PROJET RNTL ARCAD*
D1.4 - Coût et optimisation

Jacques Noyé, Eric Bruneton, Thierry Coupaye,
Daniel Hagimont, Jean-Marc Menaud, Eric Tanter

31 mars 2004

Résumé

La structuration des applications sous la forme d'architectures à base de composants fournit un très bon support d'implémentation de systèmes adaptables. L'existence de *points d'adaptations* dans ces architectures conduit toutefois à l'introduction d'une couche d'interprétation, composée d'indirections et de tests, qui se révèle être la source de surcoûts significatifs en temps et en mémoire. Ce rapport présente un certain nombre de travaux, menés au sein du projet, qui se sont attachés à mettre en évidence ces surcoûts et à voir comment les réduire à l'aide de techniques d'optimisations appropriées.

1 Introduction

Nous avons vu dans un rapport précédent [11] que la structuration des applications sous la forme d'architectures à base de composants fournit un très bon support d'implémentation de systèmes adaptables. En effet, la modularité de ces architectures permet de mieux localiser les adaptations, au niveau d'un composant ou d'un ensemble de composants, au niveau de l'implémentation ou de l'architecture de ces composants.

Ces adaptations peuvent intervenir à différents moments du cycle de vie d'une application. Le maximum de flexibilité est obtenu en supposant qu'elles interviennent lors de l'exécution d'une application. Ceci correspond à l'hypothèse fréquente qui consiste à dire que à la fois composants et architectures ont une existence à l'exécution. Cette hypothèse se justifie par des contraintes applicatives et des raisons méthodologiques. Pour ce qui est des contraintes applicatives, la continuité de service d'un certain nombre d'applications (téléphonie mobile, services de la toile...) impose des adaptations dynamiques, que ce soit pour assurer des évolutions logicielles ou pour réagir à des variations importantes du contexte d'exécution (modification de bande passante, déconnexion de périphérique...). Du point de vue méthodologique, il est classique de définir un langage de programmation à l'aide d'un interprète (idéalement directement dérivé d'une sémantique formelle), puis de considérer les manières d'effectuer certains des calculs de l'interprète avant l'exécution en utilisant un compilateur. Même si on s'intéresse ici plus à des infrastructures à base de composants qu'à des langages de composants à proprement parler, le pas à franchir entre infrastructure et langage associé n'est pas si grand et ces notions de sémantique, interprétation et compilation restent tout à fait pertinentes.

En pratique, les deux points de vue se mélangent. Pour une part, il est nécessaire de gérer les adaptations de manière dynamique, pour une autre part, il est plus simple d'interpréter plutôt que de compiler les adaptations. Ceci conduit à l'introduction dans les applications à base de

*Le projet ARCAD (Architecture Extensible pour Composants Adaptables) a été labellisé en décembre 2000. Les différents partenaires sont France Télécom R&D (équipe ASR - Thierry.Coupaye@francetelecom.com), INRIA (projet Oasis - Denis.Caromel@inria.fr, projet Sardes - Daniel.Hagimont@inria.fr), École des Mines de Nantes (équipe OBASCO - Thomas.Ledoux@emn.fr), le laboratoire I3S commun à l'Université de Nice - Sophia Antipolis et au CNRS (projet Rainbow - Anne-Marie.Pinna@unice.fr). Le coordonnateur du projet est Michel.Riveill@unice.fr.

composant d'une couche d'interprétation significative. Cette couche d'interprétation concerne à la fois les composants *atomiques* (c'est-à-dire insécables du point de vue architectural) qui constituent la base de l'application et l'architecture de l'application. Elle se traduit par la multiplication d'indirections et de tests qui sont la source d'un surcoût important, en temps et en mémoire, par rapport à une application qui serait conçue de manière monolithique avec des possibilités d'adaptation moindre.

Ce rapport présente un certain nombre de travaux s'attachant à mettre en évidence ces surcoûts et à les réduire de manière significative à l'aide de techniques d'optimisation appropriées.

La section 2 s'intéresse au surcoût dû à l'introduction d'objets d'indirection pour gérer les liaisons entre composants et en enrichir la sémantique par l'ajout de *services techniques* ou *aspects*. Dans le cas où ces aspects ne sont pas modifiés à l'exécution, ces objets d'indirection peuvent être éliminés soit à la compilation, soit au chargement par des techniques spécifiques d'injection de code qui réalisent une fusion des objets d'indirection et des composants. À l'opposé, il est aussi possible, dans le cas ou l'ajout d'un aspect n'a pas été anticipé statiquement, de réaliser de l'injection de code dynamique. Cette possibilité est étudiée dans la section 3. La section 4 examine l'utilisation de la réflexion et de métaobjets plutôt que d'objets d'indirection. La section 5 présente les mécanismes de réduction des coûts d'adaptation offerts par la plateforme Julia. La dernière section conclut ce rapport en présentant quelques perspectives.

2 Injection de code statique

2.1 Motivations

La structuration des applications sous la forme d'architectures à composants apporte une grande modularité qui facilite la construction et la maintenance. Cette modularité prend principalement deux formes :

- Modularité structurelle. Les applications peuvent être construites par assemblage de composants, ce qui encourage la réutilisation de composants. Cet assemblage reste visible à l'exécution, ce qui permet d'observer l'architecture de l'application en termes de composants et de modifier cette architecture (opération généralement désignée sous le terme de reconfiguration).
- Modularité des aspects. Des aspects non fonctionnels comme la sécurité ou la persistance peuvent être associés aux composants et gérés, dans la mesure du possible, indépendamment de leur code métier. Ceci permet de changer la gestion d'un aspect pour un composant donné en fonction du contexte d'utilisation du composant.

La principale technique utilisée pour implanter la modularité (structurelle ou aspectuelle) consiste à placer des objets d'indirection dans le chemin d'accès à un composant. Ces objets gèrent les deux formes de modularité :

- Modularité structurelle. Ils permettent d'identifier clairement les points d'entrée et de sortie des composants, ainsi que les connexions entre les composants qui relient les points de sortie aux points d'entrée. Cette visibilité de l'architecture de l'application permet de facilement introspecter l'application, et également de changer les connexions entre les composants, par exemple lors du remplacement d'un composant par un autre.
- Modularité aspectuelle. Les objets d'indirection permettent d'ajouter des traitements qui sont exécutés lors des interactions entre les composants. Ces traitements peuvent implanter un aspect non fonctionnel sans modification du code métier des composants. Un exemple est l'implantation d'une politique de protection qui contrôle l'accès à un composant, avec une mise en œuvre à la frontière entre les composants grâce aux objets d'indirection.

La multiplication des objets d'indirection dans les structures d'exécution des environnements à composants peut générer un surcoût significatif en termes de temps d'exécution et d'occupation mémoire. Ce surcoût est d'autant plus élevé que l'architecture logicielle est modulaire. L'évolution dans le domaine va même vers une augmentation de ces structures d'exécution, car on s'oriente

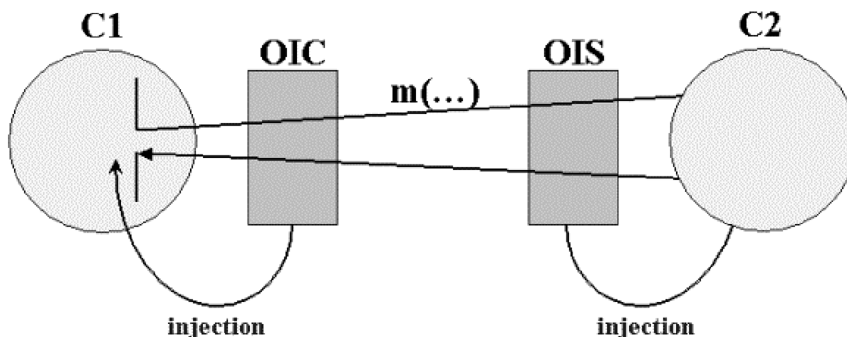

FIG. 1 – *Injection de code non fonctionnel*

progressivement vers des modèles à composants composites (c'est-à-dire imbriqués, comme Fractal [5]). De plus, les intergiciels eux-mêmes (comme l'environnement d'exécution d'un système à composants CCM [12] ou EJB [7]) ont besoin d'être modularisés et structurés en termes de composants, afin de pouvoir être adaptés en fonction de l'environnement d'exécution. Les coûts engendrés par ces structures d'exécution (voir section 2.5) sont donc difficilement acceptables et vont le devenir de moins en moins.

2.2 Approche

Notre objectif est de mettre en œuvre des techniques d'optimisation de la gestion des composants dans les intergiciels. Ces optimisations visent à supprimer dans la mesure du possible les objets d'indirection, sans toutefois perdre les bénéfices de la modularité.

La technique que nous proposons et dont nous rapportons les résultats dans cette section est la transformation du code métier de l'application (automatiquement et par le système), afin d'optimiser la gestion des aspects non fonctionnels.

Dans un intergiciel à composants, une interaction entre deux composants s'effectue généralement au travers de deux objets d'indirection, le premier représentant un port de sortie du composant appelant et le second représentant le port d'entrée du composant appelé. Dans cette section, nous nous référerons à ces objets d'indirection en utilisant les termes OIC pour Objet d'Indirection Client (port de sortie) et OIS pour Objet d'Indirection Serveur (port d'entrée). Notons que dans certains intergiciels à composants, on ne trouve qu'un seul de ces objets d'indirection.

Les objets d'indirection capturent les interactions entre les composants et permettent d'insérer des traitements non fonctionnels entre le composant appelant et le composant appelé. Dans la suite, nous présentons deux expériences : l'une menée avec un aspect gérant des composants partagés et dupliqués, et l'autre prenant en considération la protection à l'aide de capacités logicielles. Dans ces expériences, le code non fonctionnel embarqué dans les objets d'indirection consiste à contrôler la liaison à un composant local (dans le cas de service d'objets partagés dupliqués) ou bien à vérifier des droits d'accès (dans l'aspect lié à la protection).

Le principe général que nous appliquons pour optimiser ces applications consiste à injecter le code non fonctionnel, initialement implanté dans les objets d'indirection, directement dans le code de l'application (figure 1). En général, les traitements additionnels qui étaient implantés dans l'OIC (respectivement l'OIS) sont injectés dans le code métier du composant appelant (respectivement le composant appelé). Cette injection de code peut être effectuée au moment de la compilation ou au moment du chargement.

Nous appliquons ce principe aux deux applications citées ci-dessus, implantées dans l'environnement Java. Nous montrons tout d'abord que cette injection du code non fonctionnel d'un aspect dans le code métier des composants de l'application n'est pas triviale, mais qu'elle est possible.

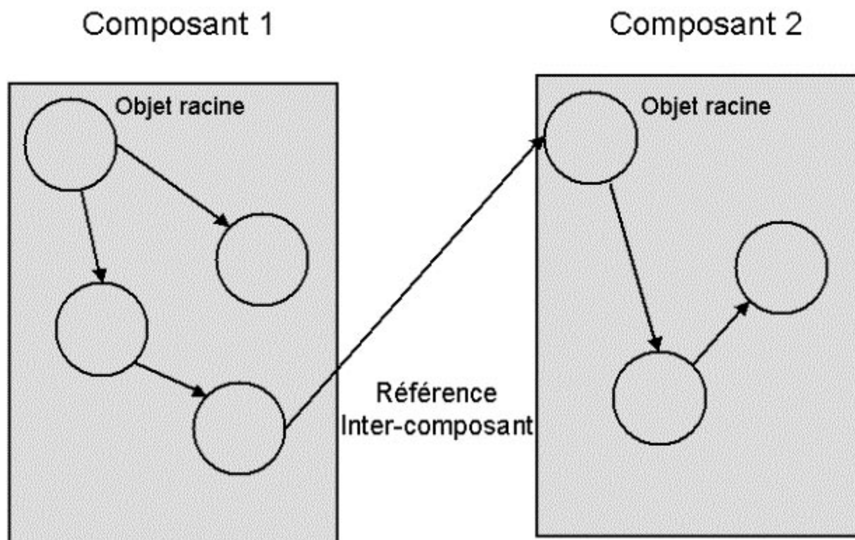

FIG. 2 – *Modèle à composants*

Nous montrons également à travers ces deux exemples qu'il se dégage un schéma (*pattern*) permettant d'envisager un transformateur de code générique, paramétré par une description de l'aspect à prendre en compte.

