Implémentation d'une application de simulation selon le modèle PAC-Amodeus

Thierry Duval

IRISA
Campus de Beaulieu
F-35042 Rennes cedex
Thierry.Duval@irisa.fr

Laurence Nigay

CLIPS-IMAG
B.P. 53
F-38041 Grenoble cedex 9
Laurence.Nigay@imag.fr

RÉSUMÉ

Cet article a trait à la conception logicielle d'un système interactif de simulation suivant le modèle d'architecture PAC-Amodeus ainsi qu'à sa réalisation logicielle en langage objet. Nous présentons une façon originale et efficace de transformer une application de simulation non graphique et non interactive en une application interactive reposant sur une visualisation graphique. Cette transformation prend ses fondements dans l'application du modèle d'architecture PAC-Amodeus et consiste à associer des composants de contrôle aux objets du noyau fonctionnel existant. La réussite d'une telle transformation repose donc sur une interprétation du modèle PAC-Amodeus éclairée par les techniques objet offertes par des langages courants tels que C++, Eiffel ou Java : l'héritage, le polymorphisme et la liaison dynamique.

MOTS CLÉS : Architecture logicielle, Application de simulation, Programmation objet, Ingénierie de la simulation.

INTRODUCTION

Un spécialiste de la simulation d'un domaine donné peut structurer une application de simulation en une hiérarchie de classes (avec des langages tels que C++, Eiffel ou Java). Comme il est souligné dans [2], l'approche de structuration est très souvent du haut vers le bas pour obtenir une hiérarchie : par exemple, l'objet de simulation "avion" sera décomposé en deux objets "zone passager" et "cockpit". La structuration en objets résultante est dirigée par le domaine d'application de simulation et par conséquent la sémantique des objets de simulation [1]. Cette structuration ne prend pas en compte l'interaction entre l'utilisateur et les objets de simulation.

La réalisation logicielle des applications de simulation peut reposer sur un environnement de développement (Simulation Model Development Environment ou SMDE) [1] [7]. L'objectif de ces outils est double : réalisation rapide de prototypes et développement incrémental d'applications (par réutilisation d'objets de simulation). Les applications développées avec de tels outils offrent une visualisation de la simulation (interface en sortie) mais ne permettent pas toujours d'interaction entre l'utilisateur et les objets de simulation (interface en entrée et en sortie). Aussi le code des applications développées avec ces outils est organisé selon une architecture logicielle dépendante de l'outil, qui n'est pas adaptée à l'interaction. En effet, l'architecture consiste en un ensemble de composants logiciels traduisant le modèle de simulation et par conséquent l'architecture n'intègre pas de gestion du dialogue entre l'utilisateur et l'application. Par exemple dans l'environnement proposé dans [1], la génération de code est effectuée automatiquement à partir de la description hiérarchique du modèle de simulation, ce modèle étant spécifié par manipulation directe grâce à un éditeur graphique dédié.

Les applications de simulation actuelles sont globalement peu interactives. Nous identifions néanmoins trois classes d'applications de simulation selon le degré d'interaction qu'elles offrent : la première classe d'applications de simulation offre des résultats statistiques à la fin de la simulation [8][5] ; la deuxième classe regroupe des applications qui permettent de visualiser graphiquement l'évolution de la simulation [1] ; la troisième classe comprend des applications qui offrent des possibilités d'interagir avec l'environnement de simulation, comme ce peut être le cas dans certaines simulations de trafic automobile [7] ou encore dans le domaine de la réalité virtuelle [12].

Dans les deux premiers cas, les applications de simulation ne sont pas interactives et l'utilisateur n'a pas de contrôle sur l'évolution de la simulation. Dans ce contexte, notre objectif est de rendre interactive une application de simulation : l'enjeu majeur est de laisser à l'utilisateur la possibilité de modifier les paramètres de la simulation afin de pouvoir évaluer efficacement le modèle de la simulation.

Dans le troisième cas, l'utilisateur est lié à un environnement de développement particulier, qui lui impose une architecture logicielle bien définie (il faut en général créer des composants par héritage de composants fournis), ce qui ne permet pas de réutiliser une hiérarchie d'objets existante.

Partant de ce constat, nous proposons une méthode d'im-

plémentation qui permet de rendre interactive une application de simulation sans avoir à tout reprogrammer. Notre objectif est de minimiser les coûts de conception et réalisation logicielles de cette transformation. Nous identifions deux corollaires à cet objectif :

- ne pas remettre en cause la hiérarchie d'objets conçus et développés par des experts en simulation,
- distinguer l'implémentation de la simulation de celle de l'interface, dont la responsabilité revient à des spécialistes en interface.

Nous partons donc d'un noyau fonctionnel existant, qui n'offre pas de visualisation ni de possibilité d'interaction, en vue d'obtenir une application interactive. L'Interface Homme-Machine (IHM) peut être conçue indépendamment de l'implémentation de ce noyau initial ; l'IHM manipule néanmoins les concepts maintenus par le noyau. Nous proposons une méthode afin de relier le plus efficacement possible le noyau et l'IHM, en produisant donc une architecture la plus adaptée possible à ces besoins.

