Smalltalk : du mono-processeur et mono-utilisateur à la programmation concurrente et répartie

Jean-Pierre Briot * — **Rachid Guerraoui** **

** Laboratoire d'Informatique de Paris 6*
Paris 6 - Case 169, 4 place Jussieu, 75252 Paris Cedex 05, France

Jean-Pierre.Briot@lip6.fr

*** Département d'Informatique*
École Polytechnique Fédérale de Lausanne, CH-1015, Lausanne, Suisse

guerraoui@di.epfl.ch

RÉSUMÉ. *Cet article défend la thèse paradoxale suivante : « Bien qu'ayant été dès l'origine fondamentalement conçu comme un environnement de programmation mono-utilisateur et mono-processeur, Smalltalk est une excellente fondation pour prototyper des environnements de programmation concurrente et répartie ». Les deux principales raisons qui soutiennent notre affirmation sont les suivantes : la richesse et la réutilisabilité des bibliothèques de classes et l'extrême flexibilité du langage. Elle complètent les capacités des langages à objets en général.*

ABSTRACT. *This paper argues that Smalltalk, although designed as a mono-user and mono-processor programming language, is a very good foundation for prototyping concurrent and distributed programming environments. The two main reasons behind this likely paradox are: the rich and reusable library of classes, and the extreme flexibility of Smalltalk. These two specificities complement more general object-oriented characteristics and advantages.*

MOTS-CLÉS : *Concurrence, répartition, bibliothèques, flexibilité, réflexion.*
KEY WORDS : *Concurrency, distribution, libraries, flexibility, reflection.*

1. Introduction

Cet article défend la thèse suivante : « Smalltalk est une excellente fondation pour le prototypage d'environnements de programmation concurrente et répartie ». Si l'on connaît un tant soit peu l'histoire de Smalltalk, cette déclaration peut apparaître presque comme un paradoxe. En effet, Smalltalk [1] a été conçu dès le départ comme un environnement de programmation mono-utilisateur et mono-processeur. Il l'est d'ailleurs toujours resté, en tant que langage. Toutefois, la richesse de ses bibliothèques standard et étendues ainsi que son extrême flexibilité permettent comme nous le verrons de compenser cette limitation du langage. Nous pouvons résumer ces raisons dans les quatre points suivants :

– Smalltalk, en tant que langage à objets (« object-based » [2]) offre *encapsulation* et *structuration* pour bâtir des services et mécanismes variés et modulaires. En même temps, la *généricité* du concept d'objet permet de modéliser différents types d'architectures logicielles et matérielles, indépendamment de la topologie, du fait de la métaphore de transmission de messages [BG 96].

– Smalltalk, en tant que langage à classes et à héritage, offre des techniques de *classification* et de *spécialisation* permettant d'organiser et de particulariser différents mécanismes et protocoles de gestion de la concurrence, de la répartition et de la tolérance aux fautes.

– Smalltalk, en tant qu'*environnement* de programmation et de développement, offre des *bibliothèques standard* d'abstractions de base de gestion de la concurrence, de représentation de structures d'objets échangeables et de communication distante.

– Smalltalk, en tant que langage *réflexif* (auto-décrit et adaptable), facilite la *spécialisation* et l'*intégration* de mécanismes et protocoles de gestion de la concurrence et de la répartition.

De manière à mieux illustrer notre thèse, et après avoir détaillé ces caractéristiques de Smalltalk, nous décrirons brièvement la construction de deux plateformes, toutes deux développées en Smalltalk. La première plateforme, Actalk [Bri 94], a été destinée à la programmation concurrente et la seconde, GARF [GGM 94], à la programmation répartie.

2. Objets pour la programmation concurrente et répartie

Il est maintenant bien établi que la notion d'*objet* offre une bonne fondation conceptuelle pour aborder les nouveaux enjeux de la programmation parallèle et répartie [BG 96]. D'ailleurs, les principales initiatives en matière d'informatique

1. Notre analyse s'applique de manière générale à l'ensemble des dialectes Smalltalk existants. Cependant, dans la suite de l'article les détails (classes, noms de méthode, etc.) feront toujours référence à ParcPlace VisualWorks 2.5.

2. Peter Wegner a défini dans [Weg 87] comme « *object-based* » un langage de programmation garantissant l'encapsulation et comme « *object-oriented* » un langage incluant les concepts de classe et d'héritage.

répartie, Open Distributed Processing (ODP) et Object Management Group (OMG), sont fondées sur le concept d'objet [NWM 93].