2.3 Aspects non fonctionnels considérés

Nous prenons en considération deux aspects non fonctionnels dans cette section, liés d'une part à la gestion de composants dupliqués et d'autre part à la prise en compte de règles de protection associées aux composants. Nous donnons ci-après les informations qu'il est nécessaire de connaître sur ces aspects non fonctionnels pour comprendre la suite. Pour chacun de ces aspects, une implémentation utilisant des objets d'indirection a été réalisée et évaluée sur une plateforme à composants. Pour plus d'informations sur ces implémentations, le lecteur pourra se référer à [8, 9].

Les composants que nous considérons sont des groupes d'objets Java, constitués par l'ensemble des objets accessibles à partir d'un objet racine (figure 2). Les objets Java à l'intérieur d'un composant sont appelés des objets locaux. Un composant peut transmettre à un autre composant une réference de composant (référence inter-composant), mais ne peut pas transmettre des références vers des objets locaux. L'accès à un composant se fait au travers d'une référence à son objet racine.

Ce modèle à composants est simple, mais les résultats présentés dans cette section peuvent être appliqués à des modèles à composants plus complexe.

Gestion de composants dupliqués

Nous souhaitons fournir au programmeur d'application la possibilité de dupliquer les composants de son choix. Pour ce faire, le programmeur doit configurer les composants d'une application en leur associant des étiquettes (dupliqué / non dupliqué, type d'accès des méthodes, etc).

La duplication des composants reste toutefois transparente pour le programmeur, dans le sens où l'accès à un composant ne dépend pas du fait que le composant soit dupliqué ou non (la cohérence est assurée par la mise en œuvre de la duplication).

Les interfaces des composants sont annotées avec des mots clés qui sont utilisés par le protocole de cohérence des composants dupliqués. Il est possible d'associer un mode (lecteur ou rédacteur) à chaque méthode. Un exemple de description d'interface de composant est donné ci-dessous :

```
public class MyComponent implements MyComponent_itf {
   public void method1 (Comp_itf obj);  read
   public Comp_itf2 method2();           write
```

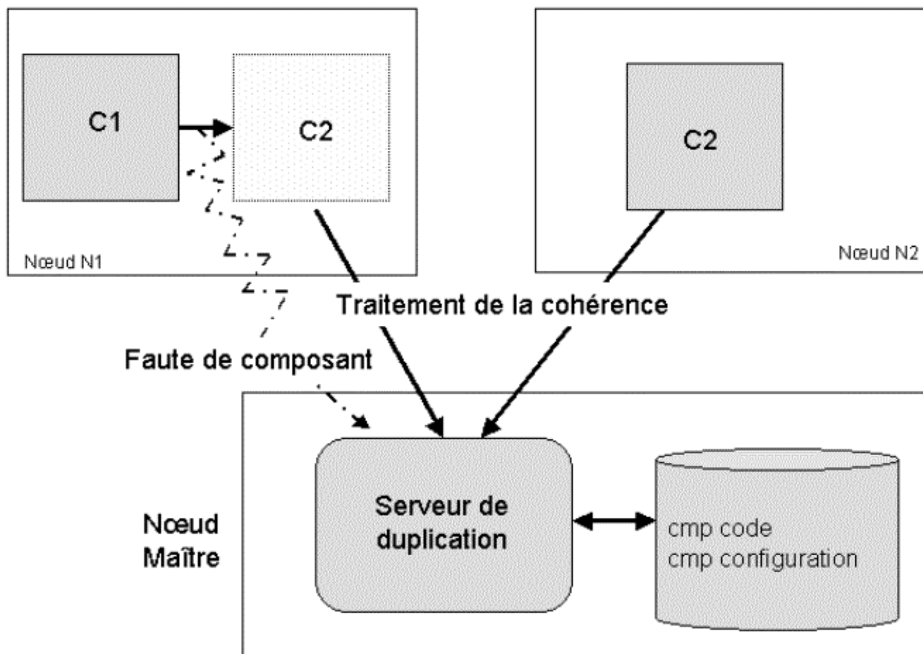

FIG. 3 – *Architecture générale de la gestion de la duplication*

```
}
```

La gestion de composants dupliqués nécessite des mécanismes pour traiter les fautes de composants, l'invalidation et la mise à jour des copies de composants. Lorsqu'une méthode est invoquée sur un composant, une faute de composant se produit si le composant n'est pas présent localement ou s'il est invalide (c'est-à-dire la copie n'est pas cohérente). Dans ce cas, une copie cohérente du composant est amenée sur le nœud requis et cette copie est mise en cache jusqu'à invalidation par le protocole de cohérence. Pour traiter la liaison et la cohérence entre les composants, les applications doivent interagir avec un serveur de duplication. Ce serveur (figure 3) gère la cohérence des copies des composants dupliqués. Un protocole de cohérence à lecteurs multiples et écrivain unique est implanté [1]. Lorsqu'une copie de composant est verrouillée en écriture, les autres copies sont invalidées par le serveur. La figure suivante présente l'architecture générale de la gestion de la duplication.

La mise en œuvre des composants dupliqués basée sur des objets d'indirection place entre chaque paire de composants communicants, un ou plusieurs objets qui interceptent les échanges entre ces composants. Ces objets appliquent le contrôle requis par l'aspect non fonctionnel qu'ils mettent en œuvre.

Dans le cas présent, on retrouve les deux objets d'indirection (l'OIC et l'OIS), qui sont insérés entre un composant appelant et un composant appelé. L'OIC intercepte les appels sortants du composant auquel il est associé, alors que l'OIS intercepte les appels entrants vers un composant.

Plus précisément, le rôle de l'OIC est de gérer le cache local du composant appelé. Au moment d'un appel, si le composant appelé n'est pas présent localement, l'OIC communique avec le serveur de duplication pour récupérer une copie du composant. Un OIC contient l'identification unique du composant appelé, afin de pouvoir identifier le composant à charger localement lors d'une faute de composant. Un OIS a pour fonction de verrouiller le composant auquel il est associé lors d'un appel entrant. Lors d'un appel de méthode sur ce composant, si le verrou requis n'est pas déjà disponible localement, l'OIS communique avec le serveur de duplication pour l'obtenir (et

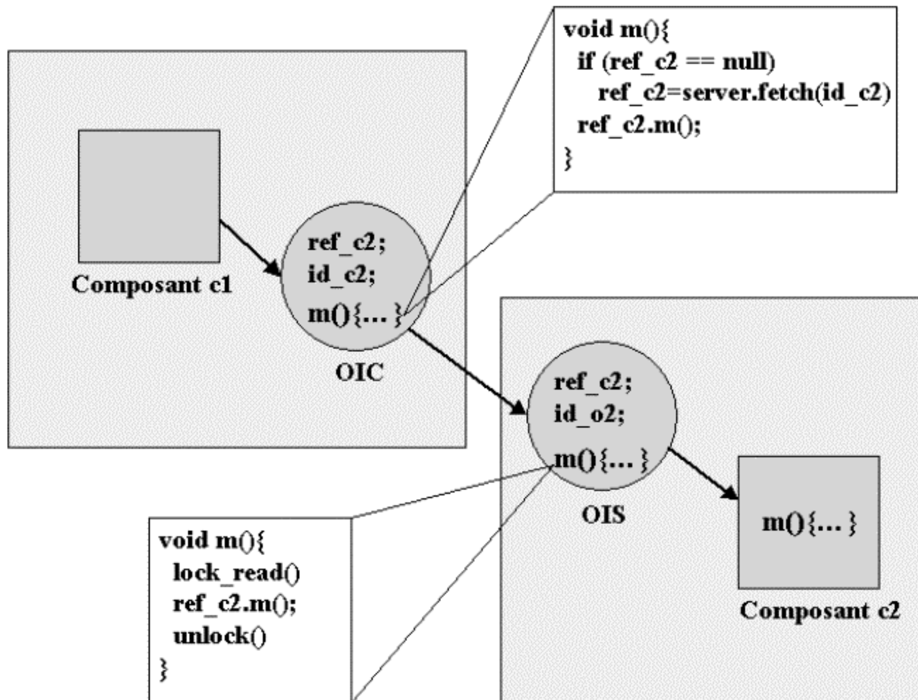

FIG. 4 – *Implémentation des composants dupliqués basée sur des objets d'indirection*

obtenir également une copie cohérente). Les objets d'indirection (OIC et OIS) sont transparents pour le programmeur. Pour ce faire, un OIC implémente l'interface du composant appelé. Pour un composant donné, il y a donc autant d'OIC que de composants différents référencés depuis ce composant. Il n'y a en revanche qu'un OIS par composant.

La figure 4 illustre l'invocation d'une méthode `m()` par le composant appelant `C1` sur le composant appelé `C2`. La référence détenue par `C1` sur `C2` est matérialisée par l'objet OIC. La méthode `m()` du composant `C2` est une méthode de lecture. L'OIS effectue donc un verrouillage de l'objet en lecture avant l'exécution de cette méthode, et libère le verrou après exécution.

Une copie locale d'un composant (soit `C2`) est invalidée en fixant à nulle la référence `ref_C2` contenue dans l'OIS associé au composant. Il existe par conséquent deux types de faute de composants :

- Faute dans un OIC : quand la référence détenue par un OIC est nulle (car aucune copie de `C2` n'a jamais été amenée sur le site), une copie du composant référencé est amenée localement. Cette copie inclut l'OIS associé au composant.
- Faute dans un OIS : quand la référence détenue par un OIS est nulle (car elle a été invalidée), une copie du composant est amenée localement. Cette copie n'inclut pas l'OIS étant donné qu'il est déjà présent localement. Il n'est pas nécessaire de modifier les références présentes dans les OIC lors de l'invalidation d'un composant, puisque ces références pointent sur l'OIS associé au composant et non pas sur le composant directement.

L'architecture qui est mise en œuvre dans le cas de la gestion de composants dupliqués est donc ici une architecture qui introduit deux objets d'indirection sur une chaîne de liaison entre un composant appelant et un composant appelé.

Gestion de composants protégés

La protection est un aspect non fonctionnel qui peut être géré par des capacités logicielles attachées aux composants partagés. Une capacité est une structure de donnée dynamique, qui identifie un composant et contient des droits d'accès sur ce composant (sous-ensemble des méthodes

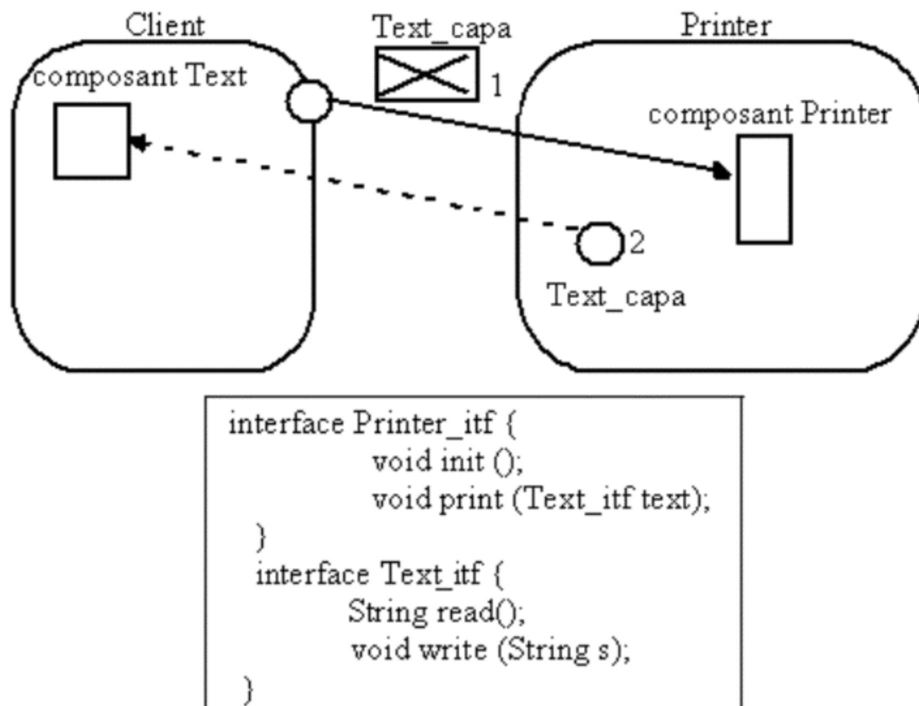

FIG. 5 – *L'exemple du serveur d'impression*

qui peuvent être invoquées), comme une clé ou un jeton. Pour pouvoir accéder à un composant `C`, une application doit détenir une capacité sur `C` avec les droits d'accès correspondants. Quand un composant est créé, la capacité qui est retournée au créateur contient généralement l'ensemble des droits d'accès sur le composant. Elle peut être transmise à d'autres applications avec des droits restreints. Pour illustrer ce modèle de protection, nous considérons l'exemple d'un service d'impression `Printer` qui permet à des clients d'imprimer du texte (figure 5). Nous considérons que le service `Printer` et ses clients sont mutuellement méfiants. Le client souhaite pouvoir imprimer un texte (implanté par un composant) tout en interdisant la modification de ce texte. Le composant `Printer` souhaite pour sa part interdire à un client quelconque de réinitialiser l'imprimante.

