Premier délivrable du projet :
Application de la programmation par aspect au code mobile en Java.
Mise en œuvre à l'aide de techniques réflexives.

Java et la réflexion

Noury Bouraqadi-Saâdani
Noury.Bouraqadi@emn.fr
École des Mines de Nantes

Table des matières

1 Introduction à la réflexion **3**
1.1 Généralités . . . 3
1.1.1 Introspection vs. intercession . . . 3
1.1.2 Réflexion structurelle vs. comportementale . . . 4
1.1.3 Méta-niveaux et régression infinie . . . 4
1.2 Réflexion et objets . . . 6
1.2.1 Méta-objet . . . 6
1.2.2 Réification . . . 6
1.2.3 Lien méta . . . 6
1.2.4 Métaclasse . . . 7
1.2.5 MOP . . . 8

2 La réflexion dans Java **9**
2.1 Capacités réflexives propres à Java . . . 9
2.1.1 JDK 1.1 et 1.2 . . . 9
2.1.2 JDK 1.3 . . . 10
2.2 Principales applications des capacités réflexives de Java . . . 13
2.2.1 Serialisation . . . 13
2.2.2 Java Beans . . . 13
2.2.3 RMI . . . 14

3 Principales extensions réflexives de Java **16**
3.1 Extensions basées sur la transformation de code source . . . 17
3.1.1 Reflective Java . . . 17
3.1.2 OpenJava . . . 19
3.1.3 Proactive . . . 19
3.2 Extensions basées sur la transformation de byte-code . . . 22
3.2.1 Dalang . . . 22
3.2.2 Kava . . . 22

3.2.3 Javassist 25
3.3 Extensions basées sur la modification de la machine virtuelle 25
3.3.1 MetaXa 25
3.3.2 Guaraná 27

4 Comparaison des principales extensions réflexives de Java 28
4.1 Comparaison des modèles réflexifs 28
4.1.1 Quand a lieu le passage d'un niveau à l'autre? 29
4.1.2 Qui réalise le contrôle méta? 31
4.1.3 Quelle est l'arité du lien méta? 31
4.1.4 Quelles opérations provoquent le passage au méta-niveau? 33
4.1.5 Comment est arrêtée la régression du lien méta? 33
4.2 Comparaison des mises en œuvre 34
4.2.1 Comment sont introduits les intercepteurs? 35
4.2.2 Quand sont introduits les intercepteurs? 36
4.2.3 Quel est le degré de fiabilité d'interception? 37
4.2.4 Quelles sont les possibilités d'adaptabilité dynamique? 38
4.2.5 Quel est le degré de difficulté de l'implémentation? 38
4.2.6 Quel est le degré d'efficacité? 39

5 Conclusion 41

1 Introduction à la réflexion

D'après Brian Smith, la réflexion est "*la capacité d'une entité à s'auto-représenter et plus généralement à se manipuler elle-même, de la même manière qu'elle représente et manipule son principal sujet.*" [1] [Smi90]. La réflexion [2] caractérise donc la propriété d'un système capable de raisonner et d'agir sur lui-même. Ainsi, un système réflexif est capable de contrôler son propre fonctionnement et de l'adapter en fonction des évolutions des besoins ou du domaine d'application du système. Cette faculté a suscité l'intérêt de plusieurs communautés, en particulier dans les domaines de l'intelligence artificielle, de la programmation logique, de la programmation fonctionnelle ou encore de la programmation par objets [DM95].

Les travaux de Pattie Maes [Mae87b] [Mae87a] ont fortement contribué à introduire la réflexion dans les systèmes et langages à objets. Cette association de la réflexion à l'approche objet est intéressante en particulier parce qu'elle offre deux niveaux de réutilisation complémentaires, car correspondant à différentes granularités. D'une part, la réflexion sépare les programmes et les mécanismes d'exécution, et permet donc de les modifier et de les réutiliser indépendamment les uns des autres. D'autre part, l'approche objet permet de modifier l'implémentation d'un objet et de la réutiliser sans modifier les objets avec lesquels il interagit.

1.1 Généralités

1.1.1 Introspection vs. intercession

Deux formes de réflexion peuvent être distinguées : *l'introspection* et *l'intercession*. Lorsqu'un système réflexif se contente de s'observer et de répondre à des questions sur son état, nous parlons de *l'introspection*. Quand il s'auto-modifie, nous parlons de *l'intercession*. Bobrow et al. donnent une définition qui résume bien ces deux formes de réflexion :

> "La réflexion est la capacité d'un programme à manipuler en tant que données les entités qui représentent l'état du programme pendant sa propre exécution. Il y a deux aspects d'une telle manipulation : *l'introspection* et *l'intercession*. L'introspection est la capacité d'un programme d'observer et donc de raisonner sur son propre état. L'intercession est la capacité d'un programme de modifier son propre état d'exécution, ou d'altérer sa propre interprétation ou signification. Ces deux aspects nécessitent un mécanisme pour encoder l'état d'exécution; fournir un tel encodage est appelé *réification*." [3] [BGW93]

1. *An entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates or and deals with its primary subject matter.*

2. Certains auteurs préfèrent utiliser le terme *réflexivité*.

3. *Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation:* introspection *and* intercession. *Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpreta-*

Introspection et intercession se basent donc sur le mécanisme de *réification* qui fournit une auto-représentation d'un système réflexif. La *réification* consiste à représenter l'état d'un système sous forme de données manipulables par ce même système.

1.1.2 Réflexion structurelle vs. comportementale

Il est d'usage de faire la distinction entre deux types de réflexion : la *réflexion structurelle* et la *réflexion comportementale* [Coi88].

> "[...] une distinction est traditionnellement faite entre les réflexions *structurelle* (ou réflexion de structure) et *comportementale* (ou réflexion de comportement) :
>
> • La *réflexion structurelle* implique la réification complète à la fois du programme en cours d'exécution ainsi que de ses types de données abstraits;
>
> • La *réflexion comportementale* implique la réification complète de sa propre sémantique (son processeur) de même qu'une réification complète des données utilisées pour exécuter le programme courant.
>
> Cette distinction a été faite principalement parce qu'il est beaucoup plus facile d'implanter la réflexion de structure efficacement que la réflexion comportementale." [Mal97, page 74]

La réflexion structurelle correspond à la capacité d'un système à représenter et manipuler sa face statique, i.e. les structures de données qui le constituent. Elle se traduit par la réification des entités utilisées pour la construction du système. Par exemple, dans un langage à classes de la famille de SMALLTALK [GR83], la réflexion de structure peut se manifester par la possibilité de modifier la représentation des classes.

La réflexion comportementale correspond à la capacité d'un système à représenter et manipuler sa face dynamique, i.e. sa manière de réaliser les services qu'il offre. Elle se traduit par la réification des entités utilisées pour le fonctionnement du système. Par exemple, la réflexion de comportement peut se manifester dans un système par la réification de la pile d'exécution ou de l'interprète du système.

Une relation forte lie la réflexion de structure et la réflexion de comportement. En effet, l'exécution d'un système fait appel à la description de ce système. Le contrôle de l'exécution d'un système (i.e. réflexion de comportement) nécessite donc d'avoir accès à la description de ce système (i.e. réflexion de structure).

1.1.3 Méta-niveaux et régression infinie

De manière générale, un système réflexif peut être vu comme un système qui contient son propre interprète [Smi84]. Le système étant capable de s'auto-modifier, il est donc capable de modifier son propre interprète, ce qui a pour conséquence de modifier le déroulement

tion meaning. Both aspects require a mechanism for encoding execution state as data ; providing such an encoding is called reification.

de son exécution. Comme le système peut raisonner sur son interprète et le modifier, l'interprète peut être vu comme un programme exécuté par un autre interprète. Ce deuxième interprète fait également partie du système et donc peut être modifié. Il peut donc être vu comme un programme exécuté par un troisième interprète, et ainsi de suite. Chaque interprète joue aussi le rôle d'un programme exécuté par un autre interprète. Cette dualité programme/interprète a pour conséquence de structurer les systèmes réflexifs en couches ou *méta-niveaux*. Chaque niveau étant décrit et contrôlé par celui qui est immédiatement au-dessus.

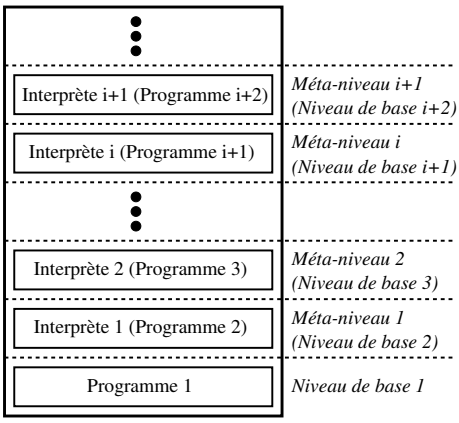

FIG. 1 – *Un système réflexif comporte un nombre potentiellement infini de méta-niveaux*

Un système réflexif peut donc être décomposé en différents niveaux (cf. figure 1). Le premier niveau, appelé *niveau de base*, décrit les traitements à réaliser, i.e. les tâches que le système doit réaliser. Le second niveau appelé *méta-niveau* (ou *niveau méta*) interprète le niveau de base. Du fait de la réflexion, l'interprète du niveau de base est réifié et peut donc être lui même manipulé et modifié. L'interprète peut lui-même être considéré comme un programme exécuté par un autre interprète, qui serait exécuté par un troisième interprète et ainsi de suite. Ainsi, le méta-niveau i représente le niveau de base pour le méta-niveau i+1 et ainsi de suite indéfiniment. Nous sommes donc confrontés à une régression potentiellement infinie de méta-niveaux. Cette régression doit obligatoirement être gérée dans chaque système réflexif afin que celui-ci réalise les tâches qui lui incombent avec des ressources finies (espace mémoire, temps, ...).

Note : Dans la pratique, l'étude d'un système réflexif se réduit le plus souvent à l'étude des deux premier niveaux : un niveau de base et un niveau méta. L'uniformité des systèmes réflexifs fait qu'un raisonnement fait sur un niveau méta (respectivement niveau de base) peut être transposé à un autre niveau méta (respectivement niveau de base).

1.2 Réflexion et objets

1.2.1 Méta-objet

Nous appelons *méta-objet* un objet qui joue le rôle d'interprète pour un ou plusieurs autres objets. Les méta-objets permettent de *contrôler* la structure et le comportement d'autres objets. Par exemple, un méta-objet peut définir la manière d'allouer la mémoire qui correspond à la structure des objets qu'il contrôle.

1.2.2 Réification

Nous appelons *réification* le résultat de l'opération du même nom[4] qui consiste à représenter des éléments de programmes sous forme d'objets. A titre d'exemple, les classes et les méthodes d'un programme peuvent être réifiées dans un langage réflexif.

1.2.3 Lien méta

Nous appelons *lien méta* la relation qui lie les objets aux méta-objets. Différentes variations du lien méta existent. Plus particulièrement, l'arité de ce lien peut varier d'un système à l'autre.

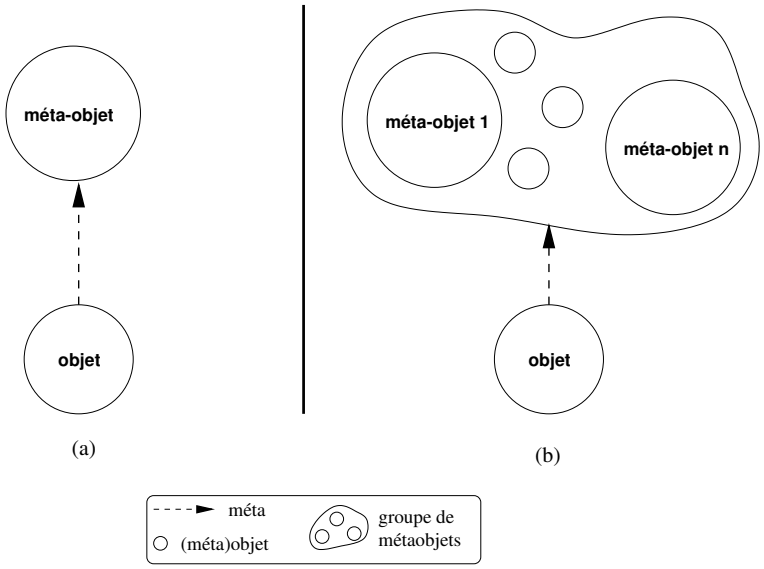

FIG. 2 – *Suivant les systèmes, un objet peut être contrôlé par un ou plusieurs méta-objets*

Suivant le cas, un objet peut être contrôlé par un seul méta-objet (cf. figure 2(a)) ou par plusieurs méta-objets (cf. figure 2(b)). Par exemple, dans les langages 3-KRS [Mae87b] et ABCL/R [WY88], un objet ne peut être contrôlé que par un seul méta-objet. Dans

4. "**Réification** [...] Action de réifier ; son résultat.
Réifier [...] Transformer en chose [...] **chosifier**." [Dictionnaire Le Petit Robert].

d'autres cas, comme pour le système concurrent CODA [McA95] ou le système d'exploitation APERTOS [Yok92], un objet peut être contrôlé par plusieurs méta-objets qui jouent des rôles complémentaires. La sémantique d'exécution d'un objet est fournie par la coopération de ses différents méta-objets.

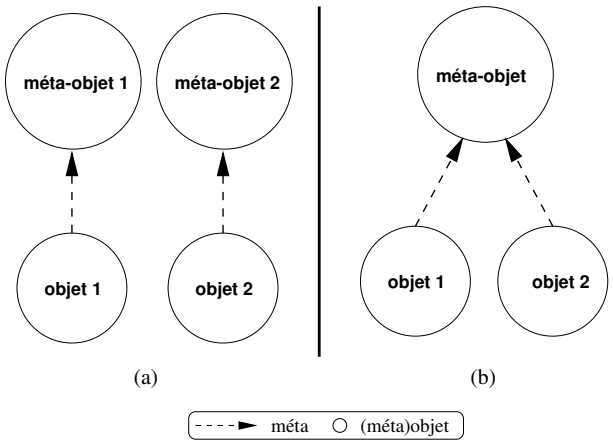

FIG. 3 – *Suivant les systèmes, un méta-objet peut contrôler un ou plusieurs objets*

Le nombre d'objets qui peuvent être contrôlés par un même méta-objet peut également varier. Ainsi, dans certains langages tels que ABCL/R [WY88], un méta-objet contrôle un unique objet (cf. figure 3(a)), alors que dans d'autres langages tels que METAXA [Gol97], un méta-objet peut contrôler plusieurs objets (cf. figure 3(b)). Notons que le nombre d'objets contrôlés par un même méta-objet n'est pas lié au nombre de méta-objets qui peuvent contrôler un objet. Ainsi, plusieurs objets peuvent être contrôlés par un même groupe de méta-objets.