Remarquons tout d'abord que la concurrence était déjà présente dès l'origine du concept d'objet. Dans le premier langage de programmation à objets, Simula-67 (Smalltalk-72 étant le second), le concept d'objet était intimement lié avec la notion de coroutine. De plus, on constate qu'objets et processus partagent de nombreuses caractéristiques communes : variables, données persistantes, encapsulation, et moyens de communication [Mey 93].

Un objet apparaît également de manière naturelle comme une unité potentielle de répartition. La transmission de messages assure, en effet, non seulement l'indépendance entre les services offerts par un objet et sa représentation interne, mais également l'indépendance vis-à-vis de son *emplacement* sur tel ou tel site (processeur, mémoire, ou machine).

3. Smalltalk

Smalltalk est généralement considéré comme l'un des plus « purs » exemples de langage à objets. Son « motto » est d'être fondé sur le minimum de concepts (objet, transmission de messages, classe, héritage) et de les appliquer de manière *uniforme* à l'ensemble du langage et de son environnement. Une conséquence en est l'excessive *simplicité* du langage. La richesse de Smalltalk vient de la richesse et de la variété de ses bibliothèques de classes, comme nous allons le voir.

3.1. *Bibliothèques standard*

Smalltalk possède en standard une grande variété de bibliothèques de classes. Elles représentent différents types de structures (structures de contrôle, structures de données...), ressources (messages, processus, compilateur...), et un environnement de programmation sophistiqué à outils intégrés (flâneur, inspecteur, dévermineur...).

Notez que même les structures de contrôle, telles que les conditionnelles et les boucles ne sont pas des primitives du langage mais des méthodes comme les autres, utilisant la généricité de la transmission de messages. Les structures de contrôle ont pour paramètres des *blocs*. Un bloc est une séquence quelconque d'expressions entourée de crochets (`[` et `]`) dont l'évaluation est retardée. Les blocs (qui sont des instances de la classe `BlockClosure`) sont à la base des structures de contrôle que l'utilisateur peut ainsi étendre à volonté. Les blocs sont également la base du multi-tâche. L'envoi du message `fork` à un bloc crée et démarre un nouveau processus.[3] La classe `Process` décrit leur représentation et les méthodes associées pour leur gestion (suspension, reprise, changement de

3. Smalltalk utilise le terme *processus* même si la terminologie actuelle parlerait maintenant plutôt de *processus de poids léger* ou encore de *tâche*.

priorité...). Le séquenceur des processus est lui-même décrit par une classe, nommée `ProcessorScheduler`.

La primitive de synchronization de base est en Smalltalk le *sémaphore*, représenté par la classe `Semaphore`. Les bibliothèques standard de classes incluent également des abstractions de plus haut niveau, en particulier la classe `SharedQueue` comme moyen de communication entre processus et la classe `Promise` pour l'évaluation anticipée (une valeur calculée simultanément par un processus). Ces deux abstractions servent à implanter des types de communication asynchrone et à réponse anticipée (« future ») pour la programmation concurrente par objets. De manière plus générale, il est relativement aisé de construire des abstractions de programmation concurrente de plus en plus sophistiquées sur la base de la bibliothèque standard de Smalltalk (par exemple dans la plateforme Actalk qui sera décrite au § 4.1).

Smalltalk offre également, en grande partie en standard, des bibliothèques de communication distante (sockets Unix, remote procedure call (RPC)...) ainsi que des bibliothèques de stockage et d'échange de structures d'objects (la bibliothèque *Binary Object Streaming Service : BOSS*), briques de base permettant la mise en œuvre de la persistance, de points de sauvegarde pour des transactions et d'encodage/décodage pour les communications distantes (par exemple dans la plateforme GARF qui sera décrite au § 4.2).

3.2. *Flexibilité et intégration*

Le fait que la quasi-totalité des constructions du langage Smalltalk (classes, méthodes, messages, structures de contrôle, contextes, processus...) et des mécanismes (compilateur, séquenceur...) soient des objets « de première classe », représentés par des librairies de classes, rend le langage auto-décrit et extensible. Une classe, en tant qu'objet « de première classe », est elle-même instance d'une classe appelée *métaclasse* [BC 89]. Une méta-description encore plus fine pourra également être individualisée au niveau de l'objet lui-même, on parlera alors de *méta-objet* [Mae 87]. Ces choix correspondent à différents degrés du concept général de système informatique *réflexif*, c'est-à-dire qui peut décrire et adapter sa propre structure et son propre comportement.