Pour pouvoir imprimer un texte, les applications clientes doivent posséder une capacité sur le composant Printer leur permettant d'appeler la méthode `print()`. Quand un client invoque cette méthode, il transmet en paramètre une référence à un composant `Text` contenant les données. Pour imprimer ces données, le composant `Printer` doit avoir les droits d'accès à la méthode `read()` sur le composant `Text`. Lors de l'invocation de la méthode `print()` (1), le client transmet une capacité (`Text_capa`) en lecture seule sur le composant `Text`. Cette capacité permet au composant `Printer` de lire le texte à imprimer (2), et lui en interdit la modification.

Les capacités logicielles qui, durant l'exécution, sont échangées entre les composants, sont décrites par le programmeur d'applications, au travers de règles de protection. Comme pour la gestion de composants dupliqués, un objectif important est de séparer la description de ces règles du code fonctionnel de l'application. Les règles de protection sont décrites au travers de vues.

Une vue s'applique à une interface donnée. Étant donnée une interface `I`, une vue décrit précisément :

- les méthodes de `I` qui sont autorisées,
- pour chaque référence de composant transmise en paramètre d'une méthode autorisée, la vue qui doit être transférée,
- pour chaque référence reçue en résultat d'une méthode autorisée, la vue à associer à cette

référence.

Chaque application (ici `Client` et `Printer`) donne sa propre définition des vues.

Le code ci-dessous présente un exemple de description de vues pour le serveur d'impression décrit précédemment. L'application cliente définit la vue `Text_server_view` qui autorise seulement la lecture sur un composant de type `Text`. Cette application définit également une vue `Printer_client_view` associée au composant `Printer`, qui exprime le fait qu'une référence sur le composant `Text` ayant la vue `Text_server_view` doit être transférée en paramètre lors de l'invocation de la méthode `print()`.

L'application `Printer` définit pour sa part la vue `Printer_server_view` qui permet d'interdire à un client d'invoquer la méthode `init()` pour réinitialiser l'imprimante. Notons que le client n'a aucune raison de s'empêcher lui même d'utiliser la méthode `init()`, c'est une décision prise par le serveur d'impression.

```
// Vues définies par l'application Client

view Text_server_view implements Text_itf {
     String read();
     void not write (String s);
}
view Printer_client_view implements Printer_itf {
     void init ();
     void print (Text_itf text {\bf pass Text_server_view});
}

// Vues définies par l'application Printer

View Printer_server_view  implements Printer_itf {
     void not init ();
     void print (Text_itf text);
}
```

À l'exécution, lorsqu'un client C1 possède une référence sur un composant `C2`, deux vues sont mises en jeu : une vue client et une vue serveur. La vue client assure la protection de `C1` en tant que client de `C2`. Elle spécifie :

- le contrôle à appliquer sur les références qui sortent de `C1` (références sortantes),
- le contrôle à appliquer sur les références qui entrent chez `C1` (références entrantes).

De manière symétrique, la vue serveur assure la protection de `C2` et spécifie :

- les méthodes pouvant être invoquées sur `C2`,
- le contrôle à appliquer sur les références qui sortent de `C2`,
- le contrôle à appliquer sur les références qui entrent dans `C2`.

Le contrôle sur une référence sortante (soit `r`) permet de limiter les méthodes autorisées sur `r` pour le composant qui reçoit `r`. Ceci est fait en associant une vue serveur à `r`. Le contrôle sur une référence entrante (soit `r`) dans un composant protège ce dernier par rapport à ses futures invocations sur `r` (paramètres transmis et résultats reçus). Ceci est fait en associant une vue client à `r`.

Durant l'exécution d'une application, les vues sont mises en œuvre à l'aide de capacités logicielles. Une capacité est une structure de donnée qui permet à un composant appelant d'invoquer une méthode sur un composant appelé, en respectant les règles de protection décrites par le programmeur. Une capacité inclut la référence du composant appelé, ainsi que les identifications de la vue client et de la vue serveur associées à la liaison entre le composant appelant et appelé. Dans le cas du service d'impression, le client doit posséder une capacité sur le composant `Printer` pour pouvoir invoquer la méthode `print`. Cette capacité désigne les vues suivantes : `Printer_client_view` (vue client) et `Printer_server_view` (vue serveur).

Pour mettre en œuvre la gestion de composants protégés, des objets d'indirection sont utilisés. Durant l'exécution, toute liaison entre un composant appelant et un composant appelé fait

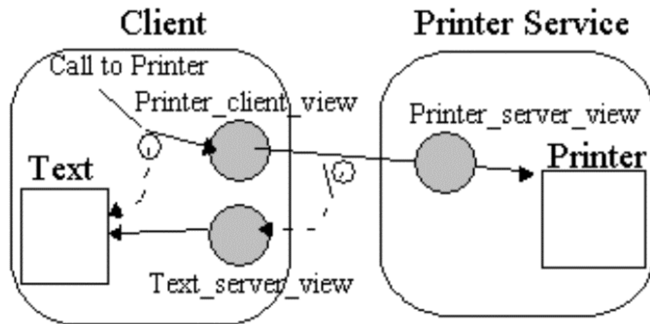

FIG. 6 – *Illustration des objets d'indirection (filtres) créés dynamiquement*

intervenir deux objets d'indirection (OIC et OIS, tels que définis précédemment) correspondant respectivement aux vues client et serveur de la liaison considérée. Tout objet d'indirection (OIC ou OIS) est spécifique à la vue qu'il implémente (les classes des objets d'indirection sont générées par un préprocesseur).

Un objet d'indirection implémente les méthodes déclarées dans l'interface correspondant à la vue et encapsule la référence vers le composant appelé. L'invocation d'une méthode non autorisée provoque une exception.

Un objet d'indirection est également chargé de gérer les références qui entrent et qui sortent depuis un composant, en fonction des directives spécifiées dans la vue qu'il implémente. Tout transfert de paramètre référence est remplacé par le transfert de la référence à l'objet d'indirection qui implémente la vue associée à la référence.

Dans le cas du serveur d'impression (figure 6), l'invocation du composant `Printer`, effectuée par `Client`, transmet une référence vers le composant `Text` en paramètre. L'OIC `Printer_client_view` et l'OIS `Printer_server_view` implémentent respectivement les vues client et serveur liant le composant `Client` au composant `Printer`. L'OIC `Printer_client_view` transmet la référence de l'OIS `Text_server_view` à la place d'une référence directe vers le composant `Text` lors de l'appel de la méthode `print()`.

Le code correspondant aux classes des objets d'indirection pour l'exemple du serveur d'impression est donné ci-après.

```
// Classes des objets d'indirection / application Client

public class Text_server_view implements Text_itf {
     // reference of the component
     Text_itf comp;
     public Text_server_view(Text_itf c) {
          comp = c;
     }
     public String read() {
          return comp.read();
     }
     public void write(String s) {
          // Exception !!!
     }
}

public class Printer_client_view implements Printer_itf {
     // reference of the component
     Printer_itf comp;
     public Printer_client_view(Printer_itf p) {
```

```
            comp = p;
        }
        public void init() {
            comp.init();
        }
        public void print(Text_itf text) {
            Text_server_view ois = new Text_server_view (text);
            comp.print(ois);
        }
}

// Classes des objets d'indirection / application Printer

public class Printer_server_view implements Printer_itf {
        // reference of the component
        Printer_itf comp;
        public Printer_server_view (Printer_itf c) {
            comp = c;
        }
        public void init() {
            // Exception !!!
        }
        public void print(Text_itf text) {
            comp.print(text);
        }
}
```

2.4 Implémentation des aspects non fonctionnels basée sur de l'injection de code

Composants dupliqués

Nous montrons comment utiliser l'injection de code pour implémenter le service de duplication comme une propriété non fonctionnelle sans utiliser d'objets d'indirection. Nous présentons les principes de cette implémentation en nous appuyant sur l'exemple de la figure 4.

Voici les transformations de code à appliquer du côté appelant :

– Toute référence à un objet dupliqué est étendue avec l'identifiant unique du composant dupliqué. Dans le cas de notre exemple, le composant `c1` détient une référence vers un composant `c2`. Nous injectons la déclaration d'une nouvelle variable qui contient l'identifiant du composant référencé, qui était auparavant dans l'OIC (`id_c2`). Cette nouvelle variable doit être affectée à chaque fois que la variable contenant la référence vers le composant est affectée (le code des méthodes est modifié en conséquence). La référence à `c2` dans `c1` est maintenant une référence directe vers l'objet racine du composant `c2` (étant donné que le code de l'OIS est aussi injecté dans le composant `c2`).

– Lors d'un appel depuis un composant appelant vers un composant appelé, il faut injecter dans le code du composant appelant le code qui vérifie la liaison. L'appel d'une méthode est donc précédé d'un test : si la référence vers le composant appelé est nulle, un appel au serveur (comme dans l'OIC) est effectué afin de ramener une copie du composant. Cet appel au serveur prend en paramètre l'identificateur unique du composant appelé (injecté précédemment pour étendre la référence). Dans notre exemple, nous injectons dans `c1` le code qui teste si une copie de `c2` est présente localement. Si ce n'est pas le cas, `c1` va obtenir la dernière version de `c2` auprès du serveur en utilisant l'identificateur `id_c2` (injecté dans `c1`).

– Quand une référence vers un composant dupliqué est passée en paramètre d'une méthode, l'identifiant unique injecté pour étendre cette référence est également passé. Ainsi, le com-

posant recevant la référence en paramètre reçoit aussi l'identifiant unique associé. Toute méthode dont la signature contient une référence de composant dupliqué est donc modifiée. Dans notre exemple, quand une référence vers `c3` est passée en paramètre d'un appel entre `c1` et `c2`, l'identificateur de `c3` est aussi transmis. La signature de la méthode `m()` de `c2` est modifiée pour ajouter un paramètre correspondant à l'identifiant de `c3`.

