Le projet MMS[1]: l'informatique artificielle appliquée à l'implantation et au test de logiciels industriels

Jean Vaucher, Gregor v. Bochmann, Bernard Lefebvre, Stéphane Desmarais, Paul Gamache

Université de Montréal

Résumé

L'article présente un cas concret d'application des techniques et outils de l'intelligence artificielle dans un cadre industriel. Le projet MMS, commandité par IBM, avait deux objectifs: 1) évaluer la pertinence de Prolog pour le développement de logiciels de télécommunication et 2) explorer d'autres techniques "modernes" afin d'améliorer le cycle de développement des logiciels. L'étude s'est faite dans le contexte d'un produit de télécommunication en parachèvement: le Manufacturing Message Service (MMS) du Manufacturing Automation Protocol (MAP) destiné a l'interconnexion de robots et de machines outils dans les usines.

Notre étude a montré qu'il était possible d'implanter rapidement des parties d'un tel logiciel en Prolog. Cependant, vu la lenteur d'exécution, il n'y a de pas gros avantages à utiliser Prolog de cette façon. Par contre, nous avons trouvé un meilleur créneau pour la manipulation symbolique typique des langages d'intelligence artificielle comme Prolog: celui des outils d'aide aux développeurs.

Ayant constaté que la moitié des efforts pour l'implantation d'un produit comme MMS était destiné a l'activité TEST, nous avons mis sur pied une suite de logiciels "intelligents" pour automatiser une partie de cette phase. Au coeur du système, est une base de connaissance (BC) construite automatiquement à partir d'une analyse de la spécification formelle de l'interface des diverses fonctions. Plusieurs modules se greffent sur la base. Un "interviewer" s'occupe d'acquérir les informations spécifiques au test comme les valeurs "représentatives" des paramètres. Un "sélecteur" détermine un ensemble minimal de tests qui couvre toutes les possibilités intéressantes et un "générateur" se charge de produire des programmes de tests complets (en C) qui tiennent compte des initialisations particulières a chaque type de fonction et de paramètre. Des programmes compilables de plus de 800 lignes ont été générés.

La conclusion principale du projet est que l'intelligence informatique est assez efficace et robuste pour l'implantation d'outils utiles pour la conception et la

[1]Le projet a été financé par le Laboratoire d'IBM Canada à Toronto et les travaux ont été réalisés à l'Université de Montréal en collaboration avec le Centre Informatique de Recherche de Montréal (CRIM).

vérification de logiciels dans un cadre industriel.

1. Introduction

Le développement de logiciel est un processus difficile, long et coûteux. Il débute traditionnellement par l'analyse des besoins, passe par une ou plusieurs phases de spécifications pour aboutir à une implantation qui devra être testée avant d'être mise en application. Malgré des progrès certains, les méthodologies de développement modernes restent confrontées à l'augmentation de la complexité des systèmes à réaliser ainsi que leur coût. Récemment, de nouvelles approches ont été suggérées pour augmenter l'efficacité du processus de développement: le prototypage, les spécifications exécutables, la programmation orientée objet, les systèmes experts et la programmation logique.

Cet article explore l'utilisation d'une de ces voies: "l'intelligence informatique", c'est à dire l'emploi des techniques et des outils de l'intelligence artificielle, dans un contexte industriel. L'initiateur du projet a été le laboratoire d'IBM Canada à Toronto qui voulait explorer la pertinence de ces techniques pour le développement de produits logiciels en télécommunication. Initialement, le projet visait surtout l'essai de Prolog, un langage de programmation logique conçu pour le traitement symbolique typique des applications en intelligence artificielle; et un langage qui a connu beaucoup de succès pour le traitement du langage naturel, la représentation des connaissances et les systèmes experts [5, 6, 16]. Prolog devait être comparé à C dans un cadre d'implantation traditionnelle; mais, il est rapidement apparu que Prolog pourrait mieux servir à la création d'outils d'aide aux développeurs et cette voie a aussi été explorée.

L'application choisie par nos partenaires industriels était l'implantation du protocole MMS ("Manufacturing Message Spécification") [11, 12, 9 10], un protocole d'application pour l'automatisation des usines ("Manufacturing Automation Protocol", MAP) qui se place parmi l'ensemble des normes de protocoles de communication développées dans le cadre des Systèmes ouverts (voir par exemple [14]).

Une caractéristique des logiciels de communication est le fait que chaque implantation d'un protocole de communication, comme celui de MMS, doit se conformer à toutes les règles définies dans la spécification du protocole afin d'assurer sa compatibilité avec les logiciels de communication d'autres systèmes. Dans le cas des protocoles normalisés, ces spécifications sont données par les documents décrivant les normes. Pour organiser la complexité des protocoles de communication, les organismes de normalisation

(l'Organisation internationale de normalisation (ISO) et le Comité Consultatif International de Télégraphie et Téléphonie (CCITT)) ont élaboré un Modèle de référence de sept couches de protocoles. A partir de la première couche, qui fournit le service de transmission physique, jusqu'à la sixième, appelée la couche de présentation, chaque couche de protocole ajoute au service de communication qui est offert à la septième couche, appelée la couche d'application. Cette dernière contient les protocoles spécifiques aux applications. Le protocole MMS fait partie de cette catégorie.

Comme l'indique la figure 1, le protocole MMS a été conçu pour permettre une communication simple entre programmes d'application pour l'automatisation des usines. Par exemple, le protocole peut être utilisé par une application pour charger un programme de contrôle de robot sur un autre ordinateur, et pour ensuite communiquer avec celui-ci en échangeant des informations sur son statut et sur le contenu de ses variables. L'entité de protocole MMS, dans chaque ordinateur, fournit une interface (elle aussi normalisée) par laquelle les programmes d'application peuvent communiquer en échangeant des interactions, appelées primitives de services. L'entité du protocole transforme ces interactions en messages de protocoles (aussi appelés "unités de données de protocoles") qui sont échangés entre les entités à travers le service de communication offert par les couches 1 à 6 du Modèle de référence et le protocole de contrôle d'association (ACSE, "Association Control Service Element") qui fait partie de la couche sept, et qui sert à l'établissement de la connexion initiale entre les deux programmes d'application.

