Les Langages Concurrents à Objets*

Rachid Guerraoui

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

RÉSUMÉ. *L'intégration de la programmation par objets et de la programmation concurrente a suscité beaucoup d'intérêt durant ces dernières années. Cette intégration a donné naissance à de nombreux* langages concurrents à objets*. Dans le but de mieux comprendre et de distinguer ces langages, nous proposons de les classifier à travers les quatre critères suivants, qui concernent l'expression et le contrôle de la concurrence: (1) l'association entre les objets et les processus, (2) la granularité de la concurrence, (3) le contrôle intra-objet, et (4) le contrôle inter-objets. Pour chacun de ces critères, nous présentons les différentes alternatives que nous illustrons par des exemples.*

ABSTRACT. *There has been a great interest since a few years in integrating object-based programming and concurrent programming. This integration has given birth to many* object-based concurrent languages*. For a better understanding and distinction of these languages, we propose to classify them based on the following four criteria, which concern concurrency expressivity and concurrency control: (1) the association between objects and processes, (2) the granularity of concurrency, (3) the intra-object control, and (4) the inter-objects control. For each of these criteria, we present the various alternatives which we illustrate through examples.*

MOTS-CLÉS : *Objet, concurrence, contrôle intra-objet, contrôle inter-objets, encapsulation, granularité, modularité.*
KEY WORDS : *Object, concurrency, intra-object control, inter-objects control, encapsulation, granularity, modularity.*

*. Article publié dans la revue Techniques et Sciences Informatiques, 1995.

1. Introduction

1.1. *Les langages à objets*

La programmation par objets consiste à *encapsuler* des données logiquement reliées et les opérations permettant de les manipuler dans une même entité appelée *objet*. D'une part, cette *encapsulation* favorise l'*abstraction*, car nul besoin de connaître la réalisation des opérations et la structure des données d'un objet pour l'utiliser. D'autre part, l'encapsulation favorise la *modularité*, car la réalisation des opérations et la structure des données peuvent être modifiées sans répercussion sur les autres objets d'une application.

Tous les langages de programmation permettent de définir un programme en termes d'ensembles regroupant données et opérations (des objets) [MAS 89]. Cependant, seuls certains offrent un mécanisme d'encapsulation permettant de manipuler directement ces objets. Parmi ces langages, certains offrent, en plus, des mécanismes tels que l'*instanciation* et l'*héritage* qui exploitent les capacités de réutilisabilité offertes par le concept d'objet. Grâce à l'instanciation, le programmeur définit des moules d'objets, appelés *classes*, à partir desquels il peut engendrer des objets. L'héritage permet de définir des classes par dérivation de classes existantes. P. Wegner [WEG 87] a proposé la dénomination *"object-based languages"* pour les langages offrant un mécanisme d'encapsulation, *"class-based languages"* pour ceux proposant en plus un mécanisme d'instanciation, et *"object-oriented languages"* pour ceux proposant en plus un mécanisme d'héritage. ADA [BAR 89] par exemple appartient à la première catégorie, SIMULA [BIR 73] à la seconde, et SMALLTALK [GOL 83] à la troisième. Notre étude porte sur la première catégorie *("object-based languages")*, que nous appelons *Langages à Objets (LOs)*, et qui englobe les deux autres.

1.2. *Les langages concurrents à objets*

Un langage concurrent est un langage qui permet de distinguer les activités d'un programme qui peuvent être en cours d'exécution au même instant. Ces activités peuvent s'exécuter simultanément dans un système parallèle, en temps partagé dans un système quasi-parallèle, ou en *coroutines* dans un système pseudo-parallèle [SCH 86]. L'intégration des concepts d'objet et de concurrence a suscité l'intérêt de nombreuses communautés de chercheurs dans des domaines différents, allant de l'intelligence artificielle aux systèmes d'exploitation, en passant par les langages de programmation et les bases de données. Parmi les objectifs de cette intégration :

- Une meilleure modélisation des objets concrets du monde réel qui souvent se comportent de manière concurrente.
- L'amélioration des performances d'exécution en utilisant la concurrence exprimée dans les programmes.
- L'exploitation des bénéfices de modularité offerts par le concept d'objet dans les programmes concurrents souvent complexes.

- L'extension du champs d'application des LOs à d'autres types de problèmes, tels que le multi-fenêtrage, la répartition etc.

- De nombreux *Langages Concurrents à Objets (LCOs)* sont apparus dans la littérature. Certains sont des extensions concurrentes de LOs *traditionnels*, alors que d'autres ont été conçus avec le soucis initial d'intégrer la concurrence.

1.3. *Vers une synthèse des langages concurrents à objets*

Les LCOs présentés dans la littérature diffèrent sur plusieurs critères, dont certains concernent la manière d'exprimer et de contrôler la concurrence. Nous proposons dans cet article une synthèse de ces LCOs à travers ces derniers critères. Nous renvoyons le lecteur à [WEG 87], [MAS 89], et [NIE 89] pour d'autres critères, tels que l'héritage simple ou multiple, le typage et le sous-typage, la généricité et l'homogéneité, permettant de distinguer les LOs en général.

Concernant l'expression de la concurrence, nous avons considéré les critères suivants : (1) l'association entre les objets et les processus, et (2) la granularité de la concurrence. Concernant le contrôle de la concurrence, nous avons considéré : (3) le contrôle intra-objet, chargé de préserver la cohérence interne de chaque objet, indépendemment des autres objets, et (4) le contrôle inter-objets, chargé de préserver la cohérence globale des objets, en tenant compte de leurs relations. Pour chacun de ces critères, nous présentons les différentes alternatives adoptées par les LCOs, que nous illustrons par des exemples. En fonction de ces alternatives, nous proposons une classification des différents LCOs. Cette classification complète, en ce qui concerne les différentes manières d'aborder la concurrence, les classifications précédentes présentées dans [WEG 87] et [PAP 92].

Cet article est organisé de la manière suivante. La section 2 propose un survol des LCOs étudiés dans cet article. La section 3 présente les différentes alternatives adoptées par ces LCOs, concernant l'expression de la concurrence (critères (1) et (2) ci-dessus). La section 4 présente les différentes alternatives adoptées par ces LCOs, concernant le contrôle de la concurrence (critères (3) et (4) ci-dessus). La conclusion résume notre classification et évoque des critères complémentaires de classification.

2. Les langages concurrents à objets

Un système concurrent est caractérisé par le fait que plusieurs activités peuvent être en cours d'exécution au même moment. Les langages dits concurrents permettent d'exprimer, directement dans les programmes, ces activités qui sont exécutées par des *processus*. Nous considérons ici le terme *processus* au sens abstrait du terme et indépendemment de toute architecture sous-jacente, i.e parallèle, pseudo-parallèle, ou quasi-parallèle [SCH 86].

Les langages concurrents à objets (LCOs) offrent, à la fois des mécanismes d'encapsulation, et des mécanismes de manipulation de processus. Cette section propose un bref survol des LCOs étudiés dans cet article (figure 1). Notre liste, sans être exhaustive, inclut les LCOs les plus représentatifs de la littérature et des différentes approches possibles.

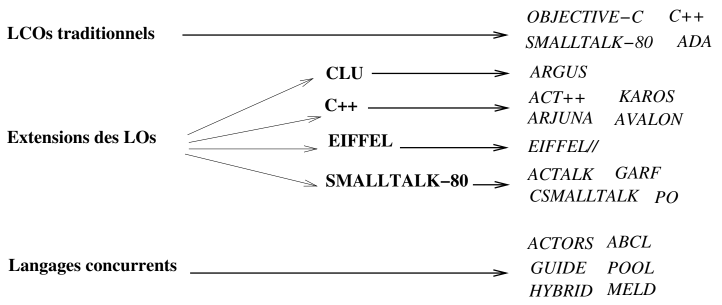

Figure 1. *Les langages concurrents à objets*

2.1. *Les LCOs "traditionnels"*

Les langages C++ [STR 86] et OBJECTIVE-C [COX 86] n'offrent pas de mécanismes propres de gestion de processus. Cependant, du fait que ces langages dérivent du langage C [KER 78], ils offrent toutes les fonctions de C, et en l'occurrence celles permettant d'accéder au système d'exploitation. Le programmeur peut ainsi, en C++, ou en OBJECTIVE-C, manipuler les processus du système d'exploitation sous-jacent. SMALLTALK-80 [GOL 83] et ADA [BAR 89] offrent des primitives propres de manipulation de processus (*tâches* en ADA).

