CHAPITRE 2

Modéliser le comportement

Ce chapitre présente un modèle de programmation concurrente fondé sur la notion de composant. Une première partie introduit la notion de composant et suggère son application à l'interaction. Une deuxième partie définit une sémantique opérationnelle des systèmes de composants. Une troisième partie associe aux systèmes de composants un modèle concurrent de haut niveau puis définit la notion de composant actif ou « acteur », ayant pour objectif la modélisation du comportement individuel des éléments, autonomes ou indépendants, d'un système.

2.1 La programmation par composants

Le terme « composant » est fréquemment utilisé par un grand nombre d'industriels du génie logiciel qui l'emploient chacun sous une signification vague et, *a priori*, différente. Son irruption dans la terminologie informatique courante correspond à une problématique majeure qui n'est encore que partiellement résolue : définir une architecture modulaire « orientée objets » indépendante d'un langage sous-jacent.

Composants multimédias : des objets sans classe ni prototype

Cette problématique est proche de la démarche de formalisation de l'interaction présentée en introduction dans la mesure ou la plupart des modèles graphiques, musicaux ou d'interface utilisateur courants sont « orientés objets ». Par exemple, l'environnement de production multimédia Apple Media Tool [AmarBureSoqu93], le logiciel de production musicale Audio Logic [Emag93] ou le système de documents composites OpenDoc [Open94] reposent sur une terminologie et des concepts proches de ceux des langages de programmation à objets [DahlNyga66, Gold84] ou à prototypes [Lieb86, UngaChamChan91]. On sépare toujours :

- d'une part, une notion de composant élémentaire capable de réunir en son sein du comportement et des données;
- d'autre part, une structure, souvent figée, qui définit les relations « naturelles » pouvant exister entre composants (voir table 1 page 10).

TABLE 1 Exemples de composants et de structures

Média	Composant		Structure
	Données	Comportement	
Musique	Séquence de notes	Asservissement à une base de temps	Hiérarchie de bases de temps
3D	Surface triangulée	Réponse à un choc	Hiérarchie de changements de repère
Interface utilisateur	Nom et dimension d'un bouton	Réponse à un clic de souris	Empilement de cadres, boutons et champs

La notion de composant correspond donc au plus grand dénominateur commun de l'ensemble des modèles multimédias. C'est un objet débarrassé de toute structure : classe, prototype, héritage…

Des solutions industrielles

Les solutions proposées actuellement par l'industrie pour faire interagir des composants développés séparément reposent sur deux entités :

- **Le gestionnaire de composants**. Il s'agit d'une extension du système d'exploitation permettant de charger dynamiquement des objets ou des classes définis dans un modèle à objets uniforme sous-jacent[1].
- **Le cadre de travail** (*framework*). Il incite à construire une application ou un document composite en commençant par définir de nouvelles classes par héritage de classes existantes puis en utilisant des mécanismes prédéfinis de composition et d'interaction pour assembler des objets (voir table 2 page 11).

TABLE 2 Exemples de gestionnaires de composants et cadres de travail

Gestionnaire de composants	Cadre de travail	Exemples de composants
Component Manager (MacOS)	Quicktime	Compresseur vidéo Synthétiseur sonore Base de temps
System Object Model (IBM)	OpenDoc	Correcteur orthographique Butineur WWW Editeur de dessin

Le défaut majeur de cette approche résulte de la trop grande rigidité des cadres de travail qui ont été proposés jusqu'à présent. Par exemple, l'utilisation automatique d'un correcteur orthographique dans un document OpenDoc multilingue est problématique dans la mesure où la seule forme d'interaction entre un composant de texte et un composant de correction d'orthographe est une interaction uniforme donc ne faisant intervenir qu'une seule langue. Un éditeur de logiciel [Yone90], confronté à cette limitation inhérente au caractère figé des formes d'interaction prévues par OpenDoc, a donc dû réclamer aux concepteurs de ce cadre de travail, l'intégration future d'interactions tenant compte de la langue!

1. La notion de bibliothèque partagée ne permet pas d'obtenir ce résultat. En effet, l'implantation du mécanisme de sélection de méthode d'un langage à objets statique comme C++ dépend du compilateur. Deux classes compilées séparément risquent donc de ne pas être compatibles au moment de l'exécution.

SOM : des composants réflexifs

La rigidité des cadres de travail actuels est souvent le pendant d'une trop grande uniformité du modèle à objets de certains gestionnaires de composants comme *Component Manager* [Appl93b] ou *Common Object Model*[1]. En effet, ces gestionnaires proposent une sémantique unique et non paramétrable de l'instanciation et de l'héritage. Si cette sémantique convient aux cadres de travail fermés pour lesquels le gestionnaire a été développé, elle devient souvent inadaptée à des cadres de travail mettant en oeuvre des formes d'interaction qui n'auraient pas été envisagées.

Le gestionnaire SOM[2] (*System Object Model*) propose de résoudre ce manque de flexibilité en faisant appel à des notions avancées de programmation à objets issues de travaux menés sur la méta-programmation et la réflexion [DanfForm89]. A la différence de ses prédécesseurs, SOM permet à chaque classe de définir une sémantique de l'héritage et de l'instanciation qui lui est propre. Ainsi, plusieurs modèles à objets peuvent co-exister au sein d'un même système de composants.

Acteurs-observateurs : un cadre de travail réflexif

Le modèle Acteur-Observateur présenté dans cette thèse peut être compris comme une application des principes de méta-programmation et de réflexion à la définition d'un cadre de travail adapté à l'interaction multimédia. L'objectif que nous nous sommes fixé est de permettre la co-existence d'interactions en nombre et en type quelconques au sein d'un même cadre de travail. Le principe proposé est simple : une interaction est un composant capable de modifier *a posteriori*

1. Gestionnaire de composants proposé par Microsoft et servant de fondement au cadre de travail OLE.

2. Gestionnaire de composants proposé par IBM puis adopté par Apple, IBM et leurs partenaires des CI Labs pour l'implantation du cadre de travail OpenDoc.

la sémantique du modèle à objets sous-jacent d'autres composants.

2.2 Une sémantique des systèmes de composants

La présentation du modèle acteur-observateur repose sur la notion de composant qui correspond à une pratique industrielle encore naissante[1]. C'est pourquoi nous allons développer ici une terminologie précise reposant sur une sémantique opérationnelle originale[2] des systèmes de composants. Cette sémantique s'appuie sur l'adoption d'un langage commun correspondant au choix d'un noyau de primitives permettant de décrire les accès, la synchronisation et les communications dans un système de composants.