La distinction entre le code de l'application et celui de l'interface a motivé notre choix d'asseoir notre méthode d'implémentation sur un modèle d'architecture logicielle de systèmes interactifs. Parmi les modèles d'architecture existants [9], nous nous sommes tournés naturellement vers les modèles multi-agent PAC [3] et PAC-Amodeus [11] qui sont adaptés à une implémentation objet [4].

Dans cet article, nous rappelons succinctement les éléments directeurs des modèles PAC et PAC-Amodeus, puis nous présentons les problèmes d'implémentation constatés en appliquant directement ces deux modèles. Nous exposons subséquemment notre méthode d'implémentation. Nous illustrons notre méthode par un exemple d'application intitulée "La vie des bugs" développée en Java.

LE MODÈLE MULTI-AGENTS PAC

PAC est un modèle multi-agent [3] qui a comme principes directeurs le concept d'agent à facettes et une décomposition récursive. Un système interactif est modélisé par une hiérarchie d'agents PAC, comme le schématise la figure 1.

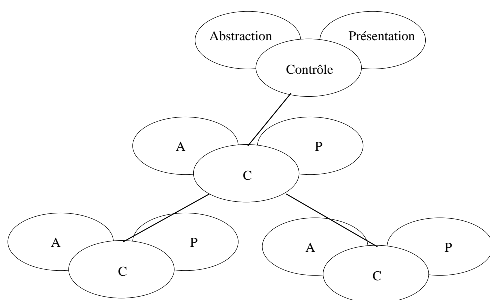

Figure 1 : Les composants logiciels d'un système interactif selon le modèle PAC

Un agent PAC adopte trois perspectives complémentaires : la Présentation, l'Abstraction et le Contrôle :

- La Présentation définit le comportement perceptible de l'agent pour un agent humain, elle concerne à la fois les entrées et les sorties c'est-à-dire les modalités d'action accessibles à l'utilisateur et la restitution perceptible.
- L'Abstraction, avec ses fonctions et ses attributs internes, définit la compétence de l'agent indépendamment des considérations de présentation ; elle constitue le noyau fonctionnel de l'agent.
- Le Contrôle a un double rôle : il sert de pont entre les facettes Présentation et Abstraction de l'agent, et il gère des relations avec d'autres agents PAC.

Tout échange d'informations entre l'Abstraction et la Présentation s'effectue via le Contrôle. C'est aussi par leur Contrôle que deux agents PAC communiquent.

Le modèle hybride PAC-Amodeus

La figure 2 présente les composants logiciels de PAC-Amodeus. Dans ses grandes lignes, PAC-Amodeus [11] reprend les composants du modèle ARCH [6] dont il affine le contrôleur de dialogue, composant principal, en termes d'agents PAC.

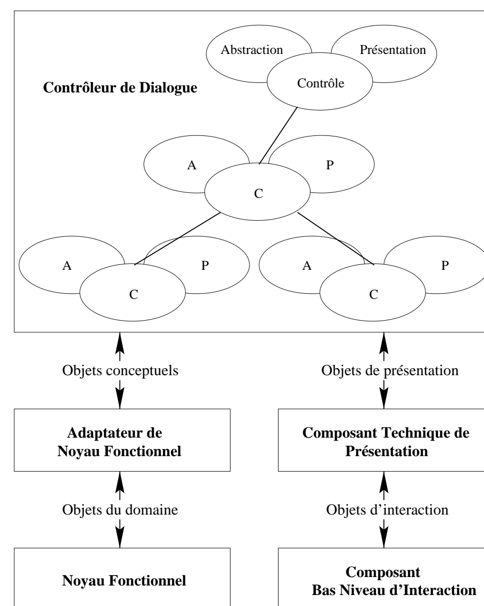

Figure 2 : Les composants logiciels d'un système interactif selon le modèle PAC-Amodeus

Le Noyau Fonctionnel (NF) réalise les concepts du domaine. L'Adaptateur du Noyau Fonctionnel (ANF) sert d'interface entre les objets du domaine et les objets conceptuels exportés vers l'utilisateur. Le Contrôleur de Dialogue (CD), clé de voûte du système interactif, prend en charge l'enchaînement des tâches, gère chaque fil de dialogue au moyen d'une grappe d'agents PAC, et assure la correspondance entre objets conceptuels et objets de présentation. Le Composant Technique de Présentation (CTP) définit les règles de correspondance entre les objets

de présentation et les objets d'interaction. Par exemple, la classe de présentation "choix multiple" est concrétisée sous forme d'un "menu de boutons radio". Le Composant Bas Niveau d'Interaction (CBNI) désigne la plate-forme d'accueil logicielle et matérielle. Ce niveau regroupe les services d'acquisition, d'estampille et de répartition des événements mais aussi les objets d'interaction des boîtes à outils.

Après avoir rappelé les éléments directeurs des modèles PAC et PAC-Amodeus, nous étudions maintenant comment rendre interactive une application de simulation en nous appuyant sur ces deux modèles d'architecture. Afin de simplifier l'exposé qui suit, nous faisons l'hypothèse que tous les objets de simulation sont à rendre interactifs.

