

La réflexivité dans les architectures multi-niveaux : application aux systèmes tolérant les fautes

François Taïani

▶ **To cite this version:**

François Taïani. La réflexivité dans les architectures multi-niveaux : application aux systèmes tolérant les fautes. Réseaux et télécommunications [cs.NI]. Université Paul Sabatier - Toulouse III, 2004. Français. NNT : . tel-00010940

HAL Id: tel-00010940
https://theses.hal.science/tel-00010940

Submitted on 9 Nov 2005

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.

Numéro d'ordre : XXXX - Année 2004

Thèse

préparée au

Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

en vue de l'obtention du

Doctorat de l'Université Paul Sabatier de Toulouse
Spécialité : Systèmes Informatiques

par

François TAÏANI
Ingénieur de l'École Centrale Paris
Diplom Informatiker de l'Université de Stuttgart

La Réflexivité dans les architectures multi–niveaux : application aux systèmes tolérant les fautes.

Soutenue le 12 janvier 2004 devant le jury :

Président	**Louis**	**FERAUD**
Rapporteurs	**Charles**	**CONSEL**
	Michel	**RAYNAL**
Examinateurs	**Gordon**	**BLAIR**
	Jean-Charles	**FABRE**
	Jean-Bernard	**STEFANI**
Invité	**Alain**	**ROSSIGNOL**

Directeurs de thèse :

Jean-Charles FABRE
Marc-Olivier KILLIJIAN

Cette thèse a été préparée au LAAS-CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4

Rapport LAAS Numéro XXXXX / Version générée le 3 juin 2004

Avant-propos

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je remercie Messieurs Jean-Claude Laprie et Malik Ghallab, qui ont assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire.

Je remercie également Messieurs David Powell et Jean Arlat, Directeurs de Recherche CNRS, responsables successifs du groupe de recherche Tolérance aux Fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser ces travaux dans ce groupe.

J'exprime ma très sincère reconnaissance à Messieurs Jean-Charles Fabre et Marc-Olivier Killijian, respectivement Directeur et Chargé de Recherche CNRS, pour m'avoir encadré, soutenu, et encouragé tout au long de cette thèse. L'expérience, l'enthousiasme, et les conseils de Jean-Charles ; la patience de Marc-Olivier, sa disponibilité, et son ouverture, ont été essentiels au long travail de maturation dont ce mémoire est le résultat. Leurs qualités humaines, leur amitié, et leur aide — dans les situations les plus variées — ont fait de ces trois ans une aventure exceptionnelle.

Je remercie Monsieur Louis Féraud, Professeur à l'Université Paul Sabatier, pour l'honneur qu'il me fait en présidant mon jury de thèse, ainsi que :

- Monsieur Charles Consel, Professeur à l'École Nationale Supérieure d'Électronique, Informatique et de Radiocommunications de Bordeaux ;
- Monsieur Michel Raynal, Professeur à l'Université de Rennes 1 ;
- Monsieur Gordon Blair, Professeur à l'Université de Lancaster (UK) ;
- Monsieur Jean-Bernard Stéfani, Directeur de Recherche à l'INRIA, Chef Ingénieur du Corps des Télécommunications ;

pour l'honneur qu'ils me font en participant à mon jury. Je remercie particulièrement Messieurs Charles Consel et Michel Raynal qui ont accepté la charge d'être rapporteurs.

Au delà de leur encadrement exemplaire, Jean-Charles Fabre et Marc-Olivier Killijian ont su créer et animer autour d'eux une équipe formidable avec laquelle j'ai pris un grand plaisir à travailler. Je tiens à les remercier chaleureusement : Juan-Carlos Ruiz-Garcia (« On

est bien peu de chose ! »), Eric Marsden (notre kangourou quadri–national), Ludovic Courtès (« Powergnu ! »).

Je tiens également à remercier les membres du groupe TSF, permanents, doctorants et stagiaires, ainsi que Joëlle Penavayre pour leur constante disponibilité et leur gentillesse. Je réserve ici une mention particulière à Arnaud Albinet, pour nos nombreuses soirées jeux, et pour m'avoir prêté son vélo, à Guillaume Lussier, pour ses conseils en matière de thé et de culture nippone, à Eric Marsden, pour ses corrections en anglais et ses nombreux conseils sur Linux et sur CORBA, à Tahar Jarboui et Mourad Rabah, pour leurs soirées merguez, à et Olfa Kaddour pour sa bonne humeur et son attention.

Nombreux sont ceux qui, à des titres divers, auront participé à l'aboutissement de ces travaux, qu'ils en soient ici remerciés : Ludovic Courtès, Eric Marsden et Anne Dibusi pour avoir accepté de relire ce mémoire ainsi que Jean Arlat, Yves Crouzet, Yves Deswartes, Arnaud Albinet, Ali Kalakech, Benjamin Lussier, Cristina Simache, Nicolas Salatge et Anis Youssef pour m'avoir assisté lors des répétitions de ma présentation.

Mes remerciements s'adressent également à tous les membres des services *Informatique* et *Instrumentation*, *Documentation–Édition*, *Magasin*, *Entretien*, *Direction–Gestion*, *Réception–Standard* qui m'ont toujours permis de travailler dans d'excellentes conditions.

Au delà du simple contexte professionnel, j'ai trouvé au LAAS un environnement de travail exceptionnellement chaleureux. J'aimerais remercier tout ceux qui ont pu m'aider dans les moments les plus variés. Je pense particulièrement à Jessica Kuchenbecker, à Marido Cabanne, et à Mario Paludetto, qui, en plus de bien d'autres choses, m'ont apporté leur soutien et leur amitié lors de ce Noël 2000 bien difficile.

Enfin, bien sûr, je remercie chaleureusement tous ceux qui, en dehors du laboratoire, m'ont accompagné et soutenu : ma famille, présente depuis le début, mes nombreux amis de Toulouse et d'ailleurs, que je ne peux pas citer tous ici, et enfin Anne, pour sa patience, son soutien, et son aide.

Table des matières

Avant-propos **iii**

Table des matières **v**

Table des figures **vii**

Introduction **xi**

1 Sûreté de fonctionnement et logiciels complexes **1**
1.1 La Sûreté de fonctionnement et la tolérance aux fautes 2
1.1.1 Notions de base : fautes, erreurs et défaillances 2
1.1.2 Redondance et diversité 4
1.1.3 Modèles de fautes 5
1.1.4 Les mécanismes de tolérance aux fautes 6
1.1.5 Systèmes distribués et tolérance aux fautes 7
1.2 Systèmes logiciels complexes 8
1.2.1 Buts et difficultés du génie logiciel 8
1.2.2 Système d'exploitation et intergiciel 10
1.2.3 L'orienté objet 12
1.2.4 Les intergiciels et les objets : les bus à objets 14
1.2.5 Le problème posé par les systèmes multi-couches 16
1.3 Conclusion du chapitre 18

2 Les architectures réflexives **19**
2.1 La réflexivité au quotidien 20
2.2 La réflexivité en informatique 22
2.2.1 La réflexivité des langages interprétés 23
2.2.2 La réflexivité des langages compilés et des substrats d'exécution 26
2.2.3 Exemple de plates-formes réflexives 28
2.3 Conclusion du chapitre 33

3 Les plates-formes pour la tolérance aux fautes **35**
3.1 Pourquoi des plates-formes tolérantes aux fautes ? 35
3.1.1 Aspects de la tolérance aux fautes 36
3.1.2 Propriétés des plates-formes 38

3.1.3 Types d'architectures pour la tolérance aux fautes . . . 39
3.2 Exemple de plates-formes pour la tolérance aux fautes . . . 41
3.2.1 Les pionniers : les plates-formes explicites . . . 41
3.2.2 Plates-formes par intégration et interception . . . 43
3.2.3 Les solutions réflexives . . . 47
3.3 Les limites des approches proposées . . . 50

4 La réflexivité multi–niveaux : notions et principes 53
4.1 Un méta-modèle des architectures en couches . . . 54
4.1.1 Structure et cognition . . . 54
4.1.2 Niveaux, couches, interfaces et modèles . . . 55
4.1.3 Liens inter-niveaux . . . 56
4.1.4 Utilisation des liens inter-niveaux . . . 58
4.2 Empreintes et validité des algorithmes . . . 59
4.2.1 Notion d'empreinte réflexive . . . 59
4.2.2 Modèles algorithmiques et choix d'implémentation . . . 62
4.2.3 Granularité du contrôle et validité des algorithmes . . . 64
4.2.4 Défaut de perception sur un exemple pratique . . . 67
4.2.5 Conclusion sur l'implémentation réflexive des algorithmes . . . 71
4.3 Une démarche de développement multi-niveaux . . . 72
4.4 Conclusion . . . 73

5 Application à la réplication 75
5.1 Empreinte réflexive de la réplication . . . 75
5.1.1 Présentation des mécanismes étudiés . . . 76
5.1.2 Empreinte réflexive . . . 77
5.2 Réplication multi-niveaux sur CORBA/POSIX . . . 78
5.2.1 Le modèle de programmation CORBA, liens avec l'OS . . . 80
5.2.2 Multitraitement et non-déterminisme . . . 82
5.2.3 Déterminisme des bus à objets à brins d'exécution multiples . . . 84
5.2.4 Capture de l'état d'un bus à objets à brins d'exécution multiples . . . 89
5.2.5 Le méta-modèle résultant, conclusion . . . 93
5.3 Extraction de méta-modèles : problématique et solutions . . . 95
5.3.1 La problématique . . . 95
5.3.2 Extraction automatique d'informations comportementales . . . 97
5.4 Cas d'étude & prototypage sur un ORB commercial . . . 102
5.4.1 Construction d'un méta-modèle de l'ORB . . . 102
5.4.2 Principe de réalisation du prototype . . . 106
5.4.3 La bibliothèque `libuspi.so` : réflexivité des primitives OS . . . 107
5.4.4 La bibliothèque `meta-corba.so` : la méta-interface multi-niveaux . . . 111
5.4.5 Résultats et conclusion . . . 113
5.5 Conclusion . . . 115

Conclusion et perspectives 117
Résumé de nos travaux de thèse . . . 118
Leçons et perspectives . . . 120

A Le théorème d'incomplétude de Gödel 123

B CosmOpen : rétro–conception des logiciels multi-couches **127**
B.1 Architecture de l'outil 128
B.2 OPSBrowser, l'abstracteur de CosmOpen 129
B.2.1 Aperçu 129
B.2.2 Graphes structurels et graphes comportementaux 129
B.2.3 Les opérateurs proposés par OPSBrowser 130
B.3 OPSBrowser en action 132
B.3.1 Le programme étudié 132
B.3.2 Causalités cachées et « sauts » temporels 133
B.3.3 S 'abstraire des détails tout en gardant leur trace 135
B.3.4 En résumé 137
B.4 Conclusion 137

Bibliographie **141**

Index **150**

Table des figures

1.1 Faute, erreurs et défaillance . . . 3
1.2 Récursivité des fautes, erreurs et défaillances . . . 4
1.3 Un déménageur sans polymorphisme . . . 13
1.4 Factorisation de la « déménageabilité » par polymorphisme . . . 14

2.1 La réflexivité dans la littérature contemporaine [Goscinny & Uderzo 1965] . . . 21
2.2 Architecture d'un système réflexif . . . 25

3.1 Capture d'état : $A + B$ et $A + C$ doivent être des comportements valides . . . 37
3.2 Cohérence d'un état distribué . . . 38

4.1 Couche, interface et modèle de programmation . . . 55
4.2 Récursivité des modèles de programmation . . . 56
4.3 Exemple de liens entre niveaux d'abstraction . . . 58
4.4 Propagation d'une exigence utilisateur sur l'architecture . . . 60
4.5 Évaluation de l'impact d'une faute sur l'utilisateur . . . 60
4.6 Diversité des correspondances entre algorithme et implémentation . . . 63
4.7 « Robustesse » d'un algorithme à l'approximation . . . 65
4.8 Entrée d'un protocole de capture d'état distribué . . . 66
4.9 Sortie d'un protocole de capture d'état distribué . . . 66
4.10 Approximation entre deux scénarios de communication . . . 67
4.11 Pourquoi l'approximation fonctionne : les chemins en zigzag . . . 67
4.12 Deux brins d'exécution interagissent avec un objet . . . 68
4.13 Problème de la cohérence entre objets à brins d'exécution multiples . . . 68
4.14 Une approximation des interactions brin / objet [Kasbekar et al. 1999] . . . 69
4.15 Interprétation plus précise des interactions brin / objet . . . 70
4.16 Démarche de développement pour la réflexivité multi-niveaux . . . 72

5.1 Architecture concrète considérée . . . 78
5.2 Vue d'ensemble (simplifiée) d'un bus à objets CORBA . . . 81
5.3 Modèle générique d'un ORB à réserve de brins (*thread pool*) . . . 81
5.4 Exemple de deux brins d'exécution concurrents . . . 83
5.5 Pertinence d'un verrou pour le déterminisme . . . 85
5.6 Faille de l'approche mono-niveau applicative . . . 87
5.7 Un réification du cycle de vie des requêtes . . . 88

5.8 Classification des requêtes selon leur degré d'avancement 90
5.9 Méta-interface retenue pour la réplication d'un ORB 93
5.10 L'état à restaurer 94
5.11 Restauration de l'état du système par notre approche 94
5.12 Racines et ramifications dans l'analyse des liens inter-niveaux 96
5.13 Un programme élémentaire à brins d'exécution multiples 98
5.14 Architecture du multitraitement sous LINUX 2.4 99
5.15 Une trace d'appel sur l'appel système `clone` et sa représentation XML 100
5.16 Mécanisme de création de brins avec le noyau LINUX 2.4 101
5.17 Le comportement du programme de la fig. 5.13 page 98 après abstraction . . 101
5.18 Ramifications structurelles de la bibliothèque `libpthread.so` dans ORBACUS 103
5.19 Initialisation, réception et traitement d'une requête dans ORBACUS 105
5.20 La classe `MetaMutex` 107
5.21 Contextes sémantiques d'un appel de bas niveau (ici `pthread_mutex_init()`) 109
5.22 La classe `MetaThreadInfo` 110
5.23 Diagramme de classes des méta-objets du niveau POSIX 110
5.24 Principe de fonctionnement d'une usine à méta-verrous 111
5.25 Comment les verrous devant être réifiés sont sélectionnés 112
5.26 Diagramme d'héritage de `RequestContentionPoint` 112
5.27 Traçage de l'activité d'ORBACUS par notre prototype 114
5.28 Un nouveau modèle à composants réflexifs 122

A.1 Projection d'un système axiomatique dans l'espace de son discours 125

B.1 Architecture générale de l'outil COSMOPEN 128
B.2 Deux représentations d'un même graphe comportemental 130
B.3 Graphe de la figure B.2 après l'opération "`remove C::*`" 132
B.4 Création de deux brins par la bibliothèque de multitraitement de LINUX 2.4 . 133
B.5 Graphe B.4 après une opération de saut « `leapOver` » 135
B.6 Sélection des actions internes à la bibliothèque de multitraitement 136
B.7 Le graphe résultant du retrait (`exclude`) du graphe B.6 137
B.8 Après abstraction des méthodes POSIX `pthread_`... du graphe B.7 137
B.9 Le filtre final d'abstraction permettant d'obtenir le graphe B.8 138

Introduction

Les systèmes informatiques sont aujourd'hui en passe de s'intégrer à notre environnement quotidien à un degré jamais égalé jusqu'alors, à la fois riche de promesses et de dangers. Alors que ces systèmes, auxquels des richesses, des vies, sont confiées, gagnent chaque jour en complexité, de nouvelles approches sont requises pour maîtriser leur développement et garantir leur sûreté de fonctionnement.

Ce problème se pose pour les systèmes à très court cycle de vie, pour lesquels l'objectif de *time to market* est prioritaire, qui sont développés aussi rapidement que possible à partir de composants obtenus sur étagère. Elle se pose aussi pour les grands systèmes, souvent critiques, qui suivent des processus de développement plus longs et plus rigoureux, mais qui doivent, pour des raisons économiques, eux aussi réutiliser des composants du marché, tout en permettant une évolution au cours du temps, tant au niveau de leurs fonctionnalités, que pour prendre en compte des mutations environnementales. Pour tous ces systèmes, très hétérogènes, la maîtrise de leur sûreté de fonctionnement, récurrente en informatique, prend une dimension dramatique : Peut-on agréger tel ou tel composant tout en garantissant des propriétés de sûreté ? Maîtrise-t-on la complexité et disposons nous de suffisamment de connaissances sur les composants pour faire évoluer les mécanismes de sûreté en fonction du contexte opérationnel ? Même lorsqu'une solution à un problème est connue, peut-on assurer le respect de ses hypothèses dans tel ou tel contexte ? Cette liste de questions n'est pas exhaustive, mais éclaire, au moins partiellement, le défi que représente la sûreté des systèmes informatiques d'aujourd'hui !

Le développement de systèmes sûrs de fonctionnement (tolérance aux fautes, modélisation, test) a certes une longue histoire scientifique et industrielle, mais ce qui est applicable sur de petits systèmes, ne l'est plus dès que l'on cherche à passer à l'échelle des systèmes réels, utilisés en pratique. La complexité de ces systèmes rend la maîtrise de l'agrégation, de la réutilisation et de l'évolution d'aspects transversaux tels que de sûreté de fonctionnement bien plus délicate. L'un des objectifs scientifiques actuels est de tirer parti des résultats déjà obtenus pour construire des systèmes complexes, qui superposent de nombreuses couches de logiciel, tout en permettant la réutilisation de mécanismes transversaux sous forme de composants, d'une manière qui garantisse leur transparence vis-à-vis des applications, et

favorise une évolution indépendante du système vis-à-vis de ses mécanismes de sûreté de fonctionnement, et réciproquement.

La réponse actuelle pour maîtriser la complexité se résume trop souvent au seul slogan de l'« approche à composants » : distinguons interface et implémentation ! Cette approche a beaucoup d'avantages que nous ne renions pas, l'histoire récente de l'informatique l'a montré, notamment avec le succès de l'orienté objet. Ce que nous remettons en cause dans cette thèse, c'est son adéquation à construire des systèmes sûrs de fonctionnement complexes en utilisant uniquement les concepts qui sous-tendent cette approche, c'est-à-dire, l'encapsulation et le masquage de l'implémentation qui est un frein à certaines formes de réutilisation. Rendons ses lettres de noblesse à l'implémentation, ou plus exactement évitons l'ignorance des détails d'implémentation, argument derrière lequel aiment à se retrancher certains concepteurs de systèmes !

Pour relever ce défi, nous pensons qu'une nouvelle approche à composants est nécessaire. Nous ne prétendons pas en avoir défini tous les « tenants et les aboutissants », mais notre objectif est d'en donner une définition, de justifier la technologie sous-jacente et d'en fournir une illustration pratique. À ce titre, les travaux présentés dans cette thèse s'intéressent à la réalisation de mécanismes de tolérance aux fautes pour des architectures complexes, organisées en couches logicielles multiples, et intégrant un grand nombre de composants hétérogènes. La tolérance aux fautes est effet une propriété globale des systèmes considérés, orthogonale à tous les niveaux d'abstraction rencontrés, et il est donc nécessaire de trouver un paradigme conceptuel qui soit cohérent avec cet impératif.

Des travaux récents ont montré que des techniques existent pour intégrer de manière souple et flexible la tolérance aux fautes à des systèmes informatiques. Les architectures réflexives et la programmation par aspects sont de celles-là. Elles ont fait leurs preuves sur des architectures simples pour ajouter des mécanismes transversaux (dont la tolérance aux fautes) à des prototypes souvent réalisés *ex nihilo* pour expérimentation. Cependant, seuls de très rares travaux se sont, à notre connaissance, penchés sur l'utilisation de ces techniques dans des architectures complexes, composées de nombreuses couches hétérogènes. Ces rares travaux n'ont en outre jamais abordé le point de vue de la tolérance aux fautes dans ce contexte. C'est la problématique générale de cette thèse. Plus particulièrement, nous étudions comment le paradigme réflexif, proposé pour intégrer de manière transparente et méthodique des aspects transversaux dans des programmes informatiques, peut être adapté pour prendre en compte l'ensemble des niveaux d'abstraction présents dans un logiciel complexe (systèmes opératoires, intergiciels, machines virtuelles *etc.*).

La thèse s'ouvre sur un chapitre introductif (page 1) qui pose le problème de la sûreté de fonctionnement, introduit les principes de la tolérance aux fautes, et définit ce que nous entendons par complexité des logiciels. Le second chapitre (page 19) introduit le paradigme réflexif, et motive son intérêt pour notre problématique. Nous poursuivons dans le chapitre 3 (page 35) par un état de l'art des plates-formes logicielles de tolérance aux fautes, en présentant quels travaux ont utilisé la réflexivité et en discutant leurs limites. Nous motivons alors pourquoi et comment la réflexivité doit être étendue pour être utilisée sur des logiciels complexes multi-niveaux. Cette analyse nous permet, dans le chapitre 4 (page 53),

de développer les principes du nouveau paradigme architectural que nous proposons, *la réflexivité multi-niveaux*, et d'aboutir à une démarche de développement de système basée sur ce paradigme. Nous validons cette démarche dans un dernier chapitre (page 75) sur une plate-forme CORBA/POSIX largement répandue dans l'industrie, et présentons comment nous avons implémenté un premier prototype de notre architecture sur un système composé du système d'exploitation GNU/LINUX et du bus à objet commercial ORBACUS.

Enfin, notre conclusion (page 117) motive la poursuite de l'étude des concepts élaborés dans cette thèse pour le développement d'une nouvelle approche à composants réflexifs, permettant de rendre les systèmes logicielles complexes, en accord avec la réalité industrielle, à la fois tolérant les fautes et adaptables.

Chapitre 1

Sûreté de fonctionnement et logiciels complexes

POUR répondre aux exigences du marché en termes de fonctionnalités, de coût ou de flexibilité, les systèmes logiciels sont depuis longtemps poussés vers une modularité toujours accrue, qui passe notamment par la réutilisation de nombreux composants développés indépendamment (systèmes opératoires, bibliothèques spécialisées, intergiciels génériques ou métiers) [Stolper 1999] [MITRE 2003]. Ce haut degré de réutilisation se traduit par des architectures toujours plus complexes, intégrant des éléments hétérogènes dans des systèmes multiformes.

Le marché des applications embarquées, cependant, du fait de ses spécificités, n'a été touché que récemment par cette évolution. Par « embarqué » nous entendons ici dans un sens très large tout système en prise directe avec le monde des hommes, auquel des vies, des richesses, sont confiées. Ces systèmes agissent directement sur l'environnement humain dans des contextes critiques sans qu'un être humain ne contrôle chacune de leurs actions. Cette notion recouvre par exemple aussi bien les systèmes de contrôles aériens ou ferroviaires, que les réseaux de micro–contrôleurs enfouis dans nos voitures, ou les grands systèmes de télécommunication (téléphonie fixe et mobile, constellation de satellites...), pour ne citer que quelques exemples.

Tous ces systèmes ont en commun de devoir assurer un haut degré de sûreté de fonctionnement. Leurs défaillances, lorsqu'elles se produisent, ont souvent des conséquences humaines, environnementales ou économiques catastrophiques. L'utilisation d'architectures logicielles complexes intégrant de nombreux composants hétérogènes pose dans ce contexte un certain nombre de problèmes que nous allons tenter de mettre en lumière dans ce premier chapitre. Pour ce faire, nous introduisons tout d'abord quelques notions de sûreté

de fonctionnement, et notamment de tolérance aux fautes, puisque c'est cet aspect que nous abordons plus spécifiquement dans ce mémoire.

Nous revenons ensuite sur la notion de complexité dans les systèmes informatiques, notamment telle qu'elle a pu être abordée par le génie logiciel. Nous tâcherons à cette occasion d'identifier ce qui derrière la notion de « complexité » pose problème pour la sûreté de fonctionnement.

L'étude de ces deux aspects, sûreté de fonctionnement et complexité, des systèmes logiciels embarqués, nous conduira naturellement à préciser en conclusion les termes de notre problématique en termes d'architecture et de découplage.

1.1 La Sûreté de fonctionnement et la tolérance aux fautes

1.1.1 Notions de base : fautes, erreurs et défaillances

L'histoire de la sûreté des systèmes fabriqués par l'homme reste à faire, mais elle n'est bien évidemment pas née avec l'informatique. En ne remontant qu'à la révolution industrielle, on trouvait déjà dans les premiers systèmes de signalisation ferroviaire ou les constructions maritimes des considérations élaborées de sûreté de fonctionnement.

Ce mémoire s'intéresse plus particulièrement aux systèmes informatiques. Même s'ils partagent des points communs avec les réalisations industrielles du début du siècle, ceux-ci nécessitent de développer des approches qui leur soient spécifiques. Une tâche essentielle a en particulier consisté à proposer un ensemble de notions claires pour appréhender la sûreté de fonctionnement indépendamment de la nature du système à laquelle elle s'applique. Dans ce mémoire, nous utilisons la taxonomie développée entre autres par Jean-Claude Laprie, Brian Randell et Algirdas Avizienis, qui fait référence dans notre domaine. La suite de cette section a pour objectif de définir dans leurs grandes lignes les notions de *sûreté de fonctionnement*, de *faute*, d'*erreur* et de *défaillance*, qui nous permettront de poser la problématique de cette thèse. Nous dirons ensuite quelques mots sur les moyens proposés pour réaliser des systèmes sûrs de fonctionnement, en détaillant plus particulièrement l'un d'entre eux dans la section qui suit : la tolérance aux fautes.

La *sûreté de fonctionnement* d'un système est définie comme :

> *[L]a propriété qui permet à ses utilisateurs de placer une confiance justifiée dans la qualité du service qu'il leur délivre.*
>
> *[Laprie 1996]*

On notera dans cette définition l'importance de l'utilisateur, placé immédiatement comme étalon de la sûreté de fonctionnement (« *utilisateur* » est à prendre ici au sens large : tous ceux à qui le système procure de la valeur ajoutée ; une banque étant autant utilisatrice d'un distributeur automatique que son client). Nous reviendrons dans la suite de ce mémoire

sur l'importance du point de vue de l'utilisateur pour orienter la mise en œuvre de la sûreté de fonctionnement dans les systèmes informatiques complexes.

Un système *défaille* lorsque le service qu'il délivre diverge du service attendu. La sûreté de fonctionnement cherche donc à éviter les défaillances, ou au moins à prévenir les plus catastrophiques. C'est ainsi qu'on préférera faire arriver les voyageurs d'un train en retard (ce qui est en soi une « défaillance » du système), plutôt que de leur faire risquer une collision mortelle (qui est aussi une défaillance, catastrophique cette fois).

Prévenir les défaillances requiert d'introduire deux nouvelles notions : *faute* et *erreur*. Une *faute* est la cause d'une défaillance. La notion de causalité étant récursive, l'identification d'une faute impose toujours une part de subjectivité. Par exemple, si un ion lourd traverse le circuit d'un satellite, entraînant sa perte, doit–on incriminer l'ion lourd (faute physique, externe au système, opérationnelle), ou doit-on chercher du côté du blindage du circuit qui serait mal conçu (faute humaine, « interne » au système, de conception) ? Une défaillance peut aussi résulter d'une combinaison de fautes multiples, mais faute de place, nous ne rentrerons pas dans ces subtilités.

Les fautes étant les causes des défaillances, la sûreté de fonctionnement cherche à les combattre, si possible en évitant qu'elle se produisent (*prévention*, c'est le premier moyen de la sûreté de fonctionnement), ou en les éliminant (*élimination*, c'est le second moyen de la sûreté de fonctionnement). Ces deux moyens ne sont cependant pas suffisants du fait de l'usure inévitable des systèmes (pour leur partie matérielle), des environnements souvent agressifs dans lesquels ils évoluent, et, particulièrement pour les logiciels, de l'impossibilité de concevoir des systèmes totalement exempts de fautes. Ces constatations conduisent à introduire la troisième notion, celle d'*erreur*, qui fait le lien entre fautes et défaillances. Un état erroné désigne l'état anormal d'un système, résultant de l'activation d'une faute, qui peut potentiellement amener à une défaillance. La notion d'erreur permet de désigner la chaîne de causalité qui conduit de la faute à la défaillance perçue par l'utilisateur (1.1).

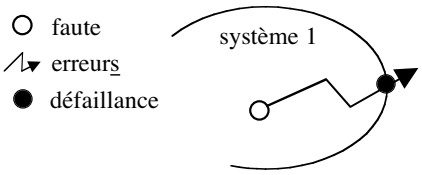


FIG. 1.1 : *Faute, erreurs et défaillance*

En cassant cette chaîne de causalité, c'est-à-dire en empêchant une erreur de se propager jusqu'à l'utilisateur, on améliore la sûreté de fonctionnement d'un système tout en tolérant ses fautes : c'est le troisième moyen de la sûreté de fonctionnement, la *tolérance aux fautes*, que nous détaillons plus avant dans la section qui suit.

La tolérance aux fautes

Un système est usuellement composé d'autres systèmes, plus petits, qui lui délivrent leurs services. La défaillance d'un de ses composants devient alors une faute pour le système global (figure 1.2). La tolérance aux fautes consiste à protéger l'utilisateur externe des conséquences des défaillances des composants internes du système.

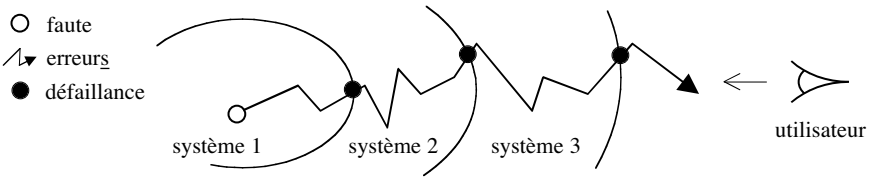


FIG. 1.2 : *Récursivité des fautes, erreurs et défaillances*

Cette protection peut prendre plusieurs formes, selon que l'on cherche à masquer complètement les fautes à l'utilisateur (par exemple dans un réseau local, en basculant de manière transparente d'un serveur DNS — résolveur de nom – primaire sur un serveur secondaire), à assurer une continuité de service en acceptant une dégradation temporaire de sa qualité, ou à prévenir les conséquences les plus catastrophiques d'une défaillance en mettant le système dans un état sûr (comme d'arrêter un train dès que l'on n'est pas certain que le tronçon de voie suivant soit libre). Ici se révèle le caractère polymorphique de la sûreté de fonctionnement, qui se décline selon de nombreux sous–aspects tels que disponibilité, fiabilité, robustesse, testabilité, maintenabilité, *etc.*, ces aspects pouvant parfois être antinomiques. Le réseau ferroviaire le plus fiable est par exemple celui où tous les trains sont à l'arrêt (état sûr), mais c'est aussi celui avec la plus faible disponibilité.

1.1.2 Redondance et diversité

En dépit de cette grande hétérogénéité de situations, deux grandes idées traversent la tolérance aux fautes : celles de *redondance* et de *diversité*. La redondance consiste à construire dans un système une sorte de « surcapacité » de service ou d'information, qui peut être partielle (un code correcteur d'erreurs, la vérification en ligne d'une condition de validité), ou totale (embarquer deux calculateurs là où un seul suffit en fonctionnement normal). La diversité consiste à s'assurer que la « surcapacité » ainsi ajoutée défaille indépendamment du service non redondant, ce que Laprie *et al.* dénomment comme « l'indépendance des redondances par rapport aux processus de création et d'activation des fautes » [Laprie 1996]. Le concept de diversité est essentiel, et est étroitement lié aux fautes que l'on cherche à tolérer, ceci en vertu d'une constatation très simple : aucun élément redondant ne peut protéger des fautes qui l'amènent à défaillir de la même manière et au même moment que le système qu'il double. C'est ainsi que le vol 501 d'Ariane 5 a échoué en 1996 suite à une défaillance de son système de référence inertielle, en dépit de ses deux calculateurs embarqués [ESA 1996]. La cause de la défaillance était une faute du logiciel et impactait

les deux calculateurs de la même manière, ceux-ci étant rigoureusement identiques, et faisant tourner le même programme. S'il s'était agi d'une avarie matérielle, les conséquences auraient été toutes autres, le second calculateur prenant la main, et masquant ainsi la faute. On voit ici qu'à chaque type de redondance (matérielle dans le cas d'Ariane 5, de conception dans l'avionique civile, d'information avec les codes correcteurs d'erreur), correspond une classe de fautes qui peuvent être tolérées.

1.1.3 Modèles de fautes

Nous l'avons vu avec l'exemple d'Ariane 5, il est essentiel de bien cerner la nature des fautes que l'on cherche à tolérer pour construire des mécanismes de tolérance qui contiennent la diversité appropriée pour être efficaces. De nombreuses classifications des fautes existent : on peut par exemple classer les fautes en fonction de la phase du cycle de vie du système au cours duquel elles apparaissent (fautes de conception — le cas du vol A.501, faute opérationnelle), de leur origine par rapport aux frontières du système (internes, externes), de leur nature phénoménologique (fautes physiques, fautes dues à l'homme), de leur persistance (fautes transitoires, fautes permanentes), *etc.* Chacune de ces classes de fautes appelle un traitement spécifique de tolérance aux fautes. Ainsi les fautes de conception ne peuvent être tolérées par une simple réplication matérielle à l'identique, c'est pourquoi en avionique civile des programmes répondant aux mêmes spécifications sont développés en plusieurs versions par des équipes différentes, ce qu'on appelle en anglais le *N-version programming* [Avizienis & Kelly 1984].

Dans un système composite, dans lequel les défaillances des composants sont considérées comme les fautes à combattre, on distingue plusieurs types de défaillances (qui sont donc des fautes pour le système global) selon le comportement du composant défaillant :

Silence sur défaillance
: Un composant à *silence sur défaillance* soit se comporte conformément à sa spécification, soit cesse toute interaction avec les autres composants du système (ce qu'est une interaction dépendra du modèle du système considéré : envoyer des messages dans un système de communication par messages, interagir avec la mémoire partagée dans un système à mémoire partagée).

Défaillance incontrôlée
: Un composant qui suit un modèle de *défaillances incontrôlées* peut se comporter de manière totalement arbitraire. Dans un système de communication par messages, un tel composant peut envoyer des messages avec un contenu incorrect (défaillance en valeur), des messages trop tôt ou trop tard (défaillance temporelle), voir même des messages non prévus [Powell 1991]. Ces défaillances sont parfois appelées des défaillances *byzantines*.

1.1.4 Les mécanismes de tolérance aux fautes

Les mécanismes de tolérance aux fautes font typiquement intervenir les deux primitives suivantes :

La détection d'erreur permet au système d'identifier un état erroné au cours de son fonctionnement. La détection s'accompagne éventuellement d'un diagnostic pour évaluer le degré de propagation de l'erreur, et l'importance des dommages causés.

Le recouvrement d'erreur consiste à ramener le système vers un fonctionnement sûr (mais éventuellement dégradé) en remplaçant l'état erroné et donc potentiellement dangereux, par un état sans erreur.

On distingue trois formes de recouvrement :

Recouvrement par reprise
Le système est ramené dans un état antérieur à partir duquel il reprend son fonctionnement (un exemple de la vie quotidienne consiste à rallumer son ordinateur de bureau après qu'il s'est bloqué, et à recommencer à travailler sur la dernière version sauvegardée de son travail).

Recouvrement par poursuite
Le système est amené dans un nouvel état, connu à priori, à partir duquel il peut continuer son fonctionnement (souvent de manière dégradée). Ce mode de recouvrement est souvent très dépendant de l'application.

Recouvrement par compensation
La redondance contenue dans l'état erroné suffit à éliminer l'erreur et à poursuivre l'exécution.

Si l'on estime que la faute est encore présente après le recouvrement (par exemple dans le cas d'une faute *persistante*), et que la probabilité d'une nouvelle activation n'est pas suffisamment faible, un traitement de faute, constitué d'un *diagnostic* suivi d'une *passivation* s'impose en complément. Nous ne les aborderons pas plus en détail dans ce mémoire.

La redondance est souvent utilisée pour réaliser détection d'erreur et recouvrement : des erreurs peuvent ainsi être détectées en comparant des résultats fournis par plusieurs répliques, ou en vérifiant la conformité d'un résultat fourni par un composant par rapport à des contraintes exprimées sur son fonctionnement (on parle alors d'un composant *auto-testable*). La programmation par contrat [Meyer 1997] peut par exemple être vue comme une forme de redondance dans lequel le comportement d'un module logiciel est exprimé deux fois de manières différentes : de par son implémentation d'une part, et de par les pré- et post–conditions de son contrat d'autre part. Le principe de l'*extreme programming* [Beck 1999] qui consiste à faire coder à un même programmeur des tests unitaires avant l'implémentation d'un module relève de la même intuition. Là encore, le programmeur exprime deux fois de deux manières différentes son intention, d'abord sous forme de test, puis dans une implémentation. L'activation du test révèle toute incohérence entre les deux.

La redondance joue aussi un rôle essentiel dans le recouvrement par compensation (par exemple plusieurs répliques d'un même service votent pour décider du résultat).

1.1.5 Systèmes distribués et tolérance aux fautes

Répartition physique[1] et tolérance aux fautes sont très souvent associées, la réplication d'un même service de manière distribuée permettant d'assurer redondance et indépendance des modes de défaillance. Quels modes de défaillance sont indépendants dépend de la nature de la distribution. Deux répliques installées physiquement dans le même bâtiment ne seront pas par exemple tolérantes à un incendie qui ravagerait le bâtiment.

La distribution impose de coordonner les répliques par des moyens de synchronisation en grande partie conditionnés par la nature des communications entre les répliques. Ces moyens de synchronisation diffèrent notamment par le degré de contrôle imposé aux répliques, leur granularité, leur robustesse, leur prédictabilité. Les architectures rencontrées vont ainsi de processeurs synchronisés sur la même horloge[2] [Taylor & Wilson 1989] (*lockstep processors*), dont les sorties sont comparées, à des systèmes utilisant des réseaux locaux à jetons [Powell 1991], en passant par les liaisons point–à–point rencontrées dans l'avionique civile (norme ARINC 429 [AEEC]).

Ces différentes architectures se répartissent en deux grands types : les systèmes distribués *synchrones*, et ceux dits *asynchrones*. Un système *synchrone* est un système dans lequel les latences de communication et les vitesses respectives d'exécution des entités distribuées sont connues. Un système asynchrone est un système dans lequel l'une de ces informations n'est pas garantie. Systèmes synchrones et asynchrones sont en fait les deux extrêmes d'un spectre continu de propriétés. Il existe ainsi plusieurs modèles de synchronie partielle, comme l'asynchronie temporisée (*Timed Asynchronous Systems*) [Fetzer & Cristian 1995].

