELEMENT is
pr TRIPLET ( Triv-string | Triv-attributs | Triv-sub-elements)
sort Element .
subsort Element < Sub-element .
endfm
```

Figure 4.19: Module `ELEMENT`

Dans Full Maude une correspondance syntaxixe et sémantique est faite entre les paramètres formels déclarés dans le module paramétré (telque `TRIPLET`) et les paramètres effectifs du module après instanciation.

B) Les fichiers schéma XML

L'intérêt de cette forme de spécification est que pour formaliser un fichier schéma XML, nous reprenant les mêmes modules utilisées au niveau du fichier XML, nous réutilisons le module paramétré `TRIPLET`,.et la vue `Triv-attributs`. (voir figure 4.20)

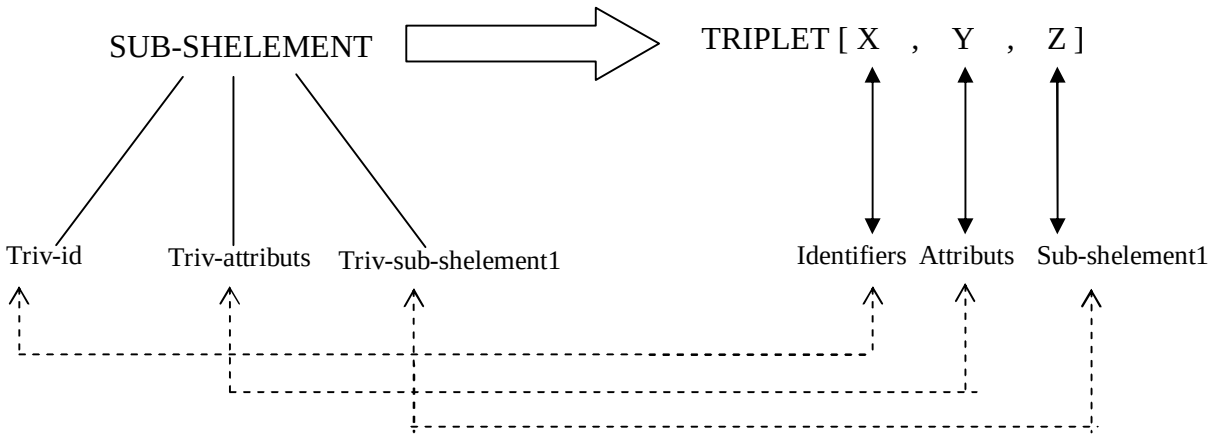


Figure 4.20: Modules et vues nécessaires à la génération de SUB-SHELEMENT

Le module résultat est présenté ainsi (voir figure 4.21)

```
fmod SUB-SHELEMENT is
protecting TRIPLET (Triv-id | Triv-attribut | Triv-sub-
shelement1).
endfm
```

Figure 4.21: Module SUB-SHELEMENT

Le module IDENTIFIERS (voir figure 4.22) remplace le module STRING dans un document XML, du moment que les noms autorisés dans un document schéma, sont de type prédéfini. La vue correspondante est obtenue aussi de façon très simple.

```
fmod IDENTIFIERS is

sort Id .

ops
 xsd:element : -> Id .
 xsd:complextype :->Id .
 xsd:sequence : -> Id .
 xsd:attribute : -> Id .
 _ is-Id-type : Striing -> Bool .          (29)

var X : String .

eq "xsd:string" is-Id-type = true .
eq "xsd:float" is-Id-type = true .
eq "xsd:binary" is-Id-type = true .
eq "xsd:decimal" is-Id-type = true .
eq "xsd:integer" is-Id-type = true .
eq "xsd:nonpositiveInteger" is-Id-type = true .
eq "xsd:nonnegativeInteger" is-Id-type = true .
eq "xsd:positiveInteger" is-Id-type = true .
eq "xsd:negativeInteger" is-Id-type = true .
eq "xsd:long" is-Id-type = true .
eq "xsd:int" is-Id-type = true .
eq "xsd:short" is-Id-type = true .
```

```
eq "xsd:byte" is-Id-type = true .
eq "xsd:date" is-Id-type = true .
eq "xsd:month" is-Id-type = true .
eq "xsd:year" is-Id-type = true .
eq "xsd:century" is-Id-type = true .
eq X is-Id-type = false .
endfm
```

Figure 4.22: Module IDENTIFIERS

Le module SUB-ELEMENT1 (voir figure 4.23) et la vue correspondante Triv-sub-element1 sont obtenus aussi de façon similaire.

```
fmod SUB-SHELEMENT1 is
Including STRING .
Sort Sub-shelement .
ops
sh-vide : à Sub-shelement .

endfm
```

Figure 4.23: Module SUB-SHELEMENT1

```
( view Triv-id from TRIV to IDENTIFIERS is
        sort Elt to Id .                          endv)

( view Triv-sub-shelement1 from TRIV to SUB-SHELEMENT1 is
        sort Elt to Sub-shelement1 .                    endv)
```

Nous complétons la spécification de la liste des contenus d'élément XML schéma, en donnant un nouveau module Maude appelé SUB-SHELEMENTS, formé comme suit (figure 4.24):

```
fmod SUB-SHELEMENTS is
extending SUB-SHELEMENT BOOL .
sort Sub-shelements .
```

```
subsorts Sub-shelement < Sub-shelements .
ops
_+_ : Sub-shelements Sub-shelements à Sub-shelements . [ACI: sh-
vide]
Access-shelem (_,_) : String Sub-shelements à Sub-shelements .
reste (_,_) : Sub-shelements Sub-shelement à Sub-shelements .
var x : String . var e : Sub-shelement .
var z s : Sub-shelements .
eq Acces-shelem(x, sh-vide) = sh-vide .
eq Acces-shelem(x, e + z) = e if 1er(e) == x
                          else Acces-shelem(x, z).
eq reste ( sh-vide , e) = sh-vide .
eq reste (s + z , e) = s + reste (z , e) .
eq reste (e + z, e) = z .

endfm
```

Figure 4.24: Module SUB-SHELEMENTS

Le module Maude spécifiant les éléments XML schéma (voir figure 4.26), est trouvé par l'instanciation du module paramétré TRIPLET, en définissant une vue supplémentaire concernant les sous éléments (contenus) (voir figure 4.25)

```
( view Triv-sub-shelements from TRIV to SUB-SHELEMENTS is
        sort Elt to Sub-shelements .                          endv)
```

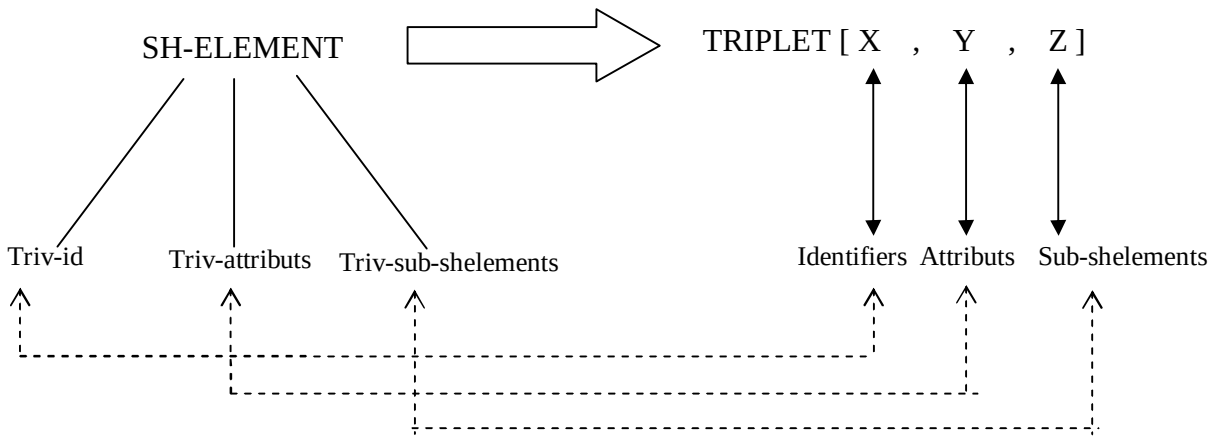


Figure 4.25: Modules et vues nécessaires à la génération de SH-ELEMENT

```
Fmod SH-ELEMENT is
pr TRIPLET ( Tiv-id | Triv-attributs | Triv-sub-shelements)
sort Sh-element .
subsort Sh-element < Sub-shelement .
endfm
```

Figure 4.26: Module SH-ELEMENTS

En exploitant la similarité étroite qui existe entre la structure d'un document XML et celle d'un schéma XML, le module Validate (voir figure 4.27) dans lequel nous validons un document XML E par rapport à son schéma XML E' est donné de façon très simplifiée ainsi nous remarquons pour ce module qu'on utilise les même opérations pour accéder aux composants de E ou E'.

La spécification ainsi est plus compacte et concise.