TRANSFORMATIONS DIRECTES

Application directe du modèle PAC

En appliquant le modèle PAC, il convient d'organiser l'application en une hiérarchie d'agents. Chaque objet de simulation existant dans l'application initiale sera alors une Abstraction d'un agent. Dans l'application finale interactive, un agent PAC pour chaque objet de simulation est programmé. Pour transformer une application de simulation, il convient donc de programmer un composant Contrôle et un composant Présentation par objet existant. Cette solution de transformation, bien que simple et immédiate, soulève néanmoins des problèmes. L'un des problèmes majeurs réside dans le fait que les objets de simulation communiquent entre eux. Considérer chaque objet comme un composant Abstraction d'un agent PAC implique donc que les agents PAC communiquent entre eux par leurs Abstractions. Or, comme nous l'avons exposé auparavant, deux agents PAC ne communiquent que par leurs facettes Contrôle. Ceci implique de modifier l'architecture initiale de l'application : les éléments du noyau initial sont modifiés. De plus l'architecture initiale est reproduite au niveau des nouveaux composants Contrôles. Nous illustrons ce problème par un exemple simple présenté à la figure 3 : un objet `A1` envoie le message "`bouger`" à un objet `A2`.

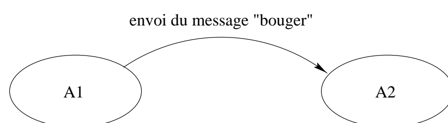

Figure 3 : Un exemple de relation initiale entre deux objets de simulation

En appliquant la méthode ci-dessus, nous obtenons deux agents PAC, comme présentés à la figure 4. Comme les deux agents communiquent par leurs facettes Contrôle, la connaissance de l'objet `A2` n'est plus utile à l'objet `A1` : c'est au contrôle `C1` correspondant d'envoyer un message au contrôle `C2`. Il faut donc reproduire au niveau de `C1` la connaissance des objets de l'application qu'avait initialement `A1`. De plus `A1` doit être modifié pour communiquer avec `C1`, sa facette Contrôle. Cette transformation n'est donc pas satisfaisante en termes de coût de modification et de développement. Un autre problème fondamental à cette méthode de transformation est que la hiérarchie d'agents PAC obtenus correspond à l'organisation initiale des objets de simulation. Or selon le modèle PAC, la hiérarchie d'agents traduit des niveaux d'abstraction dans le traitement du dialogue entre le système et l'utilisateur. Ainsi la hiérarchie d'agents obtenus correspond à l'organisation des objets de simulation et non au dialogue entre le système et l'utilisateur.

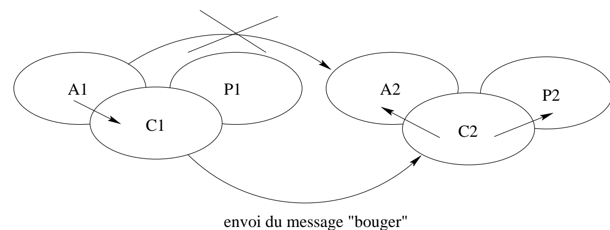

Figure 4 : Transformation directe de l'exemple de la figure 3 selon le modèle PAC

Les problèmes identifiés en appliquant le modèle PAC proviennent principalement du fait que le modèle est purement multi-agent. La transformation implique donc de plaquer la structure initiale en objets de simulation sur une autre structure induite par le modèle. Cette transformation ne peut donc être effectuée à moindre coût sans modifier la structure initiale. Partant de ce constat, nous considérons maintenant un modèle multi-agent hybride, PAC-Amodeus, comme fondement à notre méthode d'implémentation.

Application directe du modèle PAC-Amodeus

En se fondant sur le modèle PAC-Amodeus, les objets de simulation constituent le Noyau Fonctionnel, comme le montre la figure 5. Ces objets peuvent alors être adaptés au niveau du composant Adaptateur du Noyau Fonctionnel (ANF) pour ensuite communiquer avec une Abstraction d'un agent PAC du Contrôleur de Dialogue.

Nous identifions plusieurs avantages à l'application du modèle PAC-Amodeus pour notre transformation :

- Les objets de simulation constituent un composant logiciel indépendant.
- La hiérarchie d'agents PAC constituant le Contrôleur de Dialogue est indépendante de la structure des objets de simulation. En effet, ces derniers communiquent via l'ANF avec une Abstraction d'un agent PAC.
- Un agent PAC du Contrôleur de Dialogue est à l'écoute de l'utilisateur par sa facette Présentation et à l'écoute du Noyau Fonctionnel (NF), peuplé d'objets de simulation, par sa facette Abstraction. En effet dans le cas de simulation, le NF est source d'événements au même niveau que l'utilisateur. Le NF