1.2.4 Métaclasse

Les métaclasses sont des classes dont les instances sont aussi des classes. Elles sont exploitées dans différents langages et modèles à classes pour définir différentes sortes de classes en leur attribuant différentes *propriétés de classe*. Une propriété de classe correspond à une sémantique particulière de la classe par rapport à la sémantique par défaut des classes. Etre abstraite ou avoir plusieurs superclasses directes (héritage multiple) constituent deux exemples de propriétés de classe. De nombreuses autres propriétés ont été identifiées par Thomas Ledoux et al. [LC96].

Dans certains systèmes ou langages comme CLOS [Kee89], CLASSTALK [BC89], SOM [FCDR95], NÉOCLASSTALK [Riv97] ou METACLASSTALK [BS99b], les métaclasses peuvent être directement manipulées par les programmeurs notamment pour définir des propriétés de classes réutilisables. Pour définir une classe avec un comportement donné, il est nécessaire que cette classe soit instance de la métaclasse qui implémente la propriété souhaitée.

Une forme "primaire" de réutilisation est obtenue en définissant toutes les classes ayant la même propriété comme instances de la même métaclasse.

1.2.5 MOP

> "Les protocoles de méta-objets [les MOPs] sont des interfaces au langage qui offrent aux utilisateurs la possibilité de modifier incrémentalement le comportement et l'implémentation du langage, de manière analogue à la possibilité d'écrire des programmes avec le langage. [...] L'approche protocole de méta-objet [...] est basée sur l'idée que l'on peut et on doit 'ouvrir les langages', pour permettre aux utilisateurs d'ajuster la conception et l'implémentation à leurs besoins particuliers. [...] l'implémentation résultante ne représente pas un point unique dans l'espace global des conceptions de langages, mais plutôt une *région* entière à l'intérieur de cet espace." [5] [KdRB91, page 1]

Partant de la subdivision proposée par Chris Zimmermann [Zim96], nous considérons qu'il existe deux catégories de MOPs: les *MOPs explicites* et les *MOPs implicites*. Les MOPs implicites correspondent aux protocoles qui sont utilisés de manière transparente au niveau de base. De tels MOPs sont constitués de méthodes qui sont appelées de manière implicite et automatique pour contrôler l'exécution du niveau de base. A titre d'exemple, un message reçu par un prototype dans le langage réflexif à prototypes MOOSTRAP [Mul95] n'est pas immédiatement exécuté. En effet, le méta-objet de l'objet receveur prend automatiquement la main et contrôle l'envoi puis l'application de la méthode adéquate. Les phases de recherche et d'application des méthodes sont réalisées, respectivement, par les méthodes lookup et apply du méta-objet. Comme ces méthodes du méta-niveau sont exécutées à l'insu de l'objet receveur, elles font partie du MOP implicite de MOOSTRAP.

A l'opposé des MOPs implicites, les MOPs explicites correspondent aux protocoles qui sont utilisés explicitement. Il apparaissent explicitement dans le code du niveau de base. A titre d'exemple, le MOP de JAVA [6] est exclusivement explicite. Tous les traitements méta disponibles ne sont exécutées que si une méthode est explicitement appelée. Par exemple, la classe des méthodes réifiées java.lang.reflect.Method définit une méthode invoke(...) qui permet d'invoquer une méthode réifiée sur un objet. Cette méthode n'est exécutée que si elle apparaît explicitement dans un programme. Elle n'est jamais appelée implicitement. En effet, lors de l'appel d'une méthode m, la machine virtuelle prend directement en charge l'exécution de m. La méthode invoke(...) de la classe Method n'est pas appelée.

5. *Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the lanugage's behavior and implementation, as the ability to write programs within the language. [...] The metaobject protocol appproach [...] is based on the idea that one can and should "open languages up", allowing users to adjust the design and implementation to suit their particular needs. [...] the resulting implementation does not represent a single point in the overall space of language designs, but rather an entire* region *within that space.*

6. Depuis l'arrivée JDK 1.3, JAVA comporte un petit MOP implicite.

2 La réflexion dans Java

2.1 Capacités réflexives propres à Java

Au fil des versions, le langage JAVA [AG96] s'est progressivement ouvert à la réflexion. Les différentes versions ont successivement offert des possibilités d'introspection et d'intercession de plus en plus large. La toute dernière version (JDK 1.3) offre même la possibilité de contrôler l'appel de méthode.

2.1.1 JDK 1.1 et 1.2

> "L'API réflexion représente, [...], les classes, les interfaces et les objets dans la machine virtuelle courante de Java. [...] Avec l'API réflexion vous pouvez :
>
> - Connaître la classe d'un objet.
> - Obtenir l'information sur les qualificateurs d'une classe, ses champs, ses méthodes, ses constructeurs et ses superclasses.
> - Découvrir les constantes et les déclarations de méthodes qui appartiennent à une interface.
> - Créer une instance d'une classe dont le nom n'est connu qu'à l'exécution.
> - Lire et écrire la valeur d'un champ d'un objet, même si le nom du champ n'est connu qu'à l'exécution.
> - Appeler une méthode sur un objet, même si la méthode n'est connue qu'à l'exécution.
> - Créer un nouveau tableau, dont la taille et le type des éléments ne sont connus qu'à l'exécution, et modifier les éléments du tableau." [7] [Gre99].

Si les JDK 1.1 et 1.2 ne permettent pas de définir des méta-objets, ils offrent néanmoins la possibilité de manipuler plusieurs sortes de réifications. En effet, il existe des

7. *The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine. [...] With the reflection API you can:*
   - *Determine the class of an object.*
   - *Get information about a class's modifiers, fields, methods, constructors, and superclasses.*
   - *Find out what constants and method declarations belong to an interface.*
   - *Create an instance of a class whose name is not known until runtime.*
   - *Get and set the value of an object's field, even if the field name is unknown to your program until runtime.*
   - *Invoke a method on an object, even if the method is not known until runtime.*
   - *Create a new array, whose size and component type are not known until runtime, and then modify the array's components.*

réifications pour les constructeurs, les champs (i.e. variables d'instance), les méthodes, les qualificateurs[8] (e.g. final, static, ...), les tableaux et les classes.

Les réifications des champs et les objets sont liés de sorte que la modification de l'un se répercute sur l'autre. Dans la version 1.1 du JDK, ces modifications n'étaient possibles que pour les champs publics pour garantir la sécurité. La possibilité de contourner cette sécurité fait partie des nouveautés apportées par la version 1.2 du JDK. Ainsi, il est possible de réifier et modifier même les champs private ou protected, sous résèrves que ces champs ait été explicitement marqués comme modifiables (utilisation des mecanismes de sécurité de Java). De la même manière, le JDK 1.2 permet de manipuler et d'exécuter des réifications de méthodes private ou protected à condition que ces méthodes ait été explicitement marquées comme tel.

Le passage d'un niveau à l'autre se fait de manière explicite, puisque le MOP de JAVA (JDK 1.1 et 1.2) est explicite. Par exemple, il faut explicitement créer des réifications de champs pour les manipuler. Du fait de l'absence de passage implicite d'un niveau à l'autre, les possibilités de la réflexion dans les JDK 1.1 et 1.2 sont très restreintes. Ces restrictions sont d'autant plus importantes qu'il n'est pas possible d'étendre le langage ou de le modifier en modifiant les classes des réifications ou en les sous-classant. Par exemple, la classe des méthodes réifiées java.lang.reflect.Method est qualifiée final (i.e. pas sous-classable). Il n'est donc pas possible de définir une sous-classe de méthodes de réifiées qui redéfinit la manière d'exécuter les méthodes (i.e. redéfinition de la méthode invoke(...)).

2.1.2 JDK 1.3

Le JDK 1.3 (actuellement en version bêta) introduit les encapsulateurs (*wrappers*) d'objets afin de permettre de capturer les appels de méthodes. Pour ce faire, le paquetage (*package*) java.lang.reflect a été étendu avec la classe Proxy et l'interface InvocationHandler [Sun99]. Grâce à la classe Proxy, il est possible de définir un encapsulateur pour un objet donné. Le type de l'encapsulateur ainsi créé est un sous-type de l'objet encapsulé. Lorsqu'un encapsulateur reçoit un appel de méthode donné, il appelle la méthode invoke(...) pour un objet chargé de la gestion des appels de méthodes. L'objet chargé de la gestion des appels de méthodes peut être différent de l'objet encapsulé. La seule contrainte est qu'il doit être instance d'une classe qui implémente l'interface java.lang.reflect.InvocationHandler, ce qui se traduit par la définition de la méthode invoke(...).

Pour créer un encapsulateur, le développeur doit seulement appeler la méthode statique newProxyInstance(...) de la classe Proxy. Cependant, si la création de l'encapsulateur est automatisée, la garantie de l'encapsulation reste à la charge du développeur. En effet, le développeur a la responsabilité de référencer l'encapsulateur en lieu et place de l'objet encapsulé pour assurer la capture des appels de méthodes.

Les figures 4 et 5 donnent respectivement le schéma et le code source d'un exemple d'utilisation de proxy. Cet exemple est inspiré de la documentation du JDK 1.3 [Sun99] et consiste à produire une trace avant et après chaque appel de méthode des instances de la

8. *modifiers.*

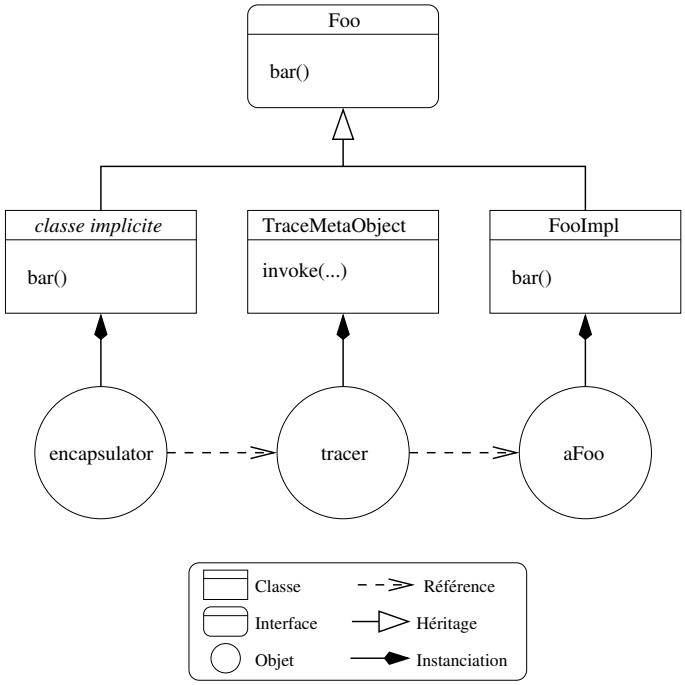

FIG. 4 – *Exemple d'utilisation des encapsulateurs du JDK 1.3*

classe FooImpl(...).

Une instance de la classe FooImpl correspond à l'objet aFoo que nous souhaitons encapsuler (cf. ligne 3). Le contrôle des messages destinés à aFoo est réalisé par tracer une instance de la classe TraceMetaObject (cf. ligne 4). Le méta-objet tracer référence aFoo, auquel il transmettra les appels de méthodes après avoir produit une trace. Afin que tracer réalise cette trace, il est nécessaire qu'il reçoive les appels de méthodes destinés à aFoo. Cette redirection des appels de méthodes est réalisée par un encapsulateur encapsulator instance de Proxy.

L'encapsulateur encapsulator est créé en appelant la méthode statique newProxyInstance(...) de la classe Proxy. Cette méthode prend trois arguments : un chargeur de classes (*class loader*), un tableau d'interfaces et le méta-objet chargé de la gestion des appels de méthodes (tracer). Le chargeur de classe est nécessaire au système pour y définir une classe implicite dont l'encapsulateur est l'instance. Le tableau d'interfaces définit le type du proxy généré. En effet, la classe implicite du proxy implémente ces différentes interfaces.

L'encapsulateur ainsi généré ne cache pas l'objet aFoo. Ainsi, il est possible d'envoyer l'appel d'une méthode bar() directement à aFoo (cf. ligne 6). Un tel appel provoque simplement l'exécution de cette méthode. En revanche, si l'appel de la méthode bar() est envoyé à l'encapsulateur encapsulator (cf. ligne 7), ce dernier appelle la méthode invoke(...) (cf. ligne 22) pour le méta-objet tracer. La méthode invoke(...) prend comme arguments le proxy encapsulator qui a généré l'appel, la réification de la méthode appelée bar() et un tableau avec les arguments de l'appel de bar(). Cet appel de la méthode invoke(...) génère

```
1:  public class MyApplication {
2:    public static void main(String [] args) {
3:      FooImpl aFoo = new FooImpl();   //objet à encapsuler
4:      TraceMetaObject tracer = new TraceMetaObject(aFoo);   //méta-objet
5:      Foo encapsulator = Proxy.newProxyInstance(aFoo.getClass().getClassLoader(),
                                                  aFoo.getClass().getInterfaces(),
                                                  tracer);

6:      aFoo.bar();   //appel non-tracé

7:      encapsulator.bar();   //appel tracé
8:    }
9:  }
```

```
10:  public interface Foo {
11:    Object bar() throws BarException;
12:  }
```

```
13:  public class FooImpl implements Foo {
14:    Object bar() throws BarException {
     //...
15:    }
16:  }
```

```
17:  public class TraceMetaObject implements java.lang.reflect.InvocationHandler {
18:    private Object obj;

19:    public TraceMetaObject(Object obj) {
20:      this.obj = obj;
21:    }

22:    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
23:      Object result;
24:      try {
25:        System.out.println("before method " + m.getName());
26:        result = m.invoke(obj, args);
27:      } finally {
28:        System.out.println("after method " + m.getName());
29:      }
30:      return result;
31:    }
32:  }
```

FIG. 5 – *Exemple d'utilisation des encapsulateurs du JDK 1.3*

une trace avant et après l'exécution de la méthode bar() pour aFoo (utilisation du protocole de méthodes réifiées). La trace est générée au sein d'un bloc try/finally pour produire une trace même si l'appel de bar() génère une exception.

