SYNTHÈSE

Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances

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

** LAFORIA, IBP, CNRS - Univ. Paris-VI*
4 place Jussieu
75252 Paris Cedex 05, France

The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, Japon

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

RÉSUMÉ. *Cet article étudie comment les notions et les acquis des objets ont été utilisés dans le contexte du parallélisme et de la répartition. Nous distinguons l'approche applicative, l'approche intégrée, et l'approche réflexive. L'approche applicative utilise les concepts de la programmation par objets, en tant que tels, pour la structuration de systèmes informatiques parallèles et répartis sous la forme de bibliothèques. L'approche intégrée vise à la fusion plus ou moins aboutie de concepts tels que : objet et activité, transmission de message et transaction. L'approche réflexive a pour objectif d'intégrer des bibliothèques de protocoles avec un langage de programmation.*

ABSTRACT. *This paper studies how object notions and current technology have been used in the context of parallelism and distribution. We consider an applicative approach, an integrative approach, and a reflexive approach. The applicative approach applies object-oriented concepts, as they are, to structure parallel and distributed computer systems through libraries. The integrative approach aims at merging concepts such as: object and activity, message passing and transaction. The reflexive approach aims at integrating protocol libraries within a programming language.*

MOTS-CLÉS : *Objet, transmission de message, parallélisme, répartition.*

KEY WORDS : *Object, message passing, parallelism, distribution.*

1. Introduction

Bien que la majorité des ordinateurs et des programmes restent éminemment séquentiels, la prise en compte du parallélisme et de la répartition s'impose peu à peu et de manière inéluctable. Des domaines d'application représentatifs sont, par exemple : le travail coopératif assisté par ordinateur [ELL 91] et les jeux électroniques multi-utilisateurs. La popularité du réseau Internet et des applications associées (telles que le « World Wide Web » (WWW) [BER 92]) laisse prévoir un engouement vers le développement de divers services répartis. Les concepts de la programmation par objets offrent de bonnes bases pour aborder ces nouveaux enjeux. Les concepts sont en effet à la fois assez forts pour assurer structuration, modularité et encapsulation, et assez souples pour recouvrir également granularités variées, hétérogénéité éventuelle et enfin de nombreux protocoles spécialisés. Ce n'est ainsi pas un hasard si la quasi-totalité des architectures de systèmes d'exploitation répartis développées actuellement se fondent sur la notion d'objet.

Il est intéressant de noter que lors de la genèse des concepts objets à la fin des années 60, ces concepts n'étaient pas spécifiquement restreints à la programmation séquentielle. Cependant les contraintes technologiques, et sans doute également culturelles, de l'époque, ont alors concentré les développements de la technologie objet dans le monde de la programmation séquentielle. Un premier enjeu est donc tout d'abord la redécouverte des concepts plus généraux, prenant en compte des activités et interactions potentiellement simultanées. Un deuxième enjeu, plus important encore, est l'intégration plus ou moins complète de protocoles et mécanismes pour gérer les différents aspects liés à cette simultanéité, et plus encore à la répartition des objets et leurs activités.

Cet article a pour ambition de récapituler en quoi le concept d'objet est une bonne fondation pour la programmation parallèle et répartie, et de distinguer les tendances actuelles pour intégrer les nouveaux enjeux qui en découlent. Il est important de noter que, bien que certains exemples seront utilisés pour illustrer les différentes tendances, l'objectif de cet article n'est pas de présenter exhaustivement les langages et systèmes à objets parallèles et répartis. En particulier, un autre article récent et complémentaire se concentre lui sur les différents modèles de programmation concurrente par objets [GUE 95b]. De plus, deux ouvrages récents, [GUE 94] et [BRI 96c], regroupent un certain nombre de contributions dans le domaine de la programmation parallèle et répartie par objets. Enfin, il faut noter aussi que des aspects complémentaires, tels que : formalismes et preuves de programmes, visualisation et mise au point, méthodologies d'analyse et de conception, techniques de mise en œuvre et d'optimisation, ne seront pas abordés dans cet article, malgré leur importance.

2. Enjeux et besoins

Du fait de l'extension continuelle de la portée des applications informatiques, les problèmes à traiter sont de plus en plus variés et complexes.

De pair avec cette tendance, les utilisateurs deviennent de plus en plus exigeants sur la qualité, la sûreté et enfin l'efficacité des services offerts.

2.1. Enjeux

Un premier enjeu est le *parallélisme*. Une des motivations est le gain d'efficacité visé en exploitant les architectures de machines à plusieurs processeurs. Ce n'est cependant pas la seule motivation. Pour de nombreux problèmes, la formulation se fait de manière beaucoup plus naturelle sous la forme d'activités simultanées et coordonnées, que sous la forme d'un algorithme séquentiel. Il faut donc tout d'abord faciliter une expression *logique* (*concurrence*) du parallélisme potentiel au niveau du programme, puis une utilisation optimale des ressources matérielles disponibles pour mettre en œuvre cette concurrence sous forme de *parallélisme* (*physique*) et donc de gain de performance.

Un deuxième enjeu est la *répartition*. Certaines applications sont intrinsèquement réparties (par exemple le travail coopératif). D'autres, pour des raisons de partage des ressources (par exemple une imprimante), de tolérance aux fautes, ou de répartition de charges, requièrent des architectures réparties. Comme pour la gestion du parallélisme, il est nécessaire, d'une part de faciliter l'expression des programmes sur de telles architectures, et d'autre part d'offrir les mécanismes sous-jacents adéquats de contrôle. Parmi ces mécanismes, on peut trouver par exemple le contrôle de concurrence entre transactions [BER 87], et la tolérance aux fautes [SIM 87].

Enfin il est de plus en plus désirable de pouvoir changer les modèles et mécanismes d'un système informatique pour pouvoir intégrer de nouveaux services, partenaires, et besoins, *au cours* de son fonctionnement. Ceci est lié au concept de système *ouvert*. Internet est un exemple représentatif et à large échelle de système ouvert.

2.2. Besoins

Nous pouvons maintenant résumer brièvement les principaux besoins qu'il faut pouvoir exprimer et satisfaire au mieux, dans le cadre des systèmes informatiques parallèles et répartis[1] :

- *(Dé)composition* : fondations conceptuelles et structurelles pour une bonne décomposition logique et physique de l'application en composants logiciels autonomes et susceptibles d'activités indépendantes ; et à l'inverse, fondations pour la composition de modules logiciels développés indépendamment.

1. Par *systèmes informatiques*, nous considérons ici : matériel (ordinateurs), logiciels (systèmes d'exploitation, langages et environnements de programmation), et applications. Notez que la plupart des besoins énumérés ici correspondent aux besoins classiques de génie logiciel. Les trois derniers besoins sont eux plus spécifiques.

- *Coordination* : protocoles évolués de synchronisation et de coordination des différentes activités pour assurer leur consistance et leur conjugaison.
- *Modularité* : indépendance des différents composants pour assurer leur interchangeabilité et leur protection.
- *Réutilisation* : généricité et réutilisation des composants logiciels.
- *Extensibilité et adaptabilité* : possibilité d'ajout dynamique de nouveaux services, composants et protocoles, en cours de fonctionnement.
- *Hétérogénéité* : diversité éventuelle de composants logiciels et matériels, de même que des protocoles et mécanismes qui peuvent être offerts.
- *Répartition* : mécanismes abstraits de gestion de la répartition et la duplication, statique et éventuellement dynamique (migration), des entités et services.
- *Sûreté* : protections et preuves de propriétés au niveau logiciel, et tolérance aux fautes au niveau matériel.
- *Efficacité* : utilisation optimale des ressources offertes par les architectures matérielles parallèles et réparties.

2.3. Classes de systèmes

Cet énoncé de besoins est relativement générique. Certains systèmes et applications pourront se concentrer sur certains d'entre eux au détriment d'autres. Nous pouvons ainsi regrouper trois grandes classes de systèmes informatiques parallèles ou/et répartis :

	principal objectif	*nombre de processeurs*	*structure*	*répartition des objets*	*tolérance aux fautes*
Multi-processeurs à mémoire partagée	efficacité	dizaine	homogène	non	rare
Ordinateurs massivement parallèles	efficacité	centaine ou plus	homogène	oui, mais surtout statique	rare
Informatique répartie	coopération	indé-terminé	hétérogène	pré-exis-tante et dynamique	indispen-sable

3. Objets comme fondation

La méthodologie de programmation par objets offre d'excellentes fondations pour aborder les besoins exprimés ci-dessus comme nous allons le voir dans cette section.

3.1. Concepts et avantages

La programmation par objets repose sur quelques concepts simples et unificateurs. Un programme se décompose en un ensemble de modules autonomes, appelés *objets*, qui interagissent au travers d'un protocole de communication unifié, la *transmission de message*. Les objets correspondent à différentes entités des domaine et problème considérés. Un objet se présente comme une *capsule* contenant à la fois des données et des opérations opérant sur celles-ci. Le principe de transmission de message introduit une séparation entre la signature des opérations qui pourront être invoquées de l'extérieur (appelée *interface*) et la représentation interne de l'objet. Ceci est appelé le principe d'*encapsulation* et il favorise la séparation entre spécification et mise en œuvre. Il permet également abstraction et *généricité*. Ainsi, par exemple, l'opérateur addition (+) des objets numériques est naturellement générique, étant interprété différemment (addition entière, flottante, rationnelle, etc.) suivant le type de l'objet numérique qui reçoit une telle invocation.

Ces principes fondateurs de la programmation par objets permettent à la fois décomposition, modularité et protection. De manière à favoriser l'abstraction et la factorisation d'objets semblables, la plupart des langages et systèmes à objets reposent sur la notion de *classe*, comme abstraction et factorisation d'objets. De plus [2], des mécanismes de spécialisation, et en premier lieu l'*héritage*, permettent de réutiliser des propriétés déjà définies par certaines classes d'objets pour les étendre et/ou les modifier partiellement. Ces mécanismes augmentent encore la généricité et la réutilisation des programmes et ont permis de valider les intérêts de la programmation par objets au niveau du génie logiciel [MEY 88]. Enfin, la programmation par objets intègre de manière naturelle une gestion dynamique des ressources, puisque l'on peut créer de nouveaux objets au fur et à mesure des besoins.

Comme nous l'avons souligné dans l'introduction, les concepts fondateurs de la programmation par objets sont à la fois suffisamment forts pour structurer et rendre modulaire les programmes, et en même temps suffisamment souples pour permettre une grande généricité des programmes. Ceci provient de la distinction entre interface (services offerts) et intérieur (représentation/mise en œuvre) d'un objet. Notez également que la granularité même de l'objet n'est pas fixée. Ce dernier point est particulièrement intéressant pour réaliser des architectures logicielles hétérogènes.

3.2. Concurrence potentielle

Les concepts fondateurs des objets, tels qu'ils ont été exprimés à la fin des années 60, contenaient déjà implicitement les promesses de la programmation

2. Peter Wegner [WEG 87] a d'ailleurs proposé une approche terminologique par couches successives : *object-based* pour les langages et systèmes fondés sur la notion d'*objet*, *class-based* si l'on ajoute le concept de *classe*, et enfin *object-oriented* en présence du mécanisme d'*héritage*.

concurrente. D'ailleurs, le premier langage de programmation à parler d'objets, Simula-67 [Bir 73], offrait déjà les germes d'une activité autonome et simultanée associée à chaque objet. Une classe Simula-67 peut définir un corps de programme (une séquence d'instructions appelée « body » qui sera exécutée par chaque objet lors de sa création). Ce corps de programme est plus qu'un simple texte d'initialisation puisque l'objet peut alors explicitement se suspendre et relancer un autre objet sous la forme de coroutines, permettant ainsi une exécution quasi-simultanée. Les objets se montraient ainsi dès l'origine proches de la notion d'activité simultanée (*processus*). Objets et processus partagent d'ailleurs de nombreuses caractéristiques communes (variables, données persistantes, encapsulation, moyens de communication) comme le souligne Bertrand Meyer [Mey 93b].

