Les Composants Logiciels : Evolution technologique ou nouveau paradigme ?

Frédéric Peschanski, Thomas Meurisse, Jean-Pierre Briot

Laboratoire d'Informatique de Paris 6
Thème OASIS

{Frederic.Peschanski, Thomas.Meurisse, Jean-Pierre.Briot}@lip6.fr

Résumé :

Nous proposons dans cet article une analyse de la notion de composant logiciel, très en vogue à l'heure actuelle. Deux définitions préliminaires sont présentées, l'une provenant du thème de recherche des architectures logicielles, l'autre plutôt issue du monde industriel : les composants binaires. Nous débattons des spécificités des approches basées sur les composants (par rapport aux approches objet en particulier), que ce soit dans les technologies industrielles ou dans nos propres travaux de recherche dans le domaine. Une proposition de notion identitaire de composant logiciel, associée à un cahier des charges minimal, est élaborée. Nous expliquons pourquoi les approches industrielles ne satisfont pas complètement ce cahier des charges. Nos architectures Comet et Maleva, ainsi que leurs instanciations, présentées ensuite, fournissent selon nous des éléments de réponses.

1. Introduction

La notion de composant logiciel occupe aujourd'hui une partie importante du paysage informatique, notamment dans les domaines du génie logiciel, des systèmes distribués ou encore des réseaux. Comme tout concept émergent, l'ambiguïté prévaut lorsqu'il s'agit d'en donner une définition claire. Nous pouvons cependant nous reposer sur une définition relativement consensuelle :

Les composants logiciels sont des unités de composition de logiciels

Cette proposition est bien sûr peu satisfaisante mais elle est un point de convergence de nombreux travaux autour du concept de composant. Dès les années 80, le terme de composant sera souvent employé en informatique (par exemple dans [1] et [2]). Nous pouvons cependant dégager deux approches fondatrices : les architectures logicielles et les composants binaires.

Issu de la recherche, le domaine des architectures logicielles propose de décrire la structure d'un logiciel comme un assemblage, ou plutôt une interconnexion, de composants. Dans cette optique, le critère de neutralité associé à la notion de composant est très fort : il n'est fait aucune supposition sur le rôle d'un composant. Dans [3], D. Garlan and M. Shaw proposent la définition suivante :

Une architecture d'un système donné est une collection de composants calculables - ou plus simplement composants - associée à une description des interactions entre ces composants - les connecteurs.

Du point de vue industriel, les besoins en terme de composants logiciels s'expriment plutôt par la nécessité de disposer de technologies permettant de composer des applications à partir d'éléments logiciels réutilisables [4]. L'hypothèse forte qui prévaut ici est la notion de diffusion sous forme binaire de ces éléments appelés composants. Nous pouvons extraire de ces concepts la définition suivante :

Un composant est une entité logicielle (d'implémentation), élaborée et vendue (sous forme binaire) par une société et utilisée par un (ou des) client(s) pour composer un logiciel.

Les nombreuses solutions industrielles à base de composants qui émergent aujourd'hui (COM, JavaBeans, EJB, Corba Components) s'inspirent principalement de ces deux courants. Nous commençons en section 2 par décrire de façon synthétique ces différentes approches. Après ce tour d'horizon, nous présentons ce qui, pour nous, peut conduire à une véritable notion identitaire de composant logiciel (section 3). Ce point de vue semble cependant entrer en conflit avec certaines fondations sur lesquelles reposent les modèles industriels, ce qui a motivé l'élaboration de deux modèles de composant, différents et complémentaires, au sein de notre équipe OASIS. Ces approches de recherche, Comet et Maleva, sont présentées en section 4. Nous expliquons également dans cette section dans quelle mesure ces travaux satisfont la notion identitaire de composant logiciel proposée précédemment. Finalement, la section 5 conclut cet article.

2. Propositions industrielles

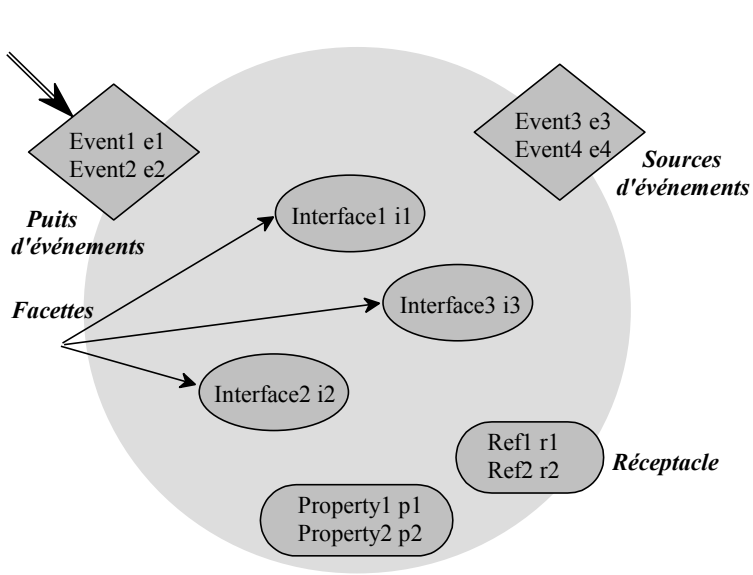
Depuis 1995, Microsoft développe un ensemble de technologies permettant (d'après la documentation) la composition d'applications à partir de briques logicielles développées séparément. Dans ces technologies, COM et dérivés [5], issues de OLE (*Object Linking & Embedding*), Microsoft privilégie l'approche des composants binaires : l'utilisateur n'a aucun accès au code des composants qu'il manipule. La vision de Microsoft est assez extrême puisque les informations disponibles sur les composants sont fortement réduites. L'abstraction principale concernant la composition est l'interface. Plus précisément, le couplage entre composants est défini en terme de satisfaction d'interface. Du point de vue technologique, COM s'appuie principalement sur un mécanisme de découverte d'interfaces à l'exécution. Ceci permet la composition dynamique des applications par des composants conçus indépendamment les uns des autres. En terme de réutilisation, les interfaces peuvent être composées par confinement ou agrégation. Selon Microsoft (et contrairement au même concept dans le monde objet), un composant agrégé est accessible directement depuis l'extérieur du composant agrégat[1]. Un composant confiné, pour sa part, ne peut exporter d'interface au niveau du composant qui le contient. Des problèmes liés à ces outils de composition sont identifiés et précisément analysés dans [18].

