HAL
open science

Aide au développement incrémental et à la vérification d'architectures logicielles

Than-Liem Phan, Thomas Lambolais, Anne-Lise Courbis

To cite this version:

Than-Liem Phan, Thomas Lambolais, Anne-Lise Courbis. Aide au développement incrémental et à la vérification d'architectures logicielles. Lambda Mu, 2012, France. Communication 8-B2 7p. hal-00797470

HAL Id: hal-00797470

https://hal.science/hal-00797470v1

Submitted on 6 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Titre : Aide au développement incrémental et à la vérification d'architectures logicielles

Supporting Incremental Development and Verification of Software Architectures

Thanh-Liem Phan, Thomas Lambolais et Anne-Lise Courbis
LGI2P, École des Mines d'Alès
Adresse : Site de Nîmes – Parc Georges Besse, F 30035 Nîmes cedex 1
Téléphone : +33 (0) 4 66 38 40 14, fax : +33 (0) 4 66 38 70 74
Email : {thanh-liem.phan ; thomas.lambolais ; anne-lise.courbis}@mines-ales.fr

Objectifs

Notre objectif est d'aider au développement de modèles architecturaux, en fournissant des techniques et outils de détection d'erreurs pendant les phases de spécification et de conception. Les modèles réalisés portent sur les aspects comportementaux de systèmes réactifs : nous cherchons à savoir si les propriétés de vivacité des modèles sont préservées durant la mise au point et l'enrichissement des modèles.

Contexte

Les systèmes critiques sont des systèmes dans lesquels certaines défaillances sont inacceptables. Nous qualifions de plus ces systèmes de *réactifs* s'ils sont en interaction continue avec leur environnement, à une vitesse imposée par l'environnement. La réactivité des systèmes se traduit par des propriétés de vivacité (« l'airbag se déclenche effectivement en cas de choc »). La criticité des systèmes se traduit quant à elle plutôt par des propriétés de sûreté (« les portes de l'ascenseur, à l'étage, ne s'ouvrent pas si la cabine n'est pas présente »).

La part du logiciel étant croissante dans de tels systèmes, leur fiabilité repose de plus en plus sur la fiabilité du logiciel. Or, le développement de logiciels corrects *a priori* est encore un défi scientifique [3]. D'autre part, les techniques de test logicielles, bien que nécessaires, resteront à jamais insuffisantes pour garantir la fiabilité du logiciel. De plus, ces techniques sont excessivement coûteuses.

Ces dernières années ont vu émerger la notion de développement agile et d'extreme programming [7], ce qui rejoint les cycles de développement en spirale et le prototypage rapide. Par ailleurs, les méthodes formelles, sur lesquelles beaucoup d'espoirs reposaient dans les années 1980 et 1990, n'ont pas connu le développement escompté. Le développement de modèles formels de spécification et de réalisation a été confronté à plusieurs objections industrielles : réaliser des modèles formels en supplément des développements habituels est délicat et coûteux, mais de plus toute garantie de bon fonctionnement obtenue sur les modèles formels, n'a de sens que si les modèles sont valides, ce qui ne peut être établi que de façon informelle [3]. Face à ces critiques, le développement de modèles et de code par raffinements successifs est une première réponse intéressante : les modèles formels sont pleinement intégrés au développement, ce qui répond au surcoût de développement mais aussi en partie aux préoccupations de validation. Les vérifications et preuves sont conduites progressivement, sur des modèles de plus en plus élaborés, ce qui facilite leur mise en œuvre. Cependant, la mise au point des premiers modèles, qui doivent englober l'ensemble des fonctionnalités souhaitées mais de façon abstraite, est très délicate. De plus, les développements par raffinements successifs correspondent à un cycle en cascade, ou éventuellement en V, où la réalisation aboutie est repoussée à la fin. Ceci est générateur de stress important pour les équipes de développement, ne facilite pas les interactions avec les clients et ne s'adapte pas à la demande des marchés visant des cycles de développement courts. C'est en opposition avec les principes du développement agile.