C++, OBJECTIVE-C, SMALLTALK-80 et ADA sont donc considérés, suivant notre terminologie, comme des LCOs. CLU [LIS 79], EIFFEL [MEY 88] et CLOS [BOB 88], quant à eux, ne seront pas considérés comme tels, n'offrant pas de mécanismes de gestion de processus.

2.2. *Les extensions concurrentes des LOs*

Plusieurs LCOs, et particulièrement les prototypes universitaires, sont des extensions de LOs classiques offrant leur propre gestion de processus. En se basant sur des LOs existants, leurs concepteurs ont pu faire l'économie du développement des mécanismes d'encapsulation, de classification ou d'héritage. Parmi les extensions de SMALLTALK-80, on peut citer CSMALLTALK ("Concurrent Smalltalk") [YOK 87], PO [COR 87], ACTALK [BRI 89, BRI 94] et GARF [GAR 94]. CSMALLTALK a été développé à l'université de Keio dans le but de supporter des applications réparties. PO a été développé à l'université de Bologne et utilisé pour expérimenter l'adéquation du concept d'objet aux architectures massivements parallèles. ACTALK a été développé à l'université de Paris-VI et utilisé dans un but didactique. GARF, développé à l'Ecole Polytechnique Fédérale de Lausanne, est une bibliothèque de classes permettant la programmation d'applications réparties tolérantes aux pannes. ARJUNA [PAR 88], AVALON [HER 87], ACT++ [KAF 89] et KAROS [GUE 92b] sont autant d'extensions concurrentes de C++. ARJUNA, AVALON et KAROS ont été développés dans le but de supporter des applications tolérantes aux pannes, respectivement à l'université de Newcastle, de Carnegie Mellon et au CEA-Saclay. ACT++ a été développé à l'institut polytechnique de Virginie pour des applications temps-réel. EIFFEL// [CAR 93] est une extension du langage EIFFEL, développé à l'université de Nancy, et ARGUS [LIS 83] une extension de CLU, développé à l'institut technologique du Massachussets. Le premier est souvent présenté sur des exemples de simulation, alors que le second supporte des applications réparties tolérantes aux pannes.

2.3. *Les langages concurrents*

Certains LCOs ont été créés avec le soucis initial d'intégrer la concurrence. Parmi ceux-ci, HYBRID [NIE 87], développé à l'université de Genève, et GUIDE [KRA 90], à l'université de Grenoble, supportent des applications réparties. ABCL [YON 87], développé à l'université de Tokyo, POOL [AME 87], aux laboratoires Philips, MELD [KAI 89] à l'université de Columbia, et ACTORS [AGH 86], développé à l'institut technologique du Massachussets, sont adaptés à des architectures massivement parallèles. De nombreux autres langages, tels que PLASMA [ARC 94], MERING IV [FER 91] et PRAL-1 [LAN 91] ont été developpés à partir du modèle dit *acteur*, sous-jacent au langage ACTORS. Nous considérerons donc ACTORS comme représentatif de ces langages. Le modèle *acteur* a aussi inspiré les langages ACTALK, ACT++, et ABCL. Les LCOs peuvent être caractérisés par leurs mécanismes d'expression et de contrôle de la concurrence. Les deux sections suivantes portent respectivement sur le premier et le second aspect.

3. Expression de la concurrence

Cette section étudie les différentes approches adoptées par les LCOs pour exprimer la concurrence. Nous considérons tout d'abord les différentes manières d'associer les objets et les processus, puis les différentes granularités de concurrence de ces langages.

3.1. *Association entre objets et processus*

La figure 2 résume les différentes alternatives adoptées par les LCOs pour associer les processus aux objets. Ces alternatives ne sont pas disjointes, et certains LCOs en offrent plusieurs.

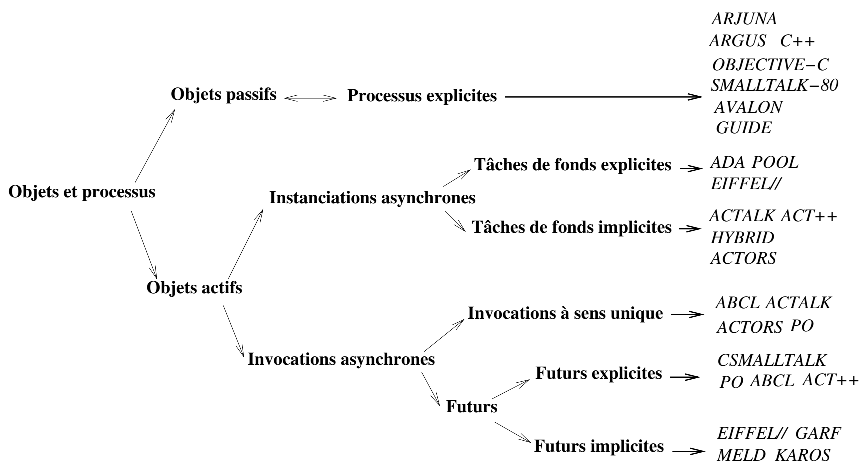

Figure 2. *L'association entre les objets et les processus dans les LCOs*

Deux approches principales peuvent être distinguées. La première considère les processus de manière explicite et orthogonale aux objets : des *processus concurrents* accèdent à des objets *passifs*. La seconde consiste à unifier les processus et les objets : les objets sont dits *actifs*.

3.1.1.*Les objets passifs*

Un LCO à objets passifs dispose, en plus des primitives d'instanciation d'objets et de communication entre objets, des primitives de création de processus : *Task, Process, Fork, Cobegin Coend,* etc. Nous ne décrirons pas ici ces primitives qui ne sont pas liées au concept d'objet.

Cette approche est adoptée dans ARGUS, ARJUNA, AVALON, C++, GUIDE, OBJECTIVE-C et SMALLTALK-80. Dans ces langages, l'identification des activités concurrentes est indépendante du concept d'objet. Comme nous le verrons dans la section 4, certains de ces langages, par exemple GUIDE, permettent néanmoins d'intégrer les mécanismes de synchronisation entre processus aux mécanismes de communication entre objets.

3.1.2.*Les objets actifs*

Une approche unificatrice consiste à considérer les exécutions des opérations sur les objets (du moins certaines d'entre elles) comme des activités logiquement indépendantes. Ces activités s'exécutent par des processus dédiés. Cette unification rend implicite la création des processus car l'invocation de l'une de ces opérations entraîne implicitement la création de processus. L'expression de la concurrence se fait à travers les objets qui sont actifs. Nous parlerons d'*instanciation asynchrone* lorsqu'une opération particulière d'instanciation d'un objet est exécutée par un processus dédié. Lorsque plusieurs *opérations fonctionnelles* d'un objet peuvent être exécutées par des processus dédiés, nous parlerons d'*invocations asynchrones.* Nous utilisons le terme *opérations fonctionnelles* pour désigner les opérations usuellement associées aux objets dans un contexte séquentiel.

a.Les instanciations asynchrones

Cette approche suppose que chaque objet actif dispose d'une opération spécifique d'instanciation exécutée par un processus dédié. Cette opération, appelée *tâche de fond*, est active pendant toute la durée de vie de l'objet. L'instanciation est dite asynchrone car le créateur d'un objet actif n'attend pas la terminaison de la tâche de fond créée pour continuer son exécution. La figure 3 illustre une instanciation asynchrone d'un objet actif *(objet2),* à partir d'un autre objet actif *(objet1).* Dès la création de *(objet2)* (via sa classe), une tâche de fond est exécutée par un processus dédié. Cette tâche de fond peut être soit *implicite* et inaccessible aux programmeurs, soit *explicite* et programmable.

