Qualification d'architectures fonctionnelles

Démarche et formalismes pour la spécification et la qualification d'architectures fonctionnelles réparties

Marius Bozga — Pierre Combes* — Susanne Graf** — Wei Monin* — Nicolas Moteau***

**France Telecom – R&D – 38 Rue du Général Leclerc, 92 Issy-les-moulineaux*
{pierre.combes, wei.monin ,nicolas.moteau}@francetelecom.com
***VERIMAG, 2 Avenue de Vignate, 38610 Gieres*
{Marius.Bozga, Susanne.Graf}@imag.fr

RÉSUMÉ. *L'objectif des travaux présentés dans ce papier est le développement d'une démarche outillée pour la qualification d'architectures fonctionnelles réparties. La qualification de l'architecture fonctionnelle porte en particulier sur des critères de qualité de service comme les performances et le dimensionnement des ressources. Cette qualification passe par la modélisation fonctionnelle de l'application et de ses sous-systèmes dans une vue structurée et hiérarchisée, en utilisant différents diagrammes d'UML2. L'architecture fonctionnelle du système doit être validée en termes de sûreté de fonctionnement et de conformité, mais également en termes de performances et de dimensionnement des ressources. L'analyse du comportement fonctionnel est basée sur une utilisation des diagrammes d'activités d'UML2, et le papier présente des propositions pour des nouveaux opérateurs permettant une plus grande lisibilité et puissance d'expression. Les diagrammes UML annotés sont alors la base d'une traduction vers des modèles et techniques pour la prédiction de performances.*

ABSTRACT. *The scope of this paper is the development of a method supported by tools for the qualification of distributed functional architectures. Architecture qualification is concerned by the analysis of Quality of service criteria, such as performance, and resource dimensioning. This qualification implies the modelling of the functional architecture and its components, in a structured and step-wise view, using different UML2 diagrams. The functional architecture should be validated in terms of dependability and conformity, but also in terms of performance and resource dimensioning. The functional behaviour analysis is based on the use of UML activity diagrams, and this paper proposes some new operators in order to improve the readability and efficiency of activity diagrams. Finally the annotated UML diagrams are translated towards models for performance evaluation.*

MOTS-CLÉS : *architecture fonctionnelle, UML2, diagrammes d'activité, prédiction de performances, qualité de service*

KEY WORDS: *functional architecture, UML2, activity diagrams, performance analysis, quality of service.*

1. Introduction

Notre objectif se situe parmi les démarches apportant une plus grande maîtrise et une meilleure efficacité dans la réalisation d'applications logicielles. Il vise à intégrer l'analyse de performance dans les décisions techniques et architecturales des différentes étapes de conception et de construction d'une application, à mesure qu'apparaissent les contraintes exprimées par l'ensemble des interlocuteurs.

On constate que de nombreux systèmes sont surdimensionnés sans que l'on sache si cela est dû à des solutions techniques inadaptées ou à une mauvaise répartition des logiciels sur les serveurs. On sait mal prédire les risques de surcharge, alors qu'un tel événement, même local et de courte durée, peut provoquer une congestion et s'étendre en une paralysie généralisée. Il n'est pas rare que des applications logicielles développées ne répondent pas aux exigences de performance et nécessitent d'être reprises depuis la conception. Les techniques d'évaluation de performance sont peu répandues dans le domaine du logiciel et restent confinées à une population d'ingénieurs spécialisés, principalement en raison de la technicité de l'accès aux méthodes et aux outils. Pour mieux intégrer ces techniques à l'ingénierie logicielle, de façon à les rendre accessibles aux architectes de logiciels, il faut établir un fort support méthodologique accompagné d'outils appropriés. Il s'agit d'intégrer les techniques d'analyse de performance avec des techniques pour la validation fonctionnelle dans le cycle de développement de telle sorte qu'elles soient applicables aux productions des différentes étapes du cycle.

Dans un cycle de développement de logiciels, nous pouvons observer que la conception et le développement d'une application font intervenir des interlocuteurs de différentes disciplines, notamment : la Maîtrise d'ouvrage pour l'expression d'un besoin fonctionnel et de l'exigence de performance attendue; l'architecte pour la définition d'une solution répondant le mieux possible à ces besoins; des spécialistes pour l'apport d'une évaluation sur les aspects qualité de la solution proposée: vérification fonctionnelle, performance, disponibilité, fiabilité, sécurité, tests de validation, etc. ; des ingénieurs de développement pour la réalisation de code. Ensuite, les différents interlocuteurs doivent échanger et accorder leur point de vue sur l'application. Dans ce but, nous proposons une démarche et un support pour la construction des spécifications d'architecture fonctionnelle et d'architecture logicielle technique. Chaque production pourra être contrôlée et évaluée qualitativement et quantitativement, et par la dérivation de modèles de simulation et de vérification fonctionnelles ou de performance adaptés (voir Figure 1).