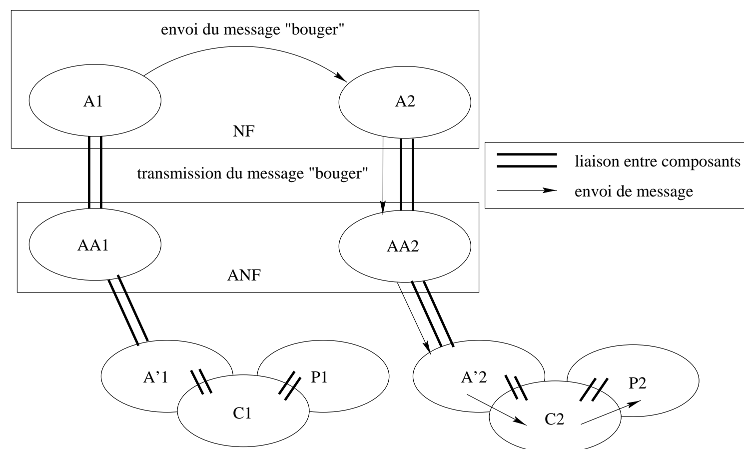

Figure 5 : Transformation directe de l'exemple de la figure 3 selon le modèle PAC-Amodeus

n'est pas asservi à l'utilisateur : ce n'est pas un serveur sémantique qui répond à des requêtes de l'utilisateur.

Comme dans la solution précédente selon le modèle PAC, l'inconvénient majeur de cette transformation réside dans la modification nécessaire des objets de simulation pour qu'ils puissent communiquer avec l'ANF. De plus le coût de programmation de cette transformation est très élevé : si l'implémentation d'un agent PAC consiste en trois classes, il faut donc programmer quatre classes par objets de simulation. Par exemple pour l'objet de simulation `A1` de la figure 5, les classes `AA1`, `A'1`, `C1` et `P1` doivent être créées. Comme le préconise le mécanisme "Slinky" associé au modèle ARCH et donc à PAC-Amodeus, il est possible de réduire le nombre de classes en supprimant le composant ANF (`AA1` et `AA2`). Les objets de simulation communiquent alors directement avec une facette Abstraction d'un agent PAC, `A'1` et `A'2`.

En conclusion, l'application directe des modèles PAC et PAC-Amodeus soulève un problème majeur qui réside dans la modification nécessaire des objets de simulation. Dans le paragraphe suivant, nous proposons de résoudre ce problème avec une méthode de transformation éclairée par les techniques objet, offertes par des langages courants tels que C++ ou Java.

MÉTHODE DE TRANSFORMATION EFFICACE

Nous présentons d'abord notre méthode de transformation qui repose sur le modèle PAC-Amodeus et la technique d'héritage des modèles objets, associée au polymorphisme et à la liaison dynamique. Nous proposons ensuite une optimisation afin de réduire le coût d'implémentation. Nous développons enfin un exemple de taille raisonnable pour illustrer notre méthode.

Principes de la transformation

Nous exploitons une technique objet fort répandue, l'héritage associé au polymorphisme et à la liaison dynamique. Ces techniques sont présentes dans les langages qui nous intéressent plus particulièrement, C++ et Java.

Comme le montre la figure 6, nous proposons dans un premier temps de créer les composants de l'Adaptateur du Noyau Fonctionnel (ANF), `AA1` et `AA2`, par héritage des composants de l'application initiale, le Noyau Fonctionnel (NF). Il est alors possible de substituer à tous les composants de l'application, leurs composants ANF correspondants : en effet grâce à l'héritage public les composants `AA1` et `AA2` sont eux aussi des composants du NF. Il convient ensuite de redéfinir les méthodes nécessaires dans ces composants de l'ANF, la liaison dynamique assurant le bon fonctionnement des envois de messages.

Pour l'exemple très simple de la figure 3, deux composants `AA1` et `AA2` de l'ANF héritent respectivement des composants initiaux `A1` et `A2`. Il convient ensuite d'associer ces composants `AA1` et `AA2` à des agents PAC du Contrôleur de Dialogue, comme présenté à la figure 6 : la facette Abstraction de chaque agent est mise en relation avec un composant de l'ANF. Ainsi les composants de l'ANF, `AA1` et `AA2`, assurent la liaison entre l'application initiale et l'interface-utilisateur tout en laissant ses parties héritées gérer les fonctions propres à la simulation (NF). Par exemple la partie de `AA1` héritée de `A1` déclenche l'envoi du message "`bouger`" vers `A2` qui est en fait `AA2`. `AA2` est alors en mesure de répercuter les modifications vers le Contrôleur de Dialogue, c'est-à-dire la facette Abstraction de l'agent PAC correspondant, `A'2`.