2.2.1 Une approche anthropomorphique de la programmation

Avant de proposer une sémantique du composant isolé puis une sémantique de la communication dans les systèmes de composants, commençons par illustrer ce qu'est un composant par analogie anthropomorphique.

Une vision réductrice de l'homme conduit à distinguer l'état (âge, taille, poids, couleur des yeux etc.) du caractère (comportement, réactions etc.). La société est comprise comme un ensemble d'hommes qui interagissent par échange de messages. Lorsqu'un homme parle à un autre homme, il lui envoie un message vocal; lorsqu'un homme touche un autre homme, c'est un message tactile; lorsqu'un homme réfléchit, il s'envoie des messages à lui-même. L'état d'un homme évo-

1. En 1996, le JIPDEC (Japan Information Processing Development Center) a choisi de lancer un comité consultatif sur l'applicabilité industrielle de la programmation par composants. C'est pourquoi on peut estimer qu'il faudra attendre cinq à dix ans pour assister à l'adoption de la programmation par composants en informatique de gestion (celle qui fait de NEC, Fujitsu, Hitachi 3 des 5 premiers informaticiens du monde).

2. L'originalité du modèle présenté est de définir directement une sémantique des systèmes de composants, indépendante de tout langage ou de tout modèle pré-existant. Cette sémantique s'inspire de notions de programmation développées dans les années 70 et 80 par la communauté scientifique de l'intelligence artificielle et de l'informatique avancée.

lue alors en fonction des messages qu'il reçoit et des réactions induites par ceux-ci : lorsqu'un homme reçoit un aliment, son poids augmente parfois.

A l'instar de l'homme, aggrégation d'état et de caractère, un composant est une entité inséparable contenant des données et des programmes (voir figure 2 page 14). A l'instar d'une société d'hommes, un système de composants est un ensemble de composants capables de communiquer par échange de messages. La réception d'un message déclenche la sélection puis l'exécution du programme approprié et conduit éventuellement à une modification de ses données.

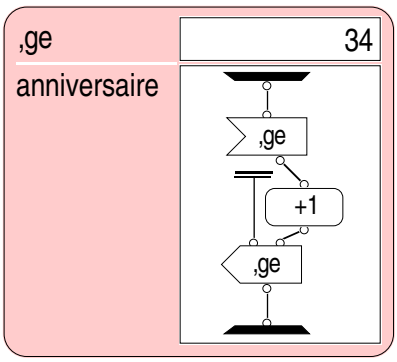
2.2.2 Le composant isolé

L'analogie anthropomorphique conduit à une définition formelle de la notion de composant .

Définition 1 : Un composant C est un ensemble de couples (i,e), appelés champs, où i est un identificateur et e une expression d'un langage algébrique L.

FIGURE 2

Un composant modélisant un individu contient deux champs : *âge* et *anniversaire*. L'envoi du message *anniversaire* au composant incrémente l'âge



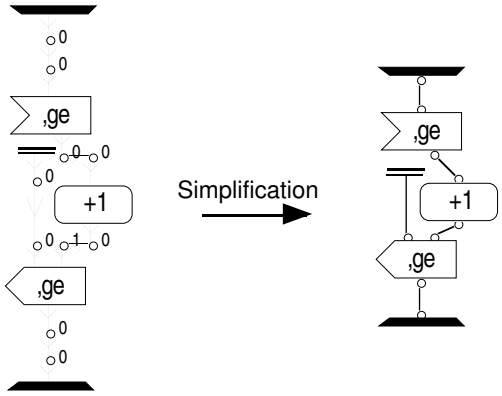
Syntaxe graphique

Si l'on identifie les expressions de L à leur syntaxe abstraite, une expression de L est un graphe orienté étiqueté. C'est

pourquoi, les expressions de ***L*** seront désormais représentées graphiquement sous forme d'un graphe comprenant neuf types de noeuds correspondant aux primitives sémantiques de ***L*** (Valeur, Paramètre, Lecture, Ecriture, Message, Entrée, Sortie, Clôture, Synchronisation) et trois types de données de base (Vide, Entier, Identificateur) pour les étiquettes (voir table 3 page 16).

FIGURE 3

Représentation simplifiée de l'expression *anniversaire* du composant *individu*. On a omis les flèches des arêtes en prenant pour convention qu'elles sont toujours orientées vers le bas.



Pour des raisons évidentes de lisibilité (voir figure 3 page 15), la plupart des étiquettes ainsi que certaines arêtes seront omises lors du dessin d'un graphe de ***L***, les conventions[1] de lecture étant :

- Une arête entre un noeud paramètre et tout autre noeud est représentée en plaçant le noeud paramètre dans une position tangente à l'autre noeud.
- L'orientation des arêtes est du haut vers le bas.
- Les étiquettes à valeur entière des noeuds paramètres partant vers ou arrivant d'un autre noeud qui ne soit ni un paramètre ni une valeur, sont numérotées incrémentalement à partir de 0 en allant de gauche à droite.

1. Ces conventions s'inspirent librement du langage graphique Prograph [Prog93].

TABLE 3 Primitives graphiques d'un langage de composants

Primitive graphique	Etiquette	Sémantique
Valeur	Vide	Auto-référence
	Entier	Type de donnée de base pour l'arithmétique et l'accès indexé aux champs d'un composant
	Identificateur	Type de donnée de base pour l'accès paramétré aux champs d'un composant
Paramètre	Entier	Index d'un paramètre en entrée ou en sortie d'un message
Lecture	Identificateur ou entier	Lecture du champ d'un composant
Ecriture	Identificateur ou entier	Ecriture du champ d'un composant
Message	Identificateur ou entier	Sélection puis exécution d'une méthode d'un composant
Entrée	Vide	Entrée des paramètres d'exécution d'un programme
Sortie	Vide	Résultat de l'exécution d'un programme
Clôture	Vide	Passage d'une expression comme paramètre
Synchronisation	Vide	Contrainte d'ordre d'évaluation

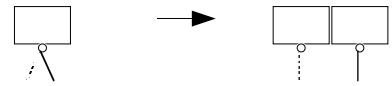
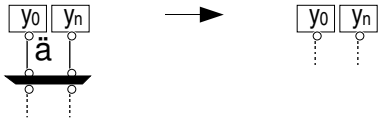
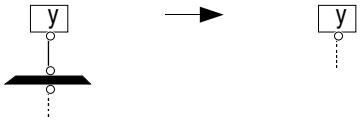
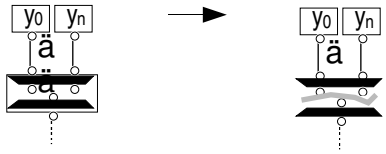
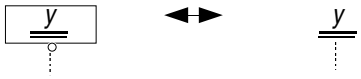
Règles de réécriture

Pour définir la sémantique opérationnelle de *L*, il suffit de :

- proposer un ensemble de règles de réécriture des graphes de *L*;
- associer à chacune de ces règles l'évolution correspondante de l'état d'une machine abstraite.