Nous préconisons une démarche de développement *incrémental*, ce qui permet de tirer profit de techniques de vérification formelle dans une démarche agile. La différence principale avec les techniques de raffinement classiques est que les premiers modèles ne sont pas nécessairement des modèles englobants : ils peuvent correspondre à une description partielle, où seules quelques fonctionnalités et caractéristiques ont été prises en compte. Contrairement au raffinement, il faudra donc être capable de compléter les modèles en offrant de nouveaux comportements, ceci au risque de violer des propriétés de sûreté (puisque les nouveaux comportements figurent peut-être parmi les interdits). Les

propriétés que nous chercherons dans tous les cas à garantir entre deux incréments successifs sont celles de vivacité.

Afin de ne pas multiplier les langages de modélisation et de faciliter les travaux de développement, nous ne voulons pas décrire explicitement les propriétés à préserver. Les vérifications que nous préconisons consistent à comparer un modèle étendu ou détaillé au modèle dont il est issu. Nous avons étudié pour cela l'utilisation de relations de conformité comportementale.

Nous nous intéressons à la construction de modèles architecturaux de systèmes critiques-réactifs exprimés sous forme d'assemblage de composants UML. Le comportement des composants est représenté par des machines d'états UML.

Méthode

Dans le contexte des architectures logicielles, une amélioration peut être le fait d'ajouter un nouveau composant ou de remplacer un composant par un autre. Afin d'assurer que l'ajout ou la substitution d'un composant ne cause pas de risques pour l'architecture de systèmes, l'analyse d'informations statiques (les interfaces avec les contrats) du composant n'est pas suffisante. Ceci ne prévient pas les risques de blocage et de refus. Nous devons considérer non seulement les interfaces du composant mais aussi son comportement.

Dans notre travail, les risques mis en évidence dans les architectures sont ceux de possibilité de *refus*. Un refus correspond au fait que le système ne répondra pas à certains des stimuli auxquels il devait répondre. À l'extrême, si le modèle refuse tout stimulus, ceci conduit à l'un des deux cas suivants : blocage ou live-lock critique. Un live-lock est critique s'il est impossible d'en sortir.

L'application d'une démarche incrémentale au développement de modèles d'architectures logicielles se traduit de la façon suivante :

- Nous partons d'un premier modèle partiel, dont on vérifie l'absence de blocages et de live-lock critiques ;
- Nous faisons évoluer ce modèle par l'une des opérations suivantes :
    - ajout d'un nouveau composant ;
    - substitution d'un composant par un autre composant ou par un assemblage de composants ;
    - scission d'un composant en plusieurs composants ;
    - reconfiguration de l'architecture.
- À chaque étape, nous vérifions si la version courante du modèle architectural est en *accord* avec la version précédente. Cette comparaison se fait par une relation de conformité. Plusieurs relations de conformité sont utilisables. Nous les présentons ci-dessous.
- Si la comparaison fait apparaître un problème, nous avons deux réactions possibles :
    - Le concepteur du modèle découvre une de ses erreurs et cherche à la corriger. Il peut s'aider pour cela du verdict d'erreur donné par l'outil, présentant les scénarios d'interactions conduisant à des situations où son modèle refuse des actions qu'il ne devrait pas rejeter;
    - Le concepteur du modèle a conscience d'avoir introduit une rupture de développement. Il souhaite préserver cette différence entre les modèles. Dans ce cas, il peut décider, soit de définir le modèle courant comme étant le nouveau modèle de référence, soit de corriger le modèle précédent.

Les relations de conformité que nous utilisons ont été formalisées par Brinksma [4] et Leduc [8] sur des systèmes de transitions étiquetées (*Labelled Transition Systems*, LTS). Elles traduisent l'idée du test de conformité dont la méthodologie a été définie par l'ISO. Informellement, une implantation est conforme à sa spécification si elle implante correctement toutes les parties nécessaires de la spécification. Une spécification peut en effet être indéterministe : suite à une séquence d'interactions observables (ce que l'on appelle une trace) elle définit des réponses nécessaires, mais également des réponses juste *possibles*, qu'il est donc possible également de refuser.