Les JavaBeans [6], développés par Sun Microsystems, représentent une sorte de contrepoint à l'approche Microsoft. Du point de vue conceptuel, un JavaBeans est d'après les spécifications *"un composant logiciel réutilisable qui peut être manipulé graphiquement dans un outil de conception"*. Inspiré par les architectures logicielles, ce modèle de composant privilégie l'extraction de la relation de couplage via l'utilisation d'un mode de communication de type implicite, basé sur les événements. Un Beans n'identifie pas directement le destinataire des événements qu'il émet, c'est le concepteur d'application qui explicite ces dépendances. Point important, le modèle de mise en oeuvre proposé est clairement séparé du modèle conceptuel. Au niveau opérationnel, la réflexion [7] (introspection,

[1] Dans la terminologie objet, un agrégat est un objet utilisé en tant qu'attribut d'un autre objet et donc, par définition, inaccessible depuis l'extérieur (sous peine de violation du principe d'encapsulation).

intercession) joue un rôle très important. Les Beans sont de plus empaquetés dans un format binaire (jar), ils ne sont donc pas incompatibles avec l'approche des composants binaires.

Les Enterprise JavaBeans [8] et les Corba Components [9] représentent les deux approches les plus récentes en terme de composants logiciels. Leur objectif principal, dans les deux cas, concerne l'élaboration de la partie fonctionnelle des serveurs dans le cadre d'applications industrielles de type 3-tiers : client/serveur/troisième tiers (essentiellement support de données). La logique métier est donc conçue indépendamment de la logique système qui est déléguée au support d'exécution. Pour définir la logique métier des serveurs, les EJB et les Corba Components proposent en premier lieu un modèle de composant. Les EJB peuvent être comparés à une extension des JavaBeans de ce point de vue. Les Corba Components proposent plutôt une sorte de fusion entre les approches de Sun et de Microsoft. Deux outils de composition, présentés comme complémentaires, sont fournis : les interfaces et la communication implicite.




```
component {
  attribute Property1 p1;
  attribute Property2 p2;

  provides Interface1 i1;
  provides Interface2 i2;
  provides Interface3 i3;

  consumes Event1 e1;
  consumes Event2 e2;

  uses Ref1 r1;
  uses multiple Ref2 r2;

  emits Event3 e3;
  publishes Event4 e4;
} exemple;
```

component exemple

Figure 1. Modèle abstrait des composants Corba

Chaque composant Corba propose une ou plusieurs interfaces (les facettes) permettant d'offrir des services dans une approche de type objets distribués. Les composants peuvent également s'abonner et/ou diffuser des événements, à la manière des JavaBeans. Sur la figure 1, le composant exemple propose trois facettes/interfaces : i1, i2 et i3. Les puits d'événement e1 et e2 abonnent respectivement le composant aux événements de type Event1 et Event2. La source d'événement e3 correspond à une simple émission, e4 étant utilisée pour la diffusion. L'interface IDL décrivant ce composant est également présentée en figure 1. En plus de ces modèles abstraits, de nombreuses technologies utilitaires sont proposées : gestion du cycle de vie, support des transactions, outils de déploiement, etc. Le modèle des Corba Components peut être perçu, globalement, comme une sorte d'extension du modèle EJB, notamment en terme d'interopérabilité[2].

3. Vers une notion identitaire de composant logiciel

Les différentes approches industrielles, si elles ne sont pas forcément en compétition, proposent chacune leur propre vision de la notion de composant logiciel. Pour Microsoft, un composant est une

[2] Les EJB sont restreints à l'environnement Java.

boîte noire prête à être instanciée dans un logiciel. Sun propose plutôt, via les JavaBeans, un modèle de programmation graphique. Les EJB et les Corba Components sont un peu à part puisqu'ils se placent dans un cadre applicatif assez spécifique. Mais l'axe de travail commun se situe presque exclusivement dans la sphère technologique. Il semble difficile d'en extraire une notion prégnante et identitaire de composant logiciel.

Deux axes de recherche complémentaires peuvent être envisagés pour dégager une problématique adéquate. Tout d'abord, il est possible de reconnaître, en guise de bilan de plus de vingt années de recherche dans le domaine, un certain nombre de limites à l'approche objet, notamment dans le cadre des systèmes distribués [20]. A l'autre bout du spectre, de nouveaux besoins, par exemple en terme de dynamicité, émergent aujourd'hui et nécessitent parfois d'importantes remises en question.

Nous pouvons distinguer deux familles de limites potentielles des modèles objets courants : les limites d'ordre structurel et les limites d'ordre comportemental, c'est-à-dire relatives au contrôle.

3.1. Aspect structurel

Le point de vue structurel identifie l'ensemble des concepts tangibles au niveau du modèle. Concernant les objets, nous avons extrait à ce niveau trois types de problèmes : granularité, couplage et hiérarchisation.

