ADAPTABILITE ET TOLERANCE AUX FAUTES : Intérêt des intergiciels réflexifs face à l'évolutivité des systèmes informatiques

François Taïani[1]*

Directeur(s) de thèse: Jean-Charles Fabre*, Marc Olivier Killijian*

Laboratoire d'Accueil :

* LAAS-CNRS
7, avenue du Colonel Roche
F-31077 Toulouse Cedex 4

Établissement d'Inscription :

Université Paul Sabatier - Toulouse III
118, Route de Narbonne
F-31062 Toulouse Cedex

Résumé

Les systèmes informatiques d'aujourd'hui sont utilisés dans des environnements en constante évolution, et nécessitent par conséquent des implémentations particulièrement " malléables ", notamment capables de reconfiguration " à chaud ". L'utilisation de tels systèmes dans des contextes critiques pose cependant le problème de leur tolérance aux fautes. La réflexivité semble apporter une solution intéressante à ce problème, en permettant l'auto-représentation et l'auto-modification d'un système informatique. Dans cet article, nous nous intéressons à l'utilisation de la réflexivité pour la tolérance aux fautes de systèmes hautement adaptables, et notamment la répartition des mécanismes de réflexivité dans les différentes couches du support d'exécution (noyau, intergiciel, etc...).

Mots clés

Tolérance aux fautes, Systèmes informatiques, Intergiciel, Système d'exploitation, Adaptabilité, Réflexivité.

1 Introduction

Depuis plusieurs années, il est classique de souligner l'utilisation de plus en plus massive de briques logicielles standardisées et réutilisables pour la construction de nouveaux systèmes informatiques. Il n'est que de citer le succès des composants sur étagères, COTS en anglais (« *Commercial Off The Shelf* »), ou celui du source libre pour témoigner de cette tendance.

Il est cependant des marchés informatiques qui du fait de leurs spécificités n'ont été touchés que plus récemment par cette évolution de fond. Le marché des applications embarquées, à fort besoin de sûreté de fonctionnement, est de ceux-là. L'utilisation de composants du marché pour de telles applications pose en effet problème, aussi bien pour le matériel que le logiciel. Les enjeux aussi bien économiques qu'humains exigent la mise en place de mécanismes de tolérance aux fautes, indépendamment des briques réutilisées, pour durcir leur robustesse et exclure tout scénario catastrophe.

La réflexivité logicielle fait partie des approches qui ont été proposées pour résoudre ce défi technologique. Dans cet article, après avoir dans une première partie rappelé les notions de tolérance aux fautes et de réflexivité, nous montrons dans une seconde partie pourquoi la réflexivité doit être étendue pour englober dans un même cadre conceptuel l'ensemble des couches d'un système. Nous détaillons enfin dans une dernière partie la méthodologie que nous avons mise en place et les premiers résultats que nous avons obtenus. Nous concluons sur les perspectives de notre travail.

---

1 francois.taiani@laas.fr
Le travail présenté dans cet article a été financé par le Projet Européen IST DSoS n°1999-11585.

2 Tolérance aux Fautes et Réflexivité

2.1. Tolérance aux Fautes

La tolérance aux fautes est l'un des moyens de la sûreté de fonctionnement dont le principe consiste à réagir « à chaud » aux états erronés d'un système et à empêcher que ces erreurs ne conduisent à un dysfonctionnement visible pour l'utilisateur du système considéré [1]. Ce moyen de la sûreté de fonctionnement est notamment adapté à la réutilisation de composants logiciels dont on ne maîtrise pas l'implémentation, pour en permettre le durcissement par l'utilisation d'encapsulateurs (« *wrappers* »), c'est-à-dire d'une couche de confinement logiciel, qui en associant observation, diagnostic et contrôle, permet de stopper la propagation d'erreurs au plus tôt. Un exemple de cette technique appliquée à des micro-noyaux peut être trouvé par exemple dans [2].