```
mod Validate is
Including Element Sh-element .

ops
validate(-,-) : Sh-element Element -> Bool .
Name-corr(-,-) : Attributs String -> Bool .
Att-corr(-,-) : Sh-element Attributs à Bool .
Sub-corr(-,-) : Sh-element Sub-elements -> Bool .
Type-corr(_,_) : Attributs Attributs à Bool .
Att-corr'(_,_) : Sub-shelements Attributs à Bool .
Sub-corr'(_,_) : Sub-shelements Sub-elements à Bool .

var e :Sh-element .
var e':Element .
var e1:Sub-shelements .
var e2:Sub-element .
var e3:Sub-elements .
var b : Attribut .
vars A B : Attributs .
var x : String .


crl1 : validate (e,e') à true
       if 1er (e) == xsd:element  and Name-corr(2eme(e),1er(e'))
       and Att-corr(e,2eme(e')) and Sub-corr(e, 3eme(e')) .
```

```
crl2 : Name-corr(A,x)à true if att-val(Acces-att ("name", A)) == x else false.

crl3: Att-corr'(e1 , ⌀)à true if
               Acces-shelem (xsd:attribute, e1) == sh-vide
               else false .

crl4 : Att-corr(e, b B) à Type-corr(2eme(e), 2eme(3eme(e)))
and Att-corr'(3eme(3eme(e)),b B).

crl5 : Att-corr'(e1, b B) à true if att-name (b)== att-val(Acces-att ("name",
2eme(Acces-shelem (xsd:attribute, e1))).
and att-corr(reste(e1,Acces-shelem(xsd:attribute,e1) ) ,B ).
else false .

crl6 : Type-corr (A,B)à true if Acces-att("type",A)==⌀
and Acces-att("name",B)== ⌀.

crl7 : Type-corr (A,B)à true if att-val(Acces-att ("type",A))==
att-val(Acces-att ("name",B)).

crl8 : Sub-corr( e , e2/e3) à Type-corr(2eme(e), 2eme(3eme(e)))
and Sub-corr'(3eme(3eme(e)),e2/e3)

crl9 : Sub-corr'( e1 , e3) à true if e1== sh-vide and
e3 is-const and
att-val(Acces-att ("type",2eme(e1)))is-Id-type
else false.

crl10 : Sub-corr' (e1,e2/e3)= true
if validate (Acces-shelem (xsd:element,3eme(Acces-shelem(xsd:sequence,e1))), e2)
and Sub-corr'(reste(3eme(Acces-shelem(xsd:sequence,e1)), Acces-
shelem(xsd:element,3eme(Acces-shelem (xsd:sequence,e1)))), e3)

crl11: Sub-corr'(e1, ⌀) = true if Acces-shelem(xsd:element, e1)== sh-vide    else
false