2.2 Principales applications des capacités réflexives de Java

Bien que limitées, les capacités réflexives de JAVA ont de nombreuses applications dont les plus connus sont la documentation [Ben98] et la réalisation d'outils de développement (browsers, inspecteurs, debogueurs,...) [Tre98]. Nous présentons ici trois autres applications importantes que sont la serialisation, les composants *Java Beans* et les communications distantes via RMI.

2.2.1 Serialisation

Depuis la version 1.1, JAVA offre la possibilité de *serialiser* les objets, i.e. d'en fournir une représentation binaire qui peut être sauvegardée dans le but de servir ultérieurement à la reconstruction des objets [Eck98, chapitre 10, page 395]. Cette représentation est nécessaire pour réaliser la persistance en sauvegardant les objets sur un support de masse. Elle est également indispensable pour transférer les objets à travers un réseau.

Par défaut, les capacités réflexives de JAVA sont exploitées aussi bien lors de production des représentations binaires des objets (i.e. serialisation) que lors de la reconstruction des objets (i.e. dé-serialisation). Lors de la serialisation d'un objet[9], l'ensemble des références qu'il comporte vers d'autres objets sont parcourues récursivement [Sun98a]. Pour ce faire, l'ensemble des champs de l'objet (hormis ceux qualifiés *static* ou *transient*) sont lus et successivement serialisés. La lecture des champs est réalisée en appelant la méthode get(...) pour les réifications des champs (instances de java.lang.reflect.Field). Ces réifications sont obtenues en invoquant des méthodes sur la classe de l'objet et les superclasses de celle-ci.

La dé-serialisation des objets se fait de manière symétrique. L'ensemble des réifications des champs de l'objet à reconstruire sont obtenues en parcourant la classe de l'objet et les superclasses de celle-ci. L'objet recréé est initialisé en appelant la méthode set(...) (définie dans la classe java.lang.Field) pour chacune des réifications de ses champs.

2.2.2 Java Beans

D'après les spécifications données par Sun :

> "un *Java Bean* est un composant logiciel réutilisable qui peut être manipulé visuellement dans un outil de construction[10]" [Ham97, page 9].

9. JAVA ne permet une serialisation que si la classe de l'objet à sérialiser hérite de l'interface marqueur Serialisable.

10. *A Java Bean is a reusable software component that can be manipulated visually in a builder tool.*

Cette définition très générale est précisée par la liste des caractéristiques communes à tous les *beans* :

- **La communication par événements**[11] **:**
  Un bean doit supporter la communication par événements. L'idée de base de ce modèle de communication consiste à ce qu'un bean source notifie à des beans écouteurs la réalisation de faits (d'événements) intéressants.

- **Les propriétés :**
  Un bean peut disposer d'un ensemble de propriétés qui correspondent à des attributs qui peuvent être accédées par l'intermédiaire de méthodes particulières (*getters/setters*).

- **L'introspection :**
  Un bean doit offrir la possibilité d'être introspecté afin que les outils de construction (tel que la *BeanBox*) puissent le manipuler.

- **La personnalisation**[12] **:**
  Un développeur doit pouvoir personnaliser l'apparence et le comportement d'un bean.

- **La persistance :**
  Un bean doit être persistant dans le sens où les personnalisations qui lui sont apportées ne doivent pas être perdues. Elles doivent être sauvegardées afin qu'elles soient retrouvées par la suite.

Dans les java beans, les capacités réflexives de JAVA sont exploitées principalement pour la persistance et l'introspection. Afin de réaliser la persistance, le modèle des Java Beans s'appuie sur les mécanismes de serialisation de JAVA (cf. section 2.2.1). Ces mécanismes de serialisation utilisent les capacités réflexives de JAVA en matière d'introspection pour déterminer la structure des beans afin de la sauvegarder ou de la restaurer. Afin de supporter l'introspection, un bean peut fournir une définition de ses caractéristiques sous la forme d'une classe *BeanInfo* associée. En l'absence de cette classe, l'introspection est réalisée en analysant les signatures et les qualificateurs des méthodes du bean. Cette analyse exploite directement l'API réflexion de JAVA et notamment la classe java.lang.reflect.Method.

2.2.3 RMI

L'API de distribution RMI (*Remote Method Invocation*) introduite par JAVA constitue un cadre de développement destiné à simplifier la réalisation des applications réparties [Sun98b]. Plus concrètement, RMI permet aux objets gérés par une JVM[13] d'appeler les méthodes d'objets gérés par d'autres JVMs. Ces autres JVMs peuvent, éventuellement, être localisées sur des machines distantes [Jaw98, chapitre 37, page 793].

11. *Events.*
12. *Customization*
13. *Java Virtual Machine.*

L'API RMI exploite les mécanismes de serialisation de JAVA afin de transmettre les arguments des appels de méthodes ainsi que les résultats des ces appels à travers un réseau. De ce fait, RMI utilise les capacités réflexives de JAVA de manière indirecte, puisque ces dernières sont utilisées lors de la serialisation/dé-serialisation des objets. Par ailleurs, depuis la version 1.2 du JDK, la réflexion est directement exploitée pour déterminer la méthode à exécuter sur le serveur lors d'un appel. Pour ce faire, des méthodes réifiées (instances de java.lang.reflect.Method) sont manipulées. Cette mise en œuvre de la réflexion est réalisée comme suit.

Un client RMI dispose d'un objet proxy qui référence l'objet serveur. La classe de l'objet proxy implémente au moins une interface implémentée par la classe de l'objet serveur. Ce proxy traduit chaque appel de méthode qu'il reçoit en requête qu'il transmet au serveur. Une requête est constituée entre autres des arguments de l'appel et d'une méthode m(...) réifiée qui appartient à l'une des interfaces implémentées par la classe du serveur. Après la transmission de la requête via le réseau, la méthode m(...) réifiée est exécutée sur le serveur. Pour réaliser cette exécution, m(...) reçoit un appel de méthode invoke(...).

3 Principales extensions réflexives de Java

A l'exception du contrôle de l'appel de méthodes introduit tout récemment avec les encapsulateurs du JDK 1.3, JAVA ne dispose pas de MOP implicite. Les services offerts par une application (i.e. ses "fonctionnalités") ne sont pas contrôlés par le méta-niveau. Autrement dit, les capacités réflexives de JAVA se limitent à la réflexion de structure et n'offrent pas de possibilités pour la réflexion de comportement. Dès lors, il n'est pas possible de profiter d'une grande partie des avantages offerts par la réflexion tel que le développement de systèmes ouverts [KP94] [Kic96] ou la séparation des aspects [Led98] [BS99a].

Afin d'étendre les capacités réflexives des JDK 1.1 et 1.2 et permettre de contrôler l'exécution des applications, diverses extensions de JAVA ont vu le jour (METAXA [KG96], OPENJAVA[CT98], PROACTIVE [CKV98], REFLECTIVE JAVA [Zhi98], DALANG [WS99], GUARANÁ [OB99]...). Les approches adoptées dans ces extensions peuvent être classées en deux familles suivant que le passage du niveau de base au niveau méta est réalisé statiquement (*approches statiques*) ou dynamiquement (*approches dynamiques*).

Les approches statiques consistent à réaliser le passage du niveau de base au niveau méta avant l'exécution. Elles se traduisent par la fusion des différents niveaux (de base et méta) et donc "l'aplatissement" de la tour des méta-niveaux. Concrètement, cette fusion consiste à transformer le code (source ou byte-code) correspondant au niveau de base des applications afin d'y incorporer le code du niveau méta.

Les approches dynamiques consistent à conserver la tour des méta-niveaux à l'exécution et à réaliser les passages d'un niveau à l'autre dynamiquement, chaque fois que cela est nécessaire. Pour ce faire, il faut *d'intercepter* (capturer) les traitements du niveau de base afin de réaliser les traitements du méta-niveau. Afin de capturer les traitements du niveau de base, il est nécessaire de disposer *d'intercepteurs* (*hooks*) qui réalisent le passage du niveau de base au niveau méta. L'introduction des intercepteurs peut se faire soit par transformation de code (source ou byte-code), soit par la modification de l'interprète JAVA (interprète du code source ou du byte-code i.e. machine virtuelle).

En résumé, quatre types d'intervention permettent l'extension des capacités réflexives de JAVA:

1. Le premier type d'intervention consiste à transformer le code source des applications soit pour introduire des intercepteurs (approches dynamiques), soit pour fusionner les niveaux (approches statiques).

2. Le second type consiste à transformer le byte-code des applications. De la même manière que pour la transformation de code source, ce type d'intervention peut être utilisé aussi bien dans les approches statiques que dynamiques.

3. Le troisième type d'intervention consiste à modifier la machine virtuelle JAVA pour introduire des intercepteurs. Ce type d'intervention n'est utilisable que dans les approches dynamiques.

4. Le quatrième type d'intervention consiste à modifier un interprète de code source JAVA afin d'intercepter les opérations du niveau de base. De même que pour les interventions sur la machine virtuelle JAVA, ce type d'intervention ne peut être mis en œuvre que dans les approches dynamiques.

Dans ce qui suit, nous présentons brièvement les principales extensions réflexives de JAVA. Nous les avons classées suivant le type d'intervention adopté pour étendre les capacités réflexives de JAVA. Parmi les quatre types d'intervention possibles, seuls les trois premiers sont représentés (transformation de code source, transformation de byte-code et modification de la machine virtuelle). En effet, à notre connaissance, le quatrième type d'intervention a été adopté par une seule équipe dont les travaux sont encore en cours [DS99].

Nous nous retrouvons donc avec trois groupes d'extensions réflexives de JAVA correspondant à trois types d'intervention. Le premier groupe correspond aux extensions basées sur la transformation du source des applications soit pour introduire des intercepteurs (approches dynamiques), soit pour fusionner les niveaux (approches statiques). Le second groupe réunit les extensions basées sur la transformation du byte-code des applications. Bien que la transformation de byte-code puisse être utilisée aussi bien dans les approches statiques que les approches dynamiques, ce groupe ne réunit que des représentants de la famille des approches dynamiques. Il n'existe pas, à notre connaissance, de représentant de la famille des approches statiques qui exploitent la transformation de byte-code. Enfin, le troisième groupe contient des extensions réflexives de JAVA basées sur la modification de la machine virtuelle de JAVA (approches dynamiques uniquement).

3.1 Extensions basées sur la transformation de code source

Les trois extensions de JAVA que sont REFLECTIVE JAVA [WS97], OPENJAVA [CT98] et PROACTIVE [CKV98] font partie de cette famille. REFLECTIVE JAVA et OPENJAVA s'appuient sur des pré-processeurs ad hoc. Ces pré-processeurs permettent d'introduire dans le processus de compilation une étape de transformation du code source des applications pour lier le code du niveau de base à celui du niveau méta. Ainsi, partant d'une application qui n'exploite pas la réflexion, il est possible d'introduire des traitement méta en utilisant les pré-processeurs qui transforment le code source. PROACTIVE ne dispose pas d'un tel pré-processeur. Pour exploiter la réflexion dans une application existante, les développeurs doivent en modifier le code source manuellement.

3.1.1 Reflective Java

REFLECTIVE JAVA offre la possibilité de capturer les appels de méthodes [WS97] [Zhi98] [Tas97]. Pour ce faire, un pré-processeur génère, pour chacune des classes désignées (dans un fichier) par les développeurs, une sous-classe qui redéfinit les méthodes à contrôler. Ces redéfinitions consistent en des appels de méthodes des méta-objets (définis par les développeurs) afin de réaliser les traitements méta. Ainsi, pour pouvoir utiliser les capacités

réflexives de REFLECTIVE JAVA, il est nécessaire d'instancier les sous-classes générées par le pré-processeur.

```
1:   class CompteBancaire {
2:     protected float solde;
3:     public float lireSolde() { return solde; }
4:     public void debit(float montant) { solde -= montant; }
5:     public void credit(float montant) { solde += montant; }
6:   }
7:   refl_class CompteBancaire : MetaSynchro {
8:     public float lireSolde() : 100;
9:     public void debit(float montant) : 200;
10:    public void credit(float montant) : 200;
11:  }
12:  MetaSynchro extends MetaObject {
13:    Lock monitor;
14:    public void metaMethod(MID methodID, CID categoryID, Pack args, Pack result){
15:      switch (categoryID) {
16:      case 100 : monitor.set_read_lock(); break;
17:      case 200 : monitor.set_write_lock(); break;
18:      }
19:      result = callBaseLevel(methodID, args);
20:      switch (categoryID) {
21:      case 100 : monitor.release_read_lock(); break;
22:      case 200 : monitor.release_write_lock(); break;
23:      }
24:    }
25:  }
```

FIG. 6 – *Exemple d'utilisation des catégories de méthodes de* REFLECTIVE JAVA

Un langage de script permet de définir les liens entre objets et méta-objets. En particulier, il permet de définir les méthodes dont l'appel doit être capturé. Chacune de ces méthodes est associées à une *catégorie de méthode*. Ces catégories permettent de déterminer le traitement méta à réaliser lorsqu'un appel de méthode est capturé.

Afin d'illustrer les catégories de méthodes, considèrons l'exemple de la figure 6 Les lignes 1 à 6 correspondent à la définition d'une classe de comptes bancaires. Pour utiliser cette classe dans un contexte concurrent, il est nécessaire de synchroniser les accès au champ solde. Cette synchronisation est réalisée à l'aide de méta-objets instance de la classe MetaSynchro (cette classe définie par les lignes 12 à 25). Le lien entre les instances de CompteBancaire et les méta-objets instances de MetaSynchro sont définis à l'aide du langage

de script de REFLECTIVE JAVA (ligne 7) De plus de ce lien, le langage de script permet de définir les catégories des méthodes de la classe CompteBancaire. Pour chaque méthode, l'entier identifiant de sa catégorie est donnée (lignes 8 à 11). Suivant la valeur de cet entier, le traitement méta réalisé par la méthode metaMethod(...) change (lignes 14 à 25).