Insérer Figure 1 près d'ici

Dans ce qui suit, nous commençons par examiner, en général, le rôle des spécifications dans le développement de logiciel; puis, nous discutons, plus particulièrement, des normes internationales utilisées pour spécifier les protocoles de télécommunication. Ensuite, à la section 3, nous présentons l’approche “intelligente” à l’implantation de systèmes basée sur l’exploitation de connaissances. Nous soulignons le problème crucial de cette approche, soit l’acquisition des connaissances, et nous décrivons comment nous avons réalisé la création d’une base de connaissances à partir des documents de normalisation. À la section 4, nous présentons le résultat de nos expériences dans la première phase du projet MMS, avec Prolog comme langage d’implantation de logiciel de communication. Dans la section 5, nous décrivons nos résultats pour la deuxième

phase: utilisation de Prolog pour l'implantation d'outils d'aide aux développeurs - plus particulièrement, pour l'aide aux tests de validation des fonctions du produit. La conclusion sur le projet est donnée à la section 6.

2. Le rôle des spécifications

Les spécifications jouent un rôle clé dans le cycle de vie d'un système informatique. A chaque étape, la spécification sert au développement d'une implantation (ou d'une spécification plus détaillée). Elle peut aussi servir pour la sélection et l'analyse des tests à appliquer et, si elle est exécutable, la spécification peut être testée elle-même pour assurer sa validité (voir figure 2a).

Insérer Figure 2 près d'ici

Pour rendre le processus de développement de logiciel plus efficace, on essaie d'automatiser certains aspects du cycle de développement. Une automatisation implique aussi une certaine formalisation. Pour cette raison, des langages de spécification formelle ont été développés pour différents types d'applications et différents niveaux d'abstraction. Plusieurs langages de spécification, ou "techniques de description formelles", ont été développés pour les applications dans le domaine des protocoles de communication [1]. Toutefois, des documents informels feront toujours partie des spécifications, et il est important d'assurer que la correspondance entre les documents formels et informels soit clairement visible.

La figure 2b montre la relation entre spécifications formelles et les documents de normes qui définissent les protocoles souvent d'une façon informelle. Elle montre aussi leur utilisation pour les différentes activités du cycle de développement. Chaque flèche dans la figure correspond à une activité pour laquelle des outils d'automatisation seraient souhaitables. Dans le cadre des protocoles de communication, un bon nombre d'outils ont été développés pour les activités de la conception interactive, la validation, l'implantation, la sélection de tests et l'analyse de leur résultats [2]. Les sections suivantes décrivent les aspects d'automatisation pour l'implantation, la validation, la sélection et l'exécution de tests.

2.1. La spécification de systèmes répartis

Une phase importante dans la conception d'un système est la décomposition en modules et la spécification des interfaces entre ces modules. Dans le cas des systèmes répartis, il s'agit souvent d'adopter certains protocoles de communication déjà définis. Ces protocoles impliquent normalement une structure en couches, comme définie par le Modèle de référence des systèmes ouverts [14], et illustrée par l'exemple de la figure 1. Pour chaque protocole qui fait partie de la hiérarchie de protocoles adoptée, chaque ordinateur interconnecté contient un module, appelé "entité de protocole", qui réalise les opérations du protocole. En particulier, en fonction des interactions avec son usager dans la couche supérieure adjacente, il échange des messages de protocole codés avec son partenaire dans un autre ordinateur, utilisant le service de communication fourni par la couche de protocole inférieur adjacente.

Pour chaque couche de protocole, les spécifications des types suivants sont donc importantes (voir aussi [1]):

(a) La spécification de service, qui décrit (en termes abstraits) le service de communication fourni par la couche de protocole en question. Ceci inclut une description des interactions, appelées "primitives de service", qui sont exécutées à chaque point d'accès.

(b) La spécification de protocole, basée sur la spécification de service, décrit les messages échangés avec l'entité à distance en relation avec les primitives de services, et le codage de ces messages.

(c) Pour augmenter la transportabilité du code des programmes d'application qui utilisent le service de communication, il est utile de définir la forme de l'interface par laquelle les primitives de service sont réalisées à l'intérieur d'un ordinateur.

Dans le cas de MMS, les spécifications de service, du protocole et de l'interface sont décrites dans des documents de normes [11, 12, 9 10], essentiellement en langue naturelle, mais aussi par des diagrammes de transitions d'automates. Les structures de données des messages de protocoles, ainsi que leur codage, sont définies en utilisant le langage ASN.1 [13], couramment utilisé pour les protocoles d'application normalisés. Une définition similaire est fournie pour les structures de données de l'interface d'application. Une version de cette interface est destinée aux programmes écrits en langage C. Les structures de données des primitives de services réalisées par l'interface sont donc définies dans ce langage.

Les documents MMS sont très volumineux. Il est donc difficile de bien les comprendre ou de les vérifier. Cependant, ces spécifications peuvent être considérées comme une base de connaissances sur le protocole qui pourra être utilisée pour toutes les activités associées au développement des implantations. Dans la section suivante, nous explorons cette approche.

3. La programmation à base de connaissances

De plus en plus, les praticiens du génie logiciel et de l'intelligence artificielle convergent vers une approche commune au développement de systèmes complexes: la programmation à base de connaissances. Selon cette technique, on scinde un système en deux parties: une base de connaissances, contenant des règles qui régissent le comportement attendu du système, et un moteur d'inférence. Les règles se veulent déclaratives, spécifiant quoi faire sans dire comment faire et, pour être plus précis, nous définirons "connaissances" comme la représentation déclarative d'une partie d'un problème exprimée dans un langage "formel" (traitable par ordinateur). L'autre composante, le moteur d'inférence, a la tâche de dépister et d'interpréter ces connaissances pour atteindre un résultat suite à un stimulus ou une requête d'un usager.

Les systèmes à base de connaissances ont été rendus célèbres par le succès des systèmes experts; et le fait que les moteurs -- qui sont des programmes de taille et de complexité considérable -- sont invariants et réutilisables pour toute une gamme d'application a donné naissance à l'industrie des coquilles de systèmes experts.

Il y a aussi une autre façon d'exploiter la décomposition (moteur+connaissances). Avec un formalisme approprié, on peut réutiliser une base de connaissances commune pour diverses fins en lui adjoignant divers moteurs. C'est justement cette voie que nous avons exploité dans le projet MMS: nous avons d'abord conçu une base de connaissances centrale pour l'application que nous avons ensuite exploitée avec toute une suite d'outils implantés sous forme de moteurs d'inférence. L'avantage à long terme, c'est que les outils développés ne sont pas particuliers au projet MMS; ils pourront servir pour d'autres projets.

Pour que l'approche par connaissances soit rentable, il faut pouvoir acquérir ces connaissances. Dans le cas des systèmes experts, l'acquisition se fait généralement par des cogniticiens à partir de documents et d'entrevues avec les experts du domaine. C'est

un processus fastidieux reconnu comme étant le goulot d'étranglement du développement des systèmes intelligents [7]. Par contre, il existe souvent des documents qui ont le potentiel d'être analysés et convertis automatiquement en "connaissances". Le problème ici est de concevoir ces analyseurs et de s'assurer que le processus d'extraction est fiable.