Le chapitre 2 présente une démarche rigoureuse permettant de composer et structurer progressivement l'architecture fonctionnelle de l'application de manière hiérarchique en partant des scénarios d'utilisation des services de l'application. Le chapitre 3 présente les techniques de modélisation de l'architecture fonctionnelle basées sur la combinaison de diagrammes UML2. Le chapitre 4 présente l'utilisation des diagrammes d'activités pour cette modélisation. Il présente une proposition de nouvelles constructions pour ces diagrammes. Le chapitre 5 présente des annotations

qui doivent être associées aux différents diagrammes afin de permettre une analyse des performances du système en fonction des ressources utilisées.

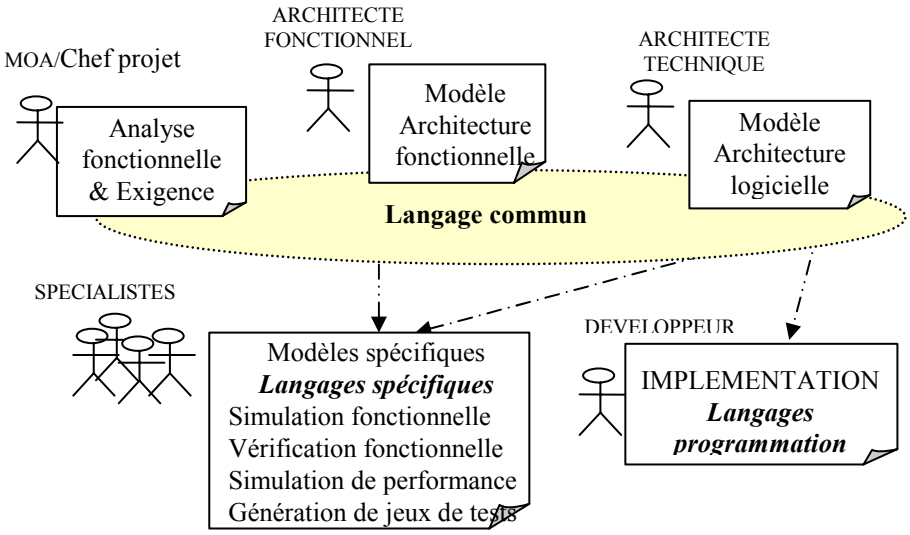

Figure 1: Intervenants et modèles dans le cycle de développement de logiciels

2. Architecture fonctionnelle structurée et hiérarchique

Une architecture fonctionnelle doit prendre en compte une découpe en sous-systèmes ainsi que les interactions définissant la manière dont les sous-systèmes coopèrent pour la réalisation des services offerts par l'application. Les sous systèmes peuvent être des logiciels à développer ou des logiciels existants à intégrer. Nous proposons une procédure rigoureuse permettant de composer et structurer progressivement l'*architecture fonctionnelle* de l'application de manière hiérarchique. En partant des scénarios d'utilisation des services de l'application, on identifie les fonctions composantes et on les regroupe par entité fonctionnelle. Une entité fonctionnelle est une entité qui supporte une ou plusieurs fonctions composantes, et peut être: *un processus, une plate forme, un groupe d'applications ou une application*. Chaque fonction composante est raffinée en fonctions composantes plus fines, elles-mêmes regroupées par entité fonctionnelle: *sous processus, groupe d'applications* ou *application*, et ainsi de suite, pour aller éventuellement jusqu'au niveau d'abstraction des *serveurs*. En procédant ainsi, on arrive, à la fin du processus, à identifier les fonctions composantes des modules (ou applications) logiciels. Le choix de la granularité des fonctions composantes est du ressort de l'architecte qui prend en compte l'importance du niveau d'abstraction qu'il souhaite et l'information dont il dispose.

Une fonction composante est une *fonctionnalité* abstraite qui a sa propre cohérence et, combinée avec d'autres fonctions composantes, est supportée par une entité

fonctionnelle. Par exemple, un sous système est une fonction composante du système englobant, une méthode est une fonction composante du serveur qui l'implémente, etc. La définition est très générique et sa granularité peut aller d'une méthode à un processus. Pour identifier les fonctions composantes et les interactions permettant de les structurer, on s'appuie sur les scénarios d'utilisation. En effet, chaque scénario correspond à un déroulement séquentiel, ou en parallèle avec éventuellement des points de synchronisation, des fonctions composantes. Un tel enchaînement est appelé *flot d'exécution* (workflow). L'identification des fonctions composantes est déterminée par les flots d'exécution.