Cependant, pour des raisons technologiques (les ordinateurs de l'époque étaient rarement parallèles) et culturelles (la programmation était construite sur la notion de séquence d'instructions, et elle le reste d'ailleurs encore majoritairement), cette dimension d'activité concurrente a plutôt régressé parmi les successeurs de Simula. Ces potentialités n'ont commencé à être réellement développées que plus tard, au cours du milieu des années 80, les langages d'acteurs [Lie 87, Agh 86] faisant figure de nouveaux pionniers.

3.3. Répartition potentielle

Un objet apparaît de manière naturelle comme une unité potentielle de répartition. La transmission de message 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). Si l'objet appelant et l'objet appelé se trouvent sur deux sites différents, la métaphore de l'envoi de message prend alors tout son sens, un réel « message » étant transmis à travers le réseau. Enfin l'autonomie et relative « complétude » des objets (comme capsule de données et d'opérations associées) facilite la prise en compte de migration ou de duplication éventuelles.

Il est intéressant de noter que l'architecture *client/serveur*, à la base de la plupart des systèmes répartis ou d'interfaces homme-machine actuels, possède déjà une partie des caractéristiques « objet » [Nic 93]. Cependant la dichotomie client/serveur n'est pas statique en programmation par objets, puisque tout objet peut être considéré comme un client ou un serveur suivant qu'il envoie ou reçoit un message. De fait, la grande majorité des projets d'architectures réparties actuelles est fondée sur le concept d'objet. Le standard ODP (« Open Distributed Processing ») [Nic 93], censé définir un modèle général de programmation répartie, est d'ailleurs conçu selon le modèle objet. Enfin remarquons qu'il existe une double tendance : d'une part les chercheurs issus de la communauté « langages de programmation par objets » étendent les environnements de programmation vers des architectures réparties, et d'autre part les chercheurs issus de la communauté « systèmes d'exploitation » adoptent le modèle objet pour structurer les architectures de systèmes.

3.4. Limitations

Malgré les potentialités de ses concepts fondateurs, la technologie objet, développée quasi-exclusivement dans un contexte séquentiel, ne se transpose pas toujours parfaitement telle quelle à la programmation parallèle et répartie. Par exemple, le mécanisme d'héritage induit un certain nombre de limitations à l'égard du parallélisme comme de la répartition comme nous le verrons au § 6.6. Plus encore, le concept d'objet, en tant que tel, n'apporte pas des réponses complètes aux besoins énoncés (au § 2.2).

Il se trouve qu'en parallèle au développement de la technologie objet, diverses communautés, et en particulier : programmation parallèle, systèmes d'exploitation, et bases de données, ont abordé ces enjeux de parallélisme et de répartition. Ainsi différents types d'abstractions et de mécanismes, tels que : synchronisation, « capabilities », et transactions, ont été proposés, pour aborder différents besoins et enjeux tels que : contrôle de la concurrence, gestion des ressources, et tolérance aux fautes. Un enjeu d'importance est donc l'adaptation de tels acquis dans la technologie et la méthodologie objet. Il est également important de pouvoir réutiliser au mieux la technologie objet déjà validée, voire jusqu'à des modules/programmes déjà développés.

Toute la question est ainsi en conséquence : « De quelle manière doit-on lier les concepts objets aux enjeux et acquis de la programmation parallèle et répartie ? ». Autrement dit, doit-on procéder par application, identification, extension, évolution, etc. La suite de cet article a justement pour ambition d'éclairer cette question et les tentatives actuelles pour y répondre.

4. Objets pour le parallélisme et la répartition : différentes approches

Pour utiliser les concepts objets dans le contexte de la programmation parallèle et répartie, plusieurs voies sont possibles, comme en témoigne la diversité des systèmes et projets proposés. Nous les regroupons ici en trois approches générales.

4.1. Approches

La première approche est une approche *applicative* (§ 5). Il s'agit d'appliquer, *tels quels*, les concepts objets à la conception de programmes et systèmes parallèles et répartis. Les différents composants de ces systèmes (processus, fichiers, serveurs de nom, etc.) seront alors représentés par différentes classes spécifiques d'objets. Ceci assurera la généricité des architectures logicielles. La programmation reste ainsi essentiellement de la programmation par objets séquentielle traditionnelle. On procède donc avec une approche d'extension par *bibliothèques* plutôt que par extension des concepts et des langages de programmation les incarnant.

La deuxième approche est une approche *intégrée* (§ 6). Il s'agit alors d'étendre les concepts fondateurs des objets pour y intégrer les enjeux de la programmation parallèle et répartie. Les systèmes qui suivent cette approche intègrent (unifient) souvent objet avec activité (la notion d'*objet actif*, voir au § 6.3), et la transmission de message avec différents protocoles de synchronisation (synchronisation client/serveur, transactions, etc., voir au § 6.4). Les intégrations ne se font cependant pas toujours sans mal, et certains aspects créent des conflits, par exemple : héritage et synchronisation, duplication et communication (voir au § 6.6).

La troisième approche est une approche *réflexive* (§ 7). Il s'agit de séparer le programme proprement dit, des différents aspects de sa mise en œuvre (modèle de calcul, de communication, de répartition, etc.), eux-mêmes décrits dans un *méta*-programme. La programmation s'applique ainsi au contrôle de sa propre mise en œuvre, d'où le nom donné de « *réflexion* ». La réflexion permet également d'abstraire la gestion des ressources (équilibre de charges, dépendance du temps, etc.) et de la décrire avec toute la puissance d'un langage de programmation.

4.2. Complémentarité

Il est important de souligner dès à présent que ces trois approches ne sont en concurrence qu'en apparence. Pour être plus précis, leurs développements respectifs ont des objectifs complémentaires. L'approche *applicative* est destinée aux concepteurs de systèmes et vise à identifier les abstractions fondamentales des systèmes informatiques parallèles et répartis. L'approche *intégrée* est destinée aux concepteurs d'applications, et vise à la définition d'un langage de programmation de haut niveau comportant un nombre minimal de concepts. L'approche *réflexive* est destinée aux concepteurs d'application qui souhaitent spécialiser le système pour les besoins propres à leur type d'application, et de manière duale aux concepteurs de système qui souhaitent ainsi concevoir leur système selon une telle architecture « ouverte » et adaptable. L'approche réflexive vise donc à la définition d'une architecture permettant la spécialisation dynamique du système avec un impact minimal sur le programme de l'application.

5. Approche applicative

5.1. Besoins de modularité et de structure

L'idée sous-jacente à l'approche *applicative* est d'appliquer les concepts d'encapsulation et d'abstraction, voire de classe et d'héritage, à la conception et la mise en œuvre des systèmes parallèles et répartis. Il s'agit de programmer un système parallèle ou réparti avec une méthodologie à objets et dans un langage à objets. Le résultat est une bibliothèque de classes (organisée selon ce qui a récemment été appelé un « *pattern* », ou encore un « *framework* » [Gam 95]) représentant le système en question.

La première motivation est d'accroître la modularité de ces systèmes. En effet, en décomposant un système en un ensemble de modules dotés d'interfaces bien définies, on peut espérer pouvoir modifier la mise en œuvre de certains composants avec un minimum d'impact sur les autres. D'une certaine manière, une telle modularité permettrait d'assurer la portabilité de ces systèmes.

La deuxième motivation est de mieux structurer les systèmes parallèles et répartis, en évitant des conceptions « à la Unix », dans lesquelles les niveaux d'abstractions sont difficilement discernables. Il est bien évidemment plus facile de comprendre un système lorsque ce dernier est constitué d'un ensemble de composants dont les interfaces (les services) sont bien définis. Ceci a d'ailleurs donné lieu à la conception actuelle des systèmes d'exploitation récents, tels que Chorus [Roz 92], Mach [Acc 86], et Choices [Cam 93], fondés sur un noyau minimal (« micro-kernel ») et dont les différents services sont assurés par différents serveurs spécialisés. Le bénéfice escompté est, d'une part de rendre les systèmes parallèles et répartis plus facilement utilisables et extensibles, et d'autre part d'améliorer les performances en n'utilisant dans chaque exécution que les modules qui sont strictement nécessaires.

Nous présentons maintenant trois exemples de l'approche applicative : tout d'abord les cas de Smalltalk et Eiffel, qui introduisent divers aspects de programmation concurrente et répartie sous la forme de bibliothèques de classes, puis le cas du système d'exploitation Choices qui illustre l'application de la méthodologie objet à la structuration et la généricité de systèmes d'exploitation répartis.

5.2. L'exemple Smalltalk

L'une des principales originalités de l'environnement de programmation Smalltalk-80 [Gol 83] est d'offrir, d'une part un langage à objets minimal, et d'autre part des bibliothèques de classes représentant divers mécanismes et outils de programmation, qui font ainsi la richesse de l'environnement. Il s'agit donc bien d'un « tout objet » uniformément appliqué à l'ensemble des concepts de programmation. Par exemple, les structures de contrôle (tests, boucles, itérations) sont représentées par des méthodes utilisant la seule généricité de l'envoi de message, et non pas par des tests booléens primitifs comme dans la plupart des autres langages de programmation. Leurs arguments sont des « blocs/contextes de programmes », qui sont des portions de programme, délimitées par des crochets (`[` et `]`), et dont l'évaluation est différée. Ils sont représentés comme instances de la classe `BlockContext`. Le fait que les structures de contrôle sont en Smalltalk-80 des méthodes standard comme les autres, permet au programmeur d'en définir si besoin de nouvelles à son goût.

5.2.1. Programmation concurrente – La notion de bloc/contexte est également la base du multi-tâche en Smalltalk-80. En réponse au message `fork`, un bloc crée un objet *processus* qui exécute le bloc de programme en question, de manière concurrente à la séquence d'appel. Un processus est représenté par une instance de la classe `Process`. Les méthodes associées à cette classe permettent de gérer les processus (suspendre, relancer, changer sa priorité, etc.). Le séquenceur est lui-même représenté par un objet, instance de la classe `ProcessorScheduler`. Enfin, la primitive de synchronisation de base est le (classique) sémaphore, représenté par la classe `Semaphore`. Des abstractions de plus haut niveau existent également en standard : la classe `SharedQueue` pour communication et synchronisation entre processus, et la classe `Promise` pour représenter l'évaluation anticipée d'une expression par un processus concurrent.

Grâce à cette approche par bibliothèques, les mécanismes nécessaires à la programmation concurrente sont encapsulés dans des concepts bien définis. De par leur structuration en une hiérarchie de classes, ces concepts sont plus compréhensibles que s'ils étaient fournis en vrac comme des primitives d'un langage. Ils sont également très génériques et bien entendu extensibles. Ainsi un programmeur avisé peut étendre les classes existantes et définir de nouvelles abstractions pour la programmation concurrente, tels divers mécanismes de synchronisation (moniteurs, gardes, compteurs de synchronisation, etc.), comme par exemple dans les plates-formes Simtalk [Béz 87] et Actalk [Bri 94, Bri 96b]. De plus, dans le cadre du projet Actalk, le séquenceur standard a été étendu en un séquenceur générique, pour paramétrer et classifier différentes stratégies de séquencement [Les 92].

