Abstractions pour la Programmation Concurrente dans GARF[1]

Benoît Garbinato, Xavier Défago,
Rachid Guerraoui, Karim R. Mazouni
Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
Lausanne, Suisse
e-mail: garf@lse.epfl.ch

Résumé: GARF est un environnement à objets permettant la programmation modulaire d'applications distribuées résistantes aux pannes. Cet article présente la bibliothèque de classes offertes au programmeur par GARF, pour gérer l'expression de la concurrence et de son contrôle. La mise en oeuvre des mécanismes introduisant la concurrence, ainsi que plusieurs classes proposant divers types de contrôle de concurrence *intra-objet*, sont présentées. L'extension de certaines de ces classes au contrôle de concurrence *inter-objets* est également exposée.

Abstract: GARF is an object-oriented environment that allows the programming of fault-tolerant distributed applications in a modular way. This paper presents the library of classes provided to the programmer by GARF, in order to express and control concurrency. The implementation of the mechanisms introducing concurrency, as well as several classes offering various kinds of *intra-object* concurrency control, are presented. The extension of some of those classes towards *inter-objects* concurrency control is also exposed.

1 Introduction

La programmation d'applications distribuées est une tâche difficile, en raison des problèmes inhérents au fonctionnement potentiellement intermittent de telles applications et à leur distribution. Parmi les aspects à considérer lors de la programmation de telles applications, on trouve habituellement l'expression de concurrence et son contrôle. L'environnement de Génération Automatique d'applications Résistantes aux Fautes (GARF) [8, 15] simplifie la programmation d'objets distribués, en séparant leur comportement distribué de leurs fonctionnalités. Les *fonctionnalités* d'un objet distribué sont définies par ses opérations, dont la sémantique ne se préoccupe pas des problèmes pouvant survenir lors d'une utilisation en contexte distribué ou lors d'un fonctionnement intermittent. Le *comportement* d'un objet distribué est défini par la partie de son code gérant ce type de problèmes, notamment sa désignation distante, sa concurrence et le contrôle de celle-ci, sa persistance et sa résistance aux pannes. L'idée fondamentale de l'approche GARF est de séparer la programmation d'un objet distribué en deux sous-objets, responsables des aspects fonctionnels, respectivement des aspects comportementaux. Une telle approche a l'avantage d'être modulaire, puisque le comportement d'un objet distribué peut être défini sans toucher à la programmation de ses fonctionnalités. La réutilisabilité du code responsable des aspects comportementaux, souvent difficile à programmer, est ainsi grandement améliorée. Ce code est factorisé dans des sous-objets indépendants des fonctionnalités. Le concept de classe d'objets permet de définir des comportements réutilisables d'une application à l'autre. Grâce à l'héritage, il est possible de définir de nouveaux comportements à partir de comportements existants.

Cet article présente les classes proposées par l'environnement GARF pour exprimer et contrôler la concurrence *intra-objet* et *inter-objets* [11], ainsi que certains aspects de leurs mises en oeuvre. L'approche par bibliothèque de classes est plus flexible que celles proposées jusqu'ici, puisqu'aucun modèle de concurrence particulier n'est imposé. L'environnement GARF se contente d'offrir les mécanismes de base permettant de construire une librairie extensible de classes pour la programmation concurrente.

La partie 2 de cet article décrit le modèle d'exécution GARF, ainsi que quelques élements de méthodologie de programmation induite par ce modèle. Les classes permettant l'expression et la mise en oeuvre de la concurrence, au moyen d'objets *futurs*, sont présentées en partie 3. Dans la partie 4, plusieurs classes offrant diverses approches du contrôle de concurrence *intra-objet* sont exposées, ainsi que leurs mises en oeuvre. La partie 5 fait de même pour les classes permettant de gérer le contrôle de concurrence *inter-objets*. Cet article se conclut par la partie 6, qui rappelle les abstractions et les services offerts par l'environnement GARF et les comparent à ceux fournis par d'autres plates-formes à but similaire.

[1]Recherche financée par le Fonds National Suisse pour la Recherche Scientifique (projet Mesures Spéciales, no. contrat: 5003-034344.)

2 Le modèle d'exécution

2.1 Un bref aperçu

Le modèle d'exécution de GARF est basé sur deux catégories d'objets: les objets *données*, qui définissent les *fonctionnalités* des objets distribués, et les objets *comportementaux*, qui définissent leurs *comportements*, au sens des définitions données en introduction. Dans la suite de cet article les objets données sont parfois désignés simplement par le terme *objets*, si aucune ambiguité n'est possible.

La programmation d'une classe d'objets distribués se fait en deux étapes. Tout d'abord, une classe d'objets données est programmée, comme cela se ferait dans un langage à objets *séquentiel centralisé*, tel que Smalltalk ou C++. Cette étape terminée, le code de la classe n'est plus modifié. Dans une seconde étape, une classe d'objets comportementaux est choisie, voire raffinée par héritage, à partir d'une bibliothèque de classes prédéfinies, puis associée à la classe d'objets données précédemment programmée. Grâce à cette association, chaque objet données se voit attribuer un objet comportemental unique, au moment de sa création. Cet objet comportemental reçoit toutes les invocations envoyées ou émises par l'objet données, *à l'insu de ce dernier*. La figure 1 (a) illustre le schéma d'invocation client/serveur[2] tel qu'il est perçu par les objets données et le schéma tel qu'il est réellement exécuté à travers les objets comportementaux. Chaque flèche du schéma représente une *invocation* au sens classique des langages à objets, à savoir un couple (envoi, réponse).