3.1. L'acquisition des connaissances pour MMS

Dans le cas de MMS, les documents publics (normes) que nous avons utilisés pour le projet totalisaient près de 2000 pages. Ceci représente une masse d'information qui est potentiellement traduisible en "connaissances". Ce volume d'information est en lui-même un problème pour les développeurs qui n'arrivent pas facilement à naviguer à travers ces documents soit pour avoir une vue d'ensemble de l'application soit pour vérifier certains détails ou trouver tout ce qui se rattache à un problème particulier.

Un but de notre projet de recherche était donc de mettre une partie de ces connaissances dans une forme propice au traitement automatisé. Rapidement, nous avons écarté l'approche de compréhension (par ordinateur) des parties en langage naturel car il nous semblait peu probable que cette voie aboutisse à des résultats concrets dans les échéances prévues. Nous avons plutôt poursuivi deux autres approches:

a) Certaines parties des spécifications sont déjà formellement définies; elles peuvent alors être l'objet d'un traitement automatique. En particulier, nous avons pris les structures de données en C ("C bindings") de la spécification de l'interface d'application comme point de départ pour une base de connaissance, appelée "CDEF" qui a servi dans tous les volets de notre projet: pour la documentation à l'aide d'un fouineur (browser) interactif, pour la génération de certaines parties du programme Prolog qui implante une entité du protocole (voir section 4), et pour la génération de tests, comme décrit dans la section 5. (Il est à noter que les définitions des messages du protocole en ASN.1 auraient pu, elles aussi, servir pour la construction d'une base de connaissance similaire).

b) Pour obtenir une spécification exécutable du protocole, qui pourrait servir pour la validation et l'expérimentation, et pour l'analyse de résultats de tests d'implantation, nous avons fait une traduction manuelle des spécifications dans un langage exécutable de haut niveau, à savoir Prolog. Ce modèle a été conçu de manière à être compatible avec les parties déjà formalisées (voir point (a) plus haut), comme décrit dans la section suivante.

Dans ce qui suit, nous donnons la forme de la base de connaissances créée à partir des définitions C. Nous décrivons aussi le premier outil que nous avons construit: un fouineur (*browser*) permettant la consultation facile des informations de cette base. Bien que simple, ce fut le premier de nos outils à éveiller l'intérêt de nos clients industriels.

3.2. La base de connaissances: CDEF

Dans une première phase, la base (CDEF) a été construite en "parsant" les définitions C et en convertissant ces définitions en clauses Prolog correspondantes. Ces définitions sont principalement composées d'entêtes de fonctions, de définitions de constantes et de types (typedef) ainsi que quelques variables. La figure 3 montre un extrait des définitions concernant la fonction GET_NAME_LIST du protocole MMS (get_name_list agit un peut comme DIR du protocole FTP en retournant la liste des noms d'entités accessibles sur un autre site).

Insérer Figure 3 près d'ici

Pour traiter ces définitions, nous avons conçu un petit analyseur de 400 lignes de Prolog. Pour chaque définition, nous générons une clause de la forme suivante:

```
c_def( Nom, Genre, Definition, Autres )
```

qui donne: 1) le nom de l'entité définie, 2) son genre (pe. constante, fonction ou structure), 3) sa définition (pe. la liste de noms et types des champs d'une structure) et 4) des informations connexes comme l'endroit où la définition apparaît. Le seul champ complexe est *Definition* dont la structure reflète les diverses formes de déclarations possibles.

Pour la fonction "mm_gnlist" de la figure 3, la structure Prolog est:

```
c_def( mm_gnlist,  function,
    [  'Return_code',
       [  connection_id = ['Connection_id'],
          return_event_name = ['Local_event_name'],
          input_dcb = ['Mm_GNList_in_dcb', ref],
          inout_dcb = ['Mm_GNList_out_dcb', ref, ref]]],
    ).
```

Le lecteur intéressé trouvera plus de détails dans [8]. L'analyse de toutes les définitions utilisées dans le projet prend 50 minutes sur un IBM PS/2 modèle 80 avec Arity Prolog.

La base obtenue comprend 2285 définitions et son listing occupe 250 Koctets.

3.2.1. Le Fouineur

Afin de faciliter l'accès aux définitions contenues dans la base, nous avons implanté plusieurs outils de consultations (fouineurs ou *browsers*). Essentiellement, nous avons indexé les définitions selon la forme minuscule des premières lettres significatives. Ceci permet, par exemple, de retrouver la définition de la fonction **mm_read** avec les lettres "REA" ou celle de la structure **Mm_GNList_in_dcb** en ne donnant que les lettres "gnl". Les choix valables sont présentés dans un menu et le dépistage est relativement rapide (1 seconde sur la base complète). Un exemple de consultation est donné ci-dessous.

```
Search for:  read

  1  ...enough
  2  Mm_Read_in_dcb
  3  Mm_Read_info
  4  Mm_Read_out_dcb
  5  Mm_Read_specific_info
  6  mm_read

 Your choice:  6

mm_read: Function of type [Return_code]

Parameters:
   connection_id          = [Connection_id],
   return_event_name      = [Local_event_name],
   input_dcb        = [Mm_Read_in_dcb, ref],
   inout_dcb        = [Mm_Read_out_dcb, ref, ref]
```

On peut noter que l'interface est relativement primitive. A l'époque, l'environnement de développement ne permettait pas encore l'emploi facile d'interaction graphique; néanmoins, ce premier outil qui donnait un accès flexible et rapide à toutes les définitions citées dans les normes a été bien reçu par les développeurs de l'équipe MMS.

4. Implantation en Prolog du protocole MAP-MMS

L'implantation réalisée avec la version OS2 de Arity Prolog comporte plus de trois mille lignes de programme, elle ne peut être présentée que schématiquement dans un cadre aussi restreint mais le lecteur intéressé peut trouver dans [3, 4] les compléments d'information souhaités. Elle permet de simuler l'utilisation de ce protocole sur une seule machine mais entre plusieurs processus représentant des usagers distincts. Elle est suffisamment réaliste et complète pour permettre une évaluation objective des possibilités d'un langage de haut niveau comme Prolog dans ce contexte.