3.1.1. Granularité

Dans les applications, les objets issus de la conception - généralement de grain important - et les objets utilitaires - de grain plus fin[3] - se côtoient. Ce découpage quelque peu abrupt doit bien sûr être raffiné. En effet, les fonctionnalités communes aux objets, comme l'instanciation, l'utilisation (invocation des méthodes) ou la destruction ne suffisent plus aujourd'hui. Certains objets - et seulement certains - sont sujets à la migration, la persistance, le versioning, etc. La possibilité de distinguer sur le plan structurel différentes catégories d'entités semble aujourd'hui nécessaire. Ces besoins, sans être étiquetés sous l'appellation "problème de granularité", sont en partie identifiés dans les JavaBeans, EJB et Corba Components. Les composants y sont par exemple considérés comme unités d'empaquetage (JavaBeans) ou de persistance (EJB/Corba Components).

3.1.2. Couplage

Evoqué beaucoup plus souvent, le problème du couplage implicite doit également être pris en compte [19]. Dans l'expression des fonctionnalités des objets (code des méthodes), il est possible d'instancier et d'utiliser d'autres objets. Il est également possible de récupérer (en argument) des références externes. Ces liens d'utilisation n'apparaissent pas au niveau structurel mais résultent plutôt de l'exécution proprement dite des programmes. Prenons l'exemple de deux classes A et B définies (en Java) de la façon suivante :

```
class A {
  void ma() {
    B b = new B() ;
    b.mb() ;
  }
}
```

[3] Un exemple frappant est celui des entiers en Smalltalk.

```
class B {
  void mb() {
    // code de mb
  }
}
```

Ces deux classes sont clairement couplées puisque la méthode ma() de A " a besoin " de B et de sa méthode `mb()` pour fonctionner. En raffinant, on peut établir que les méthodes `A.ma()` et `B.mb()` sont fortement couplées[4]. Si l'on ne dispose pas du code de ces classes, il est impossible d'inférer ce couplage pourtant très fort (statique). En terme de réutilisation, en particulier au niveau binaire, cette forme de couplage représente un véritable obstacle. Dans notre exemple, la classe A est difficilement réutilisable si l'on ignore son lien tacite avec la classe B. Les modèles de communication implicite par diffusion/abonnement (*publish/subscribe*) apportent ici une ébauche de solution. D. Garlan et M. Shaw présentent dans [3] les avantages et inconvenients de ce style architectural (*Event-based Implicit Invocation*). L'idée principale est qu'au lieu d'invoquer directement la procédure d'une entité, un composant (publisher) propose spécifiquement un ou plusieurs événements en émission. D'autres composants (subscribers) peuvent s'enregistrer auprès du publisher afin de recevoir ces événements. Cette idée est reprise dans les EJB et les Corba Components.
Plus simplement, l'extraction du couplage statique entre entités d'un système complexe est possible en interdisant l'explicitation des destinataires des messages ou événements dans le code même de ces entités. Le couplage est alors décrit explicitement par l'intermédiaire de connexions. La partie haute de la figure 2 présente une interaction entre deux objets *un* et *autre*. Le couplage entre ces deux objets est à la fois de nature forte et implicite. La partie basse de la figure transcrit cet exemple dans le monde de la programmation orientée connexion [4].

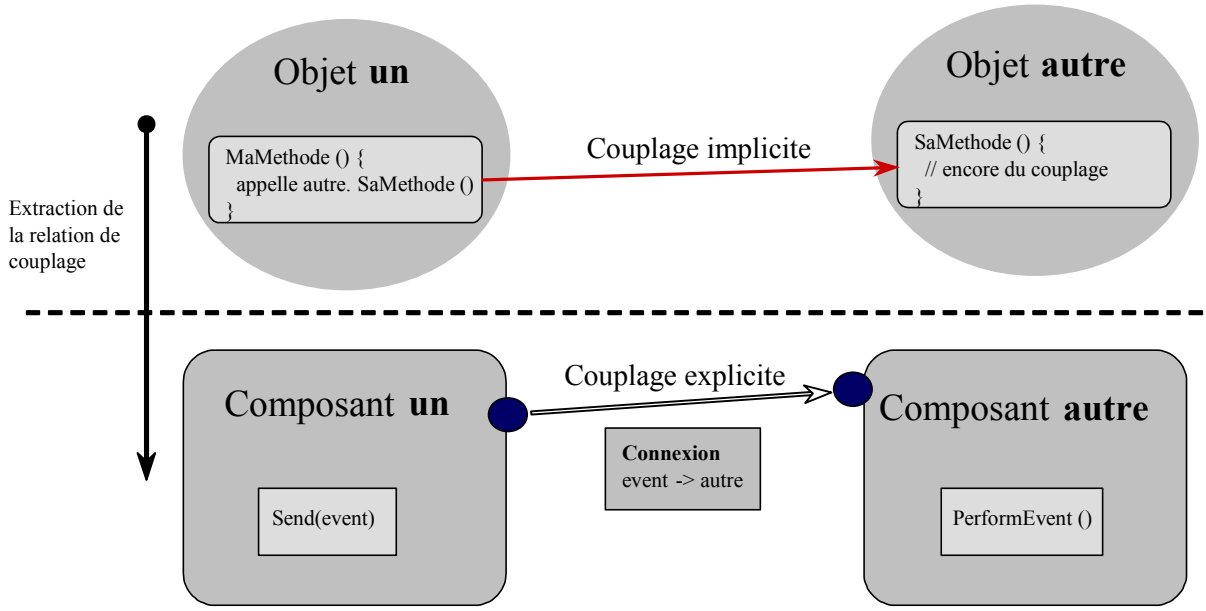


Figure 2. Des objets aux composants via l'extraction de couplage

3.1.3. Hiérarchisation

La hiérarchisation structurelle représente la possibilité d'exprimer, au niveau des modèles (ou des structures, des architectures), le fait qu'un groupe d'entités forme une autre entité de niveau supérieur. Le problème de couplage implicite empêche de "contenir" (notion de confinement) la structure d'un objet, empêchant par la même occasion la hiérarchisation structurelle. Pourtant, cette "voie

[4] Toute invocation de `A.ma()` entraîne inévitablement une invocation de `B.mb()`.

hiérarchique" semble proposer une solution très intéressante au problème de granularité. En effet, la hiérarchisation offre une grande flexibilité pour le choix du niveau de granularité. La section 4.1. illustre l'importance de ce critère par un exemple de composition hiérarchique dans la plateforme Maleva. Aucune approche industrielle ne semble se diriger vers une solution de ce type.