Dans le cas du composant isolé, l'état d'une machine abstraite est défini par le couple *(C,e)* où *C* est un composant et *e* une expression de *L*. Par souci de simplicité, nous ne présenterons ici que les principales règles de réécriture (voir table 4 page 17)[1]. Les expression valides *L'* de *L* sont les expressions ayant pour expression réduite une valeur ou une clôture (voir figure 4 page 21).

TABLE 4 Règles de réduction d'un langage de composants

Règle	Paramètres	Action
Branchement 	Aucun	Aucune
Entrée 	$y_0 \ldots y_n$ sont des expressions de L	Aucune
Sortie 	y est une expression de L	Aucune
Ouverture 	$y_0 \ldots y_n$ sont des expressions de L	Aucune
Equivalence 	y est une valeur	Aucune

1. Si l'on devait définir toutes ces règles, une partie non négligeable de celles-ci concernerait l'implantation de l'arithmétique ainsi que de tests permettant de calculer toute fonction récursive [Ster86].

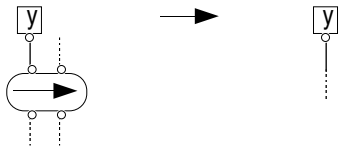
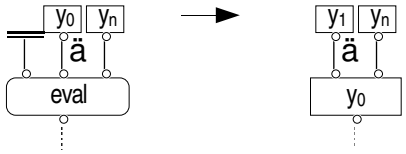
Règle	Paramètres	Action
Synchronisation 	*y* est une expression de *L*	Aucune
Evaluation 	$y_0 \ldots y_n$ sont des expressions de *L*	Aucune

TABLE 5 Introduction de primitives arithmétiques dans un langage de composants

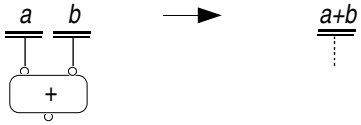
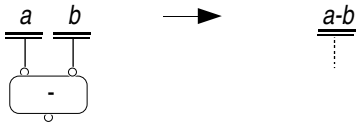
Règle	Paramètres	Action
Addition 	*a* et *b* sont des entiers	Calcul de la somme de deux entiers
Soustraction 	*a* et *b* sont des entiers	Calcul de la différence de deux entiers
etc.	etc.	etc.

TABLE 6 Tests dans un langage de composants

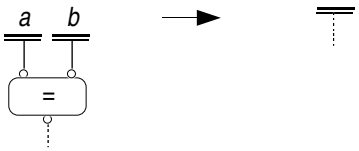
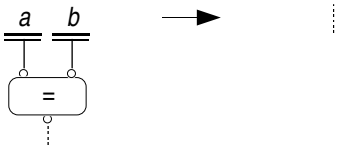
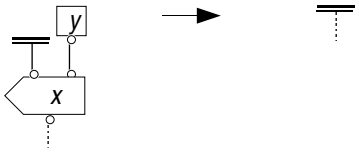
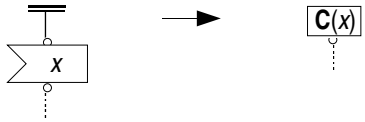
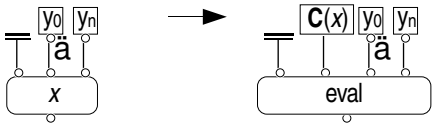
Règle	Paramètres	Action
Vrai 	*a* et *b* sont deux valeurs égales	Test d'égalité
Faux 	*a* et *b* sont deux valeurs différentes	Test d'égalité

TABLE 7 Règles spécifiques à la sémantique du composant isolé

Règle	Paramètres	Action
Ecriture 	*x* est un entier ou un identificateur; *y* est une expression de *L*	Modification du champ *x* du composant *C* $C \rightarrow C \oplus (x, y)$
Lecture 	*x* est un entier ou un identificateur	Lecture du champ *x* du composant *C*
Message 	*x* est un entier ou un identificateur	Lecture du champ *x* du composant *C*

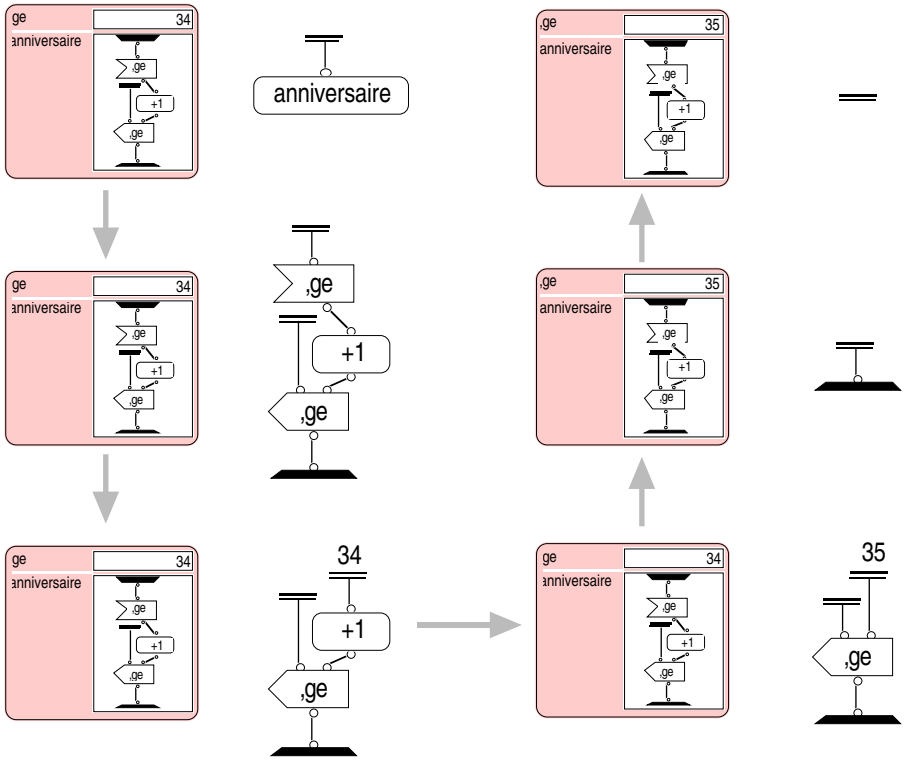
Particularité du modèle

Dans le cas d'un composant isolé, cinq particularités sémantiques doivent retenir notre attention car elles permettront plus loin d'étendre la notion de composant :