En plus de son auto-description structurelle, Smalltalk offre un certain nombre de points d'entrée (« hooks ») de manière à « ouvrir » la sémantique et la mise en œuvre de son propre comportement de base (transmission de messages, référence d'un objet...). La combinaison de l'auto-description structurelle de Smalltalk avec ces points d'entrée (pour la plupart, des méthodes ainsi appelées *méthodes réflexives*) est une raison fondamentale de la flexibilité du langage et de sa capacité à intégrer de manière transparente des extensions.

Nous allons maintenant décrire rapidement certains de ces points d'entrée (facilités réflexives) de Smalltalk en précisant particulièrement ici leur utilisation pour exprimer et intégrer des principes de programmation concurrente et distribuée dans Smalltalk.

– ***changement de la sémantique de la transmission de messages*** – Si

un message n'est pas reconnu par un objet receveur, le message `doesNotUnderstand:` est envoyé au même receveur avec le message non reconnu comme argument. La sémantique par défaut est d'ouvrir un notifieur d'erreur (qui, à son tour, peut ouvrir un dévermineur), mais toute classe peut spécialiser de manière locale le traitement d'erreur par redéfinition de la méthode `doesNotUnderstand:`. Une telle technique est utilisée pour adapter localement la sémantique de la transmission de message (transmission asynchrone, multicast, transaction, etc., voir au § 4).
– ***échange de références d'objets*** – La méthode `become:` échange les références entre deux objets. Cette primitive est utile pour préserver les références sur un objet en cas de reconstruction ou d'insertion d'une enveloppe d'interception d'envois de messages à l'aide de la technique ci-dessus. [4]
– ***contexte courant*** – La pseudo-variable [5] `thisContext` référence le contexte d'exécution courant. [6] Cette variable offre un point d'entrée sur la pile d'exécution. Elle est, par exemple, utile pour spécialiser le dévermineur pour la transmission de message asynchrone de la programmation concurrente.
– ***changement de classe d'un objet*** – La méthode `changeClassToThatOf:` change la classe d'un objet et par voie de conséquence son comportement. C'est une autre primitive également très utile pour assurer l'évolution et l'intégration de protocoles (voir au § 4.1.4). Mais elle impose une certaine compatibilité entre les deux classes.

4. Exemples

Différentes bibliothèques, extensions, plateformes et frameworks pour la programmation concurrente, parallèle et répartie ont été développés en Smalltalk. Ils mettent à profit la flexibilité et la richesse de Smalltalk qui facilitent grandement la réutilisation des programmes standard avec des bibliothèques et mécanismes de gestion de la concurrence et de la répartition. Nous allons décrire brièvement ci-dessous deux exemples de tels frameworks, développés en milieu académique : Actalk, un framework de programmation concurrente à objets et GARF, un framework de programmation répartie et tolérante aux fautes.

Il existe également un certain nombre de produits commerciaux tels que : ParcPlace Distributed Smalltalk, environnement de développement d'applications réparties suivant la norme CORBA ; IBM VisualAge add-on features for

---

4. Voir en particulier l'article de Stéphane Ducasse dans ce même numéro, qui détaille différentes techniques de ce type.

5. On utilise en général le terme « pseudo-variable », comme il s'agit d'une variable mais qui n'est pas déclarée de manière explicite dans une classe. Elle correspond à une variable de la machine virtuelle Smalltalk, rendue visible au programmeur, mais uniquement en lecture. `self` et `super` sont deux autres exemples de pseudo-variables.

6. Un contexte est un objet Smalltalk de première classe.

distribution, l'équivalent IBM mais cette fois non compatible CORBA ; et OTI Envy/Replicator, environnement de développement collectif réparti.[7]

4.1. *Actalk*

Actalk[8] est un framework de programmation concurrente par objets développé en, et intégré à, Smalltalk. Actalk offre une fondation pour modéliser et classifier différents types de constructions linguistiques et mécanismes de programmation concurrente [Bri 94]. Il a également été utilisé comme plateforme de base par de nombreux projets, tels la simulation de modèles de processus logiciels et la construction de différentes plateformes multi-agent.[9]

4.1.1. Architecture

L'architecture d'Actalk inclut un noyau qui modélise la sémantique de base de la programmation concurrente par objets à travers la notion d'*objet actif sérialisé*. Un *objet actif* est un objet doté d'une activité (processus ou tâche) propre. Un objet actif *sérialisé* traite ses messages en séquence. Ces objets communiquent par transmission de messages asynchrone et unidirectionnelle [Gue 95, BG 96]. Le noyau de l'architecture est composé d'un ensemble de classes composants, chaque *composant* décrivant un aspect différent d'un objet actif. Les trois composants principaux sont :

– ***comportement*** : qui définit le comportement applicatif de l'objet actif. La classe `ActiveObject` est la classe de base des comportements.
– ***activité/synchronisation*** : qui définit la manière dont les invocations de méthodes sont sélectionnées et synchronisées. La classe `Activity` est la classe d'activité de base.
– ***communication*** : qui définit la manière dont les transmissions de messages sont interprétées, par exemple : avec ou sans réponse, asynchrone ou synchrone, etc. La classe `Address` est la classe de communication (également appelée *adresse*) de base.