3.2. Aspect comportemental

L'aspect comportemental est également extrêmement sensible. Sous l'appellation comportement, nous réunissons tout ce qui touche de près ou de loin au contrôle sur les entités logicielles mises en jeux. Ceci concerne notamment l'expression proprement dite du contrôle, mais également tous les aspects potentiellement intrusifs à ce niveau[5].

3.2.1. Contrôle implicite

Dans les environnements objet courants, une grande partie du contrôle sur les objets est intégralement déléguée au support d'exécution. Pour de nombreux systèmes informatiques récents, ce contrôle devient complexe : concurrence, répartition, sécurité, persistance et autres caractéristiques comportementales doivent être explicitées. Le cas le plus évident de problème lié au comportement concerne la gestion du flot de contrôle dans les objets. En général, ces derniers communiquent par invocation synchrone de méthode : l'objet appelant transmet le flot de contrôle lorsqu'il invoque la méthode d'un autre objet. Supposons qu'un objet a de classe A dispose d'une méthode `ma()` effectuant un appel à une méthode `mb()` d'un objet b de classe B. Sur la figure 3, nous pouvons constater qu'après invocation de `a.ma()`, l'exécution de la méthode se poursuit jusqu'à l'invocation de `b.mb()`. Le flot de contrôle passe alors sous la responsabilité de l'objet b. Une fois l'exécution de la méthode `b.mb()` terminée, l'objet cible b redonne le contrôle à l'objet source a qui peut continuer l'exécution de sa méthode `ma()`.

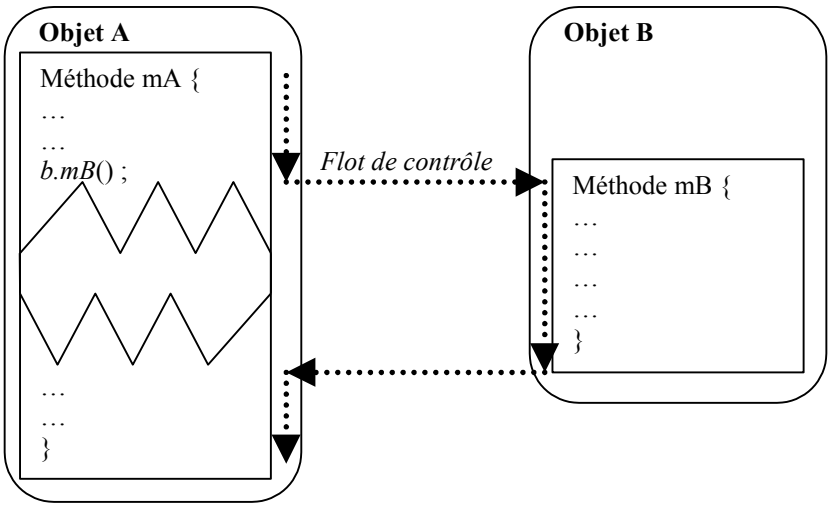


Figure 3. Gestion implicite du flot de contrôle dans les modèles objets

Dans les applications parallèles, ce mode de fonctionnement n'est pas suffisant puisqu'il est alors nécessaire de disposer d'une maîtrise - au moins partielle - sur le flot de contrôle. Nous pouvons par exemple imaginer une situation dans laquelle les méthodes `ma()` et `mb()` peuvent s'exécuter en parallèle (cf. Figure 4).

[5] Il est, par exemple, difficilement imaginable de découpler totalement contrôle et fonctionnalités.

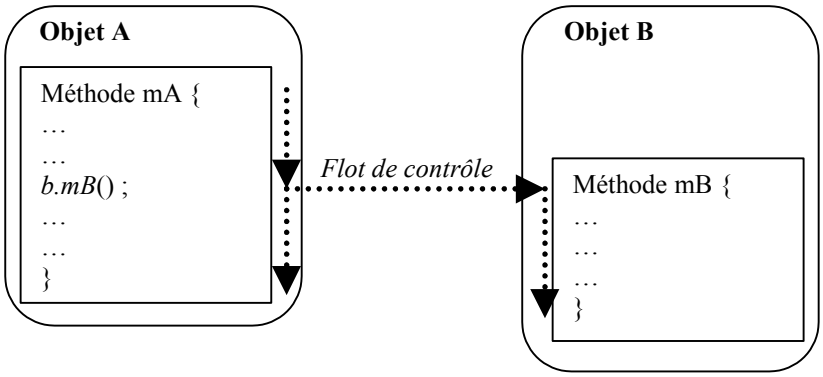


Figure 4. Dédoublement du flot de contrôle lors de l'invocation de méthodes.

Pour expliciter les fonctionnalités liées au contrôle des objets (principalement sur les aspects de concurrence et de répartition), trois familles de solutions sont proposées dans [10] : approche applicative (construction de bibliothèques), approche intégrée (construction linguistique), et approche réflexive (réification du contrôle). Nous verrons que, dans le cadre des composants, une possibilité originale peut être proposée : explicitation structurelle du contrôle (cf. présentation de Maleva). Il est important de noter que ces différentes solutions ne sont pas forcément contradictoires.