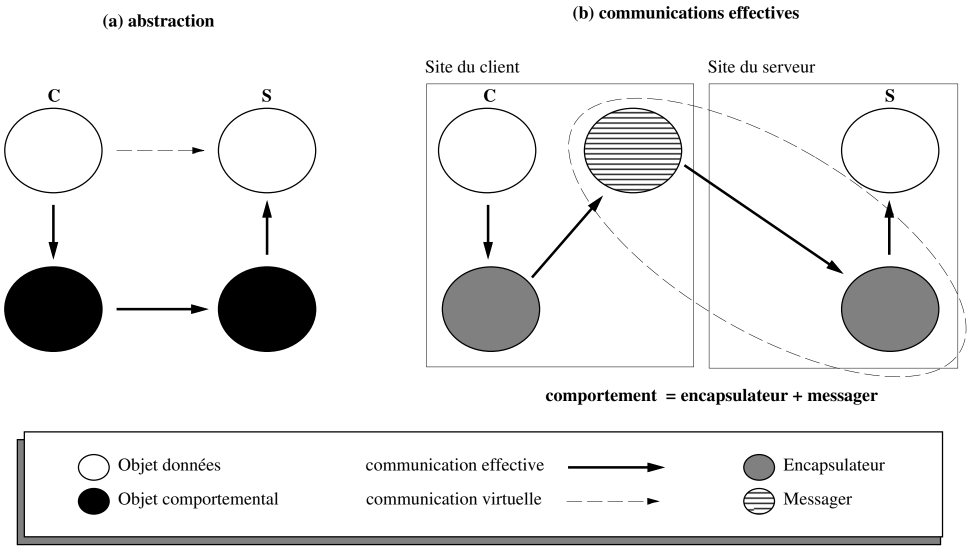

Figure 1: Schéma d'invocation dans GARF

L'association d'un objet comportemental à chaque objet données se fait au moment de la création de ce dernier. Lorsqu'une classe est invoquée en vue d'une création d'instance, l'invocation est transformée par le run-time GARF en un appel à la méthode `garfNew`. Cette méthode n'est connu qu'au niveau comportemental. En redéfinissant cette méthode pour une certaine classe d'objets données, le programmeur du niveau comportemental peut donc décider du comportement des instances de cette classe. Le code de la méthode `garfNew` doit créer un objet données et un objet comportemental, les associer, créer et renvoyer un nom GARF désignant l'objet comportemental. Au niveau comportemental, le travail du programmeur consiste donc à écrire les méthodes `garfNew` des classes d'objets données, ces classes ayant été définies au niveau fonctionnel. Une description plus détaillée de la méthodologie de programmation avec GARF peut être trouvée dans [15].

2.2 Les objets comportementaux

Les objets comportementaux sont constitués d'une partie se trouvant sur le site du client, le *messager*, et d'une partie se trouvant sur le site du serveur, l'*encapsulateur*. Chacune de ces parties est également un objet. Les encapsulateurs uniquement ont une désignation inter-sites, que le messager correspondant est seul à connaître. La localisation d'un objet distribué est donc transparente pour le client, tant au niveau fonctionnel qu'au niveau comportemental, et le messager fonctionne comme un *proxy intelligent* [4] de l'encapsulateur du serveur pour l'encapsulateur du client. La figure 1 (b) présente le schéma d'invocation à travers les parties messager et encapsulateur des objets comportementaux. Une copie du messager réside sur chaque site où se trouve au moins un *client potentiel* de l'objet distribué. Un objet est un client

[2]Les langages à objets mettent en oeuvre le modèle client/serveur de manière naturelle. Dans la suite de cet article, un objet invoqué sera parfois désigné par le terme *serveur* et l'objet invoquant par le terme *client*.

potentiel lorsqu'il possède le moyen d'invoquer l'objet distribué. Le messager est cloné à chaque invocation. Associer un comportement à un objet revient à créer un objet comportemental, constitué d'un messager et d'un encapsulateur, et à le lui associer. Ceci indique au système GARF que désormais toutes les invocations émises ou reçues par l'objet données devront être interceptées, réifiées et acheminées par la chaîne d'invocations présentée à la figure 1 (b).

La figure 2 détaille le schéma de communication à travers les objets comportementaux, en présentant les méthodes appelées à cet effet. L'invocation d'une opération `OP` sur un objet serveur s par un objet client c est automatiquement interceptée par le run-time GARF. Ce dernier invoque la méthode `outRequest` sur $En(c)$, l'encapsulateur de l'objet client c, en lui passant un messager $Ma(s)$ cloné au passage et dédié à cette invocation. L'encapsulateur $En(c)$ transmet l'invocation `OP` au messager $Ma(s)$ en invoquant `sendRequest`. Le messager fait suivre l'invocation à $En(s)$, l'encapsulateur du serveur, en appelant `inRequest`. Finalement, $En(s)$ invoque l'opération `OP` sur l'objet serveur s. Une fois l'exécution de la méthode `OP` terminée, son résultat revient à l'objet client c par le chemin inverse (réponse transmise en cascade, par retour de méthode). Les actions qu'effectuent les méthodes `outRequest`, `sendRequest` et `inRequest`, avant la transmission de l'invocation et après la réception de la réponse, détermine le comportement d'un objet distribué.