- Plus formellement, un modèle M' est *conforme* à un modèle M si après toute trace de M, M' doit accepter toute action que M doit accepter. Un modèle M' conforme à un modèle M respecte donc toutes les propriétés de vivacité de M. M' est plus déterministe que M.
- Si M' est conforme à M et possède plus de traces que M, M' est une *extension* de M.
- Si M' est conforme à M et possède moins de traces que M, M' est une *réduction* de M.
- Si tout modèle conforme à M' est aussi un modèle conforme à M, M' est un *raffinement* de M. Le raffinement ainsi défini diffère du raffinement classique défini par exemple dans la méthode

B. Ici le raffinement préserve la vivacité alors que le raffinement en B préserve la sûreté. La relation d'extension est une relation de raffinement. C'est ce que nous appelons le *raffinement horizontal*. La relation de réduction n'est pas une relation de raffinement. La relation que nous gardons comme *raffinement vertical* est l'intersection entre la relation de réduction et la relation de raffinement. Nous obtenons une relation qui préserve les propriétés de vivacité mais aussi celles de sûreté.

Exemple 1

Soit la machine d'états d'un composant SpecJobber, présentée à la Figure 1.

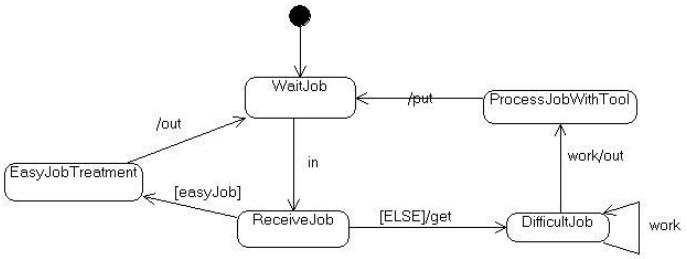

Figure 1 Machine d'états de SpecJobber

SpecJobber spécifie un composant réalisant un service pour son environnement : il reçoit des travaux (in) qu'il réalise et renvoie (out). Si les travaux le nécessitent, le composant peut utiliser un outil qu'il prend et repose (opérations get et put). L'outil demande au composant de travailler (opération work) et de manière indéterministe, l'une de ces actions met fin au travail.

Une réalisation envisageable de SpecJobber est un composant Jobber conçu pour fonctionner comme SpecJobber, mais pouvant également garder l'outil pour réaliser plusieurs travaux consécutivement (Figure 2). On peut voir que tous les comportements de SpecJobber sont inclus dans Jobber. Au sens de la simulation observationnelle de Milner [10], Jobber simule SpecJobber. Néanmoins, Jobber n'est pas conforme à la spécification SpecJobber. Après la trace in ; get ; work* ; out, le composant SpecJobber doit réaliser put alors que Jobber peut refuser cette action put.

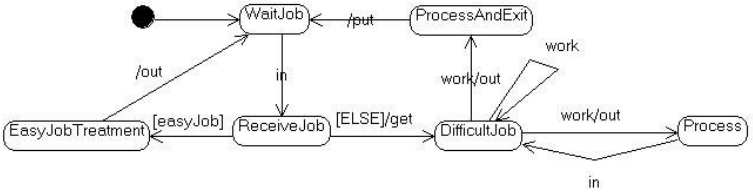

Figure 2 Machine d'états de Jobber

Il est toutefois possible d'étendre correctement la spécification pour réaliser plusieurs traitements consécutivement, en procédant comme à la Figure 3. Le nouveau comportement est conforme à celui de SpecJobber. Ainsi défini, il est en extension ou encore correspond à un raffinement horizontal de la spécification.

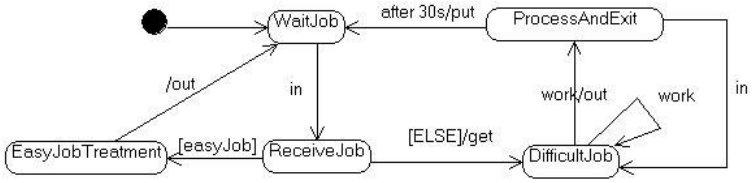

Figure 3 Machine d'états corrigée de Jobber

Construction incrémentale d'architectures

Dans une démarche de développement incrémental, on souhaite progresser par raffinement vertical et raffinement horizontal. Il est ainsi possible de compléter les modèles en ajoutant de nouveaux com-

portements, horizontalement, comme de dériver des modèles plus précis conduisant à des modèles d'implantation, verticalement. Le fait de pouvoir dériver des implantations durant le développement (correspondant à des jeux de fonctionnalités et de comportements réduits), apporte ainsi les avantages des méthodes agiles.

