PRIMA-UML : une méthode incrémentale de validation de performance basée sur les premiers diagrammes UML

Une approche critique de la méthode

Lionel Deliege
ldeliege@info.fundp.ac.be

Facultés Universitaires Notre-Dame de la Paix - Namur

Résumé Développer des applications complexes entraîne des choix au niveau du design. Ces choix peuvent avoir des conséquences relativement importantes sur les performances de la future application. Il est donc intéressant de pouvoir prédire les performances d'une application et cela dès les premières étapes du développement. La généralisation d'UML comme outil de design permet, en le couplant à des techniques d'évaluation de performance, de simuler des systèmes. Mais ces techniques couvrent-elles réellement tous les paramètres susceptibles de modifier les performances d'une application ?

1 Introduction

Le développement d'une application complexe dont le critère de performance est important demande une stricte attention dès les premières étapes du développement. Un mauvais choix au niveau de l'architecture risque de pénaliser toute l'application future et entraîner des défauts de performances, quels que soient les algorithmes ou le système utilisés, obligeant ainsi une reconception global de l'application. L'analyse des performances dès les premières étapes du développement semble être une solution à ce problème, faut-il encore savoir l'appliquer simplement.

Un certain nombre d'articles se penchent sur la question de l'estimation de la performance des logiciels dès les premières phases de l'implémentation. Ainsi, la méthode PRIMA-UML définie dans [2] se base sur la notation UML et sur le modèle de performance SPE. SimML décrit dans [3] est un outil de simulation de programmes basé sur le modèle UML. Le système simule des utilisateurs selon des critères définis par le développeur. Une troisième méthode décrite dans [4] se base elle aussi sur UML et utilise elle aussi SPE comme modèle de performance.

SPE, ou Software Performance Engineering est la science qui étudie la gestion des performances des applications. Définie par Smith dans [1] cette méthode est très lourde à utiliser et le résultat est souvent loin d'être celui escompté.

L'arrivée d'UML change la donne, en permettant d'avoir un formalisme pour tout le design de l'application et donc une vue globale du système. UML est simple, ouvert et globalement généralisé pour le design des applications Orienté Objet actuelles.

PRIMA-UML est donc une méthode pour appliquer SPE aux applications actuelles en ajoutant au schéma UML des attributs de performance. Le modèle de performance ainsi créé pourra être solutionné.

Ce document est découpé comme suit : premièrement nous ferons un rapide tour d'horizon des deux outils utilisés (UML et SPE), nous décrirons ensuite la méthode PRIMA-UML de manière théorique. Nous critiquerons ensuite cette méthode en essayant d'y apporter une amélioration et pour finir, nous donnerons nos conclusions.

2 Contexte

2.1 UML

UML est une notation considérée de plus en plus comme standard pour l'analyse, le design et l'implémentation de programmes complexes. UML est composé d'une série de notations tentant de couvrir toutes les étapes du design d'une application. Trois types de schémas sont, au minimum, intéressants dans la méthode PRIMA-UML :

Le diagramme des cas d'utilisation (UCD pour Use Case Diagram) est un schéma reprenant les différents scénarios que l'utilisateur peut utiliser. Les utilisateurs sont représentés par de petits bonshommes et divisés en classe. Chaque classe peut effectuer une série d'actions.

Le diagramme de séquence (SD pour Sequence Diagram) découpe les actions en étapes montrant l'évolution d'une fonction dans l'espace et dans le temps. Dans l'espace car la fonction peut faire appel à d'autres fonctions qui ne se trouvent pas forcément sur le même ordinateur et dans le temps car le diagramme montre la chronologie des événements.

Le diagramme de déploiement (DD pour Deployment Diagram) consiste à visualiser comment l'application va être déployée dans le système final, visualisant ainsi les connections réseaux entre les machines, les différents disques installés et les différents CPU utilisés.

A ces trois schémas, il nous semble intéressant d'ajouter le diagramme de classe (CD pour Classe Diagram). Il permet de représenter graphiquement les objets utilisés par l'application, leurs attributs ainsi qu'ils ont entre eux.