5.2.2. Programmation répartie – Bien qu'étant un environnement de programmation fondamentalement mono-utilisateur et mono-processeur, Smalltalk-80 offre également un certain nombre de bibliothèques de base permettant la construction de mécanismes de répartition [Bri 96a]. Il existe des bibliothèques quasi-standard pour les communications distantes (sockets Unix, remote procedure call). De plus une bibliothèque standard de stockage de structures d'objets : le « Binary Object Streaming Service (BOSS) », offre un mécanisme de base d'encodage/représentation des objets pour construire divers mécanismes de programmation répartie, tels que : persistance, encodage des communications réparties, et transactions.

Ainsi, dans le cadre du projet GARF [Gar 95], une bibliothèque de classes a été mise en œuvre pour la programmation répartie, comportant divers modèles de gestion d'objets (persistant, dupliqué, etc.) et de communication (« multicast », atomique, etc.). Le produit HP Distributed Smalltalk offre également un ensemble de services répartis selon le standard OMG (CORBA, voir au § 6.5.5), eux-mêmes mis en œuvre en Smalltalk-80.

5.3. L'exemple d'Eiffel

Eiffel est un langage à objets qui a été conçu pour appliquer la méthodologie et les techniques à objets au génie logiciel [MEY 88]. Bien que destiné à la programmation séquentielle, de nombreux travaux ont par la suite été menés pour étendre Eiffel vers la prise en compte de la concurrence, du parallélisme et de la répartition (certaines de ces approches ont été regroupées dans le dossier spécial [MEY 93a]).

L'approche défendue par le concepteur d'Eiffel lui-même [MEY 93b] est *applicative*, mais surtout *minimaliste*. Il s'agit d'élargir la portée des concepts et mécanismes déjà existants (et donc d'augmenter leur généricité) sans en introduire de nouveaux, du moins le strict minimum. Ainsi, la sémantique des assertions d'Eiffel, sous forme de pré- et post-conditions à satisfaire, se trouve redéfinie dans un contexte concurrent comme une attente tant que les conditions ne sont pas réunies[3]. Cette approche est très séduisante par son économie de moyens, et de ce fait sa simplicité, mais elle ne peut hélas couvrir tout le champ des besoins.

Des constructions et mécanismes supplémentaires, pouvant être nécessaires, sont décrits et mis en œuvre à l'aide de bibliothèques, en appliquant la méthodologie objet pour les organiser. Ainsi par exemple, la classe `Concurrency` [KAR 93] encapsule une activité associée à l'objet et la transmission de message sans attente et distante.

Il nous faut enfin signaler l'approche relativement originale de l'environnement ÉPÉE de programmation répartie [JÉZ 93]. ÉPÉE introduit le parallélisme (et la répartition) au niveau des données pour des structures de données complexes (par exemple des matrices), suivant en cela le modèle « Single Program Multiple Data (SPMD) »[4]. La mise en œuvre se fait par bibliothèques de structures abstraites réparties sur plusieurs processeurs, et ce sans aucun ajout au langage Eiffel.

5.4. L'exemple Choices

Choices [CAM 93] est un système d'exploitation développé à l'Université de l'Illinois depuis 1987. La principale caractéristique de ce système d'exploitation est qu'il a été conçu pour être *générique*. L'objectif de cette généricité est, non seulement de pouvoir être porté facilement sur diverses machines, mais également de pouvoir faire varier différentes caractéristiques telles que : formats de fichiers, réseaux de communication, et modèles de mémoire (partagée ou

3. Selon les principes de la synchronisation comportementale, détaillée au § 6.4.3.

4. Le parallélisme de ÉPÉE est ainsi très différent de celui des langages à *objets actifs* [YON 87]) qui introduit le parallélisme au niveau de l'activation, selon le modèle « Multiple Instructions Multiple Data (MIMD) ». Il est vrai que les objets représentent la dualité (et l'unification) entre données d'une part, et procédures (activation potentielle) d'autre part. ÉPÉE révèle donc d'une certaine manière la dimension « données » des objets pour le parallélisme.

répartie). Une méthodologie à objets est proposée pour la conception et la programmation, à la fois d'applications réparties, et d'extensions du noyau Choices.

Choices est mis en œuvre en C++, et pour être plus précis, Choices a été développé à partir d'une bibliothèque de classes C++, définie pour la programmation répartie. Aussi bien le matériel que l'interface d'application et les ressources du système sont modélisés par des classes. Parmi les classes définies, on peut citer la classe `ObjectProxy` qui réalise les communications distantes entre objets, ainsi que les classes `MemoryObject` et `FileStream` qui gèrent la mémoire, et la classe `ObjectStar` qui étend en quelque sorte le concept de pointeur. Cette dernière classe permet de rendre transparentes les invocations distantes sans nécessiter d'étape de pré-compilation. De plus, la classe `ObjectStar` permet la récupération automatique de mémoire (un objet est détruit lorsqu'il n'y a plus de référence sur lui).

Même si un système comme Choices reste un prototype universitaire, l'un de ses grands mérites est d'avoir montré qu'un système d'exploitation réparti pouvait être développé en suivant une méthodologie à objets dans un langage à objets. Cependant, le champ d'applications qui reposent sur le système Choices est encore plutôt restreint et ne permet pas de juger de sa facilité d'utilisation. Les nombreux développements autour du système (par exemple, le récupérateur automatique de mémoire adaptable) permettent néanmoins d'apprécier dès maintenant, dans une certaine mesure, son extensibilité.

5.5. A la recherche d'abstractions standard

De manière plus générale, l'idée derrière l'approche applicative est la définition et la mise en œuvre d'abstractions standard permettant, d'une part de simplifier la programmation parallèle et répartie, et d'autre part de tirer parti de la flexibilité des systèmes d'exploitation sous-jacents.

L'exemple le plus significatif dans la programmation concurrente est l'abstraction de synchronisation nommée *sémaphore* qui, grâce à une interface bien définie (opérations **`wait`** et **`signal`**), et un comportement connu (métaphore des sémaphores ferroviaires), constitue désormais un standard de la programmation concurrente. Un tel type d'abstraction sert ensuite de fondation pour construire divers mécanismes de synchronisation plus élaborés. Les possibilités de classification et de spécialisation de la programmation par objets sont particulièrement appropriées pour rendre compte d'une telle bibliothèque/hiérarchie d'abstractions, par exemple dans les plates-formes Simtalk et Actalk déjà citées au § 5.2.

Dans un contexte réparti, Andrew Black [Bla 91] a proposé une approche similaire en suggérant, comme premier exercice, de décomposer le concept de transaction en un ensemble d'abstractions. L'idée de cet exercice est de représenter des concepts tels que le « verrouillage », l'« atomicité » et la « persistance », par un ensemble d'objets que doit fournir un système pour supporter des transactions. La modularité d'une telle approche permettrait de définir divers modèles de transactions, adaptés à des applications particulières. Par exemple, une application de travail coopératif ne nécessite pas les mêmes contraintes de contrôle de concurrence entre transactions qu'une application de gestion bancaire[5]. Les concepts présentés dans les projets Venari [WIN 94] et Phoenix [GUE 95a] vont dans cette direction, en permettant de définir différents modèles transactionnels à partir d'abstractions minimales.

5.6. Bilan de l'approche applicative

En résumé, l'approche applicative tente de réduire la complexité des systèmes parallèles et répartis, en les décomposant en une bibliothèque de classes. Chaque service du système est alors représenté par un objet. Cet objectif de modularité est très important car les systèmes parallèles et répartis sont des systèmes complexes, faisant intervenir des mécanismes de très bas niveau (par exemple la communication à travers le réseau pour un système réparti). De plus, ce sont des systèmes souvent développés par des équipes de programmeurs pour lesquelles la modularité est fondamentale. Enfin, la difficulté de la maintenance et de l'extension des systèmes « à la Unix » est principalement dûe à un manque de modularité.

Bien que des tentatives soient entreprises dans ce sens, il est encore trop tôt pour considérer qu'il existe une bibliothèque de classes susceptible de devenir un standard pour la programmation parallèle ou répartie. Les difficultés d'un tel exercice (trouver les abstractions adéquates) résident, d'une part dans une bonne compréhension des mécanismes minimaux nécessaires pour ces types de programmation, et d'autre part dans un consensus sur ces mécanismes. Un tel consensus devrait mettre en jeu différentes communautés de chercheurs : langages de programmation, systèmes d'exploitation, et de certains types d'applications réparties (en particulier les bases de données pour des mécanismes transactionnels). Le fait qu'une abstraction comme le sémaphore soit devenue un standard pour la synchronisation permet néanmoins de considérer que ces difficultés sont surmontables.

6. Approche intégrée

6.1. Besoins d'unification

La multitude de concepts manipulés constitue l'une des difficultés majeures de la programmation parallèle et répartie. En plus des concepts classiques de manipulation de données d'un langage traditionnel, la programmation parallèle et répartie introduit des concepts tels que : *processus*, *sémaphore*, *moniteur*, *transaction*, etc. L'approche applicative permet de bien structurer

5. La seconde exige des garanties plus fortes (sérialisation stricte des transactions), à travers un mécanisme de verrouillage, alors que la première peut se passer d'un tel mécanisme.

les mécanismes de parallélisme et de répartition, mais a tendance à laisser le programmeur en charge de deux tâches distinctes : d'une part la programmation en terme d'objets, et d'autre part la gestion du parallélisme et de la répartition (également exprimée à l'aide d'objets, mais *pas les mêmes !*).

De plus, la programmation à l'aide de bibliothèques peut devenir un peu lourde, comme la programmation du parallélisme et de la répartition s'ajoutent au style standard de programmation. Ainsi par exemple, la classe `Concurrency`, décrite au § 5.3, oblige à une certaine dose de manipulation explicite de messages (voir en particulier [KAR 93, p. 109–11]), par opposition au modèle standard et implicite de transmission de message.

De manière à simplifier la programmation, il est alors possible d'*intégrer* de tels concepts et mécanismes directement dans un langage ainsi étendu (par exemple Eiffel// [CAR 93]), ou bien encore de créer un nouveau langage (par exemple ABCL/1 [YON 90]).

En résumé, au lieu de laisser la programmation d'une application et la programmation du parallélisme et de la répartition relativement orthogonales, l'approche *intégrée* vise à les fusionner en intégrant/unifiant les concepts autant que possible, pour offrir au programmeur un cadre conceptuel (objet) unique et simplifié.

6.2. Niveaux d'intégration

Il existe plusieurs niveaux d'identification et d'intégration entre les concepts objets et les concepts du parallélisme et de la répartition. Nous en considérons trois principaux qui sont indépendants. De ce fait, comme on va le voir par la suite, tel ou tel langage ou système pourra suivre un niveau d'intégration, mais pas un autre.

Une première intégration des concepts d'objet et d'activité (processus ou tâche) conduit au concept d'*objet actif*. En effet, un objet et un processus peuvent tous deux être considérés comme des unités contenant des données et des traitements (voir au § 3.2).

Une deuxième intégration conduit à associer la synchronisation à l'activation des objets, on parlera alors d'*objet synchronisé*. La transmission de message est alors considérée comme une synchronisation implicite entre émetteur et récepteur. En général, on associe en plus des mécanismes de contrôle de la concurrence des invocations au niveau d'un objet, par exemple en associant une *garde* au niveau de chaque méthode.

Remarquons qu'un objet actif implique déjà une forme simple d'objet synchronisé, puisque l'existence d'une activité privée à l'objet séquentialise de fait les invocations. Par contre, certains langages ou systèmes, comme Guide [BAL 94] ou Arjuna [PAR 88], associent la synchronisation aux objets tout en laissant objets et activités séparés.