Nous donnons ci-après le code original de la classe de l'objet racine associé à un composant `c1` qui communique avec des composants dupliqués `c2` et `c3`.

```
public class C1 {
    Itfc2 c2;
    Itfc3 c3;
    public void x() {
        c2.m1();
    }
    public void y(){
        c2.m2(c3) ;
    }
}
```

Le code obtenu après avoir appliqué les transformations de code coté appelant est le suivant :

```
public class C1 {
    // Référence étendue pour c2:
    transient Itfc2 c2;
    Id id_c2; // id de c2

    // Référence étendue pour c3:
    transient Itfc3 c3;
    Id id_c3; // id de c3

    public void x() {
        // Gestion de la liaison dynamique
        if (c2 == null)
            c2 = Server.fetch(id_c2);
        c2.m1();
    }
    public void y() {
        // Gestion de la liaison dynamique
        if (c2 == null)
            c2 = Server.fetch(id_c2);
        c2.m2(id_c3,c3);
    }
}
```

Les injections de code dans un composant appelé permettent d'effectuer le travail auparavant réalisé par un OIS. Nous donnons ci-dessous le code obtenu coté appelé après injection, pour le composant `c2`. Par rapport au composant initial, un nouvel attribut (`id_c2`) qui contient l'identificateur unique du composant a été rajouté. Comme dans l'OIC, il permet d'identifier le composant lors d'une requête au serveur. La gestion du verrouillage du composant a également été ajoutée ; nous injectons le code des méthodes `lock_read` / `lock_write` / `unlock` utilisées pour maintenir la cohérence. Ces méthodes utilisent une variable verrou (*lock*) qui indique si le verrou est caché localement ; cette variable est injectée dans le composant appelé. Si le verrou dans le mode requis n'est pas caché localement, il est récupéré à partir du serveur (le dernier état du composant est aussi récupéré).

```
public class C2 {
     public int id_c2;
     private short lock;
     public void m1() {
          // code de la méthode lock_read
          // code de la méthode m1()
          // code de la méthode unlock
     }
}
```

Composants protégés

Dans cette section, nous montrons comment utiliser l'injection de code pour implémenter le service de contrôle d'accès comme une propriété non fonctionnelle sans utiliser d'objets d'indirection. Nous illustrons notre démarche à l'aide de l'exemple du serveur d'impression présenté auparavant. L'optimisation du code lié à la protection consiste à effectuer les transformations suivantes :

- Toute référence possédée par un composant `C1` vers un composant `C2` est étendue avec les identifiants des vues client et serveur associées à la liaison entre `C1` et `C2`. Toute réaffectation de la référence engendre la réaffectation des identifiants des vues (le code effectuant cette tâche est injecté).
- Afin de vérifier qu'un appel de méthode est autorisé, pour toute invocation effectuée par un composant `C1` sur un composant `C2`, l'identifiant de la vue serveur associée à la liaison entre `C1` et `C2` est transmis en paramètre (additionnel) de l'invocation. Cette vue serveur est utilisée par du code injecté en début de la méthode appelée pour vérifier si la méthode est autorisée (cette vérification, auparavant réalisée dans l'OIS, est injectée dans la méthode appelée).
- Lorsqu'une référence à un composant protégé (soit `R`) est passée en paramètre, nous injectons dans la méthode appelante le code qui ajoute en paramètre additionnel la référence de la vue serveur associée à `R`. Dans la méthode appelée, est injecté le code qui affecte la vue cliente associée à `R`.

Dans l'exemple du serveur d'impression, quand un client reçoit une référence vers un composant `Printer` (à partir d'un serveur de nom), il reçoit aussi la vue serveur relative à la référence. Les invocations de méthodes sur le composant `Printer` embarquent donc deux paramètres supplémentaires (voir le code ci-dessous) :

- La vue serveur associée à la référence à `Printer`. Cette vue (`printer_server_view`) est passée en paramètre pour vérifier les droits d'accès du côté de `Printer`.
- La vue serveur relative au composant `Text` passé en paramètre. Cette vue (`text_server_view`) spécifie les droits d'accès qui sont conférés par `Client` à `Printer` pour utiliser `Text`. La vue serveur à passer avec `Text` est définie en fonction de la vue cliente associée à la référence sur `Printer`, désignée par la variable injectée `printer_client_view` dans le code ci-dessous.

```
public class ClientPrinter {
     public static void main(String args[]) {
          // Référence étendue pour printer
          Printer_itf printer_ref;
          short printer_client_view;
          short printer_server_view;

          // récupérer la référence à printer auprès d'un serveur de nom
          ...
          // Création d'un composant text
          Text text = new Text();

          // Avant l'appel à la méthode print()
```

```
            // passage de la vue serveur passée avec text, en fonction
            // de la définition de la vue printer_client_view
            switch (printer_client_view) {
                ...
                printer_ref.print(printer_server_view, text, text_server_view);
                ...
            }
        }
    }
```

Du côté du serveur (code ci-dessous), la méthode `print()` vérifie les droits d'accès en s'assurant que la vue serveur reçue en paramètre (`printer_server_view`) fait partie des vues qui autorisent la méthode. La vue serveur associée au paramètre `Text` est reçue en paramètre (`text_server_view`). Le code injecté initialise la vue cliente associée au composant `Text` reçu en paramètre. Cette vue cliente (`text_client_view`) est fonction de la vue serveur du composant `Printer` (`printer_server_view`).

```
    class Printer implements Printer_itf {
        public void print (short printer_server_view,
                           Text_itf text,
                           short tex_server_view) {
            // Référence étendue pour text
            short text_client_view;
            short text_server_view;

            // Vérification des droits d'accès à la méthode
            if (printer_server_view != ... ) {
                // Exception !!!
            }
            // Initialisation des vues pour text
            text_server_view = tex_server_view;
            switch (printer_server_view) {
                ...
                text_client_view = ...
            }
            text.read(this.text_server_view);
            return;
        }
    }
```

2.5 Évaluation de performance

Dans cette section, nous donnons les résultats des mesures effectuées sur un cas de base (illustré par la figure 7), afin d'évaluer les gains potentiels des techniques d'optimisation proposées.

Dans ce schéma, le composant `C1` invoque la méthode `m()` sur le composant `C2`. Nous considérons le cas où la méthode `m()` ne prend pas de paramètre et le cas où il prend en paramètre une référence vers un autre composant `C3`. Le code de la méthode `m()` est vide. Ces mesures ont été effectuées dans les trois situations suivantes :

- avec le service de duplication, implémenté avec des objets d'indirection ou avec l'injection de code,
- avec le schéma de protection basé sur les capacités, implémenté avec les objets d'indirection ou l'injection de code,
- avec Java, sans aucune intégration des propriétés non fonctionnelles (pour comparaison).

La table suivante présente les résultats de cette évaluation en utilisant un processeur Pentium à

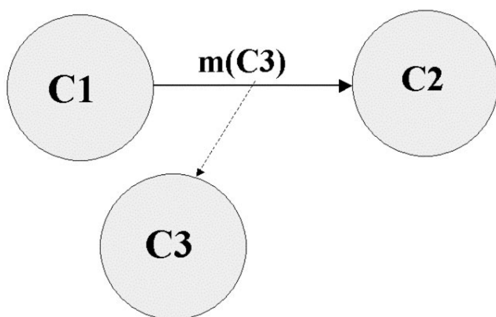

FIG. 7 – *Schéma de base pour l'évaluation de performance*

1Ghz avec 256 Mo de RAM. Ces résultats sont donnés pour 10^8 itérations sur l'appel de méthode.

Invocation	prot-indir	prot-inject	dupl-indir	dupl-inject	Java
Meth `m()`	6458 ms	3354 ms (-48%)	13889 ms	7591 ms (-45%)	1552 ms
Meth `m(o3)`	14400 ms	3565 ms (-75%)	16022 ms	8453 ms (-47%)	1713 ms

Ces mesures montrent un gain de performance en utilisant les techniques d'injection de code. Dans la suite nous détaillons chaque cas :

Protection Dans le cas d'un appel de méthode simple sans paramètre, le gain de performance est de 48%. Ce gain s'explique par le fait que nous évitons deux indirections (deux appels de méthodes). Dans le cas d'un appel de méthode avec une référence passée en paramètre, le gain de performance est de 75%. Dans la version basée sur des objets d'indirection, lorsque l'on transmet une référence vers un composant protégé (`C3`) en paramètre de l'appel entre `C1` et `C2`, l'OIC (de `C1` pour la référence à `C2`) doit instancier un OIS pour protéger `C3`. Dans la version avec l'injection de code, aucun objet d'indirection n'est instancié étant donné que le code de protection est injecté dans les composants appelant et appelé (cependant nous devons passer des paramètres supplémentaires pour implémenter le transfert de capacité). Nous donnons aussi le coup d'un appel de méthode direct pour situer le coût d'une propriété non fonctionnelle implémentée en utilisant l'injection de code par rapport au même code applicatif sans propriété non fonctionnelle.

Duplication Dans le cas d'un simple appel de méthode sans paramètre, le gain de performance est proche de 45%. De manière similaire à l'expérimentation sur les capacités, ce gain s'explique par le fait que l'on court-circuite deux appels de méthode. Dans le cas d'un appel de méthode avec une référence passée en paramètre, le gain de performance est presque identique (47%). Ceci s'explique du fait que dans la version avec les objets d'indirection, lorsque l'on passe une référence en paramètre, on passe une référence vers un OIC qui peut être partagé par plusieurs composants détenant la même référence. Par conséquent, aucune création d'objet d'indirection n'est requise pour passer des références en paramètres. L'implémentation du service de réplication a un coût plus important que l'implémentation des capacités car le service de réplication requiert des opérations de synchronisation coûteuses.

Ces mesures, bien que préliminaires, sont encourageantes. Elles montrent que les gains de performances potentiels peuvent être significatifs. Des évaluations complémentaires doivent néanmoins être réalisées, pour prendre en considération d'autres paramètres, tels que la place mémoire ou la taille du code d'une application.

2.6 Synthèse

Après avoir considéré les aspects de duplication et de protection, nous avons mis en évidence un ensemble de schémas d'extension ou de transformation de code communs aux aspects non

fonctionnels traités. Ces schémas définissent des extensions conceptuelles, qui s'appliquent sur les éléments de base d'un modèle à composants :

Référence étendue Une référence étendue étend une référence de composant avec plusieurs données qui peuvent être de type quelconque. Toute opération d'affectation, de transfert en paramètre ou de comparaison manipulant une référence étendue engendre l'affectation, le transfert ou la comparaison de l'ensemble des éléments de la référence étendue.

Composant étendu Un composant étendu est un composant dans lequel on a ajouté ou bien étendu des données et/ou des méthodes par rapport à sa définition initiale.

Méthode étendue Une méthode étendue est une méthode dont la signature et/ou le code a pu être étendu par rapport à sa définition initiale.

- Signature : cet élément permet d'ajouter un ou plusieurs paramètres.
- Code : cet élément permet d'ajouter du contrôle sur les invocations de la méthode. Ce contrôle est mis en œuvre par plusieurs traitements, qui peuvent être exécutés au niveau du composant appelant la méthode (avant/après l'invocation), ainsi qu'au niveau du composant appelé (à l'entrée / à la sortie de la méthode).

Alors que les possibilités d'extension de méthode et de composant ont déjà été proposées dans le domaine de la programmation par aspects, la notion de référence étendue est novatrice et offre des perspectives intéressantes. Elle permet d'associer à une référence de composant, des traitements (non fonctionnels) qui s'appliquent sur les utilisations de la référence (invocations). Ces traitements peuvent en outre être modifiés par les composants, lorsqu'ils transmettent ou reçoivent une référence étendue. Ceci permet d'adapter le code non fonctionnel en fonction du contexte d'exécution courant. Il faut noter que les optimisations que nous mettons en œuvre ne peuvent pas être réalisés par des outils classiques de compilation à la volée (*Just-In-Time compilers*) qui utilisent des techniques d'expansion du code appelé dans le code de l'appelant (*inlining*). En particulier, les données associées à une référence étendue doivent être affectées à chaque fois qu'une référence est affectée. Cette fonctionnalité n'est pas gérée par les méthodes d'*inlining*. Il en est de même, pour le passage de références en paramètre : les signatures des méthodes doivent être modifiées pour prendre en compte les données associées aux références étendues.

3 Injection de code dynamique

3.1 Motivations

Dans le cadre général de l'adaptation dynamique et non anticipée du logiciel, cette section traite de la mise en place d'une approche efficace pour l'ajout de fonctionnalités dans un logiciel aux performances critiques. Dans ce cadre d'étude, deux problèmes conjoints sont à résoudre : comment intégrer de nouvelles fonctionnalités dans un code non prévu à la base pour les accueillir, et comment rendre cet ajout transparent d'un point de vue des performances. En plus de ces problématiques, les contraintes logicielles issues de notre cadre d'étude imposent que l'ajout soit effectué dynamiquement sans interruption de service et sur un processus natif.

En réponse à ces problèmes, nous proposons une architecture logicielle permettant d'ajouter dynamiquement des aspects à un programme de base. Un aspect est constitué d'un composant et d'un *connecteur* permettant d'interfacer composant et programme de base. Ce connecteur est construit à la volée, lors du chargement du composant, et intégré au programme de base par tissage. Les connecteurs peuvent être rapprochés des objets d'indirection de la section 2. Il faut toutefois noter qu'alors que les objets d'indirection ne concernaient qu'un unique *point de jointure* au sein du programme de base (un appel de méthode), un connecteur est associé à un ensemble de points de jointure.

