Des techniques de contrôle de l'envoi de messages en Smalltalk*

Stéphane Ducasse
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Switzerland
ducasse@iam.unibe.ch,
http://www.iam.unibe.ch/~ducasse/
Technical Report IAM-97-004

Abstract

Note pour le lecteur : cet article est une version détaillée de l'article de même titre apparu dans le numéro spécial Smalltalk de la revue l'Objet, Hermes Edition, 1998.

Note for the reader: This article is a detailled version of the article with the same title appeared in the special issue on Smalltalk of the L'Object journal, Hermes editor 1998.

Dans un langage tel que SMALLTALK où les objets ne communiquent que par envoi de messages, le contrôle des messages est un outil fondamental permettant d'analyser le comportement des objets (trace, espionnage) ou de définir de nouvelles sémantiques (messages asynchrones, objets distants). Pour la mise en place du contrôle, différentes techniques sont utilisées dont la plus connue est la redéfinition de la méthode `doesNotUnderstand:` couplée à la définition d'objets dit *minimaux* encapsulant l'objet devant être contrôlé. D'autres techniques existent cependant comme l'utilisation de l'algorithme de recherche des méthodes défini par la machine virtuelle ou la modification du code des méthodes contrôlées. Dans cet article, nous comparons ces techniques selon les aspects réflexifs utilisés, l'étendue du contrôle, les limites, le coût en terme d'efficacité et la facilité de mise en œuvre de chaque technique.

Mots-clés : contrôle des messages, objet minimaux, doesNotUnderstand:, récupération d'erreur, spécialisation d'instances, compilation de méthodes.

In a language like SMALLTALK in which objects communicate only via message passing, message passing control is a fundamental tool for the analysis of object behavior (trace, spying) and for the definition of new semantics (asynchronous messages, proxy,...). Different techniques exist, from the well known approach based on the specialisation of the `doesNotUnderstand:` method and the definition of minimal objects to the use of the method lookup algorithm done by the virtual machine. Until now, no comparison between these techniques has been made. In this article we compare the different techniques taking into account the reflective aspects used, the scope, the limit and the cost of the control.

Keywords: message passing control, minimal objects, doesNotUnderstand:, error handling, instance specialisation, method compilation.

CR categories and Subject Descriptors: D.1.5 [Object-Oriented Programming]; D.2.2 [Tools and Techniques] (Message passing control); D.3.2 [Language Specifications] (Extensible languages, Object-Oriented Languages)

*This research is supported by the Swiss National Science Foundation, grant 2000-46947.96

1 Du besoin du contrôle de l'envoi de messages

Dans un monde "tout objet" tel que celui de SMALLTALK dans lequel l'unique moyen de communication est l'envoi de message, le contrôle des messages est un formidable instrument permettant d'introduire des notions qui sans cela auraient été difficiles à mettre en œuvre telles que des objets distribués[Ben87], des objets atomiques, des moniteurs [Pas86]... Notons dès à présent que, comme nous le montrons par la suite, le contrôle de l'envoi de messages n'implique pas nécessairement la réification de ceux-ci [Fer89, DBFP95].

Dans le modèle à classes, une classe définit le comportement d'un ensemble d'objets : ses instances. Or il est parfois nécessaire de pouvoir associer un comportement spécifique à un objet, sans pour autant changer le comportement des autres instances de sa classe. Kent Beck nomme ce procédé la *spécialisation d'instances* [Bec93b, Bec93a]. D'autre part, outre cette nécessité d'associer un comportement spécifique à un objet donné, il est parfois nécessaire de changer le comportement *d'un ensemble d'objets* sans pour autant vouloir modifier le code des classes de ces objets. En effet, un tel changement de comportement peut être temporaire et dynamique comme par exemple lors de l'espionnage du comportement d'objets. Enfin, la séparation des méta-comportements (trace, changements de sémantique des messages, ...) des comportements intrinsèques des objets permet d'avoir une meilleure lisibilité et réutilisabilité des programmes *(separation of concerns* [KdRB91]*)*.

Objectif de l'article. Différentes techniques peuvent être mises en œuvre pour contrôler les messages. La plus populaire est sans doute la définition d'objets *minimaux* couplée avec la redéfinition de la méthode `doesNotUnderstand:`. Bien que d'autres techniques souvent plus efficaces existent, aucune étude comparative n'a, à notre connaissance, été menée. Cet article se propose de mettre en lumière ces différentes techniques et de les comparer.

Dans cet article, nous abordons exclusivement les techniques qui permettent d'introduire un contrôle de l'envoi de messages *standard* depuis le langage lui-même tout en excluant les approches basées sur la définition de *méta-interprètes*. Il s'agit de pouvoir contrôler des objets définis dans l'environnement SMALLTALK [1] standard. C'est pourquoi, nous ne discutons donc pas d'approches basées sur la définition d'un nouvel envoi de message explicite comme cela pourrait être fait lors de la définition d'un mini-Smalltalk écrit en SMALLTALK.

Remarquons que SMALLTALK étant basé sur un interprète[2] de *byte-codes* [GR89, IKM+97], il est possible d'augmenter son jeu d'instructions pour y introduire un contrôle des messages [YT87]. Cependant, comme une telle solution rend l'interpréte spécifique et les applications construites à base de contrôle des messages ne sont alors plus portables, donc nous ne la présenterons pas.

Le plan de l'article est le suivant : nous examinons rapidement les différentes motivations qui ont conduit à l'utilisation du contrôle de l'envoi de messages en 1.1, nous définissons des critères de comparaisons entre les diverses techniques en 1.4 et rappelons les aspects réflexifs de SMALLTALK qui sont utilisés. Ensuite, nous présentons les différentes techniques : récupération d'erreurs (section 2), utilisation de l'algorithme de recherche de méthode défini par la machine virtuelle (section 3), modification du code des méthodes compilées (section 4) et simulation de la recherche de méthodes (section 5). Enfin nous concluons par une discussion présentant les travaux relatifs mis en œuvre dans d'autres langages à objets.

[1] Nous ne discutons pas non plus de l'approche utilisé par le débogeur de SMALLTALK qui est un interprète de byte-code écrit en SMALLTALK pour des raisons d'efficacité.

[2] Notons que les implémentations industrielles optimisent le langage par la compilation et l'invocation de code natif. Ces implémentations font appel à de la compilation JIT : Just in Time.

1.1 Une première classification

Les travaux utilisant le contrôle des messages peuvent être classés en plusieurs catégories :

Analyse des applications et introspection. Certains travaux utilisent le contrôle des messages afin d'aider à la compréhension des programmes (trace, graphes d'appels, débogage) [BH90, PWG93, Bra96, Mic96].

Extension du language. Le contrôle des messages permet de définir de nouvelles fonctionnalités au sein même du langage. Ainsi la gestion de la distribution d'objets peut être introduite de manière transparente au code de base (Garf [GGM95], Distributed Smalltalk [Ben87], [McC87]). La sémantique du langage peut être étendue pour prendre en compte l'héritage multiple [BI82] ou la notion d'interfaces à la JAVA [Sch96]. La définition de *messsages atomiques* [Pas86, Lal90] ou de *futures* [FJ89, McA95a] sont aussi basées sur un contrôle des messages.

Définition de nouveaux modèles objets. Le contrôle des messages est aussi utilisé pour définir de nouveaux modèles objets introduisant des aspects concurrents comme des objets actifs et des messages asynchrones (Actalk [Bri89]) et la synchronisation entre messages asynchrones [YT87] (Concurrent Smalltalk[3]). D'autres travaux proposent de nouveaux modèles à objets comme les filtres de composition [ABV92], l'introduction d'une programmation dirigée par les instances [Bec93b, Bec93a, Hop94] ou la prise en compte d'entités responsables de la connexion entre objets [Duc97]. CODA est un protocole à méta-objets contrôlant des différentes activités d'un objet distribué [McA95b] essentiellement basé sur un contrôle des messages. De même, [Bez87] utilise la possibilité de contrôler les messages afin d'introduire des schémas de contrôle entre objets actifs.

Notons que Siemens-Object5, un langage d'acteur fortement typé dédié à l'automatisme, utilise tous les aspects réflexifs de SMALLTALK [Sie94].

1.2 Aspects réflexifs de SMALLTALK

Avant d'aborder les différentes approches, nous présentons très succintement les mécanismes réflexifs de SMALLTALK qui sont à la base des approches de contrôle de messages présentées. Il ne s'agit donc pas d'une présentation complète des aspects réflexifs de SMALLTALK, le lecteur peut se référer à [GR89, Riv96a, Riv96b, Riv97] pour une plus ample description. Les dialectes SMALLTALK référencés sont : VISUALWORKS anciennement nommé ObjectWorks de ParcPlace (nouvellement ObjectShare), IBM SMALLTALK intégré à l'environnement VISUALAGE de IBM et VISUALSMALLTALK anciennement Smalltalk/V puis Parts de Digitalk.