- **La valeur d'auto-référence ou valeur vide** : elle correspond à un pointeur du composant sur lui-même ayant pour effet l'exécution des primitives de lecture ou d'écriture.
- **L'accès aux champs non définis** : il peut provoquer, en fonction de la sémantique de l'opérateur $\oplus$ que l'on veut donner au langage, une erreur ou la création d'un champ vide.
- **La notion de clôture** : elle permet de manipuler des expressions de L comme paramètres des primitives de L.
- **L'absence de pile** est compensée par la récursivité propre au modèle d'évaluation de clôture. Elle induit cependant une forme de non déterminisme dû aux possibilités de réduction parallèle au sein d'une même expression. Dans le cas des langages à objets séquentiels courants (Smalltalk, C++, CLOS…) un ordre de réduction est fixé et l'évaluation d'une expression par réductions successives du graphe peut être obtenue au moyen d'une pile[1].
- **Un mécanisme de synchronisation** permet de résoudre les problèmes éventuels dûs au non-déterminisme de la réduction parallèle.

1. Dans les langages de composants réflexifs, le miroir permet souvent d'accéder à la pile de réduction et de simuler ainsi des réductions parallèles (voir paragraphe 2.4.1 page 33).

FIGURE 4 Réécritures successives d'un envoi du message anniversaire au composant individu.



2.2.3 Système de composants

La notion de système de composants introduite ici consiste à permettre à un ensemble de composants isolés de s'identifier et de communiquer par envoi de messages. Il s'agit donc d'une notion de système dépourvue de toute forme de structure explicite.

Machine à réduction d'états

Afin de mettre en oeuvre un mécanisme simple de communication dans un système de composants, on introduit un nouveau type de valeur, les pointeurs (voir table 8 page 22), et un nouveau modèle de machine abstraite construit comme un triplet (S,c,e) où S est un ensemble de composants représentant l'état du système, c un pointeur sur un composant (appelé « la cible ») et e une expression de L. Pour des

besoins d'évaluation, on introduit aussi un nouveau type de noeud appelé contexte. Ce noeud est étiqueté par un pointeur.

L'ensemble des contextes imbriqués forme à chaque instant un arbre qui correspond à l'évaluation parallèle d'envois de messages (voir figure 5 page 22).

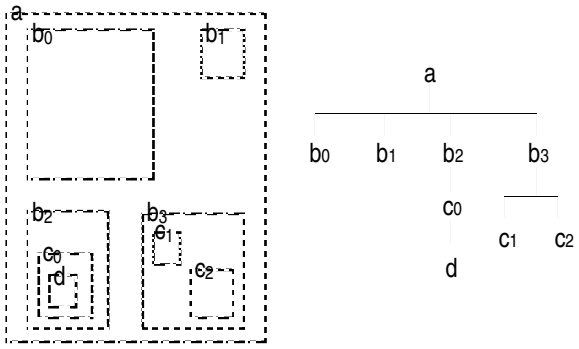
TABLE 8 Le pointeur est un nouveau type de valeur

Primitive graphique	Etiquette	Sémantique
Valeur	Pointeur	Référence sur un composant du système *S*
Contexte	Pointeur	Spécification du contexte d'évaluation d'une expression.

Définition 2 : un contexte fils de *c* est un contexte différent de *c* et appartenant au sous-arbre de racine *c* des contextes imbriqués. Un contexte parent est un contexte dont *c* est un fils.

Définition 3 : une clôture libre est une clôture dont le seul noeud paramètre en sortie n'est relié qu'à des noeuds paramètres de contextes parents.

FIGURE 5 Exemple de contextes imbriqués (à gauche) et représentation par un arbre (à droite). Chaque niveau d'imbrication correspond à une lettre de l'alphabet.



Extension de la sémantique du composant isolé

L'ensemble des règles de réécriture hormis celles touchant à l'envoi de messages correspondent à celles du composant isolé pour lesquelles C est le composant pointé par la cible c. Avant toute application d'une règle de réduction, la valeur de c est calculée **systématiquement** ; elle est égale à l'étiquette du contexte auquel appartient le noeud de e que l'on souhaite réduire. Il est donc nécessaire pour évaluer une expression e de L , de la placer d'abord dans un contexte défini.

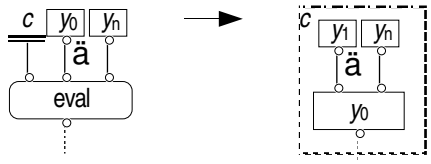
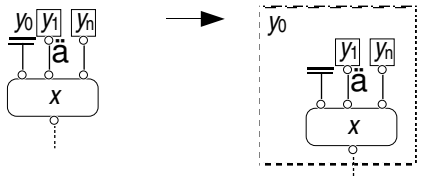
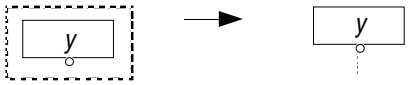
Sémantique de l'envoi de message

On définit la sémantique de l'envoi synchrone[1] de message par empilage/dépilage de contexte. Lorsque, lors d'une réduction, le paramètre 0 d'un envoi de message a pour valeur un pointeur, on décompose la réduction en trois étapes :

- **Empilement de contexte**. Le paramètre 0 est choisi comme étiquette du nouveau contexte d'évaluation.
- **Réduction récursive**. La réduction se poursuit dans le nouveau contexte d'évaluation c'est-à-dire en opérant sur le composant pointé par le paramètre 0.
- **Désempilement de contexte**. Les clôtures libres d'un contexte sont évacuées. Tout contexte vide est supprimé.

1. L'envoi de message est qualifié de synchrone car, dans un modèle séquentiel d'évaluation des expressions, l'ordre d'imbrication des contextes induit un ordre temporel d'évaluation des envois de message. On réduira un envoi de message dans un contexte fils avant de réduire un envoi de message dans un contexte père.