Notre approche est basée sur une technique efficace de tissage dynamique ne nécessitant pas de préparation du programme de base. Le tissage est effectué par réécriture du code natif. Étant donné la complexité du travail, nous limitons actuellement notre prototype au support des applications fonctionnant sur le système d'exploitation Linux et les architectures x86. Cette approche peut,

conceptuellement, servir de support pour implémenter un système à aspects pour n'importe quel langage se compilant vers du code machine. Dans nos expérimentations, nous avons choisi d'utiliser le langage C.

Au niveau des travaux connexes, des solutions d'ajout dynamique de fonctionnalités ont été proposées mais la plupart d'entre d'elles sont basées sur un langage intermédiaire interprété comme Java. L'ajout des fonctionnalités repose soit, comme dans Prose [15], sur l'utilisation des interfaces de débogage offertes par l'interpréteur soit, comme dans JAC [14], sur l'introduction par réécriture du code au chargement d'une chaîne d'objets d'interposition par composant du programme de base, ou encore, comme dans Reflex (voir section 4), sur l'utilisation de la réflexion. Ces choix ne permettent pas d'utiliser ces outils dans le domaine que nous nous proposons d'étudier. Les performances de ce type de plateformes sont trop faibles pour le domaine d'application visé.

3.2 Contexte de l'étude

L'architecture proposée est issue d'un problème réel d'adaptation non anticipée pour les *caches web*. Les caches web sont une solution éprouvée pour améliorer les performances du réseau Internet. Cependant, la multiplication des services et la génération dynamique des pages conduisent à une diversification des besoins vis-à-vis du cache. Il devient important de pouvoir spécialiser à la volée le cache en fonction du service accédé. Le téléchargement dynamique de composants n'apporte qu'une réponse partielle : dans ce cas, l'interaction entre le cache et les composants est limitée aux interfaces prévues par ce dernier. Une fois le cache développé, ces interfaces sont figées tandis que les services se multiplient et que de nouveaux besoins continuent à se faire jour. Nous proposons d'utiliser la programmation par aspects afin de pouvoir étendre à la volée les fonctionnalités offertes par les caches. Pour appliquer notre approche, nous avons conçu notre propre système à aspects, μDYNER, et dérivé du cache web libre le plus répandu, SQUID, un cache web dynamiquement extensible.

3.3 Implémentation

μDYNER est conçu autour d'un noyau minimal résidant dans le processus du cache qui traite les requêtes de tissage d'aspects. À la réception d'une requête de tissage, ce processus léger commence par charger les aspects à tisser dans l'espace mémoire du cache. Puis, il charge les bibliothèques dites *bibliothèques de crochetage* gérant les opérations de tissage associées aux coupes référencées par les aspects. Finalement, il utilise ces bibliothèques de crochetage pour réécrire à la volée le code du cache. Cette structure, et surtout la liaison dynamique entre les bibliothèques et le noyau, rend ce dernier indépendant du processus hôte, des aspects utilisés, et du langage de coupe. Cependant, les expérimentations effectuées se sont limitées au tissage d'aspect sur processus natif écrit en langage C.

Le prototype est composé essentiellement de deux outils : le compilateur d'aspect et le tisseur. Le compilateur de μDYNER conditionne les aspects compilés sous la forme d'une bibliothèque partagée. Afin de faciliter l'intégration des aspects avec des composants écrits en C, le compilateur d'aspect permet de mélanger dans un même fichier du code C et des aspects. Pour chaque aspect, le compilateur génère deux fonctions : la première sert de point d'entrée dans l'aspect et la seconde contient l'action à effectuer. Le tisseur est une librairie partagée chargée dynamiquement dans le logiciel de base. Sa principale fonction est de réécrire le code binaire du logiciel de base pour le relier au code de l'aspect compilé. Plus précisément, pour chaque point de jonction, le tisseur alloue dynamiquement un petit bloc d'instructions assembleur : le *crochet*. Il réécrit le site du programme de base comme un saut vers ce crochet, qui lui même appelle la fonction point d'entrée de l'aspect. Le crochet effectue également des opérations de services, essentiellement des sauvegardes de registre, maintenant la cohérence de l'exécution.

3.4 Évaluation

Afin d'évaluer notre prototype nous avons réalisé deux types d'expérimentation. La première consiste à effectuer des micro-évaluations pour évaluer l'efficacité du lien (le connecteur entre le programme de base et le composant) et l'efficacité de la technique d'ajout de fonctionnalité. La seconde consiste en une macro-évaluation chargée d'évaluer le surcoût global pour l'ajout d'une fonctionnalité. Dans cette dernière expérimentation, nous cherchons à mesurer en terme de temps d'exécution le coût d'une insertion dynamique sur un cas concret et dans un logiciel d'envergure et à le comparer au surcoût lié à une intégration manuelle au niveau du source de la même fonctionnalité.

3.4.1 Micro-évaluation

Sachant que nous lions aspect et programme de base par transformation dynamique du code, il devient nécessaire d'évaluer le coût de la transformation et le surcoût qu'elle introduit à l'exécution. Pour réaliser ces évaluations, nous avons implémenté l'algorithme *quicksort*. Nous avons activé un aspect dans sa fonction de partition. Nous avons conduit notre expérimentation sur un Intel Pentium 4 à 1600 MHz disposant de 256 Mo de RAM fonctionnant sous un noyau Linux 2.4.17 avec gcc 2.95.4.

Le coût de la transformation est à mettre en regard des autres outils disponibles permettant de transformer le code à l'exécution. En terme de technologie de réécriture à l'exécution de code natif dans l'espace utilisateur, nous nous référons à Dyninst [6] afin d'évaluer la qualité de notre implémentation. En moyenne, notre prototype réalise l'instrumentation en 0.04 ms quand Dyninst 3.0 demande 1269 ms. La différence réside dans le fait que Dyninst utilise l'API de débogage de Linux (*ptrace*) nécessitant, contrairement à notre approche, l'arrêt du processus, et la synchronisation de deux processus. Pour évaluer le surcoût à l'exécution, nous avons comparé le temps d'exécution de notre quicksort compilé avec les optimisations du compilateur activées sur un tableau de dix millions d'éléments. Pour ce cas, et en moyenne, le surcoût introduit est inférieur à 10%. De manière plus générale, en terme de micro-évaluation, le surcoût engendré pour rediriger un appel de fonction vers un aspect n'excède pas quelques instructions : deux sauts et un test sur une variable globale. Nous avons établi que déclencher un aspect sur un appel d'une fonction donnée, retournant immédiatement, afin d'exécuter une fonction tierce, retournant également immédiatement, est 2.24 fois plus lent qu'un appel direct à la fonction tierce.

3.4.2 Macro-évaluation

Dans ce cadre de macro-évaluation, nous comparons l'intégration d'un composant par modification manuelle des sources de Squid à l'intégration du même composant de manière outillée en utilisant notre prototype. Comme cible de déploiement, nous avons utilisé une politique de préchargement des accès dans les documents HTML. L'exemple montre que le déploiement à la demande n'induit pas de dégradation sensible des performances du cache.

Nous avons fait nos expériences sur un Celeron 650 Mhz sous un noyau LINUX 2.4.19. Nous avons utilisés GCC 3.2 comme compilateur. Pour limiter les variations introduites par le réseau, nous avons utilisé WGET 1.8.2 pour interroger localement un serveur APACHE 1.3.26 à travers SQUID 2.5.3 également installé localement. En terme de temps de téléchargement observé par le client comme en terme de temps de traitement passé dans le cache, le surcoût introduit par l'outillage est de l'ordre de 1%. Les écarts de mesures observés suggèrent que l'outillage utilisé n'introduit pas un surcoût appréciable. On constate toutefois des écarts de variations plus grands dans les performances de l'intégration outillée que dans celle effectuée à la main. Cet accroissement pourrait s'expliquer par l'existence du processus léger associé au noyau, qui augmente le nombre de changements de contextes.

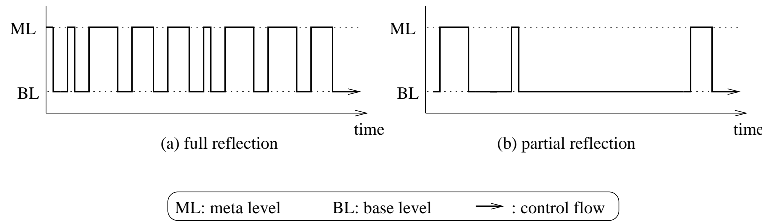

Fig. 8 – « *Réflectogramme* » *d'une application réflexive : illustration de l'évolution du flot de contrôle dans une application réflexive.*

3.5 Synthèse

Nous avons présenté dans cette section une architecture logicielle permettant d'étendre dynamiquement les fonctionnalités d'un cache Web. Le gain en extensibilité est apporté sans préparation du logiciel, sans perte d'efficacité, et sans interruption de service. Concrètement, nous utilisons la programmation par aspects pour créer à la volée les interfaces permettant de connecter des composants à SQUID. Notre approche repose sur un système de tissage dynamique et sur un langage d'aspects permettant la conception et le déploiement du connecteur. Techniquement, notre prototype réalise le tissage du connecteur par réécriture du code binaire exécuté. Cette stratégie de tissage se montre rapide et efficace. L'intégration de ces composants au cache est initiée à distance par le service accédé ou localement par l'administrateur du cache.

4 Sélection spatiale et temporelle de la réification dans Reflex

La *réflexion comportementale partielle* a été introduite comme une approche plus efficace et flexible pour les protocoles de métaobjets [16]. Cette approche propose de résoudre le traditionnel compromis entre flexibilité et efficacité en limitant le plus possible le nombre de sauts du flot de contrôle ayant lieu lors de l'exécution.

En effet, les sauts au métaniveau offrent beaucoup de puissance pour contrôler une application, mais ils ont un coût important. Un tel saut consiste premièrement à réifier une occurrence d'opération, puis à déléguer au moins une partie de son interprétation à un métaobjet. Le bout de code permettant ce saut est appelé un *crochet*. La réification d'une occurrence d'opération est une source importante de la dégradation des performances dans une application réflexive. Par exemple, dans le cas d'une invocation de méthode, la réification implique généralement l'emballage au sein d'un tableau d'objets des différents arguments de l'invocation, et éventuellement l'obtention d'un objet représentant la méthode invoquée. Ces données sont ensuite éventuellement encapsulées au sein d'un objet représentant l'invocation. Du point de vue de l'efficacité, il est donc crucial de limiter le nombre de sauts au métaniveau, de manière à ne payer le prix de la réification que lorsque réellement nécessaire.

La figure 8 représente le « réflectogramme » d'une application réflexive. Un réflectogramme illustre l'évolution du flot de contrôle entre le niveau de base et le niveau méta durant l'exécution. Avec la réflexion *complète* (figure 8(a)), toute opération du niveau de base est réifiée, et donc beaucoup de sauts au métaniveau ont lieu, parmi lesquels il peut y avoir une grande quantité de sauts inutiles, pour lesquels le métaobjet ne fait que reproduire la sémantique d'origine. Ce phénomène ne se produit pas avec la réflexion partielle (figure 8(b)). L'exécution de l'application n'est pas changée là où la réflexion n'est pas réellement nécessaire, et donc ne souffre pas de dégradation en termes de performances.

Nous faisons la distinction entre deux dimensions de la sélection minutieuse de la réification :

la dimension *spatiale* et la dimension *temporelle*.

Sélection spatiale. La sélection spatiale consiste à sélectionner ce qui doit être réifié dans une application. Une telle sélection doit pouvoir être faite aussi bien statiquement que dynamiquement. Nous distinguons trois niveaux de sélection :

- **La sélection d'entités** consiste à sélectionner les classes et objets réflexifs. Par exemple, il est possible de spécifier que les classes `A` et `B` sont *complètement* réflexives (c'est-à-dire que toutes leurs instances sont réflexives), alors que seule l'instance `c` de la classe `C` est réflexive (`C` est *partiellement* réflexive). Les autres classes et objets sont inchangés.
- **La sélection d'opération** consiste à sélectionner les opérations (envoi/réception de message, création d'objets, ...) qui sont réifiées pour une entité donnée (classe ou objet). Par exemple, il est possible de réifier l'envoi de message et l'accès aux champs pour la classe `A`, et uniquement l'envoi de message pour la classe `B`.
- **La sélection intra-opération** consiste à sélectionner finement certaines occurrences d'une opération à réifier. Par exemple, il est possible de limiter la réification de l'envoi de message au seul message `foo` envoyé depuis une instance de la classe `A`.

