UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

2004 route des Lucioles
B.P. 93
06902 Sophia-Antipolis
France

Rapports de Recherche

N°1847

Programme 1

Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

ENVIRONNEMENT DE PROGRAMMATION POUR ASN.1

Anne Marie DERY
Christian HUITEMA
William MAILLE

Fevrier 1993

Environnement de programmation pour ASN.1

Anne Marie DERY - équipe CROAP -

Christian HUITEMA - équipe RODEO -

William MAILLE - équipe RODEO -

INRIA Sophia Antipolis

Programme 1

Environnement de programmation pour ASN.1

Résumé

Ce rapport décrit un environnement de développement d'applications distribuées basé sur le langage ASN.1. Le langage ASN.1, normalisé par l'ISO, est utilisé pour spécifier les structures de données échangées par les modules d'une application distribuée. Cet environnement a été réalisé en Centaur (développé par l'équipe CROAP) et est interfacé au générateur de stubs Mavros (développé par l'équipe RODEO).

L'environnement minimal comprend un analyseur syntaxique, un éditeur et un vérificateur de types qui facilitent l'édition syntaxique de descriptions ASN.1. Les afficheurs C et Lisp qui visualisent les déclarations ASN.1 et l'interface avec le compilateur Mavros facilitent l'écriture des clients et serveurs de l'application. Nous proposons également plusieurs outils dédiés à la mise au point de programmes ASN.1 dont un optimiseur et un générateur de tests.

An ASN.1 Programming Environment

Abstract

This report describes an environment for the development of distributed applications based on ASN.1. The ASN.1 language, standardized by ISO, allows the user to specify the data structures exchanged between the components of a distributed application. Implemented with Centaur (designed by the CROAP project), this environment is linked to the stub generator called Mavros (developed by the RODEO project).

The minimal environment is composed of a parser, an editor and a type checker; tools that facilitate structured editing of ASN.1 descriptions. The C and Lisp pretty printers that display ASN.1 declarations and an interface to the Mavros compiler facilitate writing clients and servers of an application. We also propose several tools for ASN.1 description development such as an optimizer and a test generator.

Table des matières

1 Introduction

1.1 Un environnement de programmation pour ASN.1 : pourquoi ?

A l'instar de "SUN" et son "XDR" (External Data Representation) [13], l'"ISO" a normalisé au niveau des couches supérieures (Présentation essentiellement) le langage de description de données ASN.1 (*Abstract Syntax Number One*). ASN.1 permet de décrire des types simples (booléens, entiers, énumérés, chaînes de bits et chaînes d'octets) et des types composés construits à partir de CHOICE, SEQUENCE, SEQUENCE OF SET et SET OF. Une description précise de ASN.1 est donnée dans the Open Book [4]. Une spécification des types des données transférés sur le réseau est compilée par des générateurs de stubs (RPCGEN, Mavros,...). Ces derniers génèrent une bibliothèque de fonctions comprenant les routines de codage et de décodage de valeurs qui est utilisée par les deux composants : client et serveur et qui facilite la mise en oeuvre de la communication.

Il n'existe actuellement aucun environnement de développement pour le langage ASN.1. Notre objectif est de fournir un ensemble cohérent d'outils permettant d'aider l'utilisateur dans l'écriture de spécifications ASN.1 et à plus long terme dans la conception de son application. Un tel environnement doit au minimum offrir les fonctionnalités classiques d'un environnement de programmation (éditeur, analyseur syntaxique, etc.). Pour simplifier l'écriture des programmes clients et serveurs, il nous a paru important de fournir également à l'utilisateur un support d'interface avec un générateur de stubs qui gère les demandes de compilation et de visualisation du code généré. Il est aussi intéressant d'optimiser les spécifications ASN.1 décrites dans l'optique d'avoir des transferts de données plus efficaces.

L'environnement de programmation pour le langage ASN.1 et le compilateur Mavros est conçu en Centaur. Centaur est une plate-forme logicielle permettant de développer un environnement de programmation interactif pour un langage donné à partir des descriptions syntaxiques et sémantiques de ce langage. Les environnements construits en Centaur sont constitués d'outils de types suivants : analyseur syntaxique, éditeur structuré, interpréteur-dévermineur... Les divers outils doivent être spécifiés dans les formalismes appropriés : METAL, PPML, TYPOL. La documentation de Centaur [3] présente dans le détail ces trois langages, l'article [2] donne un aperçu de l'ensemble du logiciel.

Le compilateur Mavros génère à partir d'un source ASN.1, l'équivalent des types et des valeurs ainsi que les routines d'encodage et de décodage appropriées dans les langages C [6] et Lisp [8].

Ce rapport comprend quatre parties décrivant les fonctionnalités de base de l'environnement.

1. une présentation des outils de base : analyseur syntaxique, éditeur standard et vérificateur de types.