Les systèmes asynchrones, parce qu'ils imposent moins de contraintes sur la plate–forme de communication, sont plus flexibles, moins lourds à déployer, plus faciles à reconfigurer et à faire évoluer. Ils souffrent cependant d'un handicap majeur en tolérance aux fautes : ils ne permettent pas de résoudre des problèmes essentiels comme le consensus ou la diffusion atomique dès lors qu'au moins un des processus distribués est autorisé à défaillir par arrêt[3] [Fischer et al. 1985]. Intuitivement, cette impossibilité résulte du fait que dans un système asynchrone, il n'est pas possible de distinguer un système extrêmement lent d'un système qui a défailli.

En pratique ce verrou est contourné en ajoutant des hypothèses de synchronie partielle (comme des détecteurs d'erreurs imparfaits [Chandra & Toueg 1996]), qui ne sont alors pas vérifiées avec une probabilité de 1, mais lèvent le résultat d'impossibilité en factorisant dans

[1] Nous parlerons indifféremment de « systèmes distribués » ou de « systèmes répartis ».

[2] La nature « distribuée » d'un tel système est certes discutable.

[3] Notons que ce résultat d'impossibilité vaut pour les algorithmes déterministes. Il est possible de résoudre le problème du consensus dans un environnement asynchrone dès lors que l'on autorise les processus à générer des valeurs aléatoires [Or 1983].

une propriété particulière du système la quantité de synchronisme nécessaire à la résolution de la tolérance aux fautes.

1.2 Systèmes logiciels complexes

Dans cette section, nous discutons en quoi consiste la « complexité » des systèmes logiciels d'aujourd'hui. Le sujet est immense, et touche aussi bien à l'architecture des logiciels (comment organiser macroscopiquement un logiciel), qu'aux processus de développement (comment organiser l'activité de construction d'un logiciel), qu'aux langages de programmation (comment organiser microscopiquement un logiciel), *etc.* Nous tentons donc de donner un aperçu nécessairement imparfait de ces différents aspects, en faisant ressortir les lignes de forces qui selon nous soutiennent les évolutions présentes et passées.

1.2.1 Buts et difficultés du génie logiciel

Le saut de complexité qui existe aujourd'hui entre les éléments de base de l'informatique (le transistor, la porte logique) et les systèmes qu'ils permettent de réaliser est devenu tel qu'aucun être humain n'est plus capable d'appréhender seul l'ensemble de la chaîne de conception, qui à partir des équations électromagnétiques (Maxwell, Schrödinger) permet de réaliser des systèmes d'information porteurs de sens (Internet par exemple) et capable d'actions à l'échelle de la planète (un système de réservation, un système de transport). Plusieurs êtres humains doivent collaborer pour les construire. Cette coopération peut être directe et explicite, au sein d'un même projet, ou indirecte, par la (ré–)utilisation dans un système d'éléments logiciels développés indépendamment (par exemple des *COTS*, *Commercial Off the Shelf*). Sauf cas très particuliers, comme la NASA avec l'OS temps réel VXWORKS [Reeves 1997], ou la SNCF avec l'intergiciel ILOG BROKER [RIS 2002], l'utilisateur d'une plate-forme ou d'une bibliothèque n'a alors pas de contact direct avec les développeurs du produit qu'il réutilise.

Cette nécessité de collaboration induit une grande partie des difficultés rencontrées lors de la construction logicielle. Elle exige de maîtriser les interactions à la fois à l'intérieur du logiciel lui-même, et entre les personnes impliquées directement ou indirectement dans son développement. Ces interactions se déploient à la fois dans l'espace (plusieurs millions de lignes de code[4], plusieurs centaines de développeurs[5]), et dans le temps (un gros projet industriel s'étale souvent sur plusieurs années[6]). Cette double extension spatiale et temporelle, par les interdépendances qu'elle induit, rend les projets informatiques particulièrement sensibles aux problèmes de la gestion et de la propagation du changement. Les spécifications

[4] A titre d'exemple, le noyau du système d'exploitation GNU/LINUX, développé à l'origine par des bénévoles, contient plus de 3,6 millions de lignes de code de C dans sa version 2.4.18.

[5] Toujours pour LINUX 2.4.18, plus de 400 personnes sont recensées comme ayant participé au noyau, et cette liste n'est que partielle.

[6] Le courrier électronique de Linus Torvalds annonçant l'existence de ce qui allait devenir LINUX est daté du 25 août 1991.

d'un projet, mais aussi la nature des technologies disponibles, les composants tiers utilisés ont toutes les chances d'être modifiés lorsque les projets s'étalent sur une dizaine d'années, impliquent des centaines d'acteurs, et contiennent plusieurs millions de lignes de code.

Sans refaire l'histoire du génie logiciel, l'on peut dire que la notion de modularité a très tôt été considérée comme un outil essentiel pour aborder ces multiples difficultés [Parnas 1972]. La modularité permet de diviser pour régner, en assurant le partitionnement d'un système complexe en des modules plus simples. Ces modules peuvent être alors abordés par des équipes distinctes, voire réutilisés à partir de projets antérieurs ou de composants du marché. Mais pour qu'une approche par modules fonctionne, deux propriétés essentielles doivent être assurées :

L'interopérabilité
: Un consensus doit exister sur la syntaxe et la sémantique des interfaces qui vont articuler les différents modules *avant* que les modules ne soient réalisés. Bertrand Meyer dans [Meyer 1997] cite l'anecdote suivante pour illustrer les conséquences catastrophiques que peut avoir l'oubli de ce principe :

> *Last week, AMR Corp., the parent of American Airlines, Inc. said it fell on its sword trying to develop a state-of-the-art, industry-wide system that could also handle car and hotel reservations. AMR cut off development of its new Confirm reservation system only weeks after it was supposed to start taking care of transactions for partners Budget, Rent-A-Car, Hilton Hotels Corp. and Marriott Corp. […] The main pieces of the [$125 million, 4–year–old] project had been developed separately, by different methods. When put together, they did not work with each other.*
>
> *[Mercury 1992] cité dans [Meyer 1997]*

La maîtrise des interdépendances–le découplage
: Diviser pour régner est illusoire si la « division » n'est que de façade. Pour permettre à de nombreuses équipes de travailler en parallèle, en réutilisant des éléments logiciels écrits par ailleurs, il est essentiel de maîtriser les interdépendances à l'intérieur d'un code logiciel. Plus le projet est grand en taille et en durée, et plus ces interdépendances risquent de devenir des canaux de propagation du changement. Les techniques d'interfaçage choisies entre les différents modules d'un projet doivent donc à la fois permettre de tracer les interdépendances, mais aussi, et avant tout, permettre de les éviter, en assurant un découplage maximal entre les modules. Dans le cas contraire, la moindre modification à un endroit du projet peut remettre en cause tout le travail déjà effectué et obliger à de longues et fastidieuses corrections, voire, lorsque les interdépendances sont mal comprises, avoir des conséquences dramatiques sur le système lui-même.

  Ainsi, si nous revenons au vol inaugural d'Ariane 5, la désintégration du lanceur a eu pour cause un débordement de valeur lors d'une conversion de donnée dans le logiciel d'alignement de la fusée (la variable de résultat était « trop petite » pour contenir ce résultat). Ce logiciel d'alignement avait été écrit pour Ariane 4 et réutilisé dans Ariane 5. Voilà ce qu'on peut lire à son sujet dans le rapport de la commission d'enquête :

> *L'exigence qui est à l'origine de la poursuite du fonctionnement du logiciel d'alignement après le décollage a été imposée il y a plus de 10 ans pour les premiers modèles de lanceurs Ariane. [...] Cette exigence n'a pas lieu d'être sur Ariane 5, dont la séquence de préparation est différente. Elle a été conservée pour des raisons de commodité, qui semblent reposer sur le principe selon lequel il n'est pas opportun, sauf preuve contraire, de procéder à des modifications sur des logiciels qui ont bien fonctionné sur Ariane 4.*
>
> *[ESA 1996]*

L'échec du vol 501 d'Ariane 5 peut donc être vu comme un problème de réutilisation mal maîtrisée : un module hérité d'Ariane 4 qui n'était plus requis sur la nouvelle version a dû fonctionner dans un contexte contraire à ses spécifications, faute d'une bonne maîtrise des interdépendances entre le module et la trajectoire de la fusée. Des problèmes de gestion d'exceptions annexes ont finalement abouti à la catastrophe que l'on connaît.

Il est relativement facile après coup de critiquer les concepteurs d'un système qui a failli. C'est une facilité dans laquelle nous nous retiendrons de tomber. On peut relire l'histoire des avancées en architecture logicielle et en langages de programmation comme une quête continue pour sans cesse éviter ce type de catastrophes en facilitant la structuration modulaire des logiciels et leur interopérabilité. Dans les paragraphes qui suivent nous abordons quelques-uns des paradigmes qui ont façonné cette quête, *les systèmes d'exploitation*, *l'orienté objet*, *les intergiciels*, pour finir par dire un mot sur les architectures en couches.

1.2.2 Système d'exploitation et intergiciel

Les systèmes d'exploitation (*Operating System* ou *OS* en anglais) sont sans doute l'une des plus anciennes formes de réutilisation logicielle. D'abord développés comme des bibliothèques de support d'application, avant tout pour faciliter la gestion des périphériques (affichage, stockage), les systèmes d'exploitations se sont enrichis et ont évolué pour fournir les propriétés suivantes [Silberschatz et al. 2002, Tanenbaum & Woodhull 1997] :

Abstraction
: Un système d'exploitation masque les détails et les particularités de la gestion du matériel au programmeur d'application. Ce dernier n'a pas à se soucier de démarrer ou à arrêter le disque dur, à développer un mode de gestion de la mémoire physique, ou à programmer un protocole réseau particulier. Les spécificités du matériel deviennent invisibles, *transparentes*, au développeur d'application.

Protection
: Le système d'exploitation empêche les utilisations dangereuses ou non–autorisées du matériel. Parce qu'il prend la main au démarrage de la machine, l'OS a toute latitude pour contrôler les actions des utilisateurs sur le système (ce qui n'implique pas qu'il le fasse parfaitement, sans quoi la sécurité informatique en serait grandement simplifiée).

Multiplexage
: Un système d'exploitation permet de partager les ressources (limitées) d'une machine matérielle entre plusieurs programmes et plusieurs utilisateurs. Selon le contexte d'utilisation, le système d'exploitation peut tenter de maximiser le sentiment de réactivité auprès de ses utilisateurs, la juste répartition des ressources, et le taux d'utilisation de la machine (OS dit généralistes), ou tenter d'assurer des échéances d'exécution déterministes (OS temps réel), ou encore minimiser la consommation énergétique (OS embarqué).

Interopérabilité
: Si les premiers systèmes d'exploitation étaient spécifiquement liés à une plate-forme matérielle donnée, et changeaient du tout au tout même avec les machines successives d'un même constructeur, les concepts utilisés se sont aujourd'hui sédimentés dans des interfaces standard comme POSIX (la norme derrière les systèmes UNIX) ou WIN32 (l'interface de programmation de l'OS WINDOWS de MICROSOFT). Ces interfaces assurent la compatibilité des programmes écrits pour les versions successives d'un même système d'exploitation, et offrent un modèle de programmation unifié sur du matériel hétérogène. Elles renforcent donc à la fois l'interopérabilité et le découplage des programmes vis-à-vis du matériel.

Au fil du temps et par une sorte de « capillarité conceptuelle », certaines des préoccupations des systèmes d'exploitation ont « diffusé » à la fois vers le bas, dans les micro-processeurs, et vers le haut, dans ce que l'on a appelé les *intergiciels*. Les micro-processeurs modernes intègrent par exemple un mécanisme de protection matérielle sous la forme de niveaux de *privilège* [Hyde 2003]. En mode privilégié, ou *mode noyau*, toutes les opérations sur le matériel sont autorisées. Seule la partie la plus sensible du système d'exploitation, le *noyau*, utilise ce mode. Les applications qui tournent au-dessus de l'OS n'ont accès qu'au mode non–privilégié, ou *mode utilisateur*, et doivent obligatoirement passer par le noyau de l'OS pour par exemple manipuler la mémoire physique, ou commander les périphériques. Ce passage d'un mode à l'autre s'opère par un mécanisme dit de « trap » (déroutement), qui limite les interactions entre le noyau et les applications à quelques opérations bien définies, appelées *appels système* (*syscall* en anglais).

La gestion de la mémoire est un autre exemple du phénomène de diffusion vers le bas des préoccupations. La plupart des processeurs modernes contiennent un module de gestion de la mémoire (*memory management unit*, MMU), qui permet de donner l'illusion de plusieurs espaces mémoire séparés, comme si la machine avait une mémoire physique pour chacun des programmes qu'elle exécute. La projection de ces espaces mémoire « virtuels » sur la mémoire physique réelle par la MMU n'est visible qu'en mode noyau, et joue un rôle très important pour la protection (en permettant un confinement des espaces mémoire virtuels), et le multiplexage des ressources (en intégrant dans le silicium même un mécanisme de partage « transparent » de la mémoire).

Comme nous l'avons mentionné, par un mouvement inverse de cette « diffusion vers le bas », une « croissance par le haut » des systèmes d'exploitation peut aussi être observée. Des services de support d'application se sont peu à peu développés au-dessus des OS, notamment

pour la programmation distribuée, la programmation d'interfaces graphiques, ou encore la réalisation de systèmes transactionnels. Ces couches supplémentaires sont un peu aux systèmes d'exploitation ce que ces derniers sont vis-à-vis du matériel. Elles fournissent des modèles de programmation de plus haut niveau que celui des OS et simplifient ainsi le développement d'applications plus complexes tout en garantissant un plus haut niveau d'interopérabilité et de découplage.

Les bibliothèques « système » sont un premier exemple de ce phénomène : parce que le choix des appels système d'un noyau constitue un élément décisif de la conception d'un système d'exploitation, et ne sont souvent pas assez nombreux pour réaliser une norme comme POSIX, des bibliothèques de support dites *bibliothèques système*, sont usuellement fournies au-dessus du noyau pour encapsuler les appels système spécifiques, et proposer un modèle de programmation standard plus élaboré [Tanenbaum & Woodhull 1997]. Ces bibliothèques fournissent usuellement des modèles de programmation d'une abstraction bien plus riche que les primitives brutes du noyau, et rendent indépendants les appels système de l'interface présentée aux applications. Pour des raisons de portabilité, une application n'utilise normalement jamais directement les appels système d'un noyau, et passe toujours par des bibliothèques système standard. Pour cette raison, on considère usuellement les bibliothèques système comme faisant partie intégrante du système d'exploitation, bien qu'elle fonctionnent en mode utilisateur.

Lorsque des services de support sont rajoutés *au-dessus* des bibliothèques système, on les appelle usuellement des *intergiciels* (ou *middleware*), parce qu'ils servent alors d'intermédiaire entre les applications et le système d'exploitation. Dans cette thèse nous traiterons plus particulièrement des intergiciels orientés communication, et plus spécifiquement des bus à objets avec la norme CORBA [OMG 2002a]. Ce type d'intergiciels servent de « glu » à des morceaux d'application s'exécutant sur des machines distantes. Ils masquent une partie de la complexité de la programmation distribuée en permettant à un développeur de manipuler de la même manière des entités locales et distantes. Leur développement a fortement été marqué par le paradigme orienté objet, et il est difficile de parler de l'un sans l'autre. Nous commençons donc par dire quelques mots sur l'orienté objet dans la section qui suit, ce qui nous permettra de nous étendre ensuite plus longuement sur la notion d'intergiciel et de bus à objets.

1.2.3 L'orienté objet

Né à la fin des années soixante avec SIMULA [Dahl & Nygaard 1966], développé durant les années soixante-dix pour la simulation ou l'intelligence artificielle, le paradigme orienté objet a commencé à pénétrer le monde industriel durant les années 80, notamment avec le langage SMALLTALK, pour connaître un succès très important à partir des années 90, avec C++ et JAVA [Joy et al. 2000]. L'idée originelle consiste à proposer un modèle de programmation qui soit le plus proche possible du monde réel (des objets, tels de petits êtres vivants, interagissent entre eux). Une des motivations de ce paradigme est d'éviter un trop grand saut conceptuel entre l'espace d'un problème (quel doit être l'effet d'un programme à

construire sur le monde réel), et l'espace des solutions (comment construire le programme pour répondre au problème).

L'orienté objet est en fait un aboutissement des concepts développés autour de la notion de modularité dans des langages comme MODULA [Wirth 1985], ou ADA. Un objet est une entité logicielle qui encapsule un comportement et des données derrière une *interface*. Cette interface peut être vue comme la « peau » de l'objet, comme une membrane qui sépare très clairement sa structure interne de son environnement extérieur. Cette interface–membrane réglemente notamment les interactions entre l'objet et l'univers dans lequel il évolue. Elle constitue une sorte de contrat entre l'objet et son environnement, et permet de s'abstraire de l'implémentation effective de l'objet (le *comment*) en ne présentant au monde extérieur qu'une vision abstraite de son comportement (le *quoi*). Concrètement l'interface d'un objet décrit les propriétés, appelées *attributs*, de cet objet, et les commandes, appelées *méthodes*, que cet objet peut réaliser.

Cet aspect « encapsulatoire » n'est pas propre à l'orienté objet et se retrouve dans le concept de *module* proposé par des langages comme MODULA. L'orienté objet y ajoute la notion d'instance : les objets peuvent être créés dynamiquement, usuellement à partir de sorte de « moules » à objets, appelés *classes*, ce qui n'est pas possible pour un module. Un objet possède une identité, qui permet d'interagir avec lui, et qui peut-être diffusée sous forme de *référence* à d'autres objets du système. Un objet représente donc à la fois un comportement, une sorte de morceau de programme, et une donnée dynamique, qui peut servir de paramètre d'entrée à des traitements. Les classes qui organisent les objets forment usuellement une hiérarchie de types, ce qui permet alors de réaliser ce que l'on nomme le *polymorphisme*, ou encore principe de substitution [Liskov 1987]. Un programmeur n'a plus à préciser explicitement sur quelles données tourne un programme, il lui suffit, par un système de types appropriés, de préciser quel *contrat* doivent réaliser les objets auxquels s'appliquera son code.

```
class Déménageur {
  déménagerVaisselle(Vaisselle deLaVaisselle) ;
  déménagerArmoire  (Armoire   uneArmoire   ) ;
  déménagerCanapé   (Canapé    unCanapé     ) ;
  [...]
} // EndClass
```

FIG. 1.3 : *Un déménageur sans polymorphisme*

Imaginons par exemple un programme modélisant des déménageurs. Sans possibilité de substitution, la classe décrivant les objets « déménageurs » devrait contenir une méthode « déménager » pour chaque catégorie d'objets que le déménageur peut prendre en charge (figure 1.3).

Si en revanche nous factorisons le fait de pouvoir être déménagé dans une classe particulière, « `ObjetPouvantÊtreDéménagé` », nous pouvons réunir toutes les méthodes `déménager`

```
class ObjetPouvantÊtreDéménagé {
  saisirObjet() ;
  leverObjet () ;
  poserObjet () ;
  porterObjetDUnEndroitAUnAutre ( Endroit unEndroit,
                                  Endroit unAutreEndroit ) ;
  [...]
} // EndClass

class Déménageur {
  déménagerObjet(ObjetPouvantÊtreDéménagé unObjetDéménageable) ;
  [...]
} // EndClass
```

FIG. 1.4 : *Factorisation de la « déménageabilité » par polymorphisme*

en une seule méthode en déclarant chacune des classes `Vaisselle`, `Armoire`, `Canapé` comme des *sous-classes* de `ObjetPouvantÊtreDéménagé` (figure 1.4).

1.2.4 Les intergiciels et les objets : les bus à objets

Comme nous l'avons mentionné, les intergiciels étendent au-delà des considérations historiques des systèmes d'exploitation les objectifs d'abstraction, de transparence et d'interopérabilité. Les *bus à objets* (en anglais *Object Request Brokers*, ou *ORB*) sont des intergiciels orientés communications qui mettent à profit les concepts de l'orienté objet que nous venons de présenter pour faciliter l'intégration de grands systèmes. Un bus à objets applique à l'échelle macroscopique d'un système informatique les notions d'objet, d'interface, d'entité, de référence, proposant ainsi un mode d'articulation des modules d'un système intuitif et de haut niveau. Les facilités suivantes sont parmi les plus communément offertes par un bus à objets :

Transparence de la distribution
: Le développeur manipule les entités distantes comme des objets locaux. Les appels de méthodes locaux et distants sont syntaxiquement identiques, le système de désignation des objets distants est intégré dans l'ORB qui opère automatiquement la résolution des références à l'insu du développeur.

Transparence vis-à-vis des langages de programmation
: En définissant une représentation commune des données, utilisée pour faire communiquer les différents objets répartis entre eux, il devient possible de faire interagir des programmes écrits dans des langages différents. L'ORB doit alors définir pour chaque langage pris en charge une correspondance (*mapping*) entre le modèle de programma-

tion du langage (types de base, mécanisme d'invocation, syntaxe), et le modèle commun de l'ORB.

Transparence vis-à-vis des OS et du matériel
Nous avons vu comment les systèmes d'exploitation permettaient de rendre les programmes indépendants du matériel sur lequel ils tournent. Cette indépendance n'est cependant effective qu'au niveau des sources du programme, écrit en langage de haut niveau, et avant compilation en langage machine. Deux applications utilisant les mêmes primitives de communication sur des matériels différents, même avec des OS identiques, ne sont par exemple pas assurées de pouvoir se comprendre, du fait des différences de représentation binaire. Un ORB tel que CORBA en définissant son propre encodage de donnée lève cet obstacle, et permet à des plates-formes hétérogènes de collaborer selon une sémantique non–ambiguë.

Transparence vis-à-vis de la nature du réseau
Un ORB masque l'hétérogénéité du réseau sur lequel s'opèrent les communications : support physique (mémoire partagée, bus Ethernet, VME), protocoles de communication.

Il existe plusieurs normes d'ORB et toutes n'offrent pas tous ces degrés de transparence. Certaines sont spécifiques à des langages comme RMI (*Remote Method Invocation*), propre à JAVA, ou spécifiques à un système d'exploitation, comme DCOM pour WINDOWS. L'une des normes d'ORB les plus utilisées en pratique, CORBA [OMG 2002a], se distingue par son exceptionnel degré d'interopérabilité, et prend en charge les quatre aspects de transparence présentés ci-dessus. CORBA étant une norme de l'OMG (*Object Management Group*), elle n'est pas liée à un fournisseur particulier, et accorde un soin particulier aux problèmes d'interopérabilité entre différentes implémentations. Le standard CORBA s'abstrait par ailleurs volontairement de toute considération d'implémentation. Par exemple, en dehors des contraintes minimales nécessaires à l'interopérabilité, le standard CORBA ne précise pas comment une implémentation de la norme doit interagir avec le système d'exploitation qui la supporte, ni comment elle doit utiliser les couches de communication du système. En imposant le moins de contraintes possibles, la norme CORBA étend les contextes possibles d'utilisation du standard, et laisse entière liberté aux fournisseurs d'ORB pour développer des solutions optimisées pour tel ou tel contexte d'exécution (temps-réel [Schmidt 2002], embarqué [OMG 2002g], mobile [Grace et al. 2003]). Mais en coupant complètement l'utilisateur de l'ORB des choix internes d'implémentation, en ne lui laissant accès qu'à des primitives de très haut niveau (résolution de nom, envoi de requêtes), CORBA introduit une frontière d'opacité.

Nous touchons ici au lien fort qui unit en informatique les notions de *transparence* et d'*opacité*. Ces notions désignent les faces duales d'une même réalité : on dit d'une fonctionnalité dans un modèle de programmation qu'elle est transparente lorsque le programmeur n'a pas à la prendre explicitement en considération pour pouvoir l'utiliser. C'est le cas dans CORBA comme nous venons de l'expliquer ; c'est aussi le cas dans un OS multi-tâches avec le partage du processeur entre processus concurrents : il n'est pas nécessaire durant l'écriture d'un programme de prendre en compte les autres processus qui potentiellement pourront partager la machine. Cette transparence est rendue possible par *l'abstraction* des complexités

d'implémentation par l'interface du modèle de programmation. *Transparence* et *abstraction* s'accompagnent donc toujours d'une *opacification* des couches basses du système. C'est à ce problème d'opacification que nous nous intéressons dans les paragraphes suivants.

1.2.5 Le problème posé par les systèmes multi-couches

Nous venons de présenter très rapidement certains des aspects architecturaux utilisés pour assurer découplage et interopérabilité dans les logiciels d'aujourd'hui : systèmes d'exploitation, paradigme orienté objet, intergiciels. Cette présentation, inévitablement incomplète et parcellaire, fait cependant ressortir l'importance de la structuration en couches des systèmes abordés. Chaque couche, en masquant (partiellement ou complètement) la couche précédente par de nouvelles abstractions, introduit une sorte de pare-feu sémantique qui assure un découplage vertical des différents éléments du système et permet de rendre le service réalisé (perçu par l'utilisateur de la couche en question) indépendant du substrat matériel et logiciel qui le supporte.

Cette forme de structuration présente de nombreux avantages. Elle permet par exemple de construire et de valider les différents éléments d'un système de manière modulaire. En se basant — parfois implicitement — sur un système de conditions / garanties, les fournisseurs de chaque couche peuvent se concentrer sur le niveau dont ils ont la responsabilité (compilateur, OS, bibliothèque), en assurant que celui-ci se comporte correctement sous l'hypothèse que les niveaux sous-jacents sur lesquels il s'appuie sont eux aussi corrects [Meyer 1997]. Cette structuration permet aussi de développer des environnements de développement plus intuitifs, plus riches, plus faciles à mettre en œuvre et à déployer. En se basant sur des interfaces standardisées (IA32, POSIX, CORBA, Java, RMI, ...), elle rend les intégrateurs de système indépendants de leurs fournisseurs, assure la pérennité des développements en permettant l'évolution séparée des implémentations sous-jacentes.

Du point de vue de la tolérance aux fautes, cependant, cette structuration stratifiée induit un certain nombre de problèmes que nous allons tenter d'esquisser ici. Comme nous l'avons mentionné, l'abstraction fournie par chaque nouvelle couche s'opère au prix d'une opacification de l'implémentation sous-jacente. Or l'introduction d'une couche augmente les risques de fautes logicielles (fautes de conception à l'intérieur de la couche, fautes d'utilisation de la couche, *etc.*) tout en restreignant les capacités de contrôle et d'observation par ses utilisateurs. Les erreurs créées par les fautes traversent hélas les frontières d'abstraction mises en place par chaque couche, si bien qu'un développeur qui souhaite implémenter des mécanismes de tolérance aux fautes au-dessus d'une architecture complexe doit lutter en aveugle contre des dangers qui lui sont rendus « transparents », avec des moyens limités.

La tolérance aux fautes a bien sûr en partie été prise en compte dans les différents paradigmes que nous avons évoqués : confinement des espaces d'adressage virtuel au niveau micro–processeur, codes de retour d'erreur pour les appels système, exceptions dans les langages orientés objet, et dans les intergiciels. Ces moyens sont cependant souvent rudimentaires, et ne peuvent pas être modifiés par les développeurs « clients » d'une solution. La réalisation de la tolérance aux fautes dans des systèmes complexes en couches ne peut

souvent pas être abordée d'un point de vue uniquement parcellaire, en ne considérant qu'une seule couche indépendamment des autres éléments avec lesquels elle interagit. La tolérance aux fautes est un concept orthogonal aux couches d'un système, et donc impose une connaissance, qui peut être plus ou moins étendue, de ce qui s'y déroule. Pour réaliser la détection d'erreur, des informations doivent parfois être collectées à différents niveaux de l'architecture. Pour réaliser une capture d'état de manière « intelligente », des informations de niveaux d'abstraction différents sont souvent nécessaires pour une implémentation portable et efficace.

À ceci s'ajoute le souci, général pour les systèmes informatiques complexes, de découplage et de composabilité : les mécanismes de tolérance aux fautes doivent pouvoir être développés séparément du reste du système. Plusieurs raisons motivent cette préoccupation :

- l'imbrication dans le même code de préoccupations directement fonctionnelles et d'aspects transversaux comme la tolérance aux fautes rend malaisé le développement des programmes, en superposant deux logiques distinctes et souvent orthogonales ;
- la tolérance aux fautes, parce qu'elle fait intervenir plusieurs niveaux d'abstraction, introduit des dépendances supplémentaires entre les couches, et affaiblit le principe de découplage vertical ;
- le développement de mécanismes de tolérance aux fautes requiert souvent un savoir d'expert, indépendant des connaissances métier nécessaires au développement des aspects applicatifs du système.

Pour faire face aux changements, la tolérance aux fautes doit pouvoir être modifiée aisément, parfois même à l'exécution, sans arrêt du système. Des algorithmes développés indépendamment du reste du système, sous la forme de composants sur étagère, doivent pouvoir y être connectés facilement, de manière orthogonale, sans impacter les parties applicatives du système.

L'on peut donc résumer la problématique de la tolérance aux fautes dans les architectures complexes par les deux points suivants :

- Comment construire la tolérance aux fautes dans les différentes couches du système, en préservant la structuration en strates, tout en se libérant de l'opacité des interfaces entre couches ?
- Comment assurer la modularité, l'adaptabilité de la tolérance aux fautes, sa séparation des aspects directement fonctionnels du système ?

Ce constat induit finalement une nécessité d'inventer un nouveau modèle à composants dont l'encapsulation n'est pas remise en cause pour leur utilisation fonctionnelle, mais qui offrent de nouvelles propriétés pour la mise en œuvre de mécanismes non-fonctionnels. Cette affirmation est le fond de la thèse défendue dans ce mémoire.

1.3 Conclusion du chapitre

Comme par le passé, les systèmes informatiques d'aujourd'hui doivent à la fois réaliser des fonctions toujours plus complexes, être réalisés dans des temps toujours plus courts et à des prix toujours plus compétitifs. Aujourd'hui cependant, ces systèmes sont en passe de s'intégrer à notre environnement quotidien à un degré jamais égalé jusqu'alors, à la fois riche de promesses et de dangers.

L'utilisation banalisée de logiciels complexes construits à partir de nombreux composants hétérogènes pour des emplois de plus en plus critiques pose problème. Les risques encourus, aussi bien humains qu'économiques, exigent de pouvoir mettre en place des mécanismes de tolérance aux fautes indépendamment des composants réutilisés, pour durcir leur robustesse et exclure tout scénario catastrophe.

La réflexivité logicielle fait partie des technologies qui ont été proposées pour relever ce défi. Le propos de ce travail de thèse est d'étudier dans quelle mesure ces solutions réflexives peuvent s'appliquer aux systèmes complexes que nous venons de présenter, ou si elles doivent être étendues et comment. Dans la suite de ce manuscrit, après avoir rappelé les notions de réflexivité (chapitre 2 page suivante), nous présentons les architectures tolérantes aux fautes (réflexives ou non) qui ont été proposées jusqu'à maintenant, et nous montrons pourquoi la réflexivité doit être étendue pour dépasser leurs limites (chapitre 3 page 35). Nous détaillons ensuite la démarche que nous avons mise en place ainsi que les principes et concepts qui la soutiennent (chapitre 4 page 53). Nous appliquons cette démarche au cas de la réplication d'applications distribuées et validons notre approche sur un cas d'étude concret en nous intéressant à la réplication d'applications CORBA à brins d'exécution[7] multiples (chapitre 5 page 75). Nous concluons finalement sur les perspectives de notre travail (page 117).

[7] Nous utiliserons dans la suite de cet exposé l'expression « brin d'exécution », ou de manière plus concise « brin », pour traduire l'anglais *thread*. Les brins désignent les différentes activités concurrentes d'une application qui partagent un même espace d'adressage. De façon similaire, nous traduirons l'adjectif *multithreaded* par « à brins d'exécution multiples », et le substantif *multithreading* par « multitraitement ».

Chapitre 2

Les architectures réflexives

> Πότερον οὖν δὴ ῥᾴδιο τυγχάνει τὸ γνῶναι ἑαυτόν, καί τις ἦν φαῦλος ὁ τοῦτο ἀναθεὶς εἰς τὸν ἐν Πυθοῖ νεών, ἢ χαλεπόν τι καὶ οὐχὶ παντός ;
>
> *Et donc, est-il si facile de se connaître soi–même, et celui qui grava la devise du temple de Delphes était–il stupide, ou est-ce une chose difficile, qui n'est pas donnée à chacun ?*
>
> Socrate, dans *Alcibiade* de Platon

La réflexivité occupe une position centrale dans notre travail. C'est une notion très riche, commune aux philosophes et aux informaticiens, mais aussi aux linguistes et aux psychanalystes (on se souviendra par exemple du stade du miroir de Jacques Lacan). Cette diversité fonde la force d'inspiration de la réflexivité. Il nous a paru important pour cette raison d'en donner ici un aperçu le plus large possible, avant d'aborder dans le chapitre 3 ses applications à la tolérance aux fautes des systèmes informatiques.

Ce chapitre introduit la notion de réflexivité en illustrant sa place au quotidien, dans les langues humaines. Nous présentons ensuite comment la réflexivité a pu être utilisée en informatique, ce qui nous permettra de motiver son intérêt pour l'intégration modulaire d'aspects transversaux dans des architectures logicielles. Nous terminons le chapitre en présentant les caractéristiques de plusieurs plates-formes réflexives développées dans les milieux académiques et industriels.

2.1 La réflexivité au quotidien

Est réflexif un système capable d'appliquer à lui–même ses propres capacités d'action (selon le système considéré, cela peut-être des capacités de description, de calcul, de pensée). Par exemple, l'être humain, en sa qualité d'animal qui pense, est un « système réflexif », puisque l'être humain peut penser à lui-même. Dans cette section, nous utilisons un exemple issu de la vie quotidienne, le langage, pour introduire les caractéristiques fondamentales d'un système réflexif.

Toute langue humaine obéit à un ensemble de règles, *sa grammaire*, qui capturent de manière abstraite les structures qui la gouvernent. Or, il est tout à fait possible, et même très courant, de discuter de la grammaire d'une langue *dans cette langue elle même*. Une langue, *outil* de discours, devient alors l'*objet* de son propre discours (la grammaire). C'est de ce double rôle que surgit la réflexivité : un manuel de grammaire française pour écolier, écrit en français, est un texte en français qui parle de la langue française, c'est-à-dire un texte *réflexif*. Cet *auto–référencement* constitue la première caractéristique de la réflexivité. La grammaire constitue un *modèle* de la langue (que nous appellerons dans un contexte réflexif un *méta-modèle*), et ce modèle est accessible depuis l'intérieur de la langue.

La deuxième caractéristique de la réflexivité a trait aux liens qu'entretient une langue avec l'ensemble des règles qui la gouvernent, lorsque l'une ou l'autre évolue. L'évolution d'une langue est un phénomène naturel. Nous ne parlons plus le français de Molière, encore moins celui de Rutebeuf[1]. Cette évolution se répercute sur la grammaire : de nouvelles règles de grammaire sont introduites dans les manuels pour prendre en compte les changements constatés. *L'évolution de la langue entraîne celle des manuels*. Réciproquement, même s'il s'agit d'un cas beaucoup plus rare pour les langues, il arrive qu'une langue soit modifiée « par décret », l'évolution des règles précédant alors celle de la langue. La *Rechtschreibreform* des pays germanophones, réforme de l'orthographe rentrée en vigueur en 1998, constitue un des exemples les plus récents de ce mécanisme. De nouvelles règles d'orthographe de l'allemand ont été développées entre 1985 et 1995 et formellement acceptées par un traité inter–étatique en 1996. En 1999, quasiment tous les journaux de langue allemande avaient adopté la nouvelle orthographe. *Dans ce cas, l'évolution des règles d'orthographe, décidée par les états, a entraîné l'évolution de la langue*. Il existe donc *une double relation de causalité* entre une langue et les modèles qu'en construisent les grammairiens. Cette double relation de causalité constitue la deuxième caractéristique de la réflexivité.

La capacité d'une langue à être à la fois outil et objet de discours est à la base de la réflexivité. Cette capacité est intimement liée à la distinction, essentielle en linguistique moderne, entre *le sens d'un mot* et *le mot lui même*. Cette distinction a été formalisée au début du XX^{ème} par le linguiste suisse Ferdinand de Saussure à travers les notions de *signifiant* et de *signifié*. Le signifiant renvoie au mot (comme succession de lettres, de sons)

[1] Par exemple, les vers « *Que sont mes amis devenus [...] / Je crois le vent les a ôtés [...] / Ce sont amis que vent emporte / Et il ventait devant ma porte / Les emporta* », popularisés par Léo Ferré, donnent dans leur version originale : « *Que sont mi ami devenu [...] / Je cuit li vens les a osté, [...] / Ce sont ami que vens emporte, / Et il ventoit devant ma porte ; / Ses emporta* » .



FIG. 2.1 : *La réflexivité dans la littérature contemporaine [Goscinny & Uderzo 1965]*

qui dans une langue donnée (le français, l'allemand) désigne un concept (que l'on appelle alors le signifié). Ainsi « *mot* », « *Wort* », « *word* », « *palabra* » sont plusieurs signifiants dans des langues différentes pour le même signifié. Un signifiant *représente* un signifié ; il est l'élément concret (le son, le mot écrit) qui permet d'accéder au concept (le sens, la pensée).

Nous avons déjà mentionné la notion de « méta-modèle » pour parler des grammaires, c'est-à-dire d'une modélisation d'une langue à l'intérieur d'elle même. Un « méta-modèle » est de l'ordre du sens, du concept, du *signifié*. Une langue, pour être réflexive, doit donc donner accès à ce « méta-modèle » en offrant des *mots*, des éléments concrets du discours, des *signifiants*, qui permettent d'en parler. « *phrase* », « *verbe* », « *conjugaison* », « *accord* », « *déclinaison* » ou encore « *alexandrin* » (figure 2.1) sont des exemples de tels mots en français. Ces mots dessinent « l'empreinte » de la grammaire dans la langue, sont une sorte « d'interface » fournie par la langue pour donner accès à son méta-modèle. C'est ce que l'on appelle dans un contexte réflexif une *méta-interface*.