3.1.2 OpenJava

OPENJAVA [CT98] [Tat99] reprend l'idée de la réflexion à la compilation de la version 2 de OPENC++ [Chi95] réalisé au sein de la même équipe. En fonction de la définition du niveau méta, un pré-processeur transforme le code source correspondant au niveau de base de sorte à y introduire les traitements méta. Il y a alors fusion du niveau de base avec le niveau méta et la disparition de la tour de méta-niveaux. Le code ainsi transformé peut alors être compilé par un compilateur JAVA standard.

Le MOP de OPENJAVA permet la ré-écriture des éléments de programme (i.e. nœuds de l'arbre syntaxique comme les déclarations de classes, déclarations de variables, appels de méthodes, affectations...) de chaque application. Pour ce faire, ces éléments de programme sont réifiés à la compilation par le pré-processeur OPENJAVA. La classe de chacun de ces éléments de programme réifié est déterminée à partir d'annotations (extension de la syntaxe JAVA) qui apparaissent dans le code source initial d'une application OPENJAVA. Chacune de ces classes définit un ensemble de transformations à réaliser sur les éléments de programme qui en sont les instances. Ainsi, chaque élément de programme opère, sur lui-même, les transformations permettant d'introduire les traitements méta. Le résultat de ces transformations est du code JAVA standard où les traitements du niveau de base et les traitements du niveau méta sont fusionnés.

La figure 7 donne un exemple de tels transformations. Le développeur définit une classe Hello instance d'une métaclasse VerboseClass. Cette dernière métaclasse définit les transformations correspondant à un traitement méta qui consiste à produire une trace. Après transformation, ces traitements méta sont fusionnés avec la définition de la classe Hello.

3.1.3 Proactive

PROACTIVE (anciennement appelé JAVA parallèle, JAVA//) vise à constituer un support transparent pour la réalisation d'applications distribuées concurrentes comportant des objets actifs [Vay97] [CKV98]. L'approche de PROACTIVE est basée sur l'utilisation d'encapsulateurs afin d'intercepter les messages reçus par chaque objet. Pour ce faire, un protocole de création particulier doit être utilisé en lieu et place du new de JAVA afin de générer les encapsulateurs. En effet, la création d'un objet se fait par appel de l'une des méthodes statiques d'une classe particulière de la bibliothèque de PROACTIVE.

Lors de l'exécution d'une application, un tel appel destiné à instancier une classe C provoque la création d'un encapsulateur (appelé *stub*) et d'un méta-objet (appelé *proxy*). La création proprement dite de l'instance de C est laissée à la charge du méta-objet. L'encapsulateur est retourné pour recevoir les messages destinés à l'instance de C. Au préalable, si nécessaire, la classe E de l'encapsulateur est créée (génération de code source,

```
public class Hello instantiates VerboseClass {
  public static void main( String[] args ) {
    hello();
  }
  static void hello() {
    System.out.println( "Hello, world." );
  }
}
```

```
import openjava.mop.*;
import openjava.ptree.*;
public class VerboseClass instantiates Metaclass
        extends OJClass {
  public void translateDefinition()
        throws MOPException {
    OJMethod[] methods = getDeclaredMethods();
    for (int i = 0; i ¡methods.length; ++i) {
    Statement printer = makeStatement(
      "System.out.println( \"" + methods[i] +
      " is called.\" );");
    methods[i].getBody().insertElementAt( printer, 0 );
    }
  }
}
```

Avant transformation

```
public class Hello {
  public static void main( String[] args ) {
    System.out.println( "main is called." );
    hello();
  }
  static void hello() {
    System.out.println( "hello is called." );
    System.out.println( "Hello, world." );
  }
}
```

Après transformation

FIG. 7 – *Exemple de transformation réalisée par* OPENJAVA

compilation et chargement) dynamiquement. Afin d'éviter les problèmes de typage, E est définie comme sous-classe de C. Cette classe de l'encapsulateur (E) doit redéfinir toutes les méthodes définies dans C et les superclasses de C afin de permettre la réification des appels de méthodes et leurs transmettre au méta-objet. Pour réaliser cette transmission, l'encapsulateur dispose d'une référence sur un méta-objet qui référence à son tour l'objet encapsulé. Les traitements réalisés par un méta-objet dépendent en particulier de sa classe. Celle-ci varie suivant le protocole de création utilisé.

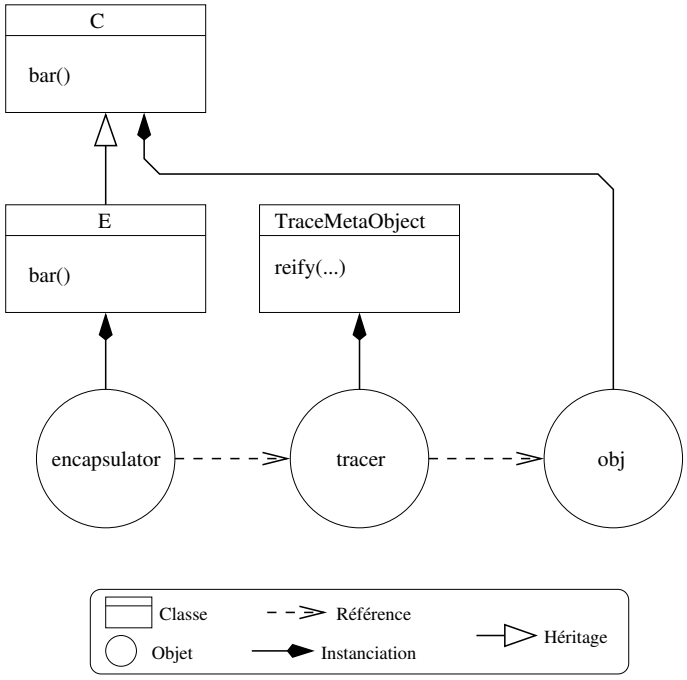

FIG. 8 – *Utilisation des encapsulateurs dans* PROACTIVE

La figure 8 donne un exemple d'utilisation des encapsulateurs dans PROACTIVE. Lors de la création d'une instance obj de la classe C, un encapsulateur encapsulator instance de E sous-classe de C est créé, ainsi qu'un méta-objet tracer. La référence de encapsulator est retournée pourqu'il recevoive les appels de méthodes destinés à obj. La classe E redéfinit les méthodes de C pour appeler la méthode reify(...) du méta-objet tracer. Cet appel qui prend pour arguments une reification du message reçu permet de réaliser le traitement méta. Ainsi, lorsque encapsulator reçoit le message bar(), il appelle la méthode reify(...) de tracer. Le méta-objet tracer réaliser alors le traitment méta qui conssite à produire une trace dans ce exemple. Puis, il appelle la méthode bar() pour obj, réalisant ainsi l'opération du niveau de base. Le résultat de cet appel est recueilli par le méta-objet qui le transmet à encapsulator qui retourne à son tour cette valeur comme résultat de l'appel de bar() qu'il a reçu.

3.2 Extensions basées sur la transformation de byte-code

Dans JAVA, la compilation d'une application produit des fichiers de *byte-code* interprètables par la machine virtuelle JAVA. La transformation de ces byte-codes constitue une approche possible pour introduire la réflexion. Approche qui a été adoptée par certaines extensions réflexives de JAVA telle que JAVASSIST [Chi98], DALANG [WS98] et KAVA [WS99]. Cette transformation peut se faire de manière statique après la compilation ou de manière paresseuse lors du chargement des classes. En effet, JAVA offre la possibilité de contrôler le chargement de classes. Ainsi, il est possible de définir une nouvelle classe de chargeurs de classe (*class loaders*) et d'en utiliser une instance pour le chargement de certaines classes. Le chargeur ainsi créé, transforme le byte-code avant de créer les classes.

3.2.1 Dalang

La transformation de byte-code opérée par DALANG [WS98] [WS99] se résume en l'introduction d'encapsulateurs. Pour ce faire, les classes du niveau de base sont remplacées par des classes d'encapsulateurs dans le but d'intercepter les appels de méthodes. Cette substitution se fait en deux étapes. La première étape consiste à renommer la classe dont les instances doivent être encapsulées, et ce en intervenant sur son byte-code. La seconde étape consiste à générer la classe des encapsulateurs avec le nom original de la classe du niveau de base manipulée. La classe d'encapsulateurs hérite d'une classe de méta-objets qui définit les traitements à réaliser avant et après chaque appel de méthode. Les encapsulateurs jouent donc également le rôle de méta-objets. Un fichier de "configuration méta" (*Meta Config*) associe à chaque classe du niveau de base la classe de méta-objets à utiliser.

Pour illustrer le mécanisme d'interception utilisé dans DALANG, prenons l'exemple de la figure 9. La classe C définie par le développeur a été renommée par DALANG en C_Original. Le nom C est attribué à une nouvelle classe généré par DALANG. Cette nouvelle classe C hérite de la classe TraceMetaObject la méthode invokeMethod(...) destinée à contrôler les appels de méthodes. Comme la nouvelle classe C remplace C_Original, elle définit les méthodes de C_Original tel que bar(). Cette définition consiste à appeler la méthode invokeMethod(...) de sorte à réaliser les traitements méta, qui correspondent à la production d'une trace dans cet exemple. Ainsi, une instance tracerEncap de la nouvelle classe C a une interface analogue à celle des instances de la classe de départ C_Original. L'objet tracerEncap joue le double rôle d'encapsulateur et de méta-objet. En effet, tracer masque une instance obj de C_original et réalise les traitements méta correspondant à des appels de la méthode invokeMethod(...).

3.2.2 Kava

Partant de leur expérience avec DALANG, Ian Welch et Robert Stroud ont réalisé une nouvelle extension réflexive de JAVA baptisée KAVA [WS99]. Afin d'éviter certaines limitations de DALANG, KAVA est basé sur l'utilisation des *encapsulateurs de byte-code* comme intercepteurs. Au lieu d'encapsuler les instances d'une classe C comme cela est fait dans DALANG, le byte-code de C est transformé de sorte à "encapsuler" certains byte-codes ou

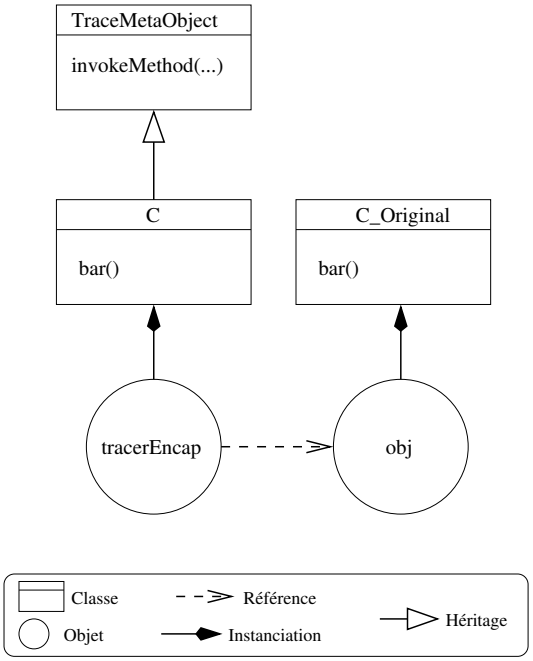

FIG. 9 – *Utilisation des encapsulateurs dans* DALANG

groupes de byte-code. Il s'agit concrètement d'introduire de nouveaux byte-codes avant et après le byte-code à encapsuler comme c'est le cas pour le byte-code2 dans l'exemple de la figure 10.

Grâce aux micro-encapsulateurs, KAVA introduit des appels au méta-niveau dans le byte-code des classes. En particulier, chaque objet référence un méta-objet qui est appelé pour contrôler les différentes opérations (e.g. appels de méthodes ou accès à la structure). Outre la réalisation de traitements d'ordre méta, un méta-objet a la possibilité d'éviter la réalisation de l'opération contrôlée. Cette possibilité est réalisée à l'aide du mécanisme d'exception. L'exemple suivant illustre ce mécanisme d'exception et l'utilisation des encapsulateurs de byte-code. Il s'agit de la transformation de l'expression suivante qui consiste en l'écriture d'une valeur dans la variable d'instance message :

```
class Hello {
  String message = "Hello World";
  public void sayHelloTo(String name){
    message = "Hello " + name;
  }
}
```

Le groupe de byte-code qui représente cette modification de message est transformé

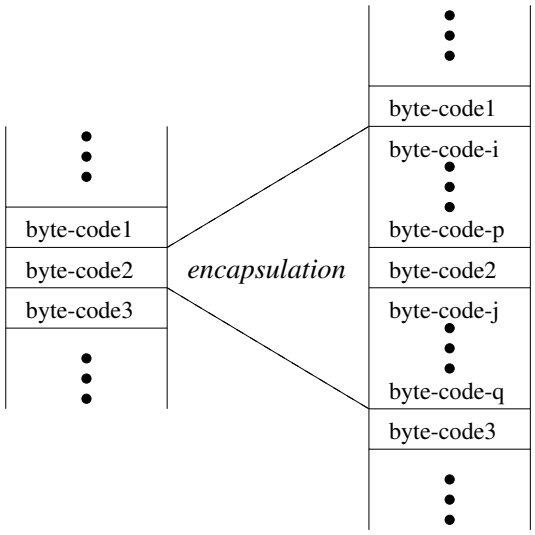

FIG. 10 – *Encapsulation de byte-code*

pour obtenir :

```
class Hello {
  String message = "Hello World";
  public void sayHelloTo(String name){
    1: Value fieldValue = Value.newValue("Hello " + name);
    2: meta.beforePutField("message", fieldValue);
    3: try {
    4:   helloWorld = (String) fieldValue.getObject();
    5: } catch(MetaSupressUpdateException e) {}
    6: meta.afterPutField("message", fieldValue);
  }
}
```

Dans le byte-code résultant de la transformation, la ligne 1 détermine la valeur que doit prendre la variable d'instance message. Cette valeur ainsi que le nom de la variable d'instance sont transmis, comme arguments de la méthode beforePutField(...), au méta-objet de l'objet dont l'état doit être modifié (ligne 2). Cette méthode réalise des traitements méta et peut éventuellement changer la valeur à affecter à la variable d'instance message. Afin d'interdire la modification de la variable d'instance, une exception MetaSupressUpdateException peut être levée par getObject() (ligne 4). En dernier lieu, la méthode afterPutField(...) du méta-objet est appelée pour réaliser les traitements méta qui suivent une écriture de variable d'instance (ligne 6).