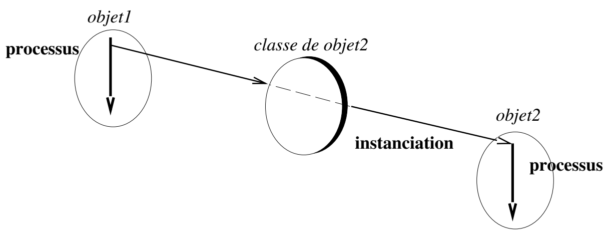

Figure 3. *Instanciation asynchrone*

a.1.*Les tâches de fond implicites:* ACTALK, ACT++ et ACTORS offrent des mécanismes d'instanciations asynchrones, avec des tâches de fond implicites. Dans ACT++ par exemple, lorsqu'un objet actif est créé, un processus associé est initié. Ce processus exécute une tâche de fond qui consiste à attendre les invocations adressées à l'objet et à les traiter (en exécutant les opérations invoquées) dans leur ordre d'arrivée. Un objet actif qui crée un autre objet actif se contente de *lancer* la tâche de fond sans attendre la fin de son exécution. Ce comportement ne peut être modifié par le programmeur et est le même pour tous les objets.

a.2.*Les tâches de fond explicites:* ADA, EIFFEL// et POOL offrent des mécanismes d'instanciations asynchrones, avec des tâches de fond explicites. Comme dans ACT++, les objets actifs de ces langages disposent d'une tâche de fond. A la différence du premier, cette tâche de fond est accessible aux programmeurs. Il est possible de la programmer en y définissant, par exemple, une initialisation particulière des objets.

b.Les invocations asynchrones

Dans un langage séquentiel à objets, l'objet (client) qui invoque une opération sur un autre objet (serveur) attend systématiquement la fin de l'exécution de l'opération invoquée pour continuer son exécution : l'invocation est dite *synchrone* (figure 4 (a)). Si l'on considère que l'opération invoquée porte sur une activité indépendante, cette dernière peut être exécutée par un processus différent de celui qui a effectué l'invocation. Le client n'est alors plus obligé d'attendre la fin de l'exécution de l'opération invoquée. Dans ce cas, l'invocation est dite *asynchrone* (figure 4 (b)).

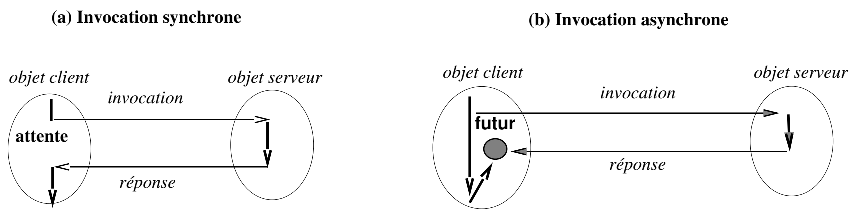

Figure 4. *Invocation synchrone vs. asynchrone*

L'asynchronisme des invocations permet d'exprimer la concurrence de manière élégante mais pose le problème de la récupération des réponses. En effet, dans le cas d'une invocation synchrone, le client attend systématiquement l'arrivée de la réponse. Dans le cas asynchrone, le client continue son calcul sans attendre la réponse. Certains LCOs règlent le problème en introduisant des mécanismes d'invocations asynchrones *à sens unique* qui suppriment la notion de réponse. D'autres LCOs introduisent des objets particuliers, appelés *futurs*, permettant au client d'une invocation asynchrone de récupérer la réponse (figure 4 (b)). On parlera de *futurs explicites* lorsque les *futurs* sont directement manipulés par les objets, et de *futurs implicites* dans le cas contraire [HAL 85].

b.1.*Les envois de messages à sens unique :* ABCL, ACTALK, HYBRID et PO permettent les invocations à sens unique. Après l'invocation d'une opération, le client est découplé du serveur. La récupération d'une réponse, si nécessaire, doit se faire grâce à une invocation du client par le serveur. Cela signifie que le client aura prévu une ou plusieurs opérations spécifiques, invoquées en cas de réponse.

b.2.*Les futurs explicites :* ACT++ et CSMALLTALK introduisent des *futurs* manipulés explicitement par le client. ABCL et PO offrent cette possibilité, en plus de l'invocation à sens unique. Dans ACT++ par exemple, une “boîte à réponses", appelée *Cbox*, doit être créée par le client et transmise en paramètre de l'invocation lorsqu'une réponse est souhaitée. Le client utilise ensuite cette *Cbox* pour attendre et récupérer la réponse.

```
Cbox maCbox;                                                /* création d'une "Cbox" */
factorielle << &ConcFact::calculFact << maCbox << n;     /* invocation de l'opération */
int k;
maCbox >> k;                                                /* lecture de la réponse */
```

Figure 5. *Exemple de futur explicite en ACT++*

Sur la figure 5, la variable *factorielle* désigne un objet actif en ACT++ qui permet de calculer des factorielles au moyen de l'opération *calculFact()*. Le client de l'objet *factorielle* crée la *Cbox maCbox*, la transmet en paramètre de l'invocation de l'opération *calculFact()*, puis y récupère ensuite la réponse dans la variable *k* (*n* désigne l'entier dont la factorielle doit être calculée). Le client sera bloqué en attente si *maCbox* est vide au moment de la récupération de la réponse.

b.3.*Les futurs implicites:* EIFFEL//, KAROS, et MELD introduisent aussi des *futurs*, mais qui ne sont pas manipulés explicitement par les objets. Une invocation asynchrone retourne une variable, appelée *futur implicite*, directement utilisée pour recevoir la réponse. Le client n'est mis en attente que lorsqu'il désire accéder à la valeur de cette variable, et que cette dernière n'est pas disponible.

```
k := factorielle.calculFact ( n );          -- invocation de l'opération
k.print ();                                  -- accès a la valeur de k
```

Figure 6. *Exemple de futur implicite en EIFFEL//*

Sur la figure 6, la variable *factorielle* désigne un objet actif en EIFFEL//. La variable *k* est un *futur* implicite. Le client de l'invocation récupère directement la réponse dans *k*. Si au moment de l'invocation de l'opération *print()* sur la variable *k*, la valeur de cette dernière n'est pas disponible, le client est mis implicitement en attente. On parle alors d'*attente par nécessité* [CAR 94].

3.2.*La granularité de la concurrence*

Le nombre maximal d'exécutions d'opération en cours, à un instant donné, pour un objet, détermine la granularité de la concurrence du langage [WEG 87].

Lorsque chaque objet exécute ses opérations l'une après l'autre, la concurrence n'est possible qu'entre des exécutions de différents objets : elle est dite *inter-objets*. Lorsque plusieurs exécutions peuvent être en cours, au même moment sur un objet : la concurrence est dite *intra-objet*. La figure 7 présente les différentes granularités de la concurrence des LCOs.

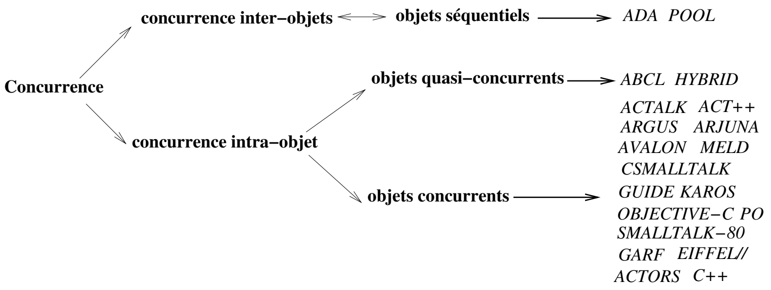

Figure 7. *La granularité de la concurrence des LCOs*

3.2.1.*La concurrence inter-objets : les objets séquentiels*

Un objet est dit séquentiel lorsqu'une seule exécution de ses opérations peut être en cours à tout instant (figure 8 (a)). Un tel objet permet une exécution après l'autre. Une opération est exécutée jusqu'à sa terminaison, sans entrelacement avec d'autres exécutions. Lorsque tous les objets sont séquentiels, la concurrence permise par le langage est uniquement inter-objets.