Pour résumer ce que nous venons de dire :

1. une langue contient des mots (signifiants) qui représentent des concepts (signifiés) ; les signifiants sont les *éléments concrets* de la langue qui *donnent accès* aux concepts ;
2. ces concepts sont associés à une modélisation particulière d'une réalité ;
3. la grammaire d'une langue modélise la langue, constitue un *méta-modèle* de cette langue ;
4. les mots d'une langue qui permettent de parler de sa grammaire forment une *méta-interface* qui donne accès *concrètement* au *méta-modèle* qu'est la grammaire.

2.2 La réflexivité en informatique

La notion de réflexivité était déjà présente dans les développements de logique théorique de l'entre-deux-guerres, qui allaient largement influencer le développement des premiers ordinateurs. On la retrouve notamment dans le théorème d'incomplétude de Kurt Gödel [Gödel 1931][2], ou encore dans la machine universelle d'Alan Turing (une machine de Turing capable d'émuler toute autre machine de Turing, et en particulier donc, de s'émuler elle-même) [Turing 1936].

De manière fort intéressante, l'une des toutes premières architectures proposées pour les ordinateurs, l'architecture de von Neumann, était justement par essence réflexive. Proposée en 1945 pour l'EDVAC, la machine qui devait succéder à l'ENIAC, l'un des premiers ordinateurs, l'architecture de von Neumann consiste à stocker les instructions d'un programme et les données qu'il manipule sur le même support physique.

Aujourd'hui, la plupart des ordinateurs sont toujours organisés selon des formes dérivées de ce principe. Les instructions d'un programme sont donc à la fois les outils du traitement, et potentiellement aussi des données pouvant être traitées. Cette caractéristique n'était cependant probablement pas l'un des objectifs principaux recherchés par John von Neumann, et pendant très longtemps, les programmes « auto–modifiants », s'ils étaient possibles, ont plutôt été considérés comme des sortes de « monstres » informatiques que comme des objets dignes d'intérêt.

C'est au fil du développement des langages évolués, notamment avec les recherches sur le langage de programmation LISP, que la réflexivité est peu à peu apparue comme une solution élégante à des problèmes récurrents d'adaptabilité et de réutilisation. La réflexivité « calculatoire » (*computational reflection*) d'un système informatique a ainsi été définie de la manière suivante :

> *« In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representation of its own operation and structures. »*
>
> *[Smith 1982] cité dans [Costa et al. 1998]*

Le grand intérêt de ce type d'approche est de permettre d'exprimer des traitements en termes extrêmement génériques, qui n'utilisent que les notions constitutives du mode de calcul du système, comme *« règle »* ou *« condition »* pour un moteur d'inférence de règles ; *« méthode »* , *« attribut »* , ou *« classe »* pour un programme orienté objet. Dans un système orienté objet, un traitement réflexif élémentaire (un *méta-traitement* ou mieux un *méta-programme*) serait par exemple :

> *« À chaque appel de méthode, imprimer le nom de la méthode. »*

[2] Le lecteur intéressé trouvera une brève discussion des aspects réflexifs du théorème d'incomplétude de Gödel dans l'Annexe A page 123.

Ce type de méta-programme est indépendant de ce que fait effectivement le reste du système, et peut s'appliquer qu'il s'agisse d'une application de facturation téléphonique ou de la commande de vol d'une fusée, à la condition *sine qua non* cependant que ces applications aient une structure orientée objet. C'est donc un programme complètement orthogonal aux aspects applicatifs du système, qui réalise une fonctionnalité transversale (ici le traçage). En résumé, la réflexivité en informatique est un moyen puissant de réaliser un découplage entre les aspects applicatifs et les aspects non-fonctionnels d'un système. Une fois réalisé, un méta-programme peut être réutilisé de manière transparente sur tout système qui utilise le modèle de programmation à la base du méta-programme (notre exemple de traçage ne pourrait pas par exemple s'appliquer à un système d'inférence de règles, la notion de méthode n'existant pas).

D'un point de vue extérieur, un système réflexif n'est pas plus puissant qu'un système qui ne l'est pas [Maes 1987]. Tous les problèmes que peut traiter un système réflexif peuvent aussi l'être par un système non–réflexif équivalent. La réflexivité, comme les langages structurés, comme l'orienté objet, est une manière d'organisation interne d'un système pour faciliter son développement, son adaptation, et sa réutilisation. Dans les paragraphes qui suivent nous détaillons un peu plus comment la réflexivité se réalise dans les langages interprétés, ce qui nous permettra ensuite d'étendre les concepts présentés à des éléments plus éloignés des langages de programmation, comme les intergiciels ou les systèmes d'exploitation.

2.2.1 La réflexivité des langages interprétés

Un langage interprété L est réflexif s'il permet d'inclure dans un programme P écrit dans ce langage du code modifiant la manière dont L est interprété. Cette approche apparaît relativement naturelle, si l'on remarque qu'un langage de programmation (en dehors du langage machine) n'a, tel–quel, aucune signification « calculatoire » pour un ordinateur. Le mini–programme LISP `'(+ 2 5)'` n'est pour un processeur que la concaténation des caractères '(', '+, ' ' *etc.*, et de même qu'un étudiant qui ne connaît pas le LISP ne peut comprendre sa signification, un ordinateur ne peut l'exécuter sans posséder un interpréteur adapté. Pour reprendre nos termes de linguistique, `'(+ 2 5)'` n'est qu'un *signifiant* et c'est la sémantique du langage LISP définie par John McCarthy en 1958 qui permet de lui faire correspondre une exécution (le *signifié*). Si John McCarthy n'avait pas inventé le LISP, nous pourrions toujours écrire la chaîne de symbole `'(+ 2 5)'`, mais elle n'aurait pas de signification. En changeant d'interpréteur, on change la sémantique du langage, et donc la signification du programme. Un programme qui modifie son interpréteur modifie sa propre signification, et se modifie donc lui-même.

Deux problèmes se posent dans cette approche :

1. Comment éviter une récursion infinie dans l'interprétation d'un programme P, si P contient son propre interpréteur ?
2. Quels éléments de l'interpréteur rendre manipulables par P ? Pour reprendre la terminologie employée précédemment : quel *méta-modèle* du langage L proposer aux

programmes écrits avec L ? par l'intermédiaire de quelle *méta-interface*, par quels éléments du langage L, donner accès à ce méta-modèle ?

Interpréteurs réflexifs et méta-circularité

Le premier problème se résout en utilisant ce qui a été appelé un *interpréteur méta-circulaire* [Maes 1987]. Un interpréteur méta-circulaire donne accès à une représentation (méta-modèle) de son propre processus d'interprétation, et permet, en modifiant cette représentation, de modifier son propre fonctionnement. Le fait de rendre visible le processus d'interprétation est appelé *réification*, par emprunt au vocabulaire philosophique, ce qui signifie rendre concret une entité abstraite. Cette « réification » du travail de l'interpréteur permet de distinguer dans un programme Q qui en fait usage des zones qui modifient l'interprétation par défaut ($Q_{réflexif}$), et des zones qui produisent des résultats pour l'environnement extérieur au programme (Q_{base}) : $Q = Q_{réflexif} \cup Q_{base}$. Si nous notons $I_{défaut}$ l'interpréteur par défaut du langage L, l'interprétation du programme réflexif Q s'opère « conceptuellement » en deux temps : $I_{défaut}$ interprète $Q_{réflexif}$, ce qui définit un nouvel interpréteur $I_1 = I_{défaut} \cup Q_{réflexif}$. C'est ce nouvel interpréteur I_1 qui est finalement utilisé pour interpréter[3] la partie non–réflexive du programme, Q_{base}.

En incluant dans L des mécanismes pour décider qui interprète qui, il est alors possible d'inclure récursivement plusieurs niveaux d'interprétation $Q^1_{réflexif}$, $Q^2_{réflexif}$, ... $Q^n_{réflexif}$ dans un programme. On obtient alors une « tour » d'interpréteurs. Cette tour se termine lorsque le programme à interpréter ne contient plus que des éléments non–réflexifs.

Choix du méta-modèle

Très souvent la partie réflexive d'un programme ($Q_{réflexif}$) ne redéfinit qu'une partie du processus d'interprétation défini par l'interpréteur par défaut (par exemple en rajoutant une fonctionnalité de traçage d'appels, ou en introduisant de nouveaux mots-clefs), et réutilise largement les fonctionnalités proposées par l'interpréteur original $I_{défaut}$. Dans une architecture réflexive, la structuration du méta-modèle exporté par l'interpréteur par défaut $I_{défaut}$ constitue donc un choix essentiel parce qu'elle détermine dans quelle mesure et de quelle manière la sémantique du langage pourra être modifiée et étendue par un programme.

Par exemple le langage JAVA [Joy et al. 2000] possède certaines capacités réflexives, mais ces capacités ne fournissent que des moyens d'observation de relativement haut niveau du processus d'interprétation du byte–code JAVA. La partie réflexive de JAVA met par exemple à disposition du programmeur des classes spécialisées telles que `Class` (les instances de `Class`, des objets donc, *représentant* les classes connues par la machine virtuelle JAVA), `Method`, ou `Constructor`, et des méthodes telles que

[3] En pratique, ces deux temps de l'interprétation s'entrelacent, l'interprétation de chacune des lignes de Q_{base} faisant intervenir $I_{défaut}$ et $Q_{réflexif}$.

```
Class.getConstructors()
Class.getMethods()
Method.invoke(..)
Constructor.newInstance(..).
```

Cependant il n'est pas possible dans JAVA de modifier l'interpréteur, en redéfinissant par exemple le mécanisme d'invocation.

Terminologie

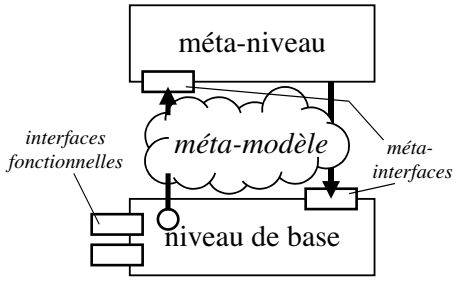


FIG. 2.2 : *Architecture d'un système réflexif*

Les parties réflexives et applicatives d'un programme réflexif permettent de distinguer deux niveaux de fonctionnement dans ce programme. On parle de *méta-calcul* pour désigner l'activité du système lorsque celui-ci interprète les parties réflexives du programme (ces parties manipulent en effet des éléments qui représentent eux-mêmes un mode d'interprétation). Les zones du programme qui définissent ce *méta-calcul* constituent ce que l'on nomme le *méta-niveau* d'un programme réflexif. Par symétrie, on désigne la partie applicative (Q_{base}) comme étant le *niveau de base* de Q. Cette structuration est illustrée sur la figure 2.2. Le méta-modèle, qui rend accessible le processus d'interprétation du niveau de base, est le « connecteur conceptuel » qui permet au méta-niveau de modifier ce processus. Ce connecteur se concrétise pour le méta-niveau par une série de *méta-interfaces* qui lui fournissent des capacités d'observation et d'action sur les activités du niveau de base.

Les capacités d'observation utilisées par le méta-niveau sont classifiées en deux catégories. Lorsque le niveau de base notifie spontanément le méta-niveau d'une évolution sous la forme d'un évènement (l'invocation d'une méthode, la déclaration d'une nouvelle variable, l'instanciation d'un objet), l'on parle de *réification*. Ces notifications matérialisent pour le méta-niveau le processus d'interprétation qui a lieu au niveau de base. Lorsque le méta-niveau requiert explicitement une information (quelles variables sont connues, quels objets ont été instanciés, quelle ligne de programme est en train d'être interprétée...), l'on parle d'*introspection*. Enfin, on parle d'*intercession* pour les capacités d'action qui permettent au méta-niveau de modifier le processus d'interprétation du niveau de base. Sur la

figure 2.2 page précédente, ces différents modes d'interaction entre niveau de base et méta-niveau sont représentés par des flèches montantes et descendantes. La granularité de ces différentes *capacités réflexives* (réification, introspection, intercession) dépend du choix du méta-modèle, et détermine, comme nous l'avons illustré sur l'exemple JAVA, la nature plus ou moins réflexive du langage considéré.

Cas particulier de l'orienté objet : les MOP

L'orienté objet permettant de structurer les logiciels de manière particulièrement flexible et modulaire, l'utilisation conjointe des paradigmes orienté objet et réflexif a très tôt reçue un intérêt marqué [Maes 1987, Kiczales et al. 1991]. L'ensemble du système étant orienté objet, le méta-niveau d'un système à la fois orienté objet et réflexif est lui aussi structuré sous forme d'objets particuliers, appelés *méta-objets*, qui encapsulent les aspects réflexifs du système. Ces *méta-objets* interagissent avec le niveau de base, organisé en *objets de base*, en utilisant les mécanismes de réification, introspection, et d'intercession que nous avons mentionnés. Du fait de la structuration en termes d'objets, on parle alors de *protocole à méta-objets* (*MOP* en anglais) pour désigner l'ensemble de conventions qui régissent les interactions entre *méta-objets* et *objets de base*, notamment en terme d'association (comment les méta-objets sont-ils associés aux objets de base ?), et de cycles de vie (comment les méta-objets sont ils créés ? détruits ? comment leur cycle de vie interagit-il avec celui des objets de base ?). Cette notion de MOP est à la base de la plupart des plates-formes réflexives que nous présentons dans la section 2.2.3.

2.2.2 La réflexivité des langages compilés et des substrats d'exécution

Les notions que nous avons présentées précédemment dans le cadre des langages interprétés peuvent s'étendre à d'autres systèmes informatiques tels que les langages compilés, ou les substrats d'exécution (réunissant systèmes d'exploitation, intergiciels, bibliothèques système *etc.*).

Les langages compilés se distinguent des langages interprétés du fait du découplage entre leur traduction en instructions du langage machine (appelée alors compilation) et leur exécution. Cependant le parallèle entre la relation programme–exécution et signifiant/signifié reste valable. L'exécutable obtenu lors d'une compilation est une sorte d'exécution « en attente ». La réflexivité du processus de traduction peut donc à la fois

- soit s'opérer au moment de la compilation, en permettant à un programme de modifier le compilateur qui le traduit ;
- soit au moment de l'exécution en incluant dans un programme des éléments modifiant les processus dynamiques de l'exécution.

La réflexivité à la compilation suppose l'utilisation d'un compilateur réflexif, appelé aussi compilateur ouvert. OPENC++ [Chiba 1995] ou OPENJAVA [Tatsubori et al. 2000] en sont des exemples. Les éléments du processus de compilation qui sont rendus visibles (tels que la

traduction d'une nouvelle classe, la traduction d'un appel de méthode, ou la traduction de la modification d'un attribut) déterminent le méta-modèle d'un tel compilateur. Les interfaces qui permettent d'interagir avec et de modifier le comportement par défaut du compilateur en constituent les méta-interfaces.

La distinction entre langages compilés et langages interprétés n'est cependant pas aussi stricte qu'il y paraît. La plupart des langages compilés incluent dans leur binaire des informations sur la structure et l'organisation de leur code source. Parce qu'elles sont de nature réflexive (ce sont des informations qui explicitent dans l'exécutable en langage machine la structure originelle du programme, donc des informations dans le programme sur lui-même), on appelle souvent ces informations des *méta-données*. Historiquement, ces méta-données furent introduites pour permettre l'utilisation de bibliothèques logicielles (une bibliothèque compilée doit au moins contenir des informations sur les symboles qu'elle exporte, et qui forment les points d'entrées d'un utilisateur dans la bibliothèque), et réaliser des débogueurs. Ces capacités ont été étendues pour permettre le chargement dynamique de code compilé comme en JAVA [Joy et al. 2000] (c'est-à-dire l'utilisation dans un programme en train de s'exécuter de code compilé après le lancement du programme), puis pour fournir à l'intérieur d'un programme en train de s'exécuter des informations sur sa propre structure (ce que fait JAVA, voir plus haut). Ainsi certaines opérations dynamiques (« dynamique » signifiant qui se réalise au moment de l'exécution par opposition à la compilation), comme l'appel de méthodes, l'allocation mémoire de structures dynamiques (notamment les objets en orienté objets), la résolution de symboles (pour les bibliothèques partagées), la résolution du polymorphisme (pour les langages supportant l'héritage de types), sont autant d'aspects liés à la sémantique d'un programme compilé qui interviennent au moment de son exécution. En permettant à un programme de modifier dans son code source ces mécanismes dynamiques, on y introduit un caractère réflexif qui s'exprime à l'exécution. Là–encore ce type de réflexivité suppose une chaîne d'exécution « compilateur + substrat d'exécution » adaptée. Le substrat d'exécution joue ici un rôle dans la mesure où certaines de ces fonctionnalités (comme par exemple celles relatives à la gestion mémoire) sont réalisées par des bibliothèques système spécifiques. Le rôle du compilateur se limite alors à inclure des points d'appel dans le binaire qu'il produit vers la bibliothèque concernée. Par exemple, dans un programme C++ compilé avec le compilateur du projet GNU g++ version 2.95, l'instanciation de nouveaux objets utilise une méthode « cachée » (`__builtin_new` de la bibliothèque partagée `libstdc++-libc6`).

Les deux types de réflexivité des langages compilés, réflexivité à la compilation et réflexivité à l'exécution, sont liés, la première permettant d'implémenter la seconde. C'est par exemple ce que réalise le système de tolérance aux fautes FRIENDS V.2 [Killijian & Fabre 2000] en utilisant le compilateur réflexif OPENC++ pour ajouter des capacités réflexives à un bus à objets CORBA.

Cette remarque nous amène naturellement à considérer la réflexivité des substrats d'exécution (ou des plates-formes d'exécution). Celle-ci nous semblera d'autant plus naturelle si nous remarquons qu'un substrat d'exécution entretient avec les programmes dont il permet l'exécution des relations similaires à celles qu'entretient un interpréteur avec les

programmes qu'il interprète : il leur fournit des primitives (de synchronisation, de communication, de gestion des ressources système), ces primitives ne prenant sens qu'au moment où elles sont invoquées.

2.2.3 Exemple de plates-formes réflexives

Cette similitude entre interprétation et résolution de symboles peut permettre de dire que c'est le substrat d'exécution qui donne (au moins partiellement) son sens à un programme compilé. En offrant à un programme une interface réflexive lui permettant de modifier les traitements de son substrat, on le rend réflexif. Parmi les pionniers de cette approche on peut citer le micro–noyaux MACH et ses paginateurs mémoire externes [Young et al. 1987], qui permettent à des applications spécifiques (systèmes transactionnels, serveurs de données) de spécifier leurs propres politiques de gestion de la mémoire.

Comme pour les langages interprétés, le choix des aspects à rendre visibles de l'extérieur est essentiel dans la conception d'un substrat d'exploitation réflexif. Dans [Kiczales & Lamping 1993] Kiczales et Lamping identifient quatre propriétés à considérer dans la réalisation d'un tel système :

Incrémentalité
: Les possibilités de modification du substrat doivent être progressives. Un changement limité d'un aspect du système d'exploitation doit se traduire en un méta-programme de taille proportionnelle.

Contrôle d'impact
: *(Scope Control)*. Il doit être possible de restreindre la zone d'impact des méta-programmes à certaines entités du niveau de base, et pas à d'autres.

Interopérabilité
: Les entités du niveau de base qui utilisent des aspects modifiés du système d'exploitation doivent pouvoir continuer à interopérer avec les autres entités du niveau de base.

Robustesse
: L'architecture globale se doit d'être robuste aux fautes impactant aussi bien les programmes du niveau de base que les méta-programmes. Elle doit par exemple prévoir des mécanismes de détection et de confinement d'erreur, de dégradation progressive et contrôlée, *etc.*

Dans la suite de cette section, nous présentons plusieurs projets d'OS ou d'intergiciels réflexifs, et nous discutons quels ont été les choix qui ont dirigé leur conception. Les projets choisis ici ne ciblent pas spécifiquement la tolérance aux fautes (nous revenons sur l'utilisation de la réflexivité pour la tolérance aux fautes dans le chapitre 3, p. 35).Ces projets cherchent à être particulièrement adaptables en termes de taille, de performance, de fonctionnalités, d'extensions, pour pouvoir être utilisés dans des contextes opératoires fortement contraints (en termes de ressources énergétiques [informatique nomade, systèmes autonomes], d'empreinte mémoire [informatique embarquée grand publique, systèmes enfouis], de performance temps-réel [multimédia, télécommunications, robots], *etc.*).

MUSE

Développé chez Sony, MUSE [Yokote et al. 1989] est l'ancêtre de l'OS temps réel APERTOS/ APERIOS [Yokote 1992] maintenant utilisé dans des produits commerciaux comme le chien robot AIBO. MUSE est un système d'exploitation orienté objet réflexif. Il est organisé selon deux types de hiérarchies : des *hiérarchies d'héritage*, classiques en orienté objets, et des *méta-hiérarchies*, qui relient chaque objet s'exécutant au-dessus du système d'exploitation à l'objet qui l'interprète (alors appelé son *méta-objet*). Chaque méta-objet est une sorte de mini–machine virtuelle, spécifique à un objet ou à un groupe d'objets (on retrouve ici la notion de *contrôle d'impact* présentée plus haut). Cette méta-hiérarchie s'arrête après un niveau de récursion, les *méta-objets* de MUSE étant interprétés par un *méta-méta-objet* générique formant une couche d'abstraction logicielle au-dessus du matériel. MUSE résout donc de manière très pragmatique le paradoxe de la méta-circularité en limitant la tour d'interpréteurs à deux niveaux.

Les exo-noyaux, AEGIS et THINK

Les exo-noyaux ne sont pas à proprement parler des systèmes d'exploitation, mais plutôt des canevas de construction pour OS. AEGIS [Engler et al. 1995], développé au MIT, peut sans doute être considéré comme le premier exo–noyau à avoir été présenté comme tel. L'objectif de ce système était de n'imposer aux applications aucune primitive système particulière, en leur permettant de construire leur propre OS, « taillé » sur mesure, à partir de services de base (essentiellement de protection du matériel, et d'exclusion mutuelle dans l'utilisation des ressources) réduits à leur plus simple expression. La même idée a été reprise et poussée plus loin dans THINK [Fassino et al. 2002] en simplifiant encore la couche de l'exo–noyau et en incluant dès le plus bas niveau les concepts orientés objet du standard RM-ODP (*Reference Model for Open Distributed Processing*) [Raymond 1995, ISO 1995], notamment les notions de *liaison* et d'*espace de nommage*. Les deux approches ont été validées par l'implémentation de systèmes d'exploitation complets. Parce qu'ils sont extrêmement modulaires (surtout dans le cas de THINK), les canevas exo–noyaux se prêtent très bien à la réalisation de substrats d'exécution réflexifs, et peuvent être considérés comme un avatar extrême de la réflexivité dans les exécutifs : le système d'exploitation est, non pas modifiable, mais « à construire » par l'application qui l'utilise.

Les intergiciels réflexifs

OPENORB

OPENORB est un ORB réflexif développé à l'Université de Lancaster [Blair et al. 1998, Kon et al. 2002]. Comme THINK après lui (voir plus haut), OPENORB utilise un modèle de programmation orienté objet directement inspiré du standard RM-ODP (voir plus haut). Les notions d'objet et d'interface sont distinctes, un objet pouvant posséder plusieurs in-

terfaces, organisées selon des hiérarchies de types[4]. OPENORB prend en charge plusieurs types d'interfaces : interfaces orientées flux (*stream*, par exemple multimédia), orientées évènement (*signal*), et orientées opération (appel de méthode). Deux interfaces compatibles peuvent être reliées par l'intermédiaire d'un objet de liaison (*binding*) explicite. Un objet (et donc, cas particulier, aussi une liaison) peut être composite. Il contient alors d'autres objets liés entre eux par des liaisons, dont l'objet « contenant » contrôle les interactions avec l'extérieur.

En tirant parti de ce modèle à objets particulièrement riche, le méta-niveau d'OPENORB est organisé selon deux principes :

1. Le méta-niveau est structuré en trois méta-espaces, qui chacun représente différents aspects du système :
   (a) Le méta-espace de *composition*, traite de la structure des objets composites. Cette structure est rendue accessible à travers un graphe représentant l'organisation des composants internes d'un objet. Ce graphe est causalement connecté à l'objet composite. Modifier le graphe modifie l'objet et réciproquement.
   (b) Celui d'*encapsulation*, permet d'observer et de manipuler les interfaces d'un objet. Dans l'implémentation originelle d'OPENORB (en PYTHON), il est possible d'inspecter les méthodes proposées par une interface, d'ajouter et de retirer des méthodes, d'insérer des pré– et post–traitements à une méthode.
   (c) Celui d'*environnement* donne accès à l'environnement d'exécution, en termes de ressources, de gestion de la concurrence, gestion des queues de message, *etc.*
2. Ces méta-espaces sont représentés par des méta-objets qui sont associés un à un à chaque objet et à chaque interface :
   (a) Chaque interface possède ses propres méta-objets d'*encapsulation* et d'*environnement*.
   (b) Chaque objet possède son propre méta-objet de *composition*.

Cette structure est fortement récursive : les méta-objets étant des objets, ils peuvent à leur tour être introspectés et modifiés par l'intermédiaire de méta-méta-objets. Pour éviter une récursion infinie, les méta-objets ne sont instanciés que sur demande par le système. La récursivité de la réflexivité permet d'enrichir progressivement le méta-niveau avec de nouvelles fonctionnalités en termes de transparence et de contrôle, si jugé nécessaire. Cette possibilité permet d'offrir dans différents contextes différents niveaux de réflexivité et d'ouverture, compatibles entre eux. La structuration en différents méta-espaces, la prise en charge explicite des notions de composition et d'encapsulation (contrôle des services importés et exportés) et l'utilisation d'objets de liaison explicites sont parmi les points forts de l'approche d'OPENORB.

[4]Dans OPENORB les interfaces sont des sortes de « points de service » typés qui présentent « une vue » particulière d'un objet sous la forme d'un groupe de méthodes. Une interface *appartient* à un objet donné, et à lui seul. Il existe ainsi usuellement plusieurs interfaces ayant le même type dans le système (chacune étant attachée à leur objet respectif). D'un point de vue purement terminologique, cette utilisation du mot interface peut hélas prêter à confusion, la notion d'interface renvoyant dans certains langages comme JAVA à la notion de type.

JONATHAN

JONATHAN [Dumant et al. 1999] a été développé par FRANCE TÉLÉCOM dans la perspective des grandes infrastructures ouvertes (télécommunication par exemple) pour lesquels la liste des applications à supporter n'est pas connue au moment du déploiement de la plate-forme. De nouvelles applications utilisant des modes d'interaction non prévus à l'origine doivent pouvoir être intégrées progressivement. La plate-forme doit par exemple permettre, là où c'est nécessaire, un contrôle fin des interactions en termes de bande passante, de temps processeur, de qualité de service, tout en prenant aussi en charge des styles de communication moins riches, mais plus répandus comme CORBA ou JAVA RMI.

Pour répondre à cette problématique, JONATHAN permet de définir des *personnalités* multiples pour faire coexister dans un même environnement des formes d'interactions hétérogènes. Par exemple, des appels distants CORBA et JAVA RMI peuvent cohabiter à l'intérieur de JONATHAN avec des flux multimédia continus (*streaming*). Comme OPENORB ou THINK, JONATHAN suit RM-ODP, et utilise la notion d'*interface* pour représenter un point d'accès aux services proposés par un objet distribué. JONATHAN est architecturé autour d'un ORB minimal, en décomposant le mécanisme de « liage » dans les ORB en trois aspects :

Référencement
: Comment désigner de manière univoque une interface dans une infrastructure répartie ?

Accès
: Comment organiser la création de liaisons de manière modulaire et extensible ?

Typage
: Comment vérifier la compatibilité des interfaces entre elles avec les différents types de liaisons possibles ? Comme permettre l'extension du système de typage ?

JONATHAN se base sur la notion d'*objets de liaison* d'ODP pour représenter explicitement les mécanismes de « liage » d'une personnalité donnée. Ces objets de liaison sont créés par des *usines à liaisons* spécifiques à une personnalité.

Pour résoudre le problème du référencement distribué JONATHAN utilise un système de nommage hiérarchique et extensible : les objets résident au sein de *capsules* qui représentent un environnement d'exécution local. Chaque capsule est responsable du nommage en local de chacune des interfaces qu'elle crée, et peut utiliser son système propre, totalement indépendamment des autres capsules du système. Par exemple, une machine virtuelle JAVA utilise l'adresse mémoire des objets qu'elle contient pour dénommer ces objets (JAVA ne distingue pas les objets de leurs « interface–point d'accès » comme ODP). Lorsqu'elles sont exportées en dehors d'une capsule, ces références locales sont encapsulées dans des structures appelées *surrogates*, qui contiennent des informations de liaison suffisantes pour créer un objet de liaison avec la capsule originelle selon une ou des personnalités données.

Les *surrogates* permettent de clairement séparer le problème du *référencement* de celui de l'*accès*. Un objet qui reçoit un *surrogate* peut le diffuser à d'autres objets même s'il n'est pas capable d'accéder à l'interface désignée par le *surrogate*, par exemple parce qu'aucune usine à liaisons correspondant aux informations du *surrogate* n'est accessible. L'accès n'a

lieu qu'au moment où les informations de liaison du *surrogate* sont utilisées dans une usine à liaisons appropriée pour créer un objet de liaison correspondant.

Dans la personnalité CORBA par exemple, les *scions* (*skeleton*) des objets serveurs correspondent aux « interfaces–point d'accès » de JONATHAN. Ces squelettes sont désignés par des références de type `CORBA::Object`, qui jouent alors le rôle de *surrogate*. Les souches (*stub*) créées côté client sont les objets de liaison de la personnalité CORBA.

Le troisième et dernier aspect, le *typage*, est abordé en permettant à chaque personnalité de définir son propre système de types. Ce système permet de typer à l'intérieur de la personnalité à la fois les objets de liaison, et les interfaces des objets. Pour permettre à une même interface d'être utilisée dans différentes personnalités, JONATHAN spécifie un système de types de référence, pour lequel chaque personnalité est tenue de fournir une correspondance avec son propre système de types.

DYNAMICTAO

DYNAMICTAO [Kon et al. 2000] est un bus à objets CORBA réflexif développé à partir de l'ORB TAO [Schmidt & Cleeland 1999]. Le projet utilise une approche par « composantisation » (*componentazing*) de certains des aspects d'implémentation de l'ORB. DYNAMICTAO tire partie du patron de conception de « stratégie » (*strategy design pattern* [Gamma et al. 1995]) utilisé par TAO pour introduire des points d'attache prédéterminés (*hooks*) auxquels des stratégies spécifiques peuvent être liées dynamiquement. Ces stratégies sont classifiées en « catégories » qui chacune correspondent à un aspect particulier de l'ORB tels que la gestion du parallélisme, la sécurité, le diagnostic en ligne, *etc.*

Les stratégies sont gérées sous forme de composants binaires au travers de *Configurators* qui prennent en charge les dépendances entre les différentes stratégies au sein du système. Ces *Configurators* forment la méta-interface de DYNAMICTAO et permettent par exemple de charger/décharger, d'activer/de désactiver une stratégie donnée, d'aider à la transition dynamique d'une stratégie vers une autre, ou encore de donner des informations sur les stratégies utilisées à un moment donné. La transition d'une stratégie donnée à une autre, par exemple d'un mode de concurrence à brins d'exécution multiples à un mode séquentiel (à un seul brin), doit à chaque fois être élaborée en prenant en compte la sémantique des stratégies impliquées pour éviter tout risque d'incohérence.

OPENCORBA

OPENCORBA [Ledoux 1999] est une implémentation de CORBA en NEOCLASSTALK, un dialecte de SMALLTALK. OPENCORBA utilise massivement les possibilités réflexives de NEOCLASSTALK, comme de pouvoir transmettre le corps d'une fonction en paramètres, ce qui autorise une réalisation aisée et concise de mécanismes de déroutement d'appels.

Ces facilités réflexives sont utilisées dans OPENCORBA pour rendre accessibles certains des mécanismes internes de l'ORB à travers des *méta-classes*. Ces *méta-classes* correspondent

à un peu près à la notion de méta-objets que nous avons déjà introduite. Elles permettent de rendre modulaires et adaptables dynamiquement les mécanismes d'invocation à distance, la vérification de type lors du traitement d'appels distants, ou encore la gestion d'exceptions lors de l'enregistrement de nouvelles interfaces dans le gestionnaire dynamique d'interfaces (*Dynamic Interface Repository*, un élément du standard CORBA).

2.3 Conclusion du chapitre

Nous venons de voir comment la réflexivité permettait de modifier dans un logiciel les éléments fondamentaux de son « mode de calcul », de son « modèle de programmation ». Ces modifications sont orthogonales à la description de ce que *fait* le système d'un point de vue applicatif, et permettent de découpler les aspects applicatifs d'un logiciel de ses aspects transversaux. Cette approche offre un outil élégant et puissant pour intégrer des aspects non–fonctionnels sous la forme de composants modulaires dans les architectures logicielles.

La réflexivité est maintenant bien ancrée dans la culture informatique, et, de par ses avantages, a su se frayer un chemin, au moins partiellement, dans certains des langages de programmation les plus populaires actuellement. Elle est à l'origine de développements tels que la programmation orientée aspects [Kiczales et al. 1997], qui intègre dans des constructions linguistiques particulièrement faciles à utiliser certains de ses principes de découplage.

La réflexivité a déjà été utilisée pour réaliser des architectures logicielles fortement adaptables, notamment des systèmes d'exploitation et des intergiciels réflexifs. Elle apparaît donc comme une solution très intéressante pour la réalisation de la tolérance aux fautes des systèmes logiciels complexes dont nous avons présenté la problématique au chapitre 1. Dans le chapitre suivant, nous présentons certaines des approches proposées pour inclure la tolérance aux fautes de manière générique dans des plates-formes logicielles, avec et sans réflexivité. Nous analyserons les limites de ces approches pour les logiciels multi-couches contenant de nombreux composants hétérogènes, et expliqueront pourquoi de nouvelles stratégies réflexives doivent être développées.

Chapitre 3

Les plates-formes pour la tolérance aux fautes

Nous nous penchons dans ce chapitre sur les travaux déjà entrepris sur les architectures génériques tolérantes aux fautes. Cette présentation nous permettra de comprendre comment la réflexivité a pu être utilisée pour la réalisation de mécanismes de tolérance aux fautes. Elle nous amènera aussi à discuter des limitations qui empêchent aujourd'hui le déploiement d'architectures réflexives dans les systèmes informatiques complexes et hétérogènes, et ainsi à poser la problématique à laquelle ce travail de thèse s'est attaché à répondre.

3.1 Pourquoi des plates-formes tolérantes aux fautes ?

Les mécanismes de tolérance aux fautes que nous avons présentés dans le chapitre 1 s'expriment pour la plupart en termes génériques, indépendants de la nature de l'application à laquelle ils s'appliquent (avec une exception notable de la *reprise avant*, très dépendante du domaine d'application). Ainsi des algorithmes de capture d'état distribué comme ceux présentés dans [Koo & Toueg 1987] ou [Chandy & Lamport 1985] sont décrits en termes de *processus*, *messages*, *canaux*, *machines à états finie*, *transitions*, *etc.* Dès lors que les éléments d'une application concrète qui correspondent à ces concepts abstraits ont été identifiés et rendus observables et contrôlables, il est possible de factoriser la tolérance aux fautes dans des briques logicielles réutilisables. Selon la nature et la richesse des « briques » fournies, on pourra parler de « kit de développement », de canevas (*framework*), ou de plate–forme. Pour ne pas alourdir la présentation, nous parlerons de « plate–forme » dans la suite de notre discussion, en gardant en mémoire le caractère plus ou moins étendu que peut revêtir cette notion.

3.1.1 Aspects de la tolérance aux fautes

Les différents aspects abordés par les plates-formes de tolérance aux fautes peuvent être approximativement classifiés de la manière suivante :

Détection d'erreur
: La détection d'erreurs consiste à identifier au plus tôt des situations anormales de fonctionnement pour pouvoir y réagir de manière adaptée. Ce type de fonctionnalité existe au moins de manière minimale dans la plupart des substrats d'exécution sous la forme de *codes de retour d'erreur* ou d'*exceptions*. Des approches plus élaborées ont été proposées, notamment à base d'architectures réflexives, pour faciliter le durcissement de systèmes. Nous en dirons quelques mots dans la section 3.2.3 page 50.

Synchronisation
: Comme nous l'avons expliqué au chapitre 1, distribution et tolérance aux fautes sont très souvent associées, la réplication d'un même service de manière distribuée permettant d'assurer l'indépendance de certains de ses modes de défaillance (quels modes sont rendus indépendants dépend de la nature de la distribution). Lorsque les répliques distribuées ne sont pas explicitement synchronisées entre elles (comme par exemple par un protocole synchrone du type TTP [Kopetz & Grünsteidl 1994]), des protocoles particuliers de synchronisation (de consensus, de multi–diffusion) doivent être mis en place pour assurer l'ordonnancement cohérent des interactions entre répliques.

Capture et transfert d'état
: Dans une architecture répliquée, il est souvent nécessaire de transférer l'état d'une réplique à une autre, par exemple lors d'une reconfiguration après une défaillance, ou dans certains types de réplication. La capture de l'état d'une réplique présente alors un aspect *données* (quels attributs, quels objets sont gérés par l'application) et un aspect *contrôle* (quels traitements sont exécutés). Lorsqu'une application revient régulièrement dans des états « d'attente » dans lesquels aucun traitement n'est en cours (c'est le cas par exemple d'une application client/serveur qui sérialise le traitement des requêtes qu'elle reçoit), il est possible de limiter l'état capturé à sa partie donnée, en ne prenant de capture qu'entre deux traitements. Lorsqu'en revanche une application parallélise le traitement des requêtes reçues, il n'est plus possible d'ignorer la partie *contrôle* de son état[1]. Il est alors nécessaire de sauvegarder l'état d'avancements des traitements internes en même temps que les données traitées.

Ces trois aspects ne sont pas complètement indépendants, et doivent être combinés en pratique pour réaliser une application tolérant les fautes. La détection de la défaillance d'une réplique distante par ses co–répliques est par exemple autant un problème de détection d'erreur que de synchronisation distribuée.