endm.
```

Figure 4.27: Module `Validate`

4 Conclusion

Dans ce chapitre nous avons proposé une formalisation de XML, dans le cadre de la logique de réécriture.

Nous avons réalisé aussi une application spécifique basée sur cette formalisation. Elle consiste à la validation d'un document XML par rapport à son document schéma XML. Cette application est vue et réalisée de deux manières, la première sert à donner un module système Maude dans lequel le processus de validation proposé est implémenté.

La deuxième suggère une vue paramétrée du problème, elle exploite alorsla programmation paramétrée fournie par Full Maude.

Chapitre V

Etude de Cas et Implémentation

1 Introduction

Le présent chapitre présente dans une première partie, la validation et l'analyse syntaxique des différents modules réalisés spécifiant le langage XML et son modèle schéma XML. Cette analyse est effectuée dans l'environnement Maude, la version Workstation 1.0. Dans une deuxième partie, une étude de cas réaliste et de grandeur naturelle est considérée pour illustrer l'approche de formalisation du langage XML que nous avons proposée. Il s'agit d'un document XML qui décrit les différentes équipes du laboratoire LIRE de notre département d'informatique, les chercheurs ainsi que les différents travaux et articles réalisés. Nous donnons également le schéma XML associé, et nous procédons à la validation de document XML par rapport au document schéma XML.

2 Implémentation de la spécification en Maude

Nous allons voir dans cette section l'implémentation des différentes théories fonctionnelles, proposées précédemment, à travers l'environnement Maude Workstation 1.0, nous procédons à analyser de ces modules ainsi que leur prototypage à travers la formulation de quelques requêtes et la réponse automatique à ces dernières.

2.1 L'environnement Maude Workstation 1.0

Nous avons choisi l'environnement Maude Workstation 1.0 parmi plusieurs versions de Maude proposées. Ces dernières peuvent générer des problèmes de compatibilité, mais l'environnement Maude Workstation 1.0 implémenté en JAVA a surmonté ces problèmes, vu la standardisation de la plate forme JAVA.

L'environnement Maude Workstation fournit un ensemble d'outils et de fonctionnalités permettant de saisir, corriger et mettre au point le code d'un ou de plusieurs modules Maude:

- Un éditeur intégrant un préprocesseur de syntaxe Maude permet de détecter toutes les erreurs syntaxiques dans les modules.
- Un émulateur pour le lancement du Maude (Core Maude et Full Maude).
- Des outils de débogage et de Trace pour la mise au point des modules Maude.
- Des zones pour l'affichage des résultats, des erreurs et pour l'introduction des commandes Maude.

2.2 L'implémentation des théories proposées

La saisie et le chargement du code associé aux différentes théories fonctionnelles, qui ont permis la spécification des composants syntaxiques d'un fichier XML et sa grammaire schéma XML, ont été réalisés de façon simple et naturelle à travers les modules Maude correspondants (voir figures : 5.1, 5.2, 5.3)

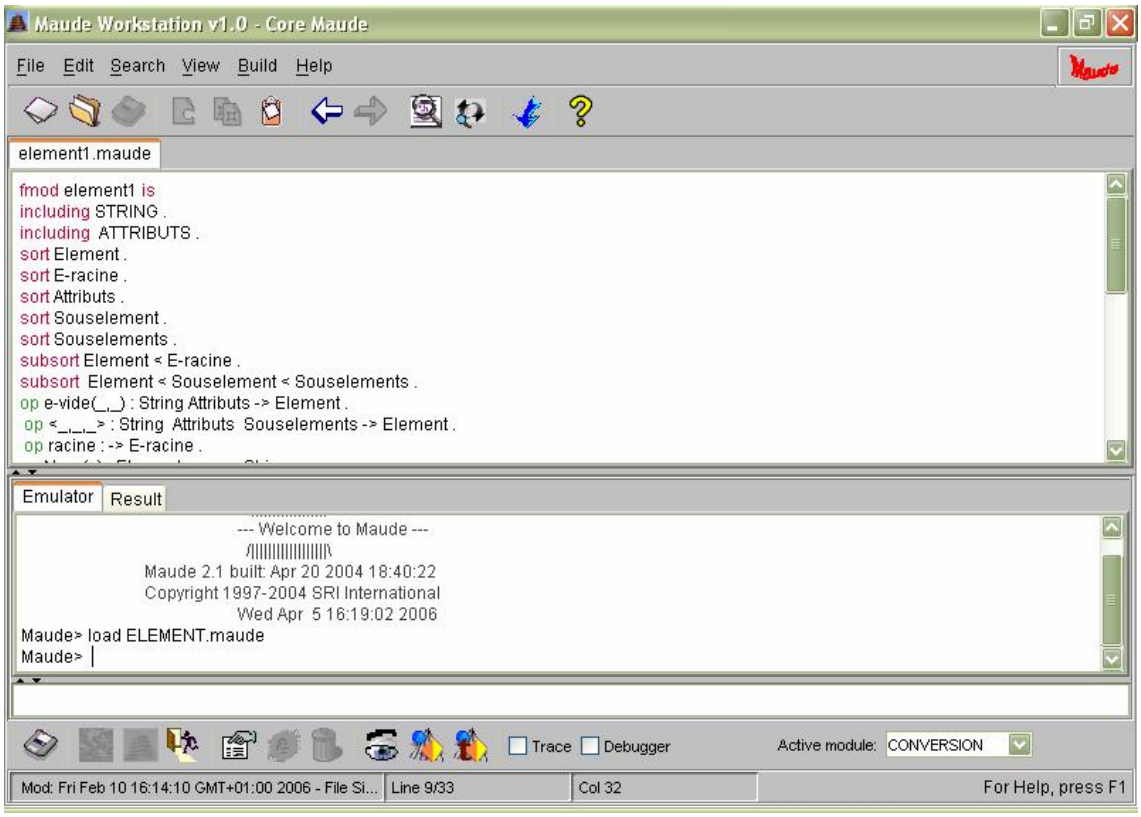


Figure 5.1 : Module `ELEMENT`.

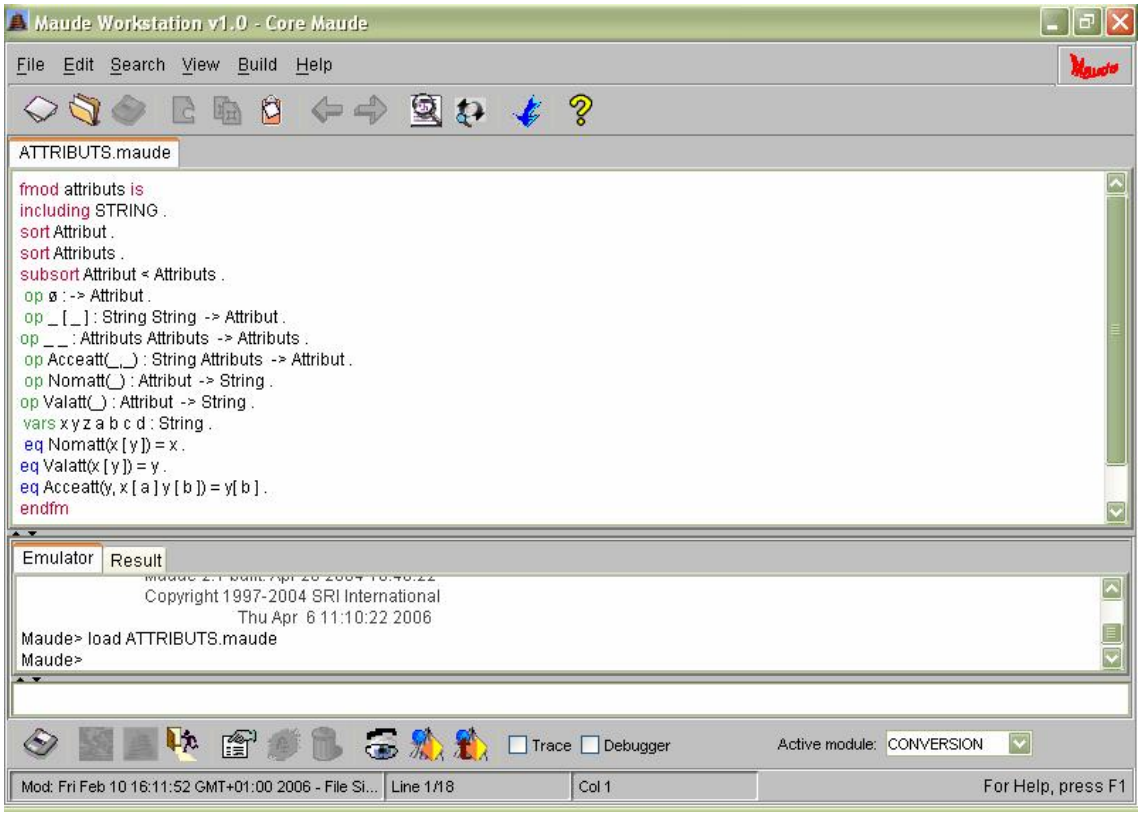


Figure 5.2 : Module `ATTRIBUTS.`

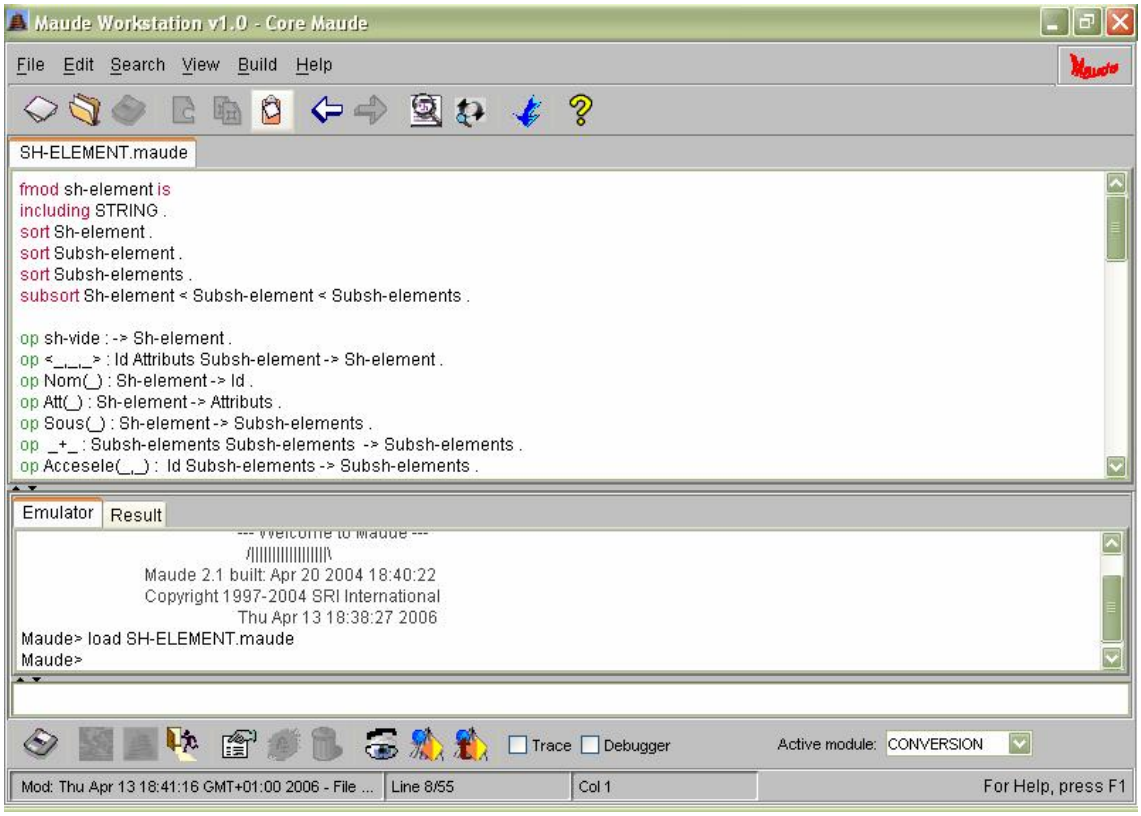


Figure 5.3 : Module `SH-ELEMENT.`

L'analyse syntaxique de ces modules est faite implicitement lors de leur chargement en Maude (commande Load, voir figures 5.1, 5.2, 5.3). Dans le cas où une erreur syntaxique est produite, le chargement ne sera pas fait correctement, et les erreurs seront affichées.

La figure 5.4 illustre une erreur syntaxique produite par l'écriture erronée du mot réservé "String" (Strin). Cette erreur a généré une suite d'erreurs durant le chargement du module ATTRIBUTS.

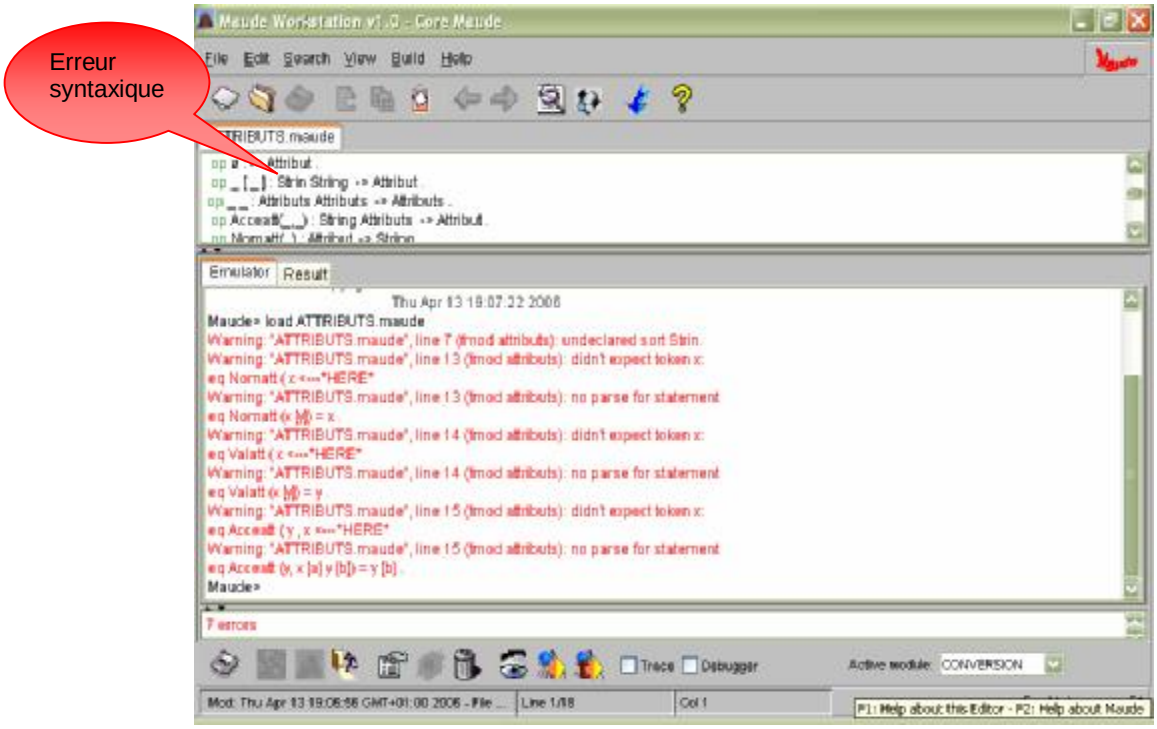


Figure 5.4 : Un exemple d'erreur syntaxique.

Une analyse plus poussée de ces modules peut être aussi faite en formulant des requêtes en Maude sur des exemples spécifiques de fichiers XML ou schéma XML.

Nous reprenons l'exemple 4.1 du chapitre précédent décrit par le terme:

<parent, ø,<children, ø,cst(Ali)>>, nous pouvons lancer plusieurs commandes pour prototyper le module ELEMENT et ATTRIBUTS. Les résultats sont présentés respectivement au niveau de la figure 5.5 :

- reduce in element Nom(<parent, ø,<children, ø,cst(Ali)>>), pour obtenir le nom du l'élément XML <parent, ø,<children, ø,cst(Ali)>>.
- reduce in attributs :Valatt (age[12]), pour obtenir la valeur de l'attribut XML age[12].
- reduce in element Sub (<parent, ø,<children, ø,cst(Ali)>>), pour obtenir l'élément fils de l'élément XML <parent, ø,<children, ø,cst(Ali)>>

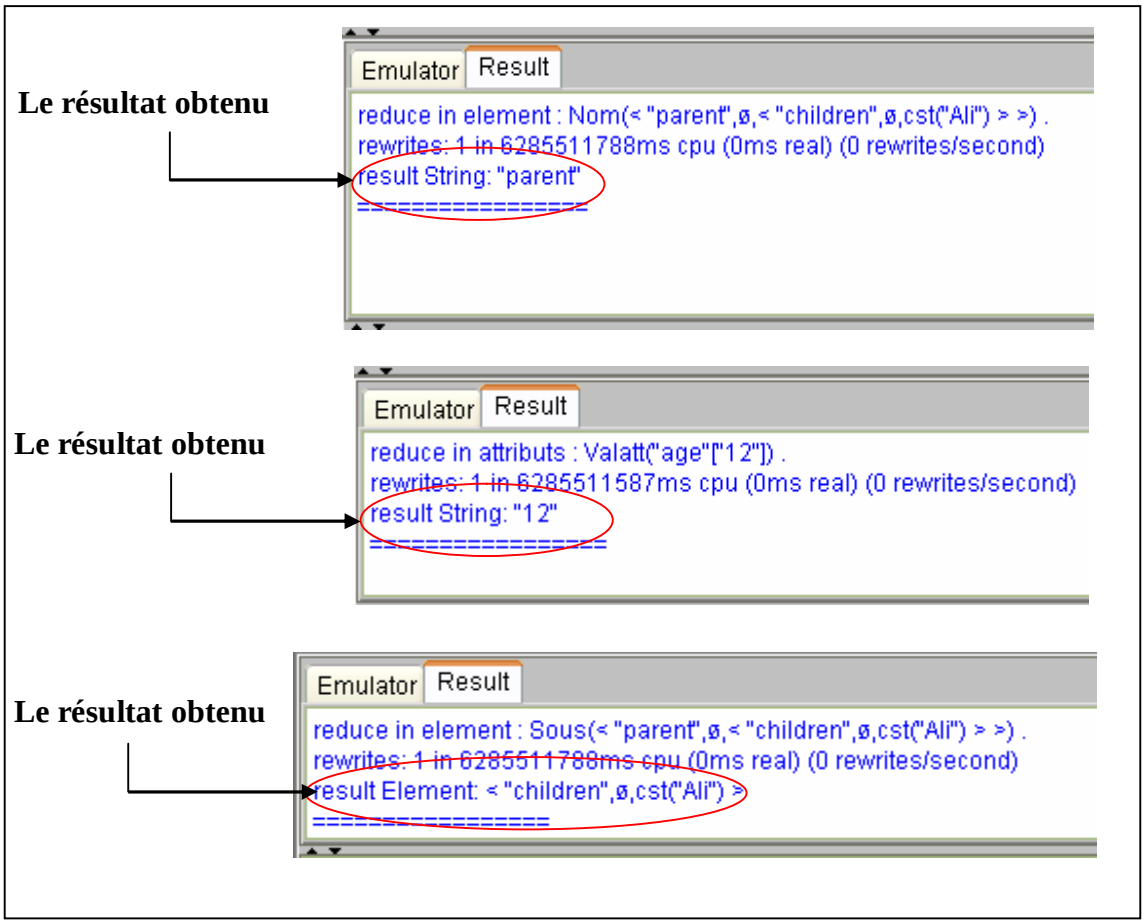


Figure 5.5 : Prototypage des modules pour un document XML.

Le prototypage du module SH-ELEMENT, est réalisé en utilisant un ensemble de requêtes à travers le schéma XML du même exemple décrit par le terme suivant (figure5.6):

< xsd:element,"name"["parent"],< xsd:complextype, ø, <xsd:sequence,ø,< xsd:element,"name"["children"] "type"["xsd:string"],sh-vide > > > >

P reduce in sh-element: Name-sh (< xsd:element,"name"["parent"],< xsd:complextype , ø, <xsd:sequence,ø,< xsd:element,"name"["children"] "type"["xsd:string"],sh-vide > > > >)

P reduce in sh-element : Att-sh (< xsd:element,"name"["parent"],< xsd:complextype , ø, <xsd:sequence,ø,< xsd:element,"name"["children"] "type"["xsd:string"],sh-vide > > > >)

P reduce in sh-element : Sub-sh(< xsd:element,"name"["parent"],< xsd:complextype , ø, <xsd:sequence,ø,< xsd:element,"name"["children"] "type"["xsd:string"],sh-vide > > > >) .

Figure 5.6 : Prototypage des modules pour un fichier schéma XML.

2.3 L'implémentation de l'application VALIDATE

Le module `VALIDATE` décrit formellement une application possible de la modélisation réalisée. Il est implémenté en Maude Workstation sous forme d'un module système tels qu'il se présente dans la figure 5.7. La présence des règles de réécritures nous permet de considérer la commande "rewrite" pour formuler des requêtes sur ce module. Nous avons considéré un ensemble de requêtes pour valider des éléments XML par rapport à leurs définitions dans le schéma XML. Nous validons:

- Premièrement le nom d'un élément XML par rapport à sa définition dans le schéma.
- Deuxièmement les attributs par rapport à sa définition dans le schéma.
- Dernièrement son contenu par rapport à sa définition dans le schéma.

Comme nous présentons la validation complète d'un élément XML (la figure 5.8).

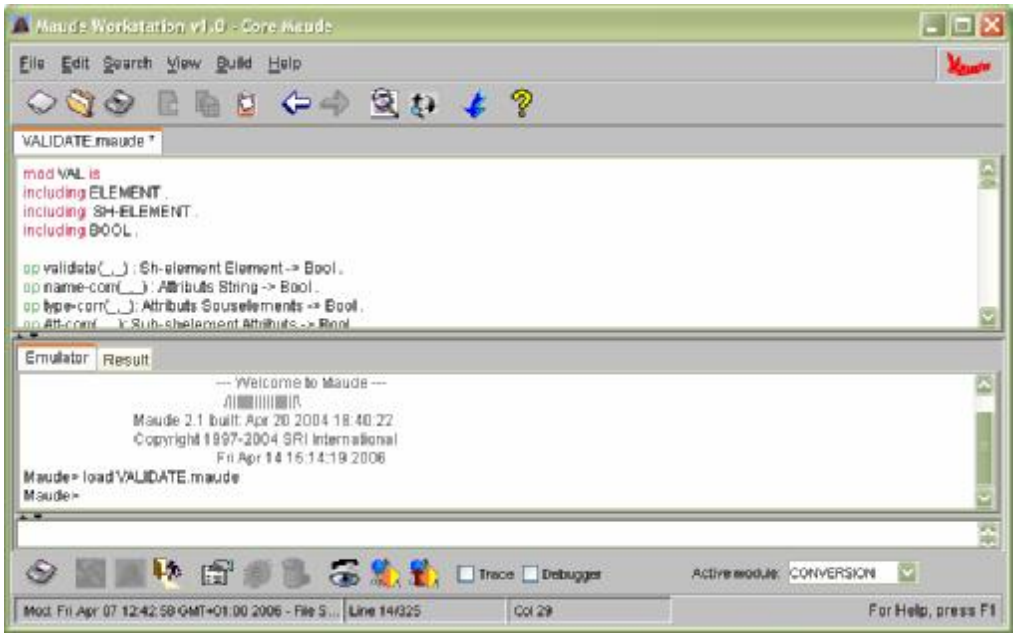


Figure 5.7 : Module `VALIDATE`.

Figure 5.8 : Prototypage de module `VALIDATE`.

Si par exemple, le nom de l'élément fils d'un élément XML ne correspond pas à sa définition dans le schéma, le résultat sera "false" (c'est-à-dire l'élément n'est pas valide par rapport à son schéma) (figure 5.9):

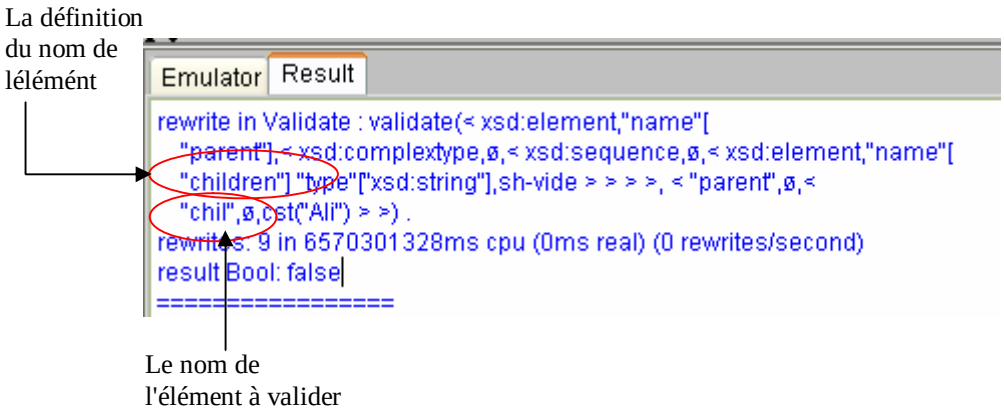


Figure 5.9 : Exemple d'un élément non valide **.**

3 Etude de cas

Nous avons pris l'initiative de décrire le laboratoire LIRE du département d'informatique de l'université Mentouri de Constantine, les différentes équipes ainsi que leurs chercheurs avec leurs travaux, et leurs activités, sous forme de fichier XML et sa grammaire génératrice schéma XML. En suite, nous procédons à l'analyse de cette application.

3.1 Description du laboratoire LIRE

Le laboratoire LIRE comporte quatre équipes de recherche :

1. Base de données et systèmes d'information avancés: BDSIA.
2. Reconnaissance des formes et intelligence artificielle: RFIA.
3. Systèmes distribués et génie logiciel: SDGL.
4. Intelligence artificielle et génie logiciel: IAGL.

Chaque équipe contient un ensemble de chercheurs, qui y sont membres et un ensemble d'activités (thèmes de travail), ainsi un ensemble d'articles réalisés par les membres de l'équipe. Elle possède donc :

1. Un identificateur (BDSIA, RFIA, SDGL, IASD).
2. Un intitulé.
3. Les membres de l'équipe, dont chacun est connu par :

- Un nom.
- Un prénom.
- Un E-Mail.
- Un titre (le grade).
- Un numéro de téléphone.
- Une responsabilité (s'il est un chef d'équipe la valeur est 1, sinon 0).
- Un numéro de Fax.

4. Un ensemble d'activités, dont chacune est connue par :

- Un identificateur.
- Un labelle (le domaine d'intérêt).
- Une description (une description de l'activité correspondante).

5. Un ensemble d'articles, dont chacun est décrit par :

- Un identificateur.
- Un titre.
- L'année de publication.
- Un code de publication.
- Le type de publication.
- La revue (dans laquelle l'article est publié).
- La conférence.
- Le nombre des pages.
- Le volume.
- Un éditeur.
- Les auteurs.
- Un résumé.

3.2 Implémentation de l'étude de cas

Dans l'implémentation de cette étude de cas, nous nous sommes contentés seulement d'une partie du fichier XML décrivant la structure du laboratoire LIRE, présentée comme suit:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <EQUIPES>
    <EQUIPE id="SDGL">
      <INTITULE>Systemes distribues et GL</INTITULE>
      <MEMBRES>
       <CHERCHEUR id="40">
        <NOM>Belala</NOM>
        <PRENOM>Faiza</PRENOM>
        <EMAIL>belalafaiza@hotmail.com</EMAIL>
        <TITRE>Maitre de Conferences</TITRE>
        <TELEPHONE>00213 31 63 90 10</TELEPHONE>
        <RESPONSABLE />
        <FAX>00213 31 63 90 10</FAX>
      </CHERCHEUR>
     </MEMBRES>
     <ACTIVITES>
      <ACTIVITE id="4">
       <LaBELLE />
       <DESCRIPTION />
      </ACTIVITE>
     </ACTIVITES>
     <ARTICLES>
      <ARTICLE id="131">
       <TITRE>A Semantic Support for XML</TITRE>
       <ANNEE>2006</ANNEE>
       <CODEPUBLI>7</CODEPUBLI>
       <TYPEPUBLI />
       <REVUE />
       <CONFERENCE />
       <PAGES />
       <VOLUME>0</VOLUME>
       <EDITEUR />
       <AUTEURS>H.Douibi, F.Belala</AUTEURS>
       <RESUME>The problem that XML has a poor semantics
constitutes a serious barrier for the efficient use of XML
document.Tosurmount this barrier we propose the provision of a
semantic support to XML itself by using Rewriting logic. Based on
this formal support, properties may be analysed, as well as
transformations and verifications can be performed in the formal
world. The objective of the suggested work will be then double.
We will contribute, on the one hand to the integrated and
effective formalization of a universal data exchange language,
and on the other hand to a new case study proposal of the
rewriting logic and its formal language (Maude)</RESUME>
      </ARTICLE>
    </ARTICLES>
   </EQUIPE>
  </EQUIPES>
```