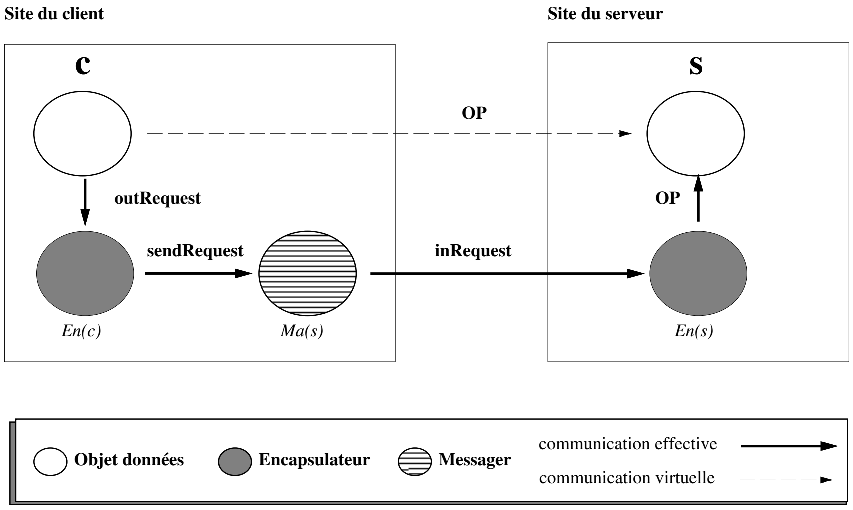

Figure 2: Invocation au moyen des encapsulateurs et du messager

2.3 La mise en oeuvre

La réalisation de GARF consiste en une extension de l'environnement de programmation Smalltalk[3] [10] au moyen de classes d'objets, dont certaines utilisent des services système, notamment pour la communication entre objets distants et pour leur désignation inter-sites. La distribution des encapsulateurs se base sur les services système de la plate-forme d'objets distribués *ROOT* (Reliable Object-Oriented Toolkit), réalisée au Laboratoire de Systèmes d'Exploitation (LSE). La plate-forme ROOT permet également la duplication d'objets en vue de les rendre résistants aux fautes [15, 9]. Un premier prototype de l'environnement de programmation GARF est actuellement opérationnel sur le réseau de stations Unix (Sun OS, architecture SPARC) du LSE.

Au niveau fonctionnel, la désignation d'un serveur s par un client c se fait à travers le *nom GARF* de s. Cet objet nom redirige, *à l'insu du client* c, les invocations de ce dernier destinées à s, sur l'encapsulateur de c. L'objet données c croit donc invoquer l'objet données s, alors qu'en réalité il invoque le nom GARF de s; c n'est conscient ni de la redirection de ses invocations, ni de la chaîne d'invocations qui en résulte au niveau comportemental. La figure 3 présente la redirection des invocations d'un objet données client c vers un objet données serveur s.

La partie *encapsulateur* d'un objet comportemental est une instance de la classe de base `BasicEncaps` ou de l'une de ses sous-classes. La classe `BasicEncaps` définit une variable d'instance `dataObject`, qui désigne l'objet données particulier auquel chaque encapsulateur est associé. En dehors du fait qu'elle permet la distribution, une instance de `BasicEncaps` ne fait que transmettre plus loin l'invocation interceptée, sans introduire de comportement supplémentaire. La figure 4 présente le code de `BasicEncaps` pour l'émission et la réception d'invocations (`outRequest:to:`, respectivement

[3]L'environnement utilisé est une version commerciale récente, proposée par la société ParcPlace.

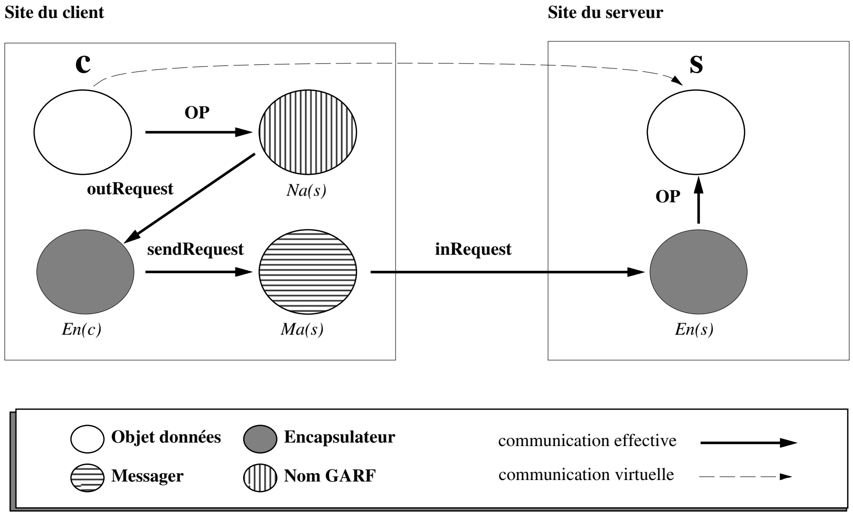

Figure 3: Désignation des objets données et redirection des invocations

`inRequest:`). A noter que le terme *message* provient de la terminologie Smalltalk et désigne un couple (envoi, réponse), autrement dit une invocation.