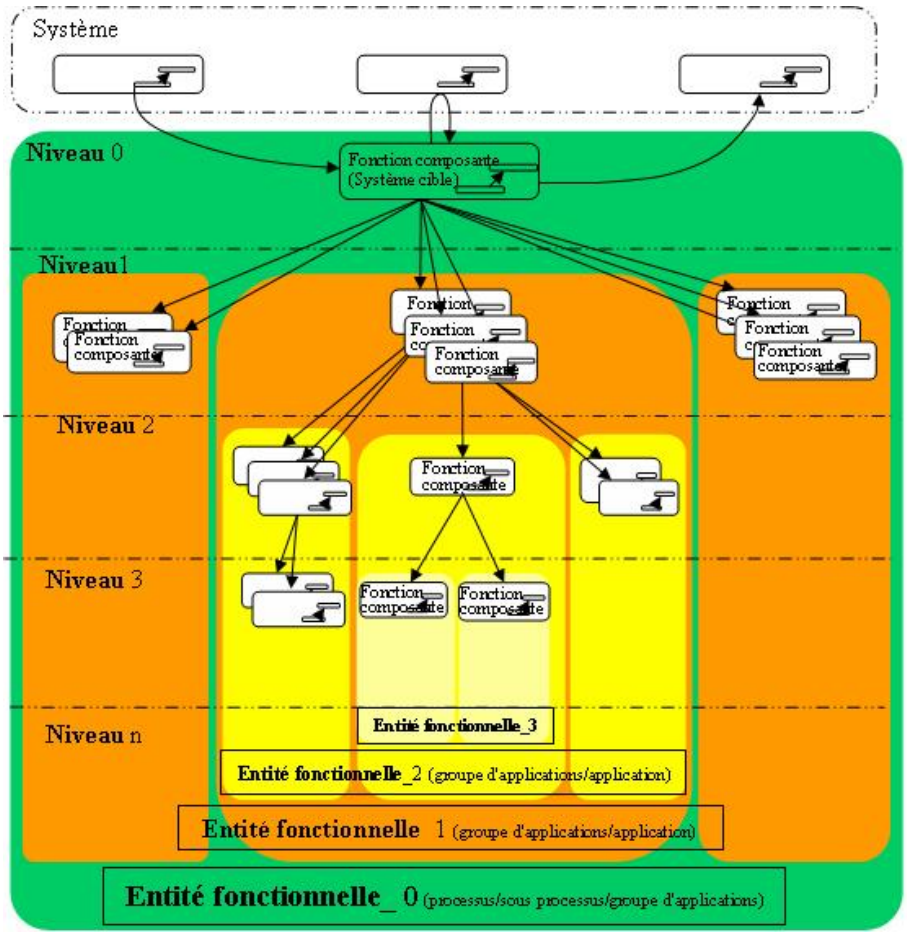

Figure 2 : procédé de la décomposition et du regroupement des fonctions composantes par niveaux d'abstraction et par entités fonctionnelles

Les flots d'exécution sont eux-mêmes identifiés en regroupant les requêtes utilisateur de l'application invoquant le même service et de même profil (attributs caractérisant

le comportement propre de l'utilisateur). Le nombre de services fournis par l'application et le nombre de profils des requêtes étant finis, les requêtes peuvent être classées en un nombre fini de groupes. Chaque groupe ainsi identifié est associé à un flot d'exécution, qui est l'enchaînement d'exécutions de fonctions composantes induit par une requête du groupe. En partant d'une requête de chaque groupe, on décompose progressivement l'application en sous systèmes ou fonctions composantes, puis sous fonctions, etc.

Les flots d'exécution sont présentés sous forme d'un graphe de fonctions composantes reliées entre elles à l'aide de structures de contrôle (si … alors … sinon, tant que, etc.). Chaque fonction composante peut à son tour se décomposer en sous fonctions, et ainsi de suite de manière incrémentale.

Les fonctions composantes identifiées suivant l'étape précédente sont ensuite structurées de telle sorte que, d'une part, on arrive à identifier la répartition des fonctions composantes sur les applications à intégrer ou à développer; et d'autre part, on y retrouve les flots d'exécution associés aux scénarios. Nous proposons, pour l'architecture de l'application, une structure hiérarchique et un procédé de regroupement des fonctions composantes par entités fonctionnelles pour chaque niveau d'abstraction.

Pour chaque niveau d'abstraction, l'on doit identifier les entités fonctionnelles qui supportent les fonctions composantes de ce niveau, identifier les entités fonctionnelles qui regroupent et supportent les fonctions composantes externes (en amont, en aval du système cible et celles qui vont être sollicitées et qui renvoient des réponses attendues par le système cible). Il faut également identifier les interactions potentielles entre ces entités fonctionnelles, et les données qui vont être manipulées lors des interactions. Le procédé est illustré schématiquement dans la Figure 2.