Réification et création dynamique. En SMALLTALK, les classes et les méthodes sont des objets comme les autres. Il est non seulement possible comme dans JAVA [Fla97] d'accéder aux informations que représentent ces entités mais aussi de modifier et de créer dynamiquement des instances de ces classes [4]. Comme nous le verrons cette distinction est fondamentale et donne à SMALLTALK un pouvoir d'expression plus riche que celui proposé dans la nouvelle version de JAVA.

Ainsi il est possible d'accéder et de modifier le lien d'héritage, le dictionnaire de méthodes, les méthodes contenues (méthodes `compiledMethodAt:`, `findSelector:`

[3] Concurrent Smalltalk est basé pour sa part sur l'extension de la machine virtuelle et la définition de nouveau byte-code. Cependant la synchronisation des messages asynchrones utilise un contrôle des messages à base de `doesNotUnderstand:`.

[4] En JAVA, il n'est pas possible de créer dynamiquement de nouvelles classes ou méthodes.

de la classe `Behavior` en VISUALWORKS). Etant donnée une méthode, il est possible de savoir la classe pour laquelle elle est compilée. D'autre part, il est possible de créer des classes dynamiquement (méthode `subclass:instanceVariableNames: classVariableNames:poolDictionaries:category:` définie sur la class `Class` en VISUALWORKS). De plus, la méthode `perform:with:` définie sur la classe `Object` permet d'envoyer explicitement un message à tout objet du système. `anObject perform: #zork with:12` envoie à l'objet `anObject` le message de sélecteur `zork` avec l'argument `12`.

En VISUALWORKS, des méthodes instances de la classe `CompiledMethod` peuvent être créées en invoquant la méthode `compile:notifying:`[5] de la classe `Behavior`. Finalement, il est possible d'invoquer l'exécution d'une méthode (méthode `valueWithReceiver:arguments:` de la classe `CompiledMethod` en VISUALWORKS et `executeWithReceiver:andArguments` en IBM SMALLTALK). Notons que cette dernière possiblité n'existait pas dans les premières implémentations de SMALLTALK et que cela explique pourquoi de nombreux travaux n'ont pas utilisé cette possibilité. D'autre part, certaines des fonctionnalités présentées ici sont qualifiées de privées dans les classes ce qui signifient qu'elles sont suceptibles de changer et ne sont pas standard.

Changement d'identité. La primitive `become:` permet de changer l'identité d'un objet en échangeant toutes les références pointant sur deux objets. Après invocation de `a become: b`, tous les pointeurs pointant sur `a` pointent sur `b` et vice et versa. Il est alors possible de considérer `a` comme une instance de la classe de `b`. Notons que la sémantique de cette primitive dépend en fait des implémentations de SMALLTALK: elle est symétrique (l'échange se fait dans les deux sens) en VISUALWORKS et asymétrique en IBM SMALLTALK.

Changement de classe. Un objet peut changer dynamiquement de classe, d'une classe *source* vers une classe *cible*. Un changement de classe peut être assimilé à une manipulation de pointeurs lorsqu'il n'y a pas de différence de structure entre les classes [Riv97]. En VISUALWORKS, la méthode `changeClassToThatOf:` attend comme argument un objet contrairement à la méthode `fixClassTo:` de IBM SMALLTALK qui prend comme argument une classe [6].

Pour qu'un changement de classe puisse se faire il faut qu'il y ait compatibilité de format entre les classes[7] source et cible. Le format d'une classe décrit la structure *(layout)* de stockage en mémoire de ses instances. L'accès au format d'une classe est assuré par les méthodes `format`, `setFormat:` définies sur la classe `Behavior` en VISUALWORKS, et `instanceShape`,`instanceShape:` définies sur la classe `Class` en IBM SMALLTALK.

Redéfinition de la récupération d'erreur. Lorsqu'un objet reçoit un message qui lui est inconnu, la machine virtuelle de SMALLTALK envoie le message `doesNotUnderstand:` à cet objet avec comme argument une réification partielle[8] du message ayant provoqué l'erreur. Sur la classe `Object` racine du graphe d'héritage,

[5] Cette méthode ne sauve pas le code source de la méthode.

[6] Les concepteurs de VISUALWORKS s'assurent ainsi que la classe cible est une classe instanciable et ceci sans avoir à faire ce test au niveau de la machine virtuelle.

[7] Plus précisément il faut : (1) étant donné un objet alloué en mémoire, étant donné le format de la classe cible, que les accès à toutes les variables d'instance définies par la nouvelle classe soient valides pour cet objet et (2) que l'objet alloué en mémoire corresponde au format de sa nouvelle classe.

[8] En VISUALWORKS, les messages invoqués par l'intermédiaire de la pseudo-variable `super` ne sont pas correctement réifiés lors de la gestion d'erreur.

cette méthode lève une erreur qui si elle n'est pas rattrapée (unhandled exception) provoque l'ouverture du débogueur[9].

1.3 Quatre principales techniques

L'étude des mécanismes utilisés pour contrôler les messages en SMALLTALK fait apparaître quatre grandes tendances que nous abordons en détail par la suite :

1. le rattrapage d'erreur et l'introduction d'objet *minimaux* [Pas86, Bri89, PWG93],
2. l'utilisation de l'algorithme de recherche des méthodes défini dans la machine virtuelle. Ceci peut être réalisé par la spécialisation explicite (sousclassage) ou implicite des méthodes (introduction de classes anonymes dans la chaîne d'instanciation [FJ89, McA95a, Mic96, Riv97], ou de tableau de dictionnaires de méthodes en VISUALSMALLTALK [Bec93b, Bec93a, Pel96]),
3. le changement du code des méthodes par utilisation du compilateur [BH90, Bou95, Bra96, Riv97] et
4. la simulation de la recherche de méthode [Mic96].

Remarque. La réification des messages peut permettre une interprétation spécifique de la sémantique des messages (continuations, asynchrones,...). Cependant, avec la seule réification des messages il est impossible de contrôler des objets spécifiques [Fer89, DBFP95]. De plus, comme le fait remarquer Adèle Goldberg dans [GR89] la réification des messages n'a été introduite en SMALLTALK que pour la gestion des erreurs et ceci pour des raisons d'efficacité. Par contre, la combinaison d'un contrôle pouvant être spécifique à une instance et la réification des messages offre une large gamme de possibilités : ainsi en CODA le contrôle peut être spécifique à chaque instance et les différentes sémantiques de messages sont basées sur la réification des messages à l'aide du mécanisme d'erreur de SMALLTALK [McA95a].

1.4 Quelques critères de comparaisons

Avant d'aborder par le détail chacunes des techniques, nous présentons quelques critères de comparaisons.

Efficacité. Pour la comparaison des coûts d'exécution, on considère que le code effectué lors du contrôle, comme l'affichage dans le cas d'une trace, est constant pour chacune des techniques. Le coût par lui-même ne prend en compte que les mécanismes mis en œuvre pour l'exécution de la méthode invoquée.

Granularité du contrôle. Le contrôle des messages est multi-forme : il peut s'agir de contrôler un message spécifique envoyé à un objet particulier ou au contraire de contrôler tous les messages envoyés à toutes les instances d'une classe.

Pour la granularité des objets contrôlés, on peut contrôler toutes les instances d'une classe de la même manière, ou ne contrôler que certaines instances (plusieurs objets peuvent partager un même contrôle sans pour autant appartenir à une même classe) ou encore contrôler des instances individuellement. Nous nommons la première possibilité un *contrôle de classe*, la seconde un *contrôle de groupe* et la dernière un *contrôle individuel*.

[9] Dans les applications *strippées* le processus s'arrête.

D'autre part, la question se pose de savoir si tous les messages envoyés à une instance sont obligatoirement contrôlés ou s'il est possible de ne contrôler que certains d'entre eux. Nous nommons la première possibilité une granularité *globale* et la seconde une granularité *sélective*.

Dynamicité de la prise de contrôle. Il faut aussi aborder la possibilité de dissocier la mise en place du mécanisme contrôlant de la prise effective du contrôle d'un objet ; c'est-à-dire est-il possible de spécifier indépendamment du mécanisme de contrôle le moment à partir duquel l'objet sera effectivement contrôlé ? Nous nommons la première une prise de contrôle *découplée* du mécanisme et la seconde *couplée*.

Intégration dans l'environnement. SMALLTALK propose un environnement très riche de programmation, nous considérons donc l'impact des techniques de contrôle sur les outils de l'environnement. Ainsi il est important de savoir si les outils comme les flâneurs et leurs fonctionalités (senders, implementors, message, class references, instance variable references,...) continuent de fonctionner.

Coût de mise en oeuvre. Finalement, il nous faut aborder si la solution proposée est aisément reproductible ou si elle demande une mise en oeuvre complexe.