Les objets actifs dans ADA et POOL sont séquentiels. Dans POOL par exemple, une file d'attente est implicitement associée à chaque objet. La file stocke et sérialise les invocations adressées à l'objet. Ce dernier est par conséquent séquentiel car il ne traite qu'une invocation à la fois.

3.2.2.*La concurrence intra-objet*

La concurrence intra-objet peut s'exprimer à travers des objets quasi-concurrents ou des objets concurrents.

a.Les objets quasi-concurrents

Comme un objet séquentiel, un objet quasi-concurrent ne permet pas d'exécutions simultanées. Cependant, l'objet peut "commuter" d'une exécution à une autre. Plus exactement, il peut interrompre une exécution d'opération et permettre une exécution concurrente (figure 8 (b)). Ce modèle permet d'introduire une concurrence intra-objet.

Les objets actifs dans ABCL et HYBRID sont quasi-concurrents. Dans

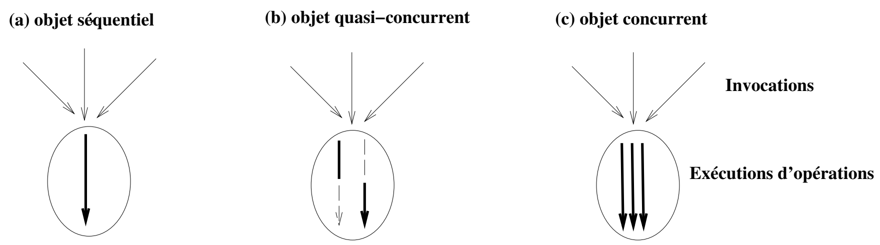

Figure 8. *Les degrés de concurrence possibles dans un objet*

HYBRID, les invocations sont synchrones. Après une invocation, le processus de l'objet client est bloqué jusqu'à l'arrivée de la réponse. Un deuxième processus peut à ce moment être créé pour traiter une invocation adressée à cet objet. Si ce deuxième processus est à son tour bloqué pendant l'exécution de l'opération invoquée, le premier est réactivé dans le cas où la réponse est disponible. Dans le cas contraire, un troisième processus peut être créé pour traiter une autre invocation, et ainsi de suite.

ABCL introduit deux catégories d'invocations : les invocations *ordinaires* et les invocations *express*. Le programmeur peut choisir l'une de ses catégories pour chaque invocation. Une invocation *express* est jugée plus prioritaire qu'une invocation *ordinaire*. Le traitement par un objet actif d'une invocation *ordinaire* est interrompu lorsque cet objet reçoit une invocation *express*. Le traitement de la première reprend lorsque le traitement de la seconde est terminé et qu'aucune autre invocation *express* n'est en attente de traitement.

b.Les objets concurrents

Un objet concurrent peut traiter des invocations de manière simultanée. Autrement dit, une invocation adressée à un objet concurrent peut être traitée en concurrence avec une autre, sans qu'aucun des deux traitements ne soit interrompu (figure 8 (c)). Il est bien évident qu'un tel modèle introduit aussi une concurrence intra-objet.

La plupart des LCOs supportent des objets concurrents : ACT++, ACTORS, ACTALK, ARGUS, ARJUNA, AVALON, CSMALLTALK, EIFFEL//, GUIDE, KAROS, MELD, PO, GARF, SMALLTALK-80, OBJECTIVE-C, et C++. Dans un objet actif de PO par exemple, un processus est créé pour chaque invocation. Au même moment, plusieurs exécutions d'opérations peuvent être en cours sur le même objet actif. Dans GUIDE, les processus sont créés indépendemment des objets. Suivant le contrôle intra-objet associé (section 4), un objet partagé peut permettre des exécutions simultanées.

4. Le contrôle de concurrence

La concurrence non contrôlée peut violer la cohérence d'une application. Cette section étudie les différentes approches proposées par les LCOs pour contrôler la concurrence. Nous distinguons deux types complémentaires de contrôle de concurrence. Le premier, que nous appelons contrôle *intra-objet*, concerne la cohérence de chaque objet d'une application, considéré indépendemment des autres objets. Le second, que nous appelons contrôle *inter-objets*, concerne la cohérence globale d'une application en prenant en compte les dépendances entre les objets.

4.1. *Le contrôle de concurrence intra-objet*

La forme la plus simple de contrôle intra-objet est l'exclusion mutuelle des exécutions d'opération sur un objet *partagé* (i.e qui peut être invoqué par plusieurs objet). Un tel contrôle permet d'éviter les incohérences dues aux interférences entre les exécutions d'opération. D'autres contrôles intra-objet, plus élaborés, sont parfois nécessaires pour assurer un ordonnancement spécifique des opérations. Prenons l'exemple d'un objet *pile*. Ce dernier peut recevoir une invocation portant sur une opération *dépiler()*, alors qu'il est dans un état *vide* (aucun élément à dépiler). L'invocation ne pouvant être traitée, il est possible de retourner un signal d'erreur. Cette solution n'est pas toujours celle souhaitée et il peut être utile de conserver cette invocation en attente d'une autre invocation portant sur l'opération *empiler()*. La pile joue alors le rôle de tampon entre des producteurs et des consommateurs. Le contrôle intra-objet doit alors assurer cet ordonnancement.

Tous les LCOs présentés dans la littérature offrent des mécanismes pour programmer des contrôles intra-objet. Ces mécanismes peuvent être *explicites* ou *implicites*, *externes* ou *internes* aux objets partagés, et *dépendants* ou *indépendants* de leurs opérations fonctionnelles (*empiler()* et *dépiler* sont, par exemple, les opérations fonctionnelles d'un objet *pile*).

La figure 9 présente les différentes approches proposées, et les LCOs correspondants. Parmi ces LCOs, certains offrent à leurs utilisateurs les moyens de programmer *entièrement* le contrôle intra-objet. D'autres offrent seulement la possibilité de programmer une partie de ce contrôle, l'autre partie étant implicite et inaccessible aux programmeurs. Le contrôle est dit *explicite* dans le premier cas et *implicite* dans le second.

4.1.1. *Les contrôles explicites*

Dans le cas de contrôle explicite, l'utilisateur dispose de toute l'algorithmique du langage pour programmer les contrôles intra-objet. Le contrôle associé à une instance d'une classe C peut être soit programmé dans C, soit programmé dans les classes clientes de C. Dans le premier cas, les clients d'une instance de C doivent se synchroniser avant d'invoquer l'une de ses opérations : nous dirons que le contrôle est *explicite et externe* (figure 10 (a)). Dans le second cas, le contrôle se passe dans les opérations de l'instance. Nous dirons que le contrôle est *explicite et interne* (figure 10) (b)).