3. Formalisme pour la modélisation de l'architecture fonctionnelle

Afin que les dossiers d'architecture fonctionnelle puissent permettre la communication entre acteurs d'expertises différentes, ces dossiers doivent être construits avec des langages et techniques de modélisation formelles. Nous proposons une technique de modélisation de l'architecture fonctionnelle, basée sur la combinaison des diagrammes de classe, d'architecture et d'activités enrichis d'UML2, et ce pour chaque niveau d'abstraction et pour chaque sous système.

Chaque entité fonctionnelle impliquée (processus, sous processus, plates formes, applications ou groupe d'applications) est déclarée sous forme de diagramme de classe et les fonctions composantes qu'elle supporte sont considérées comme les opérations de la classe (voir Figure 3). Lorsque les entités fonctionnelles correspondent à des plates formes ou à des applications logicielles, leurs fonctions composantes représentent des opérations réalisées par des méthodes

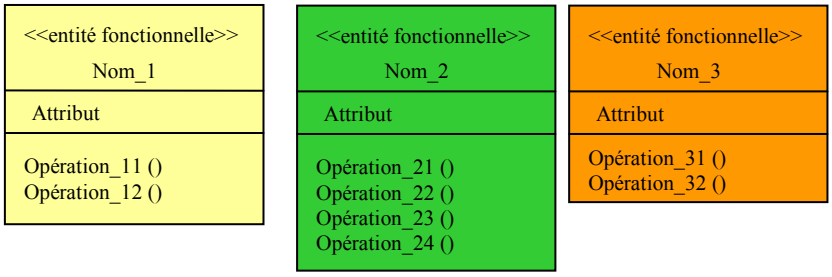

Figure 3 : Entités fonctionnelles et ses fonctions composantes

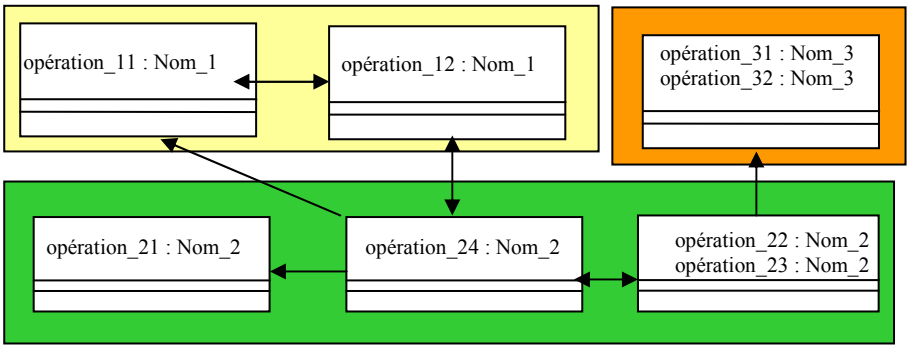

Figure 4 : Relations de dépendance entre les entités fonctionnelles

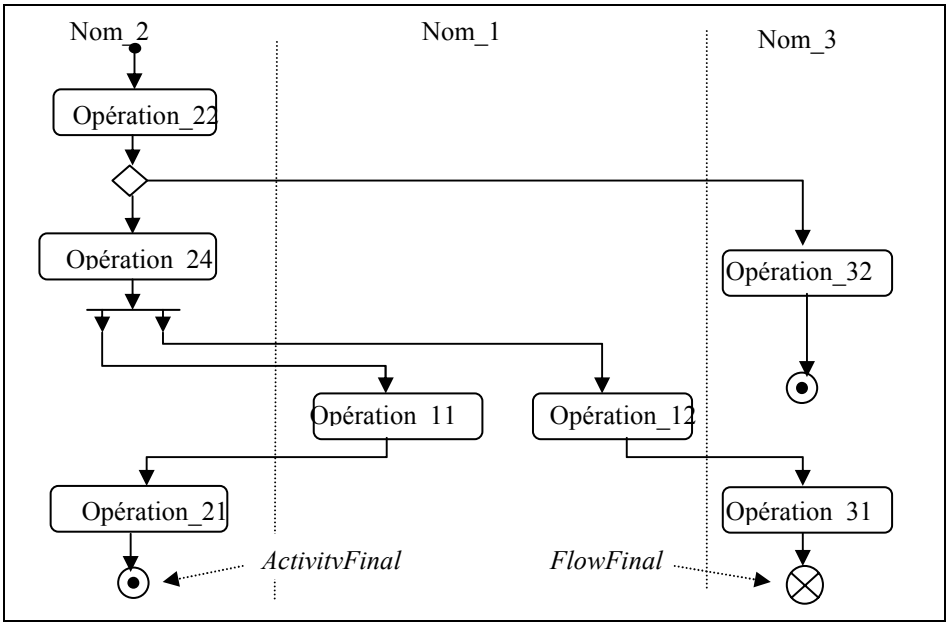

Figure 5 : modélisation du comportement dynamique

Un diagramme d'architecture relie des instances des classes suivant les relations de dépendance entre les entités fonctionnelles du sous système. C'est un graphe comportant, d'une part, des instances de classes "entités fonctionnelles" regroupant des opérations définies dans la classe (fonctions composantes de l'entité fonctionnelle), et d'autre part, des flèches qui relient les instances selon une configuration (voir Figure 4). En particulier, lorsque les entités fonctionnelles correspondent à des plates formes ou des applications logicielles, un tel graphe représente l'architecture fonctionnelle du sous système