L'implantation en Prolog respecte aussi fidèlement que possible l'architecture générale définie pour les différentes composantes du protocole par ses concepteurs. La version OS2 de Arity Prolog utilisée dans le cadre de cette implantation ne disposant pas de l'environnement de développement complet nécessaire à la réalisation de ce prototype, des extensions ont été mises en place pour palier à ces absences sous forme de prédicats systèmes. Certains de ceux ci ont été définis en langage C et intégrés à l'interpréteur, d'autres sont écrits en Prolog. Ces extensions concernent la communication entre processus et l'usage de sémaphores publics au moyen de primitives OS2, la mise en place d'une gestion de termes sous forme d'enregistrements et un mécanisme de trace spécifique. L'usage de certaines de ces extensions est illustré par l'exemple de la fonction “status” en 4.1.2.

4.1. Architecture générale

Elle peut être divisée en deux parties principales: les fonctions de la bibliothèque qui peuvent être appelées par un programme utilisateur et le fournisseur de services: SP.

Insérer Figure 4 près d'ici

La figure 4 présentée ci dessus illustre les liens entre les diverses composantes du système. On distingue deux processus indépendants: le processus usager et le processus fournisseur de services. Ce sont des tâches distinctes fonctionnant de manière asynchrone et communiquant entre elles au moyen de files d'attentes OS2. Un usager peut communiquer avec un autre usager servi par le même fournisseur de service ou par un autre au moyen de messages acheminés au travers des différentes composantes du système. Ces messages se présentent comme des appels de fonctions et peuvent être répartis en deux groupes principaux: ceux permettant d'établir une connexion et les demandes et réponses de service.

Le processus SP est formé de plusieurs modules dont le principal est l'interface du fournisseur de services (ISP). La requête d'un usager parvient à l'ISP qui en vérifie la conformité au moyen du distributeur d'événement. Elle est ensuite acheminée vers la composante appropriée en fonction de son type (service de haut niveau, service à confirmer, fonction à contexte libre,...). Le traitement réalisé par ces composantes spécifiques permet de générer une ou plusieurs requêtes élémentaires MMS à partir de

l'appel initial. Celles ci sont transmises à la machine Protocole qui est responsable de leur traduction sous la forme de messages de protocole ACSE.

Le fournisseur de services de présentation (ACSE) implante les mécanismes d'adressage entre les différents usagers. Une demande de service ainsi confirmée peut alors être acheminée de l'ISP client vers l'ISP serveur qui envoie sa réponse éventuelle sous la forme d'une indication de service.

Les différents modules et sous-modules qui composent un fournisseur de service sont implantés par des prédicats Prolog sans arguments. L'ensemble fonctionne de manière synchrone et la description du synchronisme est spécifiée très simplement au moyen de prédicats précisant l'enchaînement des travaux:

```
main_loop :-
  (internal_input_queues_empty, cominfos(isp,ISP,_),
  wait_queue(ISP), ! ; true),
  comm_provider,  queues_state, ae_state,
  connection_state,
  wait,
  queue_processing,
  acse_p_processing,
  isp_processing,
  wait, !, main_loop.
main_loop.
```

La communication entre les modules dans l'implantation Prolog est assurée au moyen de files d'attente internes définies sous la forme de bases de faits gérées dynamiquement. Ces composantes sont ainsi structurellement indépendantes les unes des autres, elles peuvent être mises au point et testées séparément. Cette disposition est exigée par l'ampleur de la réalisation et la complexité des structures manipulées.

4.2. Exemple de la fonction Status

Les exemples suivants sont tirés des composantes du système relative à la bibliothèque de fonctions et à l'interface du fournisseur de service (ISP). Ils concernent la fonction “status” dont le rôle est de décrire l'état d'un serveur.

1. La bibliothèque de fonctions

Les traitements effectués dans le cadre de cette composante réalisent des contrôles de validité de paramètres et illustrent l'usage de certains des outils évoqués précédemment et de certaines propriétés utiles du langage Prolog lui même.

```
user_program_parameter_check(R) :-
  R iNSTOF Name,
  template(Name, R),
  user_program_parameter_specific_check(R).
```

Le prédicat “template” a comme rôle d'une part de s'assurer que la forme de l'appel d'une fonction est valide, c'est à dire conforme à la description que le système possède, et d'autre part d'affecter lorsque cela est nécessaire les valeurs par défaut. Ce résultat est obtenu très simplement en Prolog au moyen du principe d'unification. Dans l'exemple d'utilisation suivant on tente d'unifier un appel “FctCall” de la fonction mm_status avec la description de type correspondante au moyen du prédicat “iNSTOF” dont le rôle est de donner à un élément une structure de terme particulière (ici de type mm_status).

```
template(mm_status, FctCall):-
  FctCall iNSTOF mm_status,
  FctCall with [return_event_name=REN].
  default_values([REN,0]).
```

L'utilisation de termes complexes pour représenter les très nombreuses structures de données de MMS rend indispensable l'emploi d'outils de manipulation tels que ceux définis dans [17] et illustrés dans l'exemple précédent. Ces outils ont été améliorés et étendus dans le cadre de cette réalisation. Ils permettent un accès par nom aux différentes positions de termes Prolog ce qui facilite grandement leur exploitation. Ces termes doivent d'abord être décrits au moyen d'un prédicat “def_record”:

```
def_record
   mm_status(
         connection_id : 'Connection_id',
         return_event_name : 'Local_event_name',
         input_dcb : 'Mm_Status_in_dcb',
         inout_dcb : 'Mm_Status_out_dcb').
def_record 'Mm_status_in_dcb'(
     context_id : 'Mm_Context_id',
     extended_derivation : 'Bool').
```

La structure des termes apparaissant dans les “def_record” est générée automatiquement à partir des définitions incluses dans CDEF. Le type de la composante “input_dcb” du terme composé “mm_status” est “Mm_Status_in_dcb” qui est lui même structuré et décrit par un autre prédicat “def_record” dans lequel les types des composantes “context_id” et “extended_derivation” sont respectivement “Mm_Context_id” et “Bool” qui correspondent à des types simples. Les termes ainsi décrits peuvent alors être utilisés à la manière de l'exemple suivant:

```
P iNSTOF mm_status,
P with [input_dcb..context_id := 1],
P with [input_dcb..context_id = X]
```

Le prédicat “iNSTOF” créé un exemplaire de type mm_status que le prédicat “with” exploite ensuite pour affecter la valeur 1 à la composante “input_dcb..context_id”. Cette valeur est ensuite demandée et unifiée à la variable X par le biais de ce même prédicat.