La possibilité d'effectuer une sélection spatiale poussée est une propriété cruciale de notre approche pour limiter le surcoût de la réflexion comportementale. Il s'agit en effet d'éviter le plus possible de réifier inutilement des occurrences d'opération lors de la transformation du code, afin de ne pas laisser cette tâche aux métaobjets lors de l'exécution.

Sélection temporelle. La sélection temporelle consiste à sélectionner *quand* une réification est effectivement active. Elle permet d'optimiser encore plus l'efficacité globale d'un système utilisant la réflexion. Au cours du temps de vie d'un objet, les besoins en matière de réflexion peuvent évoluer : certaines réifications peuvent être désactivées de telle sorte que le comportement méta ne s'applique plus, ou réciproquement, des conditions dynamiques peuvent impliquer la nécessité d'appliquer un comportement méta et donc, d'activer des réifications. Évidemment, pour que la sélection temporelle soit rentable, il faut que le coût d'une réification désactivée soit inférieur au coût d'une réification activée avec un métaobjet vide (c'est-à-dire simulant le comportement par défaut). Nos micro-évaluations de performance, présentés ci-après, démontrent ce fait dans le contexte de Java.

Dans [16], nous exposons un modèle de réflexion partielle basé sur les *ensembles de crochets*, ainsi qu'un canevas Java implémentant ces idées, Reflex.

Par ailleurs, nous avons réalisés une série de micro-évaluations préliminaires de performance afin de valider notre approche de réflexion partielle. La machine utilisée est un PC sous Linux, avec un processeur Intel Celeron 1.2GHz et 760MB de mémoire vive. La machine virtuelle Java utilisée est la machine HotSpot client de Sun Microsystems, version 1.4.1.

Ces tests, basés sur l'opération d'envoi de message, sont présentés en détails dans [16]. Les principaux résultats sont les suivants :

- le surcoût d'une invocation réflexive est entre 3 et 4 ordres de grandeur supérieur à une invocation normale, ce qui tend à souligner la nécessité de sélectionner précisément où et quand la réification a lieu.
- une réification désactivée réduit sensiblement le surcoût et, également, le coût de l'activation est négligeable lorsqu'elle est utilisée.

Toutefois, ces tests sont des tests préliminaires, et appellent d'autres mesures ; en particulier sur des applications de taille conséquente, et en considérant différentes opérations.

5 Optimisations de la plateforme Arcad Fractal/Julia

5.1 Mécanismes de bases

La plateforme Julia [4] offre deux mécanismes d'optimisation intracomposant et intercomposants. Elle offre également plusieurs implantations des interfaces de contrôle de l'API Fractal qui

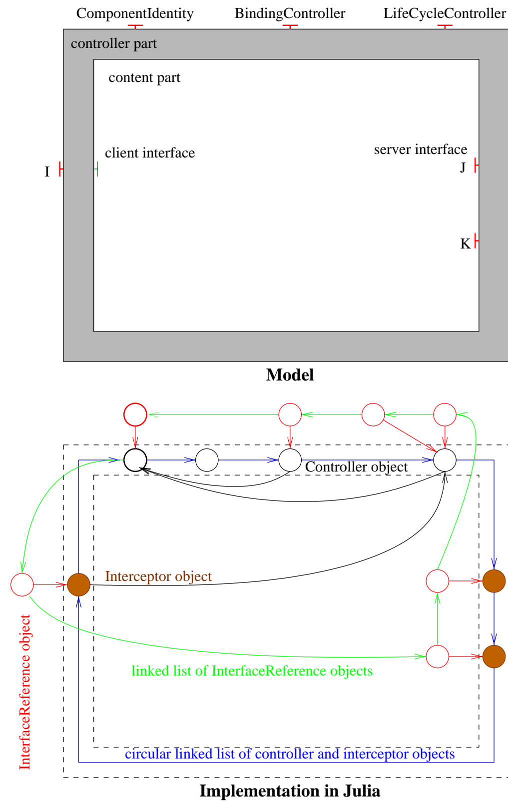

FIG. 9 – *Représentation des composants dans Julia*

permettent d'instancier une configuration donnée en faisant différents compromis entre les performances et l'adaptabilité. Cette section présente ces mécanismes, après un bref rappel des structures de données utilisées dans Julia.

5.1.1 Structures de données utilisées dans Julia

Dans le cas général un composant Fractal est représenté dans Julia par plusieurs objets Java, qui peuvent être séparés en trois groupes (cf. figure 9) :

- les objets qui représentent les interfaces serveur du composant (en rouge). Chaque objet implante `InterfaceReference`, ainsi que l'interface Java correspondant à l'interface métier référencée (comme `Account` pour un compte bancaire).
- les objets qui implantent la partie *contrôleur* du composant (en noir et marron). Parmi ces objets on distingue les intercepteurs qui, comme leur nom l'indique, interceptent les appels de méthodes entrants et/ou sortants du composant. Si le contrôleur d'un composant ne requiert pas de fonction d'interception, ces objets sont absents. Les autres objets de la partie contrôleur implantent les interfaces de contrôle du composant.
- les objets qui implantent la partie *contenu* du composant. Pour les composants composites il s'agit des objets représentant les sous-composants. Pour les composants primitifs (de type *conteneur*) il s'agit d'un objet Java instancié à partir d'une classe *utilisateur*. Plus précisément, il s'agit généralement d'un *graphe* d'objets, mais où un seul objet *racine* sert d'interface entre les parties contrôleur et contenu. Pour le contrôleur, tout se passe donc comme si la partie contenu était constituée d'un seul objet.

5.1.2 Optimisations intracomposant

Pour (re)configurer les objets de contrôle d'un composant Fractal, une solution naturelle consisterait à utiliser Fractal lui-même : chaque objet de contrôle serait alors un composant Fractal avec, par exemple, une interface `BindingController` pour le lier à d'autres objets de contrôle. Il serait également possible, avec cette approche, d'utiliser des composants composites à l'intérieur de la membrane d'un composant !

Cette solution sera peut être implantée dans une version future de Julia. La version actuelle utilise une solution plus efficace, avec laquelle il est toujours possible de fusionner les objets de contrôle en un seul objet (ce qui n'est évidemment pas possible avec la solution précédente). Cette solution est la suivante :

- chaque objet de contrôle peut fournir et requérir zéro ou plus interfaces Java. Les interfaces fournies doivent être implantées par l'objet, qui doit avoir un champ par interface requise, dont le nom doit commencer par `weaveable` pour une interface obligatoire, ou par `weaveableOpt` pour une interface optionnelle (cf. ci-dessous). Chaque objet de contrôle doit également implanter l'interface `Controller`.
- dans une configuration donnée, une interface donnée ne doit pas être fournie par plus d'un objet (sauf pour l'interface `Controller`). Autrement il serait impossible de fusionner ces objets (un objet ne peut implanter une interface donnée de plusieurs façons à la fois).
- les liaisons entre les objets de contrôle d'une configuration donnée sont établis automatiquement en deux étapes :
  - chaque objet de contrôle de la configuration est enregistré auprès d'un *service de nommage*. En pratique, ce service est l'interface `ComponentIdentity` : les objets de contrôle sont mis dans une liste chaînée, qui est utilisée par l'objet implantant `ComponentIdentity` pour retourner les références d'interfaces, d'après leur nom,
  - puis, chaque objet de contrôle s'initialise, en utilisant le service de nommage précédent pour obtenir les références des interfaces qu'il requiert.

Pour être plus précis, supposons que nous ayons quatre interfaces de contrôle `I`,`J`, `K` et `L`, et trois classes d'objet de contrôle `IImpl`, `JImpl` et `KImpl`. Ces classes doivent ressembler à ceci :

```
public class IImpl implements Controller, I {

  // link to the "controller" object that implements ComponentIdentity
  public ComponentIdentity weaveableCI;

  // indicates a required interface of type J
  public J weaveableJ;

  // indicates an optional required interface of type L
  public L weaveableOptL;

  // link to next controller object
  private Controller removeableNext;

  // other fields:
  public int foo;

  // implementation of the Controller interface

  public void initFcController (ComponentIdentity id) {
    weaveableCI = id;
    weaveableJ = (J)id.getFcInterface("_J");
    weaveableOptL = (L)id.getFcInterface("_L");
```

```
  }

  public Controller getNextFcController () {
    return removeableNext;
  }

  public void setNextFcController (Controller next) {
    this.removeableNext = next;
  }

  // other methods:

  public void foo (String name) {
    weaveableJ.bar(weaveableOptL, foo, weaveableCI.getFcInterface(name));
  }
}

public class JImpl implements Controller, J {

  public ComponentIdentity weaveableCI;
  public K weaveableK;
  private Controller removeableNext;

  // implementation of the Controller interface ...

  // other methods (not shown) ...
}

public class KImpl implements Controller, K {

  public ComponentIdentity weaveableCI;
  private Controller removeableNext;

  // implementation of the Controller interface ...

  // other methods ...
}
```

Dans le cas non optimisé, un composant avec ces trois objets de contrôle est instancié de la façon suivante :

- une instance de `BasicComponentIdentity` est créée (cette classe implante `Controller`),
- une instance de `IImpl`, `JImpl` et `KImpl` est créée,
- les objets obtenus sont liés les uns aux autres (dans n'importe quel ordre : la sémantique du composant ne dépend pas de cet ordre) avec la méthode `setNextFcController`,
- la méthode `initFcController` est appelée sur chaque objet de contrôle. La méthode `getFcInterface` utilise la liste précédemment construite pour trouver les interfaces requises par les objets de contrôle.

Dans le cas optimisé, le processus d'intanciation est le suivant :

- la classe obtenue en « fusionnant » les classes `BasicComponentIdentity`, `IImpl`, `JImpl` et `KImpl` est générée dynamiquement, ou lue depuis le disque si elle a été générée statiquement, ou simplement retourné s'il elle a déjà été chargée,
- cette classe est instanciée.

L'algorithme de « fusion » des classes est le suivant. Grossièrement, toutes les méthodes et tous les champs de chaque classe à fusionner sont copiés dans une nouvelle classe (la classe résultante ne dépend pas de l'ordre dans lequel on fait cette copie). Cependant, les champs dont le nom commence par `weaveable` sont remplacés par `this`, et ceux dont le nom commence par `weaveableOpt` sont remplacés par `this` ou par `null`, selon que l'interface requise correspondante est fournie par une des classes fusionnées ou non. Les méthodes de l'interface `Controller` ne sont pas non plus copiées, sauf celles de la classe `BasicComponentIdentity`. Le résultat est la classe suivante :

```
public class Cb234f2 implements Controller, ComponentIdentity, I, J, K, ... {

  // fields copied from the BasicComponentIdentity class (not shown) ...
  // fields copied from the IImpl class:
  public int foo;
  // fields copied from the JImpl class: none
  // fields copied from the KImpl class: none

  // methods copied from BasicComponentIdentity (not shown) ...
  // methods copied from IImpl:
  public void foo (String name) {
    bar(null, foo, getFcInterface(name);
  }
  // methods copied from JImpl (not shown) ...
  // methods copied from KImpl (not shown) ...
}
```

Notes :

- Comme l'explique la section 5.1.1, la partie contrôleur d'un composant est constituée d'objets de contrôle et d'intercepteurs. L'optimisation ci-dessus ne s'applique qu'aux objets de contrôle. Par conséquent, même avec cette optimisation, la partie contrôleur du composant reste en général constituée de plusieurs objets Java. Si tous les intercepteurs délèguent au même objet utilisateur, et s'ils n'ont pas d'interfaces conflictuelles, il est possible de n'avoir plus qu'un objet Java pour toute la membrane d'un composant. Dans ce cas, qui est valable pour la plupart des composant primitifs, le processus d'instanciation est le suivant :
  - une classe fusionnant les classes de contrôle est générée comme précédemment,
  - une sous-classe de cette classe qui implante le code d'interception pour chaque méthode de chaque interface fonctionnelle est générée ou chargée,
  - cette sous-classe est instanciée.
- Même si les objets de contrôle et les intercepteurs sont fusionnés en un seul objet, le contenu du composant est encore constitué d'un objet séparé. On peut cependant instancier le composant entier (c'est-à-dire ses objets de contrôle, ses intercepteurs et son contenu, mais pas les références à ses interfaces) en n'utilisant qu'un seul objet Java. Pour cela, la classe utilisateur est utilisée comme superclasse pour générer la classe de contrôle fusionnée, qui est elle même utilisée comme superclasse pour générer le code d'interception (comme décrit ci-dessus).
- Pour pouvoir désoptimiser dynamiquement un composant optimisé, le générateur de code qui génère des classes fusionnées devrait également générer une méthode de désoptimisation, qui créerait des objets séparés correspondant à un objet fusionné (en préservant son état). Ceci n'est pas encore implanté.