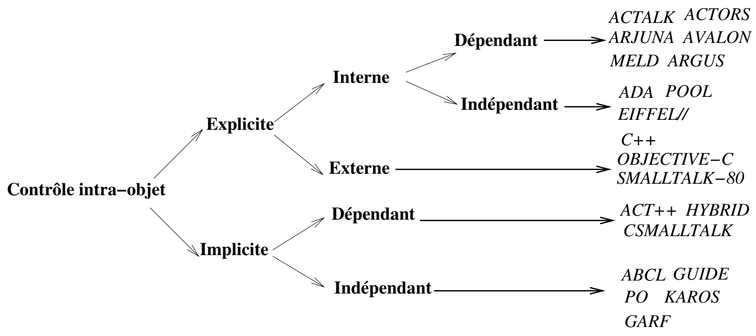

Figure 9. *Les contrôles intra-objet dans les LCOs*

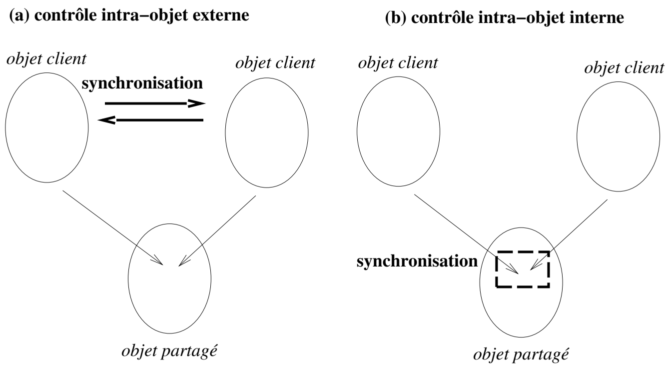

Figure 10. *Contrôle explicite externe vs. interne*

a.Les contrôles explicites et externes

SMALLTALK-80, C++ et OBJECTIVE-C offrent des primitives de synchronisation permettant la programmation de contrôles intra-objet explicites externes. SMALLTALK-80 par exemple, fournit des primitives permettant de synchroniser les processus via des sémaphores. Les clients d'un objet partagé peuvent donc se synchroniser avant d'accéder à l'objet. Les clients de l'objet *pile* pourraient, par exemple, se synchroniser avant d'invoquer les opérations *dépiler()* et *empiler()*.

b.Les contrôles explicites et internes

Lorsque le contrôle est interne, les clients d'un objet partagé ne peuvent directement se synchroniser. La synchronisation doit se faire dans l'objet partagé. Parmi les LCOs qui offrent un contrôle intra-objet explicite et interne, certains proposent de le programmer dans les opérations fonctionnelles de l'objet : nous dirons que le contrôle est *dépendant* (des opérations fonctionnelles). D'autres LCOs proposent de programmer ce contrôle dans une opération particulière: nous dirons que le contrôle est *indépendant* (des opérations fonctionnelles).

```
class  Pile
    export
        empiler, depiler
    inherit
        Process redefine Live;
    feature
        pleine   is  ...-- teste si la pile est pleine   end;
        vide     is  ...-- teste si la pile est vide     end;
        empiler  is  ...-- empiler                       end;
        depiler  is  ...-- dépiler                       end;
        Live     is
              do until FALSE
                  loop
                      if not pleine() then serve_oldest  ( &empiler )  end;
                      if not vide() then serve_oldest  ( & depiler )  end;
                  end;
end
```

Figure 11. *Exemple de contrôle explicite, interne et indépendant en EIFFEL//*

b.1.*Les contrôles explicites internes et dépendants :* Dans ACTALK, ACTORS, ARGUS, ARJUNA, AVALON, et MELD, le contrôle intra-objet est explicitement programmé dans les opérations fonctionnelles de l'objet auquel il est associé. Dans ACTORS par exemple, une primitive *become*, permet de spécifier quelles invocations peuvent être acceptées par un objet. Tant que cette primitive n'est pas exécutée, l'objet bloque en attente toutes les nouvelles invocations. Cette primitive est manipulée dans les opérations fonctionnelles de l'objet.

b.2.*Les contrôles explicites internes et indépendants :* Dans ADA, EIFFEL//, et POOL, le contrôle intra-objet est explicitement programmé dans une opération particulière, indépendante des opérations fonctionnelles. Par exemple, à chaque objet actif de EIFFEL// est associé une opération particulière, définie dans la classe de l'objet, et chargée du contrôle intra-objet. Cette opération, qui joue aussi le rôle de tâche de fond explicite (section 3), traduit les conditions d'acceptations des invocations adressées à l'objet "à la rendez-vous".

La figure 11 présente la classe *Pile* en EIFFEL//. Une instance de cette classe traite les invocations *dépiler()* (resp. *empiler()*) dans leur ordre d'arrivée, pour peu que la pile ne soit pas vide (resp. pleine). Lorsque la pile est vide (resp. pleine), les invocations portant sur l'opération *dépiler()* (resp. *empiler()*) sont mises en attente. Ce contrôle intra-objet est programmé dans l'opération *Live()*. Cette opération exécute une boucle qui, en fonction de l'état de l'objet, traite alternativement les requêtes *empiler()* et *dépiler()*. La primitive *serve-oldest(op)* permet de traiter la plus ancienne (en fonction de l'ordre d'arrivée) requête qui porte sur l'opération *op*.

4.1.2.*Les contrôles implicites*

Dans le cas de contrôle implicite, une partie du contrôle intra-objet est implicite et inaccessible aux programmeurs. Du fait que la partie explicite est toujours programmée dans les objets, nous ne distinguons pas ici les contrôles *implicites et internes*, des contrôles *implicites et externes*. Cependant, suivant que la partie explicite est programmée dans les opérations fonctionnelles de l'objet, ou dans des opérations particulières, nous distinguons les contrôles *implicites et dépendants*, des contrôles *implicites et indépendants*.

a.Les contrôles implicites et dépendants

ACT++, CSMALLTALK et HYBRID proposent une partie implicite du contrôle intra-objet et une partie explicite programmée dans les opérations fonctionnelles de l'objet. Dans HYBRID par exemple, des files d'attentes sont associées aux opérations d'un objet. Plusieurs opérations peuvent être associées à la même file d'attente. Cette association est définie à la déclaration des classes. Une file d'attente peut être dans un état *ouvert* ou dans un état *fermé*. Chacun de ces états traduit la possibilité d'exécuter ou non les opérations associées à la file d'attente. Lorsqu'à sa déclaration, une opération n'est associée à aucune file d'attente, elle est considérée comme associée à une file dans un état *ouvert* permanent. Lorsqu'une file d'attente est dans l'état *fermé*, les invocations portant sur les opérations qui lui sont associées sont bloquées en attente de son ouverture. Si la file est dans l'état *ouvert*, les opérations associées sont exécutées avec la seule contrainte de l'exclusion mutuelle.

La figure 12 présente la classe *Pile* en HYBRID. Comme le montre cette

```
type Pile:
   private { var   f_emp, f_dep : delay;  # files d'attente    }
   init: ->; {   f_emp.open(); f_dep.close(); }
    pleine :-> int; {                 ... # teste si la pile est pleine  }
    vide : -> int;  {                 ... # teste si la pile est vide   }
    empiler : Element  -> ;  uses   f_emp;  {
              if  (vide())   then f_dep.open()  ;
                                           ... # empiler
              if    (pleine())  then  f_emp.close(); }

     depiler : ->  Element;  uses    f_dep; {
              if    (pleine())  then f_emp.open();
                                            ... # dépiler
              if  (vide())       then  f_dep.close();  }

}
```

Figure 12. *Exemple de contrôle implicite et dépendant en HYBRID*

figure, la partie explicite du contrôle intra-objet est programmée dans les opérations fonctionnelles de l'objet *pile*. Ce contrôle consiste à *fermer* et à *ouvrir* les files d'attente. La partie implicite traduit l'état des files en politique d'ordonnancement des invocations. La manipulation des files d'attente, *f-emp* et *f-dep*, associées respectivement aux opérations *empiler()* et *dépiler()*, se fait dans le corps de ces opérations, à l'aide des primitives *open()* et *close()*. Ainsi, l'opération *empiler() ouvre* la file d'attente associée à l'opération *dépiler()* si cette file est *fermée*, puis *ferme* la file d'attente associée à l'opération *empiler()* si la *pile* est pleine. De même, l'opération *dépiler() ouvre* la file d'attente associée à l'opération *empiler()* si cette file est *fermée*, puis *ferme* la file d'attente associée à l'opération *dépiler()* si la *pile* est vide.

b.Les contrôles implicites et indépendants

ABCL, GUIDE, KAROS et PO proposent une partie implicite du contrôle intra-objet et une partie explicite, séparée des opérations fonctionnelles et programmée dans des opérations particulières de l'objet. Dans GUIDE par exemple, des conditions d'activation sont associées aux opérations des objets. Ce sont des opérations particulières qui retournent un résultat booléen. Ces opérations peuvent être programmées en fonction de l'état de l'objet, de ses variables d'instances, et de l'état d'exécution des autres opérations. Lorsqu'un objet reçoit une invocation, il évalue la condition d'activation associée à l'opération invoquée. Lorsque la condition est vérifiée, l'opération est exécutée. Autrement, l'invocation est bloquée et la condition est réévaluée à la fin de l'exécution d'une autre opération.