2.2. La Réflexivité

La technique de durcissement présentée précédemment suppose l'implant de sondes en des endroits stratégiques du composant à durcir, pour étendre les possibilités d'observation et d'interaction de celui-ci. Ce type d' « intrusion maîtrisée » permet de rendre concret, de réifier, des structures, événements ou politiques, qui sont le plus souvent conceptuellement présents dans la spécification du composant, mais ne sont pas directement accessibles à travers son interface de programmation. Dans le cas d'un système opératoire par exemple, la prise d'un sémaphore n'est visible que pour la tâche qui obtient le sémaphore, et à l'intérieur du noyau, mais il n'est pas possible d'enregistrer un processus utilisateur qui observerait toutes les prises de sémaphore. Une fois rendues visibles, il devient possible d'introspecter ces entités, d'interagir (intercéder) avec elles pour adapter le comportement du composant à des besoins de tolérance aux fautes.

Une telle approche, qui en associant réification, introspection, intercession permet d'exhiber une partie du comportement d'un composant, est appelée *réflexive*. Plus généralement les approches réflexives permettent de distinguer conceptuellement *un niveau de base* (le composant original) et un *méta-niveau* (les mécanismes ajoutés à ce composant), disposant de capacités d'observation et de contrôle du niveau de base. L'interaction entre ces deux niveaux s'opère au niveau d'une méta-interface. On appelle ainsi cette zone d'interaction, car elle n'est pas liée à la fonctionnalité du composant telle qu'elle est perçue extérieurement, mais à l'observation et au contrôle du composant lui-même, par une sorte de « conscience » (le terme est très exagéré dans les cas qui nous intéressent) que serait son méta-niveau. L'adjonction du composant et de son méta-niveau forme ce que l'on appelle un composant réflexif.

Une telle approche est particulièrement avantageuse pour ajouter à une application des mécanismes orthogonaux à ses caractéristiques fonctionnelles, et ceci de manière indépendante du développement de ces dernières fonctionnalités. (« *Separation of concerns* ») [3] [4] Cette caractéristique rend la réflexivité particulièrement intéressante pour l'introduction de mécanismes de tolérance aux fautes, adaptables, flexibles et reconfigurables.

3 Réflexivité Multi-Niveaux

3.1. Interfaces Standards et Modèle de Programmation

La réutilisation de parties de logicielle pré-développées n'a cessé de gagner en importance au cours des dernières années. On peut notamment penser au succès des composants sur étagère (en Anglais COTS pour « *Commercial Off The Shelf* »), ou à l'engouement pour les logiciels en source libres comme caractéristiques de ce phénomène.

L'utilisation à large échelle des telles briques logicielles pré-existantes détermine pour une grande part les possibilités d'observation du système résultant, et donc la nature du méta-niveau qui pourra être réalisé sur ce système, à la manière dont nous l'avons esquissé dans la partie précédente. Dans les paragraphes qui suivent nous tentons de cerner plus précisément les

relations qu'entretiennent entre elles les différentes « briques » d'un logiciel composite, et les conséquences de leur utilisation sur la structuration de ce logiciel.

L'on notera tout d'abord que l'utilisation de composants développés indépendamment les unes des autres ne serait pas possible sans l'existence d'un certain nombre d'interfaces de programmation normalisées, telles la norme POSIX [5] pour les systèmes d'exploitation, la spécification IA32 pour les processeurs Intel©, ou la norme CORBA [6] pour les Bus à Objet. Ces normes sont soit des standards *de facto*, qui se sont imposées par le succès de leurs applications, soit ont été publiées par un organisme de standardisation tel que l'IEEE, l'ISO ou l'OMG, pour ne citer que quelques-uns des plus importants dans le domaine logiciel.

Chacune de ces interfaces recouvre en fait bien plus qu'une simple énumération syntaxique de procédure, types, ou méthodes. Chaque interface définie un ensemble de concepts qui forment un modèle mental de programmation, une certaine manière de penser tout composant réalisant cette interface. L'ensemble des primitives offertes à l'utilisateur n'est que la concrétisation de ce modèle sémantique. En particulier, la signification de ces primitives, essentielle à leur utilisation, est indissociable du cadre conceptuel que définit le standard. On notera que l'emploi que nous faisons de la notion de primitive est ici à prendre au sens large, c'est-à-dire comme tous les éléments constitutifs du modèle de programmation d'une interface, qui sont explicitement accessibles à l'utilisateur de l'interface en question. Ces différents modèles de programmation n'ont pas tous les mêmes caractéristiques, et l'on peut notamment les classer selon la richesse de leur sémantique, leur maniabilité pour l'esprit humain, ou leur facilité d'implémentation, ces différentes propriétés étant interdépendantes et souvent contradictoires.