Vocabulaire. Nous qualifions de *contrôlantes* les entités (classes, méthodes) qui implémentent le contrôle de l'envoi de messages. Nous nommons *originales* les méthodes devant être exécutées en absence de contrôle.

2 Rattrapage d'erreur et changement d'identité

Deux approches. Un grand nombre de travaux utilisent le mécanisme de récupération d'erreur pour introduire un contrôle des messages (voir en 1.2). Une distinction apparaît au niveau de la mise en place des conditions du déclenchement de l'erreur. Pour certains travaux qui nécessitent de contrôler des objets préexistants comme les objets de la bibliothèque de SMALLTALK (flâneurs,...) [Pas86, Lal90, PWG93, GGM95], la technique consiste à définir un objet dit *minimal*[10] et à substituer cet objet minimal à l'objet que l'on souhaite contrôler en utilisant la primitive `become:` (voir en 1.2). Pour d'autres travaux [Ben87, McC87, Bez87], il ne s'agit pas de contrôler des objets préexistants mais de définir des objets contrôlables. L'échange d'identité n'est alors pas nécessaire : un message d'erreur est invoqué car les messages envoyés à l'objet lui sont simplement inconnus. Par contre, pour cette dernière catégorie un contrôle des méthodes héritées de la classe `Object` nécessite leur redéfinition afin d'y inclure le contrôle et d'effectuer la substitution des appels aux primitives par des méthodes contrôlables [McC87].

2.1 Capsule et Spy

Contrôler des objets *a priori* non prévus à cet effet est un problème général dont la technique mise en place peut s'appliquer à des cas de prise de contrôle anticipé ; c'est pourquoi nous présentons maintenant en détail cette technique. Tout d'abord, nous présentons rapidement la réification du message original, ensuite nous abordons la définition d'objets *minimaux* et la mise en place du contrôle.

[10]Un objet minimal est un objet pour lequel *idéalement* tout message provoque une erreur. Le terme minimal vient du fait que pour être viable un tel objet doit possèder un minimum de méthodes ne provoquant pas d'erreur.

Réification des messages. Lorsqu'un objet ne sait pas répondre à un message[11], la machine virtuelle de SMALLTALK invoque la méthode `doesNotUnderstand:` avec une instance de la classe `Message` qui représente le message inconnu. Par exemple, le message `3 zork: 4` conduit à l'invocation `3 doesNotUnderstand: aMessage` où l'on obtient les informations suivantes :

```
aMessage selector -> #zork:
aMessage arguments -> #(4)
```

Objet minimal. Nous abordons maintenant la création d'un objet dit *minimal* [Bri89, PWG93], nommé aussi *capsule* ou *encapsulateur*. L'idée est de créer des objets n'héritant pas de la classe `Object` afin que tous messages envoyés génèrent une erreur et soient ainsi contrôlés.

La création d'une classe héritant de `nil`[12] n'aboutit pas à la solution souhaitée. En effet, SMALLTALK permet de créer de nouvelles classes destinées à devenir de nouvelles racines du graphe d'héritage. Pour ce faire, le protocole de création des classes est redéfini sur la classe `UndefinedObject` de manière à permettre la définition d'une classe n'héritant d'aucune autre classe et ayant donc la valeur `nil` comme superclasse. Pour que de telles racines d'héritage soient effectivement utilisables et compatibles avec l'environnement SMALLTALK, le système leur associe automatiquement une version de la méthode `doesNotUnderstand:` dont le but est de recopier automatiquement et paresseusement dans cette nouvelle classe racine les méthodes de la classe `Object` nécessaires. On aboutit donc à une copie incrémentale de la classe `Object`.

La technique adéquate pour créer un objet minimal se déroule en trois étapes : (1) création d'une sous-classe de `Object`, (2) affectation du lien d'héritage à `nil` et (3) définition du comportement minimal en copiant les méthodes nécessaires de `Object`.

La figure 1 illustre alors le contrôle des messages: (1) le message original est envoyé, (2) la machine virtuelle invoque la méthode `doesNotUnderstand:` et (3) la méthode originale est exécutée. Le message `doesNotUnderstand:`, idéalement l'unique message de cette classe (voir 2.3), peut définir un contrôle des messages de la façon suivante :

```
MinimalObject>>doesNotUnderstand: someMessage
 ..."controle des messages"
 self originalObject perform: someMessage selector
                     withAguments: someMessage arguments
 "invocation de la méthode contrôlée"
```

Mise en place du contrôle sur une instance. La mise en place du contrôle utilise la primitive `become:` de SMALLTALK qui échange toutes les références pointant sur deux objets (voir en 1.2). Ainsi l'objet contrôlé est *remplacé* par un object minimal (voir figure 1). Ce dernier ayant une référence sur l'objet qu'il encapsule afin de pouvoir invoquer le comportement original.

En VISUALWORKS, les objets référençant l'objet minimal avant l'échange référencent ensuite l'objet contrôlé. Notons que cela introduit la possibilité de court-circuiter le mécanisme de contrôle en permettant d'invoquer l'objet contrôlé directement et non par le biais de son objet minimal. Par contre, pour un objet minimal

[11] Cette situation se produit quand aucune classe dans la hiérarchie d'héritage de la classe de cet objet ne définit de méthode pour le sélecteur du message.

[12] En SMALLTALK, `nil`, qui est l'unique instance de la classe `UndefinedObject`, est une pseudo-variable dont la valeur signifie une référence pointant nulle part. C'est la valeur associée par défaut aux variables d'instances.

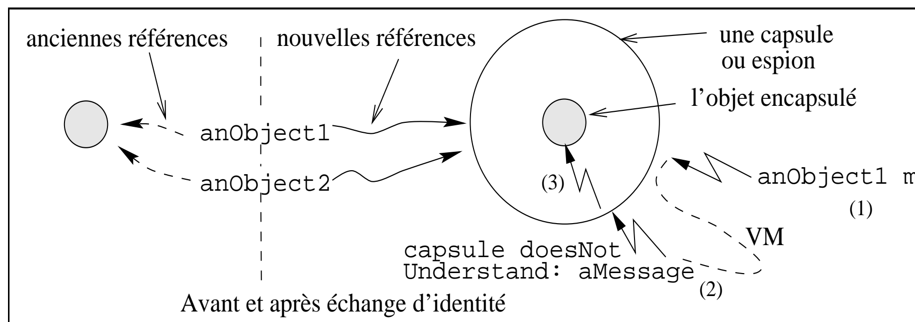

Figure 1: Mise en place du contrôle des messages par échange des identités et contrôle de l'envoi de message par génération et contrôle des erreurs

venant d'être nouvellement créé, un échange unidirectionnel est suffisant (comme en IBM Smalltalk et VisualSmalltalk). Les références pointant sur l'objet minimal peuvent être perdues car sans grande importance.

2.2 Problèmes

Trois problèmes ont été identifiés dans cette approche [PWG93].

Contrôle des messages utilisant `self`. La variable `self` est une pseudo-variable grâce à laquelle les objets s'auto-référencent sans utiliser des pointeurs explicites. Les objets peuvent s'envoyer des messages qui ne passeront pas par le biais de l'objet minimal et donc qui ne seront pas contrôlés. D'autre part, ce problème ne se pose pas uniquement quand l'objet s'envoie un message à lui-même. Pour qu'un message soit contrôlable il faut (1) que l'émetteur et le receveur soient distincts et (2) que le pointeur que possède l'émetteur sur le receveur n'est pas été installé en faisant référence à `self` [PWG93]. Les auteurs des Spies proposent une solution délicate et coûteuse de mise en œuvre basée sur l'analyse dynamique de la pile d'exécution afin de détecter si les messages envoyés doivent être ou non contrôlés.

Contrôle des classes. Cette approche rend impossible le contrôle des classes car les classes ne peuvent être substituées par des objets de nature différente, ici un objet minimal.

Object minimal. Pour être viable dans l'environnement Smalltalk, un objet minimal doit posséder un comportement qui ne se réduit pas à la seule méthode `doesNotUnderstand:`. Une vingtaine de méthodes comme `class`, `isKindOf`,`=`, `==`, `instanceVarAt:`, `myDependents`... sont nécessaires pour que l'objet soit viable. Se pose alors le problème de l'interprétation par l'objet minimal de messages qui étaient destinés à l'objet contrôlé. Le problème est alors double car non seulement le message est exécuté par l'objet minimal mais en plus l'objet contrôlé ne reçoit pas le message. Pascoe propose comme solution de dupliquer complétement la hiérarchie d'héritage et de préfixer toutes les méthodes destinées aux objet minimaux d'un E[13][Pas86]. Bien que cette solution fonctionne, elle est passablement lourde de mise en œuvre et très grosse consommatrice d'espace mémoire.