2. Une interface conviviale et facilement paramétrable permettant de relier cet environnement de programmation sous Centaur au compilateur ASN.1 de son choix, en l'occurence Mavros.

3. Des outils d'optimisation permettant la réécriture automatique ou guidée de déclarations ASN.1.

4. Des outils de mise au point de l'application distribuée dans sa totalité.

Pour terminer, nous donnerons pour exemple le protocole de transfert d'arbres de syntaxe abstraite, défini dans l'équipe CROAP. L'environnement ASN.1 et l'environnement Lisp ont permis de spécifier la communication entre un client et un serveur Lisp.

2 Environnement minimal pour ASN.1 : Analyseur syntaxique, Editeur et Vérificateur de Types

Nous présentons dans cette partie l'environnement minimal pour ASN.1, c'est à dire les outils suffisants pour écrire une spécification ASN.1 correcte : un analyseur syntaxique, un éditeur syntaxique et un vérificateur de types.

2.1 Analyseur syntaxique

Le langage ASN.1 étant normalisé, nous avons pris comme point de départ une grammaire BNF publiée par l' "ISO" (référence 8824 - septembre 1987). La grammaire des types étant non ambigüe, la syntaxe concrète en METAL a été simplement déduite de la grammaire BNF. Pour définir la syntaxe abstraite, le principe choisi est d'associer un opérateur par type ASN.1 (simple ou constructeur). Ainsi la sémantique des types du langage est conservée et il n'y a pas de perte d'information au niveau de l'arbre généré.

La figure 1 montre la forme BNF correspondant aux constructeurs *SET* et *SEQUENCE* et la traduction METAL correspondante.

L'implémentation des valeurs est plus délicate. En effet, la grammaire BNF des valeurs de types complexes est ambigüe; bien qu'il soit possible de reconnaître syntaxiquement les différents éléments rentrant dans sa construction, il n'est pas possible de savoir si une valeur complexe correspond à un *SET* ou à un *SEQUENCE*. La solution adoptée consiste à simplifier la grammaire des valeurs et à ne considérer que deux catégories de valeurs : les valeurs simples (*SimpleValue*) et les valeurs complexes (*ComplexValue*).

A partir de la seule analyse de l'arbre de syntaxe abstraite, il sera impossible de savoir si une valeur complexe provient d'un *SET*, d'un *SEQUENCE*, d'un *CHOICE* ou d'un quelconque autre type complexe. Cependant ceci n'est pas un handicap majeur puisque ce type de vérification s'effectue typiquement lors de la compilation.

Dans la figure 2, nous indiquons la forme BNF correspondant à la définiton d'une valeur d'un *SET* et d'un *SEQUENCE*.

```
...
BuiltInType                     ::=     ...
                                        | SequenceType SetType
                                        | SequenceOfType SetofType
                                        | ...

SequenceType                    ::=     SEQUENCE {ElementTypeList}
                                        | SEQUENCE {}
SetType                         ::=     SET {ElementTypeList}
                                        | SET {}
SequenceOfType                  ::=     SEQUENCE OF {TypeList}
                                        | SEQUENCE OF {}
SetOfType                       ::=     SET OF {TypeList}
                                        | SET OF{}
...
```

La forme BNF

```
...
        <BuiltInType>           ::=     <SequenceType>;
                        <SequenceType>
        <BuiltInType>           ::=     <SequenceOfType>;
                        <SequenceOfType>
        <BuiltInType>           ::=     <SetType>;
                        <SetType>
        <BuiltInType>           ::=     <SetOfType>;
                        <SetOfType>
        ...
        <SequenceType>          ::=     "SEQUENCE" "{" <ElementTypeList> "}";
                        sequencetype(<ElementTypeList>)
        <SetType>               ::=     "SET" "{" <ElementTypeList> "}";
                        settype(<ElementTypeList>)
        <SequenceOfType>        ::=     "SEQUENCE" "OF" <Type>;
                        sequenceoftype(<Type>)
        <SetOfType>             ::=     "SET" "OF" <Type>;
                        setoftype(<Type>)
...
   abstract syntax
        BUILTINTYPE             ::= ... sequencetype sequenceoftype
                                        settype setoftype...
        sequencetype            -> ELEMENTTYPELIST;
        settype                 -> ELEMENTTYPELIST ;
        sequenceoftype          -> TYPE ;
        setoftype               -> TYPE ;
...
```

La grammaire METAL

Figure 1 : *Forme BNF et grammaire METAL des types complexes ASN.1*

```
...
BuiltInValue                    ::=     ...
                                        | SequenceValue SetValue
                                        | SequenceOfValue SetofValue
                                        | ...

SequenceValue                   ::=     {ElementValueList}
                                        | {}
SetValue                 ::=     {ElementValueList}
                                        | {}
SequenceOfValue                 ::=     {ValueList}
                                        | {}
SetOfValue                 ::= {ValueList}
                                        | {}
...
```