La partie *messager* d'un objet comportemental est une instance de la classe de base `BasicMailer` ou de l'une de ses sous-classes. La classe `BasicMailer` définit une variable d'instance `shell`, qui désigne un encapsulateur potentiellement distant. Ce dernier est associé à l'objet données que represente le messager sur le site du client. Là aussi, les messagers de cette classe ne font que permettre la distribution. La figure 5 présente le code de la méthode `sendRequest:` qui réalise ce comportement.

```
outRequest: aMessage to: aMailer
  ^aMailer sendRequest: aMessage

inRequest: aMessage
 ^dataObject perform:       aMessage selector
             withArguments: aMessage arguments
```

Figure 4: Méthodes de `BasicEncaps`

```
sendRequest: aMessage
  ^shell inRequest: aMessage
```

Figure 5: Méthode de `BasicMailer`

Les classes `BasicEncaps` et `BasicMailer` constituent le comportement par défaut de tout objet données, si aucun autre comportement n'a été spécifié. Telles qu'elles ont été réalisées, ces classes assurent que le comportement d'un objet distribué est le même que celui d'un objet Smalltalk standard. Elles se trouvent à la racine des hiérarchies de classes d'encapsulateurs, respectivement de messagers, que fournit l'environnement de développement GARF. Ces hiérarchies sont extensibles à souhait, grâce au mécanisme d'héritage.

3 L'expression de la concurrence

En raison de la nature même des langages à objets séquentiels, une invocation bloque le client jusqu'à ce que l'exécution de la méthode correspondante soit terminée. Avec le comportement par défaut, défini par les classes `BasicEncaps` et

BasicMailer, cette sémantique synchrone est conservée, même en distribué. Dans GARF, la concurrence est introduite en levant cette restriction. Afin d'autoriser les invocations asynchrones, deux nouvelles classes d'objets comportementaux sont introduites: AsyncEncaps et AsyncMailer. En associant des objets de ces classes aux objets données, les invocations de ces derniers deviennent asynchrones. La mise en oeuvre de l'asynchronisme est basée sur une classe d'objets, manipulés exclusivement par les encapsulateurs et les messagers: la classe FutureName.

3.1 Le concept d'objet futur

Le modèle de programmation à deux niveaux proposé par GARF requiert que tout nouveau flot d'exécution soit lancé depuis un objet comportemental. Ce dernier, après avoir lancé l'exécution concurrente d'une méthode, doit rendre le contrôle à l'appelant, qui s'attend à recevoir un objet comme réponse. Or la réponse peut très bien ne pas être immédiatement disponible. Pour contourner ce problème, l'environnement GARF fournit la notion d'*objet futur*. Ce concept, introduit pour la première fois par le langage *MultiLISP* [12], est notamment proposé par le langage *ABCL* [20].

Un *objet futur* est un objet particulier, destiné à contenir la réponse lorsque celle-ci sera disponible, quelque part dans le futur. Tant que l'appelant n'a pas besoin du contenu de la réponse, il peut manipuler l'objet futur comme tel (stockage par référence dans une structure de données ou passage en paramètre par exemple). Ce n'est que lorsque le contenu de la réponse est vraiment nécessaire qu'il risque d'y avoir attente. Lorsque le client fournit explicitement l'objet futur où il s'attend à trouver la réponse, on parle de *futur explicite*. Le client doit alors disposer d'un moyen de se mettre en attente de la réponse, lorsque le contenu de celle-ci est requis. Si au contraire le client reçoit et manipule un objet futur sans en être conscient, on parle de *futur implicite*. Dans ce dernier cas, le run-time du système doit se charger de bloquer tout client essayant d'accéder à un objet futur dont le contenu n'est pas encore disponible.

Les deux niveaux de programmation proposés par le modèle GARF apparaissent clairement en ce qui concerne les objets futurs. Au niveau fonctionnel, l'utilisation des objets futurs est implicite, alors que les objets comportementaux les créent et les manipulent explicitement.