Le terme Maude correspondant à cette partie du fichier, selon notre approche de formalisation est:

```
<equipes,⌀,<equipe,id[SDGL],<intitule, ⌀ ,cst(système distribues et GL)>/<membres, ⌀ ,<chercheur,id[40],<nom, ⌀ ,cst(Belala)>/ <nom, ⌀ ,cst(Belala)>/ <prenom, ⌀ ,cst(Faiza)>/ <Email, ⌀ ,cst(belalafaiza@hotmail.com)>/ <titre, ⌀ ,cst(maitre de conference)>/ <telephone, ⌀ ,cst(0023131639010)>/ <responsable, ⌀ ,e-vide>>/<activites, ⌀ ,<activite,id[4],<labelle, ⌀ ,e-vide>/<description, ⌀ ,e-vide>>/<articles, ⌀ ,<article,id[131], <titre, ⌀ ,cst(a semantic support for XML)>/ <année, ⌀ ,cst(2006)>/ <codepubli, ⌀ ,cst(7)>/ <typepubli, ⌀ ,e-vide>/ <revue, ⌀ ,e-vide>/ <conference, ⌀ ,e-vide>/ <pages, ⌀ ,e-vide>/ <volume, ⌀ ,cst(0)>/ <editeur, ⌀ ,e-vide>/ <auteurs, ⌀ ,cst(H.Douibi,F.Belala)>/ <resume, ⌀ ,cst(The problem that XML has a poor semantics constitutes a serious barrier for the efficient use of XML document. To surmount this barrier we propose the provision of a semantic support to XML itself by using Rewriting logic. Based on this formal support, properties may be analysed, as well as transformations and verifications can be performed in the formal world. The objective of the suggested work will be then double. We will contribute, on the one hand to the integrated and effective formalization of a universal data exchange language, and on the other hand to a new case study proposal of the rewriting logic and its formal language Maude)>>>>>>>
```