TABLE 9 Réécriture de l'envoi de message synchrone

Règle	Paramètres	Action
c-évaluation 	*c* est un pointeur. $y_0 \ldots y_n$ sont des expressions de *L*	Empilement de contexte
c-message 	y_0 est un pointeur. $y_1 \ldots y_n$ sont des expressions de *L*	Empilement de contexte
Liberté 	*y* est une expression de *L*	Suppression de contexte

2.3 De la réflexion dans les systèmes de composants

La sémantique définie précédemment ne permet pas la création ni la suppression en cours d'évaluation de champs d'un composant ou de composants d'un système. Cette propriété, appelée dynamicité, peut être obtenue en ajoutant au système un miroir. Il s'agit d'un composant unique, noté *mirror*, pour lequel s'appliquent des règles de réécriture spécifiques. L'envoi par un composant d'un message au miroir permet au composant, par réflexion, de se donner une image de lui-même ou du système et, en la modifiant, de se modifier lui-même.

2.3.1 Un miroir pour se regarder

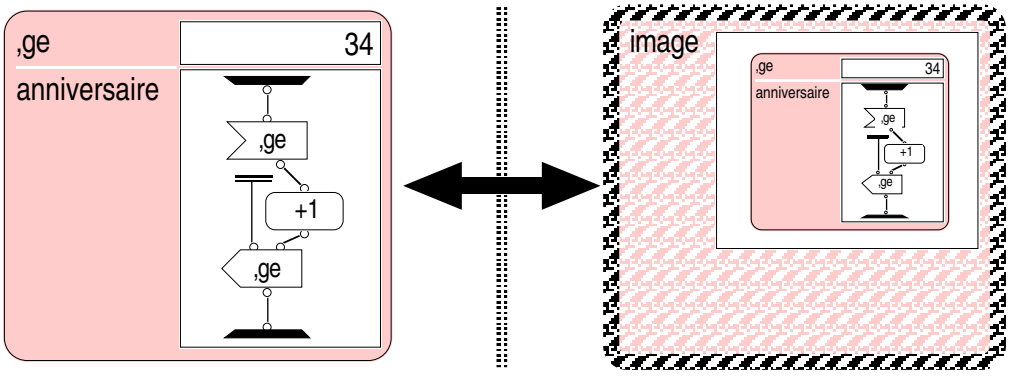
Une premier objectif du miroir est de permettre à un composant d'en savoir plus sur lui-même : quels sont ses champs, quels sont les autres composants du système etc.

Composants auto-représentables

Pour obtenir ce résultat, on construit un composant dont les champs sont une représentation des composants communiquant avec lui. Par exemple, lorsqu'un composant *c* envoie un message *getFields* au miroir, celui-ci renverra une collection d'identificateurs correspondant à l'ensemble des identificateurs des champs de *c* et non du miroir. On dit alors que le composant *c* a pu obtenir une auto-représentation partielle par réflexion dans le miroir.

La mise en oeuvre d'un miroir consiste à définir un jeu de règles de réécriture qui ne s'appliquent que dans le contexte du miroir. Ainsi, de façon similaire à la définition de primitives arithmétiques, la réduction du message *getFields* pourrait consister à renvoyer un pointeur sur une collection d'identificateurs placés dans les champs indexés d'un composant.

FIGURE 6 Le miroir permet au composant individu d'accéder à sa propre représentation en lui fournissant une image de lui-même, qu'il peut éventuellement modifier pour se «maquiller».



Protocoles du miroir

Pour achever la définition d'un miroir, il faut choisir la suite de messages permettant aux composants de s'auto-représenter. Cette suite est divisée en centres d'intérêt appelés protocoles (voir table 10 page 26) :

- **Le protocole de champ**. Il permet l'accès aux champs en lecture et en écriture, la création ou la suppression de champs et l'obtention de la liste des champs.
- **La protocole de composant**. Il permet de créer de nouveaux composants du système[1].
- **Le protocole de réduction**. Il permet de connaître le contexte de réduction courant[2].

TABLE 10 Protocoles du miroir

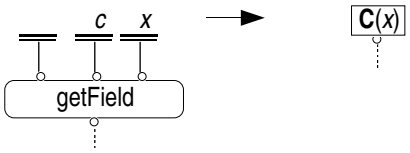
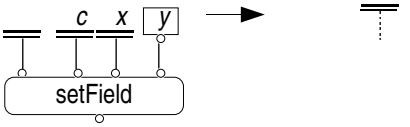
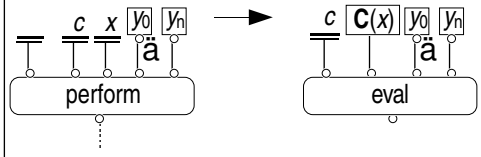
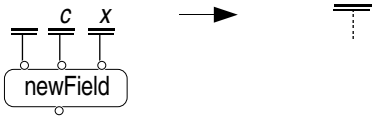
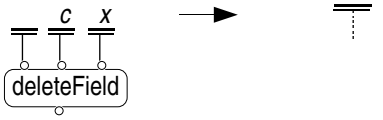
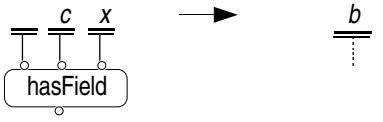
Protocole	Méthodes	Opération
Champ	hasField	Teste l'existence d'un champ du composant
	newField	Ajout d'un champ au composant
	deleteField	Suppression d'un champ au composant
	getField	Accès en lecture d'un champ d'un composant
	setField	Accès en écriture d'un champ d'un composant
	perform	Envoi de message
Composant	new	Création d'un composant
Réduction	context	Lecture du contexte courant

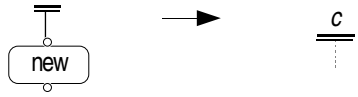
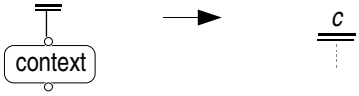
Chaque méthode est alors traduite en une nouvelle règle de réduction, valable uniquement lorsque, lors d'une réduction, le contexte du noeud à réduire est égal au pointeur du miroir. On parle alors de réduction dans le contexte du miroir.

1. La suppression de composants non référencés est assurée par un glaneur de cellules. Le protocole de composants peut aussi être étendu afin de permettre aux composants de connaître à tout instant la liste des composants du système.

2. On pourait étendre ce protocole à l'accès et l'édition de l'ensemble de l'arbre de contextes de réduction. Dans une sémantique séquentielle à pile, cela correspondrait à autoriser l'accès et l'édition de la pile. Plus généralement, le protocole de réduction peut être étendu pour permettre l'accès et la mise à jour de l'expression en cours de réduction (voir table 14 page 33).