Forme BNF

```
...
        <Value>                 ::=     <ComplexValue>;
                        value(<ComplexValue>)
        <ComplexValue>          ::=     "{" <ElementValueList>"}";
                        complexvalue(<ElementValueList>)
...
   abstract syntax
        value                   -> VALUEGEN;
        VALUEGEN                ::= SIMPLEVALUE COMPLEXVALUE;
        complexvalue            -> ELEMENTVALUELIST;
...
```

La grammaire METAL

Figure 2 : *Forme BNF et grammaire METAL des valeurs complexes ASN.1*

2.2 Un éditeur standard

Le but de cet outil est d'utiliser les possibilités de l'outil PPML et des primitives du VTP pour offrir au programmeur ASN.1 un éditeur syntaxique d'ASN.1 qui facilite la saisie des types de données.

2.2.1 Le PPML standard pour ASN.1

La figure 3 reprend une partie de la grammaire abstraite d'ASN.1 avec la règle PPML correspondante. L'exemple pris est l'assignation d'une valeur à une variable respectant la notation *nameofvalue NameOfType ::= VALUE.*
Ces spécifications PPML permettent un réaffichage de l'arbre de syntaxe abstraite dans le for-

```
...
        <ValueAssignment>       ::=     <Idvalue> <Type> "::=" <Value>;
                        valueassignment(<Idvalue>,<Type>,<Value>)
...
   abstract syntax
        valueassignment         -> IDVALUE TYPE VALUE;
...
```

La grammaire METAL

```
prettyprinter std of ASN.1 is
...
valueassignment(*id, *type, *val) -> [<h 1> *id *type "::=" *val];
...
```

Le PPML

Figure 3 : *Assignation d'une valeur en METAL avec le PPML correspondant*

malisme ASN.1 standard c'est-à-dire en conservant la syntaxe concrète définie dans le METAL d'ASN.1. Ainsi le source obtenu est à nouveau conforme à l'analyseur syntaxique.
Les critères de formattage choisis sont de séparer par une ligne les declarations; de précéder chaque élément d'un type complexe par un retour chariot et de l'indenter sur le type; le parenthésage est aligné. Des ressources permettent de distinguer à l'aide de couleurs et de fontes différentes les types, les valeurs et les mots clés du langage.

Les figures 4 et 5 représentent l'édition dans Centaur d'un même source ASN.1. Dans le premier cas, aucun PPML n'est défini (représentation textuelle de l'arbre); dans le second, Centaur utilise le PPML standard pour afficher le source ASN.1.