[13]Pascoe nomme les objets minimaux Encapsulators.

2.3 Discussion

Cette approche propose un contrôle individuel de granularité globale : toutes les méthodes étant contrôlées. La prise de contrôle est couplée. Contrairement aux autres méthodes qui présupposent la connaissance des messages devant être contrôlés, cette approche est la seule à offrir la possibilité de contrôler tous les messages envoyés. Il n'est pas nécessaire de connaître à l'avance les messages suceptibles d'être contrôlés.

Outre les problèmes évoqués précédemment cette approche est peu efficace. En effet, le contrôle est basé sur l'échec préalable de la recherche de la méthode invoquée. Ainsi chaque contrôle nécessite une recherche supplémentaire[14] et un double parcours de la pile d'exécution due à la gestion des exceptions. De plus, chaque contrôle implique une réification d'un message. Notons que cette approche est simple de mise en œuvre lorsque l'on ne tente pas de résoudre tous les problèmes qu'elle pose et notamment ceux très importants liés à l'identité de l'objet, i.e. quelle est la sémantique de `self` et de la méthode `==`.

3 Utilisation de l'algorithme de recherche de méthode de la machine virtuelle

En Smalltalk, une classe définit la structure et le comportement de ses instances. Le comportement est défini à l'aide d'un dictionnaire de méthodes associé à la classe. Quand un objet reçoit un message, la recherche de la méthode à appliquer commence dans sa classe et se poursuit en suivant le lien d'héritage. De plus, il est possible d'invoquer une méthode masquée en utilisant la pseudo-variable `super`[15].

Contrôler les messages envoyés à un objet est possible si l'on interpose entre l'objet et sa classe d'origine une classe redéfinissant les méthodes recherchées (voir figure 2). Diverses solutions peuvent être implémentées comme un sousclassage explicite traditionnel ou un sousclassage implicite basée sur la définition de classes anonymes associées à chacune des instances.

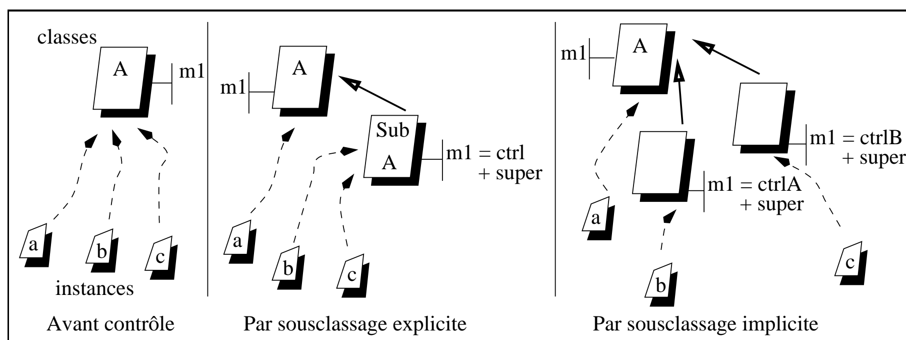

Figure 2: Deux approches de l'utilisation du sousclassage comme mise en œuvre d'un contrôle des messages - Ctrl représente le contrôle - : par sousclassage simple et par association à chaque objet contrôlé d'une classe anonyme.

Principe commun. Cette approche se décompose en trois temps : (1) création de l'entité contrôlante qui sera interposée entre l'objet et sa classe, (2) définition

[14] Notons que le mécanisme de cache de Smalltalk peut optimiser la recherche de la méthode `doesNotUnderstand:`

[15] Sa sémantique est de faire débuter la recherche de la méthode invoquée dans la superclasse de la classe dans laquelle elle est compilée. Ce mécanisme est comparable au `call-next-method` de Clos.

des méthodes contrôlantes dans cette entité et (3) changement de classe de l'objet (voir en 1.2). La définition des méthodes opérant le contrôle nécessite de créer des méthodes ayant le même sélecteur que la méthode originale.

3.1 Sousclassage explicite

La création de la classe interposée peut se faire par l'invocation de la méthode usuelle de création de classe. La méthode originale peut être invoquée de la méthode contrôlante par l'utilisation de la variable `super`.

Discussion. Le contrôle offert par cette approche possède : un contrôle de groupe et une granularité sélective. En effet, seules certaines instances peuvent être contrôlées et deux instances de la même classe partagent le même contrôle [16]. La prise de contrôle d'une instance est découplée de la mise en place du mécanisme de contrôle : la classe peut être créée et les instances contrôlées ultérieurement. La coût de mise en œuvre est relativement bas.

Supprimer le contrôle d'un objet nécessite de pouvoir changer de classe l'objet contrôlé (voir en 1.2). Le coût en terme d'exécution de méthode est celui de l'exécution d'une méthode normale. De plus, le fait que la prise de contrôle puisse être différée de la création de la classe apporte une flexibilité et une dynamicité dans l'application du contrôle. Un désavantage de cette solution est de nécessiter au minimum la création d'une classe *explicite*, ce qui ne permet pas à cette technique d'être complètement *transparente* du point de vue des objets contrôlés.

3.2 Sousclassage implicite : interposition de classes anonymes dans la chaîne d'instanciation

L'idée est d'introduire entre l'objet contrôlé et sa classe une classe anonyme et de définir dans cette dernière les méthodes contrôlantes. Ces méthodes implémentent alors la mise en oeuvre du contrôle et en particulier la façon d'invoquer la méthode originale (voir la discussion plus loin). Les étapes suivantes définissent la mise en œuvre du contrôle :

1. Création d'une classe anonyme, `nCl`, instance de la classe `Behavior`[17] en VISUALWORKS ou de la classe `Class` en IBM SMALLTALK (ou de leurs souclasses).
2. Copie de la description de la structure de la classe de l'instance (son format) dans `nCl` et affectation du lien d'héritage entre `nCl` et la classe originale de l'objet contrôlé.
3. Changement de classe de l'instance pour référencer `nCl` [18].
4. Compilation dans `nCl` des méthodes devant être contrôlées.

Cette technique est très brièvement évoquée dans [FJ89] qui nomme les entités interposées *lightweight* classes et est utilisée dans CODA [McA95a]. McAffer utilise cette technique à la fois pour implémenter des méta-objets et pour contrôler l'envoi de message à l'aide d'*interceptors* [McA95a]. Ernest Micklei propose une approche similaire [Mic96]. Cependant, la métaclasse subit elle aussi la même opération.

[16] Il serait possible de créer autant de classes que d'instances contrôlées mais il en résulterait une prolifération de classes explicites.

[17] D'après McAffer, Peter Deutsch mentionne que la classe `Behavior` avait été originalement définie pour permettre de telles manipulations [McA95a] p. 68.

[18] `fixClassTo:` ou utilisation de la primitive `VMprObjectClassColon` [Pel96] en IBM SMALLTALK 4.0 ou modification de la variable classField[McA95a] en IBM SMALLTALK 3.1 ou `changeClassToThatOf:` en VISUALWORKS

Cette approche est alors plus complexe de mise en œuvre. Pour la mise en œuvre du *pattern* de spécialisation dynamique [Riv97], NeoClasstalk utilise cette technique couplée à un changement du code des méthodes (voir en 4.3).

Implémentation en VisualWorks. La mise en place du contrôle pour `anObject` peut être la suivante. Les numéros de lignes correspondent aux étapes précédemment citées :

```
(1)  nCl := Behavior new
(2)           setInstanceFormat: anObject class format;
(2)           superclass: anObject class;
              methodDictionary: MethodDictionary new.
(3)  anObject changeClassToThatOf: nCl basicnew
```

Implémentation en IBM Smalltalk. Joseph Pelrine dans [Pel96] décrit une implémentation similaire à celle précédemment présentée.

```
(1) nCl := Class new
(2)         superclass: anObject class;
(2)         instanceShape: anObject class instanceShape
(2)              instVarNames: anObject class instVarNames;
            setMethodDictionary: MethodDictionary new.
(3) anObject changeClassTo: class
```

La méthode **`changeClassTo:`** devant être alors définie comme suit :

```
Object>>changeClassTo: aClass

 <primitive: VMprObjectClassColon>
 ^self primitiveFailed
```

Intégration et sémantique de `class`. L'intégration à l'environnement de programmation nécessite de redéfinir localement à la classe anonyme la méthode `class`. Sans cela le contrôle ne serait pas transparent car le système ferait toujours référence à la classe anonyme. Cette méthode peut être compilée sur l'instance de la classe anonyme interposée comme le montre le code de la méthode suivante. Notons que nous définissons aussi un accès à la classe anonyme depuis l'objet contrôlé.

```
AnonymousClass>>installEssentialMethods

self basicCompile: 'class ^super class superclass' notifying: nil.
self basicCompile: 'isControlled ^true' notifying: nil.
self basciCompile: 'anonymousClass ^super class' notifying: nil.
```

Invocation de la méthode originale. La définition du contrôle dans la méthode contrôlante pose le problème de savoir comment invoquer la méthode originale. La solution la plus évidente est d'utiliser directement la pseudo-variable `super` comme on l'aurait fait lors d'une spécialisation de méthodes. Cette solution n'est applicable que si le contrôle n'est effectué que par l'objet lui-même ou par un objet appartenant à la même hiérarchie d'héritage. Cette solution n'est plus applicable lorsque le contrôle des messages est effectué par un autre objet, comme par exemple un méta-objet [McA95a, Duc97]. La solution dans ce cas est de définir l'appel via `super` dans un bloc `[super methode-originale args]` qui pourra être activé ultérieurement à l'aide du message `value`. Dans ce dernier cas, le coût est plus élevé car ce type de

bloc n'est pas optimisé par le compilateur et nécessite la création d'une fermeture lexicale.