[1] Les instants sans traitement peuvent d'une part ne plus se produire du tout dans le cas d'une charge élevée. D'autre part, même lorsqu'ils se produisent (spontanément ou de manière provoquée en bloquant les requêtes entrantes), leur régularité n'est alors plus assez fine pour empêcher l'effet « domino », dont nous parlons quelques lignes plus loin.

Le choix des instants propices aux captures d'état (*checkpoint*) dans un système distribué est un autre exemple où le problème de la capture d'état se combine à celui de la synchronisation. Nous le discutons ici plus en détail. La capture de l'état d'un système (*checkpointing*) consiste à obtenir suffisamment d'informations sur ce système (entrées en attente, processus internes, ressources utilisées, valeurs des données traitées, *etc.*) à un instant t de son fonctionnement pour pouvoir reprendre ultérieurement l'exécution du système de manière cohérente avec le passé déjà observé. Considérons par exemple la figure 3.1. Un état s est capturé sur une réplique **X** après que le comportement A a été observé. Le système est ensuite redémarré sur une réplique **Y** en utilisant cet état s. C est le comportement observé après la reprise. Pour exclure toute incohérence du comportement observé, la séquence de comportement $A+C$ doit être un comportement attendu du système. Si B est le comportement *qui aurait été observé* si la réplique **X** avait poursuivi son exécution après la capture de s, cet impératif de cohérence n'implique pas que C doive être équivalent à B. Dit de manière figurée, s doit contenir la promesse d'un futur compatible (ici C) avec le passé déjà observé (A). Mais s n'est pas obligé de contraindre aux seuls futurs observables sans transfert / reprise (B).

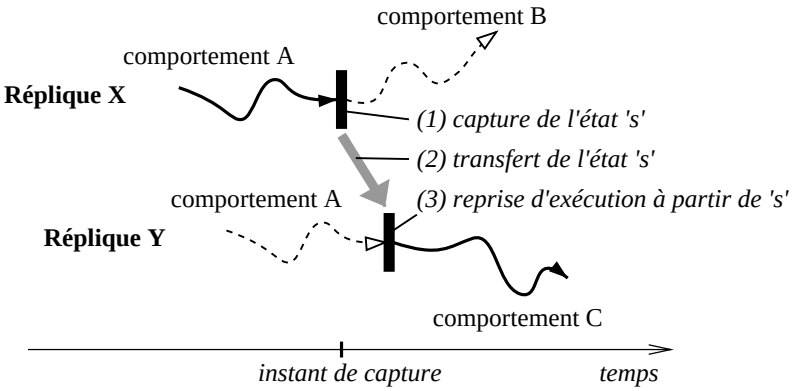


FIG. 3.1 : *Capture d'état : $A + B$ et $A + C$ doivent être des comportements valides*

On notera dans ces quelques remarques l'importance de la position de l'observateur des différents comportements. C'est ce qu'attend l'observateur du système qui détermine si $A+B$ est un comportement acceptable, et donc si l'état s contient suffisamment d'information pour une reprise cohérente. L'idée qu'a un observateur du système d'un comportement cohérent est aussi intimement lié à ses capacités d'observation. Si celui-ci est « aveugle » à certains aspects du comportement du système, ceux-ci n'influenceront pas sa perception de la cohérence, et pourront donc être ignorés dans s. Nous reviendrons plus en détail sur ce lien entre cohérence et niveau d'abstraction de l'observateur dans le chapitre 5 page 75.

Lorsque le système considéré est un système réparti, la capture d'état de chacune des entités distribuées du système (souvent alors appelées *nœuds*) revêt un degré supplémentaire de complexité parce qu'elle doit assurer la cohérence globale du système, en plus de la cohérence locale que nous venons de présenter. Il n'est pas acceptable, par exemple, de redémarrer un nœud dans un état qui *annule* pour ce nœud certaines des interactions qu'il

a pu avoir avec d'autres nœuds, si les effets de ces interactions continuent d'exister pour le reste du système. C'est le cas par exemple lorsque des messages ont été reçus d'un expéditeur qui considère ne les avoir jamais envoyés. Sur la figure 3.2, par exemple, deux processus distribués échangent les messages m_1 et m_2, tout en capturant leurs états aux points a, b, c et d. Du fait du message m_1 l'état global (a, c) n'est pas un état cohérent. Dans cet état, le processus P_1 « considère » avoir reçu le message m_1, alors que P_2 « considère » ne pas l'avoir envoyé, ce qui ne peut pas se produire dans une exécution valide du système. L'état (b, d) en revanche est cohérent : m_1 a été envoyé et reçu ; m_2 a été envoyé, sans avoir encore été reçu (m_2 est alors dit *en transit*). La présence de messages en transit est inhérente au fonctionnement normal d'un système distribué, et ne remet donc pas en cause la cohérence de l'état (b, d).

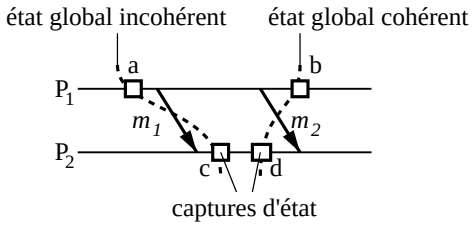


FIG. 3.2 : *Cohérence d'un état distribué*

Plus généralement, un état global distribué est cohérent si c'est un état qui *peut* se produire au cours d'une exécution sans défaillance du système. Ce problème de cohérence des états distribués a très tôt été reconnu comme un point essentiel de la tolérance aux fautes des systèmes distribués [Chandy & Lamport 1985, Koo & Toueg 1987]. En effet, lorsque des processus répartis ne coordonnent pas leurs captures d'état, il peut arriver qu'aucune combinaison des captures locales ainsi obtenues ne permette de reconstruire un état global cohérent. Dans ce cas, la défaillance d'un seul processus entraîne le redémarrage, dans son état initial, du système tout entier . Ce phénomène est appelé « effet domino » [Randell 1975]. Éviter l'effet domino nécessite de coordonner (explicitement ou implicitement) les différentes opérations de capture et de reprise, et touche donc aux problématiques de synchronisation. Il existe de nombreux protocoles de capture d'état distribué (*checkpointing protocols*) qui abordent ce problème [Elnozahy et al. 2002, Baldoni et al. 1997, Netzer & Xu 1995, Strom et al. 1988, Johnson & Zwaenepoel 1987]. Nous aurons l'occasion d'en dire un peu plus dans le chapitre 4 page 53 au sujet de la robustesse des algorithmes aux approximations de leurs entrées, et dans le chapitre 5 page 75 lorsque nous aborderons la problématique du déterminisme dans le mécanisme de réplication.

3.1.2 Propriétés des plates-formes

Du fait de leur grande complexité, les trois aspects présentés dans la section 3.1.1 page 36 ont tout d'abord été abordés indépendamment les uns des autres, une plate-forme s'intéressant plus spécifiquement à la synchronisation, une autre à la capture

d'état. Au fur et à mesure des travaux de recherche, les propriétés tenant à la *transparence* et à la *flexibilité* des plates-formes proposées sont apparues comme essentielles pour permettre une mise en pratique aisée de ces mécanismes.

Dans le contexte de la tolérance aux fautes, la notion de transparence renvoie à la possibilité pour un développeur d'application et un développeur de mécanismes de tolérance aux fautes de travailler indépendamment l'un de l'autre. Par exemple, la problématique de l'obtention de l'état peut-elle être abordée de manière orthogonale au développement de l'application ? Idéalement, le développeur d'application doit pouvoir tout ignorer de la problématique de la tolérance aux fautes (ici la capture d'état), qui lui est donc « transparente ». De même, le développeur de tolérance aux fautes doit pouvoir concevoir ses mécanismes sans avoir à assimiler la logique applicative du système auquel ils s'appliqueront. La transparence est donc intimement liée à la notion de *découplage* (pouvoir ajouter la tolérance aux fautes indépendamment du développement de l'application voire après coup), et de *séparation des préoccupations*.

La flexibilité désigne la facilité à adapter des mécanismes de tolérance aux fautes à de nouveaux besoins, à un nouveau contexte d'exécution, à permettre dans sa forme la plus avancée leur modification durant l'exécution du système. Par exemple, pour un mécanisme de capture d'état, dans quel format l'état est-il sauvegardé ? le système peut-il être redémarré dans un contexte d'exécution différent de celui de la prise d'état ? sur un nouvel OS ? dans une implémentation écrite dans un nouveau langage ?

Notons que ces problèmes, qui englobent d'autres aspects tout aussi importants comme la *réutilisabilité*, ou la *portabilité*, restent aujourd'hui des domaines de recherche actifs et rejoignent des sujets plus généraux comme la séparation des préoccupations (*separation of concerns*), ou les problèmes de composabilité en génie logiciel.

3.1.3 Types d'architectures pour la tolérance aux fautes

Plusieurs architectures ont successivement été proposées pour les plates-formes de tolérance aux fautes. En s'inspirant de la classification proposée par [Briot et al. 1998] pour l'intégration de la distribution et du parallélisme à l'orienté objet, l'on peut schématiquement classer ces plates-formes de la manière suivante :

Approches explicites
: Les mécanismes de tolérance aux fautes sont présentés sous la forme d'une API (*Application Programming Interface*, interface de programmation applicative) spécifiquement conçue pour la tolérance aux fautes. Cette API peut par exemple être fournie par une bibliothèque, par un système opératoire spécifique, ou au travers d'un service distribué (comme les services CORBA). Les plates-formes de ce type n'offrent quasiment aucune transparence, et leur utilisation n'est pas aisée. Elles fournissent en effet des interfaces qui leur sont propres, de plutôt bas niveau, qui se présentent typiquement comme un ensemble de fonctions en C pour manipuler des brins d'exécution, des messages génériques, des groupes de processus, des primitives de communication, de multi–

diffusion *etc.* Elles nécessitent une très bonne connaissance des protocoles mis en jeu, et ne peuvent facilement être utilisées par des non–spécialistes de la tolérance aux fautes. En revanche, certaines d'entre elles (les plus récentes) sont extrêmement flexibles, en offrant de nombreux mécanismes d'extension et de reconfiguration.

Approches par intégration

Dans les approches par intégration une plate-forme de développement « générique » non tolérante aux fautes (un bus à objets CORBA, un système opératoire POSIX) est durcie en construisant à l'intérieur de la plate-forme des mécanismes de tolérance aux fautes (par exemple de diffusion atomique, ou de capture d'état). L'interface originale de la plate-forme n'est pas modifiée (à l'exception de quelques extensions de configuration), ce qui permet d'atteindre un haut degré de transparence. En interne les parties de la plate-forme qui traitent explicitement de la tolérance aux fautes et celles qui n'en traitent pas sont fusionnées et ne peuvent être facilement séparées les unes des autres, ce qui nuit à leur flexibilité.

Approches par interception

Les approches par interception consistent comme les approches par intégration à étendre une plate-forme non tolérante aux fautes, mais en utilisant des mécanismes spécifiques à la plate-forme pour intercepter certains de ses traitements internes : appels système, interception de requêtes en utilisant les mécanismes d'invocation dynamique d'un bus à objets (dans CORBA le *Dynamic Skeleton Interface* ou DSI et le *Dynamic Invocation Interface* ou DII) [Felber et al. 1996]. Ce type d'architectures conserve la transparence apportée par l'intégration, et ré–introduit en même temps la flexibilité des approches explicites, en rendant accessibles les éléments qui, à l'intérieur d'une plate-forme, implémentent la tolérance aux fautes. L'utilisation d'intercepteurs permet de plus facilement modifier les mécanismes inclus dans la plate-forme ou de les réutiliser sur d'autres implémentations qui utilisent la même interface d'interception.

Approches réflexives

Les approches réflexives peuvent être vues comme une formalisation et un enrichissement des approches par interception, en proposant une méta-interface qui permette de manipuler de manière disciplinée et contrôlée les éléments liés au fonctionnement d'une plate-forme généraliste (appels de méthodes distantes, cycle de vie des objets, état de la plate-forme et des objets qui vivent au-dessus d'elle). L'approche réflexive consiste à ajouter les mécanismes de tolérance aux fautes sous forme de méta-programmes, exprimés en utilisant les éléments du modèle de programmation de la plate-forme originale. La réflexivité assure à la fois transparence et flexibilité des mécanismes, tout en proposant un modèle de programmation clair, cohérent, et robuste.

3.2 Exemple de plates-formes pour la tolérance aux fautes

3.2.1 Les pionniers : les plates-formes explicites

Plates-formes de synchronisation : ISIS, PSYNC, *etc.*

ISIS, PSYNC, TRANSIS sont des kits de développement d'applications distribuées tolérant les fautes, basés sur des protocoles de multi–diffusion qui permettent l'ordonnancement cohérent d'évènements distribués.

Le premier d'entre eux, ISIS, a été développé autour de Kenneth Birman à l'Université de Cornell sur une période allant de 1981 (début du projet) à 1998 (arrêt de la commercialisation d'ISIS par la société *Stratus Computer, Inc.*). Voici comment ISIS a été présenté par Birman :

> *Our basic premise is that the complexity of fault–tolerant distributed programs precludes their design and development by non–experts. [...] Alternatives to direct implementation of fault–tolerant systems are needed if fault–tolerance is to gain wide availability.*
> *The* ISIS *project seeks to address this need by automating the transformation of fault–intolerant program specification into fault-tolerant implementations.*
>
> *[Birman 1985]*

La première version d'ISIS fournit des services de multi–diffusion tolérants aux défaillances par arrêt organisés en trois primitives [Birman 1985]. Chacune de ces primitives impose des contraintes différentes sur l'ordonnancement respectif de leurs messages à la réception. Le modèle de développement d'ISIS est basé sur la notion d'*objet résilient*, une entité logicielle distribuée organisée en un certain nombre de répliques fonctionnant selon un mode de réplication passive.

Déjà dans sa première version ISIS fut un projet particulièrement précurseur. On y trouve des notions de langage de description d'interface (IDL) avec les « *abstract type specifications* ». La notion d'*objet résilient* annonce les groupes d'objets trouvés par exemple dans le standard FT-CORBA [OMG 2002b]. Les adresses de ces groupes, appelées *capacity*, annoncent les IOGRs (*interoperable object group references*) toujours dans FT-CORBA. Dans la foulée d'ISIS ont été développées d'autres infrastructures similaires, avec de nouveaux choix d'implémentation pour augmenter leur performance, ou des sémantiques légèrement différentes pour prendre en charge d'autres modèles de programmation. On peut notamment citer PSYNC [Peterson et al. 1989] ou TRANSIS [Amir et al. 1992].

La limitation des primitives de multi–diffusion proposées par ces différentes plates–formes, et l'impossibilité d'adapter leurs propriétés en fonction du contexte d'opération ont été à l'origine de projets qui ont plus particulièrement recherché la flexibilité tout en gardant les capacités des plates–formes précédentes. CONSUL [Mishra et al. 1993], basé sur PSYNC, HORUS [van Renesse et al. 1996] et ENSEMBLE [van Renesse et al. 1998], successeurs d'ISIS, sont de ceux-là.

Ces différents projets introduisent la notion de *micro–protocole*. Les micro–protocoles sont des modules logiciels qui transforment un flux d'événements en entrée en un flux d'événements en sortie en garantissant diverses propriétés d'ordre ou de robustesse. Ils peuvent être assemblés en de multiples combinaisons pour créer des piles de communication personnalisées selon les besoins de l'application. Deux questions en particulier sont abordées pour la réalisation de ces micro–protocoles :

1. Quelle interface générique choisir pour pouvoir composer librement les différents micro–protocoles ?
2. Comment prendre en compte les contraintes de dépendance entre les différents micro–protocoles ?

Pour répondre à la première question, la plupart de ces projets choisissent une forme d'interaction par événements : réception, envoi de message, détection d'une défaillance, entrée d'un nouveau membre dans un groupe, *etc.* Le second problème est abordé en décrivant les dépendances entre protocoles sous forme soit d'un ordre partiel (deux protocoles sont reliés si le fonctionnement correcte de l'un assure le fonctionnement correcte du second [Mishra et al. 1993]), ou de manière plus élaborée sous forme de « contrats » conditions–garanties [van Renesse et al. 1998].

ENSEMBLE, le plus récent des projets, propose même un protocole de reconfiguration dynamique de ses piles, PSP (*Protocol Switching Protocol*), structuré selon un commit en trois phases avec un coordinateur centralisé [van Renesse et al. 1998].

Plates-formes pour la capture d'état

Les approches explicites pour la capture d'état sont pour la plupart implémentées au niveau langage. À la différence des plates-formes axées sur la synchronisation, elles ne précèdent pas les approches de capture d'état intégrée aux plates-formes, mais ont plutôt été développée en parallèle. Nous en donnons ici un bref aperçu et renvoyons le lecteur à [Courtès 2003] pour un état de l'art plus détaillé.

Ces approches requièrent de la part du programmeur d'indiquer explicitement quelles données il souhaite gérer de manière persistante. Le *Persistent State Service* (PSS) de CORBA, utilise par exemple un langage de description de l'état à sauvegarder appelé PSDL (*Persistent State Description Language*) pour décrire les attributs de chaque objet distribué qui doivent pouvoir être rendus persistants [OMG 2002f]. Un autre exemple est le langage PS–ALGOL, qui rend possible la manipulation transparente d'objets stockés dans une base de donnée persistante [Atkinson et al. 1983]. Dans son code, le programmeur n'a pas à faire la distinction entre objets persistants et objets « éphémères », mais il doit choisir en amont quels objets seront gérés par la base de données.

Parce qu'elles utilisent une description de l'état à sauvegarder de haut niveau (basée sur le système de types de CORBA pour PSS, ou celui d'Algol pour PS–ALGOL), et font appel à des formats de mise à plat indépendants des plates-formes sous-jacentes, ces approches permettent de transférer des états entre des contextes d'exécution (OS, matériel)

hétérogènes. Le *Persistent State Service* de CORBA, permet même d'échanger un état entre deux implémentations d'un même service écrites dans des langages différents. Parce qu'elles sont explicites, ces techniques nécessitent cependant de la part du concepteur de préciser en amont du développement quelles données devront être gérées de manière persistante pour permettre des reprises cohérentes. Cette contrainte rend l'adaptation des choix de persistance difficile en cours de projet. Elle ne permet pas non plus de gérer la capture d'état de manière totalement transparente vis-à-vis des aspects fonctionnels de l'application. Enfin, les approches explicites de capture d'état sont mal adaptées à la capture de l'aspect « contrôle » de l'état d'une application (quels traitements sont en cours ? avec quels degrés d'avancement ?), qui est intrinsèquement lié au substrat d'exécution et se prête mal à toute approche déclarative.

3.2.2 Plates-formes par intégration et interception

Synchronisation et communication de groupe

En dépit de l'ambition affichée par Birman au début du projet ISIS (voir p.41) l'utilisation des plates–formes de réplication présentées précédemment n'est pas aisée. Le développement concomitant d'intergiciels orientés objet pour la programmation distribuée tels que CORBA ou RMI (section 1.2.4 page 14) a motivé plusieurs travaux pour réussir à combiner la facilité de développement offerte par ces intergiciels (réutilisabilité, portabilité, interopérabilité, intégration des anciennes applications) avec les propriétés de tolérance aux fautes et la très haute flexibilité assurées par les kits que nous venons de présenter.

Le projet ELECTRA [van Renesse et al. 1996, Maffeis 1995], est un exemple d'approche par intégration. ELECTRA utilise la plate-forme de communication de groupe HORUS, citée précédemment, pour implémenter un bus à objets conforme à la norme CORBA qui propose deux styles d'interaction : mono–interlocuteur (c'est-à-dire la forme d'interaction standard) et multi–discussion (où la requête est envoyée vers plusieurs objets formant un groupe). La multi–discussion est transparente : il n'y a pas de différence pour le client lors de l'invocation entre un service mono–interlocuteur et un service en multi–discussion. ELECTRA propose plusieurs manières de sélectionner la réponse d'une requête en multi–discussion : après l'obtention d'une majorité de réponses, après l'obtention de toutes les réponses, *etc.* L'interface de gestion des groupes d'objets (création, modification, génération de références pour un groupe), est implémentée dans un composant des objets CORBA appelé BOA (*Basic Object Adapter*), qui dans la version de CORBA utilisée par ELECTRA fait le lien entre les requêtes de services entrantes et l'implémentation de ces services au niveau applicatif[2].

Un des reproches faits à ELECTRA est son couplage trop fort entre communication de groupe et la plate-forme CORBA : il n'est pas possible de changer de kit de communication de groupe, ou d'ajouter une capacité de multi–discussion à une implémentation CORBA préexistante. Ces limitations ont été à l'origine de travaux *par interception*, comme ETERNAL

[2]Le *Basic Objet Adapter*, suite à des problèmes de spécification dans la norme qui gênaient l'interopérabilité a été remplacé à partie de la version 2.2 de CORBA par le *Portable Objet Adapter*, ou POA.

[Narasimhan et al. 1997] ou OGS (*Object Group Service*) [Felber 1998], qui en déroutant certaines interactions du système ajoutent des capacités de réplication de manière transparente, tout en conservant une forte modularité entre communication distribuée « mono–interlocuteur » et communication de groupe.

Les deux projets se distinguent cependant par leur niveau d'intervention : ETERNAL capture les appels faits par le bus à objets sur l'interface de communication de l'OS sous-jacent et les re–dirige sur un service de communication de groupe, ce qui permet de rendre la réplication transparente aux clients. OGS adopte une approche par *service* CORBA : les mécanismes de communication de groupe (création, gestion, invocation, reconfiguration) sont fournis explicitement sous la forme d'objets CORBA au travers de leurs interfaces IDL. Pour rendre transparent ce service, OGS utilise les mécanismes de réception et d'invocation génériques de CORBA (respectivement DSI, *Dynamic Skeleton Interface*, et DII, *Dynamic Invocation Interface*), qui permettent d'intercepter de manière transparente les requêtes faites par un client sur un serveur, et de re–diriger ces requêtes sur le service de groupe.

Ces travaux ont entre autres été à l'origine du standard FT-CORBA (*Fault-Tolerant CORBA*), une extension du standard CORBA qui originellement n'intégrait pas de considérations de tolérance aux fautes [OMG 2002b]. Les approches par interceptions ont aussi motivé l'introduction d'intercepteurs standardisés dans CORBA [OMG 2002c], qui, même s'ils sont limités (il est par exemple impossible de modifier les paramètres des requêtes), introduisent une part de réflexivité dans l'ORB.

Capture d'état intégrée

Capture d'état et gestion mémoire sont deux problématiques très proches lorsqu'on les aborde du point de vue d'un système d'exploitation. Le principe consiste à enregistrer en format binaire l'espace mémoire alloué à une application sur un support stable (un disque dur typiquement). La mise à profit des mécanismes de gestion mémoire permet notamment d'arrêter le moins longtemps possible l'application dont on capture l'état. Deux mécanismes sont essentiellement utilisés : le mécanisme de protection des pages mémoire[3] d'un OS et celui de permutation entre mémoire et disque dur (*swapping*) :

- Le mécanisme de protection des pages mémoire permet à un OS de contrôler et de modifier les droits d'accès des pages mémoire allouées à une application. Cette fonctionnalité sert essentiellement des objectifs de confinement et de protection. Par exemple, si l'application essaye d'écrire dans une page qui ne lui est accessible qu'en lecture, le système d'exploitation en est notifié et peut réagir à sa guise.
- Le système de permutation permet de donner accès à un espace mémoire plus grand que la mémoire physique disponible en utilisant la mémoire vive de l'ordinateur comme un cache des blocs mémoire sauvegardés sur le disque. À un instant donné, la

[3]Une page mémoire est un bloc mémoire de taille fixe, utilisé comme unité élémentaire de gestion mémoire par les OS et les processeurs, par exemple pour la gestion de l'adressage virtuel dont nous avons déjà parlé (section 1.2.2 page 11).

mémoire effectivement utilisée par une application se trouve répartie entre la mémoire vive et l'espace de permutation du disque, en fonction des pages mémoire les plus récemment utilisées.

Nous commençons par illustrer l'intérêt du mécanisme de protection, en présentant quatre techniques distinguées par Li *et al.* en selon le degré d'asynchronisme autorisé entre progression du programme et enregistrement sur le disque [Li et al. 1994] :

Approche séquentielle
Le plus simple des algorithmes. L'application est entièrement arrêtée. L'état de la mémoire est sauvegardée d'un bloc, puis la reprise a lieu. Cette approche produit la plus grande latence, du fait des temps d'écriture sur le disque.

Par tampon en mémoire vive
Cette méthode améliore la précédente en répliquant l'intégralité de l'espace mémoire en mémoire vive pendant le gel de l'application, puis en enregistrant cette copie sur le disque de manière asynchrone après la reprise. La latence est moins grande mais toujours importante. Cette approche et la précédente n'utilisent pas le mécanisme de protection, est assure la cohérence de l'état capturé en gelant l'application durant toute la phase de copie mémoire, ce qui est particulièrement inefficace.

Copie sur écriture (copy-on-write)
Cette méthode met le mécanisme de protection à profit pour permettre à l'application de continuer à progresser, *alors même* que son état est capturé. Les étapes de cette approche sont les suivantes :

1. L'application est gelée. Toutes les pages de l'application sont protégées en lecture seule en utilisant le mécanisme de protection, puis l'application est redémarrée.
2. De manière asynchrone, les pages sont successivement dupliquées en mémoire vive, et, uniquement après leur copie, de nouveau remises en lecture / écriture, ce qui assure une prise d'état cohérent. Si l'application tente d'accéder à une page toujours protégée en lecture seule parce qu'elle n'a pas encore été dupliquée, la duplication de cette page a lieu immédiatement pour permettre à l'application de continuer à fonctionner.
3. Lorsque toutes les pages ont été dupliquées, l'image binaire résultante est copiée sur le disque.

Cette approche offre une très bonne latence mais a comme la précédente de forts besoins en mémoire vive (le double de la taille de l'image à sauvegarder).

Copie sur écriture avec tampon
(appelé *Concurrent, Low-Latency Algorithm* dans [Li et al. 1994]) Cette approche reprend le même principe que la copie sur écriture mais commence l'écriture sur le disque des pages dupliquées (étape 3 précédente) parallèlement au processus de duplication en mémoire vive (étape 2). Cette parallèlisation supplémentaire permet de limiter la zone de mémoire vive utilisée pour la duplication des pages, et ainsi de contrôler la consommation mémoire de la capture d'état.

Le mécanisme de permutation est quant à lui utilisé dans [Espen Skoglund 2000]. Telles quelles, les pages mémoire d'une application présentes dans l'espace de permutation d'un OS (*swap*) ne sont pas utilisables pour redémarrer l'application dans un état cohérent : certaines pages peuvent ne pas être présentes sur le disque, ou être présentes dans un état obsolète, incohérent avec les autres pages de l'application. Pour utiliser la permutation pour la capture d'état, un mécanisme de mise en cohérence doit donc être mis en place. Il n'est alors plus nécessaire de sauvegarder les pages *déjà* présentes sur le disque, ce qui rend cette approche particulièrement efficace. Dans [Ousterhout et al. 1988], ce système est utilisé dans l'OS réparti SPRITE pour permettre la migration de processus.

Pour être tolérante aux fautes cependant, la mise en cohérence doit être atomique. Dans [Espen Skoglund 2000], les auteurs proposent une stratégie basée sur le dédoublement (*shadowing*) des pages mémoire qui permet de conserver la dernière capture cohérente pendant qu'une tentative de capture est réalisée en parallèle (en ceci similaire aux concepts de *tentative* et de *permanent checkpoints* présenté dans un contexte plus théorique dans [Koo & Toueg 1987]).

La capture d'état binaire d'une application, que ce soit par l'un ou l'autre des mécanismes présentés, doit cependant surmonter plusieurs problèmes :

1. L'image binaire de l'espace mémoire d'un processus ne contient pas toutes les informations nécessaires à la reprise de l'application : le système d'exploitation gère en effet *au nom de l'application* un certain nombre d'entités (descripteurs de fichier, verrous, canaux de communication) dont l'état est interne à l'OS et n'est pas capturé par la seule image de l'espace d'adressage.
2. Pour référencer les entités gérées en son nom par l'OS, mais aussi pour désigner des objets qui lui sont extérieurs (fichiers, serveurs distants), une application utilise des identifiants qu'elle stocke en mémoire, et qui lui sont fournis par exemple lorsqu'elle ouvre un fichier, ou lorsqu'elle créé un verrou. Ces identifiants ne sont plus valables lors d'une reprise. Même lorsque de nouvelles entités sont recréées dans l'OS en remplacement des anciennes disparues, ces nouvelles entités ont souvent des identifiants différents, qui doivent être « réappris » par l'application.
3. Certaines entités « extérieures » à l'application, comme des fichiers, ou des ressources graphiques, peuvent être perdues en même temps que l'application et ne plus être disponibles lors d'une reprise, ou posséder un état incohérent avec celui du reste de l'application. Le mécanisme de capture doit être capable de restaurer leur état de manière cohérente avec celui de l'application. Ce problème de la partie « externe » de l'état d'une application rejoint celui de la cohérence des états distribués que nous avons mentionnés dans la section 3.1.1 page 37.

Pour résoudre le premier problème, la plupart des approches utilisent une stratégie par journalisation qui consiste à intercepter les appels OS faits par l'application pour en inférer l'état des objets que l'OS gère au nom de l'application [Karablieh & Bazzi 2002, Dieter & Jr. 2001, Russinovich et al. 1993, Taylor & Wright 1986]. Pour résoudre le second problème, deux stratégies ont été proposées : soit modifier dans l'image binaire les descripteurs dont

on connaît la sémantique (c'est ce que fait [Dieter & Jr. 2001] pour les identifiants des brins d'exécution sous GNU/LINUX, en tirant partie du fait que la gestion du multitraitement (*multithreading*) de GNU/LINUX jusqu'à la version 2.4 du noyau est implémentée dans une bibliothèque gérée en espace utilisateur en dehors du noyau) ; soit insérer une couche de traductions des identifiants entre le système d'exploitation et l'application [Karablieh & Bazzi 2002].

La capture binaire peut donc s'avérer une solution relativement transparente, une fois surmontés les problèmes que nous venons de mentionner. Elle s'avère, cependant, très désavantageuse en termes de flexibilité. Les états ainsi capturés ne sont en effet pas portables d'une système d'exploitation à l'autre, et requièrent d'utiliser des architectures matérielles et logicielles rigoureusement identiques[4]. Cette contrainte d'homogénéité logicielle et matérielle n'est pas facile à réaliser dans de grands systèmes complexes, et se trouve de plus être particulièrement handicapante du point de vue de la tolérance aux fautes (nous renvoyons le lecteur à la section 1.1.2 page 4 pour une discussion sur l'importance de la diversité pour la tolérance aux fautes).

Ce manque de flexibilités des approches par capture binaire se retrouve lorsqu'il s'agit de porter un même mécanisme d'un système d'exploitation à l'autre, la gestion de la mémoire utilisant soit des adaptations *ad hoc* au niveau de l'OS, soit des particularités non–standards du système d'exploitation choisi (nous renvoyons à [Dieter & Jr. 2001] et [Dieter & Jr. 1999] pour une présentation d'un portage d'une bibliothèque de capture binaire d'applications à brins d'exécution multiples de SOLARIS vers GNU/LINUX).

Conclusion sur les approches par intégration et interception

Les approches par intégration et interception permettent, au contraire des approches explicites, d'assurer une grande transparence. Les approches par intégration sont en contrepartie beaucoup moins flexibles, et selon le niveau où elle sont implémentées, peuvent rendre difficile le maintien de la cohérence du système (notamment dans le cas des captures d'état binaire). Les approches par interception sont elles plus adaptables, mais parce qu'elles sont développés de manière *ad hoc*, elle souffre de l'absence modèle de programmation clair, cohérent, et robuste.

3.2.3 Les solutions réflexives

Les approches réflexives pour la tolérance aux fautes ont essentiellement mis à profit la notion de *protocole à méta-objets* (*MOP* en anglais), que nous avons introduite dans la section 2.2.1 page 26. Ce n'est cependant pas la seule approche, des approches plus

[4]Ce problème de portabilité est notamment dû aux bibliothèques partagées utilisées par l'application, en particulier les bibliothèques système, que nous avons introduites dans la section 1.2.2 page 12. Ces bibliothèques fonctionnent au-dessus du noyau en mode utilisateur. La partie de leur état relatif à une application est contenue dans l'espace mémoire de cette application, et est donc aussi capturée lors d'une capture binaire. Lors d'une restauration, les bibliothèques utilisées doivent être rigoureusement identiques aux originales, sous peine d'incompatibilité assurée entre l'image capturée et les adresses utilisées par les nouvelles versions.

classiques pouvant être apparentées à une certaine forme de réflexivité, par exemple la notion de réécriture source à source. Dans tous les cas il s'agit en pratique de la mise en œuvre de la réflexivité à un seul niveau d'abstraction : le niveau langage. Très peu de travaux se sont attaqués à l'utilisation ou à la mise en œuvre de la réflexivité pour la tolérance aux fautes des logiciels exécutifs. Nous donnons ici différents exemples de plates-formes réflexives basées sur des MOP ainsi que, rapidement, un exemple concernant la mise en place d'empaquetages (*wrapper*) de confinement par une approche réflexive sur des micro-noyaux temps-réel.

Réplication distribuée : MAUD, GARF

MAUD [Agha et al. 1992] est une plate-forme de développement distribuée qui utilise une approche réflexive pour proposer des mécanismes de tolérance aux fautes modulaires, composables, et transparents. MAUD est basée sur le langage multi–agents HAL, et associe à chaque acteur trois méta-objets différents : un *dispatcher* qui intercepte les messages sortants, une *receive queue* (queue de réception), qui traite les messages arrivants, et un objet *acquaitance* qui supervise avec quels autres acteurs un acteur donné est connecté. Une stratégie « *en pelure d'oignons* » est proposée pour permettre la composition récursive de plusieurs méta-objets. Cette stratégie impose que les hiérarchies de méta-objets du côté d'un agent expéditeur et du côté d'un agent destinataire qui communiquent par messages soient les miroirs l'une de l'autre.

Comme MAUD, GARF [Guerraoui et al. 1997] est une plate-forme distribuée orientée objet écrite en SMALLTALK qui utilise les caractéristiques réflexives du langage SMALLTALK pour intégrer de manière transparente à l'application des mécanismes de réplication distribuée. Par cette approche réflexive, GARF sépare clairement les aspects d'un système qui sont liés au parallélisme, à la distribution et à la réplication, des considérations applicatives qui peuvent continuer à être abordées selon un point de vue séquentiel et centralisé. Pour réaliser cette séparation, GARF distingue deux sortes d'objets : les *data objects* (objets–donnée) qui encapsulent la logique applicative indépendamment de toute considération de distribution et les *behavioral objects* (objets comportementaux) qui interceptent les interactions inter–objets en traitant plus particulièrement des aspects ayant trait aux mécanismes d'invocation (*encapsulators*) et de distribution (*mailers*). GARF est construit au-dessus de la plate-forme de synchronisation ISIS qui est rendue disponible par l'intermédiaire d'*encapsulators* et de *mailers* spécifiques. Cependant un développeur de tolérance aux fautes a toute latitude pour adapter les objets de réplication proposés par défaut par GARF à ses propres besoins, éventuellement en court-circuitant complètement ISIS.

Capture d'état symbolique par réécriture source à source

Une autre utilisation de la réflexivité consiste à utiliser un compilateur source à source pour exprimer l'état d'un programme au travers d'une version modifiée de ce programme, qui lorsqu'elle est compilée et exécutée se comporte de manière équivalente à l'originale

mais redémarre au point de la prise d'état [Theimer & Hayes 1991]. Cette approche a inspiré plusieurs projets [Karablieh & Bazzi 2002, Ferrari et al. 1997, Balkrishna Ramkumar 1997] pour la capture d'état d'applications à brins d'exécution multiples. Ces projets sauvegardent une représentation *symbolique* (par opposition à « binaire ») de l'état des brins d'exécution d'une application, soit en utilisant les informations de débogage incluses dans l'exécutable par les compilateurs les plus courants[5] [Balkrishna Ramkumar 1997, Ferrari et al. 1997], soit en rajoutant du code qui *simule* symboliquement l'exécution de la pile d'un brin [Karablieh & Bazzi 2002]. L'état des brins d'exécution ainsi capturé ne dépend que de la structure du code source du programme considéré, ce qui le rend extrêmement portable.

La restauration d'une pile à partir de sa représentation symbolique s'opère en rajoutant au début de chaque fonction (ou méthode) un bloc de code qui permet de restaurer les variables locales associées à celle-ci puis de *sauter* directement aux différents appels emboîtés contenus dans la fonction. La restauration commence alors au point d'entrée du programme, à la fonction *main*, le bloc de code ajouté se chargeant de « sauter » à l'appel de fonction indiqué par le pile symbolique. Ce processus se reproduit récursivement, jusqu'à restitution complète de la pile sauvegardée. les entités de synchronisation gérées par le noyau pour l'application (sémaphores, verrous, *etc.*) sont quant à elles recréées en utilisant les informations collectées par l'interception des appels OS appropriés.

Gestion de l'état et distribution en C++ : FRIENDS

Le projet FRIENDS V.1 [Pérennou & Fabre 1998] pousse plus loin l'utilisation de la réflexivité pour la tolérance aux fautes en adaptant les stratégies de GARF et MAUD à un langage de programmation courant, C++, par l'entremise du compilateur réflexif OPENC++. FRIENDS V.1 intègre en plus de la tolérance aux fautes des considérations de sécurité, d'authentification, et de capture d'état pour des objets ne contenant que des attributs ayant des types de base. Ce projet a été étendu à un environnement CORBA dans le projet FRIENDS V.2 [Killijian & Fabre 2000, Killijian et al. 1999, Killijian et al. 1998], en intégrant une fonctionnalité de capture d'état généralisée, transparente à l'utilisateur. La pierre angulaire de ces projets est constituée par le protocole d'interaction entre les objets du niveau applicatif et les méta-objets qui les manipulent au méta-niveau : le protocole à méta-objets (en anglais *MOP*, *meta-object protocol*). FRIENDS V.2 intègre ainsi des mécanismes de réification du cycle de vie d'un objet (création, destruction), de l'invocation de ses méthodes, des mécanismes d'introspection et de modification (intercession) de son état *etc.*

L'idée particulièrement innovante de FRIENDS V.2 consiste à utiliser la représentation commune des données standardisées par CORBA (*Common Data Representation*, CDR) pour mettre à plat les structures orientées objet qui sont réifiées par le compilateur ouvert OPENC++ [OMG 2002a]. Cette approche, qui peut être étendue à tout langage orienté objet pour lequel existe un *mapping* CORBA, a permis de réaliser des transferts d'état entre des programmes écrits dans des langages différents, ici C++ et Java [Killijian et al. 2002].