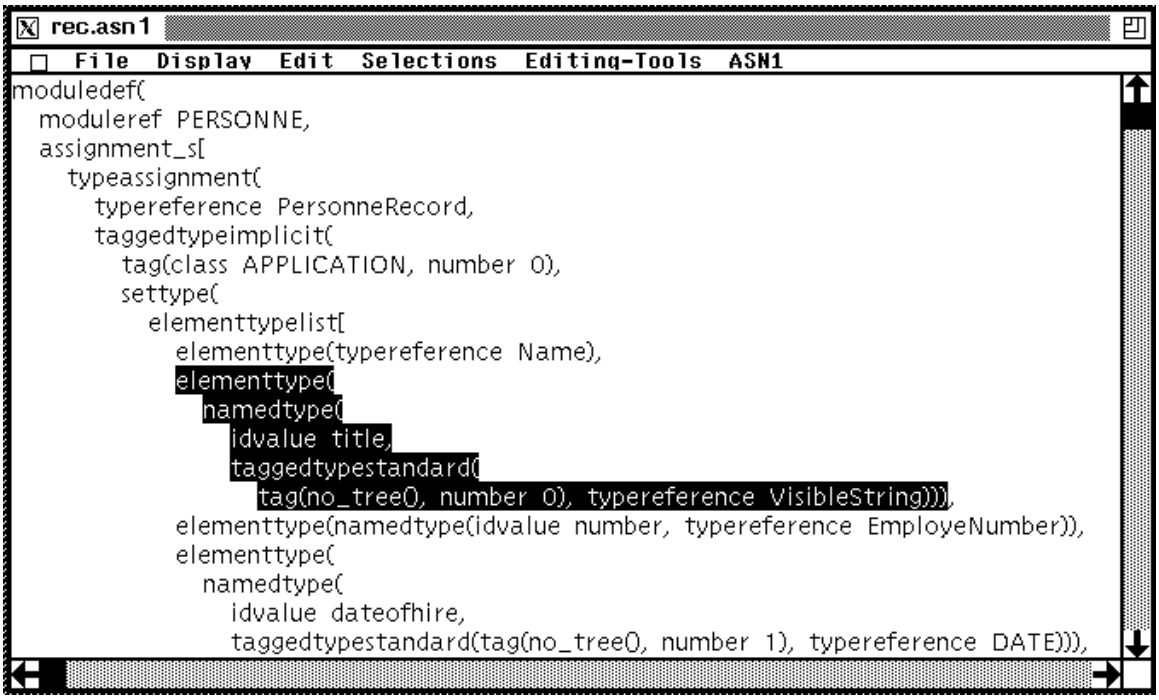

Figure 4 : *Edition par défaut d'un source ASN.1*

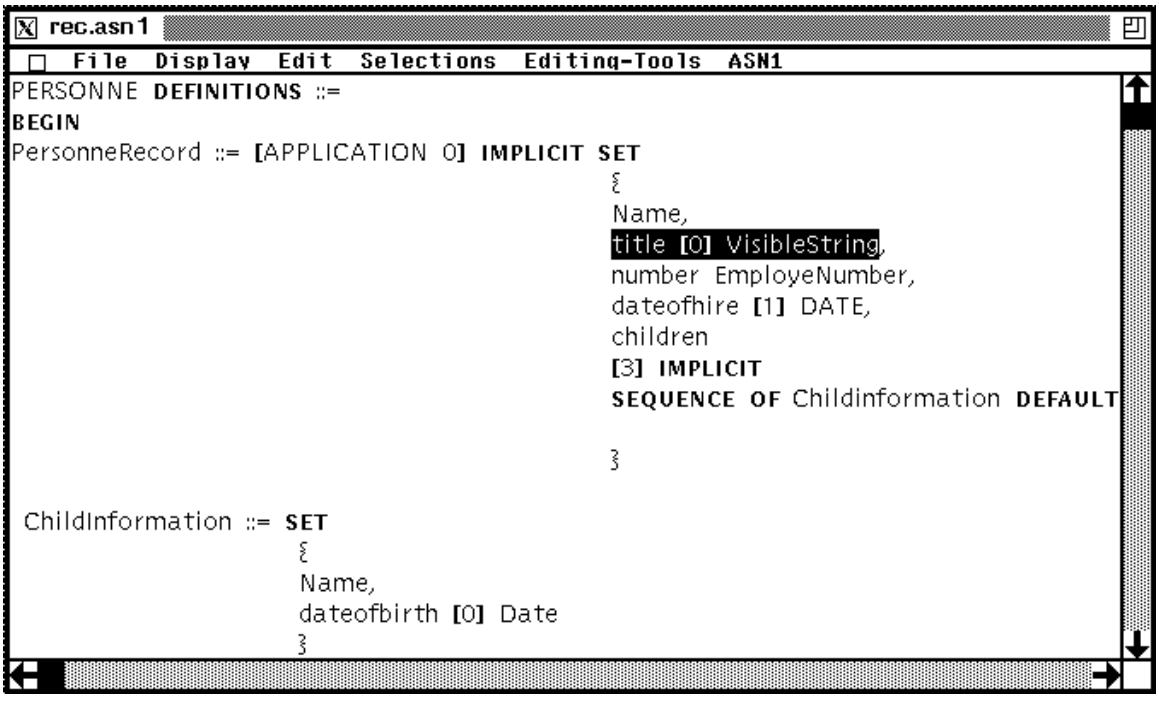

Figure 5 : *Edition avec l'afficheur standard*

2.2.2 Une fonction de recherche évoluée

En ASN.1, la définition d'un type complexe contient des références à des types déclarés en amont ou en aval de cette définition. De même la définition d'une valeur peut faire uniquement référence au nom du type. Ces types sont éventuellement définis dans un autre fichier ASN.1 et déclarés localement par l'intermédiaire de la clause IMPORTS <Nom de type> FROM <Nom de module>.
Pour l'utilisateur qui examine une spécification ASN.1, il est donc très utile de disposer d'une fonction qui permette d'obtenir la définition d'un type (qu'il soit au sein même du module ou qu'il soit importé) à partir de n'importe quelle référence à celui-ci.

Cette fonction permet donc de rechercher et d'afficher, dans une autre fenêtre, la définition d'un type ASN.1 en sélectionnant à la souris l'identificateur de ce type. Elle a été réalisée à l'aide de primitives Centaur qui permettent de décrire un menu (liste de boutons et d'actions associées) et de l'attacher à un éditeur de Centaur. Les primitives du VTP sont utilisées pour rechercher la définition d'un type dans un arbre donné.

Si le type est importé, le nom du module ASN.1 contenu dans la directive IMPORTS correspond au nom du fichier (.asn1) à charger dans Centaur afin de procéder à l'analyse syntaxique de la définition du type recherché.

La figure 6 visualise le menu **ASN1** placé automatiquement dans la barre de menus des fenêtres Centaur contenant des arbres ASN.1. Elle simule les possibilités offertes par la fonction *Recherche* : trouver la définition de **MPDU** importée de **P1** et ayant affiché cette définition, trouver la définition de **MPDU-Utilisateur**.