Chaque classe composant est elle-même à nouveau décomposée en un certain nombre de *méthodes paramètres*, conçues dans le but d'être redéfinies et ainsi spécialisées dans les sous-classes pour pouvoir modéliser différentes alternatives de conception de langages.

La relative indépendance entre les composants permet à l'utilisateur de modéliser ou/et d'associer relativement indépendamment différents aspects (modèle d'activité, de synchronisation, de communication) pour un objet/programme donné. Le détail de l'architecture d'Actalk, et en particulier la

7. C'est en fait une version répartie de l'environnement de travail collectif Envy/Developer.
8. *Actalk* signifie « *Act*ive objects, ou *act*eurs, en Small*talk*-80 ».
9. Voir notamment la plateforme DIMA, décrite dans l'article de Zahia Guessoum dans ce même numéro.

description de l'ensemble des méthodes paramètres pour chaque classe composant, est expliqué dans [Bri 94].

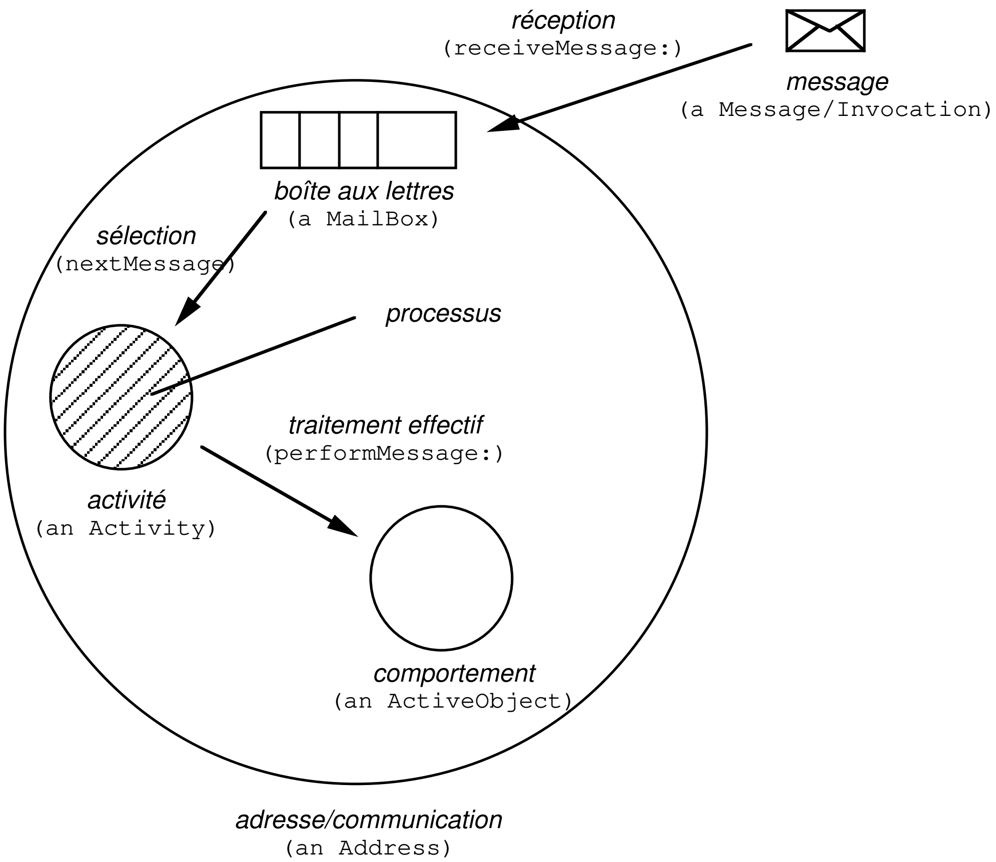

Fig. 1 - . *Composants d'un objet actif Actalk.*

4.1.2. Fonctionnement

En pratique, le programmeur crée un comportement comme instance d'une classe de comportements, sous-classe de la classe `ActiveObject`, puis lui envoie le message `active` pour l'activer. Ceci entraîne la création transparente des deux autres composants associés, activité et adresse, et le démarrage du processus associé à l'activité.