[5] Ces informations permettent par exemple d'introspecter la pile d'appels d'un brin d'exécution, ou la valeur de ses variables locales, allouées sur sa pile.

L'hétérogénéité des langages est souvent considérée comme un obstacle rédhibitoire au transfert d'état, mais lorsqu'elle peut être maîtrisée, elle apporte un avantage considérable du point de vue de la tolérance des fautes logicielles de par la diversité des implémentations qu'elle autorise.

Systèmes d'exploitation

La réflexivité a aussi été utilisée au niveau du système d'exploitation pour permettre le durcissement de micro-noyaux (*Chorus*, *LynxOS*) 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 [Rodriguez et al. 2000, Arlat et al. 2002]. Cette approche réifie des événements internes à un micro-noyau tels que la création d'un brin d'exécution ou la prise d'une sémaphore vers un *méta-noyau*, et permet alors la spécification sous forme de formules de logique temporelle linéaire de contraintes de fonctionnement « normal » qui permettent de détecter les erreurs au plus tôt, et soit d'imposer un silence sur défaillance (confinement), ou de tenter d'entreprendre des mesures correctrices [Rodriguez 2002].

3.3 Les limites des approches proposées

Nous avons présenté dans les paragraphes qui précèdent plusieurs approches pour intégrer la tolérance aux fautes, ou au moins certains de ses aspects, dans les plates-formes d'exécution. Ces différentes approches se distinguent par le degré de transparence qu'elles offrent au programmeur d'application : ce dernier doit-il prendre explicitement en compte la tolérance aux fautes lors du développement des aspects applicatifs du système ? ou ces considérations peuvent–elles être déléguées à une équipe spécialisée, travaillant indépendamment et de manière parallèle ? Ces approches se distinguent aussi par leur flexibilité : les mécanismes proposés sont-ils configurables ? Fournissent ils les moyens de développer des services non–prévus initialement à partir de « micro–services » plus élémentaires ? Est-il possible de reconfigurer dynamiquement les mécanismes de tolérance aux fautes de la plate-forme ?

Nous avons vu que les architectures réflexives représentaient l'une des formes de structuration les plus abouties pour répondre à ces problèmes. Beaucoup des autres approches que nous avons mentionnées peuvent d'ailleurs souvent se lire comme une application implicite et partielle de ces principes. Cependant, dans la perspective des systèmes multi-couches que nous avons présentée au chapitre 1, les approches réflexives pour la tolérance aux fautes se situent toutes à un seul niveau d'abstraction, le plus souvent en utilisant les informations rendues disponibles par un langage de programmation (C++ et CORBA IDL pour FRIENDS V.2, SMALLTALK pour GARF, HAL pour MAUD). Parce qu'elles se restreignent à un niveau unique, ces approches sont aveugles aux informations des niveaux inférieurs qui sont masquées par l'interface à laquelle elles se limitent. Par exemple, en n'assurant

aucune garantie sur les modes de concurrence utilisés dans l'ORB pour réaliser le traitement des requêtes, la norme CORBA ne permet pas de réaliser une capture d'état d'application CORBA à brins d'exécution multiples en se limitant à la seule sémantique fournie par le langage C++ et l'IDL CORBA [Killijian 2000]. Plus généralement, la nature des informations réflexives fournies par les différentes couches d'un système « stratifié » peut être résumée de la manière suivante :

- Les plus hauts niveaux possèdent des modèles de programmation dotés d'une sémantique riche, associée 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, parce qu'ils dépendent de choix d'implémentation et de micro-états qui ne lui sont pas accessibles.
- Les modèles de programmation des couches les plus basses 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'à 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 descripteur de fichier ?).

La réalisation de mécanismes de tolérance aux fautes a beaucoup à gagner de la combinaison des propriétés des hauts et bas niveaux d'abstraction d'une architecture complexe. L'approche proposée dans [Karablieh & Bazzi 2002], parce qu'elle intègre réflexivité à la compilation (sémantique langage), et interception des appels POSIX (sémantique OS) représente un premier pas dans cette direction.

La tolérance aux fautes, parce qu'elle englobe l'intégralité d'un système complexe nécessite pour atteindre les plus hauts degrés de couverture de combiner la puissance d'inférence sémantique des hautes couches avec la force de contrôle brute des bas niveaux en une perception globale du système.

Dans le chapitre 4, nous partons de cette constatation et abordons plus en détail pourquoi et comment combiner les capacités réflexives disponibles dans les différentes couches d'un système complexe, ce que nous avons baptisé la « *réflexivité multi-niveaux* ».

Chapitre 4

La réflexivité multi–niveaux : notions et principes

Les mots dessinent la signification du monde, ils sont une grille qui permet de le comprendre, de s'en saisir, un outil pour le rendre communicable, même si leur emprise est limitée, parfois maladroite, car le monde est toujours en avance, et désavoue par sa complexité et son clair–obscur toute tentative de le figer en significations univoques.

David Le Breton, *Du silence* [Breton 1997]

Nous avons, au chapitre précédent, souligné la complémentarité des hauts et bas niveaux d'abstraction rencontrés dans un système complexe. Le but de ce chapitre est de s'appuyer sur cette constatation pour proposer une démarche générale de développement de la tolérance aux fautes des logiciels complexes basée sur une approche réflexive qui intègre tous les niveaux d'un système complexe. Ce problème requiert de pouvoir intégrer les considérations *algorithmiques* de la tolérance aux fautes (capacités d'action et d'observation) aux contraintes *architecturales* des systèmes complexes (transparence, encapsulation, abstraction, niveaux).

Considérée d'un point de vue architectural, la tolérance aux fautes d'un système complexe peut être comprise comme un processus de « filtrage » dont l'entrée est « multi-niveaux », et la sortie « mono-niveau ». Elle doit en effet protéger l'utilisateur de fautes qui peuvent impacter un système complexe à tous les niveaux de son architecture (en-

trée « multi-niveaux »). En contrepartie, le rôle de protection de la tolérance aux fautes (son résultat, sa « sortie ») est déterminé par le service perçu par l'utilisateur (voir section 1.1.1 page 2), c'est à dire par le plus haut « niveau » d'abstraction du système (sortie mono-niveau). L'implémentation de la tolérance aux fautes dans un système complexe suppose par conséquent de comprendre comment les fautes propagent leurs erreurs à travers les couches d'un système complexe pour venir impacter l'utilisateur, et inversement comment des exigences de sûreté exprimées au niveau de l'utilisateur se répercutent en responsabilités de tolérance aux fautes sur les différents niveaux du système.

Pour répondre à ces questions, nous développons dans un premier temps un méta-modèle des architectures en couches, en introduisant les notions de *modèle de programmation*, d'*interface*, et de *lien inter-niveaux* [Taïani et al. 2002]. Une fois ce méta-modèle développé, nous analysons comment un algorithme de tolérance aux fautes peut prendre place dans une architecture multi-niveaux, en introduisant les notions d'*empreinte réflexive*, et en étudiant comment les algorithmes de tolérance aux fautes peuvent continuer à fonctionner correctement tout en ayant une perception imparfaite de la réalité.

Ces différentes notions nous permettrons de relier l'aspect architectural et l'aspect algorithmique de la tolérance aux fautes des systèmes complexes, ce que nous traduirons par la proposition d'une démarche de développement d'architectures réflexives multi-niveaux.

4.1 Un méta-modèle des architectures en couches

Réflexivité et architecture entretiennent une relation nourrie : comme nous l'avons exposé au chapitre 2 page 19, la réflexivité consiste à donner accès et à permettre la modification des éléments générateurs d'un système (les mots-clefs d'un langage, les primitives d'un substrat d'exécution). C'est cette utilisation d'éléments générateurs qui permet à un méta-programme d'être appliqué à tous les systèmes qui partagent les mêmes éléments de base, les mêmes éléments constitutifs. La question se pose donc de savoir ce qui dans une architecture en couches joue le rôle d'éléments constitutifs. En particulier il apparaît nécessaire de comprendre comme les entités observées à un niveau donné sont reliées avec celles de ses niveaux voisins. Dans cette perspective, nous revenons ici plus en détail sur la notion de *couches*, en précisant les notions de *modèle de programmation*, d'*interface*, de *niveau*, ce qui nous conduira à proposer une taxonomie des formes de liens inter-niveaux qui régentent la cohésion d'un système multi-couches.

4.1.1 Structure et cognition

Nous l'avons vu dans la section 1.2 page 8 du chapitre 1, la structuration en couches des systèmes logiciels facilite la maîtrise de leur complexité lors des activités de développement. Mais que représente cette « maîtrise » de la complexité ? Comment s'opère–t–elle ? Quels acteurs implique–t–elle ? Il nous a semblé important de souligner ici l'aspect « cognitif » du développement logiciel : écrire un programme suppose qu'un humain à un moment donné

exprime une *intention* dans un *formalisme* qui lui sert de cadre de développement. Cette description brute de l'intention du programmeur n'est rien pour la machine, qui ne connaît pour sa part que le *langage machine*. Ce *langage machine* est quant à lui complètement ignoré du développeur (le plus souvent). Ce dernier utilise un *modèle de programmation*, qui capture ce qu'il sait que le système peut comprendre, pour exprimer ce qu'il souhaite que le système réalise. Ce sont les couches du système, telles que nous les avons présentées dans le chapitre 1, qui à la manière d'un pont conceptuel relient le modèle perçu par le développeur aux instructions connues du processeur. Les abstractions fournies par les couches du système, comme les mots d'une langue humaine, sont des outils fournis aux hommes pour « penser le logiciel », et c'est l'aptitude de ces abstractions à remplir cette tâche qui constitue leur valeur.

4.1.2 Niveaux, couches, interfaces et modèles

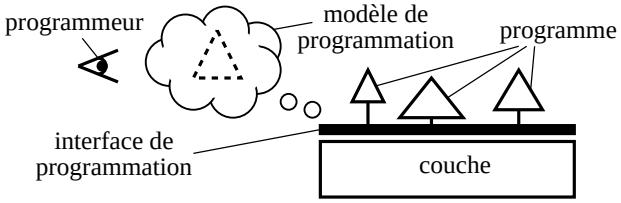


FIG. 4.1 : *Couche, interface et modèle de programmation*

Étant donné l'importance de la notion de *modèle de programmation*, nous avons choisi de l'utiliser comme grille d'analyse de la structure interne des systèmes logiciels complexes. Ce choix nous amène à distinguer trois notions, que nous avons jusqu'à maintenant silencieusement confondues pour la simplicité de notre discours (figure 4.1) :

Couche
: Une couche est un élément logiciel qui réalise un *modèle de programmation* au travers d'une *interface* de développement.

Modèle de programmation
: Un modèle de programmation est un cadre conceptuel de développement de programmes. Pour reprendre les notions introduites au chapitre 2, ce cadre propose un ensemble de *signifiés* qui constituent un espace cognitif dans lequel la pensée du développeur peut évoluer. Un « brin d'exécution » (*thread*) dans le système POSIX, une « requête » dans le modèle CORBA, un « objet » dans le langage JAVA, sont autant d'exemples de concepts utilisés pour aider à penser le logiciel.

Interface de programmation
: Une interface de programmation fournit des éléments concrèts, les *signifiants* (voir page 20), pour manipuler les concepts d'un modèle de programmation. Ces éléments concrets peuvent être les mots-clefs d'un langage, les fonctions d'une bibliothèque, les instructions d'un processeur. La fonction `pthread_create` dans POSIX, les mots-clefs

`class` ou `private` en JAVA sont des exemples de tels éléments. Ces éléments forment le socle de leurs modèles de programmation respectifs (POSIX, JAVA), et ne peuvent pas, à l'intérieur de ce modèle, être décomposés en constituants plus élémentaires (ils sont au sens éthymologique « a-tomiques », c'est à dire insécables). Ces éléments de base s'accompagnent de règles qui contraignent leur utilisation, comme la syntaxe d'un langage, ou les règles d'utilisation d'une bibliothèque[1].

Les différentes couches d'un système s'articulent au niveau de leurs interfaces de manière hiérarchique, pour former une structure étagée[2] : chaque couche utilise le modèle de programmation des couches qui la précèdent pour développer ses propres abstractions. Cette structure étagée fait ressortir des *niveaux d'abstraction* qui regroupent une interface de développement et le modèle de programmation qui lui correspond. La couche située au niveau n implémente l'interface qui sera disponible au niveau $n+1$. Pour cela, elle s'appuie sur le modèle de programmation du niveau n auquel elle se situe. Ce modèle, quant à lui, est fourni par la couche située au niveau $n-1$ (figure 4.2).

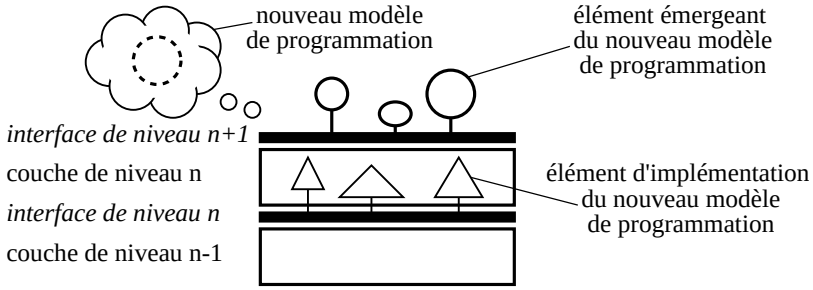


FIG. 4.2 : *Récursivité des modèles de programmation*

4.1.3 Liens inter-niveaux

À l'intérieur de la couche de niveau n « vivent » des entités d'implémentation, réalisées à partir du modèle de programmation de ce niveau. Ces entités permettent de faire émerger le nouveau modèle de programmation du niveau suivant $n+1$. Les liens que tissent les éléments du nouveau modèle du niveau $n+1$ avec les entités du niveau n qui les implémentent structurent verticalement le système. Une bonne compréhension des différentes formes de *liens* entre éléments émergeants, perçus au niveau $n+1$, et éléments d'implémentation,

[1] Il est à préciser ici, qu'un langage de programmation et une bibliothèque de classes ou de fonctions, sont, en informatique, deux choses très différentes. Notre propos ne cherche en aucune manière à les confondre, mais plutôt à souligner leur rôle commun d'un point de vue cognitif pour l'activité de programmation. Bibliothèques et langages sont l'outil mental par lequel les programmeurs *pensent* leurs programmes.

[2] Notons que la notion de « couche » est, dans notre discussion, volontairement simplifiée : comme en géologie, les « stratifications » logicielles ne sont pas en pratique forcément régulières, et présentent souvent des retournements, des décalages... Mais l'idée générale d'un escalier conceptuel, dans lequel chaque marche s'appuie sur la précédente pour présenter son propre modèle de programmation demeure.

utilisés au niveau n, s'avère donc nécessaire pour mettre en place une approche réflexive multi-niveaux.

Ces liens peuvent être abordés selon plusieurs perspectives. On peut chercher à modéliser comment les différentes entités s'agencent *structurellement* d'un niveau à l'autre pour former l'ossature du système, en termes par exemple de compositions, d'agrégations, *etc.* On peut aussi adopter un point de vue plus *comportemental*, en mettant en relief des schémas d'interaction (*interaction patterns*) entre les entités de couches différentes (cycle de vie, invocation, transfert de contrôle, de données). D'un point de vue structurel, nous avons identifié les différents types de liens inter-niveaux suivants (figure 4.3 page suivante) :

Translation
: Il y a translation lorsqu'un élément de programmation du niveau n se retrouve au niveau $n + 1$ avec une sémantique identique ou enrichie. C'est par exemple le cas pour le concept de *fonction* des langages de programmation structurés comme FORTRAN, C ou PASCAL. Le mécanisme d'appel de fonction est implémenté en utilisant les mécanismes d'empilement fournis directement par le processeur sous-jacent (registres de pile, instructions `pop` et `push`, `call`, `ret`, *etc.*).

Multiplexage
: Le multiplexage consiste à partager un élément de programmation du niveau n de manière transparente au niveau $n+1$ en multipliant par un effet d'illusion les exemplaires de cet élément. Toutes les abstractions fournies par un système d'exploitation qui « virtualisent » une ressource matérielle en donnant au-dessus de l'OS l'illusion d'une multiplicité d'exemplaires sont des exemples de multiplexage. Le partage du temps processeurs, la gestion d'espaces d'adressage virtuels (voir la section 1.2.2 page 11 à ce sujet) relèvent de cette logique.

Agrégation
: L'agrégation combine plusieurs éléments du niveau n pour fournir un nouveau paradigme, plus riche, au niveau $n + 1$. L'objectif principal des couches logicielles étant de fournir des abstractions plus faciles à utiliser, plus riches, cette forme de lien est l'une des plus courantes. Un processus UNIX par exemple agrège un (ou des) brin(s) d'exécution, dont la notion structure déjà le mode de fonctionnement d'un processeur (avec le registre compteur d'instruction, les mécanismes de pile), et un espace mémoire virtuel, fourni par la *MMU* (*Memory Management Unit*). Au niveau langage, un objet agrège des fonctions (ou plutôt des pointeurs de fonctions) et des données. Un verrou (*mutex*) agrège une opération élémentaire d'exclusion (comme une instruction *test-and-set*, ou le masquage des interruptions), et une liste d'attente.

Entités cachées
: Certaines entités d'implémentation n'apparaissent pas explicitement au niveau suivant, ni sous forme d'agrégation ni de multiplexage. Ces entités forment une partie de l'état caché du système. Parce qu'il ne les perçoit pas, elles produisent du non-déterminisme pour le programmeur du niveau suivant.

  Dans la version 2.4 du système GNU/LINUX par exemple, une bibliothèque fonctionnant en mode utilisateur, la bibliothèque `libpthread.so` est utilisée pour fournir une

interface conforme à la norme POSIX de gestion des brins d'exécution multiples. Cette bibliothèque s'appuie sur les primitives de multi-traitement du noyaux LINUX qui ne sont pas, elles, conformes à la norme. Cette bibliothèque utilise un service « caché » appelé « gestionnaire de brins » (*thread manager*) qui se charge à l'intérieur de la bibliothèque POSIX de la gestion du multitraitement « visible » à l'extérieur (synchronisation, cycle de vie, ressources). Ce « gestionnaire de brins », en fait un brin d'exécution créé à partir des primitives du noyau, reste totalement invisible à l'utilisateur de la bibliothèque.

Le *pipelining* ou les heuristiques de prédiction de branchement[3] au cœur des processeurs sont d'autres exemples de mécanismes cachés, cette fois ci « en-dessous » de la couche du langage machine.

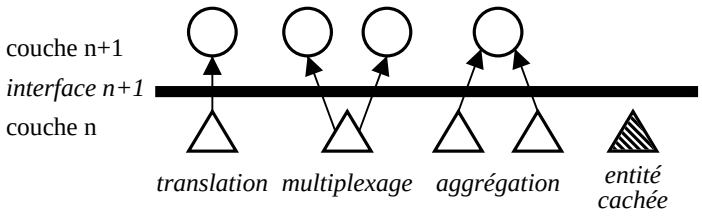


FIG. 4.3 : *Exemple de liens entre niveaux d'abstraction*

4.1.4 Utilisation des liens inter-niveaux

Nous l'avons mentionné dans l'introduction de ce chapitre, implémenter de manière transversale la tolérance aux fautes nécessite de pouvoir relier les exigences de sûreté identifiées au niveau de l'utilisateur avec les « menaces » et les moyens d'action qui sont dispersés à travers les différents niveaux du système. La notion de lien inter-niveaux que nous venons de présenter vise directement cet objectif. *Modéliser de manière réflexive les liens entre les différents niveaux d'un système permet en effet de tracer de manière transitive les relations entre des entités situées à différents niveaux d'abstraction*. Ces relations peuvent alors être utilisées pour propager de haut en bas les exigences de l'utilisateur sur l'architecture. Par exemple, supposons que les besoins de la sûreté de fonctionnement du système exigent de capturer l'état d'une entité **B** perçue par l'utilisateur au niveau de l'interface fonctionnelle (figure 4.4 page 60). Les liens inter-niveaux permettent par une fermeture transitive d'identifier récursivement les entités des différentes couches dont, potentiellement, l'état devra être pris en compte .

Inversement, en utilisant les liens inter-niveaux de bas en haut, il devient possible de mieux comprendre comment une faute logicielle survenant dans les profondeurs de l'architecture peut se propager à travers les couches du système (figure 4.5 page 60). Une

[3] Ces mécanismes d'optimisation internes aux processeurs, parce qu'ils sont cachés (au point de ne parfois pas être complètement documentés), sont un exemple typique de non-déterminisme par « masquage ». De leur fait les temps de réponse du processeur ne sont plus prédictibles, ce qui pose problème pour les applications temps-réel strict critiques. Voir en particulier [Colin & Puaut 2000] pour plus de détails.

meilleure compréhension de cette propagation facilite le choix des mécanismes à implémenter, et permet d'évaluer l'implication des décisions de conception de la tolérance aux fautes.

4.2 Empreintes et validité des algorithmes

La propagation des exigences de sûreté de fonctionnement sur l'architecture d'un système complexe permet d'identifier comment chacun des niveaux du système peut participer à la sûreté du système tout entier. Notre objectif est d'obtenir une architecture réflexive qui fournisse à chacun des niveaux ainsi identifiés les moyens de mettre en œuvre la part de tolérance aux fautes qui lui revient.

Pour cela, nous proposons dans cette section d'étudier dans un premier temps comment la tolérance aux fautes (et plus généralement tout algorithme transversal) peut s'exprimer en termes réflexifs de façon à faciliter l'adaptabilité des mécanismes ainsi réalisés. Cette discussion nous conduira à proposer la notion d'*empreinte réflexive*.

Dans un deuxième temps, nous étudierons comment la nature des informations disponibles de manière réflexive à un niveau donné peut influencer les propriétés des algorithmes qui y sont réalisés. Nous avons en effet vu à la fin du chapitre précédent que les hauts et bas niveaux d'abstraction d'un système logiciel complexe n'offraient pas du tout les mêmes capacités d'action et d'observation réflexives. Si les hauts niveaux autorisent une compréhension sémantique globale du système, ils n'offrent que peu de moyens d'action. Les bas niveaux, au contraire, s'ils permettent un contrôle très fin de la plate-forme, manquent cruellement de moyens d'analyse et de compréhension. Cette seconde discussion nous conduira à proposer la notion de *validité de la réalisation* d'un algorithme pour rendre compte de cette problématique.

4.2.1 Notion d'empreinte réflexive

Les algorithmes de tolérance aux fautes, par exemple pour la coordination d'une prise d'état distribué [Koo & Toueg 1987, Chandy & Lamport 1985, Baldoni et al. 1997], ou la réalisation d'un consensus réparti [Lamport et al. 1982], s'expriment en utilisant des modèles d'exécution « épurés ». Chandy et Lamport [Chandy & Lamport 1985], par exemple, décrivent un mécanisme de prise d'état distribué en utilisant un modèle où des *processus* communiquent de manière *asynchrone* par *messages*, et se comportent individuellement comme des *machines à états*. Ce modèle abstrait capture l'ensemble des hypothèses sur lesquelles se fonde l'exactitude (*correctness*) de l'algorithme. Un algorithme s'exprime alors relativement à ce modèle, en décrivant quelles actions exécuter en fonction d'un ensemble d'observations du système : par exemple, « *enregistrer l'état du processus X sur support stable après chaque envoi de message par X* ». Cette description s'appuie sur un ensemble de capacités d'observation (ici détecter l'envoi d'un message) et d'action (ici le déroutement de l'opération d'envoi pour effectuer la sauvegarde) qui sont supposées accessibles au « module » qui réalise l'algorithme. Ces capacités d'action et d'observation s'expriment vis-à-

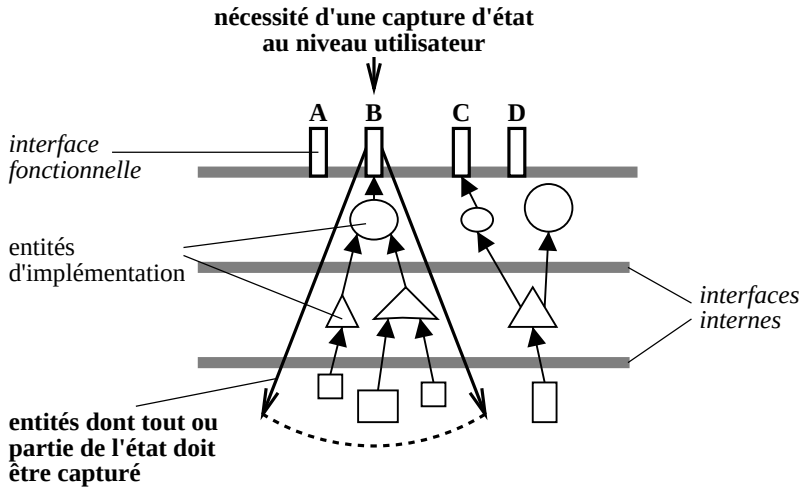


FIG. 4.4 : *Propagation d'une exigence utilisateur sur l'architecture*

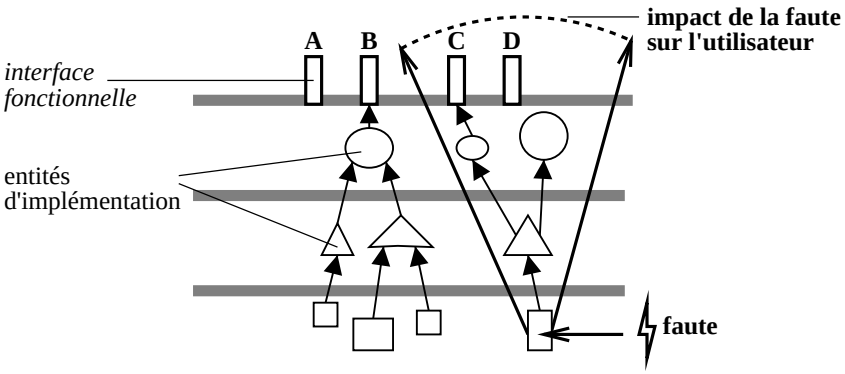


FIG. 4.5 : *Évaluation de l'impact d'une faute sur l'utilisateur*

vis d'un modèle d'exécution générique, ce qui fonde la nature transversale des algorithmes considérés, qui s'appliquent indépendamment de l'activité fonctionnelle du système sous-jacent. Cette nature transversale permet d'aborder les liens qu'entretient un algorithme avec le modèle d'exécution qui le décrit en termes réflexifs. Pour cela, nous avons choisi d'introduire la notion d'*empreinte réflexive* qui réunit les éléments suivants :

Un ensemble de capacités d'observation qui décrivent de quelle manière l'algorithme obtient des informations sur le système. Ces capacités sont parfois implicites. « *Insérer un talon contenant une estampille temporelle sur le prochain message envoyé par le processus* » suppose par exemple que l'algorithme soit notifié des messages envoyés par l'application. Ces capacités d'observation peuvent soit prendre la forme de *réification*, ou d'*introspection*, selon la terminologie introduite au chapitre 2. Les informations sur l'activité du système sont *réifiées* lorsqu'elles sont transférées sous forme d'évènements à l'algorithme, l'initiative du transfert d'information résidant alors du côté du contexte d'exécution. Dans le cas de l'introspection, les informations sont fournies à l'algorithme à sa demande explicite.

Un ensemble de capacités d'intercession permettant à l'algorithme de modifier le comportement du système, comme par exemple d'envoyer des messages particuliers, de forcer une reprise à partir d'un état déterminé, de rajouter des informations (talonnage, ou *piggybacking*) sur des requêtes distantes.

Une empreinte réflexive sert d'interface entre deux « métiers » de la tolérance aux fautes, les *algorithmiciens* et les *praticiens*. Les *algorithmiciens* proposent des solutions à des problèmes génériques en utilisant des modèles de haute abstraction, sous des hypothèses précises. Les *praticiens*, sachant qu'une solution existe à un problème posé, doivent rendre concrets les éléments du modèle développé par les *algorithmiciens* et assurer leur correction pour une plate-forme d'exécution concrète.

Les empreintes réflexives d'un ensemble de mécanismes similaires (par exemple une famille de réplications), peuvent être factorisées en une empreinte globale. Ces mécanismes similaires peuvent représenter différentes solutions au même problème, ou au contraire des solutions proches à des problèmes distincts. Leur empreinte globale réunit au sens mathématique les capacités réflexives nécessaires à l'implémentation de chacun d'entre eux. Du fait du recouvrement entre empreintes, cette empreinte globale est moins complexe que la juxtaposition de chacune de ses composantes. Par ailleurs, en permettant de retarder le choix d'un mécanisme particulier, ou de changer après coup un mécanisme déjà choisi, ce type d'empreinte réflexive rend la réalisation de la tolérance aux fautes adaptable à faible coût.

En résumé, la notion d'empreinte réflexive sépare explicitement le coeur algorithmique d'une famille de mécanismes de tolérance aux fautes (la résolution du problème) de leurs besoins opératoires (comment ces algorithmes s'enracinent dans la réalité). Elle facilite la collaboration entre algorithmiciens et praticiens, et garantit, une fois implémentée, l'adaptabilité des mécanismes choisis.

4.2.2 Modèles algorithmiques et choix d'implémentation

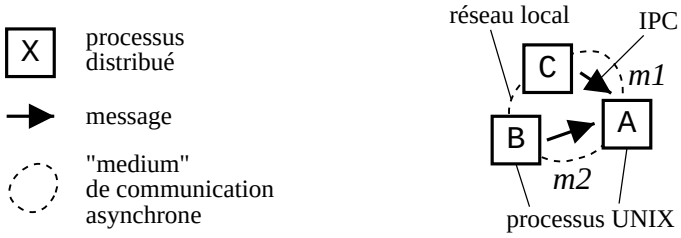
Pour utiliser en pratique une empreinte réflexive, un praticien doit établir une correspondance entre le modèle d'exécution abstrait qui la sous-tend et la plate-forme d'implémentation choisie. Cette projection de l'abstrait sur le concret doit concrétiser les éléments capturées par l'empreinte, et assurer les hypothèses nécessaires aux algorithmes en s'appuyant sur les propriétés de l'environnement d'implémentation ciblé. Si une empreinte requiert des processus distribués déterministes, les entités choisies dans le système concret pour jouer ce rôle, que ce soit des objets CORBA, des processus UNIX, des machines physiques, ou des objets C++, doivent elles aussi respecter cette hypothèse[4]. Si la modélisation requiert de modifier les messages entre entités distribuées, une technique d'interception doit être mise en place pour ces messages, que ce soit des requêtes CORBA, des invocations locales, ou des communications inter-processus.

Dans les systèmes complexes multi-couches, cet exercice de correspondance n'est pas univoque, et laisse une grande liberté pour façonner la « granularité d'implémentation » d'une empreinte réflexive : les systèmes complexes présentent en effet une structure « fractale ». Les mêmes schémas d'interaction se retrouvent à plusieurs niveaux du système, de manière emboîtée. Un même algorithme peut donc trouver à être implémenté de différentes façons, ces différents choix entraînant des propriétés différentes en termes de flexibilité, de performance, et de couverture des hypothèses. Revenons par exemple à la notion de « processus distribué » utilisée dans [Chandy & Lamport 1985], [Koo & Toueg 1987] ou [Baldoni et al. 1997], et considérons un contexte d'implémentation comprenant des applications écrites en C++ s'exécutant sur des machines UNIX en réseau. Il existe dans ce contexte plusieurs options pour « projeter » la notion de « processus distribué » sur un élément effectif du système. Une correspondance naturelle consiste par exemple à assimiler les « processus distribués » de l'algorithme aux processus UNIX des différentes applications, et les communications par messages aux interactions inter–processus (IPC) du standard POSIX (figure 4.6-b page suivante).

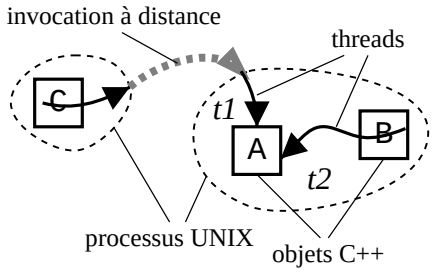
Mais il est aussi possible de choisir une granularité de « projection » plus fine entre l'algorithme et son contexte d'implémentation : dans [Kasbekar et al. 1999], les auteurs assimilent les objets du langage C++ à des « processus distribués ». Les appels de méthodes entre objets deviennent alors les messages d'un système réparti « microscopique » constitué des objets C++ de l'application (figure 4.6-c page ci-contre). Les brins d'exécution qui circulent d'un objet à l'autre forment alors une sorte de *médium de communication* par lequel les objets s'échangent des informations[5]. Cette correspondance entre « objets–langage » et « entités distribuées » est possible en supposant chaque objet protégé par un verrou exclusif :

[4] Alternativement, ces hypothèses peuvent aussi être garanties en utilisant des algorithmes supplémentaires, dont les empreintes réflexives doivent alors être « projetées » à leur tour. La démarche de projection d'un modèle algorithmique abstrait devient alors récursive, la récursion se terminant quand les capacités et hypothèses que l'on cherche à réaliser sont directement garanties par le substrat d'exécution.

[5] Notons que ce n'est pas la seule interprétation possible d'un système à mémoire partagée (ce que sont en fait ici les objets). Baldoni, Hélary et Raynal [Baldoni et al. 1998a] proposent par exemple une modélisation dans laquelle les brins d'exécution sont les entités distribuées, et la mémoire partagée (ici nos objets) forment le médium de communication.



(a) Modèle algorithmique (b) Vision macroscopique



(c) Vision microscopique

FIG. 4.6 : *Diversité des correspondances entre algorithme et implémentation*

les brins du programme transportent de l'information (paramètres d'appel et de retour) d'un objet C++ à l'autre en accédant séquentiellement à chaque objet. Leur action est alors comparable aux messages d'un réseau asynchrone : lorsque plusieurs messages sont envoyés de manière asynchrone au même destinataire, le réseau détermine de manière « non-déterministe » leur ordre d'arrivée. De la même façon, lorsque plusieurs brins tentent d'accéder de manière concurrente au même objet, leur ordre de passage est déterminé de manière « non-déterministe » par l'ordonnanceur de l'OS.

Cette approche, en descendant le niveau structurel auquel s'exécute un algorithme permet une gestion plus fine de l'état d'une application. En effet, en traçant les dépendances causales au niveau des objets langage et non plus des processus système, elle permet de limiter, lors d'une reprise sur défaillance, la propagation des recouvrements arrière dans le système. Par exemple, sur la figure 4.6-c, si le processus contenant l'objet `C` défaille de sorte à devoir annuler les interactions entre `C` et `A`, l'algorithme de Kasbekar *et al.* permet de reprendre l'exécution de `A` à partir d'une capture d'état antérieure, sans devoir nécessairement modifier l'état de l'objet `B`. De cette manière, l'algorithme recouvre de façon *partielle* et *sélective* l'état du processus contenant `A` et `B`.

La capacité des algorithmes de tolérance aux fautes à s'adapter à des « granularités d'implémentation » très différentes est fortement liée à une propriété de « robustesse » vis-à-vis de leurs modèles d'exécution, que nous abordons plus en détail dans la section suivante.

4.2.3 Granularité du contrôle et validité des algorithmes

Il peut arriver que certaines capacités réflexives soient particulièrement difficiles, voire impossibles, à réaliser à un niveau d'abstraction donné, pour des raisons de coût d'instrumentation (si la capacité n'est pas fournie par la plate-forme par défaut), de performance, techniques, *etc.* Dans la pratique, il existe donc toujours des approximations (explicites ou implicites) dans la réalisation des capacités d'action et d'observation nécessaires à l'implémentation d'un algorithme, qui pourtant ne remettent pas en cause la validité de celui-ci.

Dit autrement, la plupart des algorithmes de tolérance aux fautes restent valides même lorsque ce qu'ils perçoivent de la réalité n'est qu'une approximation imparfaite de l'état réel du système.

Nous avons vu au chapitre précédent comment la nature des informations accessibles à l'intérieur d'un système dépendait du niveau d'abstraction considéré. Pour développer une approche multi-niveaux de la tolérance aux fautes, il nous a paru nécessaire d'analyser ici comment des capacités réflexives « imparfaites » pouvait permettre d'implémenter des algorithmes de tolérance aux fautes correctes (ou comment un algorithme pouvait *raisonner juste* tout en ayant une *perception fausse*). Nous commençons par définir un peu plus formellement ce que nous entendons par approximation et validité, puis nous illustrons cette notion de « robustesse à l'approximation » en discutant des protocoles de capture d'état distribué.

Approximation et validité : définitions

Cette « robustesse » des algorithmes vis-à-vis de leur empreinte réflexive est représentée sur la figure 4.7 page suivante. Sur cette figure, un algorithme est vu comme une « fonction[6] » qui pour un ensemble d'observations données (son entrée) déclenche un ensemble d'actions sur le système (sa sortie).

Cette figure fait apparaître deux espaces : l'espace E de ce que peut percevoir un algorithme de tolérance aux fautes pour un problème donné (espace problème), et l'espace des solutions S que peut proposer cet algorithme à ce problème. La figure illustre comment un algorithme transforme un élément de l'espace problème e (ce qu'il perçoit du problème) en un élément de l'espace des solutions $f(e)$(la solution que propose l'algorithme au problème). Pour que l'algorithme soit correct, il faut, et il suffit, que la solution $f(x)$ construite par l'algorithme pour un état x du système fasse partie des solutions valides pour x, $V(x)$ (par exemple sur la figure 4.7 page ci-contre, $f(e) \in V(e)$). Si nous supposons maintenant que l'algorithme n'a en fait qu'une vision imparfaite de l'état du système, cela signifie qu'au lieu de travailler sur l'état réel (e sur la figure), l'algorithme travaille sur l'état qu'il perçoit (d), qui n'est qu'une *approximation* de la réalité.

[6]Pour simplifier nous supposons l'algorithme déterministe : à une entrée l'algorithme fait correspondre une seule sortie ; mais notre discussion serait tout à fait généralisable à un algorithme « non-déterministe ».