2.2 SPE, Software Performance Engineering

SPE, définie par Smith dans [1] est une méthode d'analyse de performance d'une application à travers l'intégralité du cycle de développement. La méthode divise l'analyse de performance en deux parties : le Software Model (SM) ou modèle de programme et Machinery Model (MM) ou modèle du système.

Le modèle de programme utilise un Graphe d'Execution (ou EG). Il représente l'exécution du programme et est composé de noeuds. Ceux-ci peuvent être basic, cyclique, conditionnel ou consister en une séparation-jointure de noeuds. Le noeud basic correspond simplement à une donnée entrante provoquant une donnée sortante, après éventuelle transformation. Le noeud cyclique est une boucle dans l'exécution du programme. Le noeud de condition est un choix exclusif pour la suite du programme. La séparation et la jointure permettent une concurrence entre deux suites d'actions qui doivent être exécutées en parallèle.

Le modèle du système utilise un modèle de réseau de files étendu (EQNM pour Extended Queueing Network Model). EQNM a besoin des composants du systèmes et de la topologie du système (liaison entre les composants) ainsi que des paramètres d'utilisation des divers composants (qui seront donnés par l'analyse du SM).

La différentiation entre le SM et le MM permet de tester l'application en gardant le même modèle de programme mais en changeant les caractéristiques de la machine, évitant ainsi de modifier toutes les caractéristiques définies dans la partie utilisation du software. Le schéma est donc plus portable car totalement indépendant de la plate forme.

3 La méthode PRIMA-UML

La méthode PRIMA-UML est une méthode dite incrémentale. Les auteurs sont parfaitement conscient que les développeurs n'ont pas envie de modifier leur méthode de travail pour tenir compte d'un détail non fonctionnel comme la performance. UML permet de créer des objets et de les améliorer, à la fin ou pendant un cycle de développement. Le développeur n'a donc pas à se soucier du critère de performance, c'est l'analyste de performances qui ajoutera les informations utiles en améliorant le schéma UML.

3.1 Étapes

Les étapes sont au nombre de 8 :

- Étape 1 : Annoter le diagramme des cas d'utilisation (UCD).
- Étape 2 : Pour tous les cas, identifieer et annoter le diagramme de séquence correspondant aux passages clés du scénario.

– Étape 3 : Procéder pour tous les diagrammes de séquence afin d'obtenir un meta-EG.
– Étape 4 : Adapter le meta-EG pour annoter le diagramme de déploiement et générer l'instance d'EG.
– Étape 5 : Dériver l'EQNM à partir du diagramme de déploiement annoté.
– Étape 6 : Assigner des paramètres numériques aux instances EG.
– Étape 7 : Joindre les instances d'EG et les EQNM dans le modèle de performance
– Étape 8 : Résoudre le modèle de performance.

L'étape 1 consiste à annoter le diagramme des cas d'utilisation. Pour ce faire, les utilisateurs et les arcs reliant les utilisateurs aux actions sont annotés avec un poids. Le poids correspond à la probabilité que cette action soit exécutée par la catégorie d'utilisateur à laquelle l'arc est relié. Le schéma 1 représente un diagramme de cas annoté.

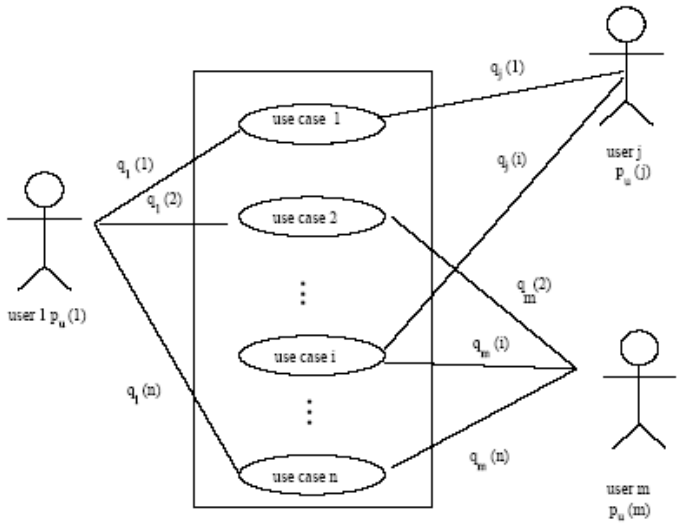

FIG. 1. Schéma 1. Un diagramme des cas d'utilisation annoté (source : [2]).