La fonctionnalité minimale d'un objet actif peut être résumée en suivant les étapes principales du cheminement d'une requête (voir la Figure 1), depuis sa réception jusqu'à son traitement :

– l'adresse *reçoit* un message, ce qui déclenche la méthode paramètre `receiveMessage:` de `Address` qui rajoute le message dans la queue de messages en attente ;
– *indépendamment*, l'activité *sélectionne* le message suivant (méthode paramètre `nextMessage` de `Activity`),
– qu'elle accepte (méthode paramètre `performMessage:` de `Activity`) en déléguant son *traitement effectif* au comportement.

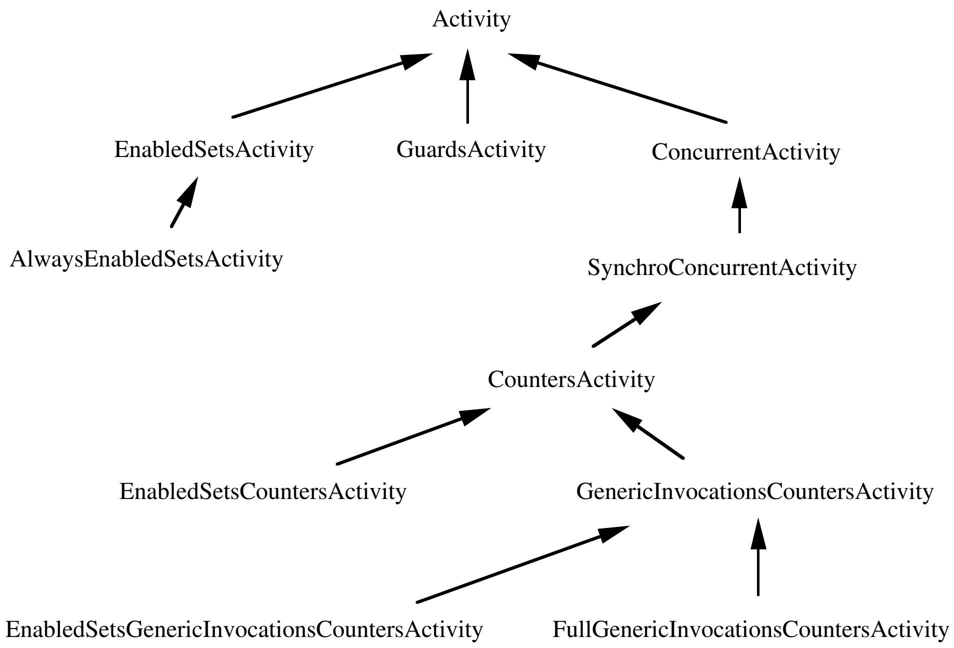

Fig. 2 - . *Hiérarchie des classes d'activité/synchronisation.*

4.1.3. Hiérarchie de composants

Un grand nombre d'extensions (sous-classes) des différentes classes composants a été développé pour implanter différents *modèles et constructions de langages*, *modèles de communication*, et *modèles de synchronisation.*

La Figure 2 illustre une partie de la hiérarchie des classes d'activité et de synchronisation. Par exemple, la classe `EnabledSetsActivity` implémente un modèle de synchronisation basé sur les ensembles de méthodes activables, la classe `GuardsActivity` implémente les gardes, c'est-à-dire des prédicats d'activation, et la classe `CountersActivity` implémente le modèle des compteurs de synchronisation. Le détail de cette hiérarchie qui représente spécialisation et combinaison progressives de classes de synchronisation est présenté dans [Bri 96].

4.1.4. Mise en œuvre

La mise en œuvre du noyau d'Actalk repose tout d'abord sur la redéfinition du message `doesNotUnderstand:` pour implanter de manière transparente la transmission de messages asynchrone. Le stockage des messages en attente est exprimé à l'aide de la classe standard `SharedQueue`. Ce type de transmission de base de la programmation concurrente peut être ensuite étendu, par exemple pour modéliser la transmission avec réponse anticipée, grâce à la classe standard `Promise`. On peut aussi définir des sous-classes de la classe standard `Message` de manière à inclure des informations supplémentaires telles que l'émetteur du message ou son temps d'arrivée, nécessaires pour des algorithmes de synchronisation à priorités [Bri 96].