La partie explicite du contrôle est programmée dans les conditions d'activation et consiste à retourner *vrai* ou *faux* suivant si l'opération correspondante peut ou non être exécutée. La partie implicite consiste à traduire les valeurs retournées par les conditions d'activation en politique d'ordonnancement des invocations.

```
CLASS   Pile
    IMPLEMENTS ProductConsom  IS
                                        ... //  variables d'instances
      METHOD
          Empiler( IN   m: Element) ;
                                        ... // empiler un élément
          Depiler( OUT  m: Element) ;
                                        ... // depiler un élément
      CONTROL
        Empiler: (completed(Empiler) – completed(Depiler) < taille)
                 ((current(Depiler) = 0)  AND  (current(Empiler) = 0));
                                  ... // condition d'activation pour empiler()

        Depiler: (completed(Empiler) > completed(Depiler))
                 ((current(Depiler) = 0)   AND  (current(Empiler) = 0));
                                  ... // condition d'activation pour dépiler()
END    Pile.
```

Figure 13. *Exemple de contrôle implicite et indépendant en GUIDE*

La figure 13 présente la classe *Pile* en GUIDE [DEC 91]. Les conditions d'activation associées aux opérations *empiler* et *dépiler* sont séparées de ces dernières et définies dans la partie *CONTROL* de la classe *Pile*. Ces conditions sont exprimées à l'aide des compteurs de synchronisation *completed(op)* et *current(op)*. Le premier compteur calcule le nombre de fois ou une opération *(op)* a été exécutée, alors que le second calcule le nombre d'exécutions courante de l'opération. La condition d'activation associée à l'opération *empiler* est une condition booléenne, évaluée à *vrai* si et seulement si (1) la différence entre le nombre d'éléments empilés et du nombre d'éléments dépilés est inférieure à la taille de la pile (ce qui signifie que la pile n'est pas pleine) et (2) aucune opération n'est en cours d'exécution. La condition d'activation associée à l'opération *dépiler* est une condition booléenne, évaluée à *vrai* si et seulement si (1) le nombre d'éléments empilés est supérieur au nombre d'éléments dépilés (ce qui signifie que la pile n'est pas vide) et (2) aucune opération n'est en cours d'exécution.

4.2. *Le contrôle de concurrence inter-objets*

Le contrôle intra-objet assure la cohérence d'une application lorsque tous les objets sont indépendants. Cela n'est pas toujours le cas car la cohérence d'une application est parfois basée sur des hypothèses faites par des objets sur l'état d'autres objets. Les traitements concurrents peuvent invalider ces hypothèses et par conséquent violer cette cohérence.

Considérons par exemple le cas d'une application bancaire dans laquelle deux objets représentant des *agents bancaires* sont respectivement chargés de la consultation et du transfert de fonds entre deux autres objets représentant des *comptes bancaires*. L'entrelacement de la consultation et du transfert peut conduire à des incohérences même si chaque *compte bancaire* dispose d'un contrôle intra-objet propre. Le problème vient du fait que les opérations de consultation et de transfert des agents bancaires doivent chacune s'exécuter *globalement* de manière atomique. Plus précisément, l'agent chargé de la consultation ne doit pas voir l'état des comptes *pendant* le transfert.

Ce problème peut être résolu en *encapsulant* chacune des opérations de consultation et de transfert dans une transaction. La propriété de *sérialisabilité* des transactions assure que tout se passe comme si ces dernières s'exécutaient de manière séquentielle. Ainsi, l'opération de consultation ne pourra voir l'état des comptes qu'*avant* ou *après* un transfert. Nous ne ferons aucune hypothèse sur la propriété d'atomicité aux pannes des transactions. Pour une définition formelle plus complète des transactions, dans le cadre d'un système à objets répartis, nous renvoyons le lecteur à [WEI 84, GUE 94b].

Le concept de transaction a été utilisé avec succès dans les systèmes de gestion de bases de données [BER 87], et de nombreux auteurs ont suggéré son intégration dans les LCOs. Cette intégration consiste, d'une part à introduire un moyen d'exprimer les transactions dans le langage, et d'autre part à concevoir des mécanismes permettant d'assurer la propriété de sérialisabilité. Ces mécanismes doivent synchroniser les invocations des objets, appelés *objets atomiques* [WEI 84], concernés par la sérialisabilité (par exemple les comptes bancaires). Nous étudions ici les différentes approches adoptées par les LCOs pour exprimer les transactions et les mécanismes de sérialisabilité.

4.2.1. *L'expression des transactions*

Deux manières d'exprimer des transactions dans un LCO ont principalement été considérées dans la littérature. La première consiste à fournir dans le langage des primitives permettant de définir les frontières des transactions sous forme de *blocs transactionnels*. La seconde consiste à intégrer les transactions aux invocations d'objets. Suivant cette seconde approche, les frontières d'une transaction deviennent implicites et sont toujours celles d'une invocation d'opération. Nous parlerons d'*invocations atomiques*.

a. Les blocs transactionnels

ARGUS, ARJUNA, AVALON, GUIDE, HYBRID, et MELD offrent des primitives permettant de définir des blocs transactionnels en délimitant les frontières des transactions. Dans MELD par exemple, l'opérateur "<" signifie *début du bloc* et l'opérateur ">" signifie *fin du bloc*. Une transaction débute avec la première instruction contenue dans un bloc, puis se termine quand la dernière instruction est exécutée et que toutes les instructions invoquées en cascade, via les invocations d'opérations, sont terminées.

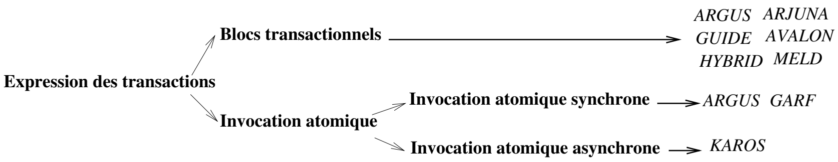

Figure 14. *L'expression des transactions dans les LCOs*

La figure 15 présente la programmation de la classe *Agent* (agent bancaire) en MELD. Le transfert (qui englobe le dépôt et le retrait sur les comptes bancaires *compte1* et *compte2*) est exécuté de manière atomique du fait qu'il est inclus dans le bloc transactionnel délimité par les opérateurs "<" et ">".

```
CLASS
   Agent::=compte1,compte2:Compte_Bancaire;
      METHODS
         Transfert() -->
               <
                compte1.depot(50);
                compte2.retrait(50);
               >
         ...
END CLASS
```

Figure 15. *Exemple de bloc transactionnel en MELD*

b.Les invocations atomiques

Les transactions sont associées aux invocations d'opérations qui sont dites *atomiques*. Les invocations des opérations de transfert et de consultation sur les agents bancaires devraient par exemple être atomiques. Suivant les types d'invocations auxquelles sont intégrées les transactions, nous distinguons les *invocations atomiques synchrones* et les *invocations atomiques asynchrones*.

b.1.*Les invocations atomiques synchrones :* ARGUS offre à la fois des blocs transactionnels et des invocations atomiques synchrones. Un programme écrit en ARGUS consiste en un ensemble d'objets appelés *Guardians*. Ces objets communiquent de manière synchrone et atomique. Chaque invocation d'opération est exécutée dans une transaction. Le début de la transaction correspond à l'invocation de l'opération chez le client. La fin de la transaction correspond à la réception de la réponse par le client. L'imbriquation des invocations se traduit par l'imbrication des transactions [MOS 85]. Plus exactement, lorsque dans une opération *op1*, une autre opération *op2* est invoquée, une sous-transaction de la transaction en cours est créée. Cette sous-transaction se termine lorsque la réponse à *op2* est retournée. La transaction parente contenant *op1* peut alors continuer son exécution. La figure 16 présente schématiquement une invocation atomique synchrone. $T1$ désigne une transaction contenant une opération *op1*, à partir de laquelle une opération *op2* est invoquée. Cette invocation se fait dans une sous-transaction $T11$ de $T1$.

