Le temps de la réflexion et le temps du calcul dans les simulateurs multi-agents

Vincent Ginot, Hervé Richard, Nicolas Kim Nguyen-Van

INRA, unité de biométrie d'Avignon,
Domaine St Paul,
84 914 Avignon Cedex 9, France
ginot@avignon.inra.fr ; richard@avignon.inra.fr

Mots clefs : SMA, XML, sérialisation, architecture

Résumé

L'essor de la modélisation individu-centrée et des systèmes multi-agents s'accompagne progressivement d'une offre logicielle destinée à aider informaticiens et modélisateurs à les construire et les utiliser. Mais une contrainte forte de ce type de modélisation est le coût calcul et le coût mémoire qu'il entraîne, au point qu'une limite pratique, souvent atteinte dans le domaine de la dynamique des populations, est la durée d'un pas de temps ainsi que le nombre total d'agents que l'on peut raisonnablement simuler. Cette contrainte est d'autant plus cruciale que ces systèmes ne possèdent aucune solution analytique susceptible de guider l'étude de leur comportement. Pour connaître ce dernier, voir même pour simplement vérifier le bon fonctionnement d'un SMA, le seul moyen est de multiplier les simulations pour explorer l'espace des possibles. Et cela devient vite problématique si une simulation demande quelques heures voire quelques jours. La vitesse d'exécution dépend de la complexité des agents et de l'intensité de leurs échanges, mais elle dépend aussi pour une bonne part des choix architecturaux de la plate-forme de simulation. Et on peut penser que cette contrainte sera d'autant plus pesante que la plate-forme sera dédiée à des utilisateurs moins informaticiens. C'est la difficulté que nous rencontrons avec Mobidyc, une plate-forme multi-agents dédiée à la construction de modèles individus-centrés en dynamique des populations. Cette plate-forme vise un publique de biologistes ou d'écologistes souhaitant rapidement prototyper des modèles, un peu comme ils le feraient dans le domaine des simulations à base d'équations différentielles avec des logiciels comme Stella, Matlab-SimuLink, ou Model Maker. Ceci sans compétences particulières en informatique. Le principe en est une programmation par composants : l'utilisateur définit l'état et le comportement de ses agents en y ajoutant des composants prédéfinis et paramétrables. Ces composants peuvent représenter des tâches complètes, par exemple un déplacement ou une reproduction, ou bien représenter de manière plus élémentaire des "primitives" de tâches, sortes de sous-actions que l'on enchaîne pour définir des tâches plus complexes. Lors de sa conception, la plate-forme ne sait donc pas si elle travaillera sur des truites ou sur du blé, ni quel sera le nom des variables définissant les états ou le nom des tâches définissant les comportements. L'important est que l'utilisateur puisse immédiatement tester un nouvel agent, activer ou désactiver ses différentes tâches, revenir assembler de nouvelles tâches au moyen des primitives, bref construire son modèle tout en l'exécutant pour le tester. La conséquence en est un module d'exécution qui mobilise d'importantes ressources à manipuler et à mémoriser des quantités d'objets forts utiles en phase de mise au point mais forts encombrants en phase d'exécution.

Du point de vue architectural, on peut donc se demander s'il ne serait pas souhaitable de séparer deux étapes dans la vie d'un modèle, la phase de mise au point, et la phase d'utilisation. La première demande de pouvoir exécuter le modèle de manière aussi interactive

que possible, la deuxième aussi rapidement que possible, et les deux semblent difficilement compatibles. Idéalement, pour la phase d'utilisation, il faudrait réécrire le simulateur en fonction du modèle que l'utilisateur vient de définir.