Le fichier schéma XML associé au fichier XML décrivant la structure du laboratoire LIRE est le suivant:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="EQUIPES">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="EQUIPE">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="INTITULE" type="xsd:string" />
       <xsd:element name="MEMBRES">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name="CHERCHEUR">
            <xsd:complexType>
             <xsd:sequence>
              <xsd:element name="NOM" type="xsd:string" />
              <xsd:element name="PRENOM" type="xsd:string" />
              <xsd:element name="EMAIL" type="xsd:string" />
              <xsd:element name="TITRE" type="xsd:string" />
              <xsd:element name="TELEPHONE" type="xsd:string"/>
              <xsd:element name="RESPONSABLE"type="xsd:string"/>
              <xsd:element name="FAX" type="xsd:string" />
             </xsd:sequence>
             <xsd:attribute name="id" type="xsd:string" />
            </xsd:complexType>
           </xsd:element>
          </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
        <xsd:element name="ACTIVITES">
         <xsd:complexType>
          <xsd:sequence>
           <xsd:element name="ACTIVITE">
            <xsd:complexType>
             <xsd:sequence>
              <xsd:element name="LABELLE" type="xsd:string" />
              <xsd:element name="DESCRIPTION"type="xsd:string"/>
             </xsd:sequence>
             <xsd:attribute name="id" type="xsd:string" />
            </xsd:complexType>
           </xsd:element>
          </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
```

```
        <xsd:element name="ARTICLES">
         <xsd:complexType>
          <xsd:sequence>
           <xsd:element name="ARTICLE">
            <xsd:complexType>
             <xsd:sequence>
              <xsd:element name="TITRE" type="xsd:string" />
              <xsd:element name="ANNEE" type="xsd:string" />
              <xsd:element name="CODE-PUB" type="xsd:string"/>
              <xsd:element name="TYPE-PUB" type="xsd:string"/>
              <xsd:element name="REVUE" type="xsd:string" />
              <xsd:element name="CONFERENCE"type="xsd:string"/>
              <xsd:element name="PAGES" type="xsd:string" />
              <xsd:element name="VOLUME" type="xsd:string" />
              <xsd:element name="EDITEUR" type="xsd:string" />
              <xsd:element name="AUTEUR" type="xsd:string" />
              <xsd:element name="RESUME" type="xsd:string" />
             </xsd:sequence>
             <xsd:attribute name="id" type="xsd:string" />
            </xsd:complexType>
           </xsd:element>
          </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:string" />
      </xsd:complexType>
     </xsd:element>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Nous avons utilisé un éditeur pour les fichiers schéma XML, afin de les visualiser de façon schématique (figure 5.10).