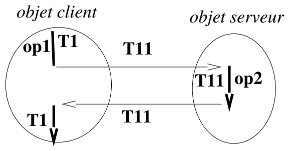

Figure 16. *Invocation atomique synchrone*

b.2.*Les invocations atomiques asynchrones :* Les objets actifs de KAROS communiquent de manière atomique et asynchrone. Autrement dit, KAROS associe les transactions aux invocations asynchrones. Lorsque dans une opération *op1*, une autre opération *op2* est invoquée, deux sous-transactions sont créées. La première contient l'invocation et se termine lorsque la réponse à *op2* est retournée. La deuxième contient la suite de l'opération *op1* et se termine lorsque cette dernière est terminée. La figure 17 représente schématiquement une invocation atomique asynchrone. $T1$ désigne une transaction contenant une opération *op1*, à partir de laquelle une opération *op2* est invoquée. Cette invocation se fait dans une sous-transaction $T11$ de $T1$. La suite de l'opération *op1* se fait dans une sous-transaction $T12$ de $T1$.

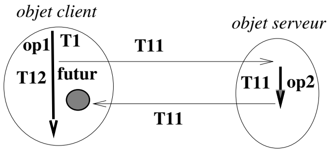

Figure 17. *Invocation atomique asynchrone*

4.2.2.*Les objets atomiques*

Les objets atomiques sont les objets pour lesquels la propriété de sérialisabilité des transactions doit être assurée. Dans l'exemple du transfert de fonds, les deux comptes bancaires doivent être des objets atomiques. Afin d'assurer la propriété de sérialisabilité sur ces objets, il est nécessaire de fournir (1) un *protocole de sérialisation*, chargé de définir un ordre global de sérialisation des transactions, (2) un mécanisme de *détection de conflits* (identification des opérations de modifications par exemple), et (3) un mécanisme de *traitement de ces conflits* (verrouillage par exemple). Alors que le protocole de sérialisation doit être le même pour tous les objets atomiques, les mécanismes de détection et de traitement des conflits peuvent être adaptés à chacun de ces objets [GUE 94b]. De nombreux auteurs ont montré que dans de nombreux cas, la prise en compte de la sémantique des objets dans ces mécanismes permet d'augmenter la concurrence entre les transactions [CAR 89].

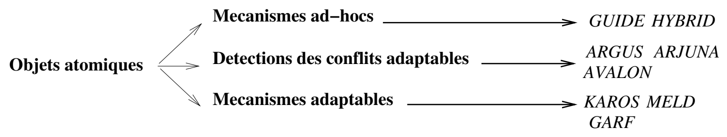

Figure 18. *Les mécanismes de synchronisation des objets atomiques dans les LCOs*

Parmi les LCOs qui supportent des transactions (figure 14), certains offrent des mécanismes ad-hoc de détection et de traitement des conflits. Ces mécanismes sont implicites et inaccessibles aux programmeurs. D'autres LCOs offrent un mécanisme ad-hoc de traitement des conflits mais permettent aux programmeurs d'adapter le mécanisme de détection de conflits à chaque objet atomique. Enfin, d'autres LCOs permettent d'adapter les deux mécanismes à chaque objet. Ces différentes alternatives, ainsi que les LCOs correspondants, sont résumées sur la figure 18.

a.Les mécanismes ad-hocs

GUIDE et HYBRID proposent des mécanismes ad-hoc de détection et de traitement des conflits. Ces mécanismes sont implicites et inaccessibles aux programmeurs. Dans GUIDE par exemple, le mécanisme de détection des conflits est basé sur la distinction *lecture/écriture*: seuls les accès en écriture provoquent des conflits. Le mécanisme de traitement de ces conflits est pessimiste, et basé sur un verrouillage à deux phases.

b.Les mécanismes de détection de conflits adaptables

ARJUNA, ARGUS et AVALON permettent aux programmeurs de raffiner le mécanisme offert par défaut pour détecter les conflits entre transactions. Cependant, le mécanisme de traitement des conflits est inaccessible aux programmeurs. Dans ARJUNA, le traitement des conflits est basé sur le verrouillage à deux phases. Ce mécanisme ne peut pas être modifié. Cependant, le programmeur peut adapter la détection des conflits à la sémantique des objets. Plus exactement, il peut définir des verrous particuliers (typés) en fonction du type des opérations des objets.

c.Les mécanismes de détection et de traitement de conflits adaptables

KAROS, MELD et GARF permettent d'adapter, aussi bien le mécanisme de détection des conflits, que le mécanisme de leur traitement. Dans KAROS par exemple, le programmeur peut définir des mécanismes de traitement des conflits propres à chaque classe d'objets atomiques. Certaines classes peuvent adopter des mécanismes optimistes alors que d'autres classes peuvent adopter des mécanismes pessimistes [GUE 93].

5. Conclusion

Les Langages Concurrents à Objets (LCOs) se proposent de mieux modéliser les applications concurrentes, de permettre l'amélioration des performances d'exécution des programmes, et d'étendre le champs d'application des langages à objets en abordant de nouveaux problèmes. De nombreux LCOs ont été développés et présentés dans la littérature.

Afin de mieux comprendre les particularités de ces langages et les modèles sous-jacents, nous avons proposé dans cet article une synthèse de leurs différentes approches de la concurrence. Plutôt que d'énumérer les différents LCOs en soulignant leurs caractéristiques, nous les avons comparés à travers quatre critères : l'association entre les objets et les processus, la granularité de la concurrence, le contrôle de concurrence intra-objet, et le contrôle de concurrence inter-objets. Nous avons présenté chacun de ces aspects, ainsi que les différentes alternatives adoptées par les LCOs. Cela nous amené à une classification des LCOs, illustrée sur la figure 19.

D'autres critères concernant les modélisations formelles [NIE 93], l'adéquation à des machines parallèles [CAP 92a][CAP 92b] ou à des architectures réparties [GUE 94], pourraient aussi être pris en compte dans une classification complémentaire. Par ailleurs, et du fait que dans de nombreux cas, seuls les modèles sont décrits dans la littérature, il est difficile d'évaluer le niveau de développement des LCOs. A part les langages commercialisés (SMALLTALK-80, OBJECTIVE-C, C++ et ADA), il existe peu de LCOs (tels que ACTALK) accessibles à travers le réseau. Pour la plupart, les LCOs décrits dans cet article sont des prototypes universitaires qui n'ont pas encore été testés sur des applications réelles.

Choix de conception	Alternatives	*ABCL*	*ACT++*	*ACTORS ACTALK*	*ADA POOL*	*ARGUS*	*ARJUNA*	*AVALON*	*CSMALLTALK*	*EIFFEL//*	*GUIDE*	*HYBRID*	*KAROS GARF*	*MELD*	*PO*	*C++ OBJECTIVE-C SMALLTALK-80*
Expression de la concurrence	Processus explicites					×	×	×			×					×
	Invocations à sens unique	×		×								×			×	
	Futurs explicites	×	×						×						×	
	Futurs implicites									×			×	×		
	Tâches de fonds explicites				×					×						
	Tâches de fonds implicites		×	×												
Granularité de la concurrence	Objets séquentiels				×											
	Objets quasi-concurrents	×										×				
	Objets concurrents		×	×		×	×	×	×	×	×		×	×	×	×
Contrôle intra-objet	Explicite externe															×
	Explicite dépendant			×		×	×	×						×		
	Explicite Indépendant					×				×						
	Implicite dépendant		×						×			×				
	Implicite indépendant	×									×		×		×	
Contrôle inter-objets — *Expression des transactions*	Blocs transactionnels					×	×	×			×	×		×		
	Invocations synchrones					×										
	Invocations asynchrones											×				
Contrôle inter-objets — *Objets atomiques*	Mécanismes ad-hocs									×	×					
	Détections adaptables					×	×	×								
	Mécanismes adaptables												×	×		

Figure 19. *Caractérisation des LCOs*

6. Remerciements

L'auteur tient à remercier Gul Agha, Jean Pierre Briot, Oscar Nierstrasz, François Pacull et André Schiper pour leur disponibilité.