5.1.3 Optimisations intercomposants

En plus des optimisations intracomposant précédentes, qui sont principalement utilisées pour économiser de la mémoire, Julia offre également une optimisation intercomposants, à savoir un

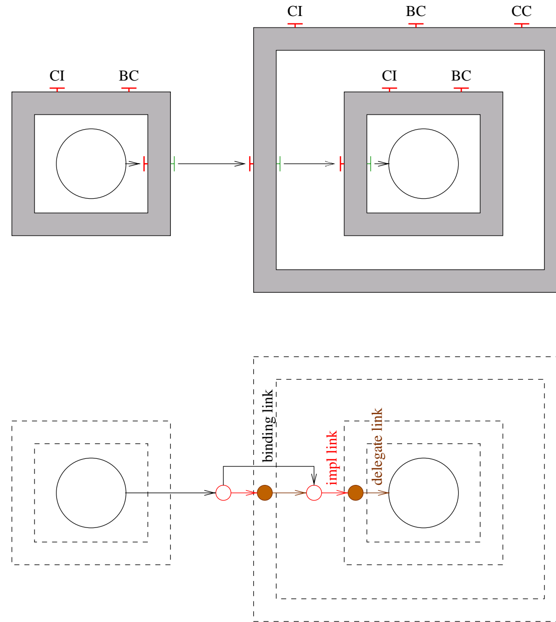

FIG. 10 – *Implémentation par défaut des liaisons*

algorithme pour créer et mettre à jour des liens raccourcis entre les composants, et dont le but est d'améliorer les performances en temps.

Comme l'explique la section 5.1.1 les interfaces serveur (externes ou internes) d'un composant sont représentées par des objets Java (non remplaçables) qui implantent l'interface `InterfaceReference`, ainsi que l'interface Java de l'interface Fractal référencée. Dans le cas d'un composant composite, ces objets Java contiennent, entre autres, les deux liens suivants :

- un lien *binding* vers l'objet Java qui représente l'interface Fractal à laquelle l'interface cliente complémentaire de cette interface serveur est liée,
- un lien *impl* vers un objet Java qui implante l'interface référencée. En général, ce lien pointe vers un intercepteur (qui implante l'interface `Interceptor`) qui, à son tour, pointe vers un objet *delegate* d'un autre composant (qui, en général, est l'objet vers lequel le lien *binding* pointe).

La différence entre ces deux liens est illustrée sur la figure 10. Les liens *binding* sont utilisés pour stocker la méta-information sur les liaisons entre composants, alors que les liens *impl* et *delegate* représentent les liaisons (ou chaînes de liaisons) elles-mêmes.

Quand un composant n'a pas besoin d'intercepter les appels entrants ou sortant il n'a pas besoin, et n'a pas, d'objet d'interception. Dans ce cas il est possible d'utiliser des liens *impl* raccourcis entre les composants, comme le montre la figure 11.

Comme `IR1` n'a pas d'intercepteur associé, et puisque les références d'interface comme `IR2` se contentent de transmettre les appels aux objets référencés par le champ *impl*, le lien *impl* de `IR1` peut pointer directement sur l'intercepteur associé à `IR2`. Cette remarque peut se généraliser en l'algorithme suivant, pour créer, modifier ou supprimer un lien *binding* :

```
void bind (InterfaceReference newBinding) {
  this.binding = newBinding;
  itfref = this.binding;
  while (itfref != null and
```

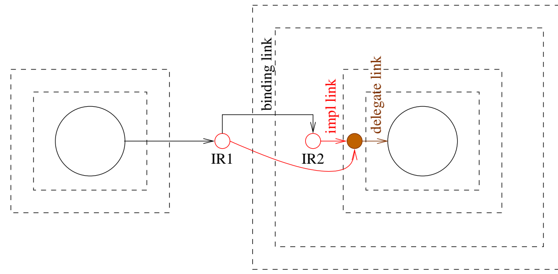

FIG. 11 – *Liaisons raccourcies*

```
          (itfref.impl is not an interceptor) and
          (itfref has a binding link))
  {
    itfref = itfref.binding;
  }
  shortcut = (itfref == null ? null : itfref.impl);
  visit(this, shortcut);
}

// follows all the bindings BACKWARDS
void visit (InterfaceReference current, Object shortcut) {
  if (current.impl is an interceptor) {
    current.impl.delegate = shortcut;
    shortcut = current.impl;
  } else {
    current.impl = shortcut;
  }
  // recursively calls visit(x,shortcut) for each interface x bound to current
}
```

Cet algorithme permet de « contourner » la membrane des composants qui n'interceptent pas les appels de méthode, tout en assurant que celles qui le font ne sont pas contournées.

5.2 Continuum statique - dynamique

Lors de l'instanciation d'un composant, l'utilisateur peut spécifier les interfaces de contrôle, les classes des objets contrôleur implantant ces interfaces, ainsi que les fonctions d'interception qu'il souhaite associer au composant. C'est grâce à cette flexibilité qu'il est possible d'associer plus ou moins de méta-informations et de fonctions d'interception à un composant, et ainsi d'obtenir des configurations dynamiques, statiques ou mixtes.

5.2.1 Configurations dynamiques

Pour instancier une application dans une configuration complètement dynamique, il suffit d'associer à chaque composant les interfaces de contrôle `BindingController` et `LifeCycleController`, ainsi que, pour les composants composites, `ContentController`. Les interfaces `BindingController` et `ContentController` permettent en effet de conserver et de modifier dynamiquement les méta-informations concernant les liaisons et le contenu d'un composant (et, indirectement, les liaisons et le contenu eux-mêmes). Quant à l'interface `LifeCycleController`, elle permet d'arrêter le composant dans un état stable pour le reconfigurer « proprement ».

Julia offre deux classes implantant l'interface `LifeCycleController`, appelées `GenericLifeCycleController` et `OptimizedLifeCycleController`. La classe générique peut

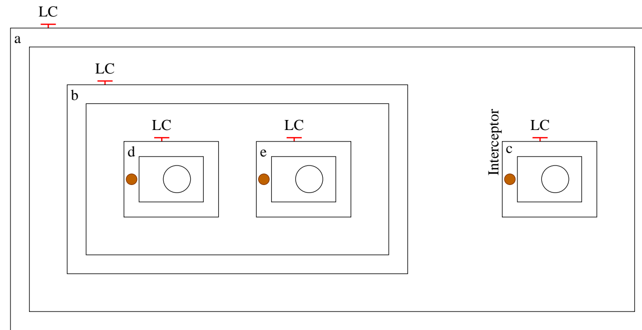

Fig. 12 – *Une configuration complètement dynamique*

être utilisée dans les composants primitifs ou composites, mais requiert la présence d'intercepteurs dans le composant. Le rôle de ces intercepteurs est de compter le nombre d'appels de méthode en cours dans le composant. L'arrêt d'un composant consiste à attendre que ce compteur devienne nul et, atomiquement, à suspendre temporairement tout nouvel appel. La classe optimisée ne requiert pas d'intercepteurs dans le composant, mais ne peut être utilisée que pour les composants composites (dans ce cas l'arrêt d'un composite est réalisé en arrêtant, de façon atomique, les premiers sous-composants directs ou indirects - en « profondeur d'abord » - du composant qui possèdent un contrôleur de cycle de vie basé sur un compteur : dans l'exemple de la figure 12, pour arrêter `a`, on commence par examiner ses sous-composants directs `b` et `c`. `c` possède un intercepteur, on arrête donc là le parcours en profondeur d'abord pour ce sous-composant. Par contre `b` ne possède pas d'intercepteur : on continue donc la visite en profondeur d'abord, qui s'arrête ensuite au niveau de `d` et `e`. Il ne reste plus alors qu'à arrêter de façon atomique les composants trouvés lors du parcours précédent, à savoir `c`, `d` et `e`. Dans l'exemple de la figure 13, on trouve `b` et `c`).

Pour instancier une application dans une configuration complètement dynamique, le mieux est d'utiliser la classe générique pour les composants primitifs, et la classe optimisée pour les composites. Cette méthode permet en effet de minimiser le nombre d'intercepteurs utilisés, et donc le nombre d'indirections entre les composants utilisateur.

5.2.2 Configurations statiques

Julia permet d'instancier des applications dans une configuration complètement statique, mais seulement d'une façon un peu indirecte [1]. En effet, si on instancie directement les composants sans aucun intercepteur ni aucune interface de contrôle (autre que l'interface `ComponentIdentity`), ce qui est parfaitement faisable, alors on ne dispose plus d'aucun moyen pour relier ces composants entre eux, puisque la seule manière standard de le faire est d'utiliser l'interface `BindingController` ! Il faut donc instancier les composants en leur associant un minimum d'interfaces de contrôle, puis supprimer les contrôleurs des composants (et donc les métadonnées) une fois que l'application est instanciée.

Plus précisément, il faut instancier les composants en leur associant les interfaces `BindingController` et `ContentController` (pour les composites), mais *pas* l'interface `LifeCycleController`. Il faut également utiliser une implantation spéciale de `BindingController` pour les composants primitifs, fournie par la classe `OptimizedBindingController`, de façon à éviter toute indirection entre les composants (avec les autres implantations de `BindingController`

1. Il n'y a aucun obstacle théorique ou pratique pour instancier directement une configuration Fractal complètement statique. Au contraire, il serait même très facile de fournir un outil très simple et beaucoup plus léger que Julia pour faire cela - en instanciant directement les classes utilisateurs, et en utilisant leur interface `UserBindingController` pour relier les composants directement. Il se trouve simplement que, pour le moment, cet outil n'est pas implanté.

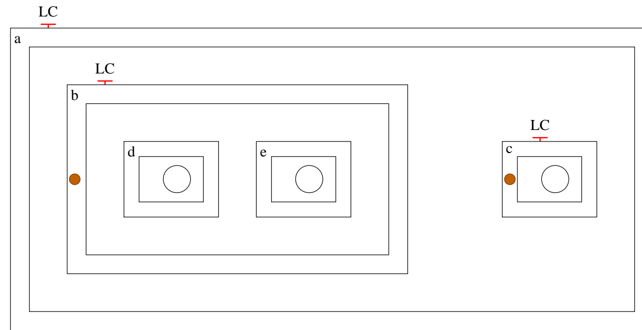

Fig. 13 – *Une configuration mixte*

il y a toujours au moins une indirection entre les composants, même en l'absence de tout intercepteur, qui est due au typage des références d'interfaces). Les composants applicatifs sont ainsi instanciés sans aucune indirection entre eux, et sans aucune référence vers les contrôleurs des composants de l'application. Ces contrôleurs, contenant la méta-information sur l'application, peuvent donc être collectés par le ramasse-miettes (si on ne conserve pas, par ailleurs, de référence vers eux) et, une fois qu'ils le sont, on obtient une configuration réellement complètement statique.

5.2.3 Configurations mixtes

Julia permet de choisir les interfaces de contrôle et leur implantation composant par composant, ce qui permet d'instancier certaines parties d'une application avec une configuration dynamique, et d'autres avec une configuration statique (ou plutôt *quasi statique* : en effet, en cas de configuration mixte, les méta-informations concernant les parties statiques ne peuvent pas être collectées par le ramasse-miettes, à cause des références des contrôleurs dynamiques vers les contrôleurs statiques).

On peut par exemple instancier les composants représentés sur la figure 12 comme indiqué sur la figure 13.