L'étape 2 consiste à identifier et annoter le diagramme de séquence correspondant aux cas définis dans l'étape numéro 1. Par annoter, nous entendons définir l'échelle de temps afin de connaître l'ordre des opérations dans la séquence ainsi que de définir la taille des données échangées lors des appels de fonctions. Cette dernière annotation permettra entre autre d'estimer la bande passante ré-

seau nécessaire ainsi que la vitesse minimum des disques durs. La partie gauche du schéma 2 représente un diagramme de séquence annoté.

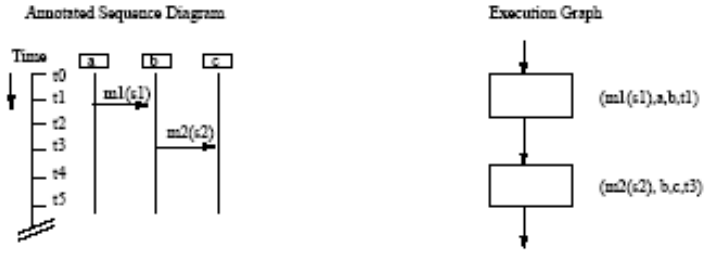

FIG. 2. Schéma 2. Un diagramme de séquence annoté avec le meta-EG correspondant (source : [2]).

L'étape 3 consiste à créer le meta-EG, c'est à dire le graphe d'exécution indépendant de toute plate forme. Ce graphe est créé à partir des informations recueillies lors de l'analyse des UCD et des SD. A ce moment, le graphe ne contient aucune information sur la vitesse à laquelle va s'exécuter une étape. Tout est sous forme de variable non initialisée. Le schéma 2 est un exemple de transformation entre SD et meta-EG, la partie de gauche représente le meta-EG.

L'étape 4 consiste à transformer le meta-EG en une instance d'exécution en utilisant les données introduites dans le diagramme de déploiement. Le diagramme de déploiement va quantifier la durée de chaque étape, en faisant le lien entre les composants du système en charge d'effectuer l'étape et le noeud du graphe d'exécution correspondant.

L'étape 5 va dériver à partir l'EQNM à partir du diagramme de déploiement annoté. A ce moment, le premier modèle du système est fini. Il est à noter qu'à ce stage du développement, les concepteurs de PRIMA-UML sont au courant que le développeur n'aura peut-être qu'une vague idée de la plate forme finale. Le schéma 3 représente un diagramme de déploiement, le schéma 4 un réseau EQNM.

L'étape 6 consiste alors à assigner des valeur numériques à l'instance d'EG.

L'étape 7 va joindre l'instance d'EG avec le ou les modèle(s) du système, l'étape 8 tentera, à partir des modèles du système défini, de résoudre le modèle de performance.

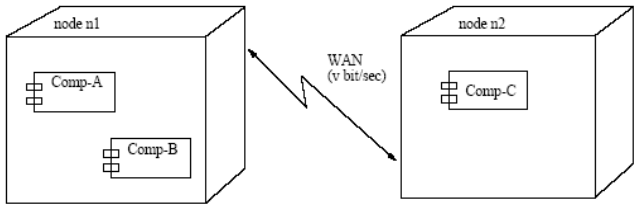

FIG. 3. Schéma 2. Un diagramme de déploiement (source : [2]).