Le comportement dynamique est modélisé sous forme de diagramme d'activités, l'entité fonctionnelle est représentée par une partition et ses fonctions composantes par des activités et se trouvent toutes dans la même partition (**Figure 5**). Nous obtenons ainsi la complémentarité des aspects statiques et dynamiques, ainsi que leur cohérence dans la spécification d'architectures fonctionnelles.

4. De l'utilisation des diagrammes d'activités

Nous nous focaliserons dans ce papier sur la notion d'activité composite (sous activité). Une telle construction joue un rôle primordial pour raffiner progressivement le comportement du système ou des fonctions composantes. Il s'agit d'encapsuler un ensemble de comportement dans une activité, celle-ci devra comporter un état d'entrée (InitialNode) correspondant aux flots qui activent le comportement interne, et un ou plusieurs états de sortie correspondants à différentes fin d'exécution du comportement de l'activité. UML2 propose deux états de sortie "Final Node", appelés "ActivityFinal" et "FlowFinal" (Figure 5). Lorsqu'un flot d'exécution se termine par "ActivityFinal", l'activité se termine et tous les autres flots d'exécution sont également terminés même s'ils étaient en cours d'exécution. L'activité terminée, le flot d'exécution peut se continuer dans le diagramme englobant. Le nœud "FlowFinal" termine seulement le flot d'exécution concerné par ce nœud, les jetons (dans une vue réseau de Petri) sont détruits. Si l'activité dispose d'un nœud "ActivityFinal", elle pourra se terminer sur ce nœud en laissant se continuer un flot d'exécution dans le diagramme englobant. Comme exemple, prenons une application de recherche de services de dépannage: l'application peut, pour accélérer la recherche, lancer simultanément plusieurs requêtes à différentes sociétés et, dès la première réception d'une réponse, l'application arrête toutes les autres recherches en cours. Dans cet exemple, l'application ne peut déterminer le nombre de requêtes que lors de l'exécution et elle arrête toutes les autres recherches (synchronisation au plus tôt) à un moment déterminé à l'exécution. Ce cas est modélisable par la partie gauche de la Figure 6. Après avoir déterminé le nombre de requêtes à envoyer, l'application déclenche aussitôt N émissions de demande et c'est la séquence ayant reçu en premier la réponse qui va interrompre toutes les autres recherches en cours et terminer ainsi la procédure.

Cependant, les primitives de type "Final Node" des diagrammes d'activités définies dans la norme UML2 sont insuffisantes pour exprimer toutes les constructions désirées. On a fréquemment besoin de représenter des comportements dans lesquels

un flot d'activité entrant dans un diagramme donne lieu à plusieurs flots en parallèle dont les issues sont indépendantes les uns des autres. Nous proposons donc un **enrichissement des diagrammes d'activité UML** par l'introduction de deux nouvelles primitives, *FlowRetour* et *FlowSynchrone*. Lorsqu'un flot atteint un *FlowRetour,* il sort du diagramme englobé où il se trouve et se poursuit immédiatement dans le diagramme d'activité englobant, indépendamment des autres flots créés en parallèle. Dans l'exemple de la Figure 5, les deux flots d'exécution du bas de la figure (Opération_21 et Opération_31) s'exécutent en parallèle, et dans le comportement souhaité, ces flots doivent rester indépendants dans l'activité englobante. Nous remplaçons alors les deux nœuds de sortie par un nœud *FlowRetour*.