2. L'interface du fournisseur de services

Cet autre exemple provient de la composante ISP et concerne les services à confirmer. Cette description de traitements montre l'utilisation du mécanisme de trace à l'entrée. Prolog est un langage interprété qui dispose d'un mécanisme de trace évolué très utile pour la mise au point de programmes. Les tests de l'implantation du protocole lui même nécessitent cependant des moyens spécifiques qui permettent de suivre simplement le cheminement des requêtes dans les différents modules et couches de la réalisation. La mise en place de ces moyens a été facilitée par l'architecture modulaire qui a été adoptée. Elle est obtenue par un prédicat “control_trace” qui comporte en argument une liste d'options. Outre la possibilité de suivre le cheminement d'une requête avec une précision variable, celles ci permettent en particulier de visualiser l'état des files d'attente et l'état des connexions. Ce prédicat définit une base de faits Prolog “tracing_options” qui est exploitée à chaque transition jugée importante au moyen du prédicat “proc_trace”.

Cet exemple illustre également la modularité, la concision et le pouvoir d'expression du langage Prolog. Ces facultés permettent une programmation proche des spécifications exprimées dans les documents décrivant le protocole.

Implantation Prolog	**Spécifications informelles** [9, 10]
csp_processing(request,U,R,Ret) :-	1. Recevoir la requête de l'usager.
proc_trace([csp_processing,request,U,R,Ret]),	
generate_invoke_id(U,I),	2. Générer l'identificateur d'invocation.
psp_processing(request,I,R,Ret),	3. Passer la requête au PSP.
psp_processing(solicit_confirmation,I,R, Ret),	4. Solliciter une confirmation du PSP.
R with [connection_id = Cid],	5. Solliciter une indication de rejet
R with [return_event_name = Ren],	auprès de l'IFA.
template(mm_reject_ind,Rj),	(Note relative à l'implantation Prolog:
Rj with [connection_id = Cid],	Génération de l'indication de rejet à

Rj with [return_event_name = Ren], ifa_processing(solicit_indication, csp_processing,I,Rj,Ret),...	l'aide d'un prototype non instancié fourni par template et instanciation des variables)

4.3. Avantages et inconvénients de l'approche Prolog

Cette implantation limitée mais réaliste du protocole MMS a pu être réalisée par une équipe restreinte de deux personnes. Le coût global de ce développement en temps de travail est inférieur à 1600 heures ce qui peut être considéré comme faible. Cela est en particulier du au pouvoir d'expression du langage de haut niveau qu'est Prolog. La mise en oeuvre des prototypes est facilité par l'extrême concision du langage et sa versatilité. Les prédicats réalisant des traitements spécifiques peuvent être décrits d'une manière générique puisque leurs arguments ne sont que faiblement typés et que ce typage restreint est implicite et de type dynamique. Le contrôle des arguments et de la forme des fonctions MMS doit être mis en place de manière explicite ce qui pourrait être évité par l'utilisation d'un langage fortement typé mettant en oeuvre un contrôle dynamique des types. Le langage C comme la plupart des langages d'usage courant ne possède pas cette caractéristique et dans ce cas, comme en Prolog, ce traitement doit être mise en place explicitement avec l'obligation supplémentaire de maintenir un ensemble volumineux et complexe de déclarations. Dans le cas de Prolog cet écueil est toutefois compensé par la possibilité de définir des mécanismes de contrôle puissants en utilisant notamment le concept d'unification. Celui ci s'est avéré en particulier très utile pour permettre l'affectation de valeurs par défaut ou pour vérifier la conformité à un modèle. A cet égard et sur un plan plus général, il ne fait pas de doute que les extensions récentes de Prolog permettant la spécification et l'utilisation de contraintes, offrent une gamme de moyens qui permettrait de traiter d'une manière plus élégante ce type de traitements. Ces solutions prometteuses n'ont toutefois pas pu être mises en place dans le cadre de ce projet faute de temps mais surtout de moyens puisque ces extensions font encore partie du domaine de la recherche et peuvent difficilement être considérées dans ce contexte.

Prolog, au même titre que la plupart des langages récents, autorise la modularité et permet ainsi la réalisation d'implantations qui respectent l'architecture et l'organisation hiérarchique des modèles. Cette modularité est mise en place dans ce cadre par le biais de prédicats qui implantent les différents niveaux.

Prolog est un langage qui peut être interprété et compilé. Son utilisation en mode interprété facilite la mise au point à divers titres. Les ajouts ou modifications de prédicats peuvent en particulier être testés, sans délais, au moyen d'un mécanisme de trace intégré et évolué qui permet de suivre les différentes phases de la résolution. La concision de l'expression des solutions Prolog est un autre facteur qui facilite la mise au point. Il est plus facile de gérer un code concis et clair qu'un ensemble volumineux de plusieurs centaines de fonctions écrites en C qui, bien que structuré de façon identique, est plus long à lire et à comprendre.

La compilation de Prolog permet d'obtenir une version exécutable beaucoup plus rapide dans le cas de programmes fortement récursifs. Comme cette caractéristique n'est que très partiellement présente dans l'implantation MMS, les gains obtenus dans ce cadre sont modiques. La version finale est environ dix fois plus lente que son équivalent écrit en C. Ce facteur temps est le handicap majeur de cette implantation Prolog. Si ce handicap (prévisible) n'a pas nui au développement et aux tests, il pourrait être plus gênant lors d'une exploitation réelle et pourrait rendre nécessaire l'utilisation d'un matériel plus performant et donc plus coûteux.

5. Outils “intelligents” pour le TEST

Dans le projet MMS, nous avions d’abord la tâche d’évaluer l’utilité de Prolog pour l’implantation; mais, très tôt, nous avons pu identifier des créneaux (autres que l’implantation) où l’intelligence informatique pouvait être appliquée avec autant sinon plus de profits. En particulier, nous avons noté qu’une partie importante du travail de l’équipe de développement était consacrée à la planification et à l'exécution de tests. Il nous est donc apparu que la conception et le développement d’outils d’aide au TEST pourraient être très utile. Cette recherche était doublement intéressante pour nous car ce domaine semble avoir été négligé par les chercheurs. Nous présentons d’abord un aperçu de l’activité TEST, puis nous exposons les outils qui ont été conçus et implantés.

5.1. Le TEST dans le cadre de MMS

Un protocole comme MMS se présente à l'usager comme un ensemble de fonctions qui donne accès aux divers services implantés. En tout il y a quelques 103 fonctions particulières à MMS qui permettent, dans un environnement partagé, de transférer et d'exécuter des programmes, de gérer des variables, des événements et des sémaphores,

etc... A ceci s'ajoutent une quinzaine de fonctions pour gérer la création et l'interconnexion de machines virtuelles et quelques 28 fonctions, dites *context free*, pour la création et la manipulation (de manière indépendante de la machine) des structures de données utilisées comme arguments dans les appels des fonctions MMS.