Un troisième niveau d'intégration conduit à considérer l'objet comme unité de répartition, on parlera d'*objet réparti*. Les objets sont alors vus comme des entités pouvant être réparties et dupliquées sur différents sites. Ceci conduit également à étendre la métaphore de transmission de message à la communication à travers un réseau quelconque.

6.3. Objets actifs

L'idée sous-jacente au concept d'objet actif consiste à considérer l'objet comme doté d'une ressource de calcul, c'est-à-dire doué d'activité propre. Cette approche, naturelle, a eu une grande influence. L'ouvrage « Object-Oriented Concurrent Programming » [Yon 87] regroupe un certain nombre de langages de programmation de cette famille, les langages d'acteurs [Lie 87, Agh 86] faisant en la matière figure de pionniers. Le concept d'objet actif a eu beaucoup de succès dans le domaine de l'intelligence artificielle, car il constitue une fondation naturelle pour construire des « agents » de plus haut niveau, pour des problèmes de gestion répartie de la connaissance (appelés « systèmes multi-agents ») [Gas 92].

Au niveau du traitement des messages par un objet actif donné, le modèle par défaut est celui d'un objet traitant les requêtes de manière séquentielle (« acteur sérialisé »). La concurrence provient donc uniquement des activités indépendantes des divers objets (à ce titre on parle de concurrence *inter-objets*). Si on permet à un objet de traiter non pas une, mais plusieurs requêtes simultanément, on parle alors de concurrence *intra-objet*, et il faut en général un contrôle supplémentaire pour assurer la consistance de l'état de l'objet, comme nous allons le voir dans le paragraphe suivant.

6.4. Objets synchronisés

La présence d'activités simultanées impose la présence d'un certain degré de synchronisation, autrement dit de restriction de la concurrence entre activités, de manière à assurer un déroulement correct du programme. La synchronisation s'associe naturellement aux objets et à leur communication, comme nous allons le voir, à travers plusieurs (sous)-niveaux d'identification.

6.4.1. Synchronisation de la transmission de message – La transposition immédiate du principe de transmission de message d'un univers séquentiel à un univers concurrent entraîne une synchronisation implicite de l'émetteur du message (appelant) sur le récepteur (appelé). On parle de transmission *synchrone* : l'objet appelant attend la fin de l'exécution de la méthode invoquée et le retour de la réponse pour poursuivre son exécution.

Dans le cas des objets actifs, l'appelant et l'appelé possèdent des activités indépendantes. Il s'avère alors intéressant d'introduire un type de transmission *asynchrone*, où l'appelant poursuit son activité aussitôt le message envoyé, c'est-à-dire qu'il n'attend pas la fin de l'exécution de la méthode invoquée. Cette forme de transmission introduit ainsi une concurrence à travers la communication. Elle est bien adaptée à une architecture répartie car l'objet appelé/serveur peut être situé sur une machine distante et le temps de communication, ajouté au temps d'exécution de la méthode invoquée, peut être considérable. Enfin, la transmission avec réponse(s) *anticipée*(s) (par exemple en ABCL/1 [YON 90]) est une forme « mixte » de transmission asynchrone. Elle permet à l'appelant d'obtenir une promesse de réponse (appelée un « objet futur »), immédiatement sans attendre la fin du traitement effectif.

6.4.2. Synchronisation des invocations au niveau de l'objet – L'association (naturelle) de la synchronisation à la transmission de messages, c'est-à-dire à l'invocation des requêtes, offre l'avantage de traiter de manière transparente une partie de la synchronisation. Ceci a en effet l'intérêt de rendre transparente aux clients de l'objet la synchronisation de leurs requêtes, celles-ci étant traitées au niveau de l'objet acceptant les requêtes.

Dans le cas général d'objet sans concurrence interne, les invocations sont, par défaut, traitées l'une après l'autre, suivant leur ordre d'arrivée. Cependant un contrôle supplémentaire plus fin ou/et au contraire plus global peut être nécessaire, comme nous allons le voir.

6.4.3. Niveaux de synchronisation – Nous distinguerons trois niveaux de synchronisation supplémentaires éventuels. Ce sont trois niveaux successifs de contrôle correspondant respectivement à l'intérieur d'un objet, à son interface, et enfin à la coordination entre plusieurs objets :

- *Synchronisation intra-objet* : Dans le cas de concurrence *intra-objet* (traitement simultané de plusieurs requêtes), il est nécessaire d'assurer un certain contrôle de cette concurrence pour préserver la consistance de l'état de l'objet. On exprime alors, en général, les restrictions en terme d'exclusions entre opérations. Ainsi, par exemple, plusieurs lecteurs peuvent accéder en lecture simultanément à un même livre. Par contre, l'accès en écriture par un écrivain exclut tous les autres (écrivains comme lecteurs) [6].
- *Synchronisation comportementale* : Il se peut qu'un objet ne puisse temporairement traiter certains types de requêtes qui font pourtant partie de son interface. Ainsi, par exemple, un tampon de taille bornée (« bounded buffer ») ne pourra accepter une requête d'insertion (`put:`) si/tant qu'il est plein. Plutôt que de signaler une erreur, il est en général plus judicieux de laisser en attente une telle requête en attendant que la condition d'accès soit satisfaite. Ceci permet une transparence des invocations de services entre objets.
- *Synchronisation inter-objets* : On peut enfin désirer assurer une cohérence, non plus seulement individuelle, mais globale entre de multiples objets. Prenons l'exemple d'un transfert de fonds entre deux comptes

6. Dans le cas d'une exclusion mutuelle systématique, on retombe sur le cas d'un objet séquentiel au niveau de son traitement des requêtes (§ 6.3).

bancaires. En l'occurrence, on veut assurer l'indivisibilité, au sens transactionnel [Ber 87], du transfert. Cela consiste à rendre invisible un état transitoire et inconsistant où le premier compte a été débité mais le second n'a pas encore été crédité du montant équivalent. La synchronisation comportementale n'est alors plus suffisante. Il faut introduire une synchronisation qui met en œuvre les deux objets représentant les comptes bancaires.

6.4.4. Formalismes de synchronisation – De nombreux formalismes de synchronisation ont été proposés pour assurer ces différents niveaux de contrôle, aucun formalisme ne couvrant encore l'ensemble des niveaux. La plupart sont issus de travaux antérieurs ou, du moins, non spécifiques à la méthodologie par objets. En particulier, divers protocoles de synchronisation sont issus du domaine des systèmes d'exploitation, et le modèle des transactions est issu des besoins des bases de données. Ces protocoles représentent en conséquence un effort, plus ou moins poussé, d'intégration de ces formalismes de synchronisation dans le modèle des objets.

Cette intégration est en général aisée. Les formalismes *centralisés*, tels que les chemins de synchronisation (« path expressions »), qui spécifient de manière abstraite les possibles entrelacements ou séquences d'invocations, s'associent de manière naturelle au niveau de la classe (par exemple dans le langage Procol [Bos 91]). Un autre exemple est le concept de « body » : une procédure centrale qui décrit explicitement les types de requêtes que l'objet va accepter au cours de son activité[7]. On le trouve en particulier dans les langages POOL [Ame 87] et Eiffel// [Car 93] (où il est nommé « live routine »).

Les formalismes *décentralisés*, en particulier les *gardes* ou conditions booléennes d'activation, s'associent eux de manière naturelle au niveau de chaque méthode (par exemple dans le langage/système Guide [Bal 94]). Les compteurs de synchronisation (compteurs mémorisant le statut des invocations pour chaque méthode, c'est-à-dire le nombre d'invocations reçues, débutées et terminées), associés aux gardes, permettent d'exprimer un contrôle fin de synchronisation intra-objet, par exemple pour les lecteurs et écrivains.

Enfin, le formalisme des *comportements abstraits* est très approprié à la synchronisation comportementale (voir au § 6.4.3). Le principe est le suivant : un objet se conforme à un certain comportement abstrait auquel correspond un ensemble de méthodes autorisées. Dans le cas du tampon borné (évoqué au § 6.4.3), trois comportements abstraits sont suffisants : `vide`, `plein`, et `intermédiaire`. L'état abstrait `intermédiaire` peut d'ailleurs s'exprimer comme union de `vide` et `plein`, et est ainsi le seul à autoriser les deux méthodes `get` et `put:`. Après traitement d'une invocation, le comportement abstrait est recalculé, et peut ainsi changer suivant l'état ou/et la disponibilité des services de l'objet.

7. Ce concept est issu du concept de « body » de Simula-67 décrit au § 3.2.

Notons, dès maintenant, que bien que ces intégrations soient en général relativement aisées, le problème de la réutilisation des spécifications de synchronisation se pose néanmoins (il sera abordé au § 6.6.1).

6.5. Objets répartis

Un objet représente une unité indépendante d'exécution, contenant données, traitements, voire des ressources propres de mise en œuvre (activité). Il est donc naturel d'identifier l'objet comme unité de répartition, et de duplication éventuelle. Cela permet de considérer une application répartie comme un ensemble d'objets, éventuellement situés sur des processeurs distincts. La transmission de message recouvre ainsi la notion d'invocation locale ou distante suivant les cas, et également l'inaccessibilité éventuelle d'un objet/service.

6.5.1. Invocation distante – Le mécanisme fondamental d'un système à objets répartis est l'invocation distante. Il est en général admis que, pour faciliter la programmation d'applications réparties, le mécanisme d'invocation distante doit, par défaut, être transparent. Autrement dit, un objet client qui invoque un autre objet serveur ne doit pas distinguer le cas où ce dernier est situé sur un autre processeur du cas où les deux objets sont situés sur le même processeur.

6.5.2. Accessibilité et tolérance aux fautes – Pour prendre en compte l'inaccessibilité des objets, le système Argus [LIS 83] permet d'associer des exceptions à chaque invocation. Si un objet est situé sur un processeur qui est inaccessible, à cause d'une faute du réseau de communication ou du processeur lui-même, l'exception est déclenchée. Cela permet par exemple d'invoquer un autre objet à la place. De manière complémentaire, pour assurer qu'une invocation qui n'a pu être exécutée ne conduit pas à des incohérences, on peut associer la notion de transaction à l'invocation synchrone. Autrement dit, si l'invocation échoue (par exemple si l'objet *serveur* invoqué devient inaccessible), les effets de l'invocation sont annulés. Etendant cette approche, le système Karos [GUE 92] permet d'associer les transactions également aux invocations asynchrones.

6.5.3. Migration – Afin d'assurer une plus grande accessibilité des objets, certains systèmes offrent des mécanismes de migration. Dans le langage Emerald [JUL 94], et la couche d'exécution générique COOL [LEA 93], le programmeur d'une application répartie peut décider qu'à un certain moment un objet doit migrer d'un processeur à un autre processeur. Le programmeur peut aussi contrôler (sous forme d'« attachements » en Emerald) quels autres objets liés doivent également migrer avec lui. L'objectif est en définitive de placer sur un même processeur les objets qui communiquent très souvent entre eux. Cela permet d'alléger les goulots d'étranglement et de minimiser les communications distantes. Certains systèmes réalisent la migration de manière automatique (sans l'intervention du programmeur), pour assurer un rééquilibrage de la charge du système.