Notre méthode permet de résoudre le problème majeur de la modification des objets de simulation, que nous avions identifié dans les solutions précédentes. En appliquant notre méthode de transformation, les objets de simulation sont inchangés. De plus notre méthode offre les mêmes avantages que ceux identifiés dans le paragraphe intitulé "Application directe de PAC-Amodeus" puisque nous nous appuyons sur le modèle PAC-Amodeus. Néanmoins le coût de transformation reste très élevé puisque quatre nouvelles classes sont à construire par objet de simulation. Nous proposons donc d'adapter cette transfor-

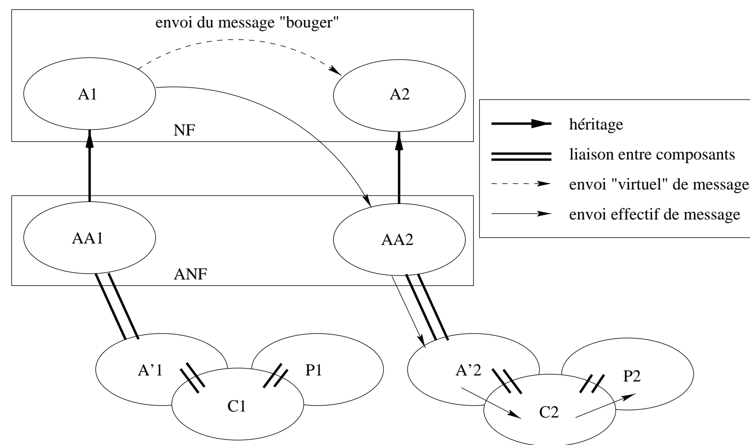

Figure 6 : Transformation “éclairée” de l’exemple de la figure 3 selon le modèle PAC-Amodeus

mation pour réduire le coût de développement.

Optimisation de la transformation

Pour réduire le nombre de classes à créer nous procédons en deux étapes. Tout d’abord, nous appliquons le mécanisme “Slinky” : les composants ANF sont supprimés au détriment de la portabilité car les agents PAC deviennent alors dépendants des objets de simulation. Dans ce cas, les facettes Abstraction héritent des objets de simulation (par exemple, le composant `A'1` hérite du composant `A1`). Dans une deuxième étape, nous considérons que les agents PAC liés à un objet de simulation sont des agents particuliers qui n’ont pas de facette Abstraction puisque leurs compétences résident dans les objets de simulation initiaux. Ainsi les agents PAC de simulation ne sont composés que d’une facette Présentation et d’une facette Contrôle. En supprimant la facette Abstraction de l’agent, c’est alors le composant Contrôle de l’agent qui hérite du composant initial du Noyau Fonctionnel.

Nous illustrons cette solution de transformation à la figure 7 dans le cas de l’exemple de la figure 3 : les Contrôles `C1` et `C2` héritent respectivement des composants du Noyau Fonctionnel `A1` et `A2`. Une Présentation est adjointe à chaque Contrôle. La partie de `C1` héritée de `A1` déclenche l’envoi du message “`bouger`” vers `A2` qui est reçu par `C2`. Ainsi deux agents PAC de simulation communiquent entre eux par leurs Contrôles via le Noyau Fonctionnel.

De cette transformation optimisée découle un Contrôleur de Dialogue organisé en une hiérarchie de deux types d’agents PAC :

- Les agents PAC de simulation sont en liaison directe avec le Noyau Fonctionnel et ne comportent que deux facettes : Contrôle et Présentation. Ces agents sont des feuilles de la hiérarchie d’agents.
- Les agents PAC dédiés au dialogue ne sont pas en liaison avec le Noyau Fonctionnel.

Il est important de noter que la hiérarchie d’agents PAC du Contrôleur de Dialogue ne correspond pas à la structuration initiale de la simulation.

Avant de développer un exemple, nous résumons les avantages de cette transformation optimisée :

- Les objets de simulation sont inchangés.
- Le code de la simulation est distingué de celui de l’interface.
- Le coût de développement est minimisé : deux classes à créer par objets de simulation.
- La structure en agents PAC du Contrôleur de Dialogue est indépendante de celle en objets de la simulation.

EXEMPLE : “LA VIE DES BUGS”

Nous présentons un exemple de simulation implémentée en Java. L’application consiste à faire évoluer des entités dans un univers en deux dimensions. Ces entités, que nous nommons “bugs”, vivent : elles se déplacent, vieillissent, mangent, se fatiguent, parfois se reproduisent, et finissent par mourir.

L’application initiale

L’application de simulation non interactive est constituée de plusieurs composants :

- différentes sortes de bugs (notés “basic”, “erratic”, “sniffer”, “hungry”, “long cruiser”, et “cannibal”), ayant des comportements différents,
- des bactéries, constituant la nourriture des bugs,
- un réservoir, contenant les bactéries,
- une population, regroupant les bugs,
- une zone de vie, gérant un réservoir de bactéries.

Les bactéries savent qu’elles sont contenues dans un réservoir : quand le niveau d’énergie d’une bactérie devient nul, elle le signale à son réservoir, afin qu’il la supprime.

Les bugs savent qu’ils font partie d’une population et qu’ils évoluent dans une zone de vie : un bug demande à sa zone de vie si de la nourriture peut être trouvée et si

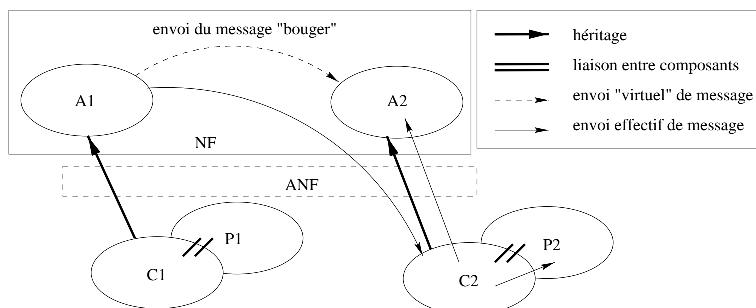

Figure 7 : Transformation optimisée de l'exemple de la figure 3 selon le modèle PAC-Amodeus

il reste à l'intérieur des limites de la zone pour chacun de ses déplacements. Lorsque le niveau d'énergie d'un bug devient nul, il le signale à sa population, afin d'être supprimé.