7. Bibliographie

[AME 87] P. AMERICA, "POOL-T: A Parallel Object-Oriented Language", *Object-Oriented Concurrent Programming*, édité par A. Yonezawa et M. Tokoro, MIT Press, 1987, p. 199-220.

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

[ARC 94] J.P ARCANGELI., A. MARCOUX, C. MAUREL et P. SALLE, "La programmation concurrente par acteurs : le système PLASMA II", *Journal des Calculateurs Parallèles*, n°22, 1994, p. 123-155.

[BAR 89] J.G.P. BARNES, *Programming in ADA*, Addison-Wesley, 1989.

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

[BIR 73] G. BIRTWISTLE, O. DAHL, K. MYHRHAUG et K. NYGAARD, *SIMULA begin*, Petrocelli Charter, 1973.

[BOB 88] D.G. BOBROW, L.G. DEMICHIEL, R.P. GABRIEL, S.E. KEENE, G. KICZALES et D.A. MOON, "Common Lisp Object System Specification", *SIGPLAN Notices*, vol. 23, 1988.

[BRI 89] J-P. BRIOT, "Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment", *European Conference on Object Oriented Programming*, 1989, Cambridge University Press, p. 109-129.

[BRI 94] J-P. BRIOT, " Modélisation et classification de langages de programmation concurrente à objets : l'expérience Actalk", *Langages et Modèles à Objets*, INRIA/IMAG/PRC-IA, Grenoble, Octobre 1994.

[CAP 92a] 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.

[CAP 92b] R. CAPOBIANCHI, R. GUERRAOUI, A. LANUSSE et P. ROUX, "Active Objects on a Parallel Machine: A Case Study", *Technology of Object-Oriented Languages and Systems*, 1992, Prentice Hall, p. 207-216.

[CAR 89] M. CART, J. FERRIÉ et H. RICHIE, "Contrôle de l'exécution de transactions concurrentes", *Techniques et Sciences Informatiques*, vol 3, 1989.

[CAR 94] D. CAROMEL, "Programmation parallèle à objets : Eiffel//", *Journal des Calculateurs Parallèles*, n°22, 1994, p. 51-85.

[COR 87] A. CORRALDI et L. LEONARDI, "An environment Based on Parallel Objects: PO", *IEEE International Conference on Computers and Communication*, 1987, p. 253-257.

[COX 86] B.J. COX, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[DEC 91] D. DECOUCHANT, P. LE DOT. M. RIVEILL, C. ROISIN et X. ROUSSET DE PINA, "A Synchronisation Mechanism for an Object-Oriented Distributed System", *IEEE International Conference on Distributed Computing Systems*, 1991, p 152-159.

[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*, 1991.

[GAR 94] B. Garbinato, X. Défago, R. Guerraoui et K. Mazouni, "Abstractions pour la Programmation Concurrente dans GARF", *Journal des Calculateurs Parallèles*, n°22, 1994, p. 85-99.

[GOL 83] A. Goldberg et D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[GUE 92a] R. Guerraoui, R. Capobianchi, A. Lanusse et P. Roux, "Nesting Actions through Asynchronous Message Passing: the ACS protocol", *European Conference on Object Oriented Programming*, 1992, LNCS, Springer Verlag, n° 615, p. 170-184.

[GUE 92b] R. Guerraoui, *Programmation répartie par objets : études et propositions*, thèse de doctorat de l'université d'Orsay, 1992.

[GUE 92c] R. Guerraoui, "Dealing with Atomicity in Object-Based Distributed Systems", *ACM OOPS Messenger*, vol. 3, n° 3, 1992.

[Gue 93] R. Guerraoui, "Towards Modular Concurrency Control for Distributed Object Oriented Systems", *IEEE Future Trends in Distributed Computing Systems*, 1993, p. 240-247.

[GUE 94a] R. Guerraoui, O. Nierstrasz et M. Riveill, *Object-Based Distributed Programming*, LNCS, Springer Verlag, n° 791, 1994.

[GUE 94b] R. Guerraoui, "Composing Atomic Objects", *European Conference on Object Oriented Programming*, 1994, LNCS, Springer Verlag, n° 821.

[HER 87] M.P. Herlihy et J. Wing, "Avalon: language support for reliable distributed systems", *IEEE Fault-Tolerant Computing Symposiums*, 1987, p. 89-94.

[KAF 89] D. Kafura et K.H. Lee, "Inheritance in actor based concurrent object-oriented languages", *European Conference on Object-Oriented Programming*, 1989, Cambridge University Press.

[KAI 89] G.E. Kaiser, S.S. Popovitch, W. Hseush et S.F. Wu, "MELDing Multiple Granularities of Parallelism", *European Conference on Object Oriented Programming*, 1989, Cambridge University Press, p. 131-145.

[KER 78] B.W. Kernighan et D.M. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[KRA 90] S. Krakowiak, M. Meysembourg, V.H Nguyen, M. Riveill et C. Roisin, "Design and Implementation of an Object-Oriented Strongly Typed Language for Distributed Applications", *Journal of Object-Oriented Programming*, vol. 3, n° 3, 1990.

[LIS 79] B. Liskov et A. Snyder, "Exception Handling in CLU", *IEEE Transactions on Software Engineering*, vol. 5, n° 6, 1979.

[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.

[LAN 91] A. Lanusse et P. Roux, "Définition du langage acteur PRAL-1", *Rapport N/Réf. DTA/LETI/DEIN/SIR 91-46*, 1991.

[MAS 89] G. Masini, A. Napoli, D. Colnet, D. Léonard et K. Tombre, *Les langages à objets*, InterEditions, 1989.

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

[MOS 85] J.E.B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[NIE 87] O. Nierstrasz, "Active Objects in Hybrid", *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1987, p. 243-253.

[NIE 89] O. Nierstrasz, "A Survey of Object-Oriented Concepts", *Object-Oriented Concepts, Databases, and Applications*, édité par W. Kim et F.H. Lochovsky, Addison Wesley, 1989, p. 3-22.

[NIE 93] O. Nierstrasz, "Regular Types for Active Objects", *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1993, p. 1-15.

[PAP 92] M. Papathomas, *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*, thèse de doctorat de l'université de Genève, 1992.

[PAR 88] G.D. Parrington et S.K. Shrivastava, "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems", *European Conference on Object Oriented Programming*, 1988, LNCS, Springer Verlag, p. 234-249.

[SCH 86] A. Schiper, *Programmation concurrente*, Presses Polytechniques Romandes, Collection Informatique, 1986.

[STR 86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[TAU 93] K. Taura, S. Matsuoka et A. Yonezawa, "An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers", *ACM Symposium on Principles and Practices of Parallel Programming*, 1993, p. 218-228.

[WEG 87] P. Wegner, "Dimensions of Object-based Language Design", *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1987, p. 168-182.

[WEI 84] W. E. Weihl, *Specification and Implementation of Atomic Data Types*, thèse de doctorat du MIT, 1984.

[YOK 87] Y. Yokote et M. Tokoro, "Concurrent Programming in Concurrent Smalltalk", *Object-Oriented Concurrent Programming*, édité par A. Yonezawa et M. Tokoro, MIT Press, 1987, p. 129-158.

[YON 87] A. Yonezawa, E. Shibayama, T. Takada et Y. Honda, "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1", *Object-Oriented Concurrent Programming*, édité par A. Yonezawa et M. Tokoro, MIT Press, 1987, p. 50-90.