3.2.3 Javassist

JAVASSIST est un assistant (*wizard*) permettant la génération automatique de code JAVA partant d'annotations [Chi98]. Cet outil a évolué pour offrir dans sa dernière version (0.3) une librairie de manipulation de byte-code [Chi99]. La réflexion constitue l'une des principales applications de cette librairie.

Code source initial	Code source équivalent au byte-code transformé

`class X {`
`  int f(int i) {`
`  return i + 1; }`
`}`
	`class X {`
	`  int _original_f(int i) {`
	`  return i + 1; }`
	
	`  int f(int i) {`
	`  /* Transmettre l'appel au méta-objet */ }`
	`}`

FIG. 11 – *Exemple de transformation de byte-code réalisée dans* JAVASSIST

Pour introduire la réflexion, JAVASSIST modifie le byte-code des classes afin de rediriger les appels de méthodes vers des méta-objets. Pour ce faire, il renomme les méthodes existantes. L'ancien nom est attribué à une nouvelle méthode qui transmet les appels au méta-objet de l'objet courant (`this`). La figure 11 donne le code source correspondant à un exemple d'une telle transformation byte-code. Après transformation de la classe `X`, la méthode `f(...)` transmet l'appel au méta-objet associé à l'objet courant. Afin de réaliser les traitements liés à la méthode `f(...)`, le méta-objet doit appeler la méthode `_original_f(...)`.

3.3 Extensions basées sur la modification de la machine virtuelle

Nous étudions deux extensions de JAVA représentant cette famille : METAXA (anciennement METAJAVA) [KG96] [GK99] et GUARANÁ [OB98b] [OB99]. Ces deux extensions utilisent une machine virtuelle modifiée afin d'intercepter l'exécution des opérations.

3.3.1 MetaXa

METAXA est une extension réflexive de JAVA [KG96] [GK99] [Gol97] [GK98]. Afin de contrôler l'exécution à l'aide de méta-objets, METAXA modifie la machine virtuelle JAVA. Le passage du niveau de base au méta-niveau s'appuie sur un mécanisme d'événements de la machine virtuelle. Avant d'exécuter une opération du niveau de base, comme par exemple l'appel d'une méthode ou l'accès à un champ, un événement est généré. Cet événement est transmis aux méta-objets abonnés à l'événement (i.e. les méta-objets associés à l'objet sur lequel l'opération du niveau de base doit être réalisée). Dans le cas où aucun méta-objet n'est abonné à cet événement, l'opération est exécutée comme dans une machine virtuelle JAVA standard.

Dans MetaXa, toutes les instances d'une classe sont contrôlées par un même méta-objet. En effet, la référence du méta-objet est placée au niveau de la classe. Il est néanmoins possible de contrôler un objet par un méta-objet différent du méta-objet partagé par toutes les instances de sa classe. Pour ce faire, l'objet devient instance d'une sous-classe implicite (*shadow class*) de sa classe initiale. MetaXa offre également la possibilité d'attacher un méta-objet à une référence particulière d'un objet. Le méta-objet ne contrôle alors que les opérations qui impliquent la référence à laquelle il est attaché.

MetaXa offre enfin la possibilité d'associer plusieurs méta-objets à un même objet. Dans ce cas, les méta-objets sont ordonnés dans une chaîne suivant l'ordre inverse de celui suivant lequel ils ont été attachés à l'objet. Chaque méta-objet peut ainsi masquer les méta-objets qui ont été attachés avant lui ou les faire participer au contrôle de l'objet.

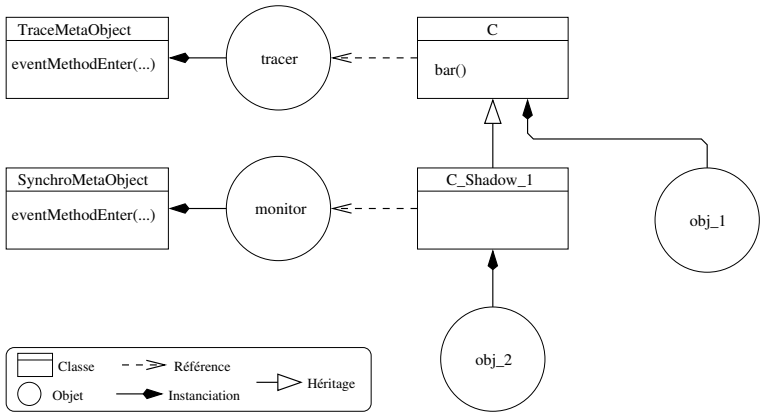

Fig. 12 – *Exemple de partage et de composition de méta-objets dans* MetaXa

La figure 12 donne un example de partage et de composition de méta-objets dans MetaXa. Pour tracer les appels de méthodes reçus par toutes les instances de la classe C par un même méta-objet tracer, ce dernier est attaché à la classe. Ainsi, lorsqu'une instance obj_1 de C reçoit un appel de méthode bar(), l'événement appel de méthode est généré ce qui provoque l'appel de la méthode eventMethodEnter(...) pour le méta-objet tracer. Une trace est ainsi généré. Dans le cas ou l'objet obj_2 reçoit un appel de méthode bar(), l'événement correspondent à cet appel est d'abord transmis au méta-objet monitor. En effet, ce dernier méta-objet a été attaché à l'objet obj_2. Il n'est donc pas partagé par les autres instances de C. Pour réaliser cet attachement de méta-objet, une classe implicite C_Shadow_1 est générée par la machine virtuelle. L'objet obj_2 devient alors instance de cette nouvelle classe. Ainsi, lorsque obj_2 reçoit un appel de méthode, l'événement appel de méthode est transmis au méta-objet monitor qui synchronise les appel de méthodes. Ce méta-objet transmet ensuite l'événement au méta-objet suivant, à savoir tracer qui produit alors une trace.

3.3.2 Guaraná

GUARANÁ [OB98b] [OGB98] [OB98a] [OB99] s'appuie sur une extension de la machine virtuelle libre KAFFE [Kaf99]. Cette extension consiste, en particulier, à étendre la structure de tous les objets par un champ implicite qui peut référencer un méta-objet. Un objet peut donc référencer au plus un méta-objet. Lorsqu'il ne référence aucun méta-objet, les opérations (e.g. appels de méthodes, accès à la structure...) sont exécutées comme dans la machine virtuelle JAVA standard. Dans le cas contraire, le méta-objet contrôle les opérations et éventuellement les transmet à d'autres méta-objets. L'ensemble de ces méta-objets qui contrôlent l'activité d'un même objet est appelé *méta-configuration*.

La méta-configuration d'un objet peut être modifiée dynamiquement. Ce changement est contrôlé par la méta-configuration courante. Ainsi, lorsqu'une application tente d'ajouter ou de retirer un ou plusieurs méta-objets d'une méta-configuration, ce changement peut être rejeté. Ce contrôle de la méta-configuration permet de garantir que les éventuelles règles de sécurité ou de cohérence du méta-niveau ne peuvent être contournées.

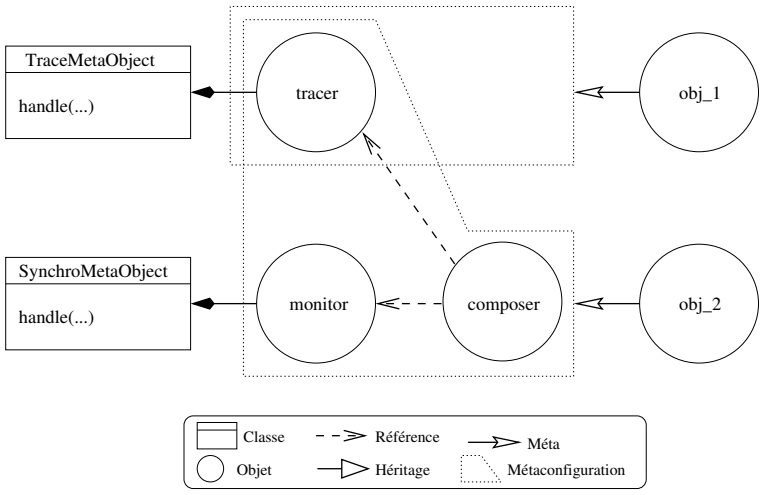

FIG. 13 – *Exemple de paratage et de composition de méta-objets dans* GUARANÁ

Afin d'illustrer le partage et la composition de méta-objets dans GUARANÁ, considèrons l'exemple de la figure 13. Dans cet exemple, la méta-configuration de l'objet obj_1 est réduite au seul méta-objet tracer. Ainsi, les appels de méthodes reçus par obj_1 provoquent l'appel de la méthode handle(...) pour le méta-objet tracer qui produit des traces. Ce méta-objet est paratagé, puisqu'il fait partie également de la méta-configuration de obj_2. Cette dernière méta-configuration, est composée de trois méta-objets. Le premier, appelé composer, gère la composition des deux autres (monitor et tracer). Lorsqu'un appel de méthode est reçu par obj_2, composer en prend le contrôle. Il peut alors délèguer ce contrôle de l'appel de méthode à monitor et tracer.

4 Comparaison des principales extensions réflexives de Java

Nous proposons ici de comparer les extensions réflexives de JAVA présentées dans la section précédente. Cette comparaison est décomposée en deux parties. La première consiste à comparer les modèles réflexifs de chaque extension. La seconde consiste à comparer les mises en œuvre.

4.1 Comparaison des modèles réflexifs

Différents critères de caractérisation des modèles de systèmes réflexifs existent dans la littérature [Fer89] [Ibr90] [Zim96] [BS99b]. Ces critères permettent d'identifier la nature du méta-niveau et les relations qui le lient au niveau de base. Nous les prenons comme point de départ pour comparer les modèles réflexifs des extensions de JAVA étudiées. Ce qui nous intéresse en particulier est de déterminer et de comparer les possibilités de réflexion de comportement de chaque extension. Nous proposons de réaliser cette comparaison en répondant pour chaque extension aux questions suivantes :

1. **Quand a lieu le passage d'un niveau à l'autre?**
   Il s'agit d'identifier le moment de passage d'un niveau à l'autre, qui peut correspondre à *la compilation*, au *chargement* ou à *l'exécution*. Ce passage a lieu à *la compilation* ou au *chargement* dans les *approches statiques*. Les traitements méta sont alors fusionné avec les traitement du niveau de base de sorte à ce qu'il n'y ait pas de passage d'un niveau à l'autre à l'exécution. A l'opposé, le passage d'un niveau à l'autre a lieu à l'exécution dans les *approches dynamiques*. Les opérations du niveau de base sont interceptées ce qui se traduit par le passage du niveau de base au niveau méta. Des traitements méta sont alors réalisés. De tels traitements peuvent éventuellement provoquer l'exécution des opérations du niveau de base interceptées.

2. **Qui réalise le contrôle méta?**
   Il s'agit d'identifier les entités du méta-niveau qui contrôlent, de manière implicite, l'exécution des opérations du niveau de base. Ces entités peuvent être des méta-objets, des groupes de méta-objets ou des éléments du programme (réifications).

3. **Quelle est l'arité du lien méta?**
   Cette question ne concerne que les systèmes comportant des méta-objets. Dans ces systèmes, suivant les modèles, un méta-objet peut ou non contrôler plusieurs objets du niveau de base. Inversement, un même objet du niveau de base peut être contrôlé par un seul ou plusieurs méta-objets.

4. **Quelles opérations provoquent le passage au méta-niveau?**
   Cette question consiste à identifier les principales opérations du niveau de base (telles que l'émission/réception d'appels de méthodes, les lectures/écritures des champs)

contrôlées de manière implicite par le niveau méta. Autrement dit, il s'agit de déterminer les principales méthodes du MOP implicite.

5. **Comment est arrêtée la régression du lien méta?**
   Les extensions réflexives de JAVA introduisent la possibilité de contrôler implicitement l'exécution des opérations du niveau de base. Ce contrôle se traduit par des passages implicites du niveau de base au niveau méta. Comme l'introduction de la réflexion se traduit par une certaine uniformité, il en résulte des régressions potentiellement infinies. Il s'agit ici de déterminer la manière d'arrêter de telles régressions et de garantir la terminaison de l'exécution.

Les tableaux 1 et 2 résument les réponses à ces différentes questions que nous développons dans ce qui suit.

	Réflexion?	Méta?	Arité?
DALANG	à l'exécution	méta-objets	1—1
GUARANÁ	à l'exécution	groupes de méta-objets	n—n
JAVASSIST	à l'exécution	méta-objets	1—1
KAVA	à l'exécution	méta-objets	1—1
METAXA	à l'exécution	chaînes de méta-objets	n—n
OPENJAVA	à la compilation	réifications à la compilation	< *néant* >
PROACTIVE	à l'exécution	méta-objets	1—1
REFLECTIVE JAVA	à l'exécution	méta-objets	1—1

Légende		
Réflexion?	:	Quand a lieu le passage d'un niveau à l'autre?
Méta?	:	Qui réalise le contrôle méta?
Arité?	:	Quelle est l'arité du lien méta? (<*nombre_entités_niveau_de_base*>—<*nombre_entités_niveau_méta*>)

TAB. 1 – *Comparaison des modèles réflexifs des principales extensions réflexives de* JAVA

4.1.1 Quand a lieu le passage d'un niveau à l'autre?

De toutes les extensions réflexives de JAVA étudiées ici, seul OPENJAVA réalise le passage du niveau de base au niveau méta de manière statique, à la compilation. Le code de l'application est alors transformé de sorte à ce que les traitements correspondant au niveau méta soient fusionnés avec ceux du niveau de base.

Dans toutes les autres extensions réflexives de JAVA que nous avons étudiés (DALANG, GUARANÁ, JAVASSIST, KAVA, METAXA, PROACTIVE et REFLECTIVE JAVA) les passages d'un niveau à l'autre sont réalisés dynamiquement, à l'exécution. Suivant les extensions, une partie plus au moins importante des opérations du niveau de base est contrôlée par le méta-niveau (cf. tableau 2). Le nombre de passages d'un niveau à l'autre est d'autant plus importants.