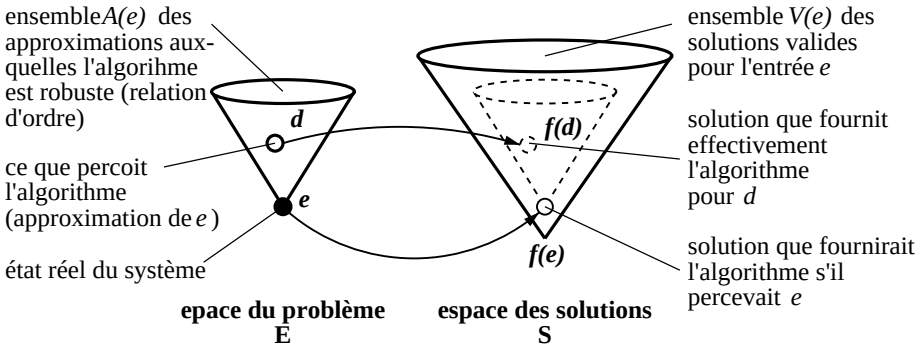


FIG. 4.7 : *« Robustesse » d'un algorithme à l'approximation*

L'on peut modéliser le fait qu'un état (perçu) en approxime un autre (réel) par une relation d'ordre « $\succ$ », que nous appellerons *relation d'approximation*. « $d \succ e$ » signifie que l'état d « approxime » l'état e. Un algorithme est « robuste » à l'approximation, s'il est possible de trouver une telle relation d'ordre, telle que si d « approxime » e, alors la solution $f(d)$ fournie par l'algorithme pour l'*état approximé* d est aussi une solution pour l'*état réel* e : $f(d) \in V(e)$. Dis plus formellement :

$$\forall (e,d) \in E \times E : d \succ e \Rightarrow f(d) \in V(e) \tag{4.1}$$

Un exemple : les protocoles de capture d'état

Les protocoles de capture d'état distribué que nous avons mentionnés dans la section 3.1.1 page 37 sont un exemple de protocoles « robustes » à l'approximation. Ces protocoles considèrent un scénario (ou trame) de communication par messages entre un ensemble de processus distribués (figure 4.8 page suivante), et permettent à ces processus de coordonner leurs prises d'état locales pour éviter l'effet domino. Selon le type de protocole, les processus peuvent être autorisés à capturer leur état de manière spontanée (c'est le cas sur la figure 4.8). Le protocole permet alors de « forcer » les différents processus distribués à capturer leur état à certains moments de leur exécution pour empêcher l'effet domino (figure 4.9) (on parle alors de protocole de capture d'état distribué *induit par les communications* [Baldoni et al. 1997]).

Dans ce type de protocoles, le scénario de communication (quels messages, dans quel ordre, entre quels processus), et, éventuellement, la position des captures d'état spontanées, constituent l'entrée de l'algorithme (sa « perception » [7]). Les captures d'état imposées par l'algorithme pour éviter l'effet domino constituent sa sortie. Or ces protocoles présentent la

[7] Il est en fait abusif de dire que l'algorithme *perçoit* le scénario de communication, puisque l'algorithme, pour être implémentable, ne peut utiliser qu'une connaissance effectivement disponible dans le système (ce qu'on appelle le *usable knowledge* en anglais). Notamment, dans un système asynchrone, un algorithme ne peut pas utiliser des informations disponibles à des locations distinctes sans réunir explicitement, par des communications, ces

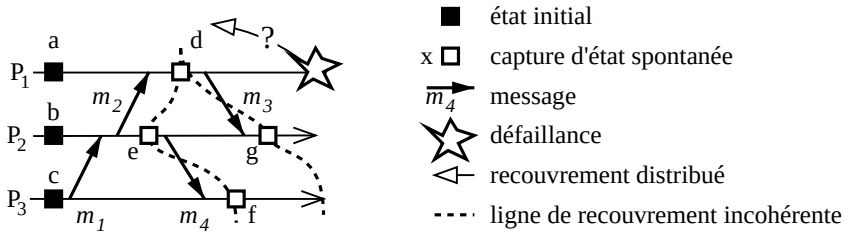


FIG. 4.8 : *Entrée d'un protocole de capture d'état distribué*

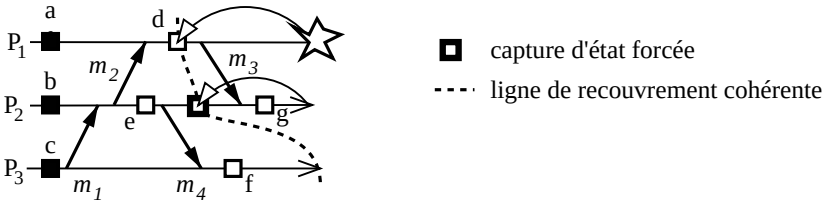


FIG. 4.9 : *Sortie d'un protocole de capture d'état distribué*

propriété suivante : si le protocole travaille sur un scénario $S1$ qui sur-approxime les interactions entre processus distribués (c'est-à-dire qui contient *plus* de messages) par rapport au scénario $S2$ des interactions *réelles* entre processus (figure 4.10 page suivante), alors toute solution qui élimine l'effet domino pour $S1$ (l'approximation), l'élimine pour $S2$ (la réalité). Nous retrouvons ici la propriété exprimée par l'équation 4.1 page précédente.

La capacité des protocoles de capture d'état distribué à fournir une solution valide à partir d'une perception imparfaite de la réalité ne provient pas tant des algorithmes eux-mêmes, que du problème à résoudre. Le problème à résoudre (éviter l'effet domino) revient en effet à empêcher l'existence de « chemins en zigzag » qui ne contiennent pas de capture d'état, et ne soient pas causalement doublés par une séquence de messages [Baldoni et al. 1998b, Netzer & Xu 1995]. Dans l'exemple de la figure 4.8, le chemin correspondant aux événements *[envoi de m_3]→[réception de m_3]→[envoi de m_4]→[réception de m_4]* (figure 4.11 page ci-contre) est un exemple d'un tel chemin en zigzag.

Ce type de chemin induit des dépendances non–causales entre *intervalles de recouvrement*. Un intervalle de recouvrement est l'intervalle entre deux captures d'état successives par un même processus. Ces intervalles déterminent la granularité de ce qu'un processus peut annuler « d'un bloc ». Par exemple sur la figure 4.11 page suivante, du fait du chemin en zigzag, si P_1 annule l'intervalle $[d, -]$, alors P_3 doit aussi annuler l'intervalle $[c, f]$. Pourtant, aucun des évènements du comportement de P_3 entre c et f ne dépend causalement de ce qu'à fait P_1 après d. Cette propagation des annulations entre intervalles n'est donc pas due à des contraintes de cohérence globale du fait de messages échangés. Elle provient d'une granularité de recouvrement permise par les captures d'état spontanées trop grossière.

informations en un même endroit. Nous n'avons pas ici la place pour détailler cet aspect mais nous renvoyons le lecteur intéressé à par exemple [Baldoni et al. 1998b] pour un exposé plus explicite.

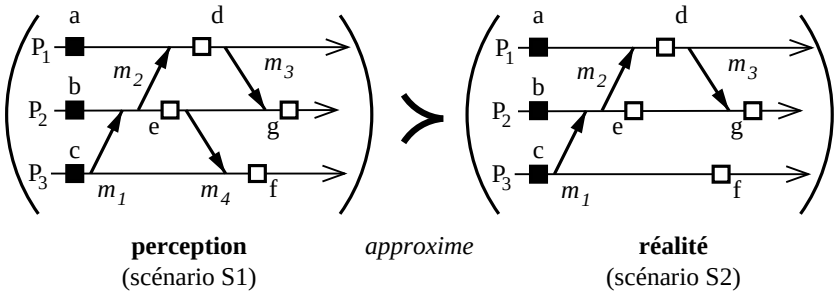


FIG. 4.10 : *Approximation entre deux scénarios de communication*

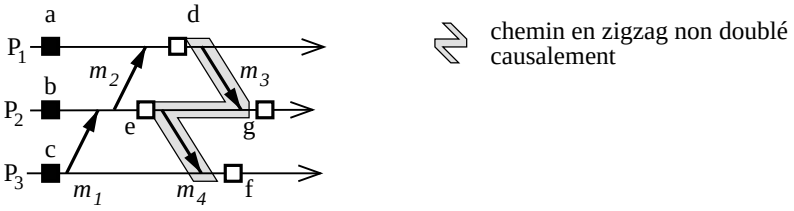


FIG. 4.11 : *Pourquoi l'approximation fonctionne : les chemins en zigzag*

Autrement dit, les captures d'état spontanées ne « quadrillent » pas suffisamment l'ensemble des messages échangés pour permettre de n'annuler *que* les intervalles qui dépendent *causalement* de la défaillance en train d'être recouverte.

Un protocole de capture d'état distribué empêche la création de tels chemins en zigzag en les cassant par des captures d'état forcées (sur la figure 4.9 page ci-contre entre les captures e et g du processus P_2) : on comprend alors qu'en percevant plus de messages qu'il n'en existe réellement, un protocole utilise une vision *pessimiste* de l'exécution du système. Pour reprendre les scénarios de communication $S1$ et $S2$ de la figure 4.10, en éliminant les chemins en zigzag du scénario $S1$, qui sur-approxime les interactions réelles, tout protocole est assuré d'éliminer ceux de $S2$, qui correspond à l'exécution réelle du système, puisque $S2$ ne peut contenir d'autres chemins en zigzag que ceux déjà présents dans $S1$.

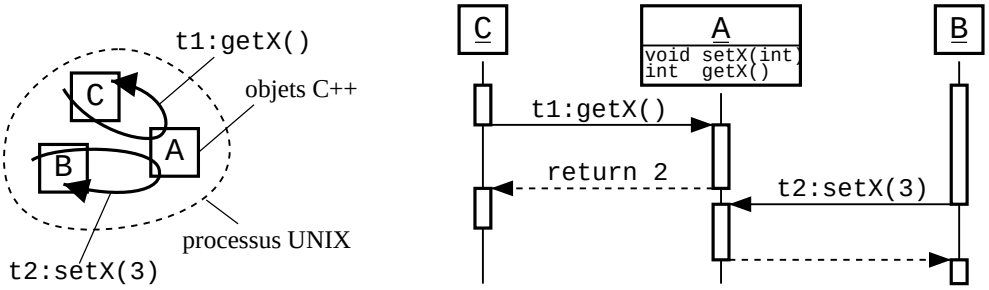
4.2.4 Défaut de perception sur un exemple pratique

Dans la pratique, la possibilité de pouvoir résoudre certains problèmes de tolérance aux fautes en utilisant une perception imparfaite de la réalité est particulièrement intéressante.

Elle permet en effet de réaliser des mécanismes de tolérance aux fautes à partir de capacités d'observation peu coûteuses, qui ne fournissent qu'une vison « floue » de l'état réel du système. Mais parce que cette vision est « floue », elle oblige à une approche pessimiste, comme par exemple, dans le cas d'un protocole de capture d'état, de déduire des dépendances causales *potentielles* qui n'ont pas nécessairement lieu d'être. L'utilisation d'un point de vue pessimiste ne remet pas en cause la validité du mécanisme ainsi réalisé, mais

rend la solution obtenue peu optimale. Nous revenons ici sur les travaux de Kasbekar *et al.* [Kasbekar et al. 1999] dont nous tirons un exemple pratique (voir page 62). L'approche de Kasbekar *et al.* revient en effet à *surestimer* les relations de causalité entre brins d'exécution et objets pour assurer la cohérence d'une application orientée objet sans induire de coûts d'observation trop élevés.

Considérons une exécution d'une application C++, dans laquelle deux brins d'exécution, t_1 et t_2, interagissent avec trois objets `A`, `B`, et `C` (figure 4.12). Le brin t_1 invoque la méthode `A.getX()` depuis l'objet `C`, le brin t_2 la méthode `A.setX(3)` depuis l'objet `B`[8]. `A.getX()` est un *accesseur*, c'est-à-dire une méthode qui renvoie des informations sur l'état de l'objet `A` sans le modifier. `A.setX(int)` est un *modifieur*, c'est-à-dire une méthode qui modifie `A`, mais ne donne pas d'information supplémentaire sur son état (type de retour `void`).



(a) Vision « distribuée » (b) Diagramme de séquence

FIG. 4.12 : *Deux brins d'exécution interagissent avec un objet*

Kasbekar *et al.* s'intéressent à la situation dans laquelle un des objets de l'application, par exemple ici l'objet `C`, doit être ramené dans un état antérieur pour annuler une interaction avec l'extérieur (figure 4.13). Le problème est alors le suivant : comment modifier l'état de `C` en conservant la *cohérence* du système et en ne modifiant pas, si possible, les autres objets `A` et `B`.

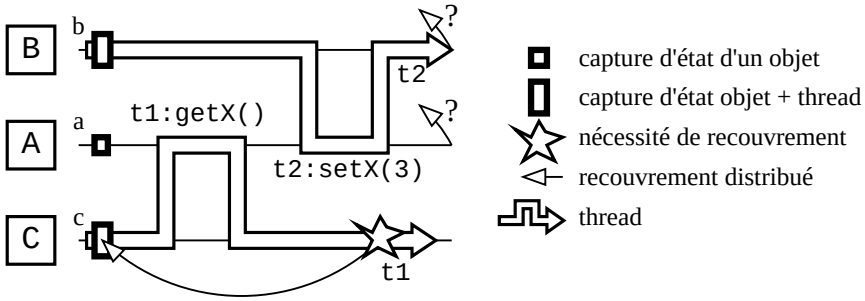


FIG. 4.13 : *Problème de la cohérence entre objets à brins d'exécution multiples*

[8]Rappelons que [Kasbekar et al. 1999] impose que toutes les opérations sur un objet soient sérialisées par un verrou.

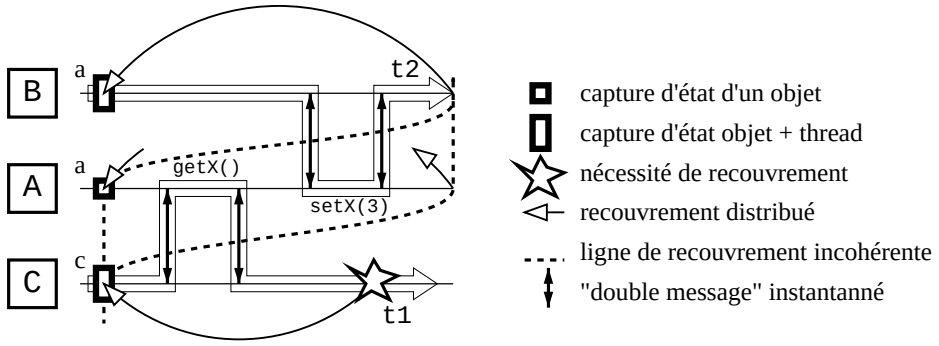


FIG. 4.14 : *Une approximation des interactions brin / objet [Kasbekar et al. 1999]*

L'approche proposée revient à interpréter l'invocation d'une méthode entre objets comme un échange d'information (figure 4.14) : lorsque qu'un objet `X` invoque un objet `Y`, de l'information provenant de `X` (les paramètres d'appel) est (potentiellement) transportée vers `Y`. Lorsque l'invocation retourne de l'objet appelé `Y` vers l'objet appelant `X`, de l'information provenant de `Y` transite potentiellement vers `X`.

Les auteurs ajoutent dans ces deux cas (invocation et retour d'invocation), une double dépendance causale (c'est la raison des doubles flèches sur la figure 4.14), pour éviter d'avoir à gérer des « invocations en transit » lors de recouvrements. En effet, dans un système réparti communiquant par messages, un état global cohérent du système peut contenir des messages qui ont été envoyés mais n'ont pas encore été reçus : ce sont des messages *en transit*, dont nous avons déjà parlé dans la section 3.1.1 page 38. Si le système de communication est supposé fiable, le mécanisme de recouvrement doit assurer la restauration de ces messages en transit. Dans un système à plusieurs brins d'exécution, les invocations et les retours d'invocation jouent le rôle des messages, et peuvent aussi être *en transit* (lorsqu'un brin est suspendu juste après avoir fait un appel ou un retour d'appel). Ces invocations et retour d'invocation « en transit » ne peuvent non plus être « perdus » du fait d'un recouvrement, et doivent donc être « réinjectés ». Cette réinjection est cependant plus délicate que dans un « vrai » système communiquant par messages, les invocations réalisées par un même brin ne pouvant être abordées séparément. Le choix de Kasbekar *et al.* d'utiliser des doubles dépendances causales garantit qu'aucune invocation « en transit » ne puisse être présente dans un état global cohérent du système, et permet donc d'écarter ce problème de réinjection.

Un état global du système est en effet cohérent si la ligne de recouvrement correspondante ne traverse aucun de ces doubles messages. Sur la figure 4.14 par exemple, aucune des lignes de recouvrement représentées n'est cohérente, et le retour de l'objet `C` dans l'état c impose de ramener `A` et `B` dans leurs états respectifs a et b. Lors d'une telle restauration d'état, les brins d'exécution sont considérés comme « un médium de communication » dont l'état doit être restauré en cohérence avec celui des objets. L'interprétation pessimiste de toute invocation comme un double échange d'information assure que le point de restauration

d'un brin soit toujours univoque (dans quel objet, avec quel état) (sur la figure 4.14, le brin t_2 est restauré dans l'objet `B` et le brin t_1 dans l'objet `C`).

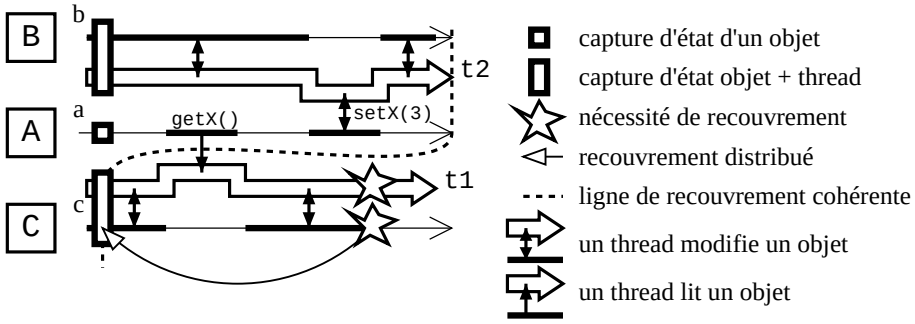


FIG. 4.15 : *Interprétation plus précise des interactions brin / objet*

Cette stratégie conduit cependant à surestimer les relations de causalité entre les objets de l'application, et donc à utiliser une vision relativement restrictive de la cohérence du système. Nous proposons avec la figure 4.15 une analyse plus précise des interactions entre brins d'exécution et objet, en nous inspirant de [Baldoni et al. 1998a]. Dans cette analyse, les brins sont représentés comme des entités à part entière, et non plus comme un simple médium de communication. Un brin qui traverse un objet peut potentiellement modifier l'état de cet objet, mais ne le fait pas nécessairement. Dans tout les cas, le brin garde en mémoire la traversée, même s'il ne lit aucune information présente dans l'objet (du fait de l'évolution de son compteur d'instruction et de sa pile). Nous distinguons donc dans cette analyse deux formes d'interaction entre un objet et un brin d'exécution :

Lecture
: Le brin emmagasine dans son état des informations provenant de l'objet. L'objet n'est pas modifié. Après cette opération, l'état du brin dépend causalement de celui de l'objet, mais l'état de l'objet reste inchangé. Il est donc possible d'annuler la lecture du point de vue du brin, sans avoir à propager l'annulation à l'objet (puisque aucune information n'est passée du brin à l'objet). En revanche, le retour de l'objet dans un état antérieur à l'opération de lecture oblige à annuler la lecture pour le brin, et donc à propager le recouvrement au brin.

Modification
: Le brin modifie l'état de l'objet. Même si le brin ne lit aucune information de l'objet, son état est modifié (compteur d'instructions) du fait de l'opération. Il y a double dépendance causale.

L'opération `get(X)` étant une opération de lecture, cette approche permet de conclure que les états des objets `A` et `B` ne dépendent pas causalement de l'évolution de `C` après la capture de son état en c. Il n'est donc pas nécessaire de propager le recouvrement de l'objet `C` aux objets `A` et `B`.

Cette seconde approche est donc plus efficace que celle originale de Kasbekar *et al.* [Kasbekar et al. 1999]. Sa mise en œuvre est cependant plus difficile. Elle requiert en effet une compréhension plus poussée de la sémantique du programme, pour pouvoir distinguer les méthodes selon qu'elles modifient ou non leurs objets. Cette corrélation entre efficacité et compréhension sémantique se retrouve dans d'autres domaines de la tolérance aux fautes, tous plus ou moins liés à la notion de cohérence et de causalité. Si nous revenons par exemple au problème de la capture d'état d'une entité distribuée (section 3.1.1 page 37), une capture d'état doit, comme nous l'avons expliqué, « englober » suffisamment d'information pour assurer la cohérence d'une reprise à partir de cet état pour un observateur extérieur. Capturer « plus » d'information que nécessaire ne remet pas en cause la validité de l'état et continue d'assurer la cohérence. Ce surplus d'information pose cependant des problèmes d'efficacité, des informations inutiles devant être capturées et restaurées. À l'inverse, cerner exactement quelle « quantité » minimale d'information suffit à assurer une reprise cohérente demande une compréhension sémantique très poussée de l'activité du système.

4.2.5 Conclusion sur l'implémentation réflexive des algorithmes

Nous venons de voir pourquoi la mise en œuvre de mécanismes de tolérance aux fautes dans les systèmes complexes laissait ouverts de nombreux choix d'implémentation. Cette liberté provient d'une part de la nature « fractale » des systèmes complexes, qui permet de « projeter » les modèles algorithmiques à différents niveaux d'un système, et d'autre part de la « robustesse » des algorithmes qui autorise l'utilisation de capacités de réification et d'introspection « imparfaites ».

Usuellement, ces capacités « imparfaites » permettent d'instrumenter un système à faible coût en ne considérant que les bas niveaux d'une architecture. Elles permettent ainsi de mettre à profit la « robustesse » des algorithmes sous-jacents pour faciliter l'implémentation de la tolérance aux fautes. Cependant, elles perdent aussi en performance et en flexibilité : les mécanismes ainsi réalisés fonctionnent en effet de façon sub-optimale sur une vision pessimiste du système. Cette perte de performance peut même aller jusqu'à remettre en cause la faisabilité de ces approches de bas niveau [Napper et al. 2003].

Notre proposition dans cette thèse est d'utiliser la compréhension sémantique apportée par l'analyse des liens entre les niveaux du système (section 4.1.4 page 58) pour réaliser des capacités réflexives plus précises et plus puissantes, et ainsi permettre une implémentation plus efficace de la tolérance aux fautes. Une telle compréhension requiert de combiner les informations présentes à différents niveaux d'abstraction, pour tirer partie de la complémentarité entre les niveaux que nous avons observée dans la section 3.3 page 51. Dans la section qui suit nous proposons une démarche de développement pour atteindre cet objectif.

4.3 Une démarche de développement multi-niveaux

Nous avons cherché, dans les sections précédentes, à relier les considérations *algorithmiques* de la tolérance aux fautes (capacités d'action et d'observation) aux contraintes *architecturales* des systèmes complexes (transparence, encapsulation, abstraction, niveaux). Notre effort nous a amenés à proposer un méta-modèle des architectures en couches (section 4.1 page 54), puis à introduire la notion d'*empreinte réflexive* pour capturer les besoins opératoires de la tolérance aux fautes (section 4.2.1 page 59) et enfin à discuter de la « projection » d'une empreinte sur une architecture concrète (section 4.2.2 page 62).

Ce travail nous permet de proposer ici une démarche de développement d'une architecture réflexive multi-niveaux pour la tolérance aux fautes qui intègre l'ensemble de ces notions en une série d'étapes successives (figure 4.16). Plus précisément, cette démarche permet d'étendre une architecture non-réflexive en lui adjoignant une méta-interface multi-niveaux qui cible une famille d'algorithmes de tolérance aux fautes. La démarche est en quatre étapes, qui s'organisent en deux phases principales. La première phase consiste à analyser les aspects algorithmiques de la famille de mécanismes choisie en construisant leur empreinte réflexive (étape 1), puis à analyser l'architecture cible en étudiant les liens structurels et comportementaux entre ses niveaux (étape 2). Fort des ces deux analyses, la deuxième phase consiste à *projeter* l'empreinte de l'étape 1 sur le modèle multi-niveaux de l'étape 2 (étape 3), puis à utiliser cette projection pour identifier les points d'instrumentation optimaux dans l'architecture choisie, afin de réaliser une méta-interface qui remplisse les exigences capturées dans l'empreinte réflexive (étape 4).

1. *Obtenir l'empreinte réflexive d'un ensemble d'algorithmes de tolérance aux fautes.*
2. *Construire un méta-modèle de l'architecture en couches choisie en analysant les liens entre les entités des différents niveaux.*
3. *Projeter l'empreinte de l'étape 1 sur l'architecture du système, en tirant parti du méta-modèle multi-niveaux ainsi obtenu. Notamment la compréhension des interactions entre couches doit permettre d'implémenter les capacités réflexives recherchées en combinant de manière optimale des informations d'abstractions hétérogènes obtenues à différents niveaux du système.*
4. *Implémenter chacune des capacités réflexives de l'empreinte dans la ou les couches les plus appropriées en utilisant les résultats de la projection.*

FIG. 4.16 : *Démarche de développement pour la réflexivité multi-niveaux*

C'est cette démarche que nous illustrons dans le chapitre qui suit pour aborder la réplication d'une plate-forme GNU/LINUX – CORBA à brins d'exécution multiples.

4.4 Conclusion

Dans ce chapitre, nous avons étudié comment la réflexivité pouvait être utilisée pour la mise en œuvre de la tolérance aux fautes dans les systèmes logiciels complexes en dépit des limitations que nous avions identifiées à la fin du chapitre 3, page 50. Dans une première étape nous nous sommes intéressés à l'organisation multi-niveaux des systèmes complexes, pour mieux maîtriser la complémentarité entre les niveaux d'abstraction, que nous avions constatée au chapitre précédent, page 51. Cette discussion nous a amenés à souligner l'importance des *liens inter-niveaux* pour la compréhension de l'organisation d'un système logiciel complexe. Dans une seconde étape, nous avons cherché à mieux cerner les besoins opératoires de la tolérance aux fautes en introduisant la notion d'*empreinte réflexive*. Cette nouvelle notion nous a permis de discuter de façon générale des choix d'implémentation possibles lors de la réalisation de mécanismes de tolérance aux fautes dans une architecture multi-niveaux.

Les notions et les enseignements tirés de cette étude nous ont finalement permis de proposer une démarche en quatre étapes pour le développement d'architectures réflexives multi-niveaux dédiées à la tolérance aux fautes des systèmes complexes. C'est cette démarche que nous proposons maintenant d'illustrer sur un exemple concret (GNU/LINUX et CORBA), dans le chapitre suivant, qui termine notre mémoire.

Chapitre 5

Application à la réplication

Nous illustrons dans ce chapitre la démarche de développement d'une *méta-interface multi-niveaux* pour la tolérance aux fautes que nous avons proposée à la fin du chapitre précédent. Nous commençons par les aspects les plus génériques de cette approche, en illustrant la notion d'*empreinte réflexive* pour une famille bien connue de mécanismes de réplication (étape 1 de notre démarche).

Nous étudions ensuite comment les problèmes de contrôle du non-déterminisme et de capture d'état soulevés par l'empreinte obtenue peuvent être abordés sur une architecture CORBA/POSIX à brins d'exécution multiples (*multithreaded*) en utilisant la notion de *lien inter-niveaux* du chapitre précédent (étapes 2 et 3 de notre démarche).

Pour finir, nous validons notre approche par la réalisation d'un prototype de méta-interface sur une plate-forme à caractère industriel (GNU/LINUX et ORBACUS), ce qui nous permet d'aborder la problématique de l'extraction d'un méta-modèle à partir des sources d'un composant, pour laquelle nous proposons une démarche générique (étape 4 de notre démarche).

5.1 Empreinte réflexive de la réplication

Nous nous intéressons dans cette section à la famille de mécanismes de réplication présentés dans le projet DELTA-4 [Powell 1991], dont nous construisons l'empreinte réflexive, conformément à l'étape 1 de la démarche proposée au chapitre précédent. Le projet DELTA-4 décrit trois stratégies de réplication d'une application client/serveur communiquant par messages : la *réplication active*, la *réplication passive* et la *réplication semi-active*. Ces trois approches ont en commun de tirer profit de la redondance acquise lorsqu'une même application est répliquée dans un système réparti pour masquer ses défaillances à l'utilisateur. Leurs

différences tiennent aux hypothèses faites sur le comportement de chacune des répliques, à la nature des fautes tolérées, à leurs difficultés respectives d'implémentation, et enfin aux sur–coûts induits en traitement et en communication.

5.1.1 Présentation des mécanismes étudiés

En *réplication active*, les différentes répliques ont des rôles symétriques, et traitent toutes les requêtes simultanément. Ce mode de réplication nécessite d'une part que les mêmes messages d'entrée soient fournis dans le même ordre à toutes les répliques (*cohérence des entrées*) ; et d'autre part que, partant du même état initial, les répliques produisent les mêmes messages de sortie, aussi dans le même ordre (*déterminisme des répliques*). Lorsque ces hypothèses sont respectées, la réplication active autorise un recouvrement très rapide des défaillances, puisque chacune des répliques peut à tout moment assumer seule le service répliqué. La réplication active permet aussi de tolérer les fautes en valeur d'une minorité de répliques par la mise en place d'un système de comparaison et de vote sur les réponses produites. On s'assure ainsi qu'une réponse valide sera toujours retournée à l'utilisateur. Le prix à payer pour ces différentes propriétés reste néanmoins une forte utilisation des ressources matérielles, tous les traitements devant être répliqués en permanence.

La *réplication passive* élimine ce fort sur-coût de traitement en introduisant une dissymétrie dans le rôle des répliques. Une seule réplique, la réplique *primaire* ou *primary*, traite les requêtes en entrée et produit des sorties. Les autres répliques, les répliques de secours ou *backups*, sont maintenues synchronisées avec la réplique primaire par un transfert périodique de captures d'état (*checkpointing*). Le choix des instants de capture est déterminant pour assurer un masquage des défaillances complet à l'utilisateur. Il s'agit en fait du même problème que celui à la base des protocoles de capture (*checkpointing protocols*), que nous avons présentés à la page 65 : les informations d'état transmises par la réplique primaire doivent permettre en cas de défaillance un basculement transparent du service sur les répliques secondaires. Dit autrement, les répliques secondaires doivent pouvoir assurer, à partir des captures d'état, une reprise qui soit cohérente avec le comportement perçu jusqu'alors par les clients du service répliqué (voir la section 3.1.1 page 37 sur le même sujet). En plus de ses avantages en termes de sur-coût de traitement en mode sans défaillances, la réplication passive, au contraire de la réplication active, ne requiert pas le déterminisme de l'application à laquelle elle s'applique. Cependant, les transferts de capture peuvent nécessiter des coûts de communication élevés, et les temps de recouvrement en cas de défaillance de la réplique primaire sont souvent plus longs, du fait de traitements qui n'étaient pas compris dans la dernière capture d'état et qui doivent être ré–effectués. Enfin, la réplication passive ne permet pas de tolérer les fautes en valeur, une seule réplique produisant des réponses.

La troisième forme de réplication, la *réplication semi-active*, a pour objectif de lever la contrainte du déterminisme qui vaut pour la réplication active. Elle ajoute des mécanismes de « transfert », inspirés de la réplication passive, au mécanisme classique de la réplication active pour assurer un consensus entre les différentes répliques vis-à-vis de leurs décisions non-déterministes. Une réplique est choisie comme « *meneuse* » et est responsable

Stratégie	**Fautes tolérées**	**Déterminisme**	Sur-coût		
			Traitement	**Communication**	**Recouvrement**
active	fautes incontrôlées	requis	important	faible	faible
passive	silence sur défaillance	non–requis	faible	important	important
semi-active	fautes incontrôlées	non–requis	important	faible	faible

TAB. 5.1 : *Propriétés et hypothèse des stratégies de réplication considérées*

des décisions non-déterministes de l'application (ordonnancement des brins d'exécution, des messages, événements asynchrones) qui sont alors communiquées sous forme de notifications aux autres répliques, « *suiveuses* ». La réplication semi-active combine la plupart des avantages des deux autres formes de réplication : ses sur-coûts de communication et de recouvrement sont faibles, comme dans la réplication active, elle ne requiert pas le déterminisme, comme la réplication passive. De plus, en supposant un protocole de diffusion fiable entre meneur et suiveurs d'une part, et en supposant que les suiveurs soient capables de rejeter les choix invalides du meneur d'autre part, alors la réplication semi-active supporte aussi les fautes en valeur. Nous renvoyons aux sections 6.7 et 7.6 de [Powell 1991] pour une discussion plus détaillée de cette dernière propriété. Les différentes propriétés et les hypothèses sur lesquelles s'appuie chacune des stratégies sont reprises sous une forme résumée dans la table 5.1.

Les trois formes de réplication ainsi présentées nécessitent la mise en place de mécanismes de détection d'erreur et de configuration, notamment pour relancer une réplique, ou mettre à jour la vision qu'ont du système les différentes répliques (protocole de groupe). Nous nous concentrons dans la suite de ce chapitre sur le problème de la capture d'état et de la maîtrise du non-déterminisme, et n'aborderons donc pas plus en détail ces aspects, qui restent essentiels cependant.

5.1.2 Empreinte réflexive

Nous présentons dans cette section une vision synthétique des *empreintes réflexives* des trois mécanismes de réplication que nous venons de présenter, conformément à l'étape 1 de notre démarche. En se basant sur les descriptions que nous venons d'en donner, les tables 5.2 à 5.4 page 79 présentent les différentes capacités réflexives nécessaires à la réalisation des mécanismes étudiés dans une architecture réflexive. Cette description s'appuie sur un modèle d'exécution *abstrait*, qui ne prend pour l'instant pas encore en compte la nature multi-niveaux de l'architecture sur laquelle nous souhaitons les implémenter. Les capacités décrites sont organisées en trois *facettes*, selon qu'elles ont trait à la *communication* entre les répliques et leurs clients, au *contrôle de l'exécution* des répliques ou aux *données* qu'elles traitent. La facette « données » tient plus à la structure d'un composant (quels attributs, quelles structures), alors que les facettes du « contrôle » et de la « communication » ont plus trait à son comportement.

Pour chacune de ces facettes, nous distinguons les entités du modèle d'exécution pour lesquelles des capacités réflexives doivent être fournies, ainsi que les aspects comporte-

mentaux de ces entités qui doivent être rendus accessibles. Nous n'avons pas distingué ici les différents types de capacités, *réification*, *introspection*, *intercession*, les trois formes intervenant sur chacune des entités considérées. On notera, comme nous l'annoncions dans la section 4.2.1 page 61, que les empreintes des stratégies individuelles se recouvrent largement, d'où le grand intérêt à considérer l'ensemble des capacités ainsi exposées dans le cadre d'une *empreinte globale* qui les réunisse toutes.

5.2 Réplication multi-niveaux sur CORBA/POSIX

Nous appliquons ici les étapes 2 (construction d'un méta-modèle multi-niveaux de l'architecture choisie) et 3 (projection de l'empreinte réflexive des mécanismes étudiés sur le méta-modèle ainsi obtenu) de notre démarche en illustrant comment l'empreinte que nous venons d'obtenir s'applique à une architecture construite autour d'un système d'exploitation POSIX et d'un bus à objets CORBA à brins d'exécution multiples. Cette section propose une analyse générique de cette architecture et débouche sur la présentation d'une méta-interface multi-niveaux pour la réplication de systèmes POSIX/CORBA.

Nous validerons la pertinence de cette méta-interface en pratique dans les deux dernières sections du chapitre (les sections 5.3 et 5.4 page 102) en montrant sur un exemple concret constitué de GNU/LINUX et de l'ORB commercial ORBACUS comment l'implémenter.

L'architecture à laquelle nous nous intéressons est représentée sur la figure 5.1. On remarquera sur cette figure la couche de bibliothèques système que nous mentionnions dans la section 1.2.2 page 12, qui fait le lien entre l'interface spécifiée par la norme POSIX et les appels système fournis par le noyau de l'OS sous-jacent.

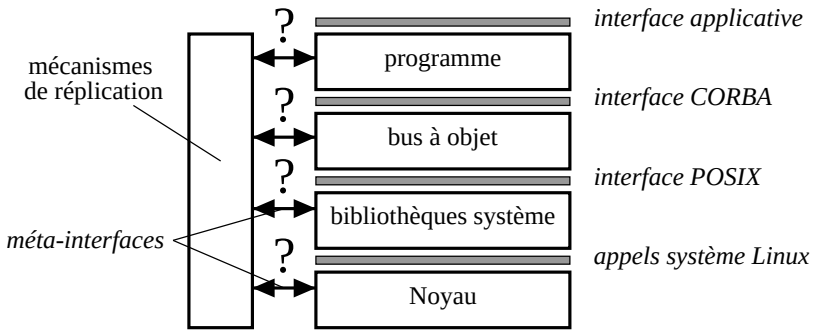


FIG. 5.1 : *Architecture concrète considérée*

Notre problème, comme nous l'avons représenté sur cette figure 5.1, est maintenant d'instrumenter les niveaux les plus appropriés de cette architecture pour réaliser l'empreinte de la section 5.1. Dans le contexte des bus à objets à brins d'exécution multiples, deux aspects particuliers des mécanismes de réplication posent particulièrement problème et ont peu été abordés : le déterminisme et la capture d'état. C'est sur ces deux aspects que nous nous concentrerons dans la suite de ce chapitre.