3.2. Architecture en Couche

La réalisation d'une brique logicielle qui implémente l'une des interfaces standards mentionnées dans la partie précédente utilise dans la pratique à son tour d'autres briques standardisées, déjà disponibles. Cette chaîne de réutilisations se traduit dans le logiciel final par une structuration en couches, dans laquelle la couche C_n réalise l'interface I_n en utilisant le modèle de programmation I_{n-1} fourni par la couche C_{n-1}.

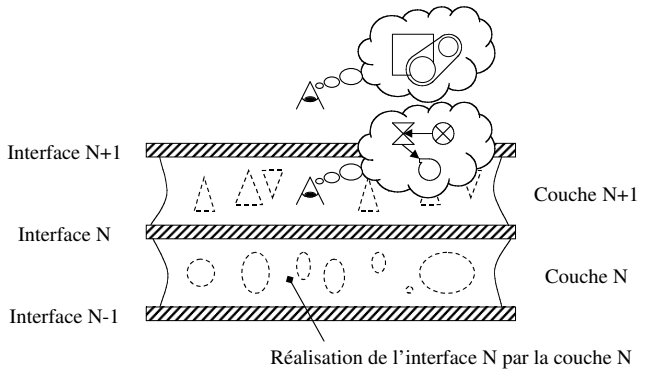

Figure 1 Couches, Interfaces, et Modèle de Programmation

La Figure 1 illustre les relations qu'entretiennent les différentes couches d'un système entre elles. Il est important sur cette figure de bien faire la distinction entre d'une part les modèles de programmation respectifs (les interfaces) réalisés par chacune des couches et la réalisation elle-même de ces modèles. Ainsi sur la figure, les détails de l'implémentation de la couche N, symbolisés par des ellipses, ne sont pas visibles depuis la couche N+1, qui elle réalise l'interface N+1. Seules sont visibles depuis la couche N+1 les primitives du modèle de programmation présenté par l'interface N (ce que nous avons représenté par une bulle en forme de nuage sur la Figure 1), et c'est ce modèle de programmation qui procure les briques à partir desquelles sont réalisées les primitives de l'interface N+1. L'instantiation du modèle de programmation de l'interface N est représentée sur la figure par des triangles. Ces triangles, à leur tour, s'ils réalisent l'interface N+1, ne se confondent pas avec elle, et demeurent cachés à tout utilisateur situé au-dessus des 2 couches.

3.3. La Réflexivité Multi-Niveaux

L'organisation en couche de la plupart des systèmes actuels se traduit donc par une encapsulation emboîtée de modèles de programmation. Rendre un système multi-couches réflexif nécessite donc d'identifier la nature des observations recherchées, et les modes d'interactions nécessaires aux objectifs poursuivis, puis pour chacun des éléments retenus, de choisir la couche dans laquelle cet élément pourra être réalisé au mieux. Or, le degré d'observabilité et de contrôle d'un système dépend fortement du niveau depuis lequel ce système est observé :

Les plus hauts niveaux possèdent généralement des modèles de programmation dotés d'une sémantique riche, associés à des primitives aux fonctions bien différenciées. Ces caractéristiques rendent ces modèles plus facilement programmables pour l'être humain, plus proches des fonctionnalités telles que perçues par un utilisateur, plutôt que de leur réalisation concrète. Cette abstraction a cependant un prix en matière de tolérance aux fautes : Parce qu'il ne perçoit qu'une représentation abstraite du système, le programmeur qui utilise un tel niveau ne dispose que de moyens d'observation et de contrôle d'une granularité relativement grossière. Certains aspects du comportement du système lui échappent totalement (indéterminisme), parce qu'ils dépendent de choix d'implémentation et de micro-états qui ne lui sont pas accessibles. L'état du pipeline d'un microprocesseur moderne constitue un exemple classique d'un tel micro-état, qui n'est pas visible pour le programmeur, mais influence les temps d'exécution du code, et donc l'entrelacement des processus sur un système multitâche.