	émission	réception	construct.	champs	création	synchronized	accès tab.	taille tab.
DALANG	–	√	√	–	–	–	–	–
GUARANÁ	–	√	√	√	–	√	√	√
JAVASSIST	–	√[a]	√	–	–	–	–	–
KAVA	√	√	√	√	–	–	–	–
METAXA	√	√	–	√	√	√	–	–
OPENJAVA	√	√	√	√	√	–	√	√
PROACTIVE	–	√	–	–	√	–	–	–
REFLECTIVE JAVA	–	√	√	–	–	–	–	–

Légende		
émission	:	émission d'appels de méthodes d'instances et de classes (*static*)
réception	:	réception d'appels de méthodes d'instances et de classes (*static*)
construct.	:	réception d'appels de constructeurs
champs	:	lecture/écriture de champs d'instances et de classes (*static*)
création	:	création des objets (`new`)
synchronized	:	entrée/sortie de la file d'attente des moniteurs
accès tab.	:	lecture/écriture des éléments des tableaux
taille tab.	:	lecture de la taille des tableaux
√	:	Oui
–	:	Non

TAB. 2 – *Eléments des MOP des principales extensions réflexives de* JAVA

[a] Dans la dernière version (0.3) de JAVASSIST la réception de méthodes qualifiées `static` n'est pas contrôlé, même si la technique d'interception le permet.

Le nombre de passages d'un niveau à l'autre pendant l'exécution influence les performances des applications en terme d'efficacité. Une application est d'autant plus efficace que le nombre de ces passages est petit. De ce point de vue, OPENJAVA est la meilleure des extensions réflexives de JAVA étudiées. OPENJAVA réalise, en effet, les passages d'un niveau à l'autre à la compilation, évitant ainsi de nuire à l'efficacité des applications qu'il permet de développer.

La contrepartie de la réalisation des passages d'un niveau à l'autre statiquement à la compilation est une perte de flexibilité. En effet, cette approche a pour conséquence un couplage fort entre les traitements du niveau de base et ceux du niveau méta. Les codes correspondants aux traitements des niveaux de base et méta sont intimement enchevêtrés. De ce fait, l'approche statique adoptée par OPENJAVA ne permet pas de changer à l'exécution les traitements méta associés à un objet donné du niveau de base pour adapter l'application à un changement de son environnement. A contrario, une telle adaptation est possible avec les extensions qui ont adopté l'approche dynamique. C'est donc sur plan de la flexibilité que l'approche dynamique est plus avantageuse que l'approche statique.

4.1.2 Qui réalise le contrôle méta?

Dans OPENJAVA, le contrôle méta est réalisé par des éléments du programme (classes, méthodes, expressions...) réifiés à la compilation. Chacune de ces entités réifiées produit un code qui représente sa définition initiale transformée pour inclure les traitements méta. Ces traitements méta varient notamment avec la classe de chaque élément du programme.

Dans toutes les autres extensions réflexives de JAVA étudiées, le contrôle des traitements du niveau de base est réalisé à l'aide de méta-objets. Lorsque le contrôle méta est réalisé par des méta-objets, une instance d'une classe C donnée et une instance d'une sous-classe de C peuvent être contrôlées par des méta-objets dont les classes sont différentes. Le contrôle d'une même opération peut être réalisé de manières différentes. Ainsi, l'appel d'une méthode m définie dans la classe C peut être géré différemment suivant que le destinataire de l'appel est une instance de C ou de l'une de ses sous-classes.

Cette possibilité de changer de traitement méta lorsqu'on se déplace sur l'hiérarchie d'héritage est la principale différence avec l'approche de OPENJAVA. En effet, dans OPENJAVA qui s'appuie sur des éléments de programme réifiés, les contrôles méta sont propagés en cas d'héritage de classes. Dans l'exemple de l'appel de méthode m de la classe C, la gestion de l'appel qu'il soit destiné à une instance de C ou de l'une de ses sous-classes est toujours géré par la méthode m. Le traitement méta lié à l'appel de la méthode m se propage donc le long de la hiérarchie d'héritage des classes, ce qui n'est pas le cas avec les méta-objets.

4.1.3 Quelle est l'arité du lien méta?

Contrairement aux autres extensions de JAVA, OPENJAVA ne comporte pas de méta-objets. De ce fait, OPENJAVA n'est pas concerné par cette question.

Dans DALANG, JAVASSIST, KAVA et REFLECTIVE JAVA, chaque classe du niveau de

base est associée à une classe de méta-objets. Toutes les instances d'une même classe du niveau de base sont contrôlées par des méta-objets instances d'une même classe du niveau méta. Un méta-objet est créé et associé à chaque nouvelle instance d'une classe du niveau de base. Ce qui fait que chaque objet est contrôlé par un unique méta-objet et inversement chaque méta-objet contrôle un unique objet.

PROACTIVE propose aussi d'associer chaque objet du niveau de base à un seul méta-objet. Le lien entre objets et méta-objets se fait à la création des objets. Pour créer des objets dans PROACTIVE, il est nécessaire d'utiliser des méthodes statiques d'une classe particulière. L'appel de l'une de ces méthodes provoque la création d'un méta-objet instance d'une classe de méta-objets donnée. Ce méta-objet est créé spécifiquement pour l'objet du niveau de base qu'il contrôle. Il y a donc un méta-objet spécifique par objet du niveau de base.

Dans GUARANÁ, chaque objet est contrôlé par un groupe de méta-objets appelé *méta-configuration*, organisés en arbre. Un méta-objet peut déléguer certains traitements méta à d'autres méta-objets de la méta-configuration. En particulier, un méta-objet peut déléguer le contrôle d'une même opération du niveau de base à plusieurs méta-objets. Cette délégation peut même se faire de manière concurrente. Le méta-objet qui délègue prend alors en charge la gestion de cette concurrence et la résolution des éventuels conflits. Notons qu'un même méta-objet peut appartenir à plusieurs méta-configurations. De ce fait, un même méta-objet peut participer au contrôle de plusieurs objets.

Tout comme dans GUARANÁ, un objet, dans METAXA, peut être contrôlé par plusieurs méta-objets et inversement un méta-objet peut participer au contrôle de plusieurs objets. En revanche, ces méta-objets sont organisés en chaîne et non-pas en arbre. Dès lors, le contrôle d'une même opération du niveau de base ne peut se faire que par un seul méta-objet à la fois. En effet, un méta-objet ne peut déléguer un traitement qu'au méta-objet suivant sur la chaîne de méta-objets.

Parmi les extensions étudiées, certaines proposent d'utiliser des groupes de méta-objets pour contrôler les objets du niveau de base, alors que d'autres proposent de n'utiliser qu'un seul méta-objet. Si l'utilisation d'un seul méta-objet n'exclu pas la délégation entre méta-objets et par conséquent le contrôle d'un objet par plusieurs méta-objets, une différence demeure. Il s'agit de la notion de groupe méta-objets qui offre un cadre clair pour la composition des méta-objets. Les liens entre méta-objets contrôlant un même objets sont en effet définis clairement par les règles de délégation (composition) au sein du groupe. Un tel cadre n'est pas disponible dans les extensions basées sur un méta-objet unique. La gestion de la composition des méta-objet reste alors à la charge des développeurs. Cette charge a néanmoins un côté positif qui est la possibilité de changer facilement le modèle de composition et d'organisation des méta-objets. Un tel changement est difficile à réaliser dans les systèmes utilisant des groupes de méta-objets. En effet, ces systèmes imposent (au moins partiellement) un schéma particulier pour la composition des méta-objets.

4.1.4 Quelles opérations provoquent le passage au méta-niveau?

Hormis OpenJava où les passages du niveau de base au niveau méta sont réalisés à la compilation, les passages d'un niveau à l'autre ont lieu uniquement à l'exécution dans toutes les autres extensions réflexives de Java étudiées. Indépendamment du fait qu'ils aient lieu à la compilation ou à l'exécution, les passages d'un niveau à l'autre correspondent aux opérations du niveau de base interceptées. La variété des opérations interceptées est reflétés par le MOP. En effet, la multiplicité des passages du niveau de base au niveau méta dépend directement de la richesse du MOP.

Le tableau 2 (page 30) donne les principaux éléments des MOPs des extensions de Java étudiées. Il montre les opération dont le contrôle parait important aux yeux des concepteurs de chaque extension. La majorité s'accorde sur la nécessité du contrôle de l'initialisation des objets (appel de constructeurs), les accès aux champs et la les appels de méthodes. Cet accord reflète d'une part l'importance de ces opérations et d'autre part la facilité de réalisation des interceptions d'autre part.

4.1.5 Comment est arrêtée la régression du lien méta?

Dans Dalang, Kava et Reflective Java, les classes dont les instances doivent être contrôlées par des méta-objets sont explicitement désignées. Les noms de tels classes et les classes de méta-objets associés apparaissent explicitement dans un fichier d'initialisation qui peut être lu lors du chargement des classes. Le byte-code des classes ainsi désignées est transformé pour introduire les intercepteurs permettant le passage d'un niveau à l'autre. Les byte-codes des autres classes ne sont pas transformées. En particulier, les byte-codes des classes de méta-objets terminaux ne sont pas transformés. Les opérations impliquant les instances de tels classes sont donc directement exécutées par la machine virtuelle Java arrêtant ainsi la régression. Dans les autres extensions étudiées, la régression du lien méta est arrêtée de manière analogue.

Dans Proactive les objets qui doivent être contrôlés par des méta-objets sont créés à l'aide d'un protocole particulier qui introduit des encapsulateurs. Pour arrêter la régression, les méta-objets terminaux sont créés à l'aide du protocole de création standard de Java (i.e. utilisation du `new`).

Dans Guaraná et MetaXa, chaque objet peut référencer un méta-objet. L'arrêt de la régression du lien méta se fait en intervenant sur cette référence. Si cette référence est nulle (`null`) pour un objet `o` donné, i.e. si aucun méta-objet n'est associé à `o`, les opérations impliquant `o` sont directement exécutées par la machine virtuelle, sans aucun contrôle méta.

En résumé, deux solutions ont été adoptées dans les différentes extensions étudiées pour arrêter la régression du lien méta. La première solution utilisée dans Dalang, Kava, Reflective Java et Proactive consiste à ne pas introduire d'intercepteurs pour certains méta-objets. Ainsi, les opérations réalisées par ces méta-objets (dit *terminaux*) ne sont contrôlées par aucun méta-méta-objet et sont directement exécutées par la machine virtuelle. La seconde solution utilisée dans Guaraná et MetaXa consiste à avoir un mécanisme d'interception *actif* qui ne transmet au méta-niveau le contrôle des opérations

du niveau de base que sous certaines condition. Aussi-bien dans GUARANÁ que dans METAXA, cette condition est l'existance d'un méta-objet associé à l'objet du niveau de base dont l'opération est interceptée. Quand cette condition est vérifiée, l'opération interceptée est exécutée par la machine virtuelle. La machine virtuelle peut alors être vue comme un méta-objet implicite par défaut. Comparée à la première solution, l'arrêt de la régression du lien méta par l'utilisation d'intercepteur actifs introduit des interceptions superflues. Cependant, l'utilistaion d'intercepteurs actifs offre plus de flexibilité puisque n'importe quel objet peut être associé à un méta-objet. Alors qu'avec la solution qui consiste à n'introduire des intercepteurs que pour les objets qui sont associés à des méta-objets, il n'est pas aisé d'adapter dynamiquement une application et contrôler à l'exécution les opérations d'objets initialement sans méta-objet associé.

Comme OPENJAVA ne dispose pas de méta-objets, le problème de régression du lien méta ne se pose pas. Cependant, OPENJAVA s'appuie sur la réification des éléments des programmes ce qui peut introduire d'autres sortes de régressions telle que la régression du lien d'instanciation. Pour arrêter ces régressions, OPENJAVA s'appuie sur une bibliothèque de classes dont les définitions ne sont pas réifiées à la compilation. Par exemple, les classes sont instances d'autres classes (des *métaclasses*), d'où une régression du lien d'instanciation. Pour arrêter cette régression, la racine du lien d'instanciation dans OPENJAVA est une classe qui n'est pas réifiée à la compilation.

4.2 Comparaison des mises en œuvre

Afin de comparer les mises en œuvre des différentes extensions réflexives de JAVA, nous identifions pour chaque extension la technique d'implémentation utilisée et nous évaluons la qualité de la technique. Pour ce faire, nous proposons de répondre aux questions suivantes pour chaque extension :

1. **Comment sont introduits les intercepteurs?**
   Il s'agit d'identifier la technique permettant d'intercepter les opérations du niveau de base pour permettre d'introduire le contrôle du méta-niveau.

2. **Quand sont introduits les intercepteurs?**
   Il s'agit de déterminer le moment d'introduction des intercepteurs. Cette introduction peut se faire à la compilation, ou au chargement, ou encore à l'exécution.

3. **Quel est le degré de fiabilité d'interception?**
   Il s'agit d'identifier les éventuelles failles qui peuvent conduire à ne pas intercepter certaines opérations, ce qui a pour conséquence de ne pas exécuter certains traitements méta. De tels failles peuvent consister en l'impossibiliter d'introduire certains intercepteurs ou en l'impossibilité d'intercepter certaines opérations (intercepteur contourné).

4. **Quelles sont les possibilités d'adaptabilité dynamique?**
   Il s'agit d'évaluer la flexibilité d'une approche et plus précisément la possibilité de changer, à l'exécution, les traitements méta associés à un objet donné.

5. **Quel est le degré de difficulté de l'implémentation ?**
   Il s'agit d'identifier les éléments permettant d'avoir une idée de la difficulté de la mise en œuvre.

6. **Quel est le degré d'efficacité ?**
   Il s'agit de déterminer les différentes causes de pertes de performances — comparé à JAVA standard — afin d'avoir une idée de l'efficacité.

Les questions 1 à 3 permettent de caractériser la technique d'implémentation utilisée. La qualité de cette technique est évaluée grâce aux questions restantes (4 à 6).