Un programme principal est responsable de la création de toutes les entités et de leur distribution dans les entités réceptacles adéquates (réservoir et population). Il fait aussi vivre ces entités tant qu'il reste des bugs dans la population.

Nous définissons des interfaces qui constituent les spécifications des classes. Nous manipulons ensuite, dans la mesure du possible, des références à des objets qui les implémenteront. Les interfaces sont les suivantes : `Bacterie`, `Bug`, `Reservoir`, `Population`, et `ZoneDeVie`.

Pour simplifier, l'interface `Bug` hérite ici de l'interface `Bacterie`, et l'interface `Population` hérite de l'interface `Reservoir`. Ces simplifications ne sont pas fondamentales car elles ne seront pas utilisées à des fins “polymorphes”. Les principales classes définies pour implémenter effectivement cette application sont les suivantes :

- `ABacterie` : implémente l'interface `Bacterie`.
- `ABug` : implémente l'interface `Bug` et hérite de `ABacterie` pour des raisons pratiques non polymorphes. Cette classe est abstraite et factorise les attributs et certains aspects de comportement communs à tous les bugs.
- `ABasic` : hérite de la classe `ABug`. Le nom de ce type de bug est explicite : il n'est pas très malin, et se déplace systématiquement en diagonale dans la zone de vie, en rebondissant sur ses frontières.
- `AErratic` : hérite de la classe `ABasic`. Ce type de bug erre aléatoirement dans la zone de vie.
- `ASniffer` : hérite de la classe `AErratic`. Ce type de bug cherche où se trouve la plus grande quantité de nourriture à sa portée avant de se déplacer.
- `AHungry` : hérite de la classe `ASniffer`. Ce type de bug ne laisse pas de nourriture sur son passage.
- `ALongCruiser` : hérite de la classe `AHungry`. Ce type de bug a une très grande capacité de déplacement.
- `ACannibal` : hérite de la classe `ASniffer`. Ce type de bug préfère manger des `Bug`, parfois même d'autres `Cannibal` plutôt que des `Bacterie`.
- `AReservoir` : implémente l'interface `Reservoir` et gère des `Bacterie`.
- `APopulation` : implémente l'interface `Population` et hérite de `AReservoir`, gère des `Bug`.
- `AZoneDeVie` : implémente l'interface `ZoneDeVie` et gère une `Population` de `Bug` et un `Reservoir` de `Bacterie`.

Chacune de ces classes ne manipule que des références à des types d'objets décrits par les interfaces précédentes. Le polymorphisme et la liaison dynamique assurent que les bonnes méthodes sont utilisées sur les bons objets au moment des appels effectifs. Les classes des bugs effectifs n'offrent pas de nouveaux services : ces classes ne font que redéfinir certains comportements des bugs dont ils héritent. Ceci permet de créer de nouveaux types de bugs sans avoir à modifier les populations qui les manipulent. Nous sommes ici dans un cas typique d'utilisation du polymorphisme.

A partir de cette application initiale non interactive, nous appliquons notre méthode de transformation optimisée, pour visualiser les entités suivantes : les bactéries, les bugs et la zone de vie. Nous définissons donc des classes Contrôle et Présentation

Les composants de contrôle

Nous définissons l'interface `CBug` comme héritant de l'interface `Bug`. Nous créons les classes suivantes :

- `CBacterie` : hérite de la classe `ABacterie`.
- `CBasic` : hérite de la classe `ABasic` et implémente l'interface `CBug`.
- `CErratic` : hérite de la classe `AErratic` et implémente l'interface `CBug`.
- `CSniffer` : hérite de la classe `ASniffer` et implémente l'interface `CBug`.
- `CHungry` : hérite de la classe `AHungry` et implémente l'interface `CBug`.
- `CLongCruiser` : hérite de la classe `ALongCruiser` et implémente l'interface `CBug`.

- CCannibal : hérite de la classe ACannibal et implémente l'interface CBug.
- CZoneDeVie : hérite de la classe AZoneDeVie.

Un Reservoir peut gérer des CBacterie : en effet, la classe CBacterie hérite de ABacterie qui implémente l'interface Bacterie. Comme en Java l'héritage est public, CBacterie rend donc au moins les mêmes services que Bacterie. Des méthodes sont alors redéfinies dans cette classe CBacterie de façon à ce qu'un changement d'état ou de position d'une bactérie soit répercuté à sa Présentation.