Et au fond pourquoi pas ? Dans le cadre d'une programmation par composants cette gageure semble envisageable. L'idée de base est extrêmement simple : si des composants sont capables de définir entièrement (ou quasi-entièrement) le comportement des agents, ces composants devraient être capables d'écrire un code d'exécution optimisé qui tienne compte de la manière dont l'utilisateur les a agencés et paramétrés. La principale difficulté vient de la nécessaire simplification du code et de la coordination des morceaux de code en provenance des différents objets. Car il s'agit d'obtenir un code aussi concis que possible et qui ne reproduise pas l'implémentation dont ils sont issus. Sinon le progrès ne serait pas très grand.

Nous nous proposons de vous exposer le travail que nous avons initié sur ce thème, en essayant de tirer parti d'un travail en cours de finalisation sur la sérialisation XML des modèles individus-centrés développés sous Mobidyc. La procédure comprend trois modules. Le premier est écrit en Smalltalk, le langage utilisé par Mobidyc. Il assure la sérialisation XML des agents au moyen de l'API DOM. Le deuxième, toujours en Smalltalk, réinstancie les objets Mobidyc à partir de leur descriptif XML pour régénérer le modèle original. Le troisième, en Java, instancie dans un premier temps des objets de même type que les objets Mobidyc. Ces objets sont ensuite exécutés, et ils produisent le code Java "optimisé". Ce code est ensuite compilé et associé au noyau d'exécution Java. Les simulations peuvent alors être lancées en Java.

Les premiers essais semblent prometteurs. Nous avons testé le concept sur deux modèles très simples. Le premier met à l'épreuve les procédures d'instanciations et de gestion de la mémoire : chaque individu possède une probabilité de survie de 0.5 mais donne naissance à un nouvel individu à chaque pas de temps. La population reste donc stable. Le deuxième teste les communications entre agents : à chaque pas de temps, chaque agent interroge un attribut de tous les autres agents et met son état à jour en fonction de l'état de tous les autres agents. Il n'y a ni naissances ni mortalité, la population reste donc stable aussi. Les deux modèles sont construits sous Mobidyc, sérialisés en XML, puis recompilés automatiquement en Java (JDK 1.4 sous Linux/Mandrake). Nous lançons ensuite les simulations en faisant varier le nombre d'individus de 100 à 100 000 pour le modèle 1 et de 100 à 10 000 pour le modèle 2. Les gains de vitesse vont d'un facteur 10 à 70, et sont globalement plus importants sur le modèle 1. Ces chiffres sont cependant susceptibles de bouger car le moteur Java n'inclut encore aucune gestion de mémorisation des résultats (mémorisation désactivée sous Mobidyc lors des tests), ni aucune structure pour lire des expériences simulatoires et gérer les paramètres qui seront rendus variables au cours des simulations. Il ne sait donc que répéter la même simulation. Inversement, aucun travail d'optimisation n'a encore été effectué sur le code alors que nous avons déjà bien travaillé cette question sous Mobidyc. Enfin nous utilisons la configuration standard de la machine Java : elle n'est sans doute pas optimale pour les grands nombres d'agents.

La sérialisation XML des modèles était indispensable, ne serai-ce que pour assurer une certaine indépendance des modèles vis-à-vis de la plate-forme Mobidyc et d'améliorer leur pérennité. Elle ouvre en outre des possibilités nouvelles en matière d'indépendance entre modèle et plate-forme à travers cette possibilité de réécriture de code à la volée. Si cette idée devait aboutir, il ne serait alors peut-être pas totalement irréaliste d'imaginer adapter le code produit à telle ou telle plate-forme existante, l'enrichissant ainsi d'un mode de programmation des comportements par composants. Tout en étant bien conscient des difficultés : d'une plate-

forme à l'autre, bien des choix architecturaux diffèrent, ne serait-ce que dans la manière dont les agents communiquent. Mais ces choix ne devraient a priori pas transparaître dans les descriptifs XML des modèles.

Nous pourrons ensuite essayer de conclure en revenant sur la question du temps : dans la démarche modélisatrice, faut-il séparer le temps de la conception du temps des calculs, et cela a t'il une incidence sur l'architecture de nos simulateurs ?