Facette	**Entités**	**Actions**	**Motivations**	**Moyens**
Communication	requêtes et réponses	envoi, réception	synchronisation des répliques	interception
Contrôle	brins d'exécution (*threads*)	création, progrès, terminaison	capture et restauration de requêtes en cours dans les serveurs à activités concurrentes	interception, instrumentation de la plate-forme
Données	données applicative et de la plate-forme	actions sur les données applicatives, interactions avec la plate-forme	capture cohérente de l'état	capture binaire, mise à plat, journalisation

TAB. 5.2 : *Empreinte réflexive de la réplication passive*

Facette	**Entités**	**Actions**	**Motivations**	**Moyens**
Communication	**comme passive**		validation des réponses et propagation aux clients	**comme passive**
Contrôle	non nécessaire			
Données	non nécessaire (le clonage n'est pas considéré)			

TAB. 5.3 : *Empreinte réflexive de la réplication active*

Facette	**Entités**	**Actions**	**Motivations**	**Moyens**
Communication	**comme passive**		contrôle des notifications entre meneur et suiveurs	**comme passive**
Contrôle	**comme passive** + les points de non-déterminisme	**comme passive** + les opérations non-déterministes	contrôle des décisions non-déterministes	**comme passive**
Données	**comme passive**		contrôle des interactions non déterministes avec la plate-forme	**comme passive**

TAB. 5.4 : *Empreinte réflexive de la réplication semi-active*

Nous commençons par étudier les *liens* entre CORBA et la norme POSIX en disant quelques mots sur le modèle de programmation d'un bus à objets CORBA. Ce premier défrichage nous permettra ensuite de proposer un méta-modèle générique de l'implémentation d'un ORB CORBA *à réserve de brins* (*thread pool*) sur un système d'exploitation POSIX. Forts de ce méta-modèle nous pourrons alors aborder le non-déterminisme des bus à objets à brins d'exécution multiples comme un cas particulier de non-déterminisme dû au multitraitement.

5.2.1 Le modèle de programmation CORBA, liens avec l'OS

La norme CORBA s'est, au fil des années, enrichie de très nombreux aspects, comme les communications par événements, ou les mécanismes transactionnels. Nous ne les aborderons pas ici et nous nous concentrerons sur le cœur de la norme : les invocations de méthode à distance. Cet aspect utilise un modèle de programmation extrêmement simple, centré autour de la notion de *référence d'objet*, et de *requête* : pour utiliser les services d'un objet CORBA, une application cliente doit tout d'abord obtenir une référence sur cet objet, c'est-à-dire une structure opaque pour l'utilisateur qui permettra au bus à objets de contacter l'objet CORBA considéré. Une fois cette référence obtenue, l'application cliente peut l'utiliser pour envoyer des requêtes de service à l'objet CORBA auquel elle correspond. La force de CORBA tient au fait qu'une référence CORBA se manipule syntaxiquement exactement comme une référence locale. Il n'y a pas pour le programmeur de différence entre une invocation locale de méthode et l'envoi d'une requête distante.

Une implémentation CORBA se doit de réaliser ce modèle de programmation en utilisant les services des couches exécutives qui la supportent (pour nous, la couche POSIX). Comme nous l'avons expliqué dans la section 1.2.4 page 15, la norme CORBA n'impose que très peu de contraintes d'implémentation. L'architecture d'un bus à objets CORBA telle qu'elle est prévue par la norme est représentée sur la figure 5.2 page suivante. Le `client` et le `servant` sont des objets du langage d'implémentation considéré. Le cercle marqué `IOR` (*Interoperable Object Reference*) dans le `client` représente la référence CORBA que possède le client sur l'objet CORBA réalisé par le `servant`. Le lien référence/servant n'est ni univoque ni permanent : le servant peut répondre aux requêtes adressées à plusieurs références distinctes. De même, le servant associé à une référence donnée peut changer de manière dynamique, voire même lors de chaque nouvelle invocation[1].

Une invocation de méthode sur une référence CORBA transite par différents modules du bus à objets : la *souche* réalise la *mise à plat* (*marshalling*) de la requête pour son envoi par le réseau. La souche est générée de manière automatique par un compilateur particulier, un *compilateur IDL*, à partir d'une description de haut niveau de l'interface de l'objet CORBA. Le moteur de l'ORB (réalisé le plus souvent sous forme d'une bibliothèque partagée) se charge ensuite d'utiliser les primitives du système d'exploitation pour faire parvenir la requête mise

[1] On comprend ici que la notion d'« objet CORBA » renvoie à l'abstraction perçue par l'application cliente qui en invoque les méthodes, mais ne peut être associée ni à un processus système (un processus peut répondre à plusieurs IOR), ni à un objet du langage d'implémentation qui jouerait en permanence le rôle de cet objet. La notion d'« objet CORBA » est un élément du modèle de programmation de la norme CORBA qui ne peut être relié de manière bijective à aucun élément concret de l'implémentation.

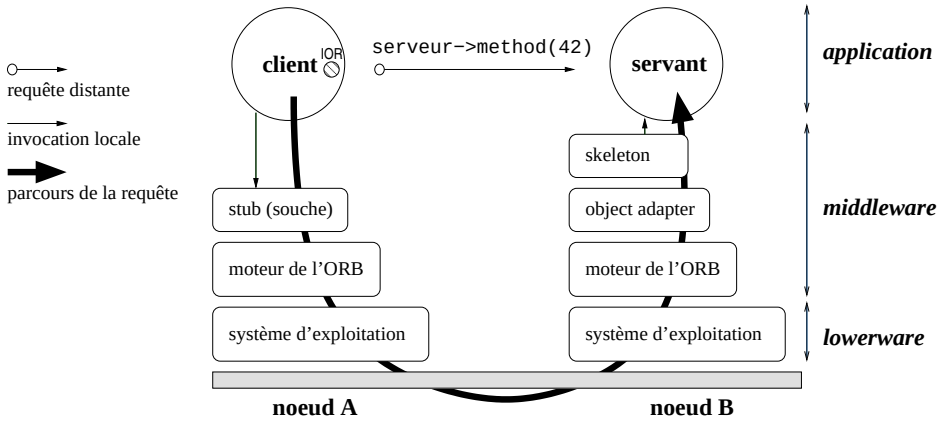


FIG. 5.2 : *Vue d'ensemble (simplifiée) d'un bus à objets CORBA*

à plat au nœud destinataire (ici le nœud B). Au nœud B, le même processus inversé se répète. L'*object adapter* (ou POA pour *Portable Object Adapter*) est chargé de sélectionner le servant auquel la requête doit être délivrée. Le *skeleton* (ou « scion ») réalise l'invocation effective sur le servant.

Une question d'implémentation essentielle a trait à la manière dont les brins d'exécution fournis par le système d'exploitation sont utilisés dans l'ORB pour réaliser les différentes étapes du traitement des requêtes. La norme CORBA est quasiment muette sur ce point, en ne distinguant que deux modes de concurrence : *sérialisé* (à tout moment, une seule requête au plus est active dans l'application), et *contrôlé par l'ORB*. Chaque implémentation est libre d'interpréter ce deuxième mode à sa guise, aucune contrainte n'étant imposée. La plupart des implémentations proposent pour ce deuxième cas un mode de concurrence basé sur une *réserve de brins*, dont le principe est représenté par un réseau de Petri sur la figure 5.3.

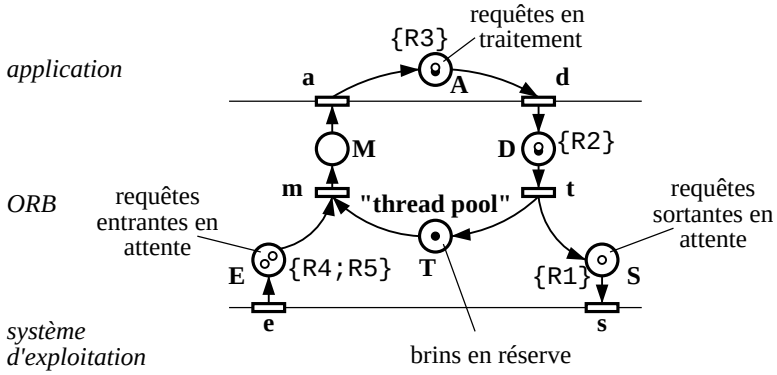


FIG. 5.3 : *Modèle générique d'un ORB à réserve de brins (*thread pool*)*

Ce mode de concurrence consiste à créer un nombre pré–fixé de brins d'exécution à l'initialisation du bus à objet pour former une *réserve de brins*, qui sera utilisée pour traiter

les requêtes (la place T sur la figure 5.3). Lorsqu'une requête arrive (transition e sur la figure 5.3 ; les requêtes `R4` et `R5` dans la place E viennent d'arriver), et si un brin est disponible dans la réserve, alors la requête est confiée au brin en question, qui prend en charge son exécution. Si aucun brin n'est disponible lors de la réception d'une requête, celle-ci est mise en attente, jusqu'à ce qu'un brin de réserve soit de nouveau libéré. Une fois associée à un brin de la réserve, la requête traverse l'ORB vers le haut (place M de notre figure, pour « montée »), jusqu'à atteindre l'application (place A, cas de la requête `R3`) où elle exécute le code prévu par le développeur utilisateur de l'ORB. Après traitement, la requête, toujours « portée » par le brin de la réserve, redescend dans l'ORB (place D sur notre figure, pour « descente »), le brin de réserve est alors libéré, et la réponse à la requête envoyée (place S, transitions t et s).

Notons que ce modèle est très générique, les implémentations réelles pouvant présenter des variantes par rapport à ce cas général. Ainsi les places E et S représentent des étapes où les requêtes sont dans l'ORB mais sont prises en charge par d'autres brins que ceux de la réserve. Or ces places peuvent ne pas exister dans certains ORB, les opérations de réception d'une requête ou d'envoi d'une réponse étant directement effectuées par un brin de la réserve. C'est par exemple le cas dans l'ORB TAO [Schmidt & Cleeland 1999], où ni la place E, ni la place S ne sont présentes. Le bus ORBACUS lui ne contient que la place E.

5.2.2 Multitraitement et non-déterminisme

Une application à brins d'exécution multiples, qu'elle contienne un bus à objets ou non, ne maîtrise pas la manière dont s'entrelacent les différents brins qui s'exécutent en son sein. Les décisions d'ordonnancement entre les différents brins d'exécution sont du ressort du système d'exploitation, qui a toute latitude pour choisir comment et dans quel ordre faire exécuter les brins. Cette liberté laissée à l'OS lui permet de partager au mieux les ressources qu'il gère en fonction de son contexte d'exécution, pour par exemple maximiser les ratios d'utilisation, la réactivité perçue par l'utilisateur, ou encore la performance globale du système. Comme l'asynchronisme dans les réseaux de communication que nous mentionnions dans la section 1.1.5 page 7, la flexibilité de l'ordonnancement (*scheduling*) des OS généralistes contribue grandement à la flexibilité des systèmes dont ils forment la base. Mais cette liberté a un prix : parce qu'il ne peut être prévu, l'ordonnancement des brins d'exécution est pour le développeur d'une application à brins d'exécution multiples *non-déterministe*. Un développeur ne peut que maîtriser partiellement l'entrelacement des brins qu'il utilise en contraignant les ordonnancements valides par l'utilisation des primitives de synchronisation comme les verrous (en anglais *mutex*, abréviation de *mutually exclusive lock*).

Dans la pratique, il est très pénalisant en termes de performance d'utiliser un grand nombre d'opérations de synchronisation dans une application. Un programmeur ne tentera donc pas par l'utilisation de verrous *ad hoc* de contraindre le système d'exploitation à utiliser un ordonnancement précis des brins de son application, mais se contentera par l'utilisation

```
thread t1 is
var a int ;
begin
  lock(l)
    a := read(v) ; -- read_1
    a := a+10    ; -- add_1
    write(v,a)   ; -- write_1
  unlock(l)
end
```

```
thread t2 is
var b int ;
begin
  lock(l)
    b := read(v) ; -- read_2
    b := b*2     ; -- mult_2
    write(v,b)   ; -- write_2
  unlock(l)
end
```

FIG. 5.4 : *Exemple de deux brins d'exécution concurrents*

de sections critiques[2], d'assurer la cohérence des traitements effectués. Par exemple, la figure 5.4 décrit dans une syntaxe inspirée du langage ADA, deux brins qui accèdent de manière concurrente à la même variable partagée `v`. Le verrou `l` assure que quelles que soient les décisions d'ordonnancement du système d'exploitation, les opérations du brin `t1` {`read_1`, `add_1`, `write_1`} se déroulent soit toutes avant, soit toutes après les opérations du brin `t2` {`read_2`, `mult_2`, `write_2`}. Autrement dit, le verrou `l` sérialise les opérations des deux brins sur la variable `v`. De cette manière, en prenant par exemple comme état initial la variable `v` à 10, l'exécution des deux brins ne peut donner comme résultat que `40` ou `30`. `t=20`, qui sans le verrou `l` serait possible, est exclu.

Les primitives de synchronisation d'un OS généraliste sont donc des moyens fournis au développeur pour composer avec le non-déterminisme de l'OS sans l'éliminer nécessairement. Plusieurs exécutions du même code peuvent donner des résultats différents, mais le respect des contraintes d'exclusion imposées par les verrous du programme assure que tous ces résultats seront cohérents avec l'*intention* du programmeur.

Lorsque, comme dans notre exemple, toutes les ressources partagées d'un programme (variables globales, canaux de communication, *etc.*) sont protégées par des verrous, et que les brins de l'application se comportent de manière déterministe entre deux allocations de verrous (ce qu'on appelle le déterminisme par morceaux ou *piecewise determinism*[3]), les résultats produits par le programme ne dépendent *que* de l'ordre d'allocation de *chacun des verrous*. Dans une telle situation, que nous prendrons comme hypothèse de travail, le choix d'allouer un verrou à tel ou tel brin constitue la seule sorte de décision non-déterministe qu'il convient de maîtriser. Dans l'exemple de la figure 5.4, si nous supposons un troisième brin `t3`, qui n'utilise ni le verrou `l`, ni la variable `v`, l'entrelacement de l'exécution de ce brin avec les deux premiers `t1` et `t2` n'influence en rien le résultat du système constitué des trois brins, et n'a besoin ni d'être observé, ni d'être contrôlé. Autrement dit, avec l'hypothèse choisie, les changements de contexte d'un brin d'exécution à l'autre ne sont pertinents pour

[2]Une section critique est une zone de code protégée par une prise de verrou. Une section critique ne peut contenir qu'un seul brin à la fois.

[3]Le déterminisme par morceaux requiert notamment que les canaux reçoivent les mêmes messages en entrées, dans le même ordre. Il exclut aussi l'utilisation d'opérations de lecture non bloquantes [Jiménez-Peris et al. 2000] (qui reviennent dans un réseau asynchrone à recevoir de manière non déterministe un message indiquant qu'« il n'y a pas de messages »), ou de données non reproductibles (comme l'heure de la journée, ou la charge de la machine).

le non-déterminisme que dans la mesure où ils traduisent une décision de synchronisation sur un verrou.

Lorsqu'une application à brins d'exécution multiples est répliquée, assurer un consensus sur la manière dont les verrous de chacune des répliques sont alloués peut se faire de manière *explicite*, en faisant communiquer les différentes répliques entre elles, comme dans la réplication semi-active. Elle peut aussi se faire de manière tacite, sans communication entre les répliques, en utilisant un algorithme déterministe d'allocation des verrous [Basile et al. 2003].

5.2.3 Déterminisme des bus à objets à brins d'exécution multiples

Les bus à objets à brins d'exécution multiples, parce qu'ils contiennent plusieurs brins d'exécution, sont intrinsèquement non-déterministes. Par exemple, si nous revenons sur la figure 5.3 page 81, la place E représente une structure qui stocke les requêtes reçues en attente. Cette structure (typiquement une file) assure le plus souvent un ordre sur les requêtes qui y sont stockées, par exemple « R4 avant R5 ». Cependant, si la réception des requêtes (transition e) est réalisée par plusieurs brins fonctionnant en parallèle (cas d'ORBACUS), l'ordre d'arrivée des messages depuis le système d'exploitation peut être totalement perdu, et ne plus se refléter dans la queue modélisée par la place E. Une réplique pourra alors sur la figure 5.3 décider de traiter R4 avant R5 avec le dernier brin de réserve, et une autre R5 avant R4, résultant en deux états incompatibles.

Approche mono-niveau : l'OS

Pour résoudre ce problème du non-déterminisme, plusieurs approches sont envisageables. La première consiste à ne pas prendre en compte la nature particulière d'un bus à objets, et à appliquer à une application qui l'utilise la méthode générale de contrôle du non-déterminisme que nous venons d'exposer. En assurant que tous les verrous utilisés dans l'ORB et dans l'application sont alloués de la même manière, et sous l'hypothèse du déterminisme par morceau, cette approche garantit que tous les objets de l'ORB et de l'application atteignent le même état dans toutes les répliques. Par exemple cette approche garantit que sur la figure 5.3, les requêtes R4 et R5 seront propagées dans le même ordre vers l'application.

Cette première approche est « aveugle » vis-à-vis du bus à objets. Elle ne perçoit de l'architecture globale du système que le niveau du système d'exploitation. Elle peut être implémentée en utilisant une plate-forme d'exécution spécifiquement instrumentée [Napper et al. 2003], ou, dans une approche réflexive, en interceptant les primitives de synchronisation. Elle comporte cependant un certain nombre de défauts, due à sa nature fondamentalement *mono-niveau*. En effet, une telle approche traite tous les verrous d'un bus à objets de manière uniforme, alors que de nombreuses opérations de synchronisation n'influencent pas son comportement tel qu'il est perçu par l'application, et donc, *in fine*, par l'utilisateur du système. Si nous reprenons l'exemple d'un bus à objets à réserve de brins de

la figure 5.3 page 81, les accès à la queue des requêtes en entrée doivent être protégés par un verrou pour assurer la cohérence de cette structure (figure 5.5, verrou **V**). Ce verrou contrôle à la fois la manière dont les requêtes sont insérées (transition e) et retirées (transition m) de la queue E.

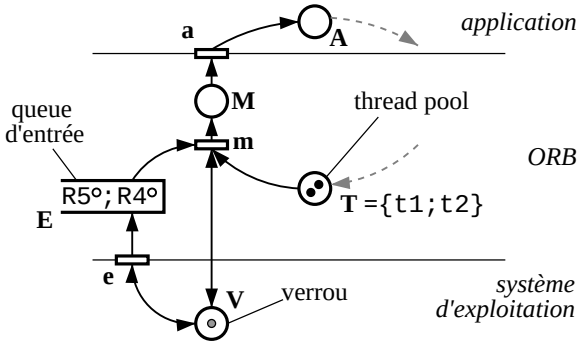


FIG. 5.5 : *Pertinence d'un verrou pour le déterminisme*

Sur la figure 5.5, deux requêtes `R4` et `R5` sont en attente, et deux brins `t1` et `t2` sont présents dans la réserve. Le contrôle des prises du verrou au niveau de la transition e (réception) assure que toutes les répliques d'un même bus à objets aient le même état de queue. Ce contrôle est donc nécessaire à la cohérence des répliques. En revanche, le contrôle du verrou **V** au niveau de la transition m, ne permet que de contrôler *quel* brin de la réserve va traiter la requête en tête de queue (ici `R4`). Or il est équivalent qu'une réplique traite la requête `R4` avec le brin `t1`, et une autre avec le brin `t2`. Les brins de la réserve sont en effet interchangeables pour le traitement des requêtes. En forçant les répliques à toutes prendre la même décision lors de l'allocation du verrou en m, l'approche par réplication de toutes les opérations de synchronisation *sur–contraint* leur exécution.

L'exemple du verrou **V** n'est qu'un exemple parmi de nombreux autres : les implémentations à brins d'exécution multiples de la norme CORBA utilisent en interne de nombreuses opérations de synchronisation qui n'influencent pas la manière dont les requêtes sont transmises à l'application. À titre d'exemple, nous avons observé dans l'ORB ORBACUS 4.1, 203 invocations à des opérations de synchronisation POSIX (c'est-à-dire des opérations commençant par `pthread_mutex_`..., comme `pthread_mutex_create` ou `pthread_mutex_lock`) entre la réception d'une requête (retour de la primitive OS de réception comme `recv`, ou `read`) et l'envoi de la réponse correspondante (retour de la primitive OS d'envoi comme `send` ou `write`). Nous n'en avons observées que 64 dans OMNIORB 4, et 52 dans TAO 1.2.1 [4], mais ces chiffres restent importants (table 5.5 page suivante).

Ces opérations de synchronisation ne déterminent pas toutes l'ordre dans lequel les requêtes atteignent le niveau applicatif. Certaines permettent de sérialiser des accès sur des compteurs (de référence, de requêtes, *etc.*), dont l'état n'est pas perceptible par l'application.

[4] Ces chiffres ne prennent pas en compte les brins qui ne traitent pas directement la requête, comme le brin ramasse miettes, ou *scavenger*, dans OMNIORB, ou les autres brins de réserve.

bus	synchronisation
ORBACUS 4.1	203
OMNIORB 4	64
TAO	52

TAB. 5.5 : *Nombre d'opérations de synchronisation durant le traitement d'une requête*

D'autres sont héritées d'un objet à l'autre du fait des relations d'héritage entre classes, mais ne sont pas nécessaires à la cohérence du bus, l'objet concerné n'étant jamais accédé de manière concurrente (c'est un cas de programmation défensive).

Cette première approche, de bas niveau, qui fait abstraction de l'existence du bus à objets et de sa sémantique en répliquant de manière aveugle toutes les décisions de synchronisation du système d'exploitation, se révèle donc particulièrement inefficace, inefficacité qui peut aller jusqu'à remettre en question sa praticabilité. Dans un autre contexte que celui des bus à objets, Napper *et al.* rapportent des sur-coûts temporels allant jusqu'à 375% du temps d'exécution sans instrumentation lorsque cette technique est utilisée à l'intérieur d'une machine virtuelle JAVA pour une application de base de données (l'application `db` de l'étalonnage SPEC JVM98) [Napper et al. 2003].

Approche mono-niveau : le point de vue applicatif

En opposition à l'approche précédente, de très bas niveau, une stratégie utilisant la réflexivité uniquement au niveau de l'application peut être envisagée. Dans cette perspective, la présence du bus à objets n'est plus ignorée, mais seule une vision comportementale de haut niveau en est retenue : un bus à objets sert de courtier à une application pour interagir avec ses pairs selon un modèle de programmation intuitif et transparent. Toute requête reçue par le bus à objets depuis le système d'exploitation, doit, en principe, finir par atteindre l'application. Il n'est donc *a priori* pas nécessaire de maîtriser comment plusieurs requêtes concurrentes s'entrelacent dans l'ORB, mais simplement de quelle manière chaque requête accède aux ressources partagées du niveau applicatif. Ainsi si deux requêtes, `R1` et `R2`, sont reçues par toutes les répliques, et tentent de modifier de manière concurrente une variable `v` protégée par un verrou, il est nécessaire que chaque réplique octroie l'accès à `v` aux requêtes dans le même ordre. En revanche, savoir quels brins de la réserve portent dans chaque réplique chacune des deux requêtes n'a pas d'importance.

Cette approche souffre cependant d'une faille majeure lorsqu'elle est utilisée sur un bus fonctionnant en mode à réserve de brins. La figure 5.6 page suivante illustre un exemple de cette faille. Supposons un ensemble de répliques fonctionnant en réplication semi-active auxquelles trois requêtes, `R1`, `R2`, `R3` sont transmises. Les bus à objets de chaque réplique fonctionnent en mode à réserve de brins, avec une réserve de brins de taille 2. Les trois requêtes modifient toutes le même objet applicatif, qui est protégé au niveau de l'application par un verrou **Va**. Le fonctionnement du bus à objets de chaque réplique étant non contraint, il est possible que la meneuse insère les requêtes dans sa queue d'admission dans l'ordre

{R1, R2, R3}, et qu'une suiveuse les insère dans l'ordre {R3, R2, R1}. Cette incohérence dans l'ordre d'insertion est possible, *même lorsque les requêtes sont délivrées dans un ordre total aux répliques par l'OS*. Nous ne maîtrisons pas en effet le fonctionnement du bus au-dessus de l'OS, qui a toute latitude pour « mélanger » les requêtes.

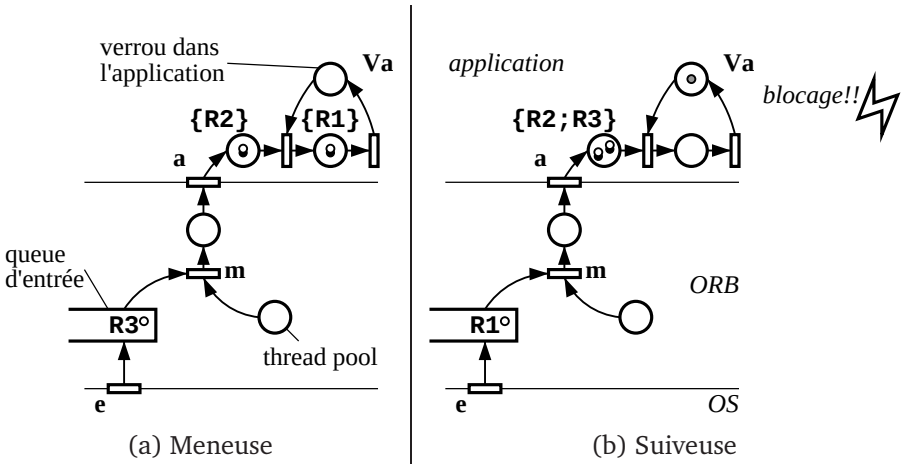


FIG. 5.6 : *Faille de l'approche mono-niveau applicative*

La réserve ne contenant que deux brins, la meneuse va transmettre R1 et R2 à l'application. R3 reste en attente dans la meneuse, puisque la réserve de brins, qui limite le degré de parallélisme du bus, a été épuisée (figure 5.6-a). La suiveuse quant à elle, parce qu'elle a inséré les requêtes dans un ordre différent, va transmettre R2 et R3 à l'application, R1 restant bloquée dans l'ORB (figure 5.6-b). Supposons maintenant que la meneuse, de manière non déterministe, choisisse de laisser la requête R1 accéder en premier au verrou **Va**, et communique ce choix non déterministe, effectué par l'OS et perceptible au niveau applicatif, à sa suiveuse. Pour suivre la décision de la meneuse, la suiveuse doit bloquer R2 et R3 au niveau du verrou **Va** pour attendre R1, à qui doit d'abord être octroyé le verrou. Mais du fait de la limitation des brins imposée par la réserve, R1 n'arrivera jamais, et la suiveuse se trouve dans l'incapacité de suivre le décision de la meneuse. Il y a *blocage*.

Approche multi-niveaux : OS + ORB + application

Pour lever le problème de l'inefficacité de la première approche, et éviter la faille de la seconde, nous proposons une stratégie basée sur les principes de la *réflexivité multi-niveaux* que nous avons présentés au chapitre précédent. Cette stratégie consiste à instrumenter le bus à objets pour capturer la sémantique de ses actions, et utiliser cette connaissance de « haut niveau » pour contrôler l'interception des opérations de synchronisation réalisées au niveau du système d'exploitation. Cette stratégie requiert de pouvoir faire le lien entre d'une part le modèle de programmation d'un bus à objets CORBA, c'est-à-dire essentiellement la notion de *requête*, et d'autre part les éléments de programmation fournis par un système d'exploitation tels que les brins, les *verrous* ou les *canaux de communication* (par exemple

sous POSIX, les *sockets* orientées connexion[5]). Dans ce but, nous avons décidé d'utiliser les étapes de traitement d'une requête dans l'intergiciel (son « cycle de vie » : réception, montée dans l'ORB, traitement par l'application, *etc.*), pour aborder l'empreinte des mécanismes de réplication des tables 5.2 à 5.4 page 79 [Taïani et al. 2003]. Comme nous nous intéressons à la maîtrise du non-déterminisme, nous nous sommes essentiellement penchés sur les facettes de la « communication » et du « contrôle » de cette empreinte.

Nous avons vu, lors du commentaire de la figure 5.5 page 85, au sujet de la réplication de toutes les décisions de synchronisation de l'OS, que certains points de décisions (comme la transition m sur la figure 5.5) étaient pertinents pour la cohérence du bus à objets, mais que d'autres ne l'étaient pas. Les points de décision « pertinents » déterminent l'ordre d'arrivée des requêtes au niveau de l'application. Il s'agit de décisions internes à l'ORB, qui doivent être *réifiées* au moment où elles se produisent, puisque comme nous venons de le voir, dans un ORB dont le degré de concurrence est limité, les conséquences de ces décisions ne peuvent plus être contrôlées au niveau applicatif. Nous avons donc choisi d'appeler ces points « pertinents » des *point de contention* pour les requêtes.

```
class RequestContentionPoint ;
class RequestID ;

class MetaRequestLifeCycle {
  void requestHasBeenReceived       (RequestID aReqID) ;
  void replyHasBeenSent             (RequestID aReqID) ;
  void requestBeforeContentionPoint (RequestID aReqID,
                                     RequestContentionPoint aContP) ;
  void requestAfterContentionPoint  (RequestID aReqID,
                                     RequestContentionPoint aContP) ;
};
```

FIG. 5.7 : *Un réification du cycle de vie des requêtes*

Pour réifier ces points au méta-niveau nous proposons d'utiliser la méta-interface décrite par la classe `MetaRequestLifeCycle` de la figure 5.7. Cette classe spécifie trois méthodes de réification :

- `requestHasBeenReceived`, qui est appelée par l'ORB lorsqu'une nouvelle requête a été reçue ;
- `replyHasBeenSent`, qui est appelée après qu'une requête a été envoyée ;
- `requestBeforeContentionPoint` et `requestAfterContentionPoint`, enfin, qui informent le méta-niveau lorsqu'une requête respectivement atteint ou dépasse un point de contention.

[5]Nous utiliserons dorénavant le mot *connexion* pour parler des *sockets* orientées connexion.

Un appel à `requestBeforeContentionPoint` signifie qu'un brin porteur d'une requête est sur le point de demander l'obtention d'un verrou géré par l'OS, et que cette allocation peut potentiellement introduire une incohérence entre les répliques si elle n'est pas maîtrisée. Lors du retour de `requestBeforeContentionPoint`, la main est rendue au code du bus à objets, et la demande d'obtention du verrou a effectivement lieu. `requestAfterContentionPoint` est appelée lorsque le brin porteur de requête a effectivement obtenu le verrou de la part de l'OS.

Dans le contexte d'une réplication semi-active, `requestAfterContentionPoint` permet à la réplique meneuse de savoir dans quel ordre les différentes requêtes ont passé un point de contention donné. En utilisant des identifiants de requêtes uniformes pour les trois répliques, elle peut propager cette décision prise par l'OS aux autres répliques. Les appels aux trois méthodes de `MetaRequestLifeCycle` étant bloquants, il est alors possible aux suiveuses, en retardant `requestBeforeContentionPoint`, d'assurer depuis le méta-niveau que leurs requêtes respectent bien le même ordre que la meneuse.

Les points de contention n'existent pas que dans le bus à objets, mais aussi dans l'application, puisque nous supposons toutes les ressources partagées de celle-ci contrôlées par des verrous. Ces points de contention doivent bien sûr aussi être réifiés à notre objet `MetaRequestLifeCycle` pour prendre en compte tous les niveaux qu'une requête est amenée à traverser.

5.2.4 Capture de l'état d'un bus à objets à brins d'exécution multiples

Nous avons dans la section précédente proposé la méta-interface `MetaRequestLifeCycle` (figure 5.7 page précédente) pour maîtriser le non-déterminisme d'un bus à objets à réserve de brins selon une approche réflexive multi-niveaux. Nous nous intéressons maintenant à un second aspect essentiel de l'empreinte réflexive proposée dans la section 5.1 : la capture et la restauration de l'état d'une plate-forme CORBA / POSIX.

Comme nous l'avons mentionné dans la section 3.1.1 page 36, la capture de l'état d'un bus à objets à brins d'exécution multiples recouvre deux aspects : la capture des données gérées par le bus, d'une part, et la capture de l'état des différents brins actifs au sein du bus d'autre part (leur pile d'appels, leurs variables locales, *etc.*). Ce deuxième aspect représente une difficulté spécifiquement due au multitraitement, et ne serait pas présent si les requêtes étaient sérialisées (auquel cas il suffirait de sauver l'état entre chaque traitement de requête).

L'état d'un bus à objets à brins d'exécution multiples peut être capturé en utilisant les approches de capture binaire que nous avons présentées au chapitre 3 page 44 : un bus à objet se présente le plus souvent sous la forme d'une bibliothèque partagée, dont l'état relatif à une application est contenu dans l'espace d'adressage de cette application. Cette approche, si elle est envisageable, souffre cependant de tous les désavantages des mécanismes de capture binaire (portabilité, remise en cohérence des objets « externes » à l'espace d'adressage, comme les entités gérées par l'OS, taille des images capturées).

Il est aussi possible de capturer l'état d'un bus à brins d'exécution multiples en utilisant des approches « symboliques » telles que celles que nous avons introduites dans la section 3.2.3 page 48. Ces approches permettent, en utilisant une réécriture automatique du code (par exemple avec un compilateur réflexif), de capturer l'état des brins d'exécution et celui des objets internes au bus à objets de manière portable et transparente [Killijian et al. 2002, Karablieh & Bazzi 2002, Balkrishna Ramkumar 1997, Ferrari et al. 1997]. Cependant, parce qu'elles nécessitent la réécriture et la modification de l'intégralité du code source du bus à objets, ces approches peuvent s'avérer particulièrement lourdes à mettre en place. À titre d'exemple, les sources de la bibliothèque partagée d'ORBACUS 4.1 contiennent plus de 110 000 lignes de code, celles de TAO plus de 340 000, et celles d'OMNIOBR 4 plus de 77 000[6]. Ces chiffres s'augmentent encore des différentes bibliothèques qu'utilisent ces ORB comme couches d'abstraction supplémentaires au-dessus de l'OS (comme par exemple ACE dans le cas de TAO [Schmidt 1998]).

Comme alternative, nous proposons de mettre à profit la maîtrise du non-déterminisme que fournit la méta-interface que nous avons proposée en figure 5.7 page 88, pour permettre la restauration de l'état d'un bus à objets sans instrumentation excessive. L'idée que nous proposons peut être vue comme une adaptation des principes de restauration d'état par journalisation (*logged-based recovery*) [Strom et al. 1988, Johnson & Zwaenepoel 1987]. Plutôt que de capturer explicitement l'état de l'intégralité des structures de l'ORB, nous proposons de ramener l'ORB dans un état qui, pour l'application, est indistinguable de l'état capturé, en utilisant une technique de « *ré–exécution* » accélérée. Pour cela nous classons les requêtes en cours dans un ORB en trois groupes selon le degré d'avancement de leur traitement (figure 5.8) :

1. le premier groupe, G_1, est constitué des requêtes reçues qui n'ont pas encore atteint le niveau applicatif ;
2. le second groupe, G_2, contient les requêtes qui sont actives dans l'application, mais dont le traitement n'est pas encore terminé ;
3. le troisième groupe, G_3, regroupe les requêtes dont le traitement est terminé, mais dont le résultat n'a pas encore été retourné à son expéditeur.

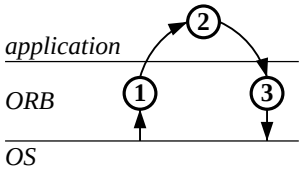


FIG. 5.8 : *Classification des requêtes selon leur degré d'avancement*

[6]Ces chiffres incluent les lignes de commentaire des sources. Ceci ne modifie cependant pas notre propos, les pourcentages de commentaire des trois ORB (en excluant les entêtes de licence) restant relativement faibles, puisqu'ils oscillent entre un peu moins de 12% pour ORBACUS, et un peu moins de 9% pour OMNIORB

La méta-interface `MetaRequestLifeCycle` de la figure 5.7 page 88, permet de savoir quelles requêtes sont présentes dans l'ORB. Pour distinguer chacun des trois groupes, il est nécessaire de rajouter à cette interface de nouveaux appels de réification :

`IntercessionCommand requestBeforeApplication(RequestID aReqID)` informe le méta-niveau qu'une requête est sur le point d'entrer dans l'application. La valeur de retour (de type `IntercessionCommand`) peut prendre deux valeurs : `continueExecution` et `skipCallToApplication`. Nous verrons l'intérêt de ces valeurs lorsque nous discuterons la restauration de l'état de l'ORB.

`void requestAfterApplication(RequestID aReqID)` notifie le méta-niveau qu'une requête est sur le point de retourner dans l'ORB après avoir été traitée. Sa réponse n'a pas encore été renvoyée au client.

À ces deux nouveaux appels doit s'ajouter une interface d'introspection, `Request`, qui à partir de l'identifiant d'une requête permet à l'ORB d'obtenir les informations relatives à cette requête (méthode appelée, valeur des paramètres d'entrée, valeur des paramètres de retour si le traitement est terminé, *etc.*). L'interface `Request` doit aussi permettre, à des fins de restauration, de ré–injecter le résultat du traitement d'une requête dans l'ORB, sans traiter la requête. Nous ne rentrons pas plus dans les détails de ces capacités réflexives au niveau d'une requête qui peuvent par exemple être implémentées en utilisant une interface similaire à celle de `CORBA::ServerRequest` décrite par la norme CORBA pour la réception dynamique de requêtes [OMG 2002d].

Pour capturer l'état d'un ORB nous proposons la démarche suivante :

1. Enregistrer le contenu de toutes les requêtes présentes dans l'ORB (c'est-à-dire faisant partie de l'un des trois groupes, G_1, G_2, G_3, que nous avons introduits.) Les informations capturées doivent être suffisantes pour « ré–injecter » les requêtes dans l'ORB lors d'une restauration.
2. Enregistrer pour chaque requête le groupe auquel elle appartient.
3. Enregistrer pour chaque point de contention, l'ordre de passage de chaque requête capturée.
4. Enregistrer les résultats des requêtes du groupe G_3 (dont le traitement est terminé).