Dans le cas d'un contrôle délégué à un autre objet (un méta-objet par exemple), le code *généré automatiquement* de la méthode contrôlante pour la méthode `setX:setY:` pourrait être le suivant, le méta-objet définissant le contrôle par le biais de la méthode `control:call:withArgs:`.

```
anAnonymousClass>>setX: t1 setY: t2
   ^self meta control: #setX:setY:
            call: [super setX: t1 setY: t2]
            withArgs: (Array with: t1 with: t2)
```

Exemple de code généré tiré de la version VisualWorks de FLO [Duc97].

3.3 La solution particulière de VisualSmalltalk

Contrairement à VisualWorks et à IBM Smalltalk dans lesquels chaque objet possède une référence sur sa classe qui elle détient le dictionnaire des méthodes, VisualSmalltalk utilise une autre implémentation. Pour autant cette différence d'implémentation permet aussi de contrôler l'envoi des messages en utilisant l'algorithme de recherche défini dans la machine virtuelle [Bec93b, Bec93a, Pel96].

En VisualSmalltalk, chaque objet possède une référence sur un tableau de dictionnaires de méthodes. Chaque dictionnaire de méthode possède une variable d'instance appelée `class` indiquant sa classe d'appartenance. De plus, les dictionnaires sont rangés par ordre croissant de la classe vers les superclasses.

En VisualSmalltalk contrôler l'envoi de messages au niveau d'une instance nécessite alors de (1) créer une copie du tableau de dictionnaire de méthodes spécifique à l'objet, (2) d'y ajouter en première place un nouveau dictionnaire de méthodes et (3) de définir dans ce dictionaire les méthodes contrôlantes.

```
(1)  nMethDictArray := anObject methodDictionaryArray copy.
(1)  anObject methoDictionaryArray: nMethoDictArray.
(2)  anObject addBehavior: MethodDictionary new.
(3)  association := Compiler compile: aString in: anObject class.
(3)  anObject methodDictionaryArray first add: association.
```

L'argument `aString` représente le source d'une méthode contrôlante.

3.4 Remarques générales

Ces approches possèdent un contrôle individuel et une granularité sélective. La prise de contrôle d'une instance est couplée à la mise en place du mécanisme de contrôle, bien qu'un mécanisme permettant de stocker temporairement l'entité anonyme soit possible. La mise en œuvre du mécanisme est simple et adaptable à différents dialectes. Cependant, une erreur lors de la mise en place du contrôle peut gravement endommager le système. En effet, les informations utilisées sont cruciales pour le système: les dictionnaires de méthodes pour l'envoi de messages et le format des instances pour le ramasse-miettes.

Tout en étant très flexible, ces méthodes restent efficaces : la recherche et l'exécution des méthodes définies dans la machine virtuelle sont utilisées à leur optimum. La prise de contrôle ne coûte que l'exécution d'une méthode et n'est pas basée sur un échec préalable de la recherche de méthodes comme c'est le cas dans la solution basée sur l'invocation de la méthode `doesNotUnderstand:` présentée en 2. Par contre, ces techniques nécessitent de savoir quelles méthodes doivent être contrôlées. Enfin, dernier point fondamental , ces méthodes ne posent pas

le problème de *l'identité* de l'objet contrôlé pour l'interprétation de certains messages car le receveur est l'objet contrôlé lui-même (voir 2.2). Les deux dernières techniques nécessitent la définition de méthodes contrôlantes, ce qui peut être considéré comme une consommation gênante de mémoire. Notons que la définition des méthodes contrôlantes peut être optimisée en stockant des skelettes de méthodes précompilées [Mic96].

4 Substitution du code des méthodes

En SMALLTALK, les méthodes définies dans une classe sont stockées dans le dictionnaire de méthodes de cette classe. Ce dictionnaire associe à chaque sélecteur de méthode (un symbole) une instance de la classe `CompiledMethod`[19].

En changeant la méthode compilée associée à un sélecteur, il est possible de contrôler les méthodes exécutées. L'objet reçoit un message, l'exécution de la méthode associée au sélecteur du message n'exécute pas la méthode originale mais une méthode qui lui a été préalablement substituée. Cette technique est utilisée dans TRACER [BH90] et par les METHODWRAPPERS [Bra96]. Elle est généralisée dans NEOCLASSTALK [Riv97].

La méthode originale peut être simplement stockée dans le dictionnaire des méthodes avec un nom différent comme dans TRACER [BH90] ou cette méthode peut être encapsulée dans la méthode contrôlante comme dans les METHODWRAPPERS [Bra96].

4.1 Méthodes cachées

Cette solution consiste à échanger le code de la méthode originale avec celui de la méthode contrôlante (voir figure 3). [Bou95] utilise cette technique pour la mise en œuvre de dépendances explicites entre objets.

L'exemple suivant illustre la transformation de code opéré dans le cas de la méthode `add:` de la classe `SortedCollection` suivante :

```
SortedCollection>>add: newObject
  |nextIndex|
  self isEmpty ifTrue:[^super addLast: newObject].
  nextIndex := self indexForInserting: newObject.
  self insert: newObject before: nextIndex.
  ^newObject.
```

```
SortedCollection>>hiddenSortedCollectionadd: newObject
  |nextIndex|
  self isEmpty ifTrue:[^super addLast: newObject].
  nextIndex := self indexForInserting: newObject.
  self insert: newObject before: nextIndex.
  ^newObject.

SortedCollection>>add: newObject
  |result |
  ...
  result := self hiddenSortedCollectionadd: newObject.
  ...
  ^result
```

`hiddenSortedCollectionadd:` est le nouveau sélecteur associé au code original de la méthode `add:` et le sélecteur `add:` est associé à la méthode contrôlante. Le code présenté qui est tiré de [BH90] a été modifié pour faire apparaître clairement

[19] En VISUALWORKS `CompiledMethod` est la classe des objets qui représentent une méthode. Des informations telles que le source, la classe, le code compilé en byte-code y sont représentées.

le procédé[20]. Les ellipses représentent des invocations aux actions à effectuer par le contrôle (trace,...). Dans ce cas précis les auteurs de TRACER mettaient en œuvre un mécanisme similaires aux méthodes auxiliaires *before:* et *after:* de Clos [Kee89][21].

Cas des méthodes non définies localement. Bien que les auteurs de TRACER ne l'évoquent pas, le problème du contrôle des méthodes héritées par la classe de l'objet contrôlé se pose. L'échange des méthodes doit rester local à la classe de l'objet contrôlé afin de ne pas contrôler par effet de bord des instances directes ou indirectes des superclasses de cette classe. Dans la figure 3, contrôler la méthode `m3` au niveau de la classe `A` entraine le contrôle de l'instance `c`. Une solution consiste à définir localement des méthodes contrôlées de même sélecteur que les méthodes héritées, ces méthodes invoquant alors le contrôle et le cas échéant la méthode originale.

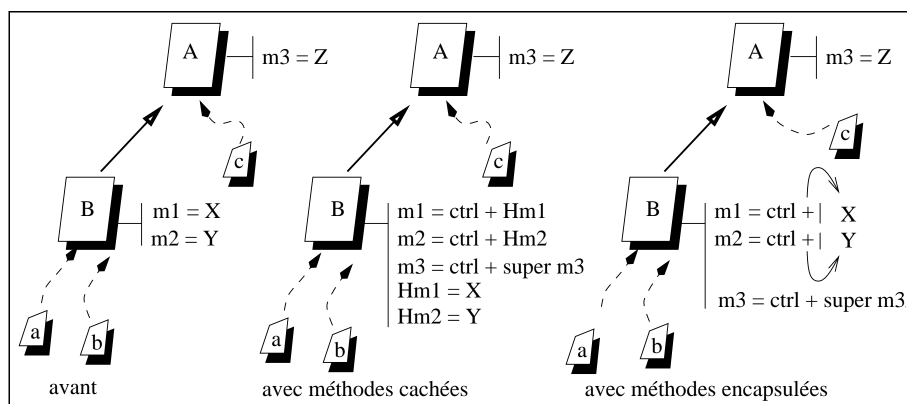

Figure 3: Mise en place du contrôle par modification des méthodes de la classe de l'objet que l'on souhaite contrôler.