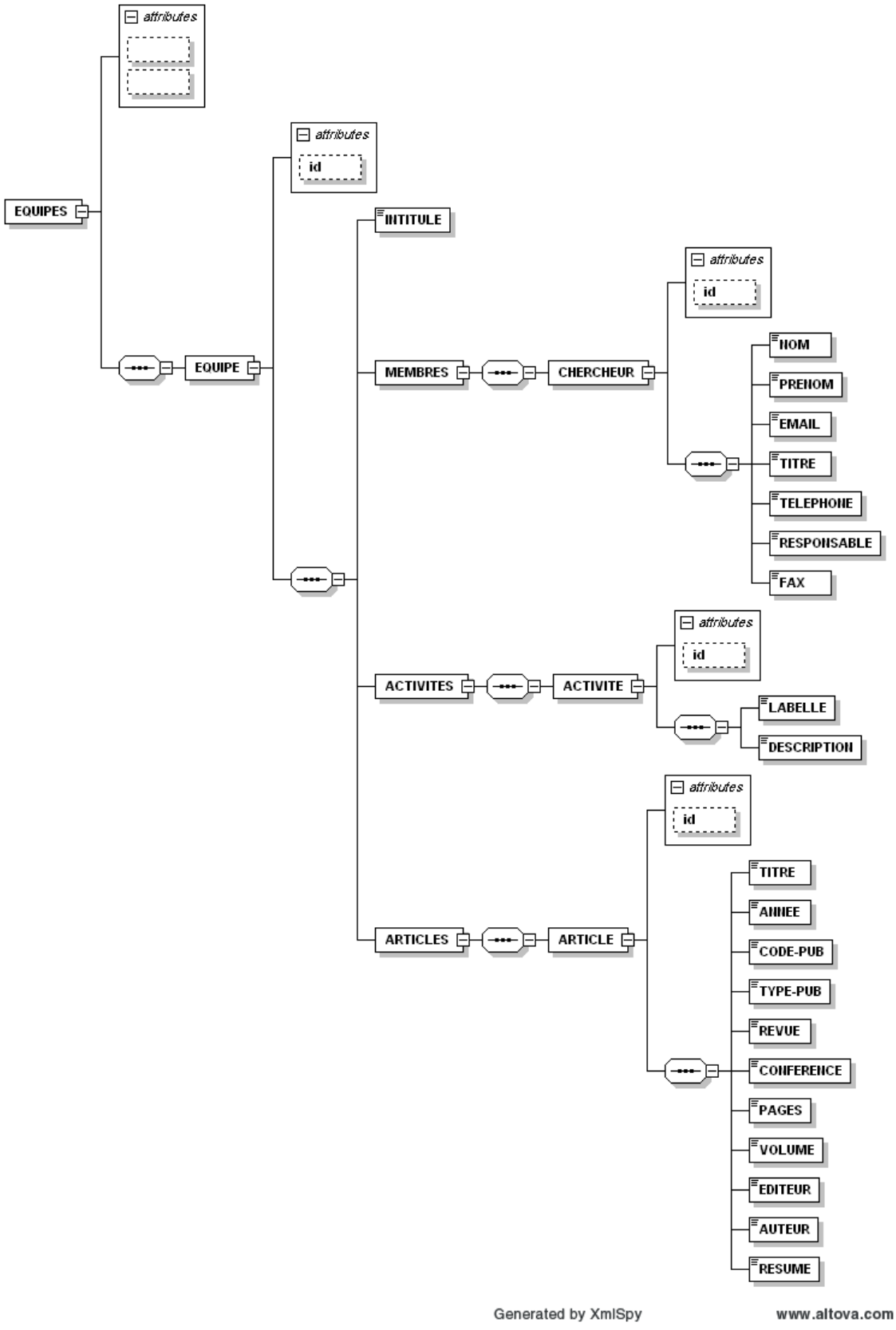


Figure 5.10 : Représentation du schéma XML en XML Spy.

Le terme Maude équivalent à ce fichier schéma XML est:

> <xsd:element ,name[EQUIPES], <xsd:complexType, ø, <xsd:sequence, ø, <xsd:element, name[EQUIPE], <xsd:complexType, ø , <xsd:sequence, ø , <xsd:element, name[INTITULE] type[xsd:string],sh-vide >/ <xsd:element, name[MEMBRES], <xsd:complexType, ø, <xsd:sequence, ø, <xsd:element name[CHERCHEUR], <xsd:complexType, ø, <xsd:sequence, ø, <xsd:element, name[NOM] type[xsd:string],sh-vide >/ <xsd:element, name[PRENOM] type[xsd:string],sh-vide >/ <xsd:element, name[EMAIL] type[xsd:string],sh-vide >/ <xsd:element, name[TITRE] type[xsd:string],sh-vide >/ <xsd:element, name[TELEPHONE] type[xsd:string],sh-vide > /<xsd:element, name[RESPONSABLE] type[xsd:string],sh-vide >/ <xsd:element, name[FAX] type[xsd:string],sh-vide >>/ <xsd:attribute, name[id] type[xsd:string], sh-vide> >> > > > /<xsd:element, name[ACTIVITES], <xsd:complexType, ø , <xsd:sequence, ø , <xsd:element, name[ACTIVITE], <xsd:complexType, ø , <xsd:sequence, ø , <xsd:element, name[LABELLE] type[xsd:string],sh-vide >/ <xsd:element, name[DESCRIPTION] type[xsd:string], sh-vide>>/ <xsd:attribute, name[id] type[xsd:string], sh-vide>>>>>> / <xsd:element, name[ARTICLES], <xsd:complexType, ø , <xsd:sequence, ø , <xsd:element, name[ARTICLE], <xsd:complexType, ø , <xsd:sequence, ø , <xsd:element, name[TITRE] type[xsd:string],sh-vide >/ <xsd:element, name[ANNEE] type[xsd:string],sh-vide >/ <xsd:element, name[CODE-PUB] type[xsd:string],sh-vide >/ <xsd:element, name[TYPE-PUB] type[xsd:string],sh-vide >/ <xsd:element, name[REVUE] type[xsd:string],sh-vide >/ <xsd:element, name[CONFERENCE] type[xsd:string],sh-vide >/ <xsd:element, name[PAAGES] type[xsd:string],sh-vide >/ <xsd:element, name[VOLUME] type[xsd:string],sh-vide >/ <xsd:element, name[EDITEUR] type[xsd:string],sh-vide >/ <xsd:element, name[AUTEUR] type[xsd:string],sh-vide >/ <xsd:element, name[RESUME] type[xsd:string],sh-vide >> / <xsd:attribute, name[id] type[xsd:string], sh-vide>>>>>>> / <xsd:attribute, name[id] type[xsd:string], sh-vide> > > > > >

Etant donnée toujours la même implémentation des différents modules spécifiant le langage XML et sa grammaire en général, nous pouvons, pour ces deux termes, executer un ensemble de commandes pour répondre à des requêtes telles que la validation du nom de fichier XML (figure 5.11).

Nous pouvons aussi montrer la validation de ce document XML ou d'autre par rapport au fichier schéma XML donné (figure 5.12).

Figure 5.11 : Prototypage du module validate pour les termes représentants le fichier XML décrivant le laboratoire LIRE

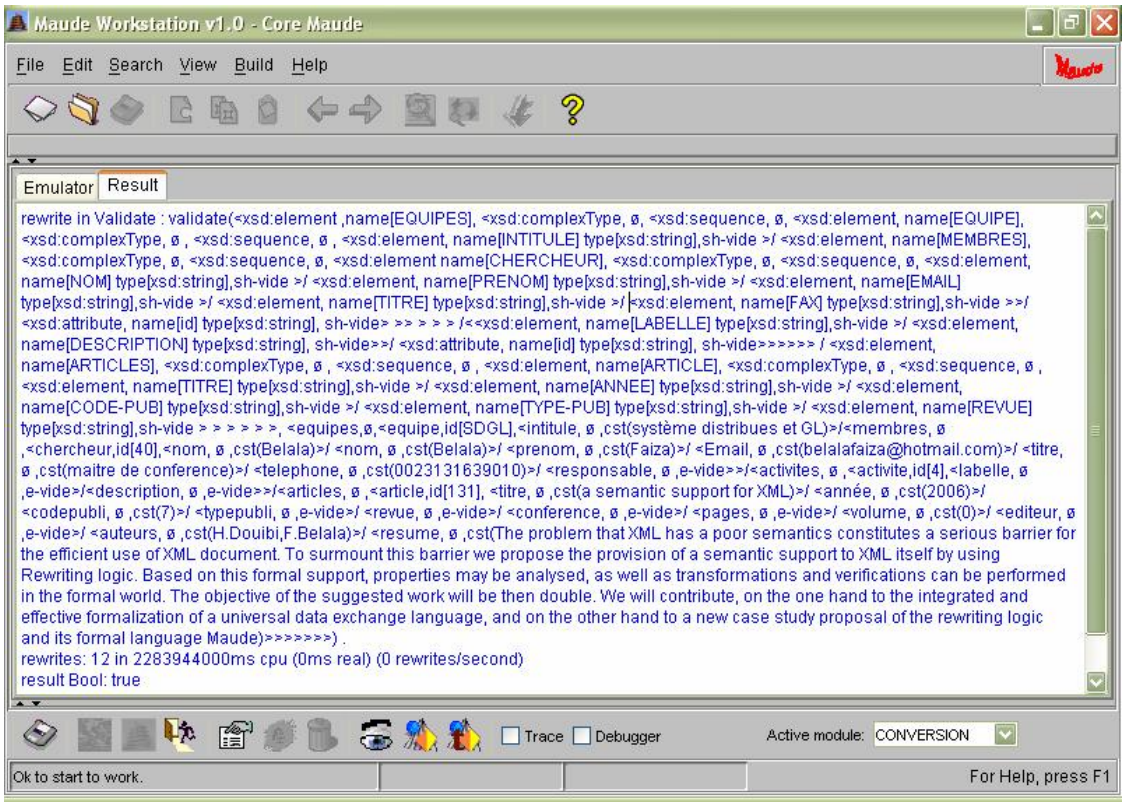


Figure 5.12 : Validation du fichier XML décrivant le labo LIRE par rapport à son schéma

4 Conclusion

Dans ce chapitre, nous avons présenter l'mplémentation des différents modules proposés dans le chapitre précédent. Nous avons montré le bon fonctionnement de ces modules à travers un ensemble de requêtes.