4.2.1 Comment sont introduits les intercepteurs ?

A l'exception de OPENJAVA qui fusionne à la compilation les traitements méta avec les traitements du niveau de base, toutes les extensions réflexives de JAVA nécessitent d'avoir des intercepteurs pour capturer à l'exécution les opérations du niveau de base et réaliser les traitements méta. Pour ce faire, l'une des quatre techniques suivante est utilisée :

- *l'encapsulation d'objets*,
- *l'encapsulation de méthodes*,
- *l'encapsulation d'expressions*,
- *la modification de la machine virtuelle.*

Encapsulation d'objets : Cette technique initialement décrite par Geoffrey Pascœ [Pas86] est exploitée dans DALANG et dans PROACTIVE. Elle consiste à intercaler un objet encapsulateur (*wrapper*) `e` entre un objet `o` et ses clients. Un objet client `c` référence donc l'encapsulateur `e` en lieu et place de l'objet `o`. L'encapsulateur `e` joue alors le rôle d'intermédiaire entre le client `c` et l'objet `o`. Ce niveau d'indirection permet d'introduire des traitements méta, avant de transférer les appels de méthodes à l'objet destinataire `o`.

DALANG et PROACTIVE mettent en œuvre cette approche de deux manières différentes. Dans DALANG l'encapsulateur joue également le rôle de méta-objet, alors que dans PROACTIVE le méta-objet et l'encapsulateur sont des objets distincts.

Encapsulation de méthodes : L'encapsulation de méthodes consiste à masquer les méthodes originales d'une classe par de nouvelles méthodes de même signature [BFJR98]. Ces nouvelles méthodes "référencent" les méta-objets, permettant ainsi le contrôle des appels de méthodes.

Cette approche a été mise en œuvre de deux manières différentes dans JAVASSIST et REFLECTIVE JAVA. Dans JAVASSIST, les méthodes à masquer sont renommées. Leurs noms originaux sont attribués aux méthodes encapsulatrices. Cette opération nécessite la transformation des définitions des classes. La mise en œuvre de REFLECTIVE JAVA est plus simple puisqu'elle utilise uniquement l'héritage. Une sous-classe est définie pour

chaque classe dont les instances doivent être associées à des méta-objets. Cette sous-classe redéfinit les méthodes dont l'appel doit être capturé. Si cette solution semble plus simple à implémenter que celle de JAVASSIST, elle nécessite cependant de modifier l'application afin d'instancier les sous-classes capturant les appels de méthodes au lieu d'instancier les classes originales.

Encapsulation d'expressions : Cette approche exploitée dans KAVA consiste à transformer les programmes (KAVA opère une transformation de byte-code) de sorte à ce que chaque expression ou groupe d'expressions qui doivent être contrôlées soient encapsulées dans une autre expression. L'expression encapsulatrice donne le contrôle à un méta-objet avant et après l'évaluation des expressions encapsulées. L'expression encapsulatrice inclut également la gestion d'une exception particulière permettant au méta-objet d'interdire l'exécution des expressions encapsulées.

Modification de la machine virtuelle : L'approche adoptée par METAXA et GUARANÁ consiste à modifier la machine virtuelle pour y inclure le support de la réflexion. Les mécanismes de représentation des éléments de programme et de leur interprétation sont étendus pour permettre à des méta-objets de contrôler différentes opérations du niveau de base. A titre d'exemple, dans GUARANÁ, la structure de tous les objets est étendue pour inclure un champ qui référence une méta-configuration. Dans METAXA, la représentation des classes est modifié pour servir de support du lien méta. Comme ces modifications sont intégrées à la machine virtuelle, les extensions réflexives de JAVA basées sur cette technique gagnent en efficacité par rapport aux autres extensions de JAVA. En revanche, elle perdent sur le plan de la portabilité puisque la réalisation des traitements méta est conditionnée à l'utilisation de machines virtuelles spécifiques.

4.2.2 Quand sont introduits les intercepteurs ?

L'introduction d'intercepteurs peut se faire à trois moments différents : à la compilation, au chargement ou à l'exécution.

- **Introduction des intercepteurs à la compilation :** L'introduction des intercepteurs à la compilation peut se faire soit au niveau du code source comme c'est le cas pour REFLECTIVE JAVA, soit au niveau du byte-code comme c'est le cas pour DALANG, JAVASSIST et KAVA. L'approche basée sur la manipulation de byte-code présente l'avantage de s'affranchir du code source qui n'est pas toujours disponible. Cet avantage est négligeable puisque la décompilation d'un byte-code pour obtenir du code source JAVA est quasi-immédiate.

- **Introduction des intercepteurs au chargement :** La transformation de byte-code peut se faire également lors du chargement des classes. DALANG, JAVASSIST et KAVA ont cette possibilité d'introduire les intercepteurs lors du chargement de classes, en plus de la possibilité de le faire à la compilation. Par rapport à l'introduction

des intercepteurs à la compilation, cette solution présente l'avantage de permettre l'introduction des intercepteurs dans un code mobile, tel que les applets.

- **Introduction des intercepteurs à l'exécution :** L'introduction des intercepteurs à l'exécution est réalisée dans GUARANÁ et METAXA en utilisant une machine virtuelle supportant la réflexion. Dans le cas de PROACTIVE, elle se fait à la création des objets. Notons que l'approche de PROACTIVE ne peut s'appliquer à un programme existant sans le modifier pour remplacer les créations d'objets à l'aide du new de JAVA par des appels du protocole de création de PROACTIVE. Cependant, aucun outil n'est fourni avec PROACTIVE pour réaliser cette opération.

4.2.3 Quel est le degré de fiabilité d'interception ?

Suivant les techniques d'interception utilisées, certaines opérations peuvent échapper au contrôle du méta-niveau. Ce problème peut apparaître soit lors de l'introduction des intercepteurs (impossibilité d'introduire un intercepteur) soit lors de l'exécution des opérations du niveau de base (impossibilité d'intercepter certaines opérations).

Dans les extensions de JAVA basées sur l'approche par encapsulation des objets (DALANG et PROACTIVE), une source de défaillance d'interception est la possibilité d'appeler des méthodes directement sur l'objet encapsulé, sans passer par l'encapsulateur. Il n'est pas possible d'intercepter de tels appels. C'est le problème de l'identité (*the Self problem*) identifié par Henry Lieberman [Lie86]. Ce problème apparaît lorsqu'un objet obtient la référence de l'objet encapsulé ou lorsque ce dernier s'auto-référence (utilisation du this ou du super).

L'introduction des intercepteurs dans REFLECTIVE JAVA s'appuie sur l'encapsulation de méthodes en redéfinissant celles-ci dans une sous-classe. Cette approche pose problème lorsque la classe à sous-classer est finale ou lorsqu'elle définit des méthodes privées ou finales (private ou final). Dans de tels cas, il n'est pas possible d'introduire des intercepteurs. Ce même problème se pose également avec PROACTIVE où les classes d'encapsulateurs héritent des classes des objets encapsulés.

Avec les extensions de JAVA introduisant les intercepteurs lors du chargement des classes (DALANG, JAVASSIST et KAVA), il est nécessaire d'utiliser un chargeur de classe particulier. Or, JAVA n'offre pas de mécanisme permettant de forcer l'utilisation d'un unique chargeur de classe. De ce fait, une classe peut être chargée avec un autre chargeur que celui qui introduit les intercepteurs. La transformation de byte-code permettant l'introduction d'intercepteurs n'a pas alors lieu.

En dernier lieu, la technique d'interception par modification de la machine virtuelle, adoptée par GUARANÁ et METAXA ne semble pas poser de problèmes de fiabilité d'interception. La machine virtuelle JAVA étant adaptée pour réaliser les passages d'un niveau à l'autre.

Note : il n'est pas question de OPENJAVA ici car cette extension de JAVA n'utilise pas d'intercepteurs.

4.2.4 Quelles sont les possibilités d'adaptabilité dynamique ?

Hormis DALANG et OPENJAVA, toutes les extensions réflexives de JAVA que nous avons étudiées (GUARANÁ, JAVASSIST, KAVA, METAXA, PROACTIVE et REFLECTIVE JAVA) permettent facilement de changer dynamiquement les traitements méta associés à un objet. Dans ces extensions, un ou plusieurs méta-objets (suivant l'arité du lien méta) peuvent être remplacés par d'autres méta-objets. Ce changement est délicat dans DALANG car le méta-objet joue le rôle d'encapsulateur. Le remplacement de celui-ci nécessite de mettre à jour tous les objets qui le référencent. Dans OPENJAVA, cette éventualité ne peut même pas être envisagée, puisque le code correspondant aux traitements méta est intimement enchevêtré avec le code correspondant aux traitements de base.

Notons que l'adaptabilité dynamique dans JAVASSIST, KAVA, PROACTIVE et REFLECTIVE JAVA nécessite d'être prévue à l'avance pour disposer de tous les intercepteurs nécessaires. En effet, dans JAVASSIST, KAVA et REFLECTIVE JAVA les intercepteurs ne peuvent être introduits qu'à la compilation ou aux chargement des classes. Dans PROACTIVE, l'introduction des intercepteurs nécessite l'utilisation d'un protocole de création particulier. Il est donc nécessaire d'identifier les objets qui doivent être associés à des méta-objets, au moment de l'implémentation.

4.2.5 Quel est le degré de difficulté de l'implémentation ?

Parmi les extensions réflexives de JAVA étudiées, PROACTIVE nous semble être la plus simple à implémenter. En effet, l'introduction des encapsulateurs se fait par l'utilisation d'un protocole de création particulier.

Si DALANG s'appuie également sur l'utilisation d'encapsulateurs d'objets, son implémentation est plus compliquée que celle de PROACTIVE. Cette complexité provient du fait que la création de classes d'encapsulateurs nécessite de manipuler le byte-code des classes chargées. Cette manipulation se limite néanmoins à renommer les classes chargées puisque leurs noms originaux sont attribués aux classes d'encapsulateurs.

JAVASSIST et KAVA s'appuient également sur la manipulation de byte-code. Cependant, l'implémentation de ces deux extensions de JAVA est nettement plus complexe que celle de DALANG, car JAVASSIST et KAVA réalisent des transformations de byte-code plus complexes que celles réalisées dans DALANG.

Comme le MOP de KAVA est plus riche que celui de JAVASSIST et que KAVA utilise des intercepteurs d'un grain plus fin que ceux utilisés dans JAVASSIST, il en résulte que KAVA nécessite une plus grande variété de transformations de byte-code. La richesse du MOP de KAVA fait qu'il nécessite que certaines transformations soient appliquées à plusieurs classes. En particulier, le MOP de KAVA permet de contrôler les accès en lecture/écriture des champs. L'interception des accès à des champs visibles en dehors d'une classe (champs qualifiés public ou sans qualificatif, i.e. visibles dans le paquetage) nécessite de modifier non-seulement le byte-code de la classe où sont définis les champs, mais également le byte-code des autres classes qui accèdent directement à ces champs. Afin de déterminer la totalité

de ces classes, il est nécessaire de faire une analyse de type très poussée[14] et introduire éventuellement des tests de type à l'exécution.

Ce besoin de transformer le code de différentes classes apparaît également dans OpenJava. Comme OpenJava opère sur du code source Java, il semble a priori plus facile à implémentater que Kava ou que Javassist qui opèrent sur du byte-code. Comparé à Reflective Java qui s'appuie également sur l'utilisation d'un pré-processeur, OpenJava parait quelque peu plus complexe à implèmenter. En effet, le pré-processeur de Reflective Java permet d'analyser des scripts écrits dans un langage simple afin de générer des classes d'encapsulateurs adéquates. Le pré-processeur de OpenJava, quant à lui, doit supporter les différentes subtilités du langage Java. Même si cette différence peu s'estomper par l'utilisation d'outils tels que JavaCC, la richesse du MOP de OpenJava rend son implémentation plus difficile que celle de Reflective Java. En effet, dans OpenJava, les traitements méta associés à une classe donnée peuvent nécessiter de transformer le code de différentes classes. C'est le cas par exemple pour réaliser des traitements méta lors de la lecture d'un champ publique. D'où le besoin d'une analyse de type dont l'implémentation est lourde.

De telles analyses ne sont pas nécessaires dans Proactive. Son MOP, moins riche que le MOP de OpenJava permet essentiellement le contrôle des appels de méthodes reçus. L'implémentation de Proactive est fondée sur la génération de classes d'encapsulateurs. Elle est donc plus facile que celle de OpenJava. Comparée à l'implémentation de Reflective Java, cette implémentation est de difficulté équivalente.

Les implémentations de Guaraná et de MetaXa nécessitent d'intervenir au niveau de la machine virtuelle, opération a priori plus lourde que la réalisation d'outils de transformation de code source ou de byte-code. Cette délicate opération, est d'autant plus lourde que l'intervention est poussée. Cette lourdeur concerne en particulier MetaXa. En effet, dans MetaXa l'intervention sur la machine virtuelle inclut la modification du compilateur à la volée (*Just In Time Compiler*) pour prendre en compte l'introduction de la réflexion.

4.2.6 Quel est le degré d'efficacité ?

Les problèmes d'efficacité qui peuvent être rencontrés dans les systèmes réflexifs sont dus aux passages d'un niveau à l'autre. Afin de supprimer ce surcoût, l'approche de la réflexion à la compilation adoptée par OpenJava évite ces passages à l'exécution, mais au détriment de l'adaptabilité.

Dans Guaraná et MetaXa, le surcoût dû aux passages d'un niveau à l'autre est réduit en chargeant la machine virtuelle de le réaliser. De plus, MetaXa utilise un compilateur en code natif (*JIT Compiler*) qui prend en compte le support de la réflexion par la machine virtuelle (comme par exemple les modifications de la représentation des classes et le chaînage de méta-objet), réduisant ainsi d'avantage le coût de la réflexion.

Dans les autres extensions de Java étudiées (Dalang, Javassist, Kava, Proactive et Reflective Java), l'introduction de la réflexion nécessite de transformer le code source

14. Il n'existe pas encore de version stable de Kava. Dans la version temporaire que nous avons évalué (0.7), ce problème reste encore mal géré.

ou le byte-code pour introduire des intercepteurs. Afin de limiter le coût de la réflexion, seuls les intercepteurs nécessaires sont introduits, afin d'éviter le passage systématique du niveau de base au niveau méta.

Cependant, avec JAVASSIST et KAVA, il existe différents cas où des passages inutiles d'un niveau à l'autre ont lieu. Par exemple, le passage au méta-niveau a lieu pour toutes les instances d'une même classe, même celles qui n'ont pas besoin d'un traitement méta particulier. En effet, les interceptions sont réalisées au niveau de la classe. Lorsqu'il s'agit d'intercepter les appels des méthodes définies dans les superclasses, et en particulier les appels destinés à la pseudo-variables `super`, il est nécessaire d'introduire des intercepteurs (encapsulateurs de méthodes) dans les superclasses. De ce fait, les appels de ces méthodes pour les instances des superclasses provoquent des interceptions qui peuvent être inutiles.