Les problèmes à résoudre pour pouvoir conduire des développements incrémentaux d'architectures de composants UML sont alors les suivants :

i) Donner une sémantique opérationnelle aux machines d'états UML, aux composants et aux architectures UML.
ii) Assurer le raffinement vertical lorsqu'on substitue un composant par un composant qui le raffine.
iii) Assurer le raffinement horizontal lorsqu'on ajoute un nouveau composant.

Nous ne considérons pas ici la reconfiguration d'une architecture ni la scission d'un composant.

i) Sémantique opérationnelle des architectures UML

Dans un travail précédent[9], nous avons défini la sémantique des composants par la transformation de machines d'états UML en systèmes de transitions étiquetées. Dans le cas d'une architecture, les compositions entre composants sont données sur l'algèbre de processus LOTOS. En LOTOS, les communications correspondent à des rendez-vous synchrones. Pour des communications asynchrones, il est nécessaire de définir explicitement des processus représentant des files d'attente. Nous avons ainsi défini une transformation des structures composites UML en spécifications Exp.Open [6]. Grâce à la sémantique de LOTOS donnée sur des LTS, il est alors possible de générer un seul LTS pour l'ensemble du comportement d'une architecture UML. Ceci nous permet d'analyser le comportement global de l'architecture initiale, dans le but d'y vérifier l'absence de blocage et d'interblocage.

Notons que ceci n'est possible en pratique que sur des architectures de taille raisonnable. C'est pourquoi nous ne pouvons appliquer cette technique que pour les premières étapes. Pour les autres développements, nous cherchons à comparer une version à la précédente en n'analysant que les parties modifiées.

ii) Substitution d'un composant par un composant qui le raffine

Supposons que l'on souhaite remplacer un composant C1 dans une architecture A1 par un composant C2, de façon à obtenir une nouvelle architecture A2. Si C2 est un raffinement (horizontal ou vertical) de C1, alors hélas, A2 n'est pas nécessairement un raffinement de A1. Les relations de raffinement ne sont pas des relations de congruence ou en d'autres mots elles n'assurent pas la substituabilité. Il en est de même pour l'ensemble des relations de conformité présentées plus haut.

Trouver la plus grande relation plus forte que la conformité, qui soit une congruence pour les opérateurs de composition parallèle et de masquage (qui sont les opérateurs utilisés dans les architectures) est resté un problème ouvert pendant de nombreuses années [8]. Le préordre *should* a été proposé par Brinksma, Rensink et Vogler [1] pour répondre à ce problème. Cependant, aucun algorithme n'a été donné à ce jour pour le vérifier. Nous nous orientons vers le préordre *conflict* proposé par Malik [11], qui bien que différent du préordre *should*, est une pré-congruence plus forte que la relation de conformité, et pour laquelle des idées d'algorithme ont été présentées. Le raffinement identifié par le préordre *conflict* est plus fort que les deux raffinements horizontal et vertical.

iii) Ajout d'un nouveau composant à l'architecture

Pour pouvoir ajouter un nouveau composant C à l'architecture, nous cherchons à vérifier si le composant est *compatible* avec la partie de l'architecture avec laquelle il va interagir, c'est-à-dire avec l'ensemble des composants de l'architecture auxquels il va être connecté (on appelle E cet ensemble). Dans [4], on a proposé une relation de compatibilité permettant de garantir l'absence de blocage dans l'architecture résultante. Néanmoins, cette relation n'est pas assez forte pour garantir la conformité entre deux architectures. Une solution est alors d'exploiter les résultats des relations de substitution (ii), en remplaçant E par le nouveau composant formé de E et C (appelé EC). La vérification du préordre de conflit entre EC et E permet de garantir la conformité de la nouvelle architecture avec l'ancienne.