Les modèles de programmation des niveaux les plus bas permettent au contraire d'observer le système de beaucoup plus près, avec une granularité plus fine. Mais si l'information accessible sur le système est alors plus importante, elle a perdu en signification, elle est moins structurée, plus proche des contraintes matérielles que des fonctionnalités recherchées. Il est par exemple possible au niveau assembleur de connaître la valeur (un entier) d'un peu près n'importe quelle case mémoire, mais sans plus d'informations il est impossible de connaître la signification de cet entier pour l'application (Un caractère ? Une température ? Une vitesse ? Un code d'opération ?).

Parce que ces deux types d'informations (forte granularité et riche sémantique / granularité fine, sémantique pauvre) sont complémentaires, la mise en place de mécanismes de tolérance aux fautes couvrant l'ensemble des couches ne peut se faire à un seul niveau. Elle nécessite au contraire la mise en place d'une méta-interface au sens où nous l'avons définie précédemment qui insère dans chacune des couches d'implémentation les « pseudo-capteurs » et « pseudo-actionneur » nécessaires aux endroits les plus favorables et les plus pertinents.

De nombreux travaux ont déjà été menés pour rendre réflexifs chacun des niveaux d'encapsulation qui constituent classiquement un système informatique comme les systèmes opératoires [7] , ou les intergiciels [8] [9]. Cependant ces approches mono-niveau sont limitées dans leurs capacités de tolérances aux fautes pour les raisons que nous venons de présenter. Ainsi il a été montré dans [9] que les intergiciels multithreadés ne pouvaient êtres appréhendés du point de vue de la tolérance aux fautes en employant uniquement des mécanismes de réflexivité au niveau langage. Des travaux ont été menés sur le problème de l'observation multi niveaux de systèmes concurrents distribués [10], d'autres sur la réalisation d'un canevas réflexif ouvert dès le niveau assembleur [11], mais à notre connaissance la conception et de la réalisation d'une méta-interface réflexive complète qui soit prévue dès le départ pour englober plusieurs couches hétérogènes n'ont pas encore été abordées. Cette notion devrait permettre de lever un verrou technologique en matière de réalisation de systèmes multicouches à fortes exigences en Sûreté de Fonctionnement.

4 Méthodologie & Premiers Résultats Expérimentaux

4.1. Méthodologie

Pour répondre aux problèmes présentés dans la section précédente nous avons mis en place la méthodologie suivante :

1. *Modélisation du passage d'un modèle de programmation à un autre.*

   Ce passage s'opère à l'intérieur d'une couche. Cette première étape a pour objectif de développer un cadre de modélisation (méta-modèle) des différents liens qui peuvent exister une interface et les couches qui la réalisent. Pour nous faire une idée de ces méta-modèles, nous avons, dans une première phase, utilisé la rétro-conception de composants en source libre, ainsi que la factorisation de modèles déjà proposés pour développer une taxonomie d'un ensemble de « fonctions de projection » [10] qui permettent de modéliser le passage d'une abstraction à l'autre.

2. *Spécification d'un ensemble de méta-interfaces basées sur la modélisation précédente.*

   Dans une seconde phase, nous utilisons les enseignements de la phase précédente pour concevoir un ensemble de méta-interfaces pour les différentes couches d'un système OS / Intergiciel. Cette spécification nous mènera à l'identification d'une méta-interface générique fortement couplée au modèle des relations inter-interfaces de la partie 1. Les questions les plus importantes à résoudre dans cette partie sont le choix des points d'insertions des pseudo-capteurs et pseudo-actionneurs, ainsi que les mécanismes d'agrégation et de transformation des données et stimuli échangés entre le niveau de base et le méta-niveau pour pouvoir présenter des primitives d'abstraction homogène au méta-niveau.