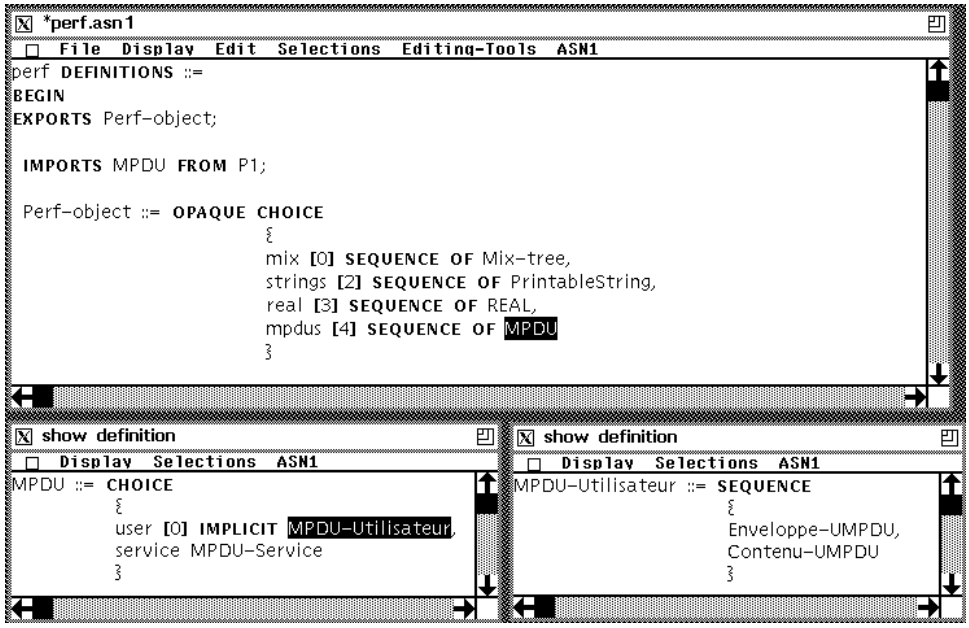

Figure 6 : *Recherche de définitions*

2.3 Un vérificateur de types pour ASN.1

L'objectif du vérificateur de types est de détecter d'éventuelles anomalies dans un source ASN.1 avant la compilation et de signaler les déclarations qui subissent un traitement particulier de la part du compilateur ou de l'optimiseur Mavros. Il est également important de suggérer au programmeur de modifier son source à certains endroits pour en améliorer l'efficacité (cf. 4).

Ce programme, décrit par des règles TYPOL, parcourt entièrement l'arbre correspondant au programme ASN.1 à vérifier et détecte deux situations particulières :

1. Une référence à un type qui ne se trouve pas dans le module ASN.1 et qui n'est pas importé. Dans ce cas, on déclenche l'affichage d'un message d'erreur indiquant que le type en question n'est pas déclaré.

2. Une déclaration de la forme

   ```
   SEQUENCE OF Type Complexe / SET OF Type Complexe
   ```

   Ces déclarations sont systématiquement expansées par Mavros en

   ```
   SEQUENCE OF Identificateur-de-type / SET OF Identificateur-de-type
   Identificateur-de-type ::= Type Complexe.
   ```

   L'affichage d'un message d'avertissement indiquant cette expansion permet alors au programmeur de l'effectuer lui-même et de choisir ainsi les identificateurs de type qui lui conviennent.