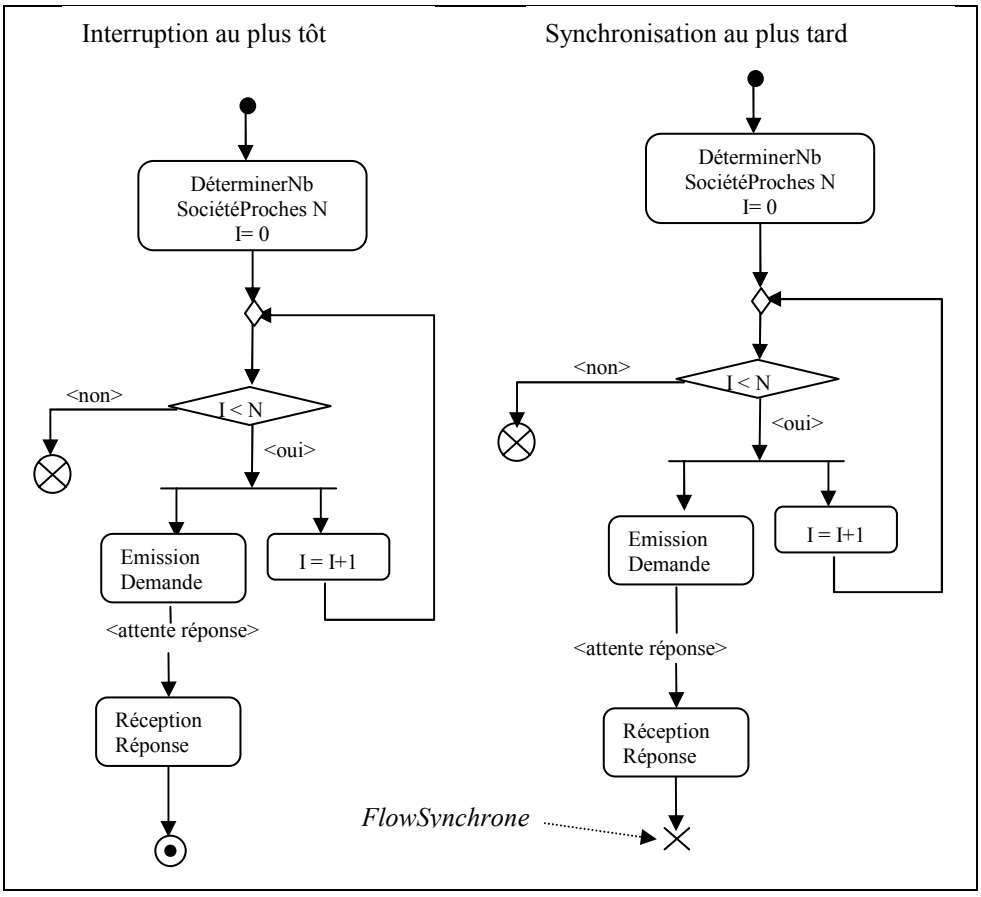

Figure 6: Synchronisation de flots d'exécution

Lorsqu'un flot atteint un *FlowSynchrone,* il se termine et se mets en attente. L'activité ne sera alors terminée que lorsque tous les flots d'exécution s'exécutant en parallèle à

l'intérieur de l'activité auront atteints un tel nœud (ou un autre nœud final). Par symétrie à l'interruption au plus tôt des séquences en parallèle, on peut rencontrer aussi des applications qui nécessitent une synchronisation de plusieurs séquences d'exécution en parallèle au plus tard. Prenons l'application de recherche de services de dépannage. Une autre version pourrait lancer les requêtes en parallèle, mais attendre que toutes les requêtes aient émis une réponse (positive ou négative) avant de faire un choix. C'est la séquence ayant envoyé en dernier le résultat de recherche qui termine la procédure de recherche globale (voir la partie droite de la Figure 6).

Figure 7: Représentation graphique des nouveaux nœuds de retour

Des travaux portent sur la formalisation des diagrammes d'activité [ESH 04, LOP 04, BOR 00] en particulier vers de réseaux de Petri ou des "Abstract State Machine" (ASM). Cependant, ces travaux n'abordent pas la notion de sous activité, et en conséquence la structuration hiérarchique des diagrammes d'activité.

5. Dérivation vers un modèle de performance

Dès la construction du premier modèle fonctionnel, on peut élaborer, à partir de celui-ci, des modèles de simulation de performance pour évaluer l'impact des configurations architecturales. La première étape consiste à identifier les flots d'exécution (scénarii) pertinents par rapport à l'objectif de l'étude de performance, en se basant sur les Use Cases dont la fréquence d'apparition dans le déploiement futur est non négligeable et qui impliquent des actions consommant de façon significative du CPU ou de la mémoire. Il peut également être intéressant de simplifier la présentation en faisant abstraction du contenu fonctionnel non pertinent d'un point de vue performance [COM 04]. La dérivation vers un modèle de performance sera basée sur l'utilisation des différents diagrammes UML2, assortis d'annotations.