Le composant racine offre l'interface `LifeCycleController`, implantée par la classe optimisée. Ses deux sous-composants directs offrent également l'interface `LifeCycleController`, implantée cette fois par la classe générique, utilisant des intercepteurs. Les deux composants primitifs restant sont eux instanciés en configuration statique : ils n'offrent pas d'interface de contrôle de cycle de vie, et peuvent utiliser `OptimizedBindingController` pour qu'il n'y ait aucune indirection entre les objets applicatifs qu'ils encapsulent (par contre les métadonnées sur les liaisons restent en mémoire, pour la raison expliquée ci-dessus, ce qui fait que la configuration de ces composants n'est pas complètement statique).

En cas de configuration mixte comme celle de la figure 13, il est important de noter que les composants configurés de façon quasi statique *peuvent* être reconfigurés dynamiquement, bien qu'ils n'offrent pas d'interface pour les arrêter dans un état stable, car le composant qui les englobe offre, lui, une telle interface. Il suffit simplement que les composants quasi statiques ne soient pas partagés et n'aient pas d'activité propre. Dans ce cas la suspension des appels entrants au niveau du composant englobant entraîne automatiquement la suspension des appels dans tous les sous-composants, ce qui permet de reconfigurer ces derniers dans un état stable. On peut donc, par exemple, arrêter le composant englobant, changer les liaisons entre les sous-composants (grâce aux interfaces `BindingController` de ces composants, et aux métadonnées qui sont conservées en configuration « quasi statique », puis redémarrer le composant englobant.

On peut même effectuer de cette façon des reconfigurations plus radicales. Par exemple, on peut créer de nouveaux sous-composants similaires à ceux existant, mais possédant en plus une interface `LifeCycleController` (et les intercepteurs correspondant), les lier de la même façon que les sous-composants existants, « transvaser » les objets applicatifs encapsulés des anciens

	surcoût mémoire (mots)	surcoût en temps (μs)
pas d'optimisation	58	0.123
fusion des contrôleurs	42	0.124
fusion de tous les objets	34	0.092

Tab. 1 – *Performances avec contrôleur et intercepteur pour le cycle de vie*

	surcoût mémoire (mots)	surcoût en temps (μs)
pas d'optimisation	38	0.013
fusion des contrôleurs	28	0.011
fusion de tous les objets	26	0.011

Tab. 2 – *Performances sans contrôleur ni intercepteur pour le cycle de vie*

sous-composants vers les nouveaux[2], et enfin supprimer les anciens sous-composants. Autrement dit, on peut passer dynamiquement d'une configuration statique (ou plutôt « quasi statique ») à une configuration dynamique (ou l'inverse) à l'intérieur d'un composant englobant (à condition que celui-ci ait une interface `LifeCycleController` implantée à l'aide d'intercepteurs).

5.3 Évaluation des gains dans Julia

Afin de mesurer le gain apporté par les méthodes d'optimisation de Julia nous avons mesuré la taille d'un objet, ainsi que la durée d'un appel de méthode vide sur un objet, et nous avons comparé ces résultats à la taille d'un composant[3] (avec les différentes options d'optimisation) et la durée d'un appel de méthode vide sur un composant. Le composant ayant un contrôleur de liaison, ou un contrôleur de liaison et un contrôleur de cycle de vie (avec un intercepteur associé). Les résultats sont donnés dans les tables 1 et 2. Les mesures ont été faites sur un Pentium III 1GHz, avec le JDK1.3, HotSpotVM, au dessus de Linux. Dans ces conditions la taille d'un objet est de deux mots de 32 bits, et un appel de méthode vide sur une interface Java prend 0,014 μs.

Comme on peut le constater, les options de fusion de classes peuvent fortement réduire la taille d'un composant (fusionner plusieurs objets en un seul permet d'économiser plusieurs en-têtes d'objets, ainsi que plusieurs champs, qui étaient utilisés pour les références entre ces objets). Le surcoût en temps d'un appel de méthode sur un composant est du même ordre de grandeur que le temps d'un appel de méthode vide, lorsqu'il n'y a pas d'intercepteur : c'est normal puisque, dans ce cas, il n'y a qu'une seule indirection, via un objet `InterfaceReference`. Lorsqu'un intercepteur pour le contrôle du cycle de vie est présent le surcoût est beaucoup plus important : il est principalement dû au temps d'exécution de deux blocs déclarés `synchronized`, utilisés pour incrémenter et decrémenter un compteur avant et après chaque appel de méthode. Dans tous les cas ce surcoût est réduit lorsque toutes les classes sont fusionnées, ce qui est dû au fait que certains champs sont remplacés par `this` lors de la fusion (et il est plus rapide d'accéder à `this` qu'à la valeur d'un champ). Même s'il est important, le surcoût dû à l'intercepteur pour le cycle de vie est très inférieur au coût d'un intercepteur générique qui réifie complètement les appels de méthodes, comme dans un MOP, qui est de l'ordre de 4,6 μs pour une méthode vide, et de l'ordre de 9 μs pour une méthode de signature `int inc (int i)`.

2. « Transvaser » un objet d'un composant primitif dans un autre revient, en définitive, à modifier des références Java indiquant que tel objet applicatif est encapsulé dans tel composant primitif. Julia permet de faire cela, mais pour ce faire, il faut mettre à jour « à la main » ces références, sans en oublier, notamment dans les intercepteurs. Il serait probablement possible d'offrir des fonctions de plus haut niveau que celles actuellement fournies pour faciliter ce genre de manipulation.

3. La taille des objets qui représentent les types de composants ne sont pas prises en compte ici. Cette taille est de l'ordre de 75 (resp. 100) mots pour un composant ayant 3 (resp. 4) interfaces. C'est beaucoup (un type de composant contient plusieurs `String`), mais peu important si beaucoup de composants de même type sont instanciés, puisque ces objets sont partagés par tous les composants de même type.

Le temps pour instancier un composant encapsulant un objet vide, à partir d'un template Fractal, est de l'ordre de 4 ms (ou 3 ms sans l'aspect cycle de vie), sans compter le temps pour la génération dynamique de classes. Le temps pour créer le template lui-même est du même ordre de grandeur. Tous ces temps sont très importants, comparés au temps pour instancier un objet vide (0,3 μs). Il y a plusieurs raisons à cela : instancier un composant nécessite d'instancier plusieurs objets, beaucoup de vérifications sont faites lors de l'instantiation d'un composant et, surtout, les performances des phases de (re)configuration n'étaient pas une priorité lors de l'implantation de Julia. Des investigations plus poussées sont nécessaires pour trouver d'où viennent ces coûts, et pour essayer de les réduire.

6 Conclusion

Munir un logiciel de capacités d'adaptation implique potentiellement un coût sur les performances de ce logiciel. Au travers de différentes expériences menées au sein du projet, nous avons quantifiés ces coûts et explorés différentes manières de les réduire.

Notons tout d'abord que cette adaptation peut se faire à l'exécution, de manière non anticipée, grâce à des techniques d'injection et de réécriture de code dynamique. C'est ce que permet μDYNER, avec un surcoût réduit. Une contrainte est toutefois que la réécriture dynamique de code soit possible, ce qui est le cas pour les applications, écrites en C, auxquelles s'intéresse μDYNER. Dans le cas d'applications écrites en Java, qui impose des contraintes permettant de raisonner plus aisément sur la sécurité des applications, cette réécriture n'est pas possible à l'exécution et ne peut s'effectuer au plus tard qu'au chargement. Dans ce cas, un minimum d'anticipation est nécessaire.

Dans ce cas, la réflexion fournit un cadre général permettant de définir et de « peupler » des points d'adaptation. Une utilisation naïve de la réflexion est toutefois rédhibitoire. Une infrastructure comme Reflex donne la possibilité d'appliquer la réflexion de manière ciblée et de définir des protocoles de communication spécialisés avec les adaptations. Alternativement, il est possible d'utiliser, sous la forme d'objets d'interposition (ou intercepteurs), ce que l'on pourrait voir comme une implémentation ad hoc de la réflexion[4].

La séparation code de base/code d'adaptation ainsi introduite a toujours un surcoût à l'exécution. S'il n'y a pas de nécessité d'adaptation dynamique, ce surcoût peut être éliminé en effectuant une injection statique du code d'adaptation dans le code de base. On peut aussi parler de « fusion » du code de base et du code d'adaptation. Cette fusion peut s'effectuer de manière ad hoc, on en a vu un exemple. Mais, il est aussi possible d'envisager l'utilisation de techniques automatiques. Ainsi, dans certaines conditions, il est possible d'éliminer statiquement la réflexion par des techniques d'évaluation partielle [3].

Dans le cas de la plateforme Fractal/Julia, dont la structure est très riche, les possibilités d'optimisation sont importantes. Il est actuellement possible de configurer chaque composant en fonction des capacités d'adaptation dynamique dont on veut le munir, et d'optimiser cette configuration en fusionnant les contrôleurs, voire l'ensemble de la membrane d'un composant. Il est aussi possible de simplifier les liaisons inter-composants en l'absence d'objets d'interposition. Des premières micro-évaluations montrent que les bénéfices de ces optimisations ont un impact important sur la taille mémoire et un impact moindre sur le temps d'exécution. Il serait toutefois utile de compléter ces évaluations, notamment sur de véritables applications. On aimerait aussi pouvoir combiner ces optimisations avec une fusion contenu/membrane. La question est là de savoir quel est le meilleur moyen d'atteindre ce but : combiner des optimisations ad hoc (qui ont l'avantage de pouvoir reposer sur de l'information sémantique relative aux concepts de la plateforme), se baser sur des outils plus généraux d'optimisation (comme l'évaluation partielle [10] ou le découpage de programme [17]) traitant du langage d'implémentation (c'est-à-dire Java), ou une combinaison des deux. C'est cette dernière approche qui semble la plus prometteuse et rend envisageable d'étendre

4. Une étude précise du passage de la réflexion aux objets d'interposition tant au point de vue des principes que des performances reste à faire.

le périmètre des optimisations citées pour prendre en compte l'optimisation du contenu du composant, qu'il soit primitif ou composite (quelques pistes dans cette direction sont explorées dans [2]). Dans ce contexte, une question difficile est toutefois de garantir l'efficacité du code optimisée, d'autant plus si ce code est généré à l'exécution.

Un autre aspect du problème dont l'étude demande à être approfondie est la possibilité de remettre en question des optimisations, soit en conservant la version générique du code, soit en régénérant tout ou ou des parties du code à la volée. Un point difficile est alors de balancer degré d'optimisation et durée d'utilisation du code.

Références

[1] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *38th IEEE Computer Society International Conference (COMPCON'93)*, Feb. 1993.

[2] G. Bobeff and J. Noyé. Molding components using program specialization techniques. In J. Bosch, C. Szyperski, and W. Weck, editors, *Eighth International Workshop on Component-Oriented Programming (WCOP 2003)*, Darmstadt, Germany, July 2003.

[3] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, pages 2–11, Boston, MA, USA, Jan. 2000. ACM Press.

[4] E. Bruneton. *Julia Tutorial*. France Telecom R&D, 2.0 edition, Nov. 2003. `http://fractal.objectweb.org/tutorials/julia/index.html`.

[5] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, Spain, June 2002.

[6] B. Buck and J. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[7] L. DeMichiel. *Enterprise JavaBeans^{TM} Specification*. SUN Microsystems, Nov. 2003. Version 2.1, Final Release.

[8] D. Hagimont and L. Ismail. A protection scheme for mobile agents on Java. In *Proceedings of the 3rd Anual ACM/IEEE International Conference on Mobile Computing and Networking*, 1997.

[9] D. Hagimont and D. Louvegnies. Javanaise: Distributed shared objects for internet cooperative applications. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Sept. 1998.

[10] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[11] T. Ledoux (ed.). État de l'art sur l'adaptabilité. Livrable D1.1, Projet RNTL ARCAD, Dec. 2001.

[12] Object Management Group. CORBA components. Adopted Specification formal/02-06-65, OMG, June 2002. Version 3.0.

[13] *OOPSLA 2003, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, USA, Oct. 2003. ACM Press.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC : A flexible solution for aspect-oriented programming in Java. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2001.

[15] A. G. Popovici A., Gross T. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, Enschede, The Netherlands, 2002. ACM Press.

[16] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In OOPSLA2003 [13].

[17] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.