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.

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 [Weihl, 1989]. 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 [Guerraoui, 1995b].

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, c'est-à-dire de variables partagées par tous les objets d'une même classe, 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 super-classes, 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 [Gransart, 1995]. 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 [Lieberman, 1987]). Intuitivement, un objet qui ne peut lui-même interpréter un message, le délèguera à un mandataire [10]. Ce dernier le traitera donc à sa place

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

ou bien le délèguera lui-même à nouveau. Cette alternative à l'héritage est très séduisante [11], 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 [Briot et Yonezawa, 1987].

6.6.7 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 (§ 6.6.6), 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 [Capobianchi *et al.*, 1992]. On peut également citer les systèmes ARGUS [Liskov et Scheifler, 1983] et KAROS [Guerraoui *et al.*, 1992] 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 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 [Caromel, 1993].

6.7 Approche réflexive

6.7.1 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 trans-

11. 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 [Stein *et al.*, 1989] (*chapitre 8*).

parence 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.

6.7.2 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* [12], 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.

6.7.3 Réflexion et objets

La réflexion s'exprime particulièrement 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)* » [Kiczales *et al.*, 1991]. Ses composants sont appelés *méta-objets* [Maes, 1987], 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 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.

6.7.4 Degrés de réflexivité

L'architecture CODA [McAffer, 1995] est un exemple représentatif d'architecture réflexive générale (autrement dit un « MOP ») à *méta-composants* [13]. 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

12. 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* » [Kiczales, 1994], qui peut être traduit par « mise en œuvre ouverte ».

13. Par la suite, nous emploierons le terme 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.

(c'est-à-dire séquentiel et passif)[14]. 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 [Briot, 1994] 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* (états abstraits, gardes, compteurs de synchronisation, etc.) [Briot, 1996], (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 [Garbinato *et al.*, 1994 ; Garbinato *et al.*, 1995] 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 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 § 6.5.2), 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 [Briot et Guerraoui, 1996], de nombreuses plates-formes de programmation concurrente, parallèle et répartie, telles que SIMTALK, ACTALK, GARF et CODA.

6.7.5 Exemples d'applications

Nous allons maintenant rapidement illustrer, pour une architecture réflexive donnée : CODA[15], l'introduction transparente de divers modèles de parallélisme et

14. Pour être plus précis, comme un objet standard SMALLTALK, car CODA est actuellement mis en œuvre en SMALLTALK.

15. Voir [McAffer, 1995] pour une description plus détaillée de son architecture et de ses bibliothèques de composants.

de répartition. Notez que, dans le cas de CODA, ainsi que pour la plupart des autres architectures réflexives décrites dans ce chapitre, le modèle de programmation de base est *intégré*, tandis que la réflexion permet une spécialisation via différentes *bibliothèques* de méta-composants.

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*[16] 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*.

Modèle d'exécution répartie

De manière à introduire la répartition, un nouveau méta-composant est *rajouté*, pour *encoder* (« *marshal* » ou encore « serialize ») 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 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*.

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 6.7.6) 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. À 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, 1994]), 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.

16. Le composant de stockage par défaut se contente de passer immédiatement chaque nouveau message reçu, directement au composant d'*exécution*.

6.7.6 (Quelques) autres exemples d'architectures réflexives

Installation dynamique et composition de protocoles

L'architecture réflexive MAUD [Agha *et al.*, 1993] 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.

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, 1994]). 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 [Okamura et Ishikawa, 1994].

Spécialisation de politiques système

Le système d'exploitation réparti APERTOS [Yokote, 1992] 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 § 6.5.4), 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.

Extension réflexive d'un système commercial existant

Une méthodologie réflexive a récemment été utilisée pour incorporer des protocoles transactionnels étendus[17] dans un système transactionnel commercial *déjà existant* [Barga et Pu, 1995]. 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 » [Barga et Pu, 1995], à partir des opérations réflexives.

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 logicielles d'exécution générique (« generic run time »), par exemple COOL [Lea *et al.*, 1993]. Ces couches génériques sont conçues pour être utilisées par divers langages, à l'aide d'un ensemble d'appels montants (upcalls), 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-agent [Ferber et Carle, 1991].

6.7.7 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 [McAffer, 1995] 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

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

de flexibilité. Un problème fondamental [18] 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 : (1) la réduction de la portée de la réflexion au niveau de la compilation, par exemple pour l'architecture OPENC++ [Chiba, 1995], ou bien (2) l'utilisation de techniques de transformation de programmes, et en particulier l'évaluation partielle [Masuhara *et al.*, 1995], pour réduire au maximum l'interprétation.

6.8 Conclusion

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

18. 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 [Kishimoto *et al.*, 1995].

7

Programmation parallèle et réactive à objets

DANS LE CADRE DE LA PROGRAMMATION PARALLÈLE, répartie, ou concurrente, nous présentons un modèle et un langage mettant en œuvre le paradigme *objet*. EIFFEL//, extension du langage EIFFEL (cf. *chapitres 2 et 3*), se situe dans la catégorie des *langages de classes*, et le parallélisme qui en résulte est de type MIMD.

L'objectif et l'originalité principale du modèle proposé est la *réutilisation* : apporter à la programmation parallèle le potentiel de réutilisation des langages à objets.

La section 7.1 présente les concepts du langage EIFFEL// ; nous les illustrons par un exemple à la section 7.2. La section 7.3 démontre comment il est possible de réaliser dans le cadre impératif proposé une programmation abstraite et déclarative du contrôle des processus. La réutilisation nous permet de proposer une démarche de programmation (section 7.4). Utilisant les possibilités d'extension par bibliothèques, la section 7.5 étend le modèle d'objet actif en introduisant un modèle d'objet réactif.

7.1 Modèle de programmation parallèle

Cette première section présente et discute le modèle de programmation parallèle asynchrone que nous proposons, son illustration et sa mise en œuvre par le langage EIFFEL//, une extension du langage EIFFEL [Meyer, 1992a] (voir *chapitre 2*).

7.1.1 Exécutions concurrentes

L'expression des exécutions concurrentes est une dimension essentielle des systèmes parallèles. Nous situant dans le cadre restreint du parallélisme asynchrone, la conception d'un modèle soulève une première question : comment exprimer les activités concurrentes ?