L'équipe chargée du test d'un produit comme MMS, doit s'assurer que chacune de ces quelques 150 fonctions se comporte comme prévu dans les spécifications. De plus, comme ces fonctions sont accessibles à l'usager, il faut non seulement en vérifier le bon fonctionnement avec des valeurs raisonnables des arguments; mais aussi le comportement pour toute combinaison de valeurs possibles et imaginables.

> Note: Pour les protocoles de télécommunication, il faudrait aller encore plus loin en explorant et vérifiant le comportement des fonctions dans tous les états prévus par le protocole. Dans nos travaux, nous nous sommes bornés au test fonctionnel: la vérification d'un seul appel de fonction sous test. Nous n'avons pas étudié la génération de suite d'appels afin d'exercer les transitions d'état.

En fait, le test exhaustif se bute rapidement au phénomène de l'explosion combinatoire et l'art du testeur réside dans le choix d'une suite minimale de tests qui exerce tous les cas typiques d'opération et qui provoque au moins une fois chaque catégorie d'erreur possible.

En pratique, le testeur détermine d'abord, pour chaque paramètre, un ensemble de valeurs représentatives des valeurs correctes et un ensemble représentatif des valeurs erronées. Souvent il sélectionnera des valeurs aux extrémités des intervalles acceptables. Par exemple, pour un paramètre défini qui doit être un entier positif (>= 0), il pourra choisir les valeurs suivantes: {0,1 et **maxint**} pour les bonnes valeurs et {**minint**, -1} pour les mauvaises.

Deuxièmement, le testeur planifie une suite d'essais qui couvre ces valeurs. Il faut que, dans cette suite, chaque bonne et mauvaise valeur ait été utilisée au moins une fois. Pour les valeurs erronées, il faut planifier un essai (appel de fonction sous test) par valeur; mais, pour les bonnes valeurs, des économies sont possibles: on peut souvent vérifier en parallèle les bonnes valeurs de plusieurs paramètres. Par exemple, si pour la fonction F(X,Y), les valeurs représentatives de X sont {a,b,c} et celles de Y sont {m,n}, on pourra tester toutes ces valeurs avec juste 3 essais: F(a,m), F(b,n) et F(c,m).

Bien que les principes soient simples, l'application dans le cas d'un produit comme MMS

est assez lourde. Ceci pour 3 raisons.

1) Typiquement, les arguments de chacune de ces fonctions sont des structures complexes comprenant un nombre important de champs imbriqués et plusieurs de ces champs doivent être créés dynamiquement. Ceci veut dire que le calcul des valeurs d'une suite d'essais pour tester adéquatement une fonction est assez complexe. D'autre part, le code qu'il faut écrire pour créer et initialiser les arguments n'est pas trivial.

2) La situation est compliquée par le fait qu'en télécommunication, des actions initiales et terminales sont nécessaires pour créer le contexte nécessaire à l'exécution d'une fonction sous test. Par exemple il faut prévoir un autre programme avec lequel le programme sous test pourra dialoguer, il faut créer une machine virtuelle (VMD), ouvrir une connexion et insérer les identificateurs de la connexion dans les paramètres d'appel. Après l'appel, il faut libérer les structures allouées et effectuer la déconnexion.

3) Finalement, il existe des relations entre divers champs des paramètres. Par exemple, pour des chaînes, on a, le plus souvent, un champ entier qui doit contenir la longueur d'une chaîne pointée par un autre champ. D'autre part, certains arguments peuvent avoir diverses structures dépendant du contenu d'un autre champ (indicateur). Tout système de génération automatique de valeurs de test doit tenir compte de ces dépendances.

5.2. Les outils d’aide au TEST

Les outils que nous avons conçus visent le test d'une fonction à la fois. Ils génèrent une suite de programmes en C qui exercent cette fonction avec toutes les valeurs représentatives de ses paramètres. Le travail est effectué en trois phases, chacune mise en oeuvre par un programme:

1) L'interviewer:
   ce programme fait l'acquisition et l'ajout à la base de connaissance CDEF des informations additionnelles nécessaires au test (valeurs représentatives et dépendances).

2) Le sélecteur:
   ce programme détermine le nombre de tests à faire et les combinaisons de valeurs de paramètres à utiliser afin de couvrir tous les cas prévus.

3) Le générateur:
celui-ci fait la génération de programmes complets en C avec le code nécessaire pour établir la connexion et initialiser les paramètres

5.2.1. L'interviewer

La première étape dans le test d'une fonction est la détermination des valeurs représentatives (bonnes et mauvaises) pour chacun des paramètres. Initialement, ces informations ne sont pas dans la base CDEF et il faut les acquérir d'une personne responsable des tests. Dans certains cas, en particulier celui des paramètres définis par énumérations, il est facile de générer des valeurs sensées mais il est préférable de les faire valider par le testeur.

L'acquisition de ces informations se fait interactivement par l'entremise d'un programme d'entrevue qui exploite les données stockées dans CDEF pour guider le dialogue. A partir des définitions dans la base le programme peut déterminer le nom et le type de chacun des champs des paramètres. Il peut donc s'assurer que toutes les indications nécessaires sont fournies. Pour chaque champ, les informations suivantes sont recueillies et associées au champ et à son type dans la base CDEF:

1) le mode: entrée ou sortie (faut-il spécifier une valeur pour le paramètre avant l'appel ou vérifier son contenu après?),
2) la liste représentative des bonnes valeurs qu'il faudra tester pour ce champ,
3) la liste correspondante de mauvaises valeurs,
4) la dépendance avec des valeurs d'autres champs.