6.5.4. Duplication – La duplication est un autre mécanisme de gestion de la répartition des objets. Une première motivation, comme pour la migration, est d'offrir une plus grande accessibilité. Ainsi un objet, sujet de beaucoup d'invocations à partir de différents clients distants, gagnera à être dupliqué sur les différents processeurs clients (ce premier mécanisme est analogue à un « cache » local). Une deuxième motivation de la duplication est la tolérance aux fautes. Lorsqu'un objet existe en plusieurs copies sur des processeurs différents, il peut tolérer les fautes de certains de ces processeurs. Dans les deux cas, se pose le problème de la cohérence des différentes copies d'un même objet (c'est-à-dire la garantie qu'elles possèdent toutes les mêmes valeurs). Afin d'assurer une telle cohérence, dans le système Electra [MAF 95] par exemple, la notion d'invocation distante a été étendue. Dans ce cas, une invocation de l'objet entraîne l'invocation de toutes les copies, et des invocations concurrentes sont exécutées dans le même ordre total par toutes les copies. Andrew Black a également présenté, dans [BLA 93], un mécanisme général d'invocation de groupe qui s'applique très bien au cas où un objet est dupliqué.

6.5.5. Tendances et standardisation – Devant la multitude de systèmes à objets répartis, deux groupes de standardisation sont apparus: le groupe autour de ODP (Open Distributed Programming), et le groupe OMG (Object Management Group) (tous deux décrits dans [NIC 93]). Alors que le premier, principalement un groupe de réflexion, est constitué de membres d'instituts de recherches qui définissent un modèle abstrait d'objet réparti, le second, composé d'industriels, a défini un modèle de mise en œuvre de système à objets répartis intitulé CORBA (Common Architecture for an Object Request Broker). Le standard CORBA a pour objectif de faire communiquer des objets situés sur des plates-formes différentes, grâce à l'utilisation d'un même langage de définition d'interfaces nommé *IDL (Interface Definition Language).*

6.6. Limites de l'approche intégrée

L'approche intégrée procède par unification des mécanismes objet avec les mécanismes du parallélisme et de la répartition. Cependant, certains conflits et limitations peuvent survenir entre ces différents mécanismes. Particulièrement significatif est le cas du mécanisme d'héritage que nous allons décrire ici. Un premier conflit (§ 6.6.1) se manifeste entre l'utilisation traditionnelle de l'héritage (pour réutiliser variables et méthodes) et son application à la spécialisation de la synchronisation. Un deuxième conflit (§ 6.6.2) est relatif à la notion de duplication qui, si elle est appliquée telle quelle aux objets, peut provoquer des duplications non désirées de certaines invocations. Un troisième conflit (§ 6.6.3) concerne l'intégration de protocoles transactionnels incompatibles. Enfin, un quatrième conflit (§ 6.6.4) provient cette fois des hypothèses fortes (mémoire centralisée) des techniques traditionnelles de mise en œuvre des mécanismes de factorisation (classes et héritage), ce qui limite de fait leur transposition immédiate à un univers réparti.

6.6.1. Spécialisation de la synchronisation – Le mécanisme d'héritage est un des grands atouts de la programmation par objets pour la réutilisation et la spécialisation des programmes. Il est en conséquence naturel de vouloir utiliser l'héritage pour spécialiser les spécifications de synchronisation exprimées dans une classe d'objets. Cependant, l'expérience montre tout d'abord que la synchronisation est une dimension plus complexe à exprimer que la structure (variables d'instance), ou que les fonctionnalités (méthodes), d'une classe. Elle est ainsi beaucoup plus difficile à spécialiser/hériter, du fait de l'interdépendance des conditions de synchronisation. De plus les différentes utilisations de l'héritage (pour hériter des variables, des méthodes, et des synchronisations), peuvent entrer en conflit, comme le remarque [McH 94]. Il se peut ainsi que dans certains cas, la définition d'une sous-classe, rajoutant une seule méthode, impose la redéfinition de l'ensemble des spécifications de synchronisation. Cette limitation de la réutilisabilité des spécifications de synchronisation par l'héritage a été nommée par Satoshi Matsuoka « the *inheritance anomaly phenomenon* » (voir [Mat 93] et [McH 94] pour deux études complètes comprenant des exemples détaillés).

Les spécifications selon les formalismes centralisés explicites (voir au § 6.4.4) s'avèrent très difficiles à réutiliser. En pratique il faut trop souvent les redéfinir entièrement. Les formalismes décentralisés, intrinsèquement plus modulaires, sont eux beaucoup plus adaptés à une spécialisation sélective. Cependant, cette décomposition extrême au niveau de chaque méthode est aussi une arme à double tranchant. L'essence du problème réside dans le fait que les spécifications de synchronisation, même décomposées au niveau de chaque méthode, restent plus ou moins interdépendantes. Ainsi, par exemple, dans le cas d'une synchronisation intra-objet avec des gardes, le rajout dans une sous-classe d'une nouvelle méthode d'écriture va imposer la spécialisation de toutes les autres gardes pour que chacune prenne en compte cette nouvelle exclusion mutuelle.

Les derniers développements des travaux dans le domaine montrent : l'intérêt de spécifier et spécialiser indépendamment des synchronisations comportementale et intra-objet [Tho 92], la possibilité d'offrir le choix entre plusieurs formalismes [Mat 93], et enfin les promesses de la généricité (en instanciant des spécifications suffisamment abstraites et génériques), plutôt que l'héritage, comme outil de réutilisation [McH 94].

6.6.2. Duplication d'invocations – Les protocoles de communication définis pour gérer la duplication des services dans un système réparti tolérant aux fautes (voir au § 6.5.4) considèrent un modèle client/serveur simple.

L'application des mêmes protocoles aux objets pose le problème de la duplication des invocations. En effet, un objet agit généralement, à la fois comme un client, et comme un serveur. Autrement dit, un objet dupliqué en tant que serveur, peut cependant à son tour invoquer d'autres objets (cette fois en tant que client). Toutes les copies de l'objet se retrouveront en conséquence à invoquer ces autres objets plusieurs fois. Le résultat est la duplication inutile des invocations. Cette duplication peut conduire, au meilleur des cas à l'inefficacité du système, et au pire des cas à des incohérences. Une même opération (par exemple d'incrémentation) peut ainsi être exécutée plusieurs fois plutôt qu'une. Ce problème est discuté dans [Maz 95] et des solutions (appelées « pre-filtering » et « post-filtering ») sont proposées. Le « pre-filtering » consiste à coordonner les traitements effectués par les copies d'un objet client, afin de ne générer qu'une invocation. Le « post-filtering » est l'opération duale, qui assure la coordination au niveau des copies du serveur, pour ne pas traiter des invocations redondantes.

6.6.3. Compatibilité entre protocoles transactionnels – Il est tentant d'intégrer le protocole de contrôle de la concurrence transactionnelle au niveau des objets. Ainsi on peut définir localement, pour un objet donné, le contrôle de concurrence adéquat qui permet un contrôle optimal. Par exemple, la prise en compte de la commutativité des opérations permet d'entrelacer (sans bloquer) des transactions pour un objet donné. Malheureusement, le gain apporté en matière de modularité et de spécialisation peut amener à des problèmes d'incompatibilité [Wei 89]. Si les objets utilisent différents protocoles de sérialisation des transactions (c'est-à-dire, ordonnancent les transactions suivant des ordres différents), les exécutions globales des transactions peuvent être incohérentes (c'est-à-dire non sérialisables). Une approche proposée pour résoudre le problème est basée sur des conditions locales à garantir par les objets, afin d'assurer la compatibilité des différents protocoles [Gue 95c].

6.6.4. Mise en œuvre des factorisations (héritage et variables globales) – Ce dernier exemple de limitation est plus général (c'est-à-dire moins spécifique à l'approche intégrée). Il provient du fait que les stratégies de mise en œuvre des mécanismes de factorisation (classes et héritage) ont souvent été fondées sur des hypothèses fortes d'informatique traditionnelle, c'est-à-dire séquentialité de l'exécution des programmes et mémoire centralisée. Elles peuvent ainsi atteindre leurs limites une fois transposées directement dans l'univers de la programmation parallèle et répartie.

L'utilisation des variables de classes, au sens Smalltalk-80, pose un problème dans un système réparti. Il semble difficile, à moins d'introduire des mécanismes transactionnels complexes, de garantir que la mise à jour d'une variable de classe soit immédiatement reflétée sur toutes les instances d'une classe, lorsque celles-ci sont situées sur plusieurs processeurs. Ce problème est en fait lié aux variables partagées en général.

De manière plus générale encore, la mise en œuvre de l'héritage dans un système réparti, pose le problème de l'accès distant au code des superclasses, à moins que celles ci ne soient dupliquées sur tous les processeurs avec le coût conséquent. Une méthode de répartition des bibliothèques de classes en modules autonomes est proposée dans [GRA 95]. Elle permet de gérer de manière abstraite la répartition, non pas seulement des instances, mais aussi des classes (c'est-à-dire du code) et donc de minimiser sa duplication.

Une approche extrême, pour remplacer le mécanisme d'héritage entre classes, est le concept de *délégation* (historiquement introduit dans le langage d'acteurs Act 1 [LIE 87]). Intuitivement, un objet qui ne peut lui-même interpréter un message, le délèguera à un mandataire[8]. Ce dernier le traitera donc à sa place ou bien le délèguera lui-même à nouveau. Cette alternative à l'héritage est très séduisante[9], car elle ne repose que sur la transmission de messages, et donc est apte à être répartie. Cependant, elle nécessite un mécanisme non trivial de synchronisation pour assurer un ordonnancement correct des messages récursifs (avant les autres messages). De ce fait la délégation ne peut offrir une solution générale et complète comme alternative à l'héritage [BRI 87].

6.7. Bilan de l'approche intégrée

En résumé, l'approche intégrée est extrêmement séduisante par la fusion qu'elle propose des concepts et mécanismes, d'une part de la programmation par objets, et d'autre part de la programmation parallèle et répartie. Elle offre ainsi au programmeur un nombre minimal de concepts et un cadre unique de vision de ces différents aspects. Cependant, comme nous l'avons vu (§ 6.6), elle souffre de limitations dans certains secteurs d'intégration.

Un autre danger potentiel est qu'une unification/intégration trop systématique peut aboutir à un modèle trop réducteur (trop d'uniformité nuit à la variété !) et posant des problèmes d'efficacité. Ainsi par exemple, dans les langages d'acteurs, tout objet est un objet actif. Or, les technologies actuelles ne permettent pas toujours de gérer efficacement la prolifération des processus qui peuvent résulter d'un tel modèle [CAP 92]. On peut également citer les systèmes Argus [LIS 83] et Karos [GUE 92] qui, en associant systématiquement des transactions aux invocations (transmissions de message), peuvent pénaliser les performances d'une application répartie dans laquelle cette association ne se justifie pas (par exemple une application de travail coopératif).

Une dernière limitation, et non la moindre, tient aux capacités parfois insuffisantes de réutiliser les programmes séquentiels déjà existants, hormis en les encapsulant dans des objets actifs. Une approche pragmatique consiste aussi

8. On peut noter que, de manière à traiter correctement la récursion, ceci impose d'inclure le receveur initial dans le message ainsi délégué.

9. Les avantages comparés de l'héritage et de la délégation, dans un contexte de programmation séquentielle, ont d'ailleurs fait l'objet d'un grand débat au cours de la deuxième moitié des années 80 [STE 89].

à considérer une cohabitation raisonnable (différentes classes) entre les objets actifs et les autres, appelés alors *objets passifs*. Cette approche, alors moins homogène, impose des règles méthodologiques pour la distinction entre objets actifs et objets passifs [CAR 93].