3.2 La classe FutureName

L'environnement GARF offre la classe FutureName, dont les instances sont des objets futurs. Cette classe définit deux variables d'instance: namedObject et available. La variable namedObject désigne l'objet représenté par l'objet futur, qui peut être de n'importe quelle classe. La variable available désigne une instance de la classe Event et permet de déterminer si le contenu de l'objet futur est disponible. L'expression available wait bloque tout flot du contrôle qui l'exécute, jusqu'à ce l'expression available signal soit évaluée. La figure 6 montre les méthodes d'accès à la variable d'instance namedObject en lecture et en écriture. L'environnement GARF se charge d'éliminer régulièrement les objets futurs dont le contenu est devenu disponible: toutes les références à un objet futur devenu inutile sont transformées en références à son contenu (désigné par namedObject).

```
namedObject                     namedObject: anObject
  available wait.                 namedObject := anObject.
  ^namedObject                    available signal
```

Figure 6: Lecture et écriture de namedObject

Afin de rendre l'utilisation des objets futurs transparente au niveau fonctionnel, il est nécessaire qu'un objet données puisse s'adresser à un objet futur comme s'il s'agissait de la réponse attendue. Il faut donc intercepter les invocations adressées à l'objet futur pour les rediriger sur son namedObject, dès que celui-ci est disponible. Dans GARF, le mécanisme interceptant les invocations est basé sur l'exception messageNotUnderstood, levée par l'environnement Smalltalk lorsqu'un objet est invoqué par une opération qu'il ne définit pas. Lorsque cette exception est levée, la méthode doesNotUnderstand: est invoquée sur l'objet sollicité par erreur. En redéfinissant cette méthode pour la classe FutureName, il est possible de rediriger toutes les invocations destinées au contenu de l'objet futur vers la variable d'instance namedObject[4]. La figure 7 présente la méthode doesNotUnderstand: telle qu'elle est redéfinie pour la classe FutureName. Le run-time Smalltalk transforme l'invocation ayant provoquée la levée d'exception en l'objet aMessage (réification), qu'elle passe en paramètre à la méthode doesNotUnderstand:.

Pour que toute invocation destinée au contenu de l'objet futur soit interceptée, il faut assurer que ce dernier ne définisse pratiquement pas d'opérations. En particulier, la classe FutureName ne doit héritée d'aucune méthode de Object, racine de la hiérarchie des classes Smalltalk. La solution choisie consiste à faire de nil la superclasse de FutureName. Cette

[4]Ce mécanisme est également utilisé dans GARF pour rediriger les invocations des objets données vers les objets comportementaux (voir paragraphe 2.1) et pour intercepter les invocations des méthodes de création (voir paragraphe 2.3). Dans ces deux cas, la méthode doesNotUnderstand: est redéfinie de manière quelque peu différente, mais le principe reste le même.

```
doesNotUnderstand: aMessage
  ^(self namedObject) perform:       aMessage selector
                      withArguments: aMessage arguments
```

Figure 7: Interception d'invocations par **doesNotUnderstand:**

technique a entre autre été utilisée pour modifier la syntaxe de Smalltalk, afin d'introduire l'héritage multiple [13]. Elle a aussi servi à introduire des changements sémantiques [5, 7, 18]. Un certain nombre de difficultés pratiques doivent être résolues lorsqu'une classe a l'objet `nil` pour superclasse. Ces problèmes ont notamment à voir avec l'utilisation d'une telle classe par l'environnement de développement Smalltalk [16].

3.3 La classe `AsyncEncaps`

Un objet comportemental dont la partie encapsulateur est de la classe `AsyncEncaps` réalise une sémantique de client effectuant ses invocations de manière asynchrone. Chaque fois que l'invocation d'un client passe par son objet comportemental, à travers la méthode `outRequest:to:`, l'encapsulateur lance un processus léger Smalltalk, auquel il passe un nouvel objet futur. Ce dernier est immédiatement retourné à l'objet client. Le processus léger transmet l'invocation au messager du serveur et se bloque en attente de la réponse. Lorsque celle-ci arrive, il la place dans l'objet futur au moyen de la méthode `namedObject:` (voir figure 6) et se termine. La figure 8 présente le code de la méthode `outRequest:to:` redéfinie par la classe `AsyncEncaps`.

```
outRequest: aMessage to: aMailer
  | future |
  future := FutureName new.
  [ | result |
    result := aMailer sendRequest: aMessage.
    future namedObject: result] fork.
  ^future
```

Figure 8: Méthode de **AsyncEncaps**

3.4 La classe `AsyncMailer`

Un objet comportemental dont la partie messager est de la classe `AsyncMailer` réalise une sémantique de serveur non bloquant pour les clients qui l'invoquent. Lorsque la méthode `sendRequest:` du messager est invoquée par l'encapsulateur du client, un processus léger Smalltalk est lancé, auquel est passé un nouvel objet futur. Ce dernier est immédiatement retourné à l'encapsulateur du client. Le processus léger transmet l'invocation à l'encapsulateur du serveur et se bloque en attente de la réponse. Lorsque celle-ci arrive, il la place dans l'objet futur au moyen de la méthode `namedObject:` et se termine. La figure 9 présente le code de la méthode `sendRequest:` redéfinie par la classe `AsyncMailer`.

```
sendRequest: aMessage
  | future |
  future := FutureName new.
  [ | result |
    result := shell inRequest: aMessage.
    future namedObject: result] fork.
  ^future
```

Figure 9: Méthode de **AsyncMailer**

Réaliser un serveur non bloquant en se basant sur le renvoi d'un objet futur par la méthode `inRequest:` n'est pas possible. Une telle solution nécessiterait que l'objet futur puisse être mis à jour sur le site du *client*, dès que la réponse a été calculée sur le site du *serveur*, c'est-à-dire qu'un objet futur puisse être désigné d'un site à l'autre. Or seuls les encapsulateurs bénéficient d'une désignation inter-sites (voir paragraphe 2.2). Même en levant cette restriction, cette solution aurait le désavantage d'engendrer deux communications inter-sites: une pour le renvoi de l'objet futur et une pour la mise à jour de son contenu.