4.2 Une variante : les MethodWrappers

La précédente solution a le désavantage d'introduire de nouveaux couples sélecteur-méthode dans le dictionnaire et de polluer l'interface de la classe des objets contrôlés. Bien qu'il soit peu probable qu'un utilisateur invoque une méthode ainsi cachée, cette solution peut être génante lors de l'*inspection* du système. L'auteur des MethodWrappers propose une variante qui a le mérite de ne pas stocker les méthodes originales au niveau du dictionnaire de la classe des objets contrôlés mais au niveau des méthodes elles-mêmes [Bra96]. Au lieu de créer une nouvelle association (sélecteur, méthode compilée) comme c'était le cas de `hiddenSortCollection-add:` dans l'exemple précédent, la méthode originale est substituée par une méthode l'encapsulant, c'est-à-dire une méthode compilée ayant une référence sur la méthode compilée associée au sélecteur que l'on souhaite contrôler.

4.2.1 Mise en œuvre.

Le code suivant décrit la classe MethodWrapper sousclasse de la classe `Compiled-Method`. La variable `clientMethod` représente la méthode compilée originale. La variable `selector` représente le sélecteur de la méthode originale.

[20] La méthode cachée était originalement écrite avec une majuscule.

[21] Ces invocations étaient implémentées par des invocations à la classe de l'objet : `SortedCollection send: #add: in: thisContext.`

```
CompiledMethod variableSubclass #MethodWrapper
   instanceVariableNames: 'clientMethod selector'
   classVariableNames: ''
   poolDictionaries:''
   category: 'Method Wrappers'
```

La méthode de classe `on:inClass:` rend une méthode (instance de la classe `MethodWrapper`) prête à être installée sur une méthode compilée à l'aide du message `install` dont le code est expliqué plus loin. L'exemple montre la mise en place du contrôle de la méthode `color` de la classe `Point`.

```
(MethodWrapper on: #color inClass: Point) install
```

4.2.2 Aspects techniques.

Nous présentons maintenant quelques points d'implémentation des MethodWrappers. Tout d'abord, à des fins d'optimisations, des prototypes des méthodes contrôlées sont générés et stockés en fonction du nombre de paramètres des méthodes. Ensuite, afin que le contrôle ne soit pas apparent pour l'utilisateur au niveau par exemple du flâneur, l'auteur utilise la possibilité d'associer un code source différent du byte-code de la méthode. Finalement, l'auteur évite l'utilisation de la pseudo-variable `thisContext`[22] par une manipulation explicite du *byte-code* représentant l'encapsulateur de méthode.

Prototypes. La méthode `on:inClass:` crée ou copie une instance de `WrapperMethod` correspondant aux nombres d'arguments de la méthode encapsulée (méthode `createMethodFor:`), puis modifie les caractéristiques de cette instance afin qu'elle puisse être appliquée sur l'objet, c-a-d. mise à jour du sélecteur de la classe `mclass` et changement des litéraux de la méthode (méthode `class:selector:`) .

```
MethodWrapper class>>on: selector inClass: class
  |wrp|
  (self canWrap: selector inClass: class) ifFalse:[ ^nil].
  wrp := (self methods at: selector numArgs ifAbsent:[
    self methods at: selector numArgs put:
              (self  createMethodFor: selector numArgs)]) copy.
   wrp class: class selector: selector.
  ^wrp

MethodWrapper>>class: aClass selector: sel
   self at: 1 put: self.
   mclass := aClass.
   selector := sel
```

Les prototypes d'encapsulateurs de méthodes ont un code générique. Le code suivant est le code d'un prototype d'une méthode ayant deux paramètres tel que l'on peut le voir en inspectant les éléments de la variable de classe `selectors`. Cependant, ce code n'est donc pas le code tel qu'il est finalement défini dans la classe de l'objet contrôlé. Ce point est expliqué en détail plus loin.

[22] En VisualWorks `thisContext` est une pseudo-variable représente le contexte d'exécution d'une méthode. Un contexte représente en autres l'émetteur et le receveur d'un message.

```
unboundMethodwith: t1 with: t2
  "       ***This is decompiled code.***
   This may reflect a problem with the configuration of your image
   and its sources and changes files. "

   ^#() receiver: self value: t1 value: t2
```

```
MethodWrapper>>receiver: object value: t1 value: t2
   | t |
   t := Array new: 2.
   t at: 1 put: t1; at: 2 put: t2.
   ^self valueFrom: object arguments: t

WrapperMethod>>valueFrom: object arguments: args
   self beforeMethod.
   ^[ clientMethod valueWithReceiver: object arguments: args]
       valueNowOrOnUnwindDo: [self afterMethod]
```

La méthode `receiver:value:value:` invoquera la méthode encapsulée par le biais de la méthode `valueFrom:arguments`[23]. La méthode `valueFrom:arguments:` met en place un mécanisme similaire aux méthodes auxiliaires de CLOS [Kee89] en invoquant la méthode originale[24].

Transparence. Le code source d'un encapsulateur de méthode ne représente pas son byte-code mais celui de la méthode qu'il encapsule. Cette différence entre le byte-code de l'encapsulateur et le source code qu'il présente permet un contrôle *transparent* des méthodes au niveau du flâneur. La méthode `install` installe l'encapsulateur effectivement sur la classe de l'objet contrôlé et met en place cette différence.

```
WrapperMethod>>install
   | definingClass |
   definingClass := mclass.
   [definingClass isNil or: [definingClass includesSelector: selector]]
       whileFalse: [definingClass := definingClass superclass].
   definingClass isNil ifTrue: [^self].
   clientMethod := definingClass compiledMethodAt: selector.
   sourceCode := clientMethod sourcePointer.
   mclass addSelector: selector withMethod: self
```

Référence à la méthode couramment invoquée. SMALLTALK n'offre pas de pseudo-variable pour faire référence à la méthode couramment invoquée. Or lors d'un envoi de message à un objet, il est nécessaire d'invoquer certaines méthodes sur l'encapsulateur (comme par exemple `receiver:value:`). Au lieu d'utiliser la pseudo-variable `thisContext` qui réifie à la demande le contexte d'exécution d'une méthode[25], l'auteur des METHODWRAPPERs modifie les litéraux[26] des méthodes générées. En effet, le code final après installation d'un encapsulateur n'est pas `^#() receiver:...` comme montré plus haut ; ce code était temporairement incorrect. Le litéral `#()` était utilisé afin de réserver de la place dans la structure représentant la méthode compilée. Ainsi lors de l'installation, ce code est remplacé par une référence

[23] Ces indirections peuvent être réduites à des fins d'optimisations, en générant un code implémentant directement le code de la méthode `valueFrom:arguments:`.

[24] à l'aide de la méthode `valueWithReceiver:arguments:` définie sur la class `CompiledMethod` en VISUALWORKS et `executeWithReceiver:andArguments:` en IBM SMALLTALK.

[25] La réification de `thisContext` est très coûteuse.

[26] Les litéraux sont des objets syntaxiques créés au moment de la compilation des méthodes.

à la méthode en cours d'exécution c'est-à-dire elle-même dans ce cas précis. Ceci est mis en œuvre par la première ligne de la méthode `class:selector:` invoquée lors de la création de l'encapsulateur par la méthode `on:inClass:`. Il faut noter qu'utiliser la pseudo-variable `self` dans le code du prototype de méthode montré plus avant n'était pas la bonne solution car `self` représentait alors l'objet sur lequel la méthode était invoquée et non elle-même.

4.3 NeoClasstalk

NeoClasstalk, qui est une implémentation de Smalltalk introduisant des méta-classes explicites [Riv97], permet notamment de définir des propriétés sur les classes, comme la trace des méthodes, des variables d'instances, des pré et post-conditions. Ces propriétés sont mises en place par la modification contrôlée du code des méthodes, introduisant de fait un contrôle des messages invoqués. NeoClasstalk utilise la même technique que les MethodWrappers en termes d'implémentations (prototypes et modification de litéraux) mais donne le contrôle à la classe du receveur[27]. De plus, NeoClasstalk utilise un changement dynamique de classe basée sur la définition de classes anonymes (voir en 3.2). NeoClasstalk propose un *framework* de composition des différentes politiques de contrôle, i.e. des propriétés introduites. Ainsi un méta-programmeur peut spécifier une partie du source qui sera automatiquement compilé dans les méthodes contrôlées.