3.2.2. Cohérence comportementale de la composition

Discuter précisément sur la cohérence sémantique des compositions logicielles semble utopique si l'on exclut cette dimension comportementale. La plupart des approches industrielles table sur une validité purement structurelle (liée aux interfaces), position hautement contestable. Considérons par exemple la gestion problématique des activités multiples dans les composants COM ou JavaBeans. Si l'on ne se repose que sur les mécanismes proposés, la création d'une nouvelle activité peut, dans certain cas, totalement déstabiliser une application.

3.3. Critère de neutralité

Un autre besoin extrêmement important concerne le critère de neutralité associé à la notion de composant que nous voulons établir. Nous devons ici effectuer la jonction avec le thème des architectures logicielles. Pour décrire la structure d'un logiciel quel qu'il soit, il ne doit être fait aucune conjecture sur le rôle d'un composant ou le cadre applicatif des architectures. Les EJB et les Corba Components posent ici problème puisque la catégorie de logiciel considérée est restreinte (quoique importante).

3.4. Proposition

Nous proposons donc d'approcher les composants logiciels par le biais de besoins d'ordres structurel - granularité, couplage, hiérarchisation - et comportemental - expression du contrôle et intégration de l'aspect comportemental dans la problématique de composition - tout en respectant le critère de neutralité, nécessaire pour satisfaire l'approche des architectures logicielles. De façon complémentaire, les composants binaires découlent, nous semble-t-il, naturellement de cette notion.

4. Deux approches de recherche

Deux modèles de composants logiciels ont été développés au sein du thème OASIS pour donner corps à la notion abstraite décrite précédemment. Nous ne ferons qu'introduire brièvement leurs principes

par rapport aux problématiques mentionnées ci-dessus. Le lecteur peut se référer respectivement à [11] et [13] pour plus de détails sur les modèles, architectures et prototypes évoquées.

4.1. Maleva

Maleva est une plate-forme de conception et de simulation de systèmes multi-agents [11]. Le but de cette approche est de fournir un modèle et des outils permettant la modélisation de chaque agent sous forme d'un réseau interconnecté de composants. Les composants mis en œuvre dans Maleva sont proches conceptuellement des JavaBeans. La principale originalité de ce modèle concerne la différenciation flot de contrôle / flot de données. Ceci est clairement explicité au niveau du graphe d'interconnexion via l'introduction de liens de contrôle s'ajoutant aux liens classiques de données (cf. figure 5). L'idée principale de Maleva est de permettre la réification au niveau structurel de certaines propriétés comportementales.

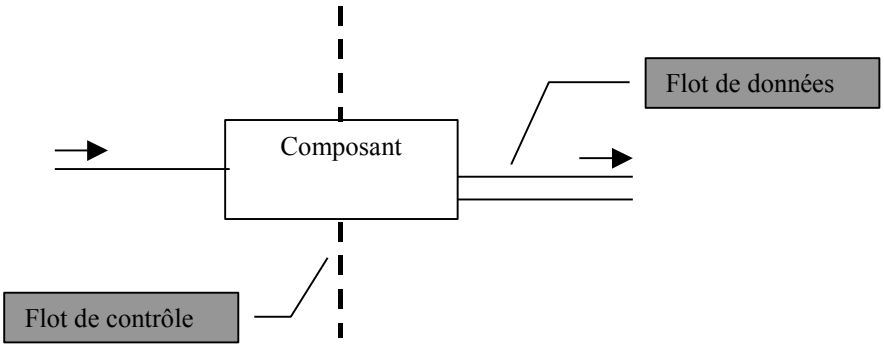


Figure 5. Modèle de composant Maleva

Ce modèle de conception propose également une approche hiérarchique. En effet, il est possible d'instancier des composants de type composite encapsulant un ensemble de composants de grain inférieur. Les composants encapsulés sont alors confinés ; ils ne sont donc plus visibles autrement que par l'intermédiaire de leur supérieur (au sens hiérarchique). La figure 6 présente un exemple de décomposition hiérarchique de l'architecture d'un agent simulant le comportement une fourmi ouvrière évoluant dans un écosystème. Le niveau supérieur de la hiérarchie fournit les abstractions les plus élevées : la capacité pour l'agent à vieillir (composant *Maturing*) et son comportement global (composant *Behavior*). Le composant *Behavior* est décomposé structurellement afin de fournir deux comportements spécialisés qui sont un comportement de mouvement aléatoire (*Random Move*) et un comportement dépendant de la position spécifique de l'agent (*Local Behavior*). Enfin un troisième niveau de décomposition est proposé permettant la réaction de l'agent fourmi dans le cas où il capterait une phéromone émise par une autre fourmi. Cet exemple montre que l'approche hiérarchique fournit un cadre intéressant pour la conception descendante. Contrairement aux méthodes de décomposition fonctionnelles de type SADT, il est important de garder à l'esprit que les composants existent également au niveau de l'implémentation. Bien sûr, les approches ascendantes et mixtes sont également envisageables.