4 Le contrôle de concurrence intra-objet

L'exécution concurrente de plusieurs méthodes sur le même objet est rendue possible grâce aux classes `AsyncMailer` et `AsyncEncaps`. L'environnement GARF fournit plusieurs classes permettant au comportement d'un serveur de gérer le contrôle de ses invocations concurrentes. Les exemples de classes présentés ici sont volontairement assez simples, afin de focaliser l'attention sur l'originalité du modèle GARF et non sur des considérations algorithmiques de contrôle de concurrence bien connues. Les classes `MutexEncaps` et `ReaderWriterEncaps` permettent de créer des comportements gérant *localement* les accès concurrents à un objet. La figure 10 présente une partie de la hiérarchie des encapsulateurs offrant des mécanismes de contrôle de concurrence *intra-objet*. Le fait qu'aucune classe de messagers n'ait été définie pour cette tâche met en évidence le rôle de l'encapsulateur par rapport à celui du messager. L'encapsulateur permet d'assurer la cohérence de l'objet données, alors que le messager permet d'assurer la transparence de sa localisation[5] par rapport à l'objet client.

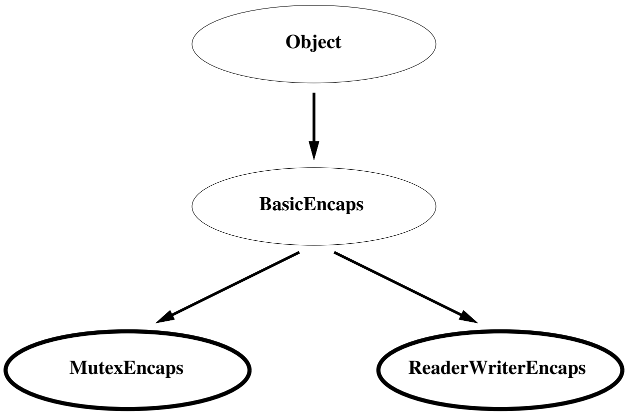

Figure 10: Classes pour le contrôle de concurrence intra-objet

4.1 La classe `MutexEncaps`

Une façon simple de contrôler les accès concurrents à un objet consiste à garantir que toutes ses opérations sont exécutées en exclusion mutuelle. Les encapsulateurs de la classe `MutexEncaps` permettent de transformer n'importe quel objet données en un objet ayant ce comportement. La classe `MutexEncaps` définit une variable d'instance `mutex` qui désigne un sémaphore d'exclusion mutuelle. La figure 11 présente le code de la méthode `inRequest:` redéfinie par `MutexEncaps`. Comme toutes les invocations arrivant à un objet encapsulé par un `MutexEncaps` passent par cette méthode et que celle-ci assure l'exclusion mutuelle, on a bien la sémantique requise.

```
inRequest: aMessager
  | result |
  mutex critical: [result := super inRequest: aMessage].
  ^result
```

Figure 11: Méthode de `MutexEncaps`

4.2 La classe `ReaderWriterEncaps`

La gestion *des lecteurs et des rédacteurs* est un problème classique de contrôle de concurrence [6]. Dans le contexte des objets, il peut s'énoncer de la façon suivante: les opérations ne modifiant pas l'état d'un objet (lectures) peuvent se dérouler concurremment, alors que toute opération altérant son état (écriture) doit s'exécuter en exclusion mutuelle.

La classe `ReaderWriterEncaps` permet d'associer à un objet données un contrôle de concurrence de type lecteurs/rédacteurs. Au moment de son association avec un objet données, un tel encapsulateur reçoit un ensemble d'identificateurs d'opérations, appelés *sélecteurs de message* dans la terminologie Smalltalk. Cet ensemble de sélecteurs représente les opérations d'écriture fournies par l'objet encapsulé. Lorsque une instance de `ReaderWriterEncaps` reçoit une invocation à l'intention de son objet données, cet ensemble, désigné par la variable d'instance `writerSelectorSet`, lui permet de

[5] Le messager peut être vu comme une généralisation de la notion de *proxy*, puisqu'il peut avoir un comportement différent d'un objet à l'autre, voire d'une invocation à l'autre.

distinguer l'arrivée d'un lecteur de celle d'un rédacteur. En fonction de cette information, l'encapsulateur choisit le traitement à effectuer. La figure 12 présente le code de la méthode `inRequest:` redéfinie par `ReaderWriterEncaps`. Cette méthode utilise les opérations `readerAction:` et `writerAction:`, définies par la classe `ReaderWriterEncaps` afin de gérer l'arrivée d'un nouveau lecteur, respectivement d'un nouveau rédacteur. En redéfinissant ces deux méthodes dans des sous-classes de `ReaderWriterEncaps`, il est possible de mettre en oeuvre différentes politiques de priorité, telles qu'elles peuvent être définies pour ce type de problèmes [19].

```
inRequest: aMessage
  | result |
  (writerSelectorSet includes: aMessage selector)
    ifTrue:  [self writerAction: [result := super inRequest: aMessage]]
    ifFalse: [self readerAction: [result := super inRequest: aMessage]].
  ^result
```