Notre système essaie de réduire au minimum le travail du testeur. Si un type est réutilisé pour une autre fonction, les valeurs déjà entrées seront proposées afin que le testeur puisse les garder ou les modifier. De même, dans le cas de types énumérés, le système propose les constantes associées au type. Les informations sur un paramètre sont codées et stockées dans CDEF sous forme d'une paire d'attributs dont la syntaxe est comme suit:

```
val  = [bonnes valeurs..]-défaut-[mauvaises
valeurs...],
cond = [ mapping des paramètres ] => condition prolog
```

La "condition" signifie que les valeurs définies pour l'attribut "val" sont à utiliser seulement si l'évaluation de la "condition prolog" réussit. Les valeurs d'un champ peuvent ainsi dépendre de celles d'un autre champ. Dans l'exemple ci-dessous, `'Mm_Identifier'` est un type MMS courant utilisé pour passer des chaînes en

paramètre. C'est une structure avec deux champs: le premier est un pointeur sur la chaîne et le second contient sa longueur. La définition donne `"Aa1_8"` et `"null"` comme des valeurs typiques d'un `'Mm_Identifier'` et `"a 1"` comme valeur erronée (a cause du blanc). La longueur dépend évidemment de la chaîne; l'attribut "val" spécifie L comme unique bonne valeur et LL comme mauvaise valeur où L et LL sont obtenues par calcul.

```
known_type('Mm_Identifier',
   [pointer = ['Uint32',
     val  = ["Aa1_8","null"] - "null" - ["a 1"],
     cond = ( => true)],
   length = [ref - ['Octet'],
     type ='Octet_pointer',
     val  = [L] - L - [LL],
     cond = ([pointer = Pointer] =>
                    name(Pointer,Sy),
                    length(Sy,L), LL is L-1]
]).
```

Une fois les valeurs représentatives et les dépendances connues, il est possible de passer à la phase suivante.

5.2.2. Le Sélecteur

Le SÉLECTEUR est un programme qui génère des combinaisons de valeurs représentatives à utiliser comme paramètres d'entrées à une fonction sous test. On distingue deux modes de fonctionnement pour le SÉLECTEUR: la génération de cas valides où l'on trouve exclusivement les "bonnes" valeurs représentatives et la génération de cas comprenant des erreurs.

Avec le premier type de test, le défi est d'utiliser toutes les valeurs représentatives avec un nombre minimum d'essais. La technique est celle que nous avons décrite précédemment, soit l'exploration "en parallèle" des suites de bonnes valeurs pour chaque paramètre. On peut aussi spécifier des regroupements "combinatoires" de champs pour lesquels on veut générer toutes les combinaisons de valeurs possibles. Dans tous les cas, notre programme respecte les dépendances spécifiées. Les détails de l'algorithme utilisé se trouvent dans [3, 8].

Insérer Figure 5 près d'ici

La figure 5 montre une sortie du programme pour les tests valides de la fonction **mm_gnlist**. Le format correspond à celui qui est employé par les testeurs d'IBM pour planifier leurs tests. On voit, à gauche, les noms abrégés des champs d'entrée. Dans la colonne suivante, on liste les valeurs représentatives pour chaque champ. Les X à droite indiquent quelle valeur de champ utiliser pour quel test. La figure montre que 12 tests seront nécessaires pour explorer toutes les valeurs proposées. Ce nombre vient des 12 valeurs prévues pour "object_class". Notez l'exploration parallèle des valeurs pour les divers champs: dans les 2 premiers tests, on utilise des valeurs différentes non seulement pour le champ "**object_class**" mais aussi pour "**return_event_name**", "**continue_afterpoi...**", et plusieurs autres. Les valeurs générées tiennent compte de plusieurs dépendances inscrites dans les spécifications: les 2 champs de "...length" ont bien comme valeur la longueur des chaînes associées. De plus, les 2 champs "**continue_after...**" n'existent que si "**mask**" a la valeur "bit(0)" et les deux derniers champs "**domain_name...**" dépendent du fait que "**object_scope**" soit égal à "Mm_OS_AA_specific".

Le programme génère aussi une autre sortie destinée à la phase suivante. L'exemple ci-dessous illustre le format utilisé pour désigner de façon complète (mais abstraite) les paramètres:

```
/* ************** Case 1 *************** */

input = [
       connection_id = 1 ,
       return_event_name = 0 ,
       input_dcb .. ref .. mask = bit(0) ,
       input_dcb .. ref .. continue_after .. pointer = "Aa1_8" ,
       input_dcb .. ref .. continue_after .. length = 5 ,
       input_dcb .. ref .. context_id = 0 ,
       input_dcb .. ref .. object_class = Mm_OC_Domain ,
       input_dcb .. ref .. object_scope = Mm_OS_AA_specific ]
```

5.2.3. Le Générateur

Ce programme génère des programmes complets en C qui effectuent les tests sélectionnés. Le problème principal de tout générateur de code est la conception préalable d'un formalisme dans lequel spécifier le programme qui doit être généré. Pour ce projet, nous avons inventé une notation simple où un programme est décrit par une suite de fonctions à générer et pour chaque fonction on peut spécifier des attributs dont les plus importants sont 1) une liste de valeurs à affecter aux paramètres d'entrée, 2) une liste de vérifications à faire sur les paramètres de sortie et 3) une liste de variables à unifier avec les variables de sorties (afin de pouvoir les utiliser comme entrée pour des

fonctions suivantes). L'exemple suivant montre la spécification complète pour la génération d'un test sur la fonction **mm_status**:

```
:- pgm(tsta001l,
     [ mm_status with [
           site = local,
           input=  [input_dcb..ref..context_id = 0,
                     input_dcb..ref..extended_derivation
                    = false],
           check= [inout_dcb..ref..[
                     size >= cast('Uint16')..0,
                     error_block..mask = '"allo"'],
                     input_dcb..ref..context_id = 0],
           output= [ inout_dcb = NULL] ] ]),
```

Insérer Figure 6 près d'ici

Ceci a généré un programme de 170 lignes qui a pu être compilé et exécuté. Un extrait de ce programme est donné à la Figure 6 et le code qui correspond directement à l'appel de fonction est encadré. Ce listing montre la diversité des éléments qui doivent être générés à partir d'un simple appel de fonction. En particulier, on peut noter:

- les commentaires,
- les énoncés d'inclusions de fichiers pertinents,
- la création de noms pour les paramètres de la fonction,
- la déclaration de ces paramètres,
- l'allocation d'espace pour des paramètres dynamiques,
- l'initialisation de la connexion,
- l'affectation des valeurs aux paramètres d'entrée des valeurs prévues pour le test,
- les messages d'erreurs appropriés au test,
- la vérification des valeurs retournées,
- la libération des structures allouées,
- la terminaison de la connexion.

Avec le même formalisme, nous avons généré des programmes de plus de 800 lignes. La génération prend moins de 10 secondes sur une station de travail.

6. Conclusions

Tout système "intelligent" apparaît assez mécanique quand on en connaît la structure interne, et c'est le cas des logiciels que nous avons développés dans le cadre du projet MMS. Cependant, il est utile de comparer notre approche avec l'approche classique de nos "clients" de l'industrie.