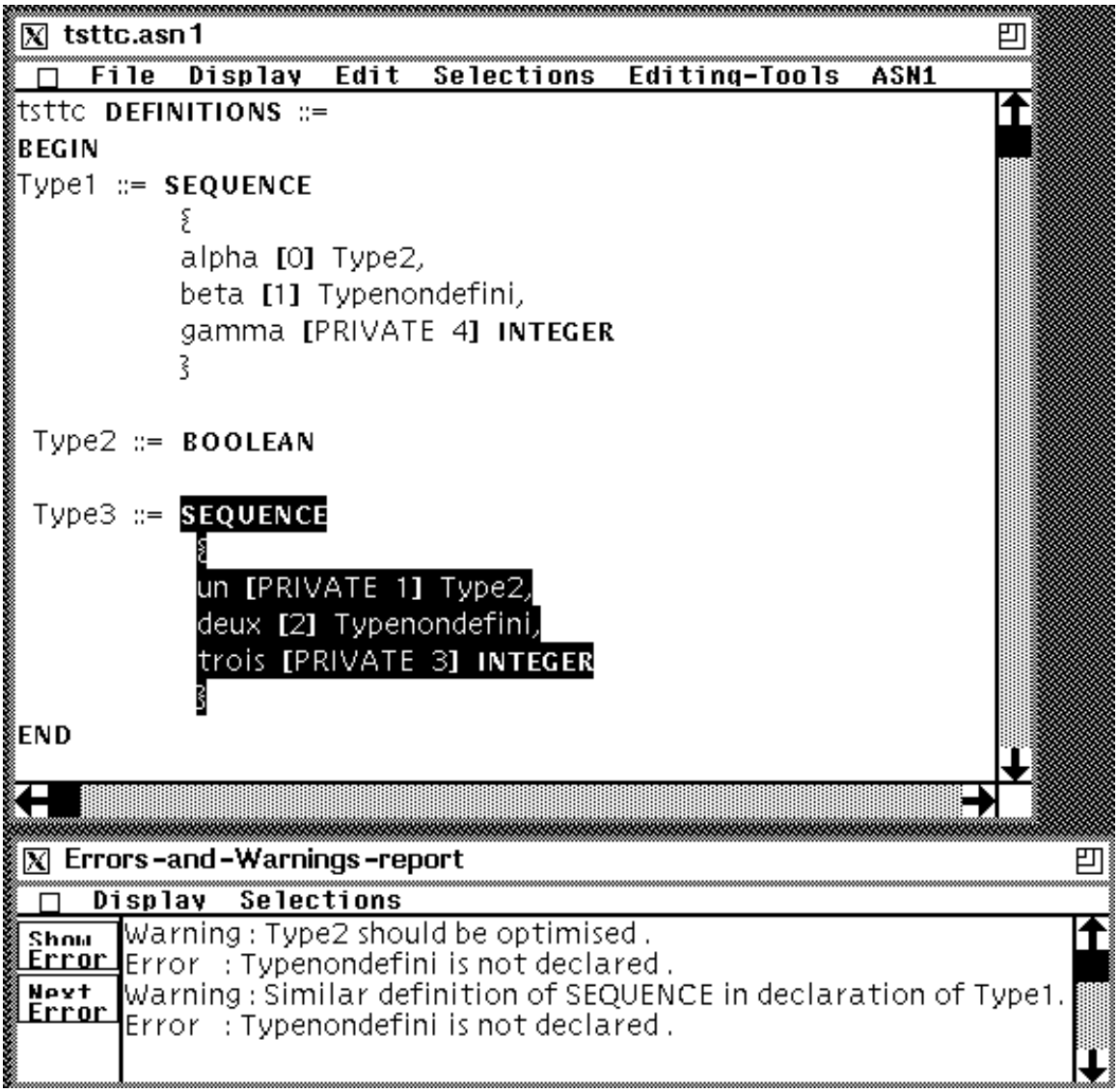

Figure 7 : Vérificateur de types

2.3.1 Spécification simplifiée du vérificateur de types en TYPOL

Nous donnons dans ce paragraphe, un exemple de spécifications TYPOL. Le premier paquet de règles construit l'environnement (ensemble des types définis dans l'arbre parcouru) et le second consulte ce paquet afin de vérifier si tous les types sont déclarés. Seules les règles significatives sont données, les autres règles permettent uniquement de se déplacer dans l'arbre.

```
program asn1_tc is
use ASN1;
import diff(_, _) from prolog;

judgement |- ASN1;

buildenv(env[] |- Modulebody -> e)  &  typecheck(e |- Modulebody)
----------------
|- moduledef(Moduleref, Modulebody) ;

   set buildenv is
   judgement ENV |- ASN1 -> ENV;

   .......

   e |- typeassignment(TPREF, _) -> env[TPREF.e] ;

   .......

   e |- typereference Idt -> env[typereference Idt.e] ;

   end buildenv;

   set typecheck is
   judgement ENV |- ASN1;

   .....

   env[typereference Tpref.e] |- typereference Tpref ;

   e |- typereference Tp
   ----------------
   env[typereference Tp1.e] |- typereference Tp ;
            provided diff(Tp, Tp1);
   end typecheck;
end asn1_tc;
```

2.4 Conclusion

Un environnement composé des trois outils présentés dans cette partie est déjà fort utile pour écrire des spécifications ASN.1. Cependant il peut paraitre insuffisant au niveau de l'aide à la définition des différents composants d'une application distribuée. Aucun outil n'est offert ni pour améliorer la communication par diverses optimisations ni pour tester et valider la spécification. Ces divers points sont présentés dans les parties suivantes.

3 Outils d'interfaçage avec Mavros

Dans cette partie, nous présentons les outils qui facilitent l'écriture des composants d'une application distribuée. Ce type d'outils est directement lié au compilateur choisi, dans notre cas, Mavros. Il s'agit de s'interfacer directement Mavros pour permettre de compiler une spécification et également pour offrir des outils de visualisation des données générées par ce générateur de stubs.

Le premier paragraphe présente les outils de visualisation du code généré et le second l'appel au compilateur Mavros.

3.1 Afficheurs du code généré

Le compilateur Mavros génère, à partir d'un source ASN.1, l'équivalent des types et des valeurs ASN.1 en C et en Lisp. La représentation C ou Lisp des valeurs est établie par le compilateur; elle entraîne parfois l'introduction de structures intermédiaires nommées automatiquement par le compilateur (cf. figure 8)

```
QP2 DEFINITIONS ::=
BEGIN

Operations ::= CHOICE
                 {op1  [0] Operation-a,
                  op2 [1] Operation-n,
                  op3 [2] Operation-e
                 } .....

END

donne

typedef struct Operations {
        int x;
        union {
                struct Operation_a * op1;
                struct Operation_n * op2;
                struct Operation_e * op3;
        } v;
} Operations;
```

Figure 8 : *Exemple de génération Mavros*

Il est indispensable pour écrire les clients et les serveurs de connaître les structures générées ainsi que les noms éventuellement introduits.