Définition d'un contrôle. En NeoClasstalk, l'exécution d'une méthode est invoquée par la méthode `execute:receiver:arguments:` définie sur la classe `AbstractClass`. La définition de cette méthode est spécifiée par la méthode `generateBodyOn:` de la classe `TemporalComposition`, c-a-d que le code source de la méthode `execute:receiver:arguments:` est spécifié par la méthode `generateBodyOn:`.

Supposons que l'on souhaite introduire un contrôle des messages réalisant une trace des méthodes exécutées. Pour cela, il faut définir une nouvelle classe `TraceAllMessages` (sous-classe de `TemporalComposition`) et spécialiser la méthode `generateApplyBodyOn:` qui contrôle une partie de la génération du code de la méthode `execute:receiver:arguments:`.

Le code suivant montre l'ajout de la définition textuelle (source) d'une trace à la définition normale de la méthode.

```
TraceAllMessages>>generateApplyBodyOn: aStream
   aStream nextPutAll: '|window|
       window := self transcript.
                      cm printNameOn: window.
                      window cr; endEntry.'.
   super generatedApplyBodyOn: aStream
```

La mise en œuvre du contrôle sur la classe `Point` s'effectue en invoquant la méthode `temporalComposition:`.

```
TraceAllMessages new temporalComposition: Point.
```

`TraceAllMessages new` crée une classe implicite ayant des encapsulateurs de méthodes. `temporalComposition: Point` change dynamiquement la classe de l'objet (ici la classe `Point`) pour que son argument (`Point`) soit de la classe du receveur (la nouvelle classe implicite).

Aspects techniques. Une partie de la génération du code de la méthode `execute:receiver:arguments:` est spécifiée par `generateApplyBodyOn:`. Cette méthode

[27] Plus exactement le contrôle est donné au résultat du message `unreifiedClass`, la classe d'un objet pouvant être encapsulée.

est invoquée par la méthode `applyMethod` définie sur la classe `TemporalComposition` qui rend le code source *complet* de la méthode `execute:receiver:arguments:`. La méthode `applyMethod` assure ainsi un cadre sémantique pour le code de méthodes générées comme la certitude que la méthode originale sera invoquée (cf. `super execute:...` ci-dessous).

```
TemporalComposition>>applyMethod
"rec is the receiver, args are the arguments of the method
 cm is the currently reified method"

|ws|
ws := (String new: 100) writeStream.
ws nextPutAll: 'execute: cm receiver: rec arguments: args';crtab;
   nextPutAll: "system generated method";cr;crtab.
self generateBodyOn:ws.     "<- the method to override"
^ws contents
```

```
TemporalComposition>>generateApplyBodyOn: aStream

  aStream crtab;
          nextPutAll: '^super execute: cm receiver: rec arguments: args'
```

Notons qu'il est possible de changer la sémantique même du contrôle afin de soumettre l'exécution de la méthode au contrôle lui-même en changeant la définition même de la méthode `generateApplyBodyOn:`. L'implémentation de FLO [Duc97] en NeoClasstalk utilise cette possibilité.

4.4 Discussion

Ces différentes techniques possèdent en commun un contrôle de classe et une granularité sélective. En effet, toutes les instances d'une classe sont contrôlées sans possibilité de les sélectionner. La prise de contrôle d'une instance est couplée à la mise en place du mécanisme de contrôle : le contrôle est effectif dès la modification des méthodes. Le coût en terme d'exécution est celui d'un appel de méthode.

La première solution basée sur la définition de nouvelles méthodes au niveau de la classe des objets contrôlés pollue l'interface de celle-ci. Ce point est mieux géré par les MethodWrappers. Dans ces deux solutions, enlever le contrôle des méthodes consiste à réassocier les méthodes originales aux sélecteurs originaux et dans le premier cas à éliminer les entrées associées aux méthodes auxiliaires du dictionnaire de méthodes.

NeoClasstalk prend à sa charge la recompilation des méthodes et propose un cadre bien défini pour la définition du contrôle des méthodes ainsi que leur composition. Cependant, la solution proposée est complexe, et ce non pas par les concepts utilisés comme la recompilation automatique mais par la définition du framework à base de métaclasses supportant le mécanisme. Contrairement aux autres approches, la reproduction du mécanisme mis en place est délicate. Notons par contre que NeoClasstalk propose des outils au développeur lambda lui permettant de sélectionner les propriétés qu'il souhaite appliquer aux classes.

Notons enfin que l'énorme intérêt de contrôler les messages par utilisation des encapsulateurs de méthodes (MethodWrapper, NeoClasstalk et des méthodes présentées en 3.2) est que tous les outils définis dans les flâneurs (implementors, senders...) continuent de fonctionner contrairement aux approches basées sur le changement d'identité.

5 Simulation de la recherche de méthodes

5.1 Objet minimal par hiérarchie parallèle

Une solution hybride a été proposée par Micklei [Mic96]. Cette solution est basée sur un mélange du concept du rattrapage d'erreur avec celui de la duplication de classes. L'idée est d'implémenter le concept d'objet minimal par une hiérarchie d'héritage parallèle à celle de la classe de l'objet que l'on veut contrôler et d'utiliser le rattrapage d'erreur afin de rechercher la méthode invoquée dans la véritable hiérarchie. Il s'agit de simuler la recherche de méthode effectuée normalement par la machine virtuelle.

Mise en œuvre. S'il l'on souhaite contrôler une instance de la classe `Point`, on crée une classe dupliquée `Point*` ayant la même structure que la classe originale. `Point*` n'hérite pas de la classe `Point` (comme c'était le cas avec la solution présentée en 3), mais hérite de la classe dupliquée `Object*` qui contrôle les comportements définis par `Object`. D'autre part, la classe `Point*` définit la méthode `doesNotUnderstand:` de telle sorte qu'elle réalise la recherche de la méthode (méthode `findMethod:` dans la définition ci-dessous).

```
aClass>>doesNotUnderstand: aMessage
|lookup result|
lookup := self findSelector: aMessage selector
lookup isNil
    ifTrue: [self doesNotUnderstand:
                (aMessage selector: aSelector arguments: argsArray)]
    ifFalse: [|method|
              method := (lookup at: 2) copy.
              method mclass: self class.
              ^self performMethod: method arguments: argsArray]
```

L'instance contrôlée devient instance de la classe dupliquée. Ensuite tout envoi de message à cette instance invoque la méthode `doesNotUnderstand:` qui elle recherche la méthode invoquée dans la classe originale `Point` et l'applique à l'instance contrôlée. Il faut noter qu'il faut changer la classe d'appartenance de la méthode compilée obtenue lors de la recherche avant de l'appliquer à l'objet contrôlé.

Discussion. Le fait de définir une classe dupliquée par instance offre un contrôle individuel des instances. Le contrôle est global car toutes les méthodes sont contrôlées par défaut. De plus, la prise de contrôle d'une instance est couplée à la mise en place du mécanisme de contrôle.

Cette solution ne nécessite pas d'encapsuler l'objet original comme la solution introduisant la notion d'objet minimal. Le problème de `self` est résolu car lorsque l'objet contrôlé s'envoie un message, ce message redéclenche le mécanisme d'erreur et donc la méthode est recherchée comme toutes les autres méthodes contrôlées. Cependant, outre la relative complexité de cette approche, son plus gros désavantage est qu'en se substituant à la machine virtuelle pour la recherche des méthodes, elle est lente. Or, une solution idéale du contrôle se doit de limiter au maximum le surcoût entraîné par le mécanisme mis en œuvre pour contrôler les messages.

6 Conclusion

Cette comparaison fait apparaître que la technique la plus célébre et utilisée basée sur la récupération d'erreur `doesnotUnderstand:` n'est pas la meilleure et qu'éton-

namment peu de travaux utilisent les méthodes plus élégantes et efficaces qui consistent à utiliser l'algorithme de recherche des méthodes défini par la machine virtuelle ou à utiliser le compilateur de SMALLTALK. La possibilité d'invoquer l'exécution d'une méthode qui n'a été introduite que tardivement dans les interprètes est sûrement en partie responsable de ces choix.

6.1 SMALLTALK.

Les aspects réflexifs de SMALLTALK comme la possibilité de créer dynamiques des classes et méthodes et le fait que la réification du langage soit causalement connecté[28] à son fonctionnement offrent de nombreux avantages pour la définition d'extensions ou de modifications du langage. Nous avons illustré ces atouts en montrant comment la définition d'un contrôle de l'envoi de messages est possible par l'utilisation des différents mécanismes réflexifs de SMALLTALK [Riv96a, Riv96b]. L'uniformité de SMALLTALK couplée à la possibilité de modifier directement les objets qui représentent le langage lui-même permettent une grande expressibilité et flexibilité. Cet étude montre, si cela devait encore l'être, la puissance d'un langage comme SMALLTALK ou CLOS dans lequel la réflexivité ne se limite pas à une réflexion introspective comme dans JAVA.