7. Approche réflexive

7.1. Vers une approche par combinaison

Comme nous l'avons vu précédemment, l'approche applicative (par bibliothèques) aide à la structuration des abstractions et mécanismes nécessaires à la programmation parallèle et répartie, grâce aux concepts d'encapsulation, de généricité, de classe, et d'héritage. L'approche intégrée, quant à elle, apporte une minimisation du nombre de concepts à disposition du programmeur et une meilleure transparence des mécanismes. Cependant, elle a tendance à restreindre la flexibilité et l'efficacité des mécanismes offerts. En effet, les langages et systèmes bâtis à partir de bibliothèques, se retrouvent en général plus extensibles que nombre de langages conçus selon l'approche intégrée. Ceci, parce que les bibliothèques peuvent construire et simuler, et donc assurent une plus grande flexibilité, alors que les nouveaux langages peuvent fixer trop tôt leurs modèles de calcul et de communication. L'idéal serait donc d'arriver à conjuguer les avantages des deux approches, autrement dit : la simplification de l'approche intégrée et la flexibilité de l'approche applicative.

Il faut signaler que l'approche applicative et l'approche intégrée sont en fait destinées à *différents* niveaux d'utilisation. L'approche intégrée est plus particulièrement destinée au programmeur d'applications, et lui offre un cadre conceptuel unique et simplifié. L'approche applicative est plus particulièrement destinée au concepteur de systèmes, ou encore à l'utilisateur averti qui souhaite les spécialiser, et leur offre une méthodologie de structuration, sous forme de bibliothèques de composants et de protocoles.

En conséquence de quoi, et contrairement à ce qui aurait pu sembler au premier abord, l'approche *applicative* et l'approche *intégrée* ne sont *pas* en concurrence, mais sont bien au contraire *complémentaires*. La question qui se pose alors est la suivante : « Comment peut-on combiner au mieux ces deux approches ? », et pour être plus précis : « Comment les interfacer ? ». Il se trouve qu'une méthodologie générale d'adaptation du comportement de systèmes informatiques, appelée *réflexion*, offre un tel type d'articulation.

7.2. Réflexion

La *réflexion* est une méthodologie générale pour décrire, contrôler, et adapter le comportement d'un système informatique quelconque. L'idée de base est de doter le système d'une représentation d'un certain nombre de caractéristiques de son propre comportement, d'où le nom donné de « *réflexion* » (« *reflection* » en anglais). Ainsi diverses caractéristiques de représentation (statique) et d'exécution (dynamique) des programmes sont rendues concrètes, également sous la forme d'un (ou plusieurs) programme(s), appelés *méta-programmes*. Ils représentent donc le comportement par défaut (interprète, compilateur, moniteur d'exécution) du système ou langage. La spécialisation de méta-programmes permettra ainsi de particulariser l'exécution d'un programme utilisateur, en changeant les choix de représentation mémoire, le contexte de calcul, la nature des mécanismes et des protocoles, etc. Notez que le *même* langage est employé, pour l'écriture des programmes, comme pour leur contrôle. Par contre, la *séparation* entre programme utilisateur (appelé *niveau objet*) et description/contrôle (*niveau méta*) est, elle, clairement et complètement établie.

La réflexion permet de décorréler les bibliothèques de méta-programmes, spécifiant les caractéristiques d'exécution (gestion de la concurrence, de la répartition, des ressources), du programme de l'application. Ceci augmente en conséquence modularité, réutilisabilité et lisibilité des programmes. Enfin, la réflexion offre une méthodologie pour « ouvrir » et rendre adaptable, via une *méta-interface*[10], les choix de mise en œuvre et de gestion des ressources. Ces derniers sont en effet trop souvent, sinon pré-cablés et fixés, du moins non modifiables au niveau du langage de programmation lui-même, car délégués au système d'exploitation sous-jacent.

En résumé, la réflexion permet d'intégrer intimement des bibliothèques de protocoles avec un langage ou un système, offrant ainsi un cadre d'interface entre les approches et niveaux applicatifs et intégrés.

7.3. Réflexion et objets

La réflexion s'exprime particulièment bien dans les modèles à objets qui assurent une bonne encapsulation des niveaux ainsi que la modularité des effets. Il est alors naturel d'organiser (décomposer) le contrôle du comportement d'un système informatique à objets, en un ensemble d'objets. Une telle organisation est appelée un « *Meta-Object Protocol (MOP)* » [Kic 91], et ses composants sont appelés *méta-objets* [Mae 87], puisque les différents méta-programmes sont représentés par des objets. Ils peuvent représenter certaines caractéristiques du contexte d'exécution d'un objet, telles que : représentation, mise en œuvre, exécution, communication, et emplacement. La spécialisation de méta-objets permet ainsi d'étendre, et de modifier, localement, le contexte d'exécution d'un ou de plusieurs objets du programme.

La réflexion aide également à exprimer et adapter la gestion des ressources, non seulement au niveau d'un objet individuel, mais également à un niveau plus large, tel que : séquenceur, processeur, espace de noms, groupe d'objets, etc. Ces

10. Cette *méta-interface* permet au programmeur client d'adapter et d'optimiser le *comportement* d'un module logiciel, indépendamment de ses *fonctionnalités*, qui restent accessibles via l'interface *standard* (« *base-interface* »). Ceci a été nommé par Gregor Kiczales le concept d'« *open implementation* » [Kic 94], qui peut être traduit par « mise en œuvre ouverte ».

ressources physiques ou logicielles sont alors représentées par des méta-objets au niveau du langage ou système. Ceci permet ainsi l'expression de politiques fines (en particulier pour l'équilibre de charges et le séquencement) avec toute la puissance algorithmique d'un langage de programmation, par opposition à des algorithmes globaux et câblés, habituellement optimisés pour un type d'application.

7.4. Degrés de réflexivité

L'architecture CodA [McA 95] est un exemple représentatif d'architecture réflexive générale (autrement dit un « MOP ») à *méta-composants*[11]. CodA considère par défaut sept méta-composants associés à chaque objet, et correspondant à : l'*envoi de message*, leur *réception*, le *stockage des messages reçus*, leur *sélection*, la *recherche de méthode*, l'*exécution*, et enfin l'*accès à l'état* de l'objet. Un objet doté des méta-composants par défaut se comporte comme un objet standard (c'est-à-dire séquentiel et passif)[12]. L'utilisation de méta-composants particuliers, permet de particulariser sélectivement tel ou tel aspect du modèle de représentation ou d'exécution de l'objet. L'interface entre les méta-composants est clairement exprimée de façon à composer, de manière relativement libre, des méta-composants de diverses origines.

Notez que d'autres architectures réflexives peuvent être plus spécialisées, et proposent, quant à elles, un nombre plus réduit de composants (également plus abstraits). Ainsi, la plate-forme Actalk [Bri 94] est spécialisée pour la programmation concurrente par objets selon l'approche intégrée. Elle classifie différents modèles (méta-composants) : (1) d'*activité* (acceptation implicite ou explicite des requêtes, concurrence intra-objet, etc.) et de *synchronisation* (comportements abstraits, gardes, compteurs de synchronisation, etc.) [Bri 96b], (2) de *communication* (synchrone, asynchrone, avec réponse anticipée, etc.). et (3) d'*invocation* (avec estampillage temporel, avec priorités, etc.). Actalk permet d'associer, relativement librement, ces différents modèles à des classes d'objets/programmes et ainsi de faire varier leurs paramètres de concurrence et de communication.

La plate-forme GARF [Gar 94, Gar 95] est, quant à elle, spécialisée pour la programmation répartie et tolérante aux fautes. Elle classifie différents modèles : (1) de gestion d'objet (persistant, dupliqué, etc.), et (2) de communication (un seul destinataire, « multicast », atomique, etc.). Ces deux dimensions semblent s'avérer en pratique suffisantes pour traiter une large gamme de problèmes d'applications réparties et tolérantes aux fautes.

De manière plus générale, selon les objectifs visés, ainsi que l'équilibre recherché entre flexibilité, généralité, simplicité et efficacité, un langage, ou

11. Par la suite, nous emploierons le terme *méta-composant*, ou encore *composant*, plutôt que le terme *méta-objet*, de manière à souligner les principes d'interchangeabilité entre composants d'une architecture réflexive telle que CodA.

12. pour être plus précis, comme un objet standard Smalltalk, car CodA est actuellement mis en œuvre en Smalltalk.

système, sera plus ou moins *réflexif*. Tout dépend en effet de la quantité et la portée des mécanismes qui seront ainsi « remontés » au méta-niveau. Ainsi, certains mécanismes peuvent s'exprimer sous forme de *méthodes réflexives*, sans faire intervenir un méta-objet explicite et complet.

Le langage Smalltalk-80 est un exemple très représentatif de cette dernière catégorie. En plus de la méta-représentation des *structures* du programme et de son exécution (les classes, méthodes, messages, contextes, processus, etc., sont des objets à part entière, voir au § 5.2), quelques mécanismes réflexifs très puissants permettent également, un certain contrôle de l'*exécution* (redéfinition de la transmission de messages en cas d'erreur, référence au contexte courant, échange de références, etc.). Ces facilités permettent la construction aisée, et l'excellente intégration [BRI 96a], de nombreuses plates-formes de programmation concurrente, parallèle et répartie, telles que Simtalk, Actalk, GARF et CodA.

7.5. Exemples d'applications

Nous allons maintenant rapidement illustrer, pour une architecture réflexive donnée : CodA [13], l'introduction transparente de divers modèles de parallélisme et de répartition. Notez que, dans le cas de CodA, ainsi que pour la plupart des autres architectures réflexives décrites dans cet article, le modèle de programmation de base est *intégré*, tandis que la réflexion permet une spécialisation via différentes *bibliothèques* de méta-composants.

7.5.1. Modèle d'exécution concurrente – De manière à introduire la concurrence au niveau d'un objet donné (en le rendant *actif*, selon les principes de l'approche intégrée), deux méta-composants sont spécialisés. Un composant spécialisé de *stockage des messages reçus*[14] est une file d'attente (« queue » de type FIFO) qui stockera les messages selon leur ordre d'arrivée. Un composant spécialisé d'*exécution* associe une activité indépendante (mise en œuvre par un processus Smalltalk) à l'objet. Ce processus exécute une boucle infinie de sélection et de traitement du premier message retiré du composant de *stockage des messages reçus*.

7.5.2. Modèle d'exécution répartie – De manière à introduire la répartition, un nouveau méta-composant est *rajouté*, pour *encoder* (« *marshal* ») les messages destinés à des objets distants. De plus, deux nouveaux objets spécifiques sont ajoutés, qui représentent les notions de *référence distante* (vers un objet distant) et d'*espace mémoire et de nom*. L'objet référence distante possède un composant spécialisé de *réception de message*, qui a la responsabilité d'encoder

13. Voir [MCA 95] pour une description plus détaillée de son architecture et de ses bibliothèques de composants.
14. Le composant par défaut se contente de passer immédiatement chaque nouveau message reçu, directement au composant d'*exécution*.

le contenu (arguments) du message en une suite d'octets, et de l'envoyer à travers le réseau jusqu'à l'objet distant. Ce dernier possède un composant spécialisé de *réception de message*, qui reconstruit et réceptionne le message. Les choix d'encodage, tels que : quel argument doit être passé par référence, par valeur (c'est-à-dire copié), jusqu'à quel niveau de profondeur, etc., peuvent être spécialisés par un *descripteur d'encodage*, détenu par le composant d'*encodage*.

7.5.3. Protocoles de migration et de duplication – La migration est introduite à travers un nouveau méta-composant qui décrit les protocoles (*comment*) et les politiques (*quand*, voir au 7.6.2) de migration. La duplication, mécanisme dual, est gérée par deux nouveaux méta-composants supplémentaires. Le premier contrôle l'accès à l'état de l'objet original. Le second contrôle l'accès à l'état d'une copie. A nouveau, les choix d'encodage, tels que : quel argument doit être passé par référence, par valeur, par « déplacement » (« *call-by-move* », c'est-à-dire migré, comme dans le langage Emerald [JUL 94]), avec attachements, etc., peuvent être spécialisés par un *descripteur d'encodage*, détenu par le composant de *duplication* de l'objet original. Le descripteur permet également de spécialiser les caractéristiques suivantes : quelles parties de l'objet doivent être dupliquées (permettant ainsi une duplication sélective des seules parties critiques d'un objet donné), et diverses politiques de maintien de la consistance entre l'original et ses copies.