Exemple 2

Soit l'architecture formée de la composition de Jobber et d'un outil, telle que donnée à la Figure 4

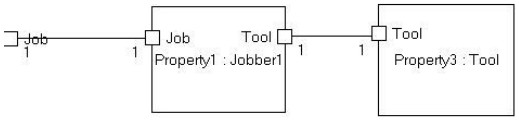

Figure 4 Architecture : diagramme composite UML formé d'un composant Jobber et d'un composant Tool

Le composant Jobber est celui spécifié dans l'exemple 1. On pourrait être tenté de remplacer Jobber par un composant dont la machine d'états serait celle de Jobber2 (Figure 5). En effet, on peut vérifier

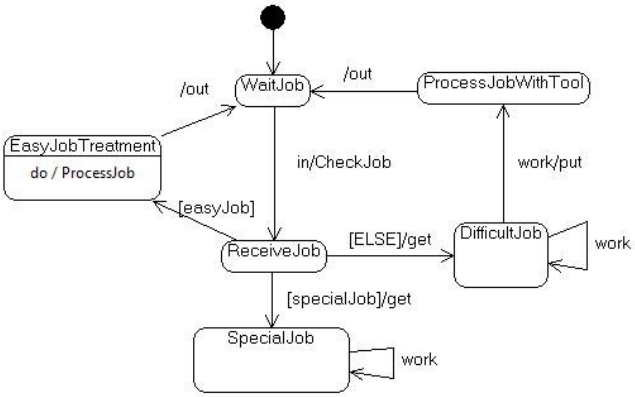

Figure 5 Machine d'état de Jobber2

que Jobber2 est conforme à SpecJobber. Cependant, une fois les opérations work, get et put masquées, l'architecture obtenue n'est pas conforme à l'architecture initiale. Il convient de comparer les comportements de Jobber par une relation plus forte. Ici, on peut vérifier que Jobber2 n'est pas comparable à SpecJobber par la relation should.

Résultats

Une démarche de développement incrémentale nous semble être un bon compromis entre des approches pragmatiques, telles que le développement Agile, et des approches formelles telles que la méthode B. Les difficultés de spécification et de conception d'architectures et de machines d'états UML sont telles qu'il est nécessaire de pouvoir élaborer les modèles progressivement d'une part, et de les évaluer en cours de développement d'autre part.

Nous avons réalisé un outillage des relations de conformité définies entre systèmes de transitions étiquetées et nous avons proposé une traduction des machines d'états et des architectures UML vers les systèmes de transitions étiquetées. Ceci est implanté dans le prototype IDCM (*Incremental Development of Conforming Models*).

Il est ainsi possible d'élaborer une architecture pas-à-pas, par substitution de composants et ajout de nouveaux composants, tout en cherchant à garantir un raffinement horizontal ou vertical en cas d'ajout de composants ou de substitution. Ces techniques de vérification nous permettent de savoir si les propriétés de vivacité du modèle initial sont préservées ou non. En cas de raffinement vertical, elles permettent de plus de savoir si les propriétés de sûreté sont préservées.

Références

1. Brinksma, E., Rensink, A., and Vogler, W. Fair Testing. *CONCUR '95: Concurrency Theory*, Springer-Verlag (1995), 313-327.

2. Brinksma, E. and Scollo, G. *Formal Notions of Implementation and Conformance in LOTOS*. Twente University of Technology, Department of Informatics, Enschede, 1986.

3. Gérard, B. À la chasse aux bugs: la vérification des programmes et circuits. *Cours à Collège de France*, 2008.

4. Lambolais, T., Courbis, A.-L., Luong, H.-V., and Phan, T.-L. Interoperability Analysis of Systems. *Proceedings of the 18th IFAC World Congress*, IFAC-PapersOnLine (2011), 7879-7884.

5. Lambolais, T. and Gout, O. Ingénierie du logiciel: allons-nous vers des systèmes plus fiables? *Revue de l'électricité et de l'électronique*, 11 (2005), 26-36.

6. Lang, F. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005*, Springer (2005).

7. Larman, C. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, Boston, Massachusetts, USA, 2003.

8. Leduc, G. Failure-based Congruences, Unfair Divergences and New Testing Theory. *Protocol Specification, Testing and Verification XIV*, Chapman & Hall (1994), 1-16.

9. Luong, H.-V. Construction Incrémentale de Spécifications de Systèmes Critiques intégrant des Procédures de Vérification. *Thèse, Université Paul Sabatier Toulouse III*, 2010.

10. Milner, R. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

11. Ware, S. and Malik, R. A State-Based Characterisation of the Conflict Preorder. *10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011)*, EPTCS (2011), 34-48.

Mots clés

UML, sûreté, vivacité, architectures logicielles, relations de conformité, développement incrémental.