Il est donc particulièrement intéressant pour le programmeur de pouvoir visualiser l'équivalent de son source ASN.1 dans le langage cible de son choix. Il dispose alors de toutes les informations nécessaires à l'écriture des composants de l'application distribuée sans avoir à consulter le code généré par le compilateur ASN.1.

Un afficheur C permet de visualiser la liste des types C (.h) correspondant à la liste des types ASN.1 déclarés dans le source. Certaines structures C doivent contenir des informations

non constantes n'apparaissant pas dans l'arbre de syntaxe abstraite telles que les noms de champs générés. C'est notamment le cas pour les structures complexes ASN.1 telles que :

```
SET
  {
   Type1,
   id Type2,
   Type3
  }
```

Dans ce cas de figure, il faut génerer une structure C de la forme :

```
 {
  Type1 x_0;  (nommage indispensable en C)
  Type2 id;
  Type3 x_2;
 }
```

Ce type de difficulté à été résolue grâce à une fonctionnalité intéressante de PPML qui permet d'afficher dans une règle une chaine résultant de l'appel à une fonction LISP. On écrit alors une règle telle que:

```
elementtype(*named) -> [<h 1> if *named in {namedtype} then seqorset::*named
                        else imbriquer::*named  getindice(*named) ";" end if ]
```

où *getindice* est la fonction LISP suivante:

```
(de :getindice(tree)
 (lets ( (pere ({tree}:up tree 1)))
  (catenate "x_" (string (- ({tree}:rank pere) 1)))))
```

Les fonctions {tree}:up et {tree}:rank sont des primitives LISP du VTP de Centaur qui permettent respectivement de remonter dans l'arbre de syntaxe abstraite et d'obtenir la position d'un élément dans une liste.

L'afficheur Lisp visualise les signatures que génère le compilateur Mavros directement à partir de l'arbre de syntaxe abstraite résultat de l'analyse syntaxique d'un source ASN.1 standard. Cependant en Lisp il est plus utile de visualiser une valeur Lisp correspondant à un type puisque le langage est non typé (cf. 6.1).

On gère les problèmes d'affichages contextuels en spécifiant pour une règle d'affichage donnée, un contexte d'utilisation.
Exemple :

```
typeassignment(*ref, *type) ->
          [<v 1,0> [<h 1> "(" "defvar" [<h 0> "'" typerefssparen:: *ref ] "(" ]
          if not ( *type in {taggedtypestandard,taggedtypeimplicit})
          then "( )"  *type else *type end if "( )" ")"  ")"];
```

La figure 9 représente l'édition Lisp dans Centaur du source ASN.1 affiché par le PPML standard, dans la figure 3.

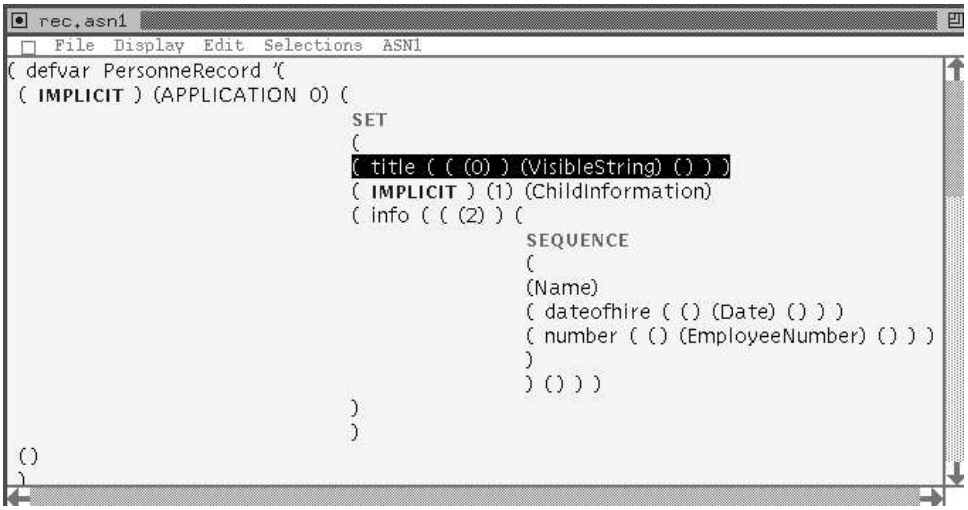

Figure 9 : *Edition avec le PPML 'LISP'*

3.2 Compilateur ASN.1

Le compilateur Mavros permet de compiler le source ASN.1 en C ou en Lisp. Il est important de permettre à l'utilisateur de faire appel au compilateur de manière interactive au sein de l'environnement.