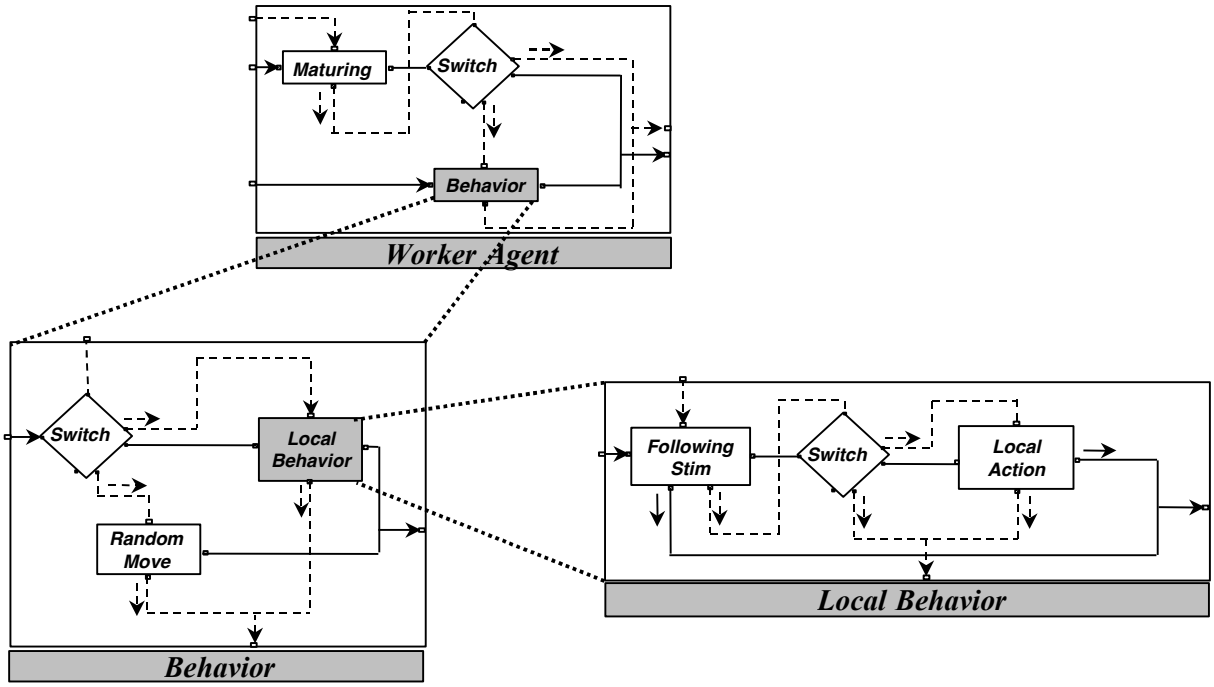


Figure 6. La hiérarchisation dans Maleva : décomposition d'un agent ouvrière

Le modèle conceptuel proposé par Maleva est complété par un modèle d'implantation bien distinct. Dans les JavaBeans cette distinction est présente, toutefois la cohérence entre les deux modèles est contestable[6]. Dans Maleva, nous avons plutôt essayé d'assurer continuité et cohérence lors de la concrétisation des concepts proposés. Cette contrainte complexifie le modèle opérationnel dans une large mesure. De nombreux mécanismes doivent en effet être mis en oeuvre : communications asynchrones via l'utilisation de boîtes aux lettres, gestion des activités multiples, typage dynamique, etc. Cependant, il est difficile à l'heure actuelle de valider cette cohérence de continuité mais il est important de mettre en évidence le fait que les concepts tangibles au niveau du modèle Maleva se retrouvent au niveau implémentation.

Deux instanciations de l'architecture ont été développées : Maleva/D conçue en Delphi, et Maleva/J en environnement Java (bâti à partir du modèle de JavaBeans). Parmi les exemples implémentés et testés à l'aide de ces deux plates-formes, nous pouvons citer la modélisation d'un écosystème (proies/prédateurs) ainsi qu'une réingénierie du projet MANTA [12] (simulation d'une colonie de fourmis).

4.2. Comet

L'objectif principal du projet Comet [13] est de fournir une aide pour la conception, l'implémentation et l'exécution de systèmes distribués. Les systèmes que nous visons plus précisément sont particulièrement dynamiques (il se modifie en cours d'exécution) et hétérogènes en terme de stratégies de contrôle.

La proposition que nous faisons est double : modèle conceptuel pour la problématique de conception et plate-forme(s) opérationnelle(s) pour l'implémentation et l'exécution. Le modèle de composant utilisé est inspiré du framework Rapide [14]. Dans ce modèle (cf. figure 7), les composants

[6] Par exemple, les émissions d'événements au niveau conceptuel correspondent à des invocations synchrones de méthodes au niveau opérationnel. Ceci peut poser problème si l'on souhaite modifier la sémantique d'émission des événements (envois asynchrones).

communiquent par événements asynchrones. Le type des composants indique quels événements ils sont susceptibles de recevoir ou d'émettre.