TABLE 11 Réductions du miroir évaluées dans le contexte du miroir

Règle	Paramètres	Action
getField 	*c* est un pointeur; *x* est un entier ou un identificateur	Lecture du champ *x* du composant pointé par *c*
setField 	*c* est un pointeur; *x* est un entier ou un identificateur; *y* est une expression de *L*	Modification du champ *x* du composant pointé par *c* $C \rightarrow C \oplus (x, y)$
perform 	*c* est un pointeur; *x* est un entier ou un identificateur; $y_0 \ldots y_m$ sont des expressions de *L*	Lecture du champ *x* du composant pointé par *c*
newField 	*c* est un pointeur; *x* est un identificateur	Création d'un champ *x* dans le composant pointé par *c*
deleteField 	*c* est un pointeur; *x* est un identificateur	Suppression du champ *x* du composant pointé par *c*
hasField 	*c* est un pointeur; *x* est un identificateur; *b* est 0 si le champ *x* n'existe pas, 1 s'il existe	Test d'existence du champ *x* du composant pointé par *c*

Règle	Paramètres	Action
new 	*c* est un pointeur sur le composant nouvellement créé	Création d'un composant
context 	*c* est un pointeur sur le composant cible	Lecture du contexte de réduction

2.3.2 Un miroir pour se maquiller

Un deuxième objectif du miroir est de permettre à un composant de modifier son état ou celui d'un autre composant.

Composants auto-modifiables

Supposons qu'un composant souhaite supprimer un certain nombre de ses champs et en ajouter d'autres. Il lui suffira d'envoyer au miroir le message *newField* ou *deleteField* avec pour paramètres les identificateurs voulus (voir table 11 page 27).

2.3.3 Un miroir sémantique déformant

Un troisième objectif du miroir est de permettre la redéfinition de la sémantique du langage. Cette caractéristique provient directement de la définition donnée du miroir.

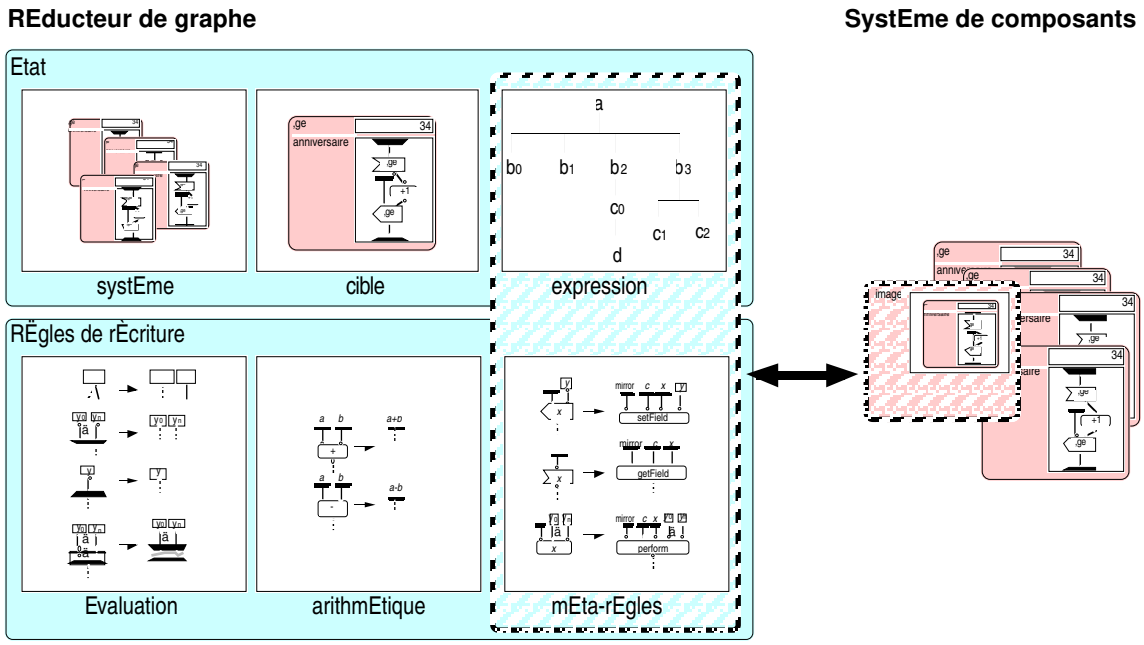
Auto-réflexion et mise en abîme sémantique

Considérons la réflexion d'un miroir, considéré comme un composant standard, sur lui-même, au moyen de règles de réduction spécifiques. Calculons, par exemple, l'envoi du message *getField* avec pour paramètre *getField*. Cette expression devrait retourner comme valeur une expression de L correspondant à un champ du miroir nommé *getField*.
Supposons que l'on puisse modifier ce champ par l'envoi

d'un message *setField*. Il semblerait cohérent que cette modification entraîne alors un changement de la sémantique initiale de la lecture de champ de manière à la faire correspondre à l'image calculée par le miroir.

En somme, l'auto-réflexion du miroir nous conduit à envisager le miroir non seulement comme un moyen de représentation et de modification dynamique des composants du système mais aussi comme une représentation partielle, au sein du système, des règles de réécriture de *L*.

FIGURE 7 Le miroir constitue une représentation partielle de la sémantique d'un langage



Rompre la mise en abîme : les méta-règles de réécriture

La mise en oeuvre de l'auto-réflexion du miroir est obtenue ici par l'ajout d'un jeu unique de règles de réécriture[1], appelées méta-règles. Ces règles traduisent les primitives de base du

1. L'approche choisie ici ne considère qu'un seul miroir pour tout le système. D'autres approches de la réflexion consistent à introduire une hiérarchie de miroirs dont le sommet correspond aux primitives du langage et la base correspond aux sémantiques obtenues par raffinement successif de la sémantique primitive initiale du langage.

langage (lecture, écriture, message) en un envoi de message au miroir. Elles remplacent les règles de réécriture introduites précédemment (lecture, écriture, message), à l'exception de la règle d'évaluation (voir table 5 page 18) et de *c*-évaluation (voir table 9 page 24).

Pour éviter une récursion infinie, elles ne sont pas appliquées, par définition de la sémantique, dès que deux contextes miroir ont été empilés consécutivement à l'extrémité de la branche courante de l'arbre de contexte. Dans ce cas, et dans ce cas seulement, les règles de réduction du miroir précédemment introduites sont appliquées (voir table 11 page 27).

TABLE 12 Règles de réécriture dans un contexte standard. Ces règles traduisent les primitives du langage en envoi de message au miroir.