Figure 12: Méthode de `ReaderWriterEncaps`

5 Le contrôle de concurrence inter-objets

Lorsque la sémantique d'un objet client exige qu'une opération impliquant plusieurs objets serveurs soit indivisible, on parle de contrôle de concurrence *inter-objets* [11]. Dans ce cas, l'exclusion mutuelle assurée *localement* par le serveur ne suffit plus et il est nécessaire d'introduire d'autres mécanismes. Le concept de *transaction* permet de rendre une opération indivisible, même si celle-ci accède à plusieurs objets. On dit alors que l'opération est *atomique*[6] [3]. Dans l'environnement GARF, la mise en oeuvre des transactions passe par les classes `ExtendedMutexEncaps`, `AtomicMailer` et `AtomicEncaps`. Le programmeur ne manipule pas explicitement de transactions; celles-ci sont entièrement gérées par le code de ces trois classes. La figure 13 présente la partie de la hiérarchie de classes offrant des mécanismes de contrôle de concurrence inter-objets.

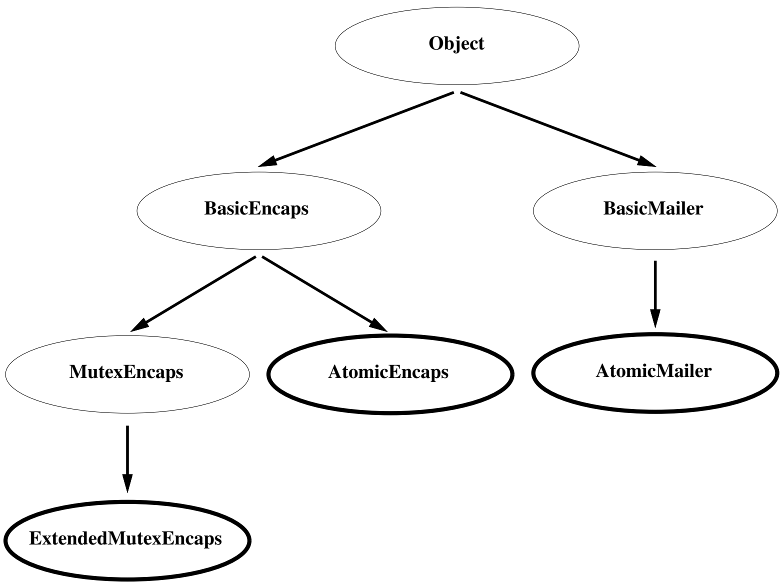

Figure 13: Classes pour le contrôle de concurrence inter-objets

5.1 La classe `ExtendedMutexEncaps`

Un objet encapsulé par une instance de la classe `ExtendedMutexEncaps` voit l'exclusion mutuelle assurée sur ses opérations s'étendre par transitivité aux objets invoqués lors de celles-ci. Pour chaque invocation, l'exclusion mutuelle est donc garantie non seulement sur l'objet lui-même, mais également sur tous les objets accédés pendant l'exécution de la méthode invoquée; celle-ci s'exécute de manière indivisible. Assurer que toutes les opérations d'un objet sont indivisibles

[6] On ne considère ici que la propriété d'atomicité à la concurrence; la propriété d'atomicité aux pannes n'est pas considérée dans cet article.

revient à créer une transaction à chaque invocation. C'est la méthode `inRequest:` de la classe `ExtendedMutexEncaps`, présentée à la figure 14, qui initie et termine la transaction; cette classe définit deux méthodes: `beginTransaction` et `endTransaction`, basées sur un gestionnaire de transactions.

```
inRequest: aMessage
  | result |
  self beginTransaction.
  result := super inRequest: aMessage.
  self endTransaction.
  ^result
```

Figure 14: Méthode de `ExtendedMutexEncaps`

Les instances de la classe `ExtendedMutexEncaps` englobent systématiquement chaque opération dans une transaction, sans tenir compte de sa sémantique; cette approche est parfois trop rigide. Un raffinement possible de cette classe pourrait tenir compte de la sémantique de chaque opération, afin de décider si une transaction est nécessaire ou non. Comme pour la classe `ReaderWriterEncaps` (voir paragraphe 4.2), cela pourrait être réalisé en passant aux instances de la classe `ExtendedMutexEncaps` l'ensemble d'identificateurs des opérations nécessitant une transaction.