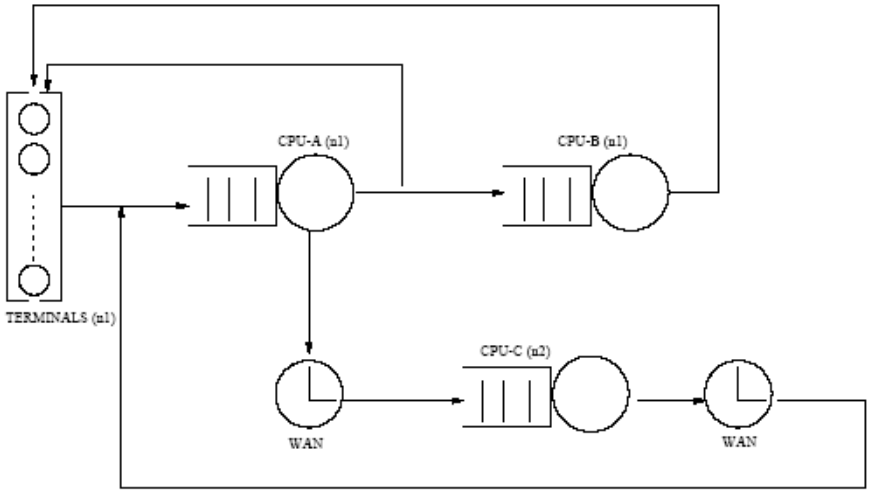

FIG. 4. Schéma 2. Un diagramme EQNM (source : [2]).

4 Critique

La méthode PRIMA-UML, comme toutes les autres méthodes basées sur UML pour tenter de déterminer les performances futures d'un logiciel, semble arriver à des résultats encourageants tout du moins dans les exemples choisis. Néanmoins, si les chiffres introduits pour justifier le pourcentage d'utilisation de telle ou telle fonction sont justifiés grâce à des calculs stochastiques, les calculs de performances CPU sont quant à eux partiellement justifiés.

Nous formulons ici deux critiques à propos du Machinery Model et une à propos du Software Model. Ces critiques peuvent probablement être appliquées à beaucoup d'autres méthodes d'analyse de performance. Elles portent d'une part sur l'évaluation du temps CPU demandé par une fonction, sur l'espace mémoire utilisé par une fonction et d'autre part sur la gestion de la concurrence entre fonctions.

Les deux exemples définis ci-après proviennent de l'interview d'un des anciens chefs de projet de la société BSB de Louvain-la-Neuve ayant travaillé à l'informatisation d'une importante banque luxembourgeoise. Le premier exemple porte sur le chargement des clients en mémoires. Le système devait être capable de charger le profil de 6000 clients en une seconde, un profil étant composé des informations clients ainsi que de l'intégralité de leur portefeuille d'actions. Le deuxième exemple est un système de batch devant être exécuté tous les jours sur le serveur. Celui-ci ne peut subir aucun retard et doit être effectué en moins de 24 heures.

4.1 Évaluation des performances CPU

Les exemples définis dans les différents articles partent toujours d'un principe simple : '*sachant que telle fonction prendra, avec tel architecture, autant de milliseconde, nous pouvons conclure que ...*' mais dans aucun cas, l'article ne définit comment les chiffres ont pu être trouvés. Un problème évident se pose alors : quelle crédibilité pouvons nous accorder à l'évaluation de performance d'une fonction qui n'est pas encore implémentée dans un langage qui n'a pas encore été choisi ? Et qu'elles seraient les conséquences d'une erreur de calcul ?

Au moment du design, aucune fonction n'a encore été codée. Le seul critère connu est une évaluation de la complexité de la fonction en terme de grand O. Ce critère permet d'évaluer si la fonction est calculable dans un temps raisonnable, mais en aucun cas ne nous permet d'évaluer de manière précise le temps nécessaire à la réalisation de la fonction. Certes, le temps défini par SPE peut-être vu comme une borne supérieur que le programmeur doit respecter, mais rien n'empêche que cette borne soit impossible à respecter. L'objectif de base, qui

consiste à éviter dès le design des erreurs coûteuses en terme de performance, ne serait donc plus atteint.