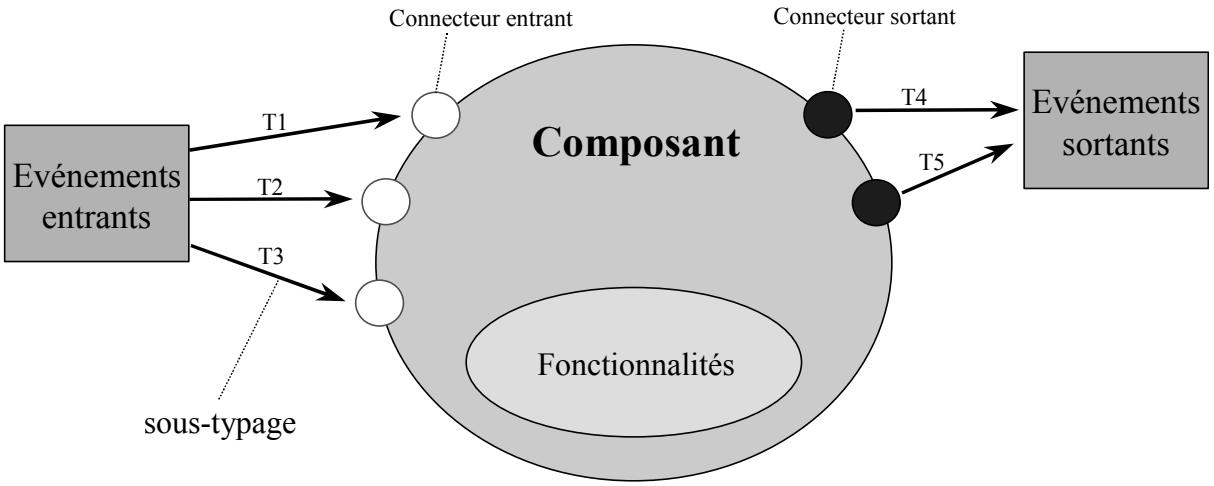


Figure 7. Comet : modèle de composant, point de vue structurel

Au niveau comportemental, la réification du contrôle se fait dans le cadre d'une méta-architecture de grain fin, inspirée de CodA [15]. L'idée de base est la réification des fonctionnalités de contrôles - sous forme de méta-entités comparables aux objets - liées au cheminement des événements au sein d'un composant (cf. figure 8). Cette approche très flexible permet la construction de nombreux modèles comportementaux interopérables.

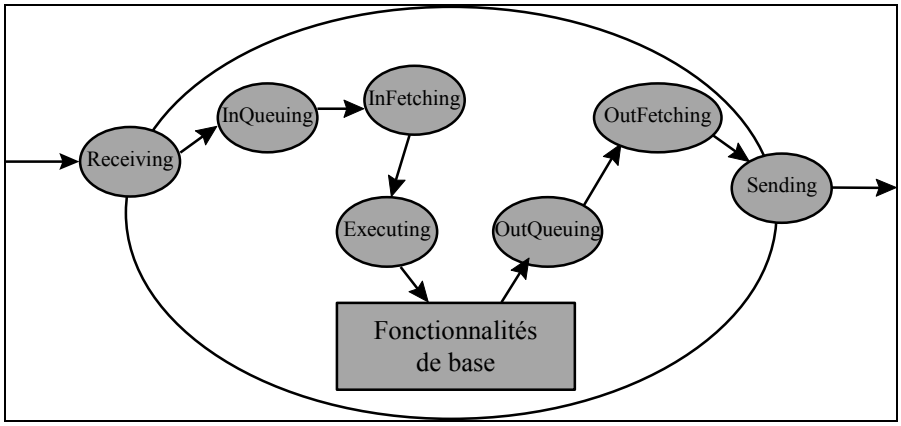


Figure 8. Réification fonctionnelle du contrôle dans Comet

L'adéquation entre le modèle structurel, le modèle comportemental et - dans un avenir plus lointain - les modèles d'implémentation de Comet est en cours d'étude via l'utilisation de méthodes formelles.

Au niveau de la granularité, Comet établit clairement une distinction entre objet (de grain minimal fin) et composant (de grain minimal plus important). Par exemple, les composants Comet sont considérés comme unités de répartition/déploiement. Ceci signifie que la localisation d'un composant donné n'est pas significative au niveau de l'application : le déploiement est réalisé de façon complètement transparente[7].

L'architecture Comet s'intéresse également de façon approfondie à la problématique de l'évolution logicielle [16]. L'idée est d'autoriser les applications à être modifiées (ou à s'auto-modifier) dynamiquement. Les possibilités sont nombreuses à ce niveau : modification des architectures

[7] Il est par exemple possible de déployer de façon transparente tous les composants d'une application sur une seule machine ou sur un vaste réseau de stations de travail.

(topologie dynamique) ou des composants (adaptabilité dynamique). Les rôles et protocoles Comet couvrent ces aspects dynamiques mais leur description dépasse le cadre de cet article.

Ces différents concepts ont été implémentés dans le framework Comet/J basé sur l'environnement Java. Pour valider ce framework, nous avons conçu et implémenté quelques systèmes distribués simples : réplication adaptative, synchronisation d'horloges, etc. Des applications d'orientation industrielle sont aujourd'hui à l'étude. D'autres instanciations de l'architecture sont également prévues.

Contrairement à Maleva, Comet ne propose pas de cadre hiérarchique mais cette extension du modèle est en cours d'étude et semble incontournable.

4.3. Synthèse des modèles

Les modèles proposés par Maleva et Comet sont différents et conçus relativement indépendamment l'un de l'autre (Maleva est antérieur à Comet) mais tous deux satisfont la notion identitaire de composant que nous proposons. Ils répondent effectivement aux besoins mis en relief par cette notion :

- **Besoins d'ordre structurel :**
    - **granularité** : nos approches distinguent sur ce plan composants et objets mais Maleva va encore plus loin en proposant un grain de départ minimal (le grain réel étant fonction du niveau hiérarchique).
    - **couplage** : l'extraction de la relation de couplage structurel est proche dans les deux cas. Comet propose de plus une approche formelle du problème.
    - **hiérarchisation** : en cours d'élaboration dans Comet (approche opérationnelle) et caractéristique centrale de Maleva.
- **Besoins d'ordre comportemental :**
    - **expression du contrôle** : Maleva réifie structurellement le contrôle, ce qui le rend manipulable. Comet propose plutôt la réification du contrôle dans une approche réflexive.
    - **composition comportementale** : le découplage entre contrôle et fonctionnalités permet dans les deux cas une discussion sur la sémantique des compositions.
- **Critère de neutralité :** nos approches, parce qu'elles prennent leur source dans le domaine des architectures logicielles, n'imposent aucune restriction sur la signification ou le rôle des composants.

Deux autres caractéristiques communes peuvent être signalées. Premièrement, Maleva et Comet proposent toutes deux une séparation claire entre le modèle conceptuel et la problématique de mise en oeuvre. Nous pensons que ceci représente une étape indispensable, souvent absente des propositions industrielles. Deuxièmement, les composants Comet/J et Maleva/J sont représentés sous forme binaire. La problématique de composition binaire n'est donc pas éludée, nous la considérons comme un effet de bord certain, et nécessaire.