D'abord, nos collaborateurs ont été très surpris de ce que nous avons pu accomplir avec des ressources limitées dans un laps de temps très court car le développement des outils que nous proposions leur semblait à priori impossible. Ils ont été ensuite conquis par ces outils - même dans leur état de prototypes - et, depuis la fin de notre implication, ces outils ont été adaptés (et adoptés) pour d'autres projets. Un aspect particulièrement intéressant de notre approche a été la généricité. Même si nous avions travaillé à générer des tests pour un protocole en particulier: MMS, les moteurs d'inférence de ces outils peuvent être utilisés pour d'autres systèmes en utilisant des bases de connaissances différentes et ces bases peuvent être obtenues automatiquement par compilation de définitions C appropriées. Par contre, avec des outils de programmation classiques, le changement de contexte nécessiterait la réécriture manuelle d'un ou de plusieurs programmes de test adaptés à chaque nouvelle fonction. Si la séparation entre les moteurs d'inférence et la base de connaissance explique, en partie, notre succès; il ne faut pas oublier le rôle primordial joué par Prolog, un langage particulièrement adapté au traitement symbolique et à la codification de connaissances.

Un deuxième résultat intéressant (et surprenant) a été l'efficacité des approches IA. Dans un cadre académique, il est rare que l'on ait de gros ensembles de connaissances sur lesquels faire des inférences. Dans ce projet, avec Prolog nous avons stocké et utilisé la totalité des définitions utilisées par les implanteurs d'IBM. Avec cette base de plus de 250 Koctets, tous nos outils ont fonctionné en temps réel (résultats produits en quelques secondes). Cependant, la rapidité du traitement est probablement du au fait que les inférences sont relativement directes, sans trop de recherches heuristiques ni de retours en arrière -- en contraste avec les applications typiques de l'IA.

Concernant l'utilisation de Prolog comme langage d'implantation, nous avons constaté que la difficulté principale, dans un projet d'implantation comme le nôtre, semble la compréhension des spécifications et la conception générale de la structure de l'implantation. Étant donné que la spécification était décrite principalement en langage naturel, ce travail était essentiellement manuel. L'utilisation du langage Prolog n'offre pas d'avantages aussi nets que dans le cadre précédent. Il est vrai que le contrôle dynamique des types est facilité par l'emploi du mécanisme d'unification et que la concision du

langage rend le programme lisible; mais en général, nous avons constaté que sa structure était très similaire à celle du programme correspondant écrit en C par l'équipe d'IBM.

Comme on pouvait s'y attendre, l'efficacité d'exécution du programme MMS en Prolog était largement inférieur à celle de l'implantation réalisée par IBM en C. Si ceci diminue l'attrait de Prolog pour la réalisation du produit livré; dans d'autres contextes, ce ne serait pas un handicap majeur, par exemple: pour tester la conformité d'une implantation avec la spécification du protocole. Dans ce cas, le programme en Prolog serait principalement destiné à la validation des spécifications ou à l'analyse de résultats de tests exécutés sur d'autres implantations du protocole.

La conclusion générale est que, pour des applications qui impliquent du traitement symbolique et des connaissances, les outils et les approches de l'intelligence artificielle sont non seulement utiles mais assez efficaces et robustes pour être utilisées dans des contextes industriels.

Remerciements

Le projet a été financé par le Laboratoire d'IBM Canada à Toronto et les travaux ont été réalisés à l'Université de Montréal en collaboration avec le Centre Informatique de Recherche de Montréal (CRIM). Julian Lebensold, du CRIM, a servi comme administrateur pour le projet.

Nous tenons à remercier tous le personnel du Laboratoire d'IBM qui a participé à cette expérience, en particulier: Gordon Lee qui a amorcé le projet et Tony Rego qui l'a maintenu, Sal Vella, Ming Wu et Ken Lee qui étaient nos collaborateurs immédiats. Nous remerçions aussi Paul Cheung, Joanne Downey, Alvin Fung, Payman Hodare et Simon Ng qui nous ont éclairé sur les mystères de MMS.

Références

1. Bochmann, G.v. "Protocol specification for OSI." Computer Networks and ISDN Systems, 18 p. 167-184, avril 1990.

2. Bochmann, G.v. "Usage of protocol development tools: the results of a survey." (invited paper), 7-th IFIP Symposium on Protocol Specification, Testing and Verification, Zurich, May 1987, pp.139-161., 1987.

3. Bochmann, G.v., S. Desmarais, P. Gamache, B. Lefebvre, et al. "Documentation of program packages." Project report D6, IBM(Toronto)-CRIM-Université de Montréal, Août 1990a.

4. Bochmann, G.v., S. Desmarais, P. Gamache, B. Lefebvre, et al. "Simulation and testing tools for the MAP MMS protocol." Project report D5, IBM (Toronto)-CRIM-Université de Montréal, Août 1990b.

5. Bratko, I. "Prolog programming for artificial intelligence." Reading, MA, Addison-Wesley, 1986.

6. Clocksin, W. and C. Mellish. "Programming in Prolog (3rd. Ed)." New York, Springer-Verlag, 1987.

7. Feigenbaum, E. "The Rise of the Expert Company." Texas Intruments, 1988.

8. Gamache, P. "Générateur intelligent de tests adaptés au domaine des protocoles de communication." MSc, Université de Montréal, 1991.

9. GM Corp. "Application Interface Model and Specification Requirements: MAP_Attachement 1 to MAP_Appendix 7." General Motors Corporation, June 1988.

10. GM Corp. "MMS Application Interface Specification (API): MAP_Attachement 6 to MAP_Appendix 7:." General Motors Corporation, June 1988.

11. ISO. "Manufacturing Message Specification - Service Definition." Draft Intern'l Standard Nº ISO/DIS 9506/1, ISO/TC 184, 1985.

12. ISO. "Manufacturing Message Specification - Protocol Specification." Draft Intn'l Standard Nº ISO/DIS 9506/2, ISO Technical Committee, December 1987.

13. ISO. "Information Processing - Open systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)." Nº IS 8824, International Standards Organisation, May 1987.

14. Knightson, K.G., T. Knowles and J. Larmouth. "Standards for Open Systems Interconnection." McGraw-Hill, 1988.

15. Pereira, F.C.N. and S.M. Shieber. "Prolog and Natural Language Analysis." Center for the Study of Language and Information Lecture Notes, Ventura Hall, Stanford, CA 94305, CLSI/Stanford, 1987.

16. Sterling, L. and E. Shapiro. "The art of Prolog: advanced programming techniques." Cambridge, Mass., MIT Press, 1986.

17. Vaucher, J. "A Record Package for Prolog." Soft. Pract. & Exp., 19 (8), p. 801-807, 1989.