7.6. (Quelques) autres exemples d'architectures réflexives

7.6.1. Installation dynamique et composition de protocoles – La méthodologie générale MAUD [AGH 93] se concentre sur la tolérance aux fautes. Elle offre un cadre pour une *installation dynamique* et la *composition* de méta-composants spécialisés pour des protocoles, tels que duplication et transactions. Trois méta-composants (envoi des messages, réception, et état) sont considérés dans MAUD. La possibilité d'associer les méta-composants (méta-objets), non seulement à des objets, mais aussi à des méta-composants (qui sont des objets de plein droit), ouvre la voie de la composition de protocoles. Le principe est le suivant : on associe un couple de méta-composants d'envoi, et de réception de messages, respectivement à un autre couple de méta-composants. On assure ainsi une composition des protocoles par couches successives. De plus, l'association dynamique de méta-composants permet une installation uniquement au cours des besoins et pendant l'exécution du programme.

7.6.2. Contrôle de la migration – L'avantage des objets, et plus encore des objets actifs, est qu'ils sont autonomes, donc plus aisés à faire migrer d'« une seule pièce ». Néanmoins, la décision éventuelle de migrer un objet est un point d'importance et qui reste souvent la responsabilité du programmeur (par exemple en Emerald [JUL 94]). Il peut être ainsi intéressant de semi-automatiser cette décision, en suivant diverses considérations, telles que les charges des différents processeurs. La réflexion permet d'intégrer de telles informations statistiques (résidentes pour les ressources physiques et partagées, telles que les charges des processeurs, ou déductibles au niveau des méta-composants pour les informations locales à l'objet, tel que le pourcentage de communications distantes), et de s'en servir pour développer divers algorithmes de migration décrits au méta-niveau dans le langage. Des exemples de tels contrôles avec toute la puissance algorithmique du langage, et indépendants du programme de l'application, sont décrits dans [OKA 94].

7.6.3. Spécialisation de politiques système – Le système d'exploitation réparti Apertos [YOK 92] représente un exemple significatif de système d'exploitation complètement conçu selon une architecture réflexive (à objets). En plus de la modularité et de la généricité apportées par l'utilisation d'une approche applicative objet (comme pour le système d'exploitation Choices, déjà décrit au § 5.4), la réflexion permet la *spécialisation* (éventuellement dynamique) du système pour les besoins d'un type donné d'application. Ainsi en Apertos, il est relativement aisé de spécialiser la politique de séquencement, par exemple pour introduire des contraintes de type temps-réel. Un gain supplémentaire est dans la taille du noyau du système, qui est particulièrement minimal, étant réduit à la mise en œuvre des opérations réflexives de base et les abstractions de ressources de base. Ce dernier point aide donc à la compréhension et au portage du système.

7.6.4. Extension réflexive d'un système commercial existant – Une méthodologie réflexive a récemment été utilisée pour incorporer des protocoles transactionnels étendus[15] dans un système transactionnel commercial *déjà existant* [BAR 95]. L'idée est d'étendre le moniteur transactionnel standard pour rendre visibles un certain nombre de caractéristiques, telles que : délégation de verrou, identification de dépendances entre transactions, définition de conflits, et de les représenter par des opérations réflexives. La mise en œuvre, minimale et modulaire, repose sur l'utilisation d'« appels montants » (« *upcalls* »). Il est alors possible de mettre en œuvre divers protocoles transactionnels étendus, tels que : « split/join », « cooperative groups » [BAR 95], à partir des opérations réflexives.

7.6.5. Le « run-time » générique comme approche duale – Notons également que, de manière plus générale, nous assistons à un rapprochement mutuel entre langages de programmation et systèmes d'exploitation. Les langages de programmation tendent à avoir une représentation de plus en plus étendue, et de haut niveau (réflexion), du modèle d'exécution sous-jacent. De manière duale, plusieurs systèmes d'exploitation répartis offrent maintenant des couches

15. c'est-à-dire, relâchant certaines des propriétés standard (« ACID » : atomicité, consistance, isolation, durabilité) des transactions.

logicielles d'exécution générique (« generic run time »), par exemple COOL [Lea 93]. Ces couches génériques sont conçues pour être utilisées par divers langages, à l'aide d'un ensemble d'appels montants (up-calls), déléguant les représentations ou fonctions spécifiques au langage de programmation.

Enfin, dans un même ordre d'idée d'interfaces entre niveaux, la réflexion a également été proposée pour lier les différents niveaux (d'objet à agent) d'un système multi-agents [Fer 91].

7.7. Bilan de l'approche réflexive

La réflexion offre un cadre méthodologique pour la spécialisation de modèles d'exécution parallèle et répartie, par spécialisation et *intégration* de (méta)-*bibliothèques* intimement avec le langage ou le système, tout en les décorrélant des programmes d'applications.

De nombreuses architectures réflexives sont actuellement proposées et évaluées. Nous en avons évoqué ici plusieurs, à travers diverses gammes d'applications. Il est encore trop tôt pour dégager, et valider, une architecture réflexive générale et optimale pour la programmation parallèle et répartie (bien que CodA [McA 95] nous semble un pas intéressant dans cette direction). Il nous faut pouvoir compléter notre expérience dans différents domaines d'applications, pour être mieux à même d'identifier les équilibres à trouver entre la flexibilité requise, la complexité de l'architecture, et l'efficacité résultante.

Une première critique de la réflexion porte sur la relative complexité des architectures réflexives. Mais, et indépendamment du temps d'apprentissage nécessaire pour une nouvelle culture, cela est en partie lié aux gains qu'elles offrent en matière de flexibilité. Un problème fondamental[16] tient à la capacité de composition arbitraire de méta-composants venant d'origines diverses, mais qui peuvent contrôler des aspects ou ressources qui se recoupent. Enfin, un problème concret tient à l'efficacité, du fait des indirections et interprétations supplémentaires que la réflexion peut entraîner. Deux approches opposées pour réduire ces surcoûts tiennent en : la réduction de la portée de la réflexion au niveau de la compilation, par exemple pour l'architecture OpenC++ [Chi 95], ou bien l'utilisation de techniques de transformation de programmes, et en particulier l'évaluation partielle [Mas 95], pour réduire au maximum l'interprétation.

8. Conclusion

La programmation par objets est, très certainement, un des meilleurs vecteurs actuels pour le développement de systèmes et applications

16. Ce problème général, de composition arbitraire de modules logiciels, dépasse d'ailleurs de loin le seul cadre de la réflexion. La réflexion offre cependant un cadre intéressant, pour aborder en particulier la notion de modules logiciels évolutifs, qui adaptent eux-mêmes leur interface à un nouvel environnement logiciel [Kis 95].

informatiques parallèles et réparties. Nous avons identifié et étudié trois approches qui se révèlent complémentaires.

L'approche *applicative* (par *bibliothèques*) aide à la structuration des systèmes parallèles et répartis, à l'aide des concepts objets. L'approche *intégrée* offre au programmeur un cadre conceptuel simplifié, de par l'unification qu'elle réalise entre concepts objets et concepts de la programmation parallèle et répartie. Enfin, l'approche *réflexive* offre un cadre conceptuel pour intégrer intimement des bibliothèques de protocoles avec un langage ou un système. La réflexion permet de spécialiser et d'adapter le modèle d'exécution (parallèle, réparti, tolérant aux fautes, temps-réel, etc.) d'une application, avec un minimum de modifications pour le programme de l'application.

Les développements respectifs de ces trois approches ont des objectifs complémentaires. L'approche *applicative* est destinée aux concepteurs de systèmes et vise à identifier les abstractions fondamentales des systèmes informatiques parallèles et répartis. Elle est plus particulièrement destinée aux concepteurs de systèmes. Sa principale limitation est que la programmation d'une application et de l'architecture parallèle et répartie sous-jacente sont représentées par des concepts et objets distincts. L'approche *intégrée* est destinée aux concepteurs d'applications, et vise à la définition d'un langage de programmation de haut niveau comportant un nombre minimal de concepts. Sa principale limitation tient en la possible réduction de flexibilité et d'efficacité qu'elle entraîne. L'approche *réflexive* est destinée aux concepteurs d'application qui souhaitent spécialiser le système pour les besoins propres à leur type d'application, et de manière duale aux concepteurs de système qui souhaitent ainsi concevoir leur système selon une telle architecture « ouverte » et adaptable. Elle apporte ainsi une articulation entre (et donc *ne remplace pas*) les approches applicative et intégrée, ainsi que leurs niveaux respectifs d'intervention.

Remerciements

La présente version de cet article a bénéficié de diverses suggestions faites par les relecteurs.

9. Bibliographie

[ACC 86] M. ACCETTA, R. BARON, W. BOLOSKY, D. GOLUB, E. RASHID, A. TEVANIAN, et M. YOUNG, « Mach: A New Kernel Foundation for Unix Development », *Summer Usenix Conference*, Atlanta GA, Etats-Unis, 1986.

[AGH 86] G. AGHA, *Actors: A Model of Concurrent Computation in Distributed Systems*, Series in Artificial Intelligence, MIT Press, 1986.

[AGH 93] G. AGHA, S. FRØLUND, R. PANWAR, et D. STURMAN, « A Linguistic Framework for Dynamic Composition of Dependability Protocols », *Dependable Computing for Critical Applications III (DCCA-3)*, IFIP Transactions, Elsevier, 1993, p. 197–207.

[AME 87] P. AMERICA, « POOL-T: A Parallel Object-Oriented Language », [YON 87], p. 199–220.

[BAL 94] R. BALTER, S. LACOURTE, et M. RIVEILL, « The Guide Language », *The Computer Journal*, Special Issue on Distributed Operating Systems, Vol. 37, n° 6, CEPIS - Oxford University Press, 1994, p. 519–530.

[BAR 95] R. BARGA et C. PU, « A Practical and Modular Implementation of Extended Transaction Models », Technical Report, n° 95-004, CSE, Oregon Graduate Institute of Science & Technology, Portland OR, Etats-Unis, 1995.

[BER 87] P. BERNSTEIN, V. HADZILACOS, et N. GOODMAN, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[BER 92] T. BERNERS-LEE, R. CAILLIAU, J. GROFF, et B. POLLERMAN, « World Wide Web: the Information Universe », *Electronic Networking: Research, Applications and Policy*, Vol. 1, 1992.

[BÉZ 87] J. BÉZIVIN, « Some Experiments in Object-Oriented Simulation », *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, Vol. 22, n° 12, Décembre 1987, p. 394–405.

[BIR 73] G.M. BIRTWISTLE, O.-J. DAHL, B. MYHRHAUG, et K. NYGAARD, *Simula Begin*, Petrocelli Charter, 1973.

[BLA 91] A.P. BLACK, « Understanding Transactions in the Operating System Context », *Operating Systems Review*, Vol. 25, n° 28, Janvier 1991, p. 73–77.

[BLA 93] A.P. BLACK et M.P. IMMEL, « Encapsulating Plurality », *European Conference on Object Oriented Programming (ECOOP'93)*, édité par O. Nierstrasz, LNCS, n° 707, Springer-Verlag, Juillet 1993, p. 57–79.