6.2 Contrôle dans d'autres langages.

CLOS, qui est le langage objet intégré à Common Lisp [Kee89], est un des rares langages à classes à offrir la possibilités de définir des méthodes spécifiques à une instance en utilisant le *specializer eql* [Kee89]. D'autre part, CLOS est aussi un des rares langages à avoir un protocole de méta-objets (MOP) [KdRB91]. Le contrôle des messages est un des points d'entrée du MOP de CLOS. En CLOS le concept d'envoi de messages est remplacé par l'application de fontions génériques[29]. Le MOP de CLOS permet de contrôler les différents aspects de l'application d'une fonction générique : l'application de la fonction générique (`compute-discriminating-function`), l'application des méthodes la composant (`compute-effective-method--function`) ou l'application d'une seule méthode composant la fonction générique effectivement appliquée (`compute-method-function`)[30]. [HM90] utilise cette possibilité pour définir un mécanisme de notification.

Dans les langages à prototypes, MOOSTRAP permet un contrôle des messages basé sur la définition d'un protocole réflexif : le méta-objet d'un prototype étant responsable de la recherche (lookup) et de l'application de la méthode (apply) [MC93, Mul95].

Dans des langages moins flexibles, les travaux autour de C++ par la définition d'OpenC++ [CM93], que l'on peut percevoir dans sa dernière version comme un compilateur ouvert [Chi95], montre l'intérêt pour un contrôle des messages. Plus récemment, la définition de MetaJava offre la possibilité d'un contrôle des messages en JAVA [Gol97]. Dans cette dernière implémentation, des classes sont interposées entre l'instance et sa classe, il faut cependant noter que l'interpréte de JAVA est étendu par l'introduction de nouveau byte-code ce qui nuit à la portabilité du système.

JAVA dans sa dernière version 1.1 introduit la notion de réification de certains aspects importants du langage comme les classes, les méthodes et les variables

[28] On dit qu'un aspect réflexif est causalement connecté au mécanisme qu'il réifie si une modification de l'aspect réflexif a des répercussions immédiates sur le mécanisme et inversement.

[29] Une fonction générique regroupe un ensemble de méthodes étant sélectionnées dynamiquement lors de l'application de la fonction générique sur un ensemble d'arguments.

[30] Les implémenteurs de CLOS utilisent des techniques de caches pour minimiser l'appel des fonctions génériques permettant de spécialiser le contrôle des messages.

d'instances (Core Reflection API [Fla97]). Cependant, cette réification se veut avant tout *introspective*. En effet, les classes `Field`, `Method` et `Constructor` sont déclarées *finales* ce qui implique qu'elles ne peuvent pas être spécialisées. De plus, seule la machine virtuelle de JAVA peut créer des instances de ces classes. Seules les valeurs des variables d'instance sont modifiables. Les méthodes réifées peuvent être invoquées via la méthode `handle()`. Une telle approche est nécessaire afin d'offrir des outils comme les flâneurs de classes, le debugger de SMALLTALK en JAVA. Cependant, il n'est en aucun cas possible de modifier les classes ou les méthodes. Cette réification n'est pas complètement causalement connectée aux objets qu'elle représente et ne peut pas être utilisée pour modifier ou étendre le langage.

Remerciements. L'auteur tient à remercier Jeff McAffer, Ernest Micklei et François Pachet pour lui avoir donner accès à leurs sources, John Brant et Joseph Pelrine pour avoir répondu à ses questions. Il remercie également Pierre Cointe, Fred Rivard et Mireille Fornarino pour leurs commentaires.

References

[ABV92] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *ECOOP'92, LNCS 615*, pages 372–395, 1992.

[Bec93a] K. Beck. Instance specific behavior: Digitalk implementation and the deep meaning of it all. *Smalltalk Report*, 2(7), May 1993.

[Bec93b] K. Beck. Instance specific behavior: How and why. *Smalltalk Report*, 2(6), Mar 1993.

[Ben87] J. K. Bennett. The Design and Implementation of Distributed Smalltalk. In *OOPSLA'87*, pages 318–330, 1987.

[Bez87] J. Bezivin. Some experiments in object-oriented simulation. In *OOPSLA'87*, pages 394–405, 1987.

[BH90] H.-D. Böcker and J. Herczeg. What tracers are made of. In *OOPSLA/ECOOP'90*, pages 89–99, 1990.

[BI82] A. H. Borning and D. H. Ingalls. Mutiple Inheritance in Smalltalk-80. In *Proc. of NCAI AAAI*, pages 234–237, 1982.

[Bou95] M. Bouabsa. Elaboration et prototypage d'interfaces homme-machine au moyen de contraintes: le système plus, 1995. Thèse de l'Université deUniversité de Paris VI.

[Bra96] J. Brant. Method Wrappers. Smalltalk goody available at http://st-www.-cs.uiuc.edu/archive.html, see also http://st-www.cs.uiuc.edu/users/brant/-Applications/MethodWrappers.html, brant@cs.uiuc.edu, 1996.

[Bri89] J. Briot. Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In *ECOOP'89*, pages 109–129, 1989.

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

[CM93] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *ECOOP'93, LNCS 707*, pages 483–502, 1993.

[DBFP95] S. Ducasse, M. Blay-Fornarino, and A. Pinna. A Reflective Model for First Class Dependencies. In *OOPSLA'95*, pages 265–280, 1995.

[Duc97] S. Ducasse. Intégration réflexive de dépendances dans un modèle à classes, 1997. Thèse de l'Université de Nice-Sophia Antipolis.

[Fer89] J. Ferber. Computational reflection in class based object oriented languages. In *OOPSLA'89*, pages 317–326, 1989.

[FJ89] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *OOPSLA'89*, pages 327–336, 1989.

[Fla97] D. Flanagan. *Java in a Nutshell*. O'Reilly, 2nd edition, 1997.

[GGM95] B. Garbinato, R. Guerraoui, and K. Mazouni. Implementation of the GARF replicated objects plateform. *Distributed Systems Engineering Journal*, Mar. 1995.

[Gol97] M. Golm. Design and Implementation of a Meta Architecture for Java. Master's thesis, IMMD at F.A. University, Erlangen-Nuernberg, 1997.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation.* Addison-Wesley, 1989. ISBN: 0-201-13688-0.

[HM90] V. Haarslev and R. Möller. A framework for visualizing object-oriented systems. In *OOPSLA/ECOOP'90*, pages 237–244, 1990.

[Hop94] T. Hopkins. Instance-Based Programming in Smalltalk. Tutorial presented at the 2nd ESUG summer school, 1994.

[IKM+97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *OOPSLA '97*, 1997.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kee89] S. E. Keene. *Object-Oriented Programming in Common-Lisp.* Addison-Wesley, 1989.

[Lal90] W. Lalonde. *Inside Smalltalk (volume two).* Prentice Hall, 1990.

[MC93] P. Mulet and P. Cointe. Definition of a reflective kernel for a prototype-based langage. In *ISOTAS'93, LNCS 742*, pages 128–144, 1993.

[McA95a] J. McAffer. *A Meta-Level Architecture for Prototyping Object Systems.* PhD thesis, University of Tokyo, 1995.

[McA95b] J. McAffer. Meta-Level Programming with CodA. In *ECOOP'95, LNCS 952*, pages 190–214, 1995.

[McC87] P. L. McCullough. Transparent Forwarding: First steps. In *OOPSLA'87*, pages 331–341, 1987.

[Mic96] E. Micklei. Spying messages to objects. Tutorial presented at the 4th ESUG summer school, for code ask emicklei@elc.com, 1996.

[Mul95] P. Mulet. Réflexion et langage à prototypes, 1995. École des Mines de Nantes, Thèse de l'Université de Nantes.

[Pas86] G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *OOPSLA'86*, pages 341–346, 1986.

[Pel96] J. Pelrine. Meta-level programming in smalltalk. Tutorial presented at the 4th ESUG summer school, pelrine@object.ch, 1996.

[PWG93] F. Pachet, F. Wolinski, and S. Giroux. Spying as an Object-Oriented Programming Paradigm. In *TOOLS EUROPE'93*, pages 109–118, 1993.

[Riv96a] F. Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, Feb. 1996.

[Riv96b] F. Rivard. Smalltalk : a Reflective Language. In *REFLECTION'96*, pages 21–38, 1996.

[Riv97] F. Rivard. Evolution du comportement des objets dans les langages à classes réflexifs, 1997. Ecole des Mines de Nantes, Thèse de l'Université de Nantes.

[Sch96] B. Schaeffer. Smalltalk: Elegance and Efficiency. Ecoop Tutorial, 1996.

[Sie94] Siemens. Simatic Object 5 Offline. Technical report, Siemens, 1994.

[YT87] Y. Yokote and M. Tokoro. Experience and Evolution of Concurrent Smalltalk. In *OOPSLA'87*, pages 406–415, 1987.