La création d'une interface CBug est un passage obligé : l'héritage étant simple en Java, il s'agit ici d'un artifice permettant de manipuler de façon similaire tous les composants contrôles issus des abstractions de bugs du noyau fonctionnel initial. En effet, l'héritage simple ne permet pas de retrouver la hiérarchie initiale de classes : ainsi il n'existe plus de relation entre CBasic, CErratic, CSniffer, CHungry, CLongCruiser et CCannibal. Comme seuls l'héritage et l'implémentation d'interfaces (conceptuellement équivalent à l'héritage public), associés à la liaison dynamique, permettent de substituer une classe Contrôle à une classe d'objets du noyau fonctionnel, nous n'avons pas ici d'autre recours que le passage par cette interface commune. C'est ici l'intérêt de cet exemple qui soulève un problème réel de réutilisation. Dans notre solution, les Contrôles de bugs implémentent l'interface CBug, comme toutes les classes de bugs du Noyau Fonctionnel implémentaient l'interface Bug. Ainsi nous pouvons faire gérer des instances de CBug par une population : les contrôles effectifs (CBasic, CErratic, etc.) implémentant l'interface CBug et donc l'interface Bug, nous avons la garantie qu'ils pourront rendre les services attendus. La préservation des comportements spécifiques des bugs de l'application initiale est alors assurée par chacun des héritages ponctuels. Des méthodes sont redéfinies dans chacune des classes CBasic, CErratic, etc., de façon à ce qu'un changement d'état ou de position d'un bug soit aussi répercuté à sa Présentation.

Enfin, une instance de la classe CZoneDeVie peut être à l'origine de mises à jour dans les instances de CBacteries et des CBugs. Ces mises à jour sont répercutées aux composants Présentation.

Les composants de présentation

Seulement trois classes de Présentation sont nécessaires pour assurer la visualisation de notre application : PBacterie, PBug et PZoneDeVie. En effet, les bugs ont tous une présentation du même type, paramétrée par le type du bug. Le détail de ces classes de présentation ne présente pas d'intérêt particulier dans le cadre de notre méthode de transformation. Néanmoins il est important de noter que les classes de Présentation possèdent la notion de Contrôle (CBacterie, CBug et CZonedeVie) car une facette Présentation peut recevoir des informations de l'utilisateur qu'il convient de répercuter à la facette Contrôle.

Leçons tirées de l'exemple

Nous utilisons donc les objets de l'application initiale, pour créer une application similaire, mais graphique et potentiellement interactive, grâce à une spécification explicite (sous la forme d'interfaces) des services rendus par les objets.

La hiérarchie des classes de l'application initiale a pu être conservée : les deux applications, classique et graphique-interactive (présentée figure 8), cohabitent et sont toutes deux opérationnelles. En pratique, l'ensemble initial des objets de simulation forme un "package" Java (simulation.jar), auquel est associé un "main" (le programme principal), qui met les objets en relation les uns avec les autres et effectue la simulation. Nous avons ensuite deux autres "packages" Java pour les Contrôles et les Présentations (controle.jar et presentation.jar) et un nouveau programme principal exploitant les trois "packages". Ce dernier met en relation les Contrôles entre eux pour obtenir une simulation graphique interactive sous la forme d'une "application" Java. Enfin, nous avons défini un troisième programme principal (associé à quelques nouveaux objets) permettant de visualiser l'ensemble sous forme d'une "applet" Java, toujours en utilisant ces trois "packages".

CONCLUSION

Dans cet article, nous avons montré comment utiliser le modèle d'architecture logicielle PAC-Amodeus pour rendre interactive une application de simulation existante. La méthode systématique de transformation s'appuie sur les mécanismes des langages objets que sont l'héritage, le polymorphisme et la liaison dynamique. Nous rappelons ici que cette méthode est générale et qu'elle peut s'appliquer à de nombreuses applications de simulation. Son atout principal est qu'elle ne modifie en rien le noyau fonctionnel initial.

Nous avons illustré notre méthode de transformation par un exemple de taille significative, programmé en Java. Dans l'exemple, la transformation repose sur une spécification précise des types d'objets manipulés en utilisant le mécanisme d'interfaces fourni par Java. La transformation aurait pu se faire d'une façon similaire avec d'autres langages objets comme Eiffel ou C++, en utilisant alors l'héritage multiple, et en remplaçant les interfaces Java par des classes abstraites.

Une perspective à nos travaux consiste à appliquer d'autres modèles d'architecture logicielle, comme MVC [10] ou AMF [13] pour rendre interactive une application de simulation. L'étape suivante consistera alors à comparer les méthodes, en terme de coût de programmation, de modifiabilité et de portabilité du code obtenu.