Annoter les flots d'exécution par les informations quantitatives pertinentes pour l'analyse de performance. Il s'agit d'annoter les activités de chaque diagramme avec des informations caractérisant la consommation des ressources, telles que la durée d'occupation de CPU et le volume de mémoire utilisé lors de l'exécution d'une opération. Ce que l'on appelle le temps de traitement unitaire d'une action correspond à l'intervalle de temps durant lequel la ressource est entièrement dédiée à l'exécution de l'action. A noter que dans les phases amont, la précision concernant la durée d'utilisation de CPU ou le volume de mémoire occupé est issue d'estimations très relatives et n'a de sens que par rapport à la granularité des opérations considérées. A mesure que l'on avance dans la conception, les modèles fonctionnels et de performance s'affinent. Les ressources physiques sur lesquelles s'exécutent des activités du processus peuvent être identifiées en tant qu'acteurs. Des données plus précises peuvent alors être injectées, grâce à des tests unitaires sur des prototypes ou sur l'implémentation. Les annotations quantitatives sont associées logiquement aux activités simples de chaque diagramme. Il est également permis

d'annoter une activité composée, dans le cas où c'est une abstraction effectuée délibérément. Cette activité sera alors considérée comme une boîte noire. Les annotations seront représentées graphiquement par des *Notes* UML (voir Figure 8). Leur syntaxe est donnée par des règles dont un exemple est donné ci-dessous (CPU pouvant être remplacé par d'autres types de ressources).

<annotation> ::= CPU [<expression-prob>]
< expression-prob > ::= <loi>(<liste de paramètres>) | <constante>
<loi> ::= uniform | exponential | Poisson | …
<paramètre> ::= <constante>
<constante> ::= <integer> | <float>

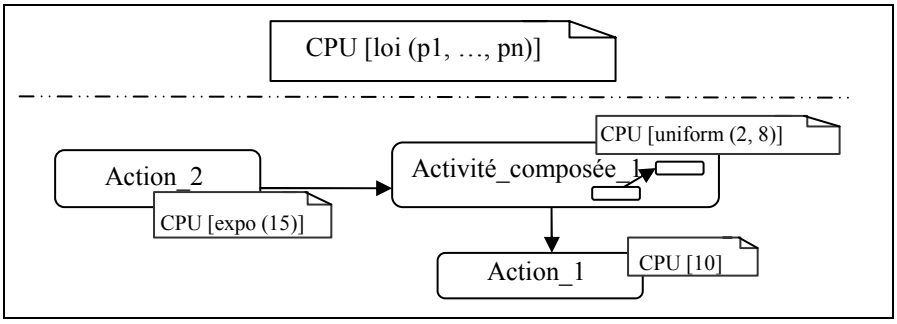

Figure 8: annotations pour les performances

Compléter par un modèle d'environnement et par un modèle de déploiement.

Modèle d'environnement (utilisateurs du système). Le modèle d'environnement spécifie notamment le débit et le rythme d'arrivée des requêtes client, ainsi que la répartition en pourcentage de ces requêtes dans les différents groupes correspondant aux flots d'exécution. On peut aussi avoir des modèles de comportement d'utilisateurs plus complexes où chaque utilisateur peut émettre de manière aléatoire plusieurs requêtes sur un intervalle de temps (une session par exemple).

Modèle de ressources Le modèle de ressources est une abstraction des ressources physiques sur lesquelles s'exécute l'application. Les ressources sont généralement les machines physiques. Cependant, comme évoqué précédemment, il se peut, dans une spécification fonctionnelle relativement amont, que l'on ne dispose pas d'informations suffisamment précises sur les machines impliquées. Nous distinguons donc deux types de ressource: celles qui correspondent à des machines physiques et celles qui correspondent à des supports applicatifs. Par exemple, lorsque l'application est distribuée, le middleware et les réseaux de transport sont considérés comme support applicatif car on regroupe les bandes passantes, les commutateurs, les protocoles et les mécanismes de contrôle en un seul bloc fonctionnel. Cette distinction est nécessaire car leur caractérisation est bien différente. Une ressource physique est caractérisée essentiellement par le nombre et la vitesse de calcul des processeurs, la politique d'ordonnancement, la gestion de priorité, etc. En revanche,

un support applicatif est caractérisé essentiellement par un délai représentant sa capacité de traitement et correspondant, pour chaque activité concurrente à la ressource, à la durée de traitement de l'activité. La spécification des ressources peut être représentée sous forme de *classes*, où l'on précise, à l'aide des attributs, le type de ressource. Un acteur peut être associé à plusieurs ressources. Pour chaque ressource machine, on indique le nombre de processeurs, la politique d'ordonnancement, la gestion de priorités. Pour une ressource applicative, on peut spécifier la discipline de traitement: par exemple, lorsque la ressource doit être affectée à deux traitements se présentant en même temps, le délai de traversée sera soit le délai unitaire pour chacun si la ressource est capable de les absorber indépendamment, soit le double du délai unitaire sinon, soit au délai unitaire pour un d'entre eux et au double pour l'autre si les traitements sont sériés.