5. Conclusion

Dans cet article, en nous reposant sur de nombreux travaux de recherche, nous établissons une proposition de notion identitaire de composant logiciel. Des besoins d'ordre structurel et comportemental doivent être satisfaits pour donner corps à cette notion. Les architecture Comet et Maleva, quoique dissemblables, se rejoignent sur ce point. Les approches industrielles, pour leur part,

ne satisfont pas l'intégralité des besoins qui conduisent, selon nous, à cette notion identitaire de composant logiciel. Par exemple, les JavaBeans proposent un cadre conceptuel très intéressant pour ce qui concerne l'assurance de cohésion structurelle des interconnexions de composants. Cependant, l'aspect comportemental est quasiment éludé. Le modèle COM pose également problème par l'absence relativement marquante d'abstractions. Le cadre conceptuel proposé est peu développé, au profit de l'axe technologique. Les modèles EJB et Corba Components semblent pour leur part trop spécialisés (architectures 3-tiers) et la délégation totale de la logique système peut être vue comme particulièrement restrictive dans le cadre d'applications où la distinction avec la logique métier est floue (ex. : gestion fonctionnelle de la sécurité ou de tout autre aspect intrusif au niveau contrôle).

Pourtant, même pourvu d'une notion prégnante de composant, de nombreuses questions restent en suspens. La cohérence sémantique des compositions semble le point le plus sensible ici[8]. Nos approches abordent le problème mais il n'existe pas - à notre connaissance - de solution complète et définitive. Dans le cadre d'Unicon [17], ce problème est traité par la définition d'une sémantique claire au niveau des connecteurs. Selon nous, il semble cependant difficile de représenter, au niveau des composants, les contraintes liées aux architectures (nous suggérons plutôt une vision protocolaire pour Comet et graphique - c'est à dire basée sur les graphes d'interconnexion - pour Maleva).

La problématique générale de composition logicielle reste donc encore largement ouverte...

6. Bibliographie

[1] Cox B., *Object-oriented Programming : An Evolutionary Approach*, Addison-Wesley, 1986.

[2] Booch G., *Software Engineering with ADA*, The Benjamin/Cummings Publishing Compagny Inc., 1983.

[3] Garlan D., Shaw M., *An Introduction to Software Architecture*, AMBRIOLA V., TORTORA G., Eds, Advances in Software Enginnering and Knowledge Engineering, vol. 1, World Scientific Publishing Company, 1993.

[4] Szyperski, C., *Component Software : Beyond Object Oriented Programming*, Eds Addison-Wesley & ACM Press, 1998.

[5] Microsoft, *Microsoft COM Technologies*, http://www.microsoft.com/com, 1999.

[6] Sun Microsystems Inc. *Javabeans Specifications 1.01.* http://java.sun.com/beans/docs/spec.html, 1998.

[7] Sun Microsystems Inc. *JAVA Core Reflection API and Specification.* http://java.sun.com/products/jdk/1.1/docs/guide/reflection/spec/java-reflectionTOC.doc.html, 1998.

[8] Sun Microsystems Inc. *Entreprise Javabeans Specifications 1.1.* http://java.sun.com/products/ejb/docs.html, 2000.

[9] Object Management Group, *Corba Component Model*, http://www.omg.org, 1999

[10] Briot J-P., Guerraoui R., Löhr K-P., *Concurrency and Distribution in Object-Oriented Programming*, ACM Computing Surveys, vol. 30, n°3, p. 291-329, 1998.

[8] Soulignons l'intervention de Linda Northrop lors d'un panel à Ecoop'99 : *"Components that plug but just don't play"*.

[11] Guillemet A., Haïk G., Meurisse T., Briot J-P., and Lhuillier M,. *Une approche à base de composants pour la conception d'agents.* Journées Francophones pour l'Intelligence Artificielle Distribuées et les Systèmes Multi-Agents (JFIADSMA'99), Ile de la Réunion, Nov. 1999, France. Edition Hermès.

[12] Drogoul A., *De la Simulation Multi-Agents à la Résolution Collective de Problèmes.* Thèse de Doctorat, Université Paris 6, 1993.

[13] Peschanski F., *Comet : Architecture Réflexive à Base de Composants pour la Construction d'Applications Concurrentes et Réparties.* LMO'2K, 26-28 Janv. 2000, Mont St-Hilaire, Canada. Edition Hermès.

[14] Luckham D. C., *Rapide : A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events*, DIMACS Partial Order Methods Workshop, vol. IV, 1996.

[15] MC Affer J., *Metalevel Programming with Coda*, Proceedings of ECOOP'95, LNCS 952, Springer Verlag, août 1995, p. 190-214.

[16] Evans H., DickMan P., *Zones, Contracts and Absorbing Change : An Approach to Software Evolution*, Proceedings of OOPSLA'99, ACM Press, Oct. 1999.

[17] Shaw M., Deline R., Klein D. V., Ross T. L., Toung D. M., Zelesnik G., *Abstractions for Software Architecture and Tools to Support Them*, IEEE Trans. On Software Engineering, vol. SE-21(N.4) avril 1995, p. 314-335.

[18] Sullivan K. J., Marchukov M., and Socha J., *Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model*, IEEE Trans. On Software Engineering, vol. SE-25(N.4) July/August 1999, p. 584-599.

[19] Briand L. C., Daly J. W., and Wüst J. K., *A Unified Framewoek for Coupling Measurement in Object-Oriented Systems*, IEEE Trans. On Software Engineering, vol. SE-25(N.1) January/February 1999, p. 91-121.

[20] Guerraoui R., *What Object-Oriented Distributed Programming may be – and What it does not have to Be.* White Paper. HP Labs. 1999