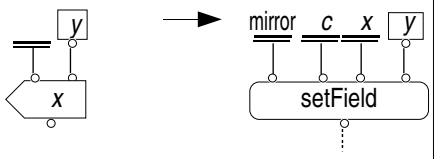
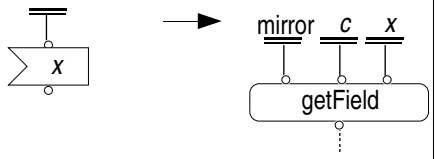
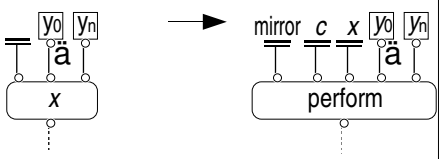
Règle	Paramètres	Action
Ecriture 	*c* est un pointeur sur la cible courante. *x* est un identificateur ou un entier. *y* est une expression de *L*	Lecture de la cible courante
Lecture 	*c* est un pointeur sur la cible courante. *x* est un identificateur ou un entier	Lecture de la cible courante
Message 	*c* est un pointeur sur la cible courante. *x* est un identificateur ou un entier. $y_0 \ldots y_n$ sont des expressions de *L*	Lecture de la cible courante

TABLE 13 Métarègle du miroir. Cette règle autorise l'empilement du contexte miroir lors d'un envoi de message.

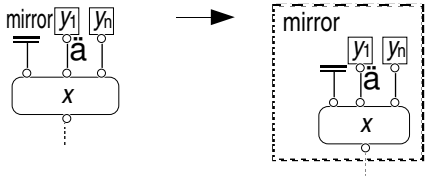
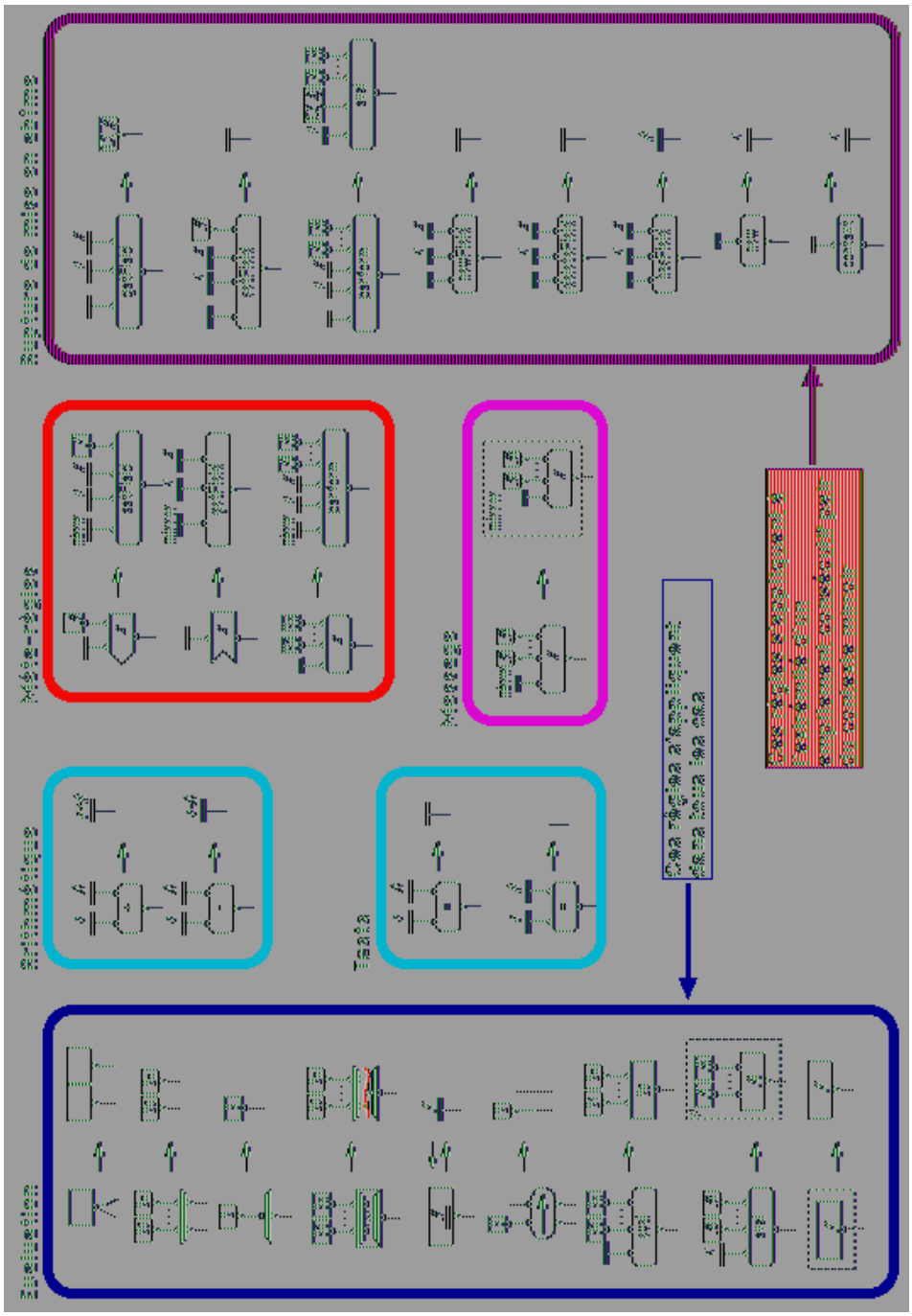
Règle	Paramètres	Action
Message 	x est un identificateur ou un entier. $y_0 \ldots y_n$ sont des expressions de L	Empilement du miroir

FIGURE 8 Vue d'ensemble des règles de réécriture définissant une sémantique réflexive des systèmes de composants



2.4 Les acteurs : des composants actifs distribués

La réflexion dans les langages de composants est une caractéristique clef permettant d'étendre la sémantique du langage par l'introduction de métaphores de haut niveau : programmation concurrente, calcul distribué et langages d'acteurs.

2.4.1 Introduction de la concurrence

L'introduction de la concurrence dans un langage de composants consiste à réduire simultanément plusieurs expressions. Elle peut être simulée par réflexion, à condition d'étendre préalablement le protocole du miroir (voir table 14 page 33). Le principe retenu ici est le partage du temps.

TABLE 14 Extension du protocole du miroir

Protocole	Méthodes	Opération
Réduction	getExpression	Lecture de l'expression en cours de réduction
	setExpression	Mise à jour de l'expression en cours de réduction

Scheduler de tâches

Un composant, appelé *scheduler*, maintient en permanence une liste de composants, appelés tâches, dans lesquels sont stockés des instantanés de l'état de la machine abstraite (expression en cours de réduction). Lors d'une réduction de graphe, le *scheduler* peut à tout instant interrompre le processus, stocker dans une tâche l'état courant de la machine abstraite et poursuivre le processus de réduction après avoir mis à jour l'état de la machine abstraite à une valeur précédemment stockée dans une autre tâche.