Cette capture de l'état de l'ORB doit s'accompagner d'une capture, complémentaire, de l'état de l'application. Le niveau applicatif possède un aspect « données » beaucoup plus riche que celui de l'ORB, et il est par conséquent beaucoup plus complexe, voire impossible, d'y appliquer une approche par ré–exécution[7]. Nous proposons de découpler la capture du niveau applicatif de celle de l'ORB, en utilisant pour l'application une stratégie « symbolique », parmi celles que nous avons présentées dans la section 3.2.3 page 48. Ces approches

[7] L'intérêt de la ré–exécution est d'éviter d'avoir à capturer explicitement l'état des différents brins d'exécution. Elle nécessite de partir d'un état « initial » à partir duquel, par une ré–exécution contrôlée, chaque brin est ramené dans un état équivalent à celui capturé. L'approche n'est intéressante que lorsqu'un tel « état initial » approprié, exempt de traitement, existe. Ce n'est pas le cas pour le niveau applicatif, dont l'état des données est à tout moment intimement lié aux traitements qui y ont lieu. Aucun découplage données / contrôle n'est alors possible

permettent de capturer l'intégralité des données gérées par l'application (par exemple en utilisant le mécanisme présenté dans [Killijian et al. 2002]), ainsi que de capturer et de restaurer *partiellement* la pile d'un brin. Cette capacité de sauvegarde partielle est un élément essentiel pour pouvoir utiliser deux techniques de capture d'état hétérogènes pour l'application et l'ORB. Considérons par exemple, un brin d'exécution correspondant à une requête du groupe G_2 (en cours de traitement). La pile de ce brin contient au moment de la capture d'état les appels suivants

$\{\texttt{appelORB}^0\texttt{()}, \ldots \texttt{appelORB}^i\texttt{()}, \ldots \texttt{appelORB}^n\texttt{()}, \texttt{appelAppli}^0\texttt{()}, \ldots \texttt{appelAppli}^n\texttt{()}\}$

(avec $\texttt{appelORB}^0$ correspondant au bas de la pile, et $\texttt{appelAppli}^n$ au haut). Les approches de capture symbolique de brin (c'est à dire utilisant une représentation basée sur les noms de méthode et les points d'appels situés dans le code source) permettent de ne capturer que la partie « supérieure » de cette pile :

$$\{\texttt{appelAppli}^0\texttt{()}, \ldots \texttt{appelAppli}^n\texttt{()}\}.$$

Cette capture permet donc au moment de la restauration de mener la pile d'un nouveau brin d'un état contenant

$$\{\texttt{appelORB}^0\texttt{()}, \ldots \texttt{appelORB}^i\texttt{()}, \ldots \texttt{appelORB}^n\texttt{()}\},$$

à l'état de la pile originale, sans ré–exécuter la couche applicative.

La figure 5.10 page 94 donne un exemple de l'état global du système ainsi capturé : les contenus des requêtes (`R1`, `R2`, `R3`, `R4`, `R5`) ; leur classification en groupes (G_1, G_2, G_3) ; l'ordre dans lequel a été passé le point de contention représenté (`R1` puis `R2` puis `R3`) ; la partie applicative de la pile du brin traitant `R2` ($\{\texttt{appelAppli}^0, \texttt{appelAppli}^1\}$) ; la partie applicative de la pile du brin traitant `R3` ($\{\texttt{appelAppli}^0\}$) ; le résultat de la requête `R1`. Une fois ces informations capturées, la restauration s'opère de la manière suivante (voir la figure 5.11 page 94) :

1. Les requêtes sont ré–injectées dans l'ORB, en utilisant une méthode d'intercession `injectRequestAtCommuncationLevel(Request r)`. La ré–injection est « contrôlée » : par le contrôle des points de contention, les appels ne sont d'abord pas autorisés à se propager vers le niveau applicatif (figure 5.11-a).
2. En utilisant le contrôle des points de contention, les requêtes des groupes G_2 (requêtes en cours de traitement) et G_3 (requêtes traitées en attente de retour) sont amenées juste avant l'entrée dans le niveau de l'application. Ceci est possible en bloquant le retour de `requestBeforeApplication(..)` au méta-niveau (figure 5.11-b).
3. La partie données du niveau applicatif est restaurée par la méthode symbolique retenue.
4. Les brins porteurs de requêtes du groupe G_2 ont maintenant des piles d'appel de la forme $\{\texttt{appelORB}^0\texttt{()}, \ldots \texttt{appelORB}^i\texttt{()}, \ldots \texttt{requestBeforeApplication()}\}$. En utilisant le mécanisme de restauration partielle de pile, ces piles sont étendues pour restituer la partie « applicative » des brins correspondant aux requêtes du groupe G_2 . Ces brins sont ensuite maintenus bloqués (figure 5.11-c).
5. Le traitement des requêtes du groupe G_3 est court-circuité en utilisant au méta-niveau la valeur de retour `skipCallToApplication` dans la méthode `request-`

`BeforeApplication(..)`. Les requêtes du groupe G_3 activent alors la méta-méthode `requestAfterApplication`. Le résultat de chacune des requêtes, présent dans la capture d'état, est rétabli en utilisant les capacités d'intercession de l'interface `Request` (voir page 91). La main est rendue à l'ORB en terminant l'appel à `requestAfterApplication` (figure 5.11-d).

6. Tous les brins de l'ORB, notamment ceux traitant les requêtes des groupes G_1 et G_2, sont débloqués.

5.2.5 Le méta-modèle résultant, conclusion

En ajoutant à la méta-interface `MetaRequestLifeCycle` présentée sur la figure 5.7 page 88 les nouvelles capacités réflexives que nous avons introduites pour la capture d'état, nous obtenons la méta-interface de la figure 5.9.

```
class RequestContentionPoint ;
class Request ;
class RequestID {
  Request getCorrespondingRequest() ;
} ;
typedef enum { continueExecution, skipCallToApplication }
  IntercessionCommand ;

class MetaRequestLifeCycle_V2 {
  void requestHasBeenReceived       (RequestID aReqID) ;
  void replyHasBeenSent             (RequestID aReqID) ;
  void requestBeforeContentionPoint (RequestID aReqID,
                                     RequestContentionPoint aContP) ;
  void requestAfterContentionPoint  (RequestID aReqID,
                                     RequestContentionPoint aContP) ;
  IntercessionCommand
       requestBeforeApplication     (RequestID aReqID) ;
  void requestAfterApplication      (RequestID aReqID) ;

  void injectRequestAtCommuncationLevel(Request r) ;
} ;
```

FIG. 5.9 : *Méta-interface retenue pour la réplication d'un ORB*

Cette méta-interface permet de maîtriser le non-déterminisme et de restaurer l'état d'un bus à objets à brins d'exécution multiples en réifiant au méta-niveau les décisions de synchronisation prises par le système d'exploitation qui influencent de manière pertinente le traitement des requêtes CORBA. Il s'agit donc véritablement d'une approche multi-niveaux,

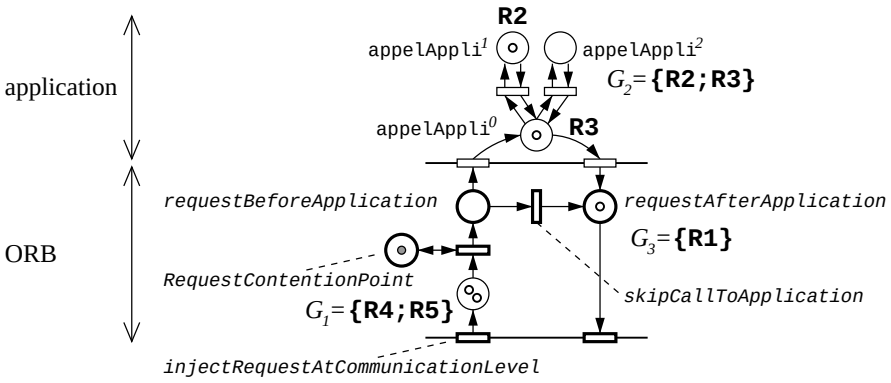
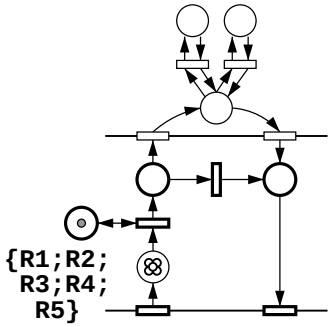
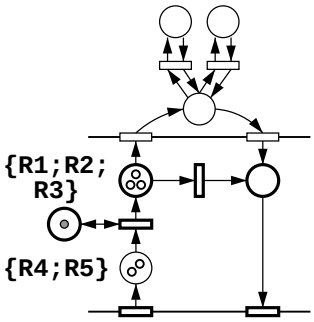


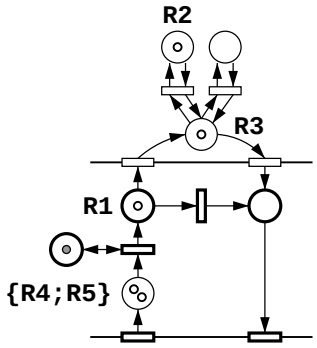
FIG. 5.10 : *L'état à restaurer*



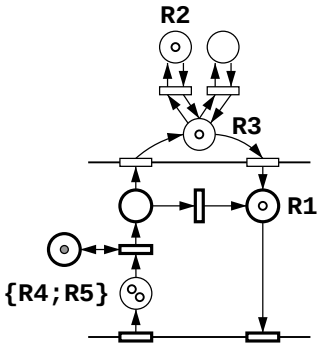
(a) Ré–injection des requêtes



(b) Contrôle des points de contention



(c) Restauration des piles du groupe G_2



(d) Court-circuit de l'application pour G_3

FIG. 5.11 : *Restauration de l'état du système par notre approche*

qui agrège des observations de niveau OS (synchronisation) et de niveau ORB (cycle de vie des requêtes). Cette approche combinée permet comme nous l'annoncions dans la conclusion du chapitre 3 page 51 de

> *« ... combiner la puissance d'inférence sémantique des hautes couches avec la force de contrôle brute des bas niveaux en une perception globale du système. »*

Cette approche multi-niveaux évite l'inefficacité et les failles des approches mono-niveau que nous avons mentionnées, en « ouvrant » partiellement l'implémentation du bus à objets, selon une méta-interface qui reste très générique, car basée sur des notions qui combinent les éléments du modèle de programmation de la norme POSIX (communication, synchronisation) et de celui de la norme CORBA (requêtes), sans préjuger d'une implémentation concrète particulière.

5.3 Extraction de méta-modèles : problématique et solutions

La méta-interface `MetaRequestLifeCycle_V2` de la section précédente capture les interactions pertinentes pour nos objectifs de réplication entre les niveaux OS et ORB d'une architecture basée sur un bus à objets CORBA et un OS POSIX. Elle constitue donc le résultat à la fois de l'étape 2 (la construction du méta-modèle multi-niveaux de l'architecture) et de l'étape 3 (la projection de l'empreinte réflexive des mécanismes) de la démarche que nous avons proposée au chapitre précédent. Cette méta-interface s'appuie sur une vision abstraite des liens entre ORB et OS, et n'est pas spécifique à une implémentation particulière. Pour terminer le développement d'une méta-interface multi-niveaux selon notre démarche il nous faut maintenant proposer une approche d'implémentation qui permette de la réaliser en pratique. Cet objectif requiert de pouvoir relier la structure d'un ORB concret au *méta-modèle générique* que nous avons utilisé (figure 5.3 page 81). Cet exercice de correspondance nous permettra alors d'identifier les points d'instrumentation les plus appropriés. Dans cette section nous présentons les techniques que nous avons utilisées pour extraire un tel *méta-modèle concret* en utilisant les sources d'un composant utilisé dans une architecture en couches (que ce soit ou non un bus à objets).

Dans la section 5.4 page 102 nous présenterons le méta-modèle que nous avons ainsi obtenu pour un bus à objets du commerce (ORBACUS) fonctionnant au-dessus de GNU/LINUX. Nous validerons alors notre démarche globale en décrivant un prototype d'implémentation de la méta-interface `MetaRequestLifeCycle_V2` de la figure 5.9 page 93 sur cet ORB.

5.3.1 La problématique

La construction du méta-modèle d'un composant à partir de ses sources revient à opérer une *rétro-conception* (*reverse-engineering*) partielle en utilisant comme guide la nature des

informations réflexives que l'on cherche à obtenir. Comme nous l'avons expliqué au chapitre 4, nous sommes essentiellement intéressés par l'étude des liens entre niveaux d'une architecture. Un composant utilisé dans une architecture en couches (CORBA, un OS, une JVM) sert d'intermédiaire entre les niveaux qui l'entourent, d'un point de vue structurel comme d'un point de vue comportemental (section 4.1.3 page 56) : ce composant utilise les entités exportées par les niveaux qui lui sont inférieurs (types, fonctions, classes, objets, *etc.*) pour fournir de nouvelles entités (structure) ; il interagit avec les composants qui lui servent de substrat d'implémentation et ceux à qui il fournit ses services (comportement). Il est donc relativement naturel pour étudier ces différents liens entre niveaux de partir des interfaces qui entourent un composant : interface haute (interface exportée vers les niveaux supérieurs), et interface basse (interface importée des niveaux inférieurs). Les entités de l'interface basse *se ramifient* à l'intérieur du composant dans une sorte de « progression vers le haut », alors que les entités de l'interface haute y *prennent racines* (figure 5.12).

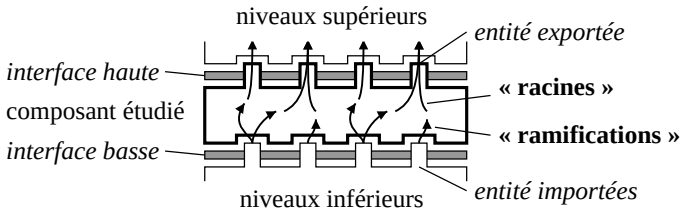


FIG. 5.12 : *Racines et ramifications dans l'analyse des liens inter-niveaux*

Nous proposons d'analyser ces racines et ramifications en extrayant de manière automatique des diagrammes structurels et comportementaux des composants étudiés. Par exemple, pour des composants orientés objet, nous utilisons les diagrammes de classes spécifiés par la norme UML [OMG 1999] pour la partie structurelle, et des diagrammes d'interaction proches de ceux d'UML mais adaptés à nos besoins pour la partie comportementale (nous y reviendrons plus en détail dans la section 5.3.2 page suivante). L'extraction automatique d'informations pour l'analyse des liens inter-niveaux se heurte cependant à deux difficultés principales :

Complexité

Du fait de la complexité des composants étudiés, les diagrammes obtenus contiennent un très grand nombre d'éléments, et dans leur forme brute ne peuvent être exploités directement par un être humain pour comprendre le logiciel. À titre d'exemple, les sources de la bibliothèque d'exécution de l'ORB commercial ORBACUS 4.1 contiennent plus de 1200 classes, reliées par 1160 liens d'héritage, et 1403 liens d'associations / compositions. L'ORB TAO 1.2 [Schmidt & Cleeland 1999] (sans son support d'exploitation ACE) contient lui plus de 1400 classes, 1014 liens d'héritage et 1846 associations. Le même foisonnement d'informations se retrouve dans l'analyse du comportement. C'est un problème classique de la rétro-conception [Chen et al. 1995], et nous l'avons résolu dans notre contexte particulier en développant un outil de manipulation de diagrammes

adapté à nos besoins (l'outil COSMOPEN [Taïani 2003], que nous présentons plus en détail dans l'Annexe B page 127).

Hétérogénéité des modèles de programmation

Deux couches voisines peuvent utiliser des modèles de programmation différents. Par exemple, les bus à objets ORBACUS et TAO sont programmés en C++, mais utilisent des bibliothèques système programmées en C. Une application tournant au-dessus d'une machine virtuelle JAVA présente un cas encore plus extrême du même type de phénomène. Cette hétérogénéité pose problème, puisqu'elle requiert des outils capables d'analyser des éléments issus de plusieurs paradigmes (orienté objet et procédural par exemple), et nécessite des modes de représentation adaptés. Nous avons contourné ce problème en considérant les types exportés par les bibliothèques système et utilisés par les ORB au même niveau d'abstraction que des classes objet.

L'extraction d'informations structurelles est maintenant prise en charge par de nombreux outils, tel DOXYGEN[8] ou RATIONAL ROSE. Nous ne donnerons donc pas ici plus de détails sur son fonctionnement, et nous contenterons d'étudier son application à la réalisation d'un méta-modèle dans la section 5.4. L'obtention d'informations comportementales pose plus de problèmes et nous nous y intéressons dans la section qui suit.

5.3.2 Extraction automatique d'informations comportementales

Problème et démarche

La construction de modèles comportementaux connaît deux grands types d'approches : les approches statiques, et les approches dynamiques. Les méthodes statiques consistent à inférer le comportement d'un programme à partir de ses sources sans l'exécuter. La construction d'arbres d'appel statiques (*la méthode `mA` contient dans son code un appel à la méthode `mB`*) est un exemple de ce type d'approche, très facile à mettre en œuvre, mais peu utile du fait de son peu de précision. À l'autre extrême du spectre des méthodes d'analyse statique en termes de puissance d'inférence, l'on trouve l'interprétation abstraite [Cousot & Cousot 1977], très puissante, mais plus difficile à mettre en œuvre sur de grands programmes orientés objet.

Les approches dynamiques, que nous avons privilégiées pour des raisons de temps et de simplicité, consistent à observer un composant « *in vivo* » lors de son exécution en utilisant des extracteurs événementiels [Reiss & Renieris 2000, Systä 2000] et une représentation adaptée. Il nous a semblé pertinent de placer la granularité de nos observations au même niveau que celle de nos diagrammes structurels, en traçant les appels de méthode entre objets. Ce choix a le mérite d'être très facile à mettre en œuvre, sans instrumentation du programme observé, en utilisant les capacités classiques de traçage des débogueurs les plus courants comme par exemple `gdb` [Stallman & Pesch 2002].

[8] http ://www.doxygen.org/

L'utilisation d'un débogueur non intrusif impose cependant des contraintes : tous les appels de méthodes et de fonctions ne peuvent être tracés sans ralentir de manière rédhibitoire et inenvisageable en pratique le programme observé. Les particularités de notre problématique (l'analyse des liens inter-niveaux) nous ont permis de contourner cette limitation en ne posant des points de traçage que sur les interfaces hautes et basses des composants étudiés. Le fonctionnement des langages structurés (c'est-à-dire l'utilisation d'une pile), nous permet à chaque fois que nous détectons une invocation sur l'un de nos points d'observation de tracer la suite d'appels imbriqués qui l'a produite. En guise d'exemple, nous appliquons dans ce qui suit cette approche à l'analyse de la bibliothèque système de multitraitement de GNU/LINUX (`libpthread.so`). Cet exemple nous permettra aussi d'illustrer l'importance de notre outil de manipulation de diagrammes COSMOPEN pour pouvoir « abstraire » une partie des événements observés et aborder l'étude de différents composants de manière séparée.

Exemple de résultats : la bibliothèque `libpthread.so` de GNU/LINUX

Le programme présenté sur la figure 5.13 crée deux brins POSIX, `threadN1` et `threadN2`, en utilisant la fonction POSIX `pthread_create`, puis se termine.

```
int main () {
   pthread_t threadN1, threadN2 ;
   pthread_create(&threadN1, NULL, StartOfThreadN1, NULL) ;
   pthread_create(&threadN2, NULL, StartOfThreadN2, NULL) ;
   pthread_join(threadN1, NULL) ;
   pthread_join(threadN2, NULL) ;
} ;
```

FIG. 5.13 : *Un programme élémentaire à brins d'exécution multiples*

Lorsqu'il est compilé puis lancé sous LINUX 2.4, ce programme fonctionne dans une architecture représentée sur la figure 5.14 : comme nous l'avions expliqué dans la section 1.2.2 page 12, les primitives de multitraitement du standard POSIX ne sont pas réalisées directement par le noyau LINUX, mais par une bibliothèque système (appelée `libpthread.so`) qui projette l'interface POSIX sur les appels système, plus élémentaires, du noyau. Parmi ceux ci, l'appel système `clone` permet notamment de créer un nouveau brin d'exécution LINUX[9].

En posant des points d'observation sur les interfaces haute (interface POSIX) et basse (appels système) de la bibliothèque de multitraitement, nous obtenons avec le débogueur `gdb` une suite de traces d'appels « brutes » similaires à celle de la figure 5.15 page 100. Cette suite de traces peut être transformée dans un format XML, facilement traitable de manière

[9] `clone` crée un *brin d'exécution système*, ou processus léger (*light weight processus* ou *LWP* en anglais). Il existe une relation 1 :1 sous LINUX entre les brins d'exécution système et les brins POSIX qui sont *perçus* par le développeur. Ce n'est pas le cas de tous les systèmes d'exploitation.

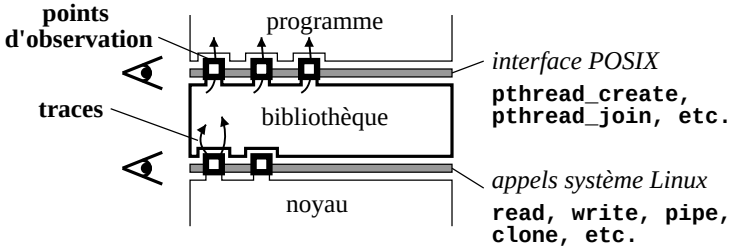


FIG. 5.14 : *Architecture du multitraitement sous* LINUX 2.4

automatique. L'observation du programme de la figure 5.13 permet ainsi d'enregistrer 86 invocations. Un sous–ensemble de ces invocations, qui correspond au mécanisme de création de brins à l'intérieur de la bibliothèque système `libpthread.so`, est représenté sous forme d'un diagramme d'interaction sur la figure 5.16 page 101. Pour lire correctement cette figure, il convient de garder à l'esprit que chaque arête représente un appel de fonction. Cet appel peut donner lieu à des appels emboîtés, puis lorsqu'il se termine « ramène » le brin concerné en arrière (retour de fonction). Ce retour n'est pas représenté. Sur la figure par exemple l'appel (2) à `__pthread_initialize_manager` produit les appels emboîtés (3), (4) et (5) à `write`, `pipe` et `clone`, puis est suivi de l'appel (7) à `write` depuis `pthread_create` (l'événement (6) correspond à la création du brin 2 par le noyau).

Cette figure appelle quelques commentaires, qui éclaireront comment nous avons obtenu le méta-modèle des interactions ORB/OS que nous présentons dans la section qui suit. Nous y voyons apparaître quatre brins, numérotés de $t1$ à $t4$. $t1$ est le brin principal du programme, celui qui exécute la fonction `main` de la figure 5.13. Les brins $t3$, et $t4$ sont les deux brins créés par le programme par l'appel à la fonction POSIX `pthread_create`, visible dans le code de la fonction `main`. L'on peut alors s'interroger sur le brin $t2$, créé par le brin $t1$ par un appel (numéro (4)) à l'appel système `clone`. Ce brin est en fait le « gestionnaire de brins » que nous mentionnions dans la section 4.1.3 page 57 au chapitre précédent. Il n'est pas visible au niveau de l'interface POSIX, et est créé lors du premier appel à la fonction `pthread_create`. C'est lui qui crée ensuite effectivement les nouveaux brins du programme ($t3$ et $t4$) en utilisant l'appel système `clone`. Le gestionnaire de brins fonctionne en mode client/serveur en traitant les requêtes qu'il reçoit par l'intermédiaire d'un *tube* (`pipe` en anglais, un flux de données entre brins), créé ici par le brin $t1$ lors de l'appel (3). Les requêtes successives de créations des brins $t3$ et $t4$ faites par le brin $t1$ transitent par ce *tube*, que nous reconnaissons sur la figure lors des appels (7) et (15) par $t1$ à l'appel système `write`. Le gestionnaire de brins $t2$ lit, lui, ces requêtes lors des appels (9) et (16) à `read` (la paire lecture/écriture (5)`write`/(8)`read` correspond à une synchronisation entre $t1$ et $t2$ pour des raisons de débogage).

Nous avons développé au sein de notre outil COSMOPEN des opérateurs d'abstraction appropriés pour pouvoir « filtrer » de manière automatique les invocations qui correspondent au fonctionnement interne de la bibliothèque de multitraitement. Ce filtrage par abstraction permet de ne plus faire apparaître qu'une sémantique de haut niveau (celle qu'avait en tête

```
#0  0x401b5b90 in clone () from /lib/libc.so.6
#1  0x4002a846 in __pthread_initialize_manager() from /lib/libpthread.so.0
#2  0x4002a9ba in pthread_create@@GLIBC_2.1() from /lib/libpthread.so.0
#3  0x0804861c in main (argc=1,argv=0xbffff854) at main-multi-thread.cpp:56
#4  0x400f3e3e in __libc_start_main() from /lib/libc.so.6

Breakpoint 5, 0x401b5b90 in clone () from /lib/libc.so.6
```

```
<trace thread="1" >
  <call entity="" method="clone" this="" />
  <call entity="" method="__pthread_initialize_manager" this="" />
  <call entity="" method="pthread_create@@GLIBC_2.1" this="" />
  <call entity="" method="main" this="" />
  <call entity="" method="__libc_start_main" this="" />
</trace>
```

FIG. 5.15 : *Une trace d'appel sur l'appel système* `clone` *et sa représentation XML*

le programmeur lorsqu'il a écrit le code de sa fonction `main`) : la création de deux brins par le brin principal (figure 5.17 page ci-contre). Nous renvoyons le lecteur intéressé à l'annexe B page 127 pour plus de détails sur la manière dont nous opérons. Cette capacité d'abstraction est particulièrement importante dès que l'on s'intéresse à des programmes qui contiennent plusieurs niveaux entre le système d'exploitation et le niveau applicatif (cas typique des systèmes utilisant un bus à objets) : la méthode d'observation que nous utilisons capture en effet tous les événements de toutes les couches situées au-dessus du noyau. Pour pouvoir analyser les interactions d'une couche avec ses niveaux voisins, il est essentiel de pouvoir filtrer les événements capturés qui ne correspondent pas à cette couche, faute de quoi la profusion de l'information obtenue ne permet pas de comprendre le système correctement. C'est cette technique que nous utilisons pour analyser un ORB dans la section qui suit.

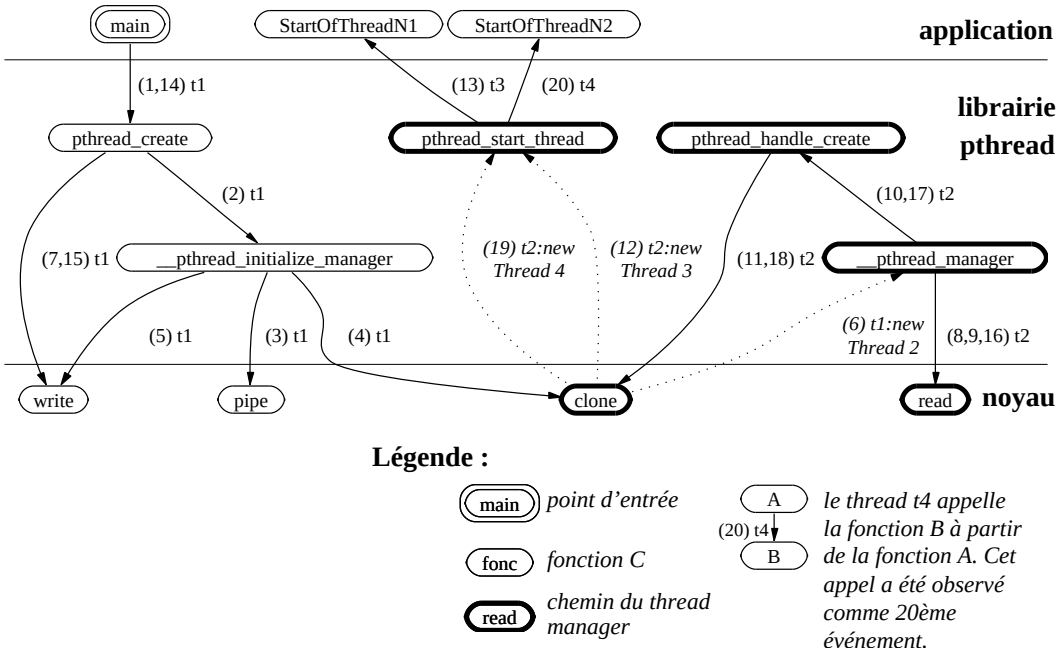


FIG. 5.16 : *Mécanisme de création de brins avec le noyau* LINUX *2.4*

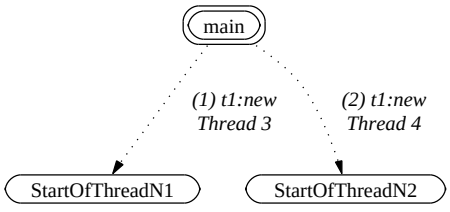


FIG. 5.17 : *Le comportement du programme de la fig. 5.13 page 98 après abstraction*

5.4 Cas d'étude & prototypage sur un ORB commercial

Nous présentons dans cette section un prototype qui implémente partiellement la méta-interface développée au cours de la section 5.2 (voir la figure 5.9 page 93). Nous avons réalisé ce prototype sur une plate-forme comprenant un bus à objets CORBA du commerce, ORBACUS de la société IONA, et un système d'exploitation en source libre, GNU/LINUX. Ces deux composants, ORBACUS et GNU/LINUX, sont aujourd'hui couramment utilisés dans l'industrie, et ce choix nous permet donc de valider la faisabilité et l'intérêt de notre démarche pour la pratique industrielle.

5.4.1 Construction d'un méta-modèle de l'ORB

Dans la section 5.2.1 page 80, nous avons développé un modèle générique des liens entre un bus à objet CORBA et un système d'exploitation POSIX (voir la figure 5.3 page 81). C'est ce modèle que nous avons ensuite utilisé au long de notre analyse dans la section 5.2, pour déboucher finalement sur la spécification d'une méta-interface multi-niveaux pour la réplication d'une plate-forme CORBA/POSIX (`MetaRequestLifeCycle_V2` sur la figure 5.9). Pour appliquer les résultats de cette analyse à ORBACUS, nous devons maintenant relier le fonctionnement concret d'ORBACUS au modèle générique de la figure 5.3, qui a servi de base à notre travail.

Pour réaliser cette mise en correspondance, nous avons commencé par extraire un méta-modèle approprié des sources d'ORBACUS 4.1 en utilisant les techniques de la section précédente. C'est ce méta-modèle que nous présentons maintenant.

Organisation d'ORBACUS

ORBACUS sur GNU/LINUX utilise les primitives système de la norme POSIX, mais ajoute en interne, au-dessus de ces primitives, des couches d'abstraction supplémentaires, notamment pour le multitraitement et les communications réseau. La figure 5.18 donne un exemple de cette organisation en couches, en montrant comment les types `pthread_t` (descripteur de brin) et `pthread_mutex_t` (descripteur de verrou) sont (en partie) utilisés par les classes d'ORBACUS (version 4.1) [10]. On remarque sur cette figure l'organisation en couches du bus à objets, qui en interne utilise une bibliothèque spécifique appelée `JTC` (JAVA-*like threads for C++*) pour « transformer » les primitives de synchronisation et de multitraitement proposées par la norme POSIX en un modèle de programmation équivalent aux primitives du langage JAVA.

[10] Ce diagramme a été obtenu en utilisant une version modifiée de l'outil DOXYGEN pour produire une version XML de la structure d'ORBACUS puis par notre outil COSMOPEN pour en extraire ce graphe.

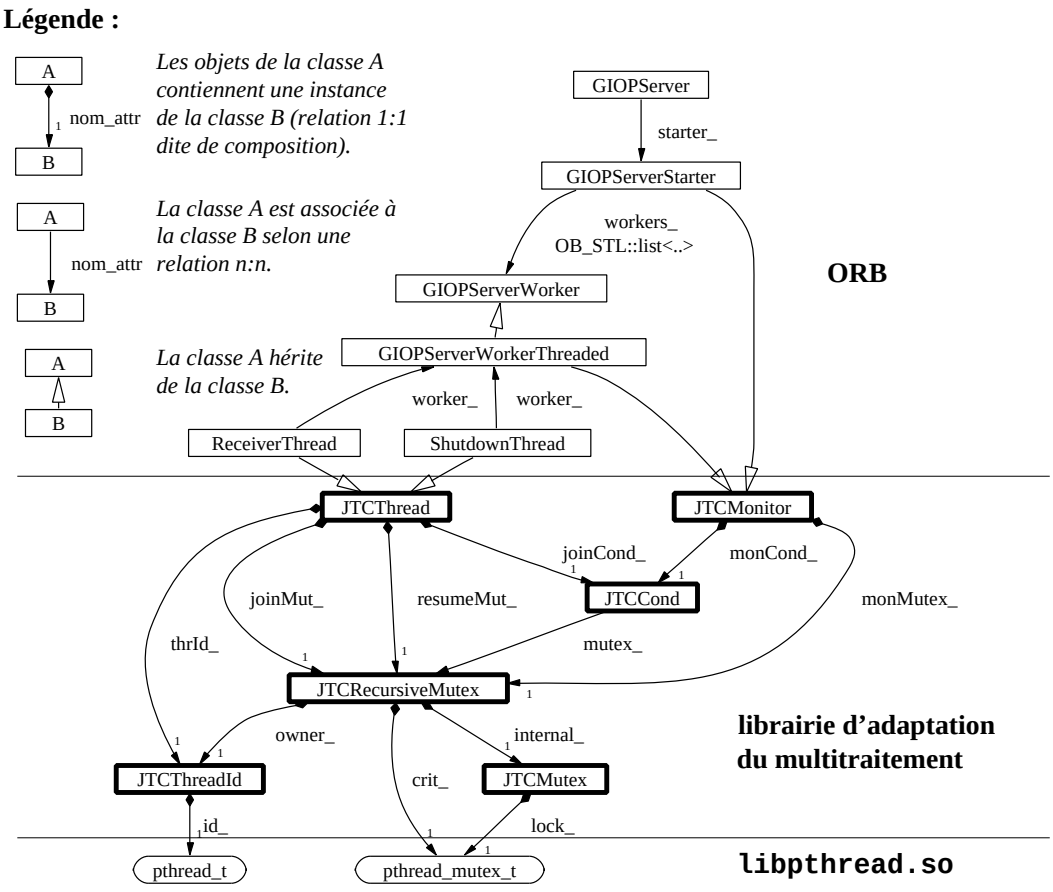


FIG. 5.18 : *Ramifications structurelles de la bibliothèque* `libpthread.so` *dans* ORBACUS

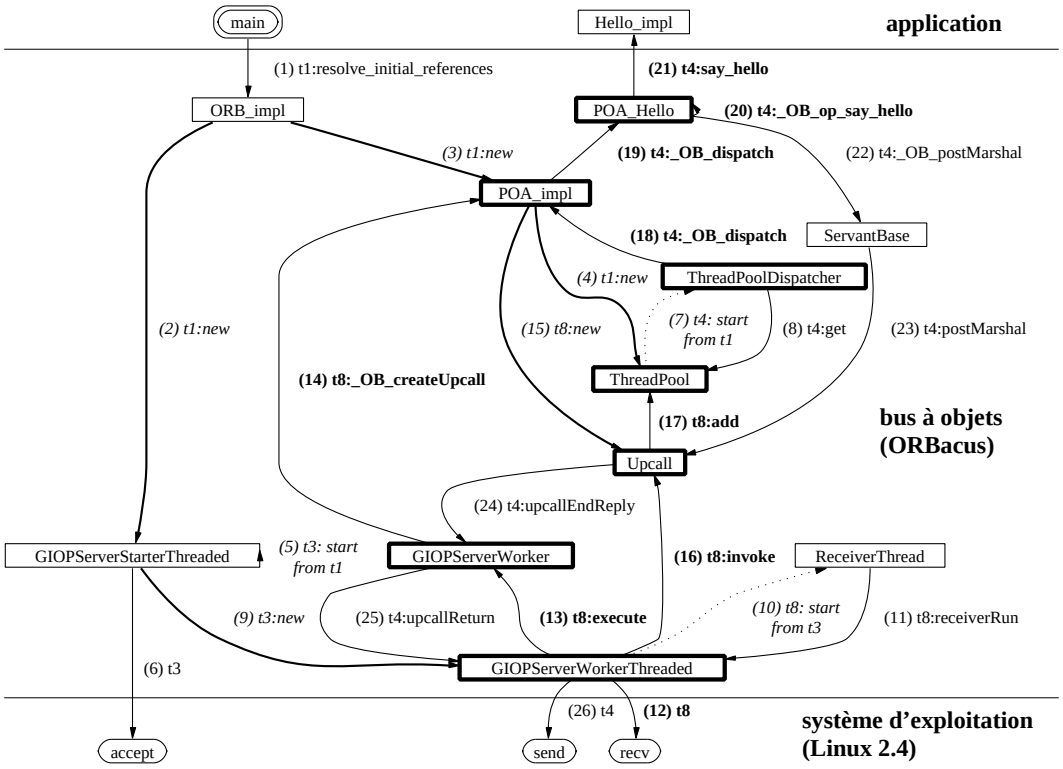
Modèle comportemental

La compréhension de la nature stratifiée d'ORBACUS est nécessaire pour ensuite pouvoir, à partir d'un ensemble de traces d'appels capturées par la technique décrite dans la section 5.3.2 page 97, « filtrer » les éléments relatifs à ces sous-couches et ainsi obtenir un modèle concis et facilement compréhensible du traitement des requêtes dans ORBACUS. Le modèle que nous obtenons par cette technique lorsqu'ORBACUS fonctionne en mode à réserve de brins est présenté sur la figure 5.19 page ci-contre. Cette figure se lit comme la figure 5.16 page 101. On remarquera sur la figure 5.19 l'absence d'appels relatifs au fonctionnement interne de la bibliothèque `libpthread.so` qui gère le multitraitement sous GNU/LINUX. De même, la bibliothèque JTC d'ORBACUS que nous mentionnons plus haut n'est plus visible. Tous ces appels ont été filtrés grâce à notre outil COSMOPEN (voir l'annexe B page 127 pour plus de détails).

Cette modélisation comportementale permet de visualiser les différentes étapes de l'initialisation et du traitement des requêtes dans ORBACUS. Les appels (1) à (8) correspondent à l'initialisation de l'ORB lui-même. Le brin principal crée d'abord *un brin d'acceptation*, le brin $t3$, dont l'activation se produit en (5)[11]. La présence d'un brin « accepteur » est liée à l'utilisation de communications orientées connexions par le protocole d'échange de requêtes IIOP défini par CORBA (*Internet Inter-ORB Protocol* [OMG 2002e]). Le protocole IIOP, requis par la norme, permet d'assurer l'interopérabilité de plusieurs implémentations CORBA entre elles, mais reste invisible aux utilisateurs de l'ORB, qui n'ont en particulier pas à se soucier de la gestion des connexions. Le standard CORBA n'imposant pas de gestion de connexions particulière, celle-ci relève en conséquent de la responsabilité de l'ORB qui doit gérer de manière transparente leur création, leur initialisation, et leur destruction. Le brin $t3$ est ici responsable de l'acceptation des demandes de connexion de la part de clients distants. Cette acceptation s'effectue par l'appel POSIX `accept` [appel (6) sur la figure]. La réserve de brins (représentée par la classe `ThreadPool`) est ensuite créée [appel (4)] ainsi que les brins qu'elle contient (création du brin $