5.2 Les classes `AtomicEncaps` et `AtomicMailer`

Pour garantir l'indivisibilité de l'opération qui l'invoque, un objet doit avoir son comportement défini par les classes `AtomicMailer` et `AtomicEncaps`. On parle alors d'objets *atomiques*. Lorsqu'une transaction soumet une invocation à un objet atomique, son messager s'enregistre auprès de la transaction. Avant de se terminer, celle-ci pourra ainsi déverrouiller les objets auxquels elle a accédé. Lorsqu'un objet est invoqué plusieurs fois au cours d'une transaction, seul le premier de ses messagers est enregistré. Cette précaution est nécessaire afin d'éviter qu'un même objet ne soit déverrouillé plusieurs fois par la même transaction. Le messager transmet ensuite l'invocation à l'encapsulateur correspondant. La figure 15 présente la méthode `sendRequest:`, redéfinie par `AtomicMailer`, qui réalise ce comportement.

```
sendRequest: aMessage
  self addToTransaction.
  ^super sendRequest: aMessage.
```

Figure 15: Méthode de `AtomicMailer`

L'encapsulateur invoqué par le messager détermine si son objet données est déjà verrouillé par une autre transaction. Si tel est le cas, la transaction dans laquelle s'exécute l'invocation est mise en attente; sinon l'objet est verrouillé à son usage. Une fois en section critique, l'encapsulateur peut transmettre l'invocation à l'objet qu'il encapsule. La méthode `inRequest:`, redéfinie par `AtomicEncaps` et présentée à la figure 16, met en oeuvre ce comportement.

```
inRequest: aMessage
  (self alreadyInSameTransaction)
    ifFalse: [lock wait.
              self registerTransaction].
  ^super inRequest: aMessage
```

Figure 16: Méthode de `AtomicEncaps`

6 Conclusion

L'intégration des concepts d'objet et de concurrence a suscité l'intérêt de nombreuses communautés de chercheurs. Divers modèles de programmation concurrente par objets sont apparus dans la littérature [1, 22]. Ces modèles diffèrent sur la manière d'exprimer la concurrence, la granularité de la concurrence autorisée, le contrôle intra-objet et le contrôle inter-objets. Aucun de ces modèles n'est optimal et chacun est adapté à un type d'applications particulier [17]. Le contrôle intra-objet explicite proposé dans POOL [2] et dans ABCL [21] est plus performant que celui implicite proposé dans ACT++ [14], mais interfère avec l'héritage. De même, le modèle d'objets concurrents de ACT++ est plus performant que le modèle d'objets séquentiels de POOL. Cependant, le premier nécessite un contrôle des accès concurrents, alors que le second permet de s'en passer. En fonction du type d'applications considéré (un ou plusieurs niveaux d'héritage, granularité des objets), le type de contrôle disponible et la granularité de la concurrence peuvent être plus ou moins adaptés.

L'environnement GARF ne propose pas de modèle de concurrence particulier. Il offre les mécanismes de base permettant de construire une bibliothèque extensible de classes pour la programmation concurrente. Cette approche favorise la modularité et la flexibilité. Dans cet article nous avons décrit la méthodologie de programmation avec GARF et présenté des exemples de classes. Ces classes représentent autant d'abstractions de modèles de concurrence. L'utilisateur peut choisir, pour une application donnée, les classes adéquates et associer leurs objets comportementaux aux objets données, sans toucher à la programmation des fonctionnalités de ces derniers. Il peut aussi étendre la bibliothèque, en définissant de nouvelles classes par héritage.

References

[1] G. Agha, P. Wegner, and A. Yonezawa. Object-based concurrent programming. *SIGPLAN Notices*, 24, 1989.

[2] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[4] S. Bijnens, W. Joosen, and P. Verbaeten. A reflective invocation scheme to realise advanced object management. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object Based Distributed Programming*. Springer Verlag, 1994.

[5] J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *European Conference on Object Oriented Programming Proceedings*, pages 109–129. Cambridge University Press, 1986.

[6] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14:667–668, 1971.

[7] B. Foote and R.E. Johnson. Reflective facilities in Smalltalk-80. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 327–335. ACM Press, 1989.

[8] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Distributed programming in GARF. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object Based Distributed Programming*. Springer Verlag, 1994.

[9] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Programming fault-tolerant applications using two orthogonal object levels. In *Proceedings of the 8th International Symposium on Computer and Information Sciences*, 1994.

[10] A.J. Goldberg and A.D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison Wesley, 1983.

[11] R. Guerraoui. *Programmation répartie par objets: études et propositions*. PhD thesis, Université d'Orsay, 1992.

[12] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Applications*, 1985.

[13] D.H.H. Ingalls and A.H. Borning. Multiple inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237. AAAI, 1982.

[14] D. Kafura and K.H. Lee. Inheritance in actor based concurrent object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*. Cambridge University Press, 1989.

[15] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Programmation d'une application distribuée résistante aux pannes avec l'environnement GARF. In *Proceedings of the 3rd Maghrebian Conference on Software Engineering and Artificial Intelligence*. Maghrebian Information Processing Society, 1994.

[16] P. L. McCullough. Transparent forwarding: First steps. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 331–341. ACM Press, 1987.

[17] M. Papathomas. *Language Design Rationale and Semantic Framework for concurrent Object-Oriented Programming*. PhD thesis, Université de Genève, 1992.

[18] G.A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings of the Conference on Systems, Languages and Applications*, pages 341–346. ACM Press, 1986.

[19] A. Schiper. *Programmation concurrente*. Collection Informatique. Presses Polytechniques Romandes, second edition, 1990.

[20] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 258–268. ACM Press, 1986.

[21] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[22] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming*. MIT Press, 1987.