Granularité du procédé

L'interruption d'un processus de réduction est obtenue en plaçant dans le champ *perform* du miroir un compteur de nombre d'appels associé à la tâche courante. La valeur du compteur exprime un temps de réduction associé à une tâche. Lorsque

ce compteur dépasse un nombre fixé, un basculement de tâche est opéré. La granularité de ce mécanisme a pour limite le temps d'éxécution et le nombre (incertain) de réductions élémentaires d'une évaluation de la méthode *perform* du miroir[1].

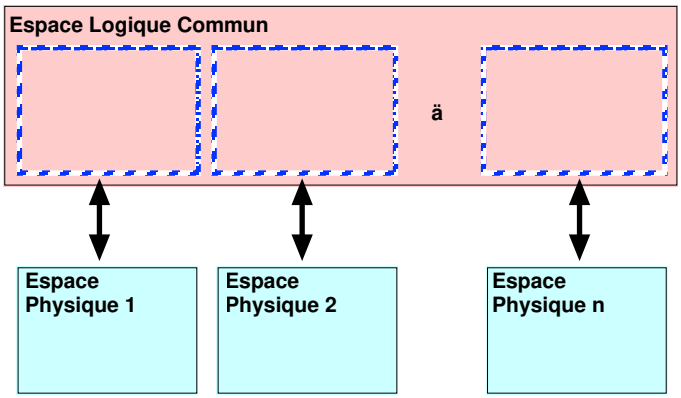
2.4.2 Composant distribué

La réflexion permet aussi d'introduire le calcul distribué dans un langage de composants.

Systèmes à mémoire distribuée

Un modèle de calcul distribué couramment utilisé consiste à considérer qu'un ensemble de systèmes répartis sur des machines et des mémoires physiques distinctes partagent virtuellement un même espace mémoire, appelé espace logique (voir figure 9 page 34).

FIGURE 9 Distribution de l'espace logique sur n espaces physiques



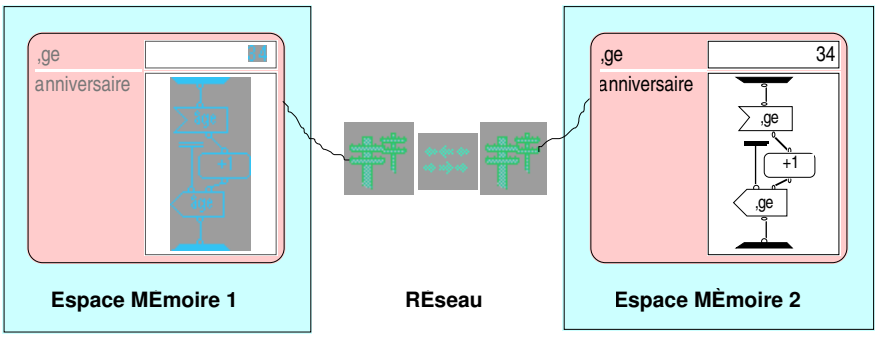
1. Dans la pratique, la concurrence est implantée directement au niveau du système d'exploitation ou à celui de la machine abstraite, par un mécanisme de pile et de fenêtre de registres.

Accès aux champs par le réseau : les composants fantômes

Lors d'une migration d'un composant vers un espace physique distant et relié par un réseau de communication, une copie fantôme est conservée sous la forme d'un composant vide à l'exclusion de l'adresse réseau à laquelle s'est effectuée la migration.

Chaque accès aux champs de la copie fantôme est alors converti en un accès réseau grâce à une définition adéquate des méthodes *getField* et *setField* du miroir (voir figure 10 page 35).

FIGURE 10 Un composant fantôme (à gauche) accède à ses champs, stockés dans un espace physique distinct (à droite) grâce à un accès au réseau.



2.4.3 Composant actif

Afin de donner à la notion de concurrence un cadre conceptuel de plus haut niveau que la réduction simultanée de graphe on introduit la notion de composant actif ou acteur pour représenter une entité active autonome. Le formalisme présenté ci-dessous s'inspire librement du système Actalk [Brio94a, Brio94b].

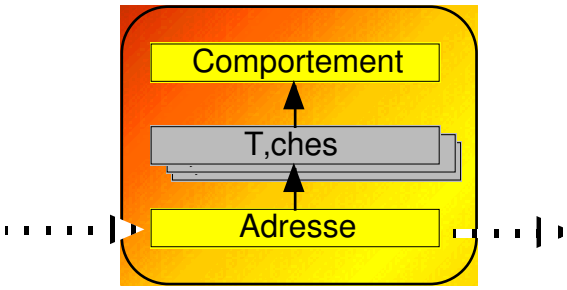
Décomposition fondamentale

Un acteur est la réunion de trois composants passifs correspondant à une analyse de la concurrence selon trois points de vue complémentaires :

- le **comportement** : il définit la réponse d'un composant actif aux messages qui lui sont envoyés par d'autres composants ou par lui-même;
- la **communication** : un composant, appelé adresse, définit la façon dont sont traités les messages reçus par un composant actif et s'ils doivent donner lieu à une réponse immédiate ou future;
- l'**autonomie** : une tâche définit le degré de concurrence dont dispose un composant pour réduire des expressions.

Cette partition offre l'avantage d'une grande modularité en autorisant la réunion et la combinaison dans un même environnement de plusieurs sémantiques de la concurrence.

FIGURE 11 Décomposition d'un composant actif en composants élémentaires



Sémantique de l'envoi de message

L'envoi de message à un composant actif est alors décomposé en plusieurs étapes :

- Un envoi de message synchrone à l'adresse du composant actif permet de stocker dans une queue de priorités le message et ses paramètres.
- En fonction des méthodes définies dans l'adresse, la tâche correspondant à l'envoi de message se poursuit (envoi de message sans réponse) ou est arrêtée (envoi de message avec réponse immédiate ou ultérieure).
- Lorsqu'une tâche est achevée ou lorsque la queue de messages n'est plus vide, une nouvelle tâche est créée pour

évaluer le premier message de la queue. En fonction de la sémantique désirée, les messages d'une queue pourront être évalués séquentiellement ou simultanément.

La mise en oeuvre de cette décomposition consiste donc à donner une définition adéquate de la méthode ***perform*** du miroir pour que tout envoi de message à une adresse conduise à l'insertion du message dans la queue des messages du composant actif.