Nous avons proposé une étude de cas réaliste et de grandeur naturelle décrivant le laboratoire LIRE de notre département d'informatique, pour illustrer mieux la faisabilité de l'approche de formalisation que nous avons élaborée.

Conclusion Générale

Les méthodes formelles permettent de raisonner rigoureusement sur les programmes informatiques, afin de démontrer leur correction, en se basant sur des raisonnements de logique mathématique. La diversification des corpus mathématiques qui sont à la base de ces méthodes, a engendré une diversification aussi de celles ci. *La logique de réécriture*, introduite par José Meseguer, est considérée comme une méthode formelle. En effet plusieurs formalismes destinés à la modélisation des systèmes concurrents (réseaux de Petri, systèmes de transitions étiquetés, etc) ont été intégrés dans cette logique unificatrice. De plus, elle a aussi une portée assez large sur les systèmes orientés objets, les langages de programmation, les générateurs de programmes, etc.

D'autre part, les concepts théoriques de cette logique sont implémentés autour d'un langage très performant appelé Maude, il supporte efficacement la déduction logique. Ce qui le rend trés extensible, il permet la programmation des applications très avancées. Maude est utilisé pour créer des environnements exécutables pour différentes logiques, langages, et modèles de concurrence. Nous citons par exemple l'implémentation de CCS, les réseaux de Petri, les systèmes de transitions étiquetés, etc.

L'apparition d'Internet a engendrée des nouveaux concepts, langages, et outils pour répondre aux besoins des utilisateurs. Cependant, le changement croissant de ces besoins montre aussitôt les limites de ces concepts. Une utilisation rationnelle et efficace de ces derniers exige leur formalisation, afin de pouvoir raisonner d'une façon rigoureuse sur tous leur aspects, et leur fournir un fondement mathématique puissant. Tous les associès impliqués dans l'échange d'information sur Internet doivent avoir une compréhension commune et un pouvoir de raisonnement correct sur ces informations.

Dans ce contexte, notre projet vise la formalisation d'un langage universel, largement utilisé par la communauté se trouvant autour du Web: XML (eXtensible Markup Langage) successeur de HTML. Nous avons suggéré une nouvelle application de la logique de

réécriture via son langage Maude, pour attribuer une sémantique formelle à ce ainsi que d'autres langages manipulant des données structurées.

XML est considéré comme un format standard des documents pour l'écriture et l'échange de l'information sur le Web. Il est utilisé dans toutes les applications liées au Web, qui importent, exportent, manipulent, stockent ou transmettent des données. XML donne la possibilité à l'utilisateur de créer ses propres balises et de les structurer selon une syntaxe bien définie. Cependant, il souffre de limites liées aux contraintes impliquées par le type d'applications nouvelles du Web. La limite majeure de XML est sa sémantique de pauvre, elle est implicitement exprimée dans les documents. D'autre part, XML n'est pas approprié pour représenter les services dans la génération suivante du Web (Web sémantique).

L'objectif principal de ce travail était de donner une spécification formelle des documents XML ainsi que leur grammaire. Cette modélisation a défini d'une part la structure générique d'une classe de documents XML, et d'autre part, la phase de validation d'un document XML par rapport à son schéma XML. L'utilisation de la logique de réécriture via son langage Maude s'est avérée un formalisme adéquat pour spécifier ce type de langage. En effet, le langage Maude étant réflexif, les programmes Maude peuvent être des données pour d'autres programmes Maude. Par conséquent, les documents XML traduit en Maude peuvent constituer des données pour des applications de Maude. Ainsi, les documents XML et les programmes les contrôlant sont exprimés dans un même formalisme.
Les méthodes formelles et leur application dans le processus de développement des systèmes informatiques constituent un domaine de recherche très vaste et de plein essor. Nous avons abordé ce problème en déchiffrant les aspects syntaxiques et sémantiques du formalisme unificateur logique de réécriture. Dans ce contexte, nous avons dévoilé un langage déclaratif et tous les aspects qui y sont autour. Il s'agit du langage à "tout usage" Maude, basé sur cette logique.
Nous nous sommes intéressé beaucoup plus à son caractère de modularité, paramétrisation et réutilisation.

D'autre part, nous nous somme rapprochées plus des documents structurés et hiérarchiques de XML, ainsi que les fonctionnalités, les intérêts et les limites de ce langage.
A la base des tentatives de formalisation restreintes de XML, nous avons proposé un nouveau support sémantique, basé logique de réécriture, pour le langage bien qu'une de ses grammaires génératrices (schéma XML).

En effet, nous avons réussi la réalisation d'une spécification modulaire et simple des documents XML et des documents schéma XML. Nous avons obtenu une spécification exécutable, par l'implémentation de ces modules dans l'environnement Maude Workstation version 1.0.

Une conséquence importante s'est alors découlée, c'est une application spécifique des documents XML spécifiés, il s'agit de la validation des documents XML par rapport à leurs modèles schéma XML. Deux approches simple et paramétrée pour programmer cette validation ont été codifiées en Maude.

Une étude de cas réaliste et de grandeur naturelle a été aussi envisagée, la traduction d'un fichier XML décrivant la structure du laboratoire LIRE de notre département d'informatique en Maude.

Notre contribution possède un double intérêt. Elle constitue, d'une part une nouvelle application pour la logique de réécriture, et une tentative de formalisation d'un langage d'échange de données XML (la structure et le fonctionnement décrit par un document XML peuvent être pris en considération par un seul formalisme).

Ce travail constitue une base théorique ouvrant de nouvelles perspectives de recherches dans la formalisation et le raisonnement formel des langages de description des composants dynamiques du Web actuel.
Certaines continuations de ce travail sont entre autres:

- L'extension de la spécification réalisée en intégrant d'une façon naturelle d'autres composants et d'autres fonctionnalités, afin de couvrir une grande partie du langage XML ainsi que son modèle de validation schéma XML.
- La définition formelle des langages de requêtes pour XML ayant les caractéristiques d’expressivité importante.
- Enrichir naturellement le modèle sémantique proposé pour permettre à XML la description du fonctionnement des composants actifs considérés par XML.
- L'expression et la preuve de propriétés des programmes XML sont naturellement déduites à partir de cette modélisation du moment que nous leur avons associé un modèle mathématique rigoureux. Les transformations de ces programmes peuvent facilement être aussi considérées et trouveront leur signification en termes d'opérations mathématiques définies dans les algèbres ou les catégories mathématiques.

- De point de vue implémentation, notre travail peut être étendu pour générer automatiquement les documents XML ou schéma XML sous forme de termes algébriques Maude.

BIBLIOGRAPHIE

[Abr 05 a] : M. A.Abrao, B. Bouchou, M. H. Ferrari,D. Laurent, M. A. Musicante, "Vérification incrémentale des contraintes d'intégrité en XML", Université Francois-Rabelais de Tours - Laboratoire d'Informatique, 2005.

[Abr 05 b] : M.A. Abrao, B. Bouchou, A. Cheriat, M. Halfeld Ferrari, D. Laurent, M.A. Musicante, "Efficient incremental schema verification and integrity constraint checking for xml", Technical Report to appear, Université Francois-Rabelais de Tours - Laboratoire d'Informatique, 2005.

[All 00] : S. Allorge, "XML", Département ASI, mai. 2000.
ftp://ftp.commentcamarche.net/docs/xml

[Ama 02]: B. Amann, I. Fundulaki, M. Scholl, C. Beeri, A. Vercoustre,"Ontology-Based Integration of XML Web Resources", International Semantic Web conference; 2002.

[Ber 00]: A. Bergholz, "Extending Your Markup: An XML Tutorial", Stanford University, 2000. http://computer.org/internet/

[Ber 01]: T. Berners-Lee, J. Handler, O. Lassila, "The Semantic Web", Scientific American 184, 2001, pp. 34 – 43.

[BM 93]: M. Bettaz, M. Maouche. "How to Specify Non Determinism and True Concurrency with Algebraic Term Nets", Lect. Notes in Comp. Scie., Springer-Verlag, 1993, pp.164-180.

[Bos 94]: A. Bossi, G. Levi, M. C. Meo, "A compositional Semantics for logic programs", TCS 122 (1-2), 2-47, 1994.