Signalons également que dans le cadre du projet Actalk, un séquenceur générique a été développé et intégré au séquenceur standard de Smalltalk-80 [Les 92]. De manière à assurer une intégration optimale de ce nouveau séquenceur avec la machine virtuelle de Smalltalk-80 (et pour être plus précis, avec la gestion de la suspension des processus sur les sémaphores), sa mise en œuvre peut changer de manière dynamique et temporaire le comportement de certains sémaphores le moment venu (grâce à la méthode réflexive `changeClassToThatOf:`).

4.2. *GARF*

GARF [10] est un framework pour la conception et la réalisation d'applications réparties et robustes sur un réseau de stations de travail [GGM 94, GGM 95]. La spécificité de GARF réside dans son modèle de programmation incrémental et dans sa bibliothèque extensible de composants génériques. GARF a été mis en œuvre en Smalltalk au dessus du framework Isis de programmation tolérante aux fautes [BvR 93]. Puis il a été porté au dessus de Phœnix [Mal 96].[11] GARF prône une approche modulaire en séparant clairement les caractéristiques comportementales concernant la concurrence, la distribution et la robustesse des caractéristiques fonctionnelles concernant la programmation séquentielle et centralisée traditionnelle. Tout d'abord, le programmeur GARF conçoit et implante les différents composants de son application dans un environnement centralisé, en se concentrant seulement sur leur fonctionnalité. Ensuite, sans modifier le code précédemment écrit, le programmeur

10. *GARF* signifie « *G*énération *A*utomatique d'*A*pplications *R*ésistantes aux *F*autes ».
11. Phœnix est un outil de type Isis, conçu à l'EPFL, avec pour objectif une répartition à grande échelle.

peut se concentrer sur les caractéristiques comportementales en exprimant et contrôlant la concurrence, en répartissant l'application à travers un réseau et en dupliquant ses composants critiques pour augmenter la robustesse. Toutes ces différentes étapes, en incluant la toute première, peuvent être effectuées simultanément mais le code écrit pour les différentes étapes est clairement séparé.

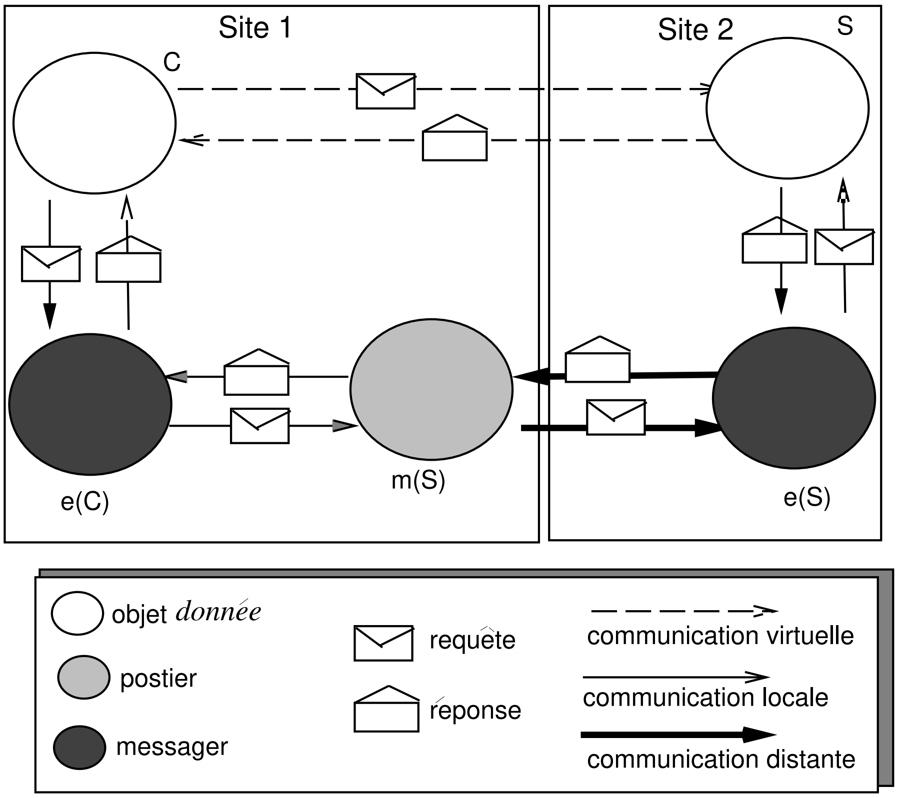

Fig. 3 - . *Schéma de communication dans GARF.*

GARF distingue deux types d'objets : objets *données* (« data objects ») et les objets *comportementaux* (« behavioral objects »). Les objets *données* servent à décrire les aspects fonctionnels de l'application (c'est-à-dire les aspects séquentiel et centralisé). Ces objets sont des entités passives (c'est-à-dire des objets Smalltalk standard) qui communiquent *point-à-point*, de manière *synchrone*, et selon un protocole *requête/réponse* traditionnel. Les objets *comportementaux* peuvent être vus comme des « *méta objets données* » [Mae 87] qui décrivent les caractéristiques comportementales (c'est-à-dire concurrence, répartition et robustesse) des objets *données*.