La construction du modèle de simulation de performance à partir des flots d'exécution sélectionnés, du modèle d'environnement et du modèle de déploiement peut alors être automatisable. Il s'agit d'établir des règles de correspondance entre les primitives de diagrammes d'activités représentant les flots d'exécution et les primitives dans un modèle de files d'attente utilisé par un simulateur de performance (Hyperformix/workbench, QNAP par exemple). Des prototypes de telles traductions sont réalisés à partir de diagrammes d'activités et de diagrammes d'états [COM 04, CAS 05]. D'autres travaux ont abordés la traduction de modèles UML vers des techniques de prédiction de performance. On peut citer les travaux de l'université de Carleton [WOO 02, XU03, PET 03], et d'universités en Italie [AMB 03, DIM 04, MAR 04], cependant ces travaux n'abordent pas la complexité des diagrammes due à la hiérarchie et la notion de sous activités.

6. Conclusion

Les travaux présentés dans ce papier permettent une modélisation hiérarchique d'architecture fonctionnelle. Cette modélisation est simple et facile d'accès, les formalismes proposés apportent un confort appréciable pour saisir visuellement la logique fonctionnelle, notamment sur des cas d'utilisation. Elle favorise la réutilisation et le partage effectif d'informations entre les interlocuteurs de cultures différentes impliqués dans le processus de développement**.** La rigueur sémantique confère la précision dans la compréhension et dans l'échange. L'interprétation du modèle à partager entre les interlocuteurs sera cohérente et univoque, et ce dernier pourra être codé. Elle offre la souplesse de pouvoir travailler à différents niveaux d'abstraction, et ce au choix des utilisateurs. La structure hiérarchique de l'architecture fonctionnelle permet une évolution progressive et cohérente vers la construction d'architectures plus techniques. L'introduction de deux nouvelles primitives dans les diagrammes d'activités joue un rôle clef pour supporter une telle structuration. Elle est adaptée à l'introduction d'outils de support**.** Les formalismes proposés permettent notamment la dérivation de modèles de simulation de performance et on peut leur appliquer des outils éprouvés de simulation, vérification et validation formelle, de génération automatique de jeux de tests, etc. Des travaux

sont en cours dans le cadre du projet RNRT/PerSiForm [PER 05], afin, d'une part de donner une sémantique formelle aux diagrammes d'activités en se basant sur les réseaux de Petri colorés stochastiques, d'autre part de réaliser des traducteurs entre les diagrammes d'activités, les réseaux de Petri et finalement les formalismes d'entrée des simulateurs événementiels pour la prédiction de performances. De même, la formalisation par des réseaux de Petri permettra d'obtenir, en ignorant les annotations de performance, des modèles pour la validation fonctionnelle.

Bibliographie

[AMB 03] D'Ambrogio A., Iazeolla G., Steps Towards the automatic production of performance models of WEB Applications, Computer Network, vol 41, 2003.

[BOR 00] Borger, E., Cavarra A., Riccobene E., An ASM Semantic for UML Activity Diagrams, Proceeding Algebraic Methodology and Software Technology, AMAST 2000, LNCS vol 1816, Springer Verlag, 2000.

[CAS 05] Castanet R., Cavalli A., Combes P., Laurençot P., Mackaya M., Mederreg A., Monin W., Zaïdi F., Une plate-forme de validation multi-protocoles et multi-services – Résultats d'expérimentation, Annales des Télécommunications, vol 60, N° 5-6, Juin 2005.

[COM 02] COMBES P., MONIN W., Modèles fonctionnels et modèles de performance, application aux services de localisation, NOTERE 2004.

[DIM 04] Di Marco A., Inverardi P., Compare D., D'Onofrio A., Automated Performance Validation of Software Design: An Industrial Experience, proceedings 19th IEEE International Conference on Automated Software Engineering, Septembre 2004.

[ESH 04] Eshuis R., Wieringa R., Tool Support for Verifying UML Activity Diagrams, IEEE Transactions on Software Engineering, vol 30, N° 7, Juillet 2004.

[LOP 04] Lopez-garo J.P., Merseguer J., Campos J., From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering, WOSP 2004.

[MAR 04] Marzolla M., Simulation-based Performance Modeling of UML Software Architectures, PHD, Université de Venise, 2004.

[PER 05] Projet RNRT/PerSiForm, "http://www-persiform.imag.fr/"

[PET03] D.B. Petriu, D. Amyot, M. Woodside, Scenario-Based Performance Engineering, SDL Forum 2003.

[UML 04] UML2.0 Superstructure Specification, http://www.omg.org

[WOO 02] M. Woodside, Tutorial Introduction to layered modeling of software performance, http://www.sce.carleton.ca/rads/lqn/lqn-documentation/

[XU03] Jing Xu, Murray Woodside, D.C. Petriu "Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time", Proc. 13th Int Conf. on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS 2003), Urbana, Illinois, USA, Sept 2003, vol. LNCS 2794.