Le système de chargement de la base de clients de la banque doit être capable de charger en mémoire 6000 clients à la seconde. Sachant que ce chargement est évalué dans le design à 0,0001 secondes, le critère de performance est normalement respecté. Après implémentation en Java, le temps de chargement d'un client est réévalué à 0,0003 seconde (l'explication de ce changement est donné dans le point suivant). En terme de performance, le fait qu'elle soit trois fois plus lente reste imperceptible vu qu'elle est de l'ordre du deux dix-millièmes de seconde, mais le chargement de 6000 clients en une seconde devient impossible.

4.2 Évaluation de la mémoire utilisée

Ce second point complète le premier en donnant une des raisons principales de ralentissement du déroulement de l'exemple. Les programmeurs avaient évalué les performances de la fonction à 0,0001 seconde, ce qui est amplement suffisant pour charger 6000 clients à la seconde, si le système n'est pas chargé et s'il dispose d'une quantité de mémoire infinie.

En simplifiant le problème, imaginons qu'un compte client représente les informations utiles de ce client, son portefeuille d'actions et la valeur actualisée de ses actions. Un compte client prend environs 10 ko, ce qui nous donne une performance disque de 60mo par seconde. Le langage utilisé est Java, qui fonctionne avec un système de pile pour gérer sa mémoire, en ne désalouant celle-ci que lors du passage du garbage collector. Le chargement du client va donc consister en la récupération des informations de la BD sous forme d'un résultset, à l'analyse de ce résultset, au parsing éventuel de certaines données, à l'actualisation de la valeur du portefeuille, à la création des objets de compte ainsi que de la copie des différentes informations provenant de la BD dans les différents attributs des objets.

Charger 6000 clients à la seconde revient à allouer 600 mo de mémoire. Certes cette mémoire sera libérée lors du passage du garbage collector mais celui-ci est très gourmand en ressources et bloque les applications pendant un certain temps afin de nettoyer l'environnement. Les performances au bout de deux ou trois secondes vont donc se dégrader, le respect des 0,0001 seconde peut-être totalement oublié et le système ne sera plus du tout en mesure de charger autant de clients à la fois.

Un autre problème de mémoire est mis en lumière avec le traitement batch. Comme nous l'avons dit, il s'agit d'un programme de gestion de portefeuille d'action d'une banque. En fin de journée, les opérations du jour devaient être analysés. La première idée fut de charger l'intégralité des données en mémoire.

Les indices de performances, évalués à partir de petites quantités de donnée, montraient que la puissance CPU du serveur applicatif était suffisante pour analyser l'intégralité du flux de données dans les temps. Le problème qui apparut ici fut le fait que la mémoire n'était pas suffisante pour gérer les opérations courantes et le traitement de ce batch. Il fallut donc diviser les données en bloque et les calculer séparément, ce qui amenait régulièrement des calculs supplémentaires, déjà effectués mais éliminés de la mémoire. Le temps de calcul fut donc fortement augmenté suite à ce changement de stratégie.

La méthode PRIMA-UML ne tient pas compte de la gestion de la mémoire. Or, en connaissant la taille moyenne des différents objets grâce au Class Diagram, nous sommes en mesure de pouvoir évaluer l'espace mémoire nécessaire pour la création d'un objet. Nous connaissons en effet les attributs de chaque objet ainsi que les relations inter-objets. En complétant les hypothèses par les interviews du client, il serait ainsi possible d'estimer la quantité de mémoire requise par un traitement batch par exemple.

4.3 Gestion de la concurrence

La gestion de la concurrence entre fonctions est un autre défaut de conception qui ne sera pas mis systématiquement en évidence avec la méthode définie. Si la concurrence entre utilisateur utilisant le même noeud est gérée de façon implicite, chaque fonction représentée étant considérée comme une boite noire qui ne peut s'exécuter en parallèle avec d'autres fonctions, la concurrence entre noeud n'est pas prise en compte.