Exprimer les caractéristiques comportementales d'une application se résume à lier (dynamiquement) des objets comportementaux aux objets *données*. GARF considère deux types d'objets comportementaux : les *encapsulateurs* et les *postiers* (« *mailers* »). Les *encapsulateurs* enveloppent les objets *données* en contrôlant la réception et l'envoi des messages. Les *postiers* gèrent les communications (distantes) entre les encapsulateurs (voir la Figure 3).

GARF offre une bibliothèque de classes encapsulateurs et postiers. Par exemple, la classe encapsulateur `Replica` gère la création de multiples copies d'un objet *donnée*. La classe postier `Abcast` garantit que les différentes copies d'un encapsulateur reçoivent des requêtes concurrentes dans le même ordre. Les protocoles qui permettent de mettre en œuvre ces classes sont eux-mêmes représentés par des objets de première classe, et constituent une bibliothèque de classes protocoles [GFG 96, GG 97].

L'interception « transparente » des messages était le point fondamental pour la réutilisation du code fonctionnel (c'est-à-dire les objets *données*) dans un contexte d'informatique concurrente, répartie et robuste (avec les objets comportementaux). L'interception des messages de création a été réalisée grâce au *dictionnaire* `Smalltalk` et à la méthode `doesNotUnderstand:` (comme pour l'interception des messages dans Actalk). L'encodage et le décodage des messages pour les communications distantes ont été réalisés à l'aide de la bibliothèque standard BOSS (Binary Object Streaming Service).

5. Conclusion

Cet article a discuté les intérêts de l'utilisation de Smalltalk comme fondation pour le développement prototype d'environnements de programmation concurrente et répartie. Bien que Smalltalk ait été initialement conçu pour un mono-utilisateur et sur un mono-processeur, la combinaison des caractéristiques standard des langages à objets (encapsulation, généricité, réutilisation) avec les caractéristiques plus spécifiques à Smalltalk, telles la richesse de ses bibliothèques et l'extrême flexibilité du langage, en font une excellente fondation.

Nous avons rapidement décrit deux frameworks académiques et mentionné quelques produits commerciaux de programmation concurrente et répartie en Smalltalk, comme exemples témoins de notre constat. Nous considérons qu'il devrait continuer à y avoir un nombre croissant de projets et produits dans cette lignée. Il est clair que l'engouement actuel pour Java et ses qualités réelles en tant que langage de programmation, orienteront une bonne partie des développements futurs. Cependant, Smalltalk garde encore une certaine avance en matière de prototypage rapide du fait de sa maturité (langage, environnement, bibliothèques, frameworks et applications) et de ses caractéristiques propres (typage dynamique et environnement de développement pleinement interactif, méta-description et primitives réflexives).

Note : Cet article est une adaptation et une évolution à partir de l'article de langue anglaise « Smalltalk for concurrent and distributed programming » des mêmes auteurs, paru dans la revue Informatik/Informatique de la Swiss Informaticians Society, n° 1, février 1996, pages 16–19. Cet article a, de plus, bénéficié de diverses suggestions faites par les relecteurs de ce numéro spécial.

6. Références

[BvR 93] K. Birman et R. van Renesse, *Reliable Distributed Computing with the ISIS Tookit*, IEEE Computer Society Press, 1993.

[Bri 94] J.-P. Briot, Modélisation et classification de langages de programmation concurrente à objets : l'expérience Actalk, *Actes du « Colloque Langages et Modèles à Objets (LMO'94) »*, INRIA/IMAG/PRC-IA, Grenoble, France, pages 153–165. Également publié comme Rapport de Recherche LITP, n° 94.59, Paris, Octobre 1994.

[Bri 96] J.-P. Briot, An Experiment in Classification and Specialization of Synchronization Schemes, *Proceedings of the « 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96) »*, édité par K. Futatsugi et S. Matsuoka, LNCS, n° 1049, Springer-Verlag, Mars 1996, pages 227–249.

[BC 89] J.-P. Briot et P. Cointe, Programming with Explicit Metaclasses in Smalltalk-80, *Proceedings of the « Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89) »*, Special Issue of *ACM Sigplan Notices*, Vol. 24, n° 10, Octobre 1989, pages 419–432.