[BOS 91] J. VAN DEN BOS et C. LAFFRA, « Procol: A Concurrent Object-Language with Protocols, Delegation and Persistence », *Acta Informatica*, n° 28, Septembre 1991, p. 511–538.

[BRI 87] J.-P. BRIOT et A. YONEZAWA, « Inheritance and Synchronization in Concurrent OOP », *European Conference on Object Oriented Programming (ECOOP'87)*, édité par J. Bézivin, J.-M. Hullot, P. Cointe et H. Lieberman, LNCS, n° 276, Springer-Verlag, Juin 1987, p. 32–40. (Egalement Chapitre 6 de [YON 90], p. 107–118).

[BRI 94] J.-P. BRIOT, « Modélisation et Classification de Langages de Programmation Concurrente à Objets : l'expérience Actalk », *Colloque Langages et Modèles à Objets (LMO'94)*, INRIA/IMAG/PRC-IA, Octobre 1994. (Egalement paru comme Rapport de Recherche LITP, n° 94/59, Institut Blaise Pascal, Paris, Octobre 1994).

[BRI 96a] J.-P. BRIOT et R. GUERRAOUI, « On the use of Smalltalk for Concurrent and Distributed Programming », *Informatik/Informatique* Journal, *Swiss Professional Informatics Societies*, Special Issue on Smalltalk (Languages, Tools, Environments), Février 1996.

[BRI 96b] J.-P. BRIOT, « An Experiment in Classification and Specialization of Synchronization Schemes », *Object Technologies for Advanced Software (ISOTAS'96)*, édité par K. Futatsugi et S. Matsuoka, LNCS, n° 1049, Springer-Verlag, Mars 1996, p. 227–249.

[BRI 96c] J.-P. BRIOT, J.-M. GEIB, et A. YONEZAWA (éditeurs), *Object-Based Parallel and Distributed Computation*, LNCS, Springer-Verlag, 1996.

[CAM 93] R. CAMPBELL, N. ISLAM, D. RAILA, et P. MADANY, « Designing and Implementing Choices: An Object-Oriented System in C++ », [MEY 93a], p. 117–126.

[CAP 92] R. CAPOBIANCHI, R. GUERRAOUI, A. LANUSSE, et P. ROUX, « Lessons from Implementing Active Objects on a Parallel Machine », *Usenix Symposium on Experiences with Distributed and Multiprocessor Systems*, 1992, p. 13–27.

[CAR 93] D. CAROMEL, « Towards a Method of Object-Oriented Concurrent Programming », [MEY 93a], p. 90–102.

[CHI 95] S. CHIBA, « A Metaobject Protocol for C++ », *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), Special Issue of Sigplan Notices*, Vol. 30, n° 10, Octobre 1995, p. 285–299.

[ELL 91] C.A. ELLIS, S.J. GIBBS, et G.L. REIN, « Groupware: Some Issues and Experiences », *Communications of the ACM (CACM)*, Vol. 34, n° 1, Janvier 1991, p. 39–58.

[FER 91] J. FERBER et P. CARLE, « Actors and Agents as Reflective Concurrent Objects: a Mering-IV Perspective », *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, n° 6, Novembre/Décembre 1991.

[GAR 94] B. GARBINATO, R. GUERRAOUI, et K.R. MAZOUNI, « Distributed Programming in GARF », [GUE 94], p. 225–239.

[GAR 95] B. GARBINATO, R. GUERRAOUI, et K.R. MAZOUNI, « Implementation of the GARF Replicated Objects Platform », *Distributed Systems Engineering Journal*, Février 1995, p. 14–27.

[GAM 95] E. GAMMA, R. HELM, R. JOHNSON et J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[GAS 92] L. GASSER et J.-P. BRIOT, « Object-Based Concurrent Programming and Distributed Artificial Intelligence », *Distributed Artificial Intelligence: Theory and Practice*, édité par N. Avouris et L. Gasser, Kluwer, 1992, p. 81–107.

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

[GRA 95] C. GRANSART, « BOX : un Modèle et un Langage à Objets pour la Programmation Parallèle et Distribuée », *Thèse d'Université*, LIFL, Université de Lille-I, Janvier 1995.

[GUE 92] R. GUERRAOUI, R. CAPOBIANCHI, A. LANUSSE, et P. ROUX, « Nesting Actions Through Asynchronous Message Passing: the ACS Protocol », *European Conference on Object Oriented Programming (ECOOP'92)*, édité par O. Lehrmann Madsen, LNCS, n° 615, Springer-Verlag, Juin 1992, p. 170–184.

[GUE 94] R. GUERRAOUI, O. NIERSTRASZ, et M. RIVEILL (éditeurs), *Object-Based Distributed Programming*, LNCS, n° 791, Springer-Verlag, 1994.

[GUE 95a] R. GUERRAOUI et A. SCHIPER, « Transaction Model vs Virtual Synchrony Model: Bridging the Gap », *Distributed Systems: From Theory to Practice*, édité par K. Birman, F. Cristian, F. Mattern, et A. Schiper, LNCS, n° 938, Springer-Verlag, 1995. p 121–132.

[GUE 95b] R. GUERRAOUI, « Les Langages Concurrents à Objets », *Technique et Science Informatiques (TSI)*, Vol. 14, n° 8, Octobre 1995, p. 945–972.

[GUE 95c] R. GUERRAOUI, « Modular Atomic Objects », *Theory and Practice of Object Systems (TAPOS)*, Vol. 1, n° 2, John Wiley & Sons, Novembre 1995, p. 89–100. Une version préliminaire est parue sous le titre « Atomic Object Composition », *European Conference on Object Oriented Programming (ECOOP'94)*, édité par M. Tokoro et R. Pareschi, LNCS, n° 821, Springer-Verlag, Juillet 1994, p. 118–138.

[JÉZ 93] J.-M. JÉZÉQUEL, « Transparent Parallelisation Through Reuse: Between a Compiler and a Library Approach », *European Conference on Object Oriented*

Programming (ECOOP'93), édité par O. Nierstrasz, LNCS, n° 707, Springer-Verlag, Juillet 1993, p. 384–405.

[Jul 94] E. Jul, « Separation of Distribution and Objects », [Gue 94], p. 47–55.

[Kar 93] M. Karaorman et J. Bruno, « Introducing Concurrency to a Sequential Language », [Mey 93a], p. 103–116.

[Kic 91] G. Kiczales, J. des Rivières, et D. Bobrow, *The Art of The Meta-Object Protocol*, MIT-Press, 1991.

[Kic 94] G. Kiczales (éditeur), « Foil For The Workshop On Open Implementation », "`http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94/foil/main.html`", Octobre 1994.

[Kis 95] Y. Kishimoto, N. Kotaka, et S. Honiden, « Adapting Object-Communication Methods Dynamically », *IEEE Software*, Vol. 12, n° 3, Mai 1995, p. 65–74.

[Lea 93] R. Lea, C. Jacquemot, et E. Pillevesse, « COOL: System Support for Distributed Programming », [Mey 93a], p. 37–47.

[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.

[Lie 87] H. Lieberman, « Concurrent Object-Oriented Programming in Act 1 », [Yon 87], p. 9–36.

[Lis 83] B. Liskov et R. Sheifler, « Guardians and Actions: Linguistic Support for Robust, Distributed Programs », *ACM Transactions on Programming Languages and Systems*, Vol. 5, n° 3, 1983.

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

[Maf 95] S. Maffeis, « Run-Time Support for Object-Oriented Distributed Programming », *Thèse d'Université*, Université de Zurich, Février 1995.

[Mas 95] H. Masuhara, S. Matsuoka, K. Asai, et A. Yonezawa, « Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation », *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), Special Issue of Sigplan Notices*, Vol. 30, n° 10, Octobre 1995, p. 300–315.

[Mat 93] S. Matsuoka et A. Yonezawa, « Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages », *Research Directions in Concurrent Object-Oriented Programming*, édité par G. Agha, P. Wegner, et A. Yonezawa, Mit Press, 1993, p. 107–150.

[Maz 95] K. Mazouni, B. Garbinato, et R. Guerraoui, « Building Reliable Client-Server Software Using Actively Replicated Objects », *Technology of Object-Oriented Languages and Systems (TOOLS-Europe'95)*, édité par I. Graham, B. Magnusson, B. Meyer, et J.-M Nerson, Prentice Hall, Mars 1995, p. 37–53.

[McA 95] J. McAffer, « Meta-Level Programming with CodA », *European Conference on Object Oriented Programming (ECOOP'95)*, édité par W. Olthoff, LNCS, n° 952, Springer-Verlag, Août 1995, p. 190–214.

[McH 94] C. McHale, « Synchronization in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance », *PhD Thesis*, Dept. of Computer Science, Trinity College, Dublin, Irlande, Octobre 1994.

[Mey 88] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[MEY 93a] B. MEYER (éditeur), « Concurrent Object-Oriented Programming », Special Issue, *Communications of the ACM (CACM)*, Vol. 36, n° 9, Septembre 1993.

[MEY 93b] B. MEYER, « Systematic Concurrent Object-Oriented Programming », [MEY 93a], p. 56–80.

[NIC 93] J. NICOL, T. WILKES, et F. MANOLA, « Object-Orientation in Heterogeneous Distributed Computing Systems », *IEEE Computer*, Vol. 26, n° 6, Juin 1993, p. 57–67.

[OKA 94] H. OKAMURA et Y. ISHIKAWA, « Object Location Control Using Meta-Level Programming », *European Conference on Object Oriented Programming (ECOOP'94)*, édité par M. Tokoro et R. Pareschi, LNCS, n° 821, Springer-Verlag, Juillet 1994, p. 299–319.

[PAR 88] G.D. PARRINGTON et S.K. SHRIVASTAVA, « Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems », *European Conference on Object Oriented Programming (ECOOP'88)*, édité par S. Gjessing et K. Nygaard, LNCS, n° 322, Springer-Verlag, 1988, p. 234–249.

[ROZ 92] M. ROZIER, « Chorus », *Usenix Symposium on Micro-Kernels and Other Kernel Architectures*, 1992, p. 27–28.

[SIM 87] B. SIMONS et A. SPECTOR (éditeurs), « Fault-Tolerant Distributed Computing », LNCS, n° 448, Springer-Verlag, 1987.

[STE 89] L.A. STEIN, H. LIEBERMAN, et D. UNGAR, « A Shared View of Sharing: the Treaty of Orlando », *Object-Oriented Concepts, Databases, and Applications*, édité par W. Kim et F.H. Lochovsky, ACM Press - Addison Wesley, 1989, p. 31–48.

[THO 92] L. THOMAS, « Extensibility and Reuse of Object-Oriented Synchronization Components », *International Conference on Parallel Languages and Environments (PARLE'92)*, LNCS, n° 605, Springer-Verlag, Juin 1992, p. 261–275.

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

[WEI 89] W. WEIHL, « Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types », *ACM Transactions on Programming Languages and Systems*, Vol. 11, n° 2, 1989.

[WIN 94] J. WING, « Decomposing and Recomposing Transaction Concepts », [GUE 94], p. 111–122.

[YOK 92] Y. YOKOTE, « The Apertos Reflective Operating System: The Concept and its Implementation », *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), Special Issue of Sigplan Notices*, Vol. 27, n° 10, Octobre 1992, p. 414–434.

[YON 87] A. YONEZAWA et M. TOKORO (éditeurs), *Object-Oriented Concurrent Programming*, Computer Systems Series, MIT Press, 1987.

[YON 90] A. YONEZAWA (éditeur), *ABCL: an Object-Oriented Concurrent System*, Computer System Series, Mit-Press, 1990.

Table des matières