[Bou 98]: C. Bouanaka, "De l'Unification des Modèles de la concurrence à Travers la logique de réécriture :Application aux structures d'Evènements", Thèse de Magister en Informatique, Institut d'Informatique Université Mentouri, Constantine Algérie, 1998.

[Bra 00] : T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, "Extensible Markup Language (XML) 1.0", W3C recommendation, 2000.
http://www.riik.ee/xml/trans/REC-xml-19980210-ee.html

[Bro 01] : A. Brown, M. Fuchs, J. Robie, P. Wadler," MSL A model for W3C XML Schema", Hong Kong, 2001.
http://homepages.inf.ed.ac.uk/wadler/papers/msl/msl.pdf

[Bru 01]: R. Bruni, U. Montarani, R. Francesca, "An Interactive Semantics of Logic Programming", Theory and Practice of Logic Programming, Vol 1, number 6, pp 647-690, 2001.

[CA 01] : B. Chatel, P. Attar ,"Les Schema XML, une introduction technique", Paris, 2001. http://./demo-schema/

[Cha 05] : G. Chagnon, "initiation aux schéma XML", mars 2005.
http://gilles.chagnon.free.fr/cours/xml/schema.html

[Clav 96]: M. Clavel, S. Eker, P. Lincoln, J. Meseguer, "Principles of Maude", Electronic Notes in Theoretical Computer Science 4, 1996.

[Clav 99 a]: M. Clavel, F. Duran,S. Eker, J. Meseguer, M. Stehr, "Maude as formal meta-tool", Lecture Notes in Computer Science, Vol. 1709, pp 1684-1703, 1999

[Clav 99 b] : M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott,"The Maude 2.0 System", Université Illinois de Urbana-Champaign, USA, 1999.
http://maude.cs.uiuc.edu/papers/pdf/maudeSlides.pdf

[Clav 00]: M. Clavel, F. Duran, S. Eker, p. Lincoln, N. Marti-Oliet, J. Meseguer, "A maude Tutorial", European joint conference on theary and practice of software, 2000.

[Clav 01]: M. Clavel, F. Duran, S. Eker, p. Lincoln, N. Marti-Oliet, J. Meseguer, J. F. Quesada, "Maude: Specification and Programming in Rewriting Logic", Theoretical Computer Science, June 2001.

[Clav 04] : M. Clavel, F. Duran, S. Eker, p. Lincoln, N. Marti-Oliet, J. Meseguer, "Maude Manual (Version 2.1)", Carolyn Talcott, 2004.
http://mirror.aiya.ru/pub/gentoo/distfiles/maude-manual.pdf

[Cov 98]: R. Cover, "XML and semantic transparency", 1998.
http://www.oasisopen.org/cover/xmlAndSemantics.html

[DB 06]: H. Douibi, F. Belala,"A semantic support for XML", the CSIT2006 4th International conference, 2006.

[DM 99]: F. Duran, J. Meseguer, "The Maude Specification of Full Maude", Computer Science Laboratory SRI International, May 1999.

[Dub 01]: E. Dubill, "The State of XML: Why Individuals Matter", 2001.
http://www.xml.com/lpt/a/2001/05/30/stateofxml.html7

[**Eng 99]**: J. English, "How to validate XML", 1999. http://www.flightab.com

[ES 01]: A. Erdmann, R. Studer, "How to structure and access XML documents with ontologies", Data and Knowledge Engineering 36, 2001.
http://www.aifb.uni-karlsruhe.de/WBS/Publ/2000/dke_merrst_2000.pdf

[Gad 96]: F. Gadducci, "On The Algebraic Approach To Concurrent Term Rewriting", Thèse PH.D TD-96-02, Department of Computer Science, University of Pisa, 1996.

[GC 01]: D. Girard, T. Crusso,"XML pour l'entreprise", 2001.
http://www.application-servers.com/livresblancs/xml/

[Lun 01]: V. Luncker Luc, "XML+XSL", 2001.
http://www.ccim.be/ccim328/xml/first.html

[Mcc 03] : T .McCombs, " Maude 2.0 Primer Version 1.0", August 2003.
http://maude.cs.uiuc.edu/primer/maude-primer.pdf

[Mes 92] : J. Meseguer, "Conditional Rewriting Logic as a unified Model of Concurrency", TCS 96, 1992, pp. 73 – 155.

[Mes 02] : J. Meseguer, "Rewriting Logic Revisited", Université Illinois de Urbana-Champaign, USA, 2002.
https://www.univ-savoie.fr/Portail/Groupes/LISTIC/Theses/these-Megzari-15mars2005_Finale-2.pdf

[Mes 03]: J. Meseguer, "Software specification and verification in rewriting logic", Models Algebras and Logic of Engineering Software, 2003.

[MM 93]: N. Marti-Oliet, J. Meseguer, "Rewriting Logic as a Logical and Semantic Framework", Technical Report SRI-CSL-93-05, Menlo Park, CA 94025, and Center for the study of language and Information Stanford University, Stanford, CA 94305, 1993.

[MM 95] : N. Marti-Oliet, J. Meseguer, "From Abstract Data types to Logical Framework", Lecture Notes in Computer Science, Springer-Verlag, Vol. 906, pp.48-80, 1995.

[MM 96]: N. Marti-Oliet et J. Meseguer, "Rewriting logic as a logical and semantic framework", Electronic Notes in Theoretical Computer Science, Vol. 4, 1996.

[MM 99]: N. Marti-Oliet, J. Meseguer, "Action and Change in rewriting logic", Applied logic Series, 1999.
http://citeseer.csail.mit.edu/cache/papers/cs/838/http:zSzzSzmozart.sip.ucm.eszSzpaperszSz1996zSzmom3.pdf/action-and-change-in.pdf

[MM 02]: N. Marti-Oliet, J. Meseguer, "Rewriting Logic: Roadmap and bibliography", Theoritical Computer Science, Vol. 285, number 2, pp 121-154, 2002.

[MR 04]: J. Meseguer, G. Rosu, "Rewriting Logic Semantics from Language Specifications to Formal Analysis Tools", University of Illinois at Urbana-Champaign, USA, 2004.
http://fsl.cs.uiuc.edu/~grosu/download/ijcar04.pdf

[Oba 04]: D. Obasanjo, "Amélioration de la validation d'un document XML avec Schematron", Microsoft Corporation, 2004.
http://www.microsoft.com/france/msdn/xml/2004-10-29-schematron.mspx

[Olv 00]: P. C. Olveczky, "Specification and Analysis of Real-Time and hybrid Systems in rewriting logic", PhD thesis, university of Bergen Norway, 2000.
http://maude.csl.sri.com/papers.

[PC 99]: G. Psaila and S. Crespi-Reghizzi, "Adding semantics to XML", Second Workshop on Attribute Grammars and their Applications, 1999, pp. 113 – 132.
[PS 03 a]: P.F. Patel-Schneider and J. Simeon, "Building the Semantic Web on XML" the Twelfth International World Wide Web Conference, ACM Press, 2003.

[PS 03 b]: P.F. Patel-Schneider, J. Simeon, "The Yin/Yang Web: A unified model for XML syntax and RDF semantics", IEEE Transactions on Knowledge and Data Engineering 15, 2003, pp. 797 – 812.

[Ram 98]: S. Rami. "De l'Unification des Modèles de la concurrence à Travers la logique de réécriture ", Thèse de Magister en Informatique, Institut d'Informatique Université Mentouri, Constantine Algérie, 1998.

[She 04]: L. Shengping and M. Jing and Y. Anbu and L. Zuoquan, "XSDL: Making XML Semantics Explicit", National Natural Science Foundation of China, 2004.

[Sil 03]: G.A. Silber, "Introduction à XML", 6 janvier 2003
http://www.cri.ensmp.fr/~silber/cours/xml

[Tha 02] P. Thati, K. Sen,, N. Marti-Oliet, "An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0", Proc. 4th. WRLA. ENTCS Elsevier, 2002.

[Tho 03]: J. J. Thomasson, "Schéma XML", Groupe Erolles, ISBN :2-212-11195-9, 2003, pp. 15 – 36.

[Usc 03]: M. Uschold, "Where are the semantics in the Semantic Web?", AI Magazine 24, 2003, pp. 25 – 36.

[VM 00]: A. Verdejo, N. Marti-Oliet, "Implementing CCS in Maude", Formal Methods For Distributed System Development, 2000.

[VM 02]: A. Verdejo, N.. Marti-Oliet, "Implementing CCS in Maude 2", Proc. 4th.WRLA. ENTCS Elsevier, 2002.

[Wor 02]: R. Worden, "MDL: A Meaning Definition Language", version 2.06, 2002.
http://www.ifs.univie.ac.at/wiim2003/77/SE_IM-Phase1.doc

[WP 04]: S. Wheller, S. Perchant, " Guide utilisateur <oXygen/>", syncRoSoft Ltd, 2004.
http://editor.oxygenxml.com/doc/oXygenUserGuide-standalone-fr.pdf

[WRL 04]: WRLA2004 5th International Workshop on Rewriting Logic and its Applications Barcelona, Spain, 2004.

Table des Matières

Chapitre 3 : Langage XML

Chapitre 4 : Un Support Sémantique pour XML