[BG 96] J.-P. Briot et R. Guerraoui, Objets pour la programmation parallèle et répartie : Intérêts, évolutions et tendances, dans le numéro spécial : *« Systèmes à objets: tendances actuelles et évolution »*, édité par Amedeo Napoli et Jean-François Perrot, *Technique et Science Informatiques (TSI)*, Vol. 15, n° 6, Hermès, Juin 1996, pages 765–800.

[GFG 96] B. Garbinato, P. Felber, et R. Guerraoui, Protocol Classes for Designing Reliable Distributed Environments, *Proceedings of the « European Conference on Object Oriented Programming (ECOOP'96) »*, édité par P. Cointe, LNCS, n° 1098, Springer-Verlag, Juin 1996, pages 316–343.

[GG 97] B. Garbinato et R. Guerraoui, Using the Strategy Design Pattern to Compose Reliable Distributed Protocols, *Proceedings of the « Usenix Conference on Object-Oriented Technologies and Systems (COOTS'97) »*, édité par S. Vinoski, Usenix, Juin 1997.

[GGM 94] B. Garbinato, R. Guerraoui, et K.R. Mazouni, Distributed Programming in GARF, *Object-Based Distributed Programming*, édité par R. Guerraoui, O. Nierstrasz, et M. Riveill, LNCS, n° 791, Springer-Verlag, 1994, pages 225–239.

[GGM 95] B. Garbinato, R. Guerraoui, et K.R. Mazouni, Implementation of the GARF Replicated Objects Platform, *Distributed Systems Engineering Journal*, n° 2, 1995.

[GR 83] A. Goldberg et D. Robson, *Smalltalk-80: the Language and its Implementation*, *Series in Computer Science*, Addison Wesley, 1983.

[Gue 95] R. Guerraoui, Les langages concurrents à objets, *Technique et Science Informatiques (TSI)*, Hermès, Octobre 1995, pages 945–973.

[Les 92] L. Lescaudron, *Prototypage d'environnements de programmation pour les langages à objets concurrents : une réalisation en Smalltalk-80 pour Actalk*, Thèse d'Université, LITP, Université Paris VI - CNRS, TH93.11, Mai 1992.

[Mae 87] P. Maes, Concepts and Experiments in Computational Reflection, *Proceedings of the « Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87) »*, Special Issue of *ACM Sigplan Notices*, Vol. 22, n° 12, Décembre 1987, pages 147–155.

[Mal 96] C. Malloth, *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*, PhD Thesis, École Polytechnique Fédérale de Lausanne, Suisse, 1996.

[Mey 93] B. Meyer, Systematic Concurrent Object-Oriented Programming, Special Issue on *« Concurrent Object-Oriented Programming »*, éditée par B. Meyer, *Communications of the ACM (CACM)*, Vol. 36, n° 9, Septembre 1993, pages 56–80.

[NWM 93] J. Nicol, T. Wilkes, et F. Manola, Object-Orientation in Heterogeneous Distributed Computing Systems, *IEEE Computer*, Vol. 26, n° 6, Juin 1993, pages 57–67.

[Weg 87] P. Wegner, Dimensions of Object-Based Language Design, *Proceedings of the « Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87) »*, Special Issue of *ACM Sigplan Notices*, Vol. 22, n° 12, Décembre 1987, pages 168–182.

Jean-Pierre Briot *est chargé de recherche au CNRS. Il est responsable du thème de recherche OASIS (Objets et Agents pour Systèmes d'Information et de Simulation) du Laboratoire d'Informatique de Paris 6 (LIP6). Il s'intéresse à la conception d'architectures logicielles fondées sur les concepts d'objet, de méta-objet, d'acteur et d'agent, et à leur application aux systèmes d'information décentralisés. Il est titulaire d'un doctorat (1984), puis d'une habilitation en informatique (1989), tous deux obtenus à l'Université Paris 6.*

Rachid Guerraoui *est chargé de recherche et d'enseignement au Département d'Informatique de l'École Polytechnique Fédérale de Lausanne. Ses centres d'intérêt actuels sont la conception et la mise en œuvre d'abstractions pour la programmation répartie. Il travaille en particulier sur les algorithmes d'accord réparti tolérants aux défaillances et sur les systèmes à objets répartis. Rachid Guerraoui a obtenu un diplome d'ingénieur en 1989, une thèse de doctorat en 1992, et un diplome d'habilitation à diriger les recherches en 1996.*