L'exigence à ce niveau est double. Il faut un moyen convivial et rapide de déclencher l'appel de la compilation d'un source ASN.1 contenu dans un éditeur de Centaur tout en disposant également d'un moyen de paramétrer le compilateur (options de compilation). Une fonction LISP qui réagit à l'activation d'un bouton *COMPILE* lance l'exécution d'un fichier de commandes externe : makefile. Ce makefile possède une variable "MODULENAME" instanciée à l'appel de MAKE avec le nom complet (nom + chemin) du fichier contenu dans l'éditeur de Centaur. Le compilateur sait ainsi sur quel fichier il doit travailler.

L'utilisateur peut modifier ce makefile en changeant les options de compilation ou en ajoutant des commandes supplémentaires pour des actions qu'il désire effectuer avant ou après la compilation.

A terme on envisage d'avoir deux processus distants Centaur et Mavros pour améliorer la communication entre ces deux outils (cf. ref biblio).

3.3 Conclusion

Les deux afficheurs utilisent des ressources identiques à celles du PPML standard; ce qui permet à l'utilisateur de comprendre rapidement le mécanisme de génération des structures de données équivalentes à celles déclarées en ASN.1. Les recherches actuellement en cours pour la compilation d'ASN.1 en C++ s'orientent vers une implémentation beaucoup plus éloignée de C (mise à profit des concepts 'objet' de C++); ce qui imposera probablement dans l'avenir d'utiliser non plus PPML mais TYPOL pour C++.

L'ensemble des outils présentés jusqu'ici constitue un environnement de développement qui facilite le développement d'applications distribuées en donnant une aide dans l'écriture des composants distants.

Cependant, d'autres types d'outils sont proposés pour optimiser la communication et pour tester la spécification écrite.

4 Optimiseur de sources ASN.1

Le langage ASN.1 est utilisé principalement pour décrire les types de données qui vont transiter sur le réseau lors du fonctionnement d'une application distribuée. La conception de ces types de données influe donc sur la taille des messages échangés et par voie de conséquence sur les performances de l'application. Il est donc particulièrement intéressant d'étudier les possibilités d'optimisation d'un programme ASN.1. Cependant, pour cela, il faut être capable d'évaluer le gain que peuvent apporter certaines transformations. En effectuant des mesures de temps de codage et de décodage des différents types ASN.1, une métrique est batie. Les optimisations concernent également la taille du code généré par Mavros. Dans un tel cas, il n'est pas utile de construire une nouvelle métrique mais simplement de comparer la taille des fichiers générés.

4.1 Une métrique pour ASN.1

La métrique proposée est définie grâce à un programme en C chargé d'encoder et de décoder une valeur ASN.1 donnée un grand nombre de fois (de 100 à 1000 selon la taille de la valeur) et d'effectuer une moyenne du temps CPU.

Ce programme permet d'évaluer le temps de codage et de décodage de tous les types ASN.1 de base en fixant certains paramètres comme la taille des chaines d'octets, de bits, des constructeurs de types; des tags, des éléments optionnels et des eléments possédant une valeur par défaut. Les indirections de type dans un type complexe (cf. exemple1) sont aussi mesurées.

Exemple 1

```
Type1 ::= Type complexe { ...,Type2,...}
Type2 ::= ...
```

Plusieurs cas test ont permis de montrer que le temps moyen de codage/décodage d'un type ASN.1 quelconque obtenu par sommation des temps élémentaires obtenus pour chacun de ses constituants se révèle exact à dix pour cent près.

Un programme TYPOL, compte tenu des mesures obtenues, évalue le temps moyen de codage/décodage de n'importe quel type ASN.1. Bien que le résultat fourni est dépendant du compilateur et d'une machine donnés, le but de disposer d'un moyen de comparer des temps de codage/décodage entre deux versions d'une même structure de données est atteint. Un tel programme permet de mesurer les gains substantiels obtenus par optimisation d'un source ASN.1.

La figure 10 donne un aperçu des mesures choisies pour les types simples et pour les types complexes. Le coût d'une séquence de types (élément d'un SEQUENCE ou d'un SET) est évalué à la somme des coûts de chacun des types; le coût d'une alternative de types (élément de CHOICE) est approximé au coût du premier type de l'alternative.

Le paragraphe suivant présente l'optimiseur de source réalisé.

4.2 Un optimiseur de source

Ce module vise principalement à informer le programmeur d'éventuelles optimisations à effectuer sur le source ASN.1 qu'il vient de saisir et à mettre en place des procédures d'optimisation automatiques auxquelles le programmeur pourra faire appel s'il le souhaite.

L'optimiseur traite deux optimisations :

1. Le regroupement de déclarations similaires 4.2.1 ce qui permet d'optimiser la taille du code généré.