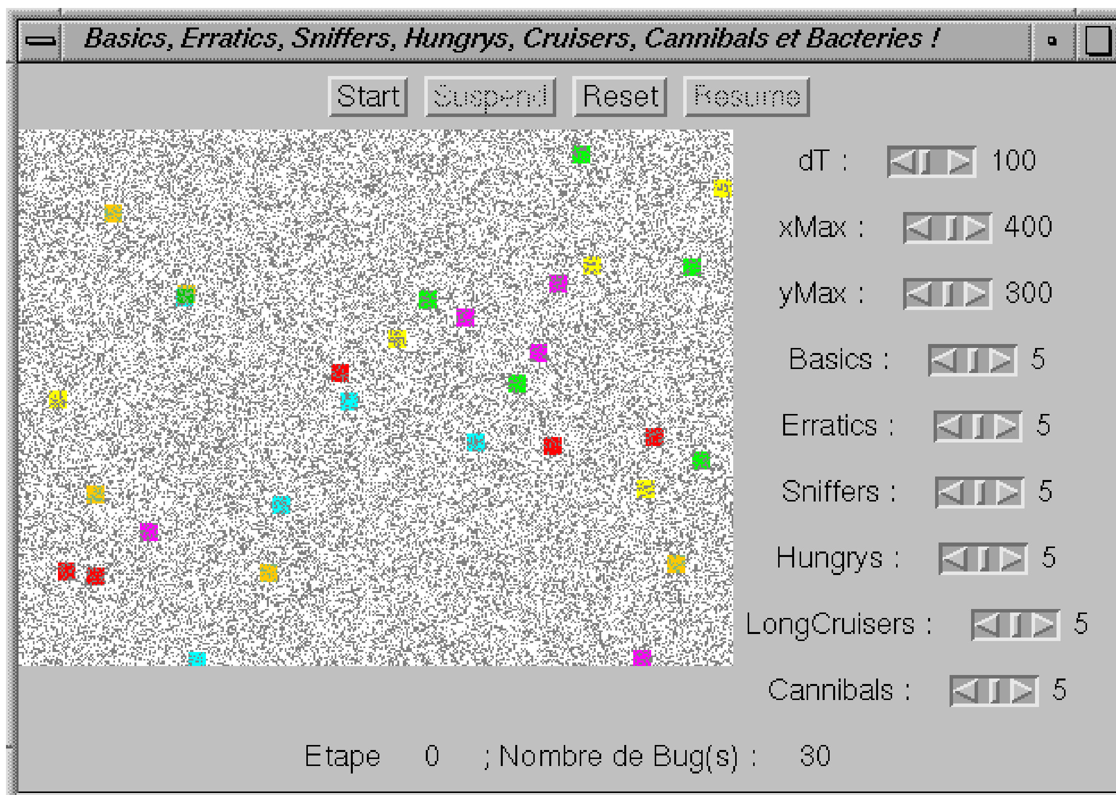

Figure 8 : Copie d'écran de l'application graphique obtenue de la simulation "La vie des bugs"

BIBLIOGRAPHIE

1. Balci O., Bertelrud A., Esterbrook C., and Nance R. Visual simulation environment. *Proceedings of the IEEE 1998 Winter Simulation Conference*, pages 279–287, décembre 1998.
2. Balci O., Bertelrud A., Esterbrook C, and Nance R. Dynamic object decomposition in the visual simulation environment. *Proceedings of the 11th European Simulation Multiconference*, pages 69–73, juin 1997.
3. Coutaz J. *Interfaces homme-ordinateur, conception et réalisation*. Dunod informatique, 1990.
4. Coutaz J., Nigay L., and Salber D. Agent-based architecture modelling for interactive systems. Critical Issues in User Interface Engineering, P. Palanque & D. Benyon Eds., Springer-Verlag: London Publ., ISBN 3-540-19964-0, pages 191-209., 1995.
5. Cremer M., Demir C., Donikian S., Espie S., and McDonald M. Investigating the impact of aicc concepts on traffic flow quality. *5th World Congress on Intelligent Transport Systems*, Séoul, Corée du sud, octobre 1998.
6. T. U. W. T. Developers. Arch : A metamodel for the runtime architecture of an interactive system. SIGCHI Bulletin, 24, 1, pp. 32-37, 1992.
7. Donikian S., Chauffaut A., Duval T., and Kulpa R. Gasp: from modular programming to distributed execution. *Computer Animation'98, IEEE, Philadelphie, USA*, juin 1998.
8. Donikian S., Espie S., Parent M., and Rousseau G. Simulation studies on the impact of acc. *5th World Congress on Intelligent Transport Systems*, Séoul, Corée du sud, octobre 1998.
9. Duval T. Modèles d'architecture pour les applications graphiques interactives : la famille seeheim. *Revue de CFAO et d'infographie, volume 9, numéro 4*, pages 529–555, 1994.
10. Krasner G. and Pope S. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, pages 26–49, Août/Septembre 1988.
11. Nigay L. Conception et modélisation logicielle des systèmes interactifs : application aux interfaces multimodales. Thèse de Doctorat de l'Université de Grenoble 1, IMAG, 1994.
12. Reignier P., Harrouet F., Morvan S., Tisseau J., and Duval T. Arévi : A virtual reality multiagent platform. *Proceedings of the First International Conference on Virtual Worlds (VW'98), Paris, Lecture Notes in Computer Science, Artifial Intelligence series (LNCS/AI 1434)*, juillet 1998.
13. Tarpin-Bernard F. et David B. Amf : Un modèle d'architecture multi-agents multi-facettes. *Technique et science informatiques, volume 18, numéro 5*, pages 555–586, 1999.