3. *Réalisation d'un Prototype.*

   En nous appuyant sur les spécifications obtenues au point n°2, nous prévoyons de réaliser un premier prototype, en instrumentant une plate-forme en source libre déjà existante (Linux + Un bus à Objet de type CORBA). Nous envisageons ensuite l'adoption d'une approche encore plus radicale implémentant une plate-forme exécutive minimale, notamment adaptée aux applications embarquées, basée dès sa conception sur les principes de *Réflexivité Multi-Niveaux* que nous aurons validés.

4.2. Premiers Résultats

L'une des fonctionnalités d'une plate-forme exécutive multi-niveaux (intergiciel + exécutif de base), qui s'étend sur toutes les couches du logiciel exécutif, est la gestion de processus / objets à activités multiples. (C'est-à-dire possédant un parallélisme interne dans un espace-mémoire partagé, notion de « *multi-threading* ».) La représentation des threads, leur contrôle, la capture de leur état nécessite une maîtrise profonde de leur mise en œuvre dans chaque couche du système, une comparaison fine de leur structure et de leur comportement à différents niveaux d'abstraction. Nous avons donc décidé dans un premier temps de focaliser notre attention sur le problème complexe de la réplication d'application multi-threadées [12]. Ce type de réplication pose en effet problème aussi bien du point de vue de la capture de l'état, du recouvrement par journalisation, que de la synchronisation et de la reprise. [13] [14]

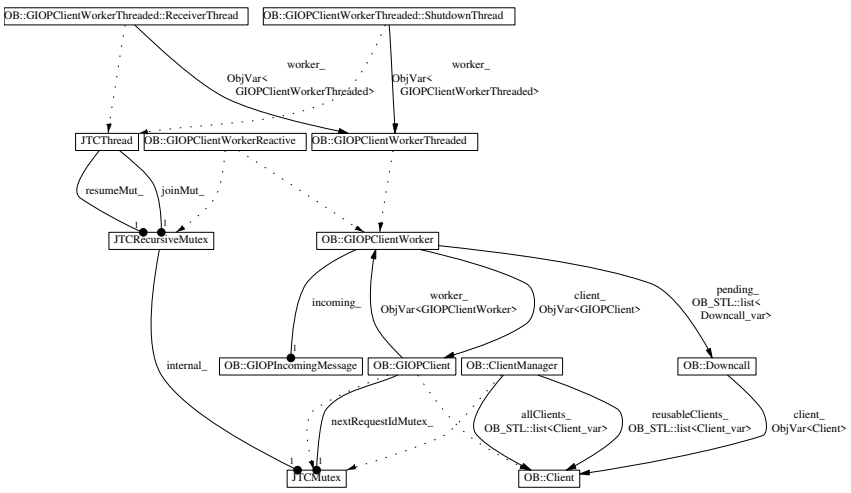

Figure 2 Exemple de Retro-Conception d'un ORB Commercial (extrait)

Dans cette optique, nous nous sommes intéressés aux modes de concurrence CORBA, et à leur implémentation dans l'ORB commercial *ORBacus*®. La Figure 2 par exemple montre un diagramme de classe issue du code source de cet ORB, focalisé sur l'implémentation des différents modes de concurrence côté-client.

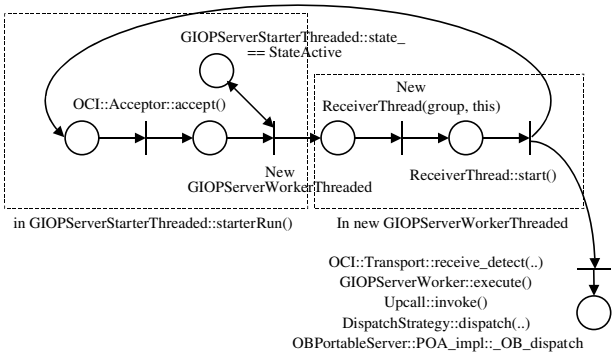

Figure 3 Extraction d'un modèle RdP de la concurrence interne d'ORBacus®

En nous basant sur les résultats fournis par les différents outils de retro-conception (analyse statique de code, traçage, débogueur), nous pouvons alors construire des modèles comportementaux et structuraux de plus haut niveaux, comme celui de la Figure 3. Cette figure modélise le mécanisme de réception et de distribution des requêtes entrantes d'un serveur multi-threadé dans *ORBacus*®.