Ainsi, si une fonction bloque une table de base de données en écriture en attendant une confirmation de l'utilisateur, le noeud bloquant la base et le noeud la débloquant seront séparés par des noeuds intermédiaires. Dans une exécution normal, imaginons trois clients A, B et C. A et B désirent passer une commande, C désire consulter un prix. A bloque la base de donnée, le programme attend sa confirmation. Le temps d'attente est évalué à trois secondes. Durant ce laps de temps, soit le CPU est en attente de confirmation et ne peut donc effectuer aucune autre action y compris la consultation d'un prix par le client C, soit le CPU est relâché permettant à C de consulter la base de donnée mais n'interdisant pas à B d'effectuer lui aussi un achat. Dans PRIMA-UML, une ressource est bloquée ou ne l'est pas. Il n'est pas possible de bloquer une partie du disque dur (une table de base de données par exemple) ou certaines fonctionnalités.

Certes, un tel design est mauvais, les systèmes de gestions de concurrences évitent de bloquer tout un système en attendant une confirmation de l'utilisateur, mais le but de PRIMA-UML est de détecter les erreurs de conceptions qui entraîneraient des baisses de performances. Une solution possible à ce problème

serait d'ajouter un attribut à chaque noeud représentant une liste de noeuds qui ne pourraient pas être exécutés tant qu'un autre noeud n'a pas été exécuté.

5 Conclusion

La méthode d'analyse de performance PRIMA-UML semble donner des résultats intéressants. Le nombre d'articles sur le sujet prenant UML comme base pour annoter les schémas de l'application dans l'espoir dans analyser les performances tentent à prouver que cette solution peut donner de bons résultats. Il permet d'évaluer l'influence d'une augmentation de la puissance CPU ou de la vitesse des disques, permettant ainsi de détecter les éventuelles goulots d'étranglements. La prise en compte des connexions réseaux permet aussi d'aider les concepteurs du système à évaluer les besoins en terme de bande passante. Le réseau coûte en effet de plus en plus cher et une surcapacité entraînerait des coûts fixes inacceptables pour les clients.

Cependant, il ne nous fut pas possible de trouver la méthode de prédiction de l'utilisation CPU. Nous mettons en doute la possibilité d'estimer correctement l'utilisation d'un CPU dès les premières phases de l'analyse, cette utilisation dépendant d'une part de la plate forme utilisée mais aussi du langage et du système d'exploitation utilisé voire du framework sur lequel l'application se base. Certes, le développeur peut utiliser le temps CPU estimer comme un résultat à atteindre, mais rien ne garantit que ce soit faisable.

La gestion de la mémoire semble être un point totalement oublié dans l'estimation des performances. Pourtant, ce point peut-être crucial dans des fonctionnalités comme un traitement batch. L'analyse du système peut démontrer que le CPU et l'occupation disque sont suffisants pour traiter un certain nombre d'informations en moins de 24 heures sans pour autant montrer que l'occupation mémoire est totalement irréaliste. L'analyse du diagramme de classe peut aider à estimer cette occupation mémoire. Il nous permet d'évaluer la taille moyenne d'un objet ainsi que les interactions entre les objets du programme, nous permettant ainsi d'estimer, pour un nombre défini d'objets, l'espace en mémoire requis.

La gestion de la concurrence est, selon nous, imparfaite car le graphe d'exécution du programme ne permet pas d'interaction entre plusieurs processus fonctionnant en concurrence. Chaque noeud est considéré comme indépendant des autres noeuds, une ressource est réservée à l'exécution d'un noeud mais ne peut pas être réservée pour certaines actions tout en interdisant d'autres actions.

Références

1. Smith, C.U. (1992) Performance Engineering of Software Systems, *Addison-Wesley, Reading, MA.*
2. Cortellessa, V., Mirandola, R. (2002) PRIMA-UML : a performance validation incremental methodology on early UML diagrams, *Science of Computer Programming*, 44, 101-129.
3. Arief, L.B., Speirs, N.A. (2000) A UML Tool for an Automatic Generation of Simulation Programs, *Submitted to the 2nd International Workshop on Software Performance (WOSP 2000)*.
4. Pooley, R.n King, P. (1999) The Unified Modeling Language and Performance Engineering, *IEE Proceedings — Software*.