Une autre source de perte d'efficacité dans DALANG, JAVASSIST, KAVA et PROACTIVE est due à la génération/manipulation de classes lors du chargement ou à l'exécution. Dans le cas de DALANG, JAVASSIST et KAVA, le chargement des classes est ralenti du fait des manipulations de byte-code réalisées. Dans PROACTIVE, le ralentissement se produit à la création des objets lorsqu'il est nécessaire de générer et de compiler une classe d'encapsulateur, lors de la première instanciation d'une classe du niveau de base.

KAVA et OPENJAVA ont en commun une autre source d'inefficacité due à leurs MOP élaborés. En effet, ces deux extensions réflexives de JAVA proposent de contrôler entre autre les accès en lecture/écriture des champs. L'interception de ces accès aux champs, en particulier pour les champs qualifiés `public` ou ceux sans qualificatif (i.e. visibles dans le paquetage) nécessite de connaître la classe des objets accèdés. L'analyse statique ne permet pas toujours d'obtenir cette information. D'où le besoin d'introduire des tests à l'exécution (utilisation de la méthode `getClass()`), source de ralentissement.

5 Conclusion

En donnant accès aux mécanismes d'exécution des applications, les langages réflexifs permettent de construire des systèmes et des applications extensibles et adaptables [KP94] [Kic96]. Cette possibilité se traduit par deux niveaux de programmation clairement séparés : un niveau de base et un niveau méta. Le niveau de base correspond à la définition des services offerts (par exemple le retrait ou le dépôt sur un compte bancaire). Le niveau méta correspond quant à lui aux mécanismes d'exécution, c'est-à-dire à la manière d'exécuter les services offert (par exemple l'introduction de communications distantes ou la gestion de la persistance).

Depuis la version 1.1 du JDK, le langage JAVA dispose de capacités réflexives qui se traduisent notamment par une API dédiée à la réflexion. Cependant, si ces possibilités réflexives ont été progressivement étendues au fil des versions, elles restent néanmoins limitées. En effet, elles permettent essentiellement de manipuler la structure des objets (réflexion de structure) sans pour autant modifier la manière d'exécuter leur comportement (réflexion de comportement). A titre d'exemple, si JAVA permet d'obtenir à l'exécution les méthodes et les champs définis dans une classe, il ne permet pas de modifier la résolution des appels de méthodes ou la manière d'accéder aux champs.

Afin de compenser ces limitations, différentes extensions réflexives de JAVA ont vu le jour [KG96] [CT98] [CKV98] [Zhi98] [WS99] [OB99]. La majorité de ces extension a adopté une approche dynamique qui fait que la manière d'exécuter les services offerts par une application est déterminée à l'exécution. A l'opposé, l'adoption de l'approche statique est plus marginale. En effet, parmi les extensions réflexives de JAVA étudiées, une seule d'entre elles permet de lier les niveaux de base (services offerts) et méta (mécanismes d'exécution) statiquement à la compilation. Comparée à l'approche dynamique, l'approche statique offre l'avantage d'une grande efficacité. Cependant, cet avantage est obtenu au détriment de la flexibilité. En effet, contrairement à l'approche dynamique, l'approche statique ne permet pas de changer dynamiquement les mécanismes d'exécution. D'où l'impossibilité d'adapter les mécanismes d'exécution aux changements de l'environnement d'exécution.

Du point de vue implémentation, les extensions réflexives de JAVA ont adopté différentes techniques. Certaines solutions tel que l'utilisation d'encapsulateurs ne permettent de contrôler que les mécanismes d'appel de méthodes et de constructeurs. D'autres plus élaborées tel que la modification de la machine virtuelle permettent de contrôler de nombreux mécanismes d'exécution comme les accès aux tableaux et le verrouillage des threads (synchronized).

Dans le cadre du présent projet, nous nous intéressons à l'exploitation de la réflexion comme infrastructure à la mobilité forte et à l'adaptabilité. L'objectif visé est d'aboutir à une extension réflexive de JAVA supportant la mobilité forte et l'adaptabilité. La mobilité forte sous-entend entre autres la migration du contexte d'exécution. Afin de définir une extension réflexive de JAVA permettant de réaliser cette mobilité, il est nécessaire d'utiliser une technique d'implémentation qui permet de contrôler finement l'exécution des application. Cette extension pourrait en particulier donner accès la pile d'exécution. Par ailleurs, le besoin d'adaptabilité impose d'adopter une approche dynamique afin de permettre le

changement des mécanismes d'exécution à l'exécution.

Remerciements

L'auteur souhaite remercier Messieurs Mario Südholt, Thomas Ledoux et Rémi Douence pour leurs commentaires sur les différentes versions de ce document, ainsi que pour les différentes discussions autour de la réflexion et du langage JAVA.

Références

[AG96] Ken Arnold and James Gosling. *Le Langage Java.* The Java Series... from the Source. International Thomson Publishing France, Paris, 1996. Traduction de Serge Chaumette et Alain Miniussi.

[BC89] Jean-Pierre Briot and Pierre Cointe. Programming with Explicit Metaclasses in Smalltalk. In *Proceedings of OOPSLA'89*, pages 419–431, New Orleans, Louisiana, USA, 1989. ACM.

[Ben98] Brent W. Benson. Reflection: How to eat your own dog food. *Sigplan notices*, 33(12):16–19, December 1998.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the Rescue. In *Proceedings of ECOOP'98*, July 1998.

[BGW93] D.G. Bobrow, R.G. Gabriel, and J.L. White. *Object Oriented Programming - The CLOS perspective*, chapter CLOS in Context - The Shape of the Design Space. In Object Oriented Programming. MIT Press, 1993.

[BS99a] M. N. Bouraqadi-Saâdani. Un cadre réflexif pour la programmation par aspects. In *Proceedings of LMO'99*, Villefranche sur Mer - France, January 1999. Hermès.

[BS99b] M. N. Bouraqadi-Saâdani. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects.* Thèse de doctorat, Université de Nantes, Nantes, France, July 1999.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceeding of OOPSLA'95*, pages 285–299, 1995.

[Chi98] Shigeru Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *Proceedings of the OOPSLA'99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[Chi99] Shigeru Chiba. http://www.hlla.is.tsukuba.ac.jp/~chiba/javassist/index.html, 1999.

[CKV98] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, pages 1043–1061, September–November 1998. Published by Wiley and Sons Ltd.

[Coi88] Pierre Cointe. A Tutorial Introduction to Metaclass Architecture as Provided by Class Oriented Languages. In *Proceedings of International Conference on Fifth Generation Computer Systems, ICOT*, Tokyo, Japan, November 1988.

[CT98] Shigeru Chiba and Michiaki Tatsubori. Yet Another java.lang.Class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[DM95] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *Workshop of IJCAI'95 on Reflection and Meta-Level Architecture and their Application in A.I.*, pages 29–38, August 1995.

[DS99] R. Douence and M. Südholt. The next 700 reflective object-oriented languages. Technical Report 99-1-INFO, École des mines de Nantes, 1999.

[Eck98] Bruce Eckel. *Thinking in Java.* Prentice Hall, 1998. http://www.BruceEckel.-com.

[FCDR95] Ira R. Forman, Michael H. Conner, Scott Danforth, and Larry K. Raper. Release-to-Release Binary Compatiblity in SOM. In *Proceedings of OOPSLA'95*, October 1995.

[Fer89] Jacques Ferber. Computational Reflection in Class-Based Object-Oriented Languages. In *Proceedings of OOPSLA'89*, pages 317–326, New Orleans, Louisiana, USA, October 1989. ACM.

[GK98] Michael Golm and Jürgen Kleinöder. MetaXa and the Future of Reflection. In *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[GK99] Michael Golm and Jürgen Kleinöder. Jumping to the Meta Level: Behavioral Reflection Can Be Fast and Flexible. In Pierre Cointe, editor, *Proceedings of Reflection'99*, number 1616 in LNCS, pages 22–39, Saint-Malo, France, July 1999. Springer.

[Gol97] Michael Golm. Design and Implementation of a Meta Architecture for Java. Master's thesis, Institut für Mathematische Maschinen und Datenvararbeitung, Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany, January 1997.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80*, volume The Language and its implementation. Addison-Wesley, 1983.

[Gre99] Dale Green. The Reflection API. http://java.sun.com/docs/books/tutorial/-reflect/index.html, 1999.

[Ham97] Graham Hamilton, editor. *JavaBeans*. Sun, July 1997. Specifications.

[Ibr90] Mamdouh Ibrahim. Reflection and Metalevel Architectures in Object-Oriented Programming. Addendum to the Proceedings of the ECOOP/OOPSLA'90 Proceedings, 1990.

[Jaw98] Jamie Jaworski. *Java 1.2 Unleashed*. SAMS, August 1998.

[Kaf99] The Kaffe Homepage. http://www.kaffe.org, 1999. An open source implementation of a Java virtual machine and class libraries.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, Massachusetts, USA, 1989.

[KG96] Jürgen Kleinöder and Michael Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. Proceedings of the International Workshop on Object Orientation in Operating Systems, IWOOOS'96, October 1996. Seatle, Washington.

[Kic96] Gregor Kiczales. Beyond The Black Box: Open Implementation. *IEEE Software*, January 1996.

[KP94] Gregor Kiczales and Andreas Pæpcke. Open Implementations and Metaobject Protocols. Expanded tutorial notes, 1994.

[LC96] Thomas Ledoux and Pierre Cointe. Explicit Metaclasses as a Tool for Improving the Design of Class Libraries. In *Proceedings of ISOTAS'96, LNCS 1049*, pages 38–55, Kanazawa, Japan, March 1996. Springer-Verlag.

[Led98] Thomas Ledoux. Adaptabilité dynamique des aspects pour la construction d'applications réparties ouvertes. In *Proceedings of Colloque International sur les NOuvelles TEchnologies de la REpartition, NOTERE'98, Montréal*, October 1998. (in French).

[Lie86] Henry Lierberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of OOPSLA'86*. ACM, 1986.

[Mae87a] Pattie Maes. *Computational Reflection*. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit, Brussel, Belgium, 1987.

[Mae87b] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[Mal97] Jacques Malenfant. *Abstraction, encapsulation et réfleion dans les langages à prototypes.* Thèse d'habilitation à diriger des recherches, Université de Nantes - Faculté des Sciences et des Techniques, April 1997.

[McA95] Jeff McAffer. Meta-level Programming with CodA. In *Proceedings of ECOOP'95*, volume LNCS 952, pages 190–214. Springer-Verlag, 1995.

[Mul95] Philippe Mulet. *Réflextion et Langages à Prototypes.* Thèse de doctorat, Université de Nantes. Faculté des Sciences et Techniques, july 1995.

[OB98a] Alexandre Oliva and Luiz Eduardo Buzato. An Overview of MOLDS: A Meta-Object Library for Distributed Systems. Technical Report IC-98-15, UNICAMP, Universidade Estadual de Campinas, Brasil, September 1998.

[OB98b] Alexandre Oliva and Luiz Eduardo Buzato. Composition of Meta-Objects in Guarana. In *Proceedings of the OOPSLA'99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[OB99] Alexandre Oliva and Luiz Eduardo Buzato. Composition of Meta-Objects in Guarana. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems (COOTS'99)*, San Diego, California, USA, May 1999.

[OGB98] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduarado Buzato. The Reflective Architecture of Guaraná. Technical Report IC-98-14, UNICAMP, Universidade Estadual de Campinas, Brasil, September 1998.

[Pas86] Geoffrey A. Pascœ. Encapsulators: A New Software Paradigm in Smalltalk-80. In *Proceedings of OOPSLA'86.* ACM, 1986.

[Riv97] Fred Rivard. *Évolution du Comportement des Objets dans les Langages à Classes Réflexifs.* Thèse de doctorat, Université de Nantes, June 1997. (In french).

[Smi84] Brian C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, POPL'84*, pages 23–35, January 1984.

[Smi90] Brian C. Smith. What Do You Mean, Meta? In *Workshop on Reflection and Metalevel Architechtures in OO Programming, ECOOP/OOPSLA'90*, Ottawa, Ontario, Canda, October 1990.

[Sun98a] Sun. Java Object Serialization Specification, November 1998.

[Sun98b] Sun. Java Remote Method Invocation Specification, October 1998.

[Sun99] Sun. Dynamic Proxy Classes. http://java.sun.com/products/jdk/1.3/docs/-guide/reflection/proxy.htm, 1999.

[Tas97] Jérôme Tassel. Quality of Service (QoS) adaption using Reflective Java. Master's thesis, University of Kent at Canterbury, September 1997.

[Tat99] Michiaki Tatsubori. An Extension Mechanism for the Java Language. Master's thesis, Granduate School of Engineering, University of Tsukuba, Tsukuba, Japan, February 1999.

[Tre98] Paul Tremblett. Java Reflection: Not just for Tool Developpers. *Dr. Dobb's*, January 1998.

[Vay97] Julien Vayssière. Programmation parallèle et distribuée en Java. Conception et implémentation de Java//. Dea, Université de Nice - Sophia Antipolis, Septembre 1997.

[WS97] Zhixue Wu and Scarlet Schwiderski. Reflective Java: Making Java Even More Flexible. Technical Report APM.1936.02, ANSA, Poseidon House, Castle Park, Cambridge CB3 0RD, United Kingdom, February 1997.

[WS98] Ian Welch and Robert Stroud. Dalang - A Reflective Java Extension. In *Proceedings of the OOPSLA'99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[WS99] Ian Welch and Robert Stroud. From Dalang to Kava - the Evolution of a Reflective Java Extension. In Pierre Cointe, editor, *Proceedings of Reflection'99*, number 1616 in Lecture Notes in Computer Science, pages 2–21, Saint-Malo, France, July 1999. Springer.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OOPSLA'88*. ACM, 1988.

[Yok92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of OOPSLA'92*. ACM, 1992.

[Zhi98] Zhixue Wu. Reflective Java and a Reflective Component-Based Transaction Architecture. In *Proceedings of the OOPSLA'99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[Zim96] Chris Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter Metalevels, MOPs and What the Fuzz is All About, pages 3–29. CRC Press, 1996.