Notre objectif est d'utiliser de telles modélisations à différents niveaux et dans différentes implémentations pour en factoriser la trame générique qui nous permettra, en regard de nos buts de tolérance aux fautes, de spécifier une méta-interface multi-niveaux telle que nous l'avons présentée dans la partie la partie 3.

5 Conclusion et Perspectives

Nous avons montré dans cet article la nécessité d'une approche globale pour l'introduction de mécanismes de tolérance aux fautes dans les systèmes à base de composants. L'approche réflexive, outre son élégance, est très bien adaptée à cet objectif, mais nécessite d'être étendue au-delà des travaux et de l'expérience capitalisés jusqu'à aujourd'hui pour prendre en compte des systèmes multi-couches, structurés autour de plusieurs niveaux d'abstraction en grande partie étanches entre eux.

Après avoir analysé les raisons de cette nécessité, nous avons proposé une méthodologie pour atteindre cet objectif, et présenté les premiers résultats issus de ce plan d'action. Il nous reste maintenant à poursuivre cet effort pour identifier une méta-interface générique multi-couche, dont les avantages en termes de flexibilité, d'adaptabilité, et de maniabilité cognitive seront validés sur plusieurs prototypes.

Références

[1] Laprie, J.-C., ed. *Le guide de la sûreté de fonctionnement*. 1996, Cepadues. 370 pages.

[2] Rodriguez, M., J.-C. Fabre, and J. Arlat. *Formal Specification for Building Robust Real-time Microkernels*. in *21st IEEE Real-Time Systems Symposium*. 2000. Orlando, Florida, USA.

[3] Maes, P. *Concepts and Experiments in Computational Reflection*. in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 1987. Orlando, Florida.

[4] Hürsch, W. and C. Videira Lopes, *Separation of Concerns*, . 1995, Northeastern University: Boston, USA.

[5] ISO-IEC, *[IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. 1996. 784.

[6] OMG, *Common Object Request Broker Architecture (CORBA/IIOP) (2.6)*. 2001. [http://www.omg.org/cgi-bin/doc?formal/01-12-35].

[7] Yokote, Y., F. Teraoka, and M. Tokoro. *A Reflective Architecture for an Object-Oriented Distributed Operating System.* in *Third European Conference on Object-Oriented Programming (ECOOP'89)*. 1989. Nottingham, UK: Cambridge University Press.

[8] Kon, F., et al. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. in IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). 2000. New York.

[9] Killijian, M.O., *Tolérance aux fautes sur CORBA par protocole à métaobjets et langages réflexifs*, Thèse de Doctorat, Institut National Polytechnique, Toulouse, 2000, 162 pages.

[10] Ottogalli, F.-G., *Observations et analyses quantitatives multi-niveaux d'applications à objets réparties*, Thèse de doctorat en informatique, Université Joseph Fourier - Grenoble I, Grenoble, France, 2001.

[11] Fassino, J.-P. and J.-B. Stefani. *Think : un noyau d'infrastructure répartie adaptable*. in *Deuxième Conférence Française sur les Systèmes d'Exploitation (CFSE-2)*. 2001. Paris.

[12] Elnozahy, E.N., *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.*, PhD thesis, Rice University, Department of Computer Department of Computer Science, Houston, Texas, 1993.

[13] Narasimhan, P., L.E. Moser, and P.M. Melliar-Smith. *Enforcing Determinism for the consistent replication of Multithreaded CORBA Applications*. in *18th Symposium on Reliable Distributed Systems*. 1999. Lausanne, Switzerland: IEEE.

[14] Jiménez-Peris, R., M. Patiño-Martínez, and S. Arévalo. *Deterministic Scheduling for Transactional Multithreaded Replicas*. in *19th IEEE Symposium on Reliable Distributed Systems (SRDS)*. 2000. Nürmberg, Germany.