Université Catholique de Louvain

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique

Étude de l'interopérabilité de deux langages de programmation basée sur la machine virtuelle de Java

Promoteur : Baudouin Le Charlier
Conseiller : Gustavo Ospina

Mémoire présenter en vue de l'obtention du Grade de diplômé d'études spécialisées en sciences appliquées par Frédéric Minne

Louvain-la-Neuve
Année académique 2002-2003

Table des matières

Chapitre 1

Interopérabilité de deux langages de programmation

1.1 Motivations

1.1.1 Pourquoi vouloir faire interagir deux ou plusieurs langages de programmations ?

Il existe aujourd'hui un très grand nombre de langages de programmation plus ou moins spécialisés et adaptés pour résoudre différentes classes de problèmes. Certains langages, comme COBOL, sont plus indiqués pour les problèmes de gestion et de finance, d'autres, comme FORTRAN ou ADA, pour les logiciels scientifiques et rigoureux, d'autres encore, comme Delphi ou Visual Basic, sont plus orientés vers le développement d'applicatiosn de bureautique, d'autres, comme C, sont des langages adaptés pour la programmation de fonctions de bas niveaux, d'autres, enfin, comme C# ou Java, permettent le développement d'applications réseau.

Devant la grande popularité d'internet et des applications distribuées via les browsers [1], il peut être très intéressant de pouvoir intégrer d'anciennes applications écrites dans des langages différents au sein d'une même application basée sur la technologie d'internet.

Une autre raison pour s'intéresser à l'interopérabilté est l'idée que, puisque de nombreux langages sont conçus pour des domaines assez spécialisés, il pourrait être utile d'implémenter les différentes parties d'une application dans le langage le plus adéquat et de faire ensuite interagir ces différentes parties entre elles.

1.1.2 Interopérabilité entre Java et Prolog

Dans le cadre de cet article, nous allons nous intéresser au cas particulier de l'interopérabilité entre le langage Java et le langage Prolog. Mais pourquoi nous intéresser à ces deux langages en particulier ? Pour répondre à cette question,

[1] Par exemple des applications pour l'intranet et les groupwares

nous allons examiner les différentes possibilités offertes par ces deux langages et voir les avantages qu'il peut y avoir à les utiliser ensemble.

Pourquoi Prolog ?

Le langage Prolog est un langage logique, de haut niveau, utilisé pour les applications d'intelligence artificielle telle que les systèmes experts, le traitement de la langue naturelle, pour les bases de données relationnelles, le datamining, et bien d'autres applications faisant intervenir des problèmes de parcours d'arbres de résolution.

Il permet de programmer de manière simple des problèmes qui peuvent être représentés sous formes d'un ensembles de règles et de faits. En Prolog, il n'est pas nécessaires, comme c'est le cas dans les langages impératifs, de construire la résolution explicite d'un problème. Il suffit simplement de modéliser une base de connaissances [2] qui seront utilisés par un moteur d'inférence pour construire les solutions à un problème donné (Voir figure 1.1).

Malgré tous ses avantages lorsqu'il faut décrire des systèmes complexes en terme de règles d'inférences et d'axiomes logiques, Prolog est devenu aujourd'hui, pour diverses raisons que nous allons quelque peu détailler, peu adéquat pour développer des applications orientées utilisateurs.

En effet, l'intégration d'un interpréteur Prolog dans un serveur web ou un browser est difficile à cause du manque d'interfaces efficaces entre Prolog et les autres langages et de l'implémentation native de l'interpréteur. Ensuite, Prolog ne fournit pas de fonctions prédéfinies pour construire une interface utilisateur graphique, ce qui rend difficile la conception de programmes interactifs "user-friendly". Prolog est également très peu adapté pour la réalisation d'application réseau.

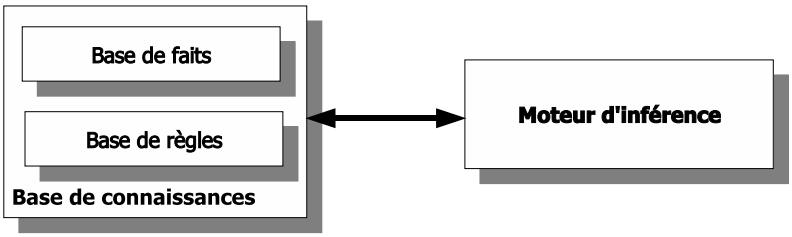

FIG. 1.1 – Architecture de Prolog

Pourquoi Java ?

Java est un langage orienté objet devenu très populaire en raison de sa simplicité de mise en oeuvre, de sa portabilité et de son intégration simple dans des applications orientées Web. Java intègre la plupart des grandes fonctions des lagages modernes : multithreading, interfaces graphiques, programmation réseau, garbage collection... Le développement d'application en Java est grandement facilitée par l'encapsulation des appels à des fonctions de bas niveaux dans des classes telles que le package SWING, AWT, java.net...

[2] Un ensemble de règles et de faits.

Java est résolument orienté réseau et possède de nombreuses extension pour l'utilisation d'objet répartis comme JavaRMI (pour les appels de méthodes à distance), JavaORB (un implémentation de CORBA) ou encore JavaRPC, ainsi que pour l'intégration de la technologie XML. Java fournit également, via JDBC, des méthodes d'accès à des bases de données relationnelles.

En ce qui concerne l'interopérabilité avec d'autres langages, Java fournit une interface pour l'invocation de méthodes natives écrites dans d'autres langages tels que C ou C++.

Applications

L'interopérabilité entre Java et Prolog va permettre de réaliser des applications intégrant de l'intelligence artificielle et possédant des interfaces plus performantes avec l'utilisateur ou permettant un déploiement sur un réseau informatique. On pourrait, par exemple, réaliser des outils de recherches sur le Web plus efficaces, des systèmes experts accessibles en ligne, des logiciels de construction et de vérification de programmes, des jeux intégrant une intelligence artificielle évoluée, des applications orientées agents,...

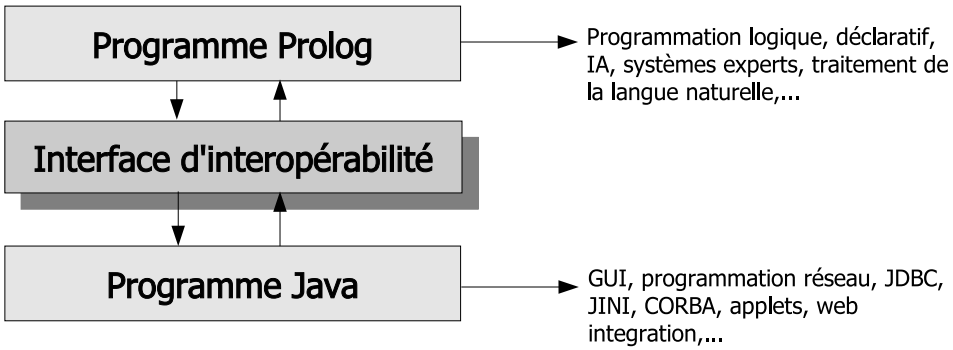

FIG. 1.2 – Interopérabilité Java Prolog

1.2 Mécanismes pour réaliser l'interopérabilité entre Java et Prolog

Nous allons maintenant nous intéresser aux différentes possibilités qui nous sont offertes pour réaliser l'interopérabilité entre Java et Prolog.

1.2.1 Mécanismes basées sur la machine virtuelle de Java

Il existe (au moins) deux mécanismes pour réaliser l'interopérabilté basées sur la machine virtuelle de Java (ou JVM) :

- L'appel de fonctions natives via l'utilisation de l'interface native Java[3],
- La compilation du langage à interfacer en fichier exécutable par la JVM.

[3]ou JNI

Ce sont ces deux mécanismes qui vont nous intéresser.

Méthodes Natives

La machine virtuelle de Java implémente un mécanisme permettant d'appeler des méthodes natives depuis des programmes Java [Eng99] [TL99] [Ven98]. L'appel de méthodes natives en Java est facilité par l'utilisation de la "Java Native Interface" [Gor98] qui fournit des primitives [4] pour interfacer Java avec des programmes natifs écrits en C/C++ (voir figure 1.3).

Pour utiliser cette interface pour permettre l'interopérabilité de Java et de Prolog, il suffit de se procurer un interpréteur Prolog écrit en C fournissant une interface avec des programmes écrits en C [5] et d'utiliser l'interface native Java pour pouvoir, d'un côté, récupérer le résultat de l'interprétation du programme Prolog dans un programme Java, et, d'un autre côté, appeler des méthodes d'un programme Java [6] depuis le programme Prolog. L'interfaçage entre les deux étant, comme nous l'avons dit, gérer par l'interface JNI.

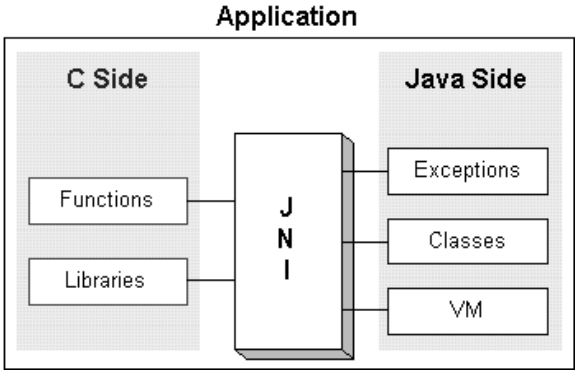

FIG. 1.3 – L'interface native Java

Compilation des langages en bytecode Java

L'idée est de transformer un programme Prolog de manières à ce qu'il soit exécutables par la JVM . Il existe deux possibilités pour cela : la compilation du programme Prolog en un programme séquentiel [Eng99] ou son interprétation à l'aide d'une machine virtuelle Prolog écrite en Java [AK91] (Voir figure 1.4).

La compilation des deux langages pour la machine virtuelle Java a l'avantage de permettre une interopérabilité directe dans les deux sens, donc plus efficace.

[4] Par exemple la conversion de données, la gestion de l'environnement, la création/invocation de la machine virtuelle Java,...

[5] De nombreux interpréteurs Prolog fournissent une telle interface, si aucune interface de ce type n'est présente, il est toujours possible d'en créer une "à la main".

[6] Par exemple des méthodes pour créer une interface graphique

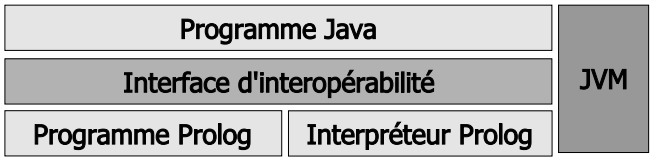

FIG. 1.4 – Interopérabilité par compilation des deux langages pour la JVM

1.2.2 Autres mécanismes

Il existe d'autres mécanismes pour réaliser l'interopérabilité qui ne sont pas basées sur la machine virtuelle. Nous allons ici dire quelques mots à leur propos. Une étude approfondie de ces mécanismes sort du cadre de cet article.

Utilisation de XML

L'idée est d'échanger les informations entre un programme Prolog et une application Java en utilisant le langage XML développé par le W3C. Cette méthode est déjà très largement utilisée pour l'interopérabilité entre systèmes d'information hétérogène. Elle est assez facile à mettre en oeuvre puisqu'il s'uffit que chaque intervenant possède un parseur XML. Les requêtes à l'un d'entre seront dès lors "emballés" dans un message XML qui lui sera transmit. Celui-ci déballera la requête, l'évaluera et emballera la réponse dans un autre message XML qui sera envoyé au premier intervenant. L'interopérabilité se fait "par fichier interposé".

CORBA

L'idée ici d'utiliser un interpréteur Prolog basé sur la technologie d'objets partagés CORBA. Requêtes et réponses entre l'application Java et le programme Prolog serait alors envoyées en utilisant le broker de CORBA. L'interface entre les deux langages sera décrite explicitement au moyen d'un langage de description d'interface (ou IDL).

Utilisation d'une architecture Client-Serveur

L'idée ici serait de créer un "serveur" Prolog qui recevra les requêtes du "client" Java et lui fournira les réponses. Pour réaliser l'interopérabilité dans les deux sens, il faudrait également posséder un "serveur" Java qui aurait comme "client" le "serveur" Prolog. Il faudra de plus créer le protocole qui permettra l'échange des requêtes et des réponses. L'interopérabilté sera ici basée sur l'utilisation des Sockets et le passage de message entre le programme "client" et le "serveur" Prolog.

Les deux cas précédents sont en fait des cas particuliers de cette architecture client serveur.

1.3 Organisation de cet article

1.3.1 Dans la suite...

Dans la suite de cet article, nous allons définir une interface d'interopérabilité générique entre Java et Prolog. Cette interface devra être totalement transparente pour le programmeur et lui cacher tous les détails de l'implémentation de Prolog sous-jacente.

Avertissement : cette interface est encore provisoire et ne résout pas tous les problèmes que pose le mélange de deux langages aux paradigmes très différents. Un des objectifs de la suite de mon travail va être de compléter ces interface et de justifier de manière formelle les différentes fonctions de son API avant de l'adapter à un mécanisme d'interopérabilité déjà existant.

1.3.2 Notations et abréviations

Voici quelques abréviations que nous allons utiliser fréquemment dans la suite :

Abréviation	Signification
API	Application Programming Interface
JVM	Java Virtual Machine
WAM	Warren's Abstract Machine
JNI	Java Native Interface

Chapitre 2

Spécification d'une interface d'interopérabilité générique entre Java et Prolog.

Dans ce chapitre, nous allons nous intéresser à la définition et à la spécification d'une interface permettant l'interopérabilité entre Java et Prolog, de manière générale et sans aborder les détails d'implémentation. L'idée est d'obtenir une interface qui pourra être adaptée à un mécanisme d'interopérabilité particulier selon les besoins de l'application et la disponibilité de ces mécanismes.

2.1 Définition du problème

Le problème est le suivant : nous allons devoir faire communiquer un langage orienté objet et fortement typé, Java, avec un langage logique, déclaratif et non typé, Prolog. Cette communication devra être possible dans les deux sens, un programme Java doit pouvoir invoquer l'évaluation d'une requête Prolog et, inversement, un programme Prolog doit pouvoir être capable d'invoquer des méthodes écrites en Java.

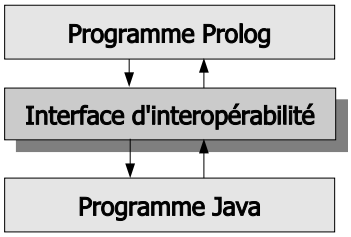

Fig. 2.1 – Interface

Les différents points que nous allons examiner dans la suite sont :
- La conversion de types de données entre Java, typé, et Prolog, non typé ;
- L'invocation d'une requête Prolog depuis Java ;

– L'invocation de méthodes Java depuis Prolog ;
  L'appel statique de méthodes Java [1] ;
  La création d'objets et l'invocation de méthodes sur ces objets ;
– La gestion des exceptions générées par Java depuis les programmes Prolog.

2.2 Conversion de types

Le langage Java est un langage orienté-objet fortement typé. A contrario, Prolog est un langage logique non typé. Pour permettre l'interopérabilité entre ces deux il va nous falloir définir une équivalence entre type de données Java et "type" de terme Prolog. La table 2.1 indique cette équivalence.

Prolog	conversion	Java
atom	<—->	java.lang.String
integer	<—->	java.lang.Integer
float	<—->	java.lang.Float
boolean	<—->	java.lang.Boolean
chars	<—->	java.lang.String ou java.lang.Char
string	<—->	java.lang.String ou java.lang.Char[]
term	<—->	java.lang.Object
list	<—->	java.lang.Object[]

TAB. 2.1 – Conversion de types

Le choix de la conversion des termes représentant des nombres en Prolog vers les objets Java correspondants, plutôt que vers les types primitifs, est justifié par le désir d'avoir une interface d'interopérabilité aussi simple que possible. En effet, tous les arguments passés à un programme Java et tous les résultats retournés par celui-ci seront de type *java.lang.Object*.

2.3 Prolog depuis Java

Nous allons spécifier l'interface permettant l'appel à l'interpréteur Prolog depuis un programme Java. Cette interface va se diviser en deux parties :

– Une partie méta-programmation permettant la gestion des théories, c'est-à-dire :
  le chargement d'une théorie ;
  l'ajout de règles ou de faits ;
  la révocation de règles ou de faits ;
– une partie interopérabilité permettant l'évaluation de requête par le moteur Prolog.

[1] C'est-à-dire ne modifiant l'état de la classe contenant la méthode invoquée. Nous reviendrons plus en détail sur ce point dans la suite.

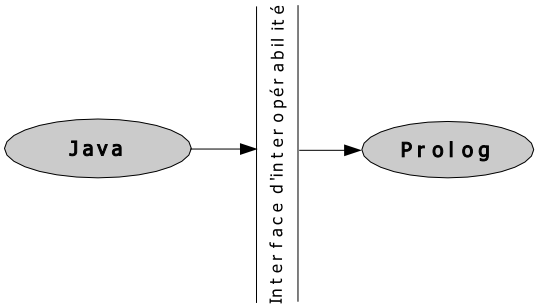

FIG. 2.2 – Interopérabilité : appel de Prolog depuis Java (J2P API)

2.3.1 Interface de gestion des théories (méta-programmation)

Elle définit trois méthodes et un objet Theory. L'objet Theory est une abstraction pour une théorie [2] Prolog. Nous ne définissons pas cet objet de manière plus précise pour le moment mais nous reviendrons dessus au chapitre suivant. En pratique, il pourra s'agir d'une chaîne de caractère, d'un fichier, d'un stream de données,...

Les trois méthodes sont :

public void consult (Theory theory) throws InvalidTheoryException, TheoryNotFoundException
Permet de charger un programme (une théorie) dans l'interpréteur Prolog.
In : Theory theory : la théorie à charger ;
Out : void ;
Pre : TRUE ;
Post : la théorie theory a été chargée dans l'interpréteur Prolog ;

public void assert (String term) throws InvalidTermException
Ajoute une règle ou un fait à la théorie chargée dans l'interpréteur Prolog. **In :** String term : le terme à ajouter au programme Prolog courant ;
Out : void ;
Pre : TRUE ;
Post : la règle ou le fait donné par le terme term à été ajouté à la théorie ;

public void revoke (String term) throws InvalidTermException, TermNotFoundException
Supprime une règle ou fait du programme Prolog courant. **In :** String term : la règle ou le fait à révoquer ;
Out : void ;
Pre : la règle ou le fait à suppirimer doit faire partie du programme chargé dans l'interpréteur Prolog ;
Post : la règle ou le fait donné par term à été retiiré de la théorie ;

[2] un ensemble de règles et de faits constituant le programme Prolog

Remarque : Pour l'instant, nous n'avons pas de moyen pour vérifier que la théorie obtenue en ajoutant ou révoquant des règles/faits au moyen des deux dernières méthodes reste cohérente.

2.3.2 Interface d'interrogation de l'interpréteur.

Nous allons definir trois méthodes pour l'interrogation de l'interpréteur Prolog :

public Object find (String query) throws InvalidQueryException, NoAnswerException
In : String query : la requête à adresser à l'interpréteur Prolog ;
Out : Objet : le premier résultat de la requête ;
Pre : TRUE ;
Post : la première réponse à la requête posée à été retournée ;

public Object findNext () throws NoMoreAnswerException
In : void ;
Out : Objet : le résultat suivant de la requête actuellement traitée par l'interpréteur Prolog ;
Pre : une requête a été demandée à l'interpréteur Prolog ;
Post : la réponse suivante à la requête posée à été retournée ;

public Object[] findAll (String query) throws InvalidQueryException, NoAnswerException
In : String query : la requête à adresser à l'interpréteur Prolog ;
Out : Objet[] : tous les résultat de la requête ;
Pre : TRUE ;
Post : toutes les réponses à la requête ont été retournées ;

2.4 Java depuis Prolog

L'interface entre Prolog et Java va être construite de manière incrémentale. Dans un premier temps, nous allons nous intéresser uniquement aux appels statiques de méthodes Java depuis un programme Prolog. Ensuite, nous aborderons le problème de la création d'objets Java et de l'invocation de méthodes sur ces objets. Nous verrons que ce dernier point va être un peu plus problématique puisqu'il va venir introduire la notion d'état dans Prolog alors que ce dernier est un langage dont les variables, liées une fois pour toute, sont sans état. Cela va introduire un mélange entre paradigme de la programmation logique et paradigme de la programmation orientée-objet qui aura de nombreuses applications, particulièrement dans la programmation orientée-agents.

L'interface d'interopérabilité de Prolog vers Java va être constituée de deux parties (figure2.3) :

- une interface côté Prolog qui spécifie les différents prédicats qui vont permettre la création d'objet, l'invocation de méthodes et l'accès aux variables ;

– une interface côté Java qui spécifie les différentes méthodes Java qui seront invoquées lors de l'appels des prédicats de l'inteface Prolog.

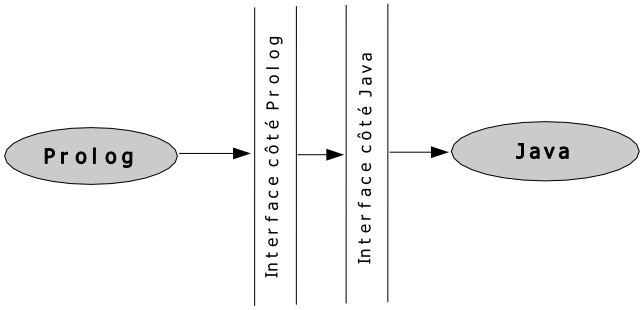

FIG. 2.3 – Interopérabilité : appel de Java depuis Prolog (P2J API)

C'est lors du passage de paramètres entre ces deux interfaces que la conversion des données entre Java et Prolog devra être effectuées.

2.4.1 Appels statiques de méthodes Java

Dans un premier temps, nous allons nous intéresser uniquement à l'invocation de méthodes statiques et à l'accès à des variables finales statiques. Ces accès et invocations vont nous permettre de définir tous les mécanismes nécessaires à l'utilisation de prédicats prédéfinis écrits en Java tels que, par exemple, l'arithmétique ou les entrées sorties. Nous verrons plus loin comment enrichir cette interface en lui ajoutant les appels non statiques et la création d'objets Java.

Pour spécifier l'interface, nous allons commencer par définir les prédicats Prolog avant de nous intéresser aux méthodes Java.

Interface côté Prolog

En Java, il existe deux types de méthodes, celle qui ne retourne aucun résultat, les méthodes "void", et celle qui en retourne un. Nous allons donc définir deux prédicats Prolog pour les appels de méthodes statiques, un pour chaque type de méthode :

java_static_void (+ClassFullName, +MethodName, +ArgsList)
+ClassFullName : le nom complet de la classe sur laquelle on veut invoquer la méthode au format Assembleur Java, c'est-à-dire avec des '/' au lieu des '.' et au format 'classpath/classname'
+MethodName : le nom de la méthode void à invoquer
+ArgsList : une liste contenant les arguments de la méthodes
Ce prédicat permet l'invocation d'une méthode statique "void" Java.

java_static (+ClassFullName, +MethodName, +ArgList, -MethodResult).
+ClassFullName : le nom complet de la classe sur laquelle on veut invoquer la méthode au format Assembleur Java, c'est-à-dire avec des '/' au lieu des '.' et au format 'classpath/classname'
+MethodName : le nom de la méthode void à invoquer
+ArgsList : une liste contenant les argumants de la méthodes
-MethodResult : contient le résultat de l'invocation de la méthodes
Ce prédicat permet l'invocation d'une méthode statique Java.

Nous avons également besoin d'un Prédicat pour accéder à la valeur d'un champ final static d'une classe Java :

java_static_getfinalfield (+ClassFullName, +FieldName, -FieldValue).
+ClassFullName : le nom complet de la classe sur laquelle on veut invoquer la méthode au format Assembleur Java, c'est-à-dire avec des '/' au lieu des '.' et au format 'classpath/classname'
+FieldName : nom de la variable finale et statique Java dont on veut obtenir la valeur
-FieldValue : contient la valeur de la variable
Ce prédicat permet l'accès à la valeur d'une variable finale statique Java.

Interface côté Java

Derrière ces prédicats se cache une interface Java qui comprend les méthodes suivantes [3] :

public void invokeStaticVoid (String classFullName, String methodName, Object[] argsList) throws ClassNotFoundException, MethodNotFoundException
In : String ClassFullName : le nom complet (fully qualified name) de la classe contenant la méthode ;
String MethodName : le nom de la méthode à invoquer ;
ArgsList : tableau d'objets contenant les arguments à passer à la méthode ;
Out : void ;
Pre : TRUE ;
Post : la méthode MethodName de la classe ClassFullName à été invoquée ;

public Object invokeStatic (String classFullName, String methodName, Object[] args) throws ClassNotFoundException, MethodNotFoundException
In : String ClassFullName : le nom complet (fully qualified name) de la classe contenant la méthode ;
String MethodName : le nom de la méthode à invoquer ;
ArgsList : tableau d'objets contenant les arguments à passer à la méthode ;
Out : Object : résultat de la méthode invoquée ;
Pre : TRUE ;
Post : la méthode MethodName de la classe ClassFullName à été invoquée et

[3] Nous compléterons cette interface au fur et à mesure que nous lui ajouterons des fonctionnalités

son résultat à été placé dans l'Object retourné ;

public Object getFinalStaticField (String classFullName, String fieldName) throws ClassNotFoundException, FieldNotFoundException
In : String ClassFullName : le nom complet (fully qualified name) de la classe contenant le champ ;
String FielddName : le nom du champ final statique dont on veut récupérer la valeur ;
Out : Object : la valeur du champ ;
Pre : TRUE ;
Post : la valeur du champ final statique FieldName de la classe ClassFullName a été retournée ;

Prédicats built-in

Les différents prédicats et les différentes méthodes que nous venons de définir fournissent tous les mécanisme nécessaires pour l'appel de prédicats built-in "écrits" en Java. Il suffit de définir les fonctions de ces prédicats dans des méthodes statiques, les classes joueront alors le rôle de bibliothèques de fonctions.

Ces fonctions pourront dès lors être appelées depuis Prolog en utilisant les prédicats que nous avons définis, ceux-ci invoqueront à leur tour les méthodes Java d'appels statiques, vues ci-dessus, qui se chargeront alors d'invoquer les fonctions correspondantes [4]

Pour illustrer notre propos, considérons l'exemple suivant, on veut :

- définir un prédicat pour l'addition d'entiers en Prolog ;
- utiliser pour cela l'addition en Java
- pour simplifier, nous ne traitons pas les exceptions [5]

Supposons que l'addition ait été implémentée en Java de la manière suivante :

```
package prolog.lib.math;

public class intMath
{
   public static Integer add(Object x, Object y)
   {
      int i = (Integer) X.intValue();
      int j = (Integer) Y.intValue();
      return new Integer(i + j);
   }
```

[4] Remarquons que, bien que nous avons supposer que ces fonctions seraient implémentées en Java, rien n'interdit qu'elles soient écrite dans un autre langage et accédées via l'interface native Java ou un autre procédé comme les sockets.

[5] Nous reviendrons sur ce point plus loin.

```
   /* reste du code */
}
```

Du côté Prolog, nous définissons le prédicat *+/3* de la manière suivante :

```
+(X, Y, Z) :- java_static('prolog/lib/math/intMath', 'add', [X, Y], Z).
```

L'interface côté Java sera invoquée de la manière suivante :

```
Object Z = invokeStatic("prolog/lib/math/intMath",  "add", {X, Y});
```

Remarque : La création de l'interface Java à partir de la définition d'un prédicat externe dans un programme Prolog peut être automatisée.

2.4.2 Création d'objets et appels non statiques

Nous allons maintenant nous intéresser à la création d'objets Java par des prédicats Prolog et à l'invocation de méthodes sur ces objets. Dans le cas précédent, les appels aux méthodes et champs ne modifiaient pas l'état de la classe concernée. Cela ne va plus être le cas et nous allons être confronté à un problème de sémantique. En effet, en ce qui concerne Java, les objets ont un état et peuvent être modifier, ils sont mutable. En Prolog, par contre, les variables sont sans état et, une fois liées, elles ne peuvent plus être modifiées, ce sont des variables non mutables.

Pour préserver la sémantique des variables Prolog, l'idée est d'invoquer les méthodes ou de modifier les champs non sur l'objet lui-même, mais sur une copie de ce dernier qui sera retournée par le prédicat appelant sous la forme d'une nouvelle variable Prolog. Les figures 2.4 et 2.5 illustrent ce mécanisme [6].

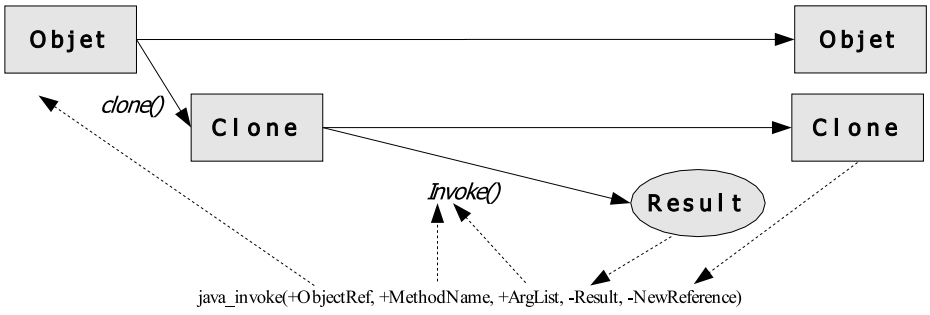

FIG. 2.4 – Création d'objets et invocation de méthodes

Nous pouvons maintenant spécifier et compléter notre interface en lui ajoutant ces appels non statiques.

[6] La spécification des prédicats concernées est donnée plus loin.

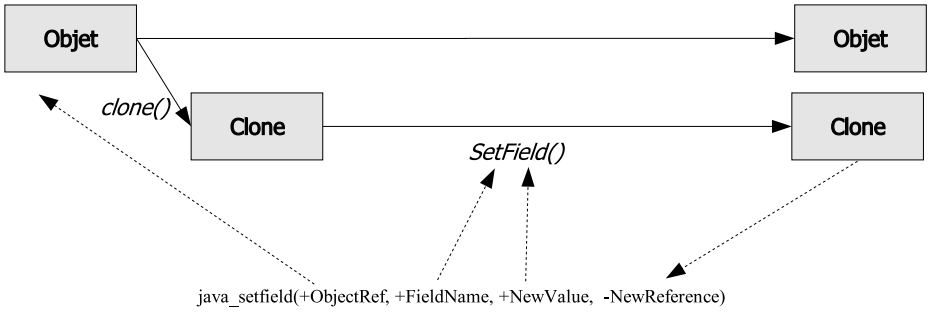

FIG. 2.5 – Modification de variable

Interface côté Prolog

java_object (+ClassFullName, +ArgList, -ObjectReference)
+ClassFullName : le nom complet de la calsse dont on veut créer une instance ;
+ArgList : liste des arguments pour l'initialisation de la classe ;
-ObjectReference : référence à l'objet créé.

java_static_getfield (+ObjectReference, +FieldName, -FieldValue).
+ObjectReference : référence à l'objet contenant le champ demandé ;
+FieldName : nom du champ statique demandé ;
-FieldValue : valeur du champ ;

java_static_setfield (+ObjectReference, +FieldName, +NewValue, -NewReference).
+ObjectReference : référence à l'objet contenant le cahmp demandé ;
+FieldName : nom du champ statique demandé ;
+NewValue : nouvelle valeur du champ ;
-NewReference : référence à l'objet modifié ;

java_getfield (+ObjectReference, +FieldName, -FieldValue).
+ObjectReference : référence à l'objet contenant le champ demandé ;
+FieldName : nom du champ demandé ;
-FieldValue : valeur du champ ;

java_setfield (+ObjectReference, +FieldName, +NewValue, -NewReference).
+ObjectReference : référence à l'objet contenant le champ demandé ;
+FieldName : nom du champ demandé ;
+NewValue : nouvelle valeur du champ ;
-NewReference : référence à l'objet modifié ;

java_invoke_void (+ObjectReference, +MethodName, +ArgList, -NewReference).
+ObjectReference : référence à l'objet contenant la méthode void à invoquer ;
+MethodName : nom de la méthode voud à invoquer ;
+ArgList : liste des argument à passer à la méthode void ;
-NewReference : référence à l'objet après l'invocation

java_invoke (+ObjectReference, +MethodName, +ArgList, -Result, -NewReference).
+ObjectReference : référence à l'objet contenant la méthode void à invoquer ;
+MethodName : nom de la méthode voud à invoquer ;
+ArgList : liste des argument à passer à la méthode void ;
-Result : variable contenant le résultat de l'invocation ;
-NewReference : référence à l'objet après l'invocation

Interface côté Java

public Object createInstance (String classFullName, Object[] ArgList) throws ClassNotFoundException, BadArgumentException **In :** String ClassFullName : le nom complet (fully qualified name) de la classe ;
Object[] ArgList : arguments d'intialisation de l'objets ;
Out : Object l'instance créée ;
Pre : TRUE ;
Post : une nouvelle instance de classFullName a été créée et retournée ;

public Object getStaticField (Object reference, String fieldName) throws NullPointerException, FieldNotFoundException
In : Object reference : référence à l'objet contenant le champ statique dont on veut obtenir la valeur ;
String fieldName : le nom du champ statique dont on veut obtenir la valeur ;
Out : Object contenant la valeur du champ statique ;
Pre : TRUE ;
Post : la valeur du champ statique fieldName de la classe classFullName a été retournée ;

public void setStaticField (Object reference, String fieldName, Object fieldValue) throws NullPointerException, FieldNotFoundException
In : Object reference : référence à l'objet contenant le champ statique à modifier ;
String fieldName : le nom du champ statique à modifier ;
Object fieldValue : nouvelle valeur du champ statique ;
Out : void ;
Pre : TRUE ;
Post : la valeur du champ statique fielddName de la classe ClassFullName à été modifiée et a la valeur fieldValue ;

public Object getField (Object reference, String fieldName) throws NullPointerException, FieldNotFoundException
In : Object reference : référence à l'objet contenant le champ à modifier ;

String fieldName : le nom du champ ;
Out : Object valeur du champ fieldName ;
Pre : TRUE ;
Post : la valeur du champ statique fielddName de la classe ClassFullName à été retournée ;

public void setField (Object reference, String fieldName, Object fieldValue) throws NullPointerException, FieldNotFoundException
In : Object reference : référence à l'objet contenant le champ à modifier ;
String fieldName : le nom du champ à modifier ;
fieldValue : nouvelle valeur du champ ;
Out : void ;
Pre : TRUE ;
Post : la valeur du champ statique fielddName de la classe ClassFullName à été modifiée et a la valeur fieldValue ;

public void invokeVoid (Object reference, String methodName, Object[] argsList) throws ClassNotFoundException, MethodNotFoundException
In : Object reference : référence à l'objet contenant la méthode ;
String MethodName : le nom de la méthode à invoquer ;
Object[] ArgsList : tableau d'objets contenant les arguments à passer à la méthode ;
Out : void ;
Pre : TRUE ;
Post : la méthode MethodName de l'instance reference à été invoquée ;

public Object invoke (Object reference, String methodName, Object[] argsList) throws ClassNotFoundException, MethodNotFoundException
In : Object reference : référence à l'objet contenant la méthode ;
String MethodName : le nom de la méthode à invoquer ;
Object[] ArgsList : tableau d'objets contenant les arguments à passer à la méthode ;
Out : Object : résultat de la méthode invoquée ;
Pre : TRUE ;
Post : la méthode MethodName de l'instance reference à été invoquée et son résultat à été placé dans l'Object retourné ;

2.5 Gestion des exceptions

Pour que notre interface soient complète, il nous faut maintenant définir, tant pour l'API Java vers Prolog (J2P) que pour l'interface Prolog vers Java (P2J), des mécanismes de traitement des exceptions.

Notre interface pour la gestion des exceptions va donc, comme les interfaces précédentes, contenir deux parties :

- une partie Prolog permettant de récupérer et de traiter les exceptions générées par Java ;

– une partie Java permettant de récupérer les excepions générées par un programme Prolog.

Le mécanisme que nous allons définir fera appel à deux piles d'exceptions : le JavaExceptionStack qui conteindra les exceptions générées par Java à l'attention de Prolog, et le PrologExceptionStack, qui contiendra les exceptions générées par Prolog. Pour accéder à ces exceptions, nous allons maintenant spécifier les prédicats et méthodes correspondants.

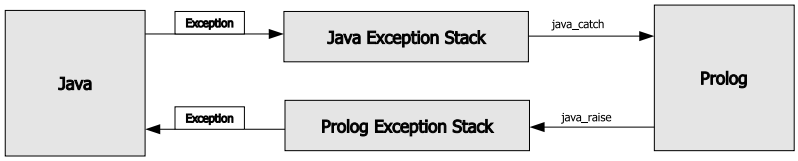

FIG. 2.6 – mécanisme de gestion des exceptions

2.5.1 Gestions des exceptions côté Prolog

Nous allons définir ici deux prédicats : java_raise/1 qui génère une exception à destination de Java depuis un programme Prolog et java_catch/1 qui capte une exception générée par Java.

Générer une exception depuis Prolog

java_raise (+Exception).
+Exception : Exception à destination de Java
Transmet l'exception Exception à Java

Derrière ce prédicat se cache la méthode Java suivante :

public void pushPrologException (Exception exception)
In : Exception exception : l'exception à placer sur la pile ;
Out : void ;
Pre : TRUE ;
Post : l'exception à été ajoutée sur la pile d'exceptions Prolog

Attraper une exception Java depuis Prolog

java_catch (-Exception).
-Exception : Exception générée par Java ;

Derrière ce prédicat se cache la méthode Java suivante :

public Exception popJavaException ()
In : void ;
Out : Exception l'exception située au sommet de la pile ;
Pre : TRUE ;
Post : l'exception à été retirée de la pile d'exceptions Java et transmise au programme Prolog

2.5.2 Gestions des exceptions côté Java

L'interface d'interopérabilité Java vers Prolog contient deux méthodes :

- une méthode pour attraper une exception générée par Prolog ;
- une méthode pour passer à Prolog un exception générée par Java.

La définition de ces deux méthodes est la suivante :

public Exception popPrologException ()
In : void ;
Out : Exception l'exception située au sommet de la pile ;
Pre : TRUE ;
Post : l'exception à été retirée de la pile d'exceptions Prolog et transmise au programme Java

public void pushJavaException (Exception exception)
In : Exception exception : l'exception à placer sur la pile ;
Out : void ;
Pre : TRUE ;
Post : l'exception à été ajoutée sur la pile d'exceptions Java

2.6 Récapitulation de l'interface d'interopérabilité de Prolog vers Java

Pour terminer cette article, voici une récapitulation des différentes méthodes et Prédicats définis par l'interface. Les tables 2.2 et 2.3 résume l'API que nous venons de définir

2.7 Ce qu'il reste à faire

Comme annoncé dans l'introduction, l'interface que nous venons de définir n'est encore que provisoire car elle ne règle pas encore tous les problèmes causés par l'interopérabilité entre Java et Prolog. Les questions encore en suspend sont, entre autres, :

Java	Rôle
consult(Theory theory)	charge une théorie dans la base de connaissance de l'interpréteur Prolog
assert(String term)	ajoute une règle/un fait à la théorie courante
revoke(String term)	enlève un fait/ une règle à la théorie courante
find(String query)	crée une requête Prolog et retourne la première réponse
findNext()	retourne la réponse suivante à le requête courante
findAll(String query)	crée une requête et retourne toutes les solutions
popPrologException()	capture une exception générée par Prolog
pushJavaException	génère une exception à l'attention de Prolog

TAB. 2.2 – Interface Java vers Prolog

Prolog	Java	Rôle
java_static_void/3	invokeStaticVoid	appel de méthode statique void
java_static/4	invokeStatic	appel de méthodes statiques
java_static_getfinalfield/3	getFinalStaticField	obtenir la valeur d'un champ final statique
java_object/3	createInstance	crée une instance de classe Java
java_static_getfield/3	getStaticField	retourne la valeur d'un champ statique
java_static_setfield/4	setStaticField	change la valeur d'un champ statique
java_getfield/3	getField	retourne la valeur d'un champ
java_setField/4	setfield	change la valeur d'un champ
java_invoke_void/4	invokeVoid	invoque une méthode void sur une instance
java_invoke/5	invoke()	invoque une méthode sur une instance
java_raise/1	pushPrologException	génère une exception Prolog
java_catch/1	popJavaException	attrape une exception Java

TAB. 2.3 – Interface Prolog vers Java

- le problème du déterminisme : il nous faudra définir un mécanisme pour pouvoir indiquer à un programme si un prédicat est déterministe ou non ;
  un prédicat est déterministe si il peut donner une réponse, échouer ou possède un nombre fini de réponse ;
  il est non déterministe si il possède un nombre infini de réponse ou s'il boucle ;
- le problème de la consistance d'une théorie après l'ajout, la révocation de règles ou de faits ;
- ajouter des exemples d'utilisation pour chaque méthode de l'interface ;
- définir des prédicats et méthodes pour la conversion des données entre Java et Prolog ;
- adapter notre API à des mécanismes déjà existant et implémenter une application sur base de cette API ;
- éventuellement, si le temps nous le permet, discuter plus en détail de l'impact de l'ajout d'éléments de paradigme orienté-objet au sein d'un langage logique et discuter de l'intéret d'un tel langage "hybride" ;
- ...

Table des figures

Liste des tableaux

Bibliographie

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine : a tutorial reconstruction*. The MIT Press, 1991.

[Eng99] Joshua Engel. *Programming for the Java Virtual Machine*. Addison-Wesley, 1999.

[Gor98] Rob Gordon. *JNI : the Java Native Interface*. Prentice Hall, 1998.

[TL99] Franck Yellin Tim Lindholm. *The Java Virtual Michine Specification (2nd Edition)*. Addison-Wesley, 1999.

[Ven98] Bill Venners. *Inside the Java Virtual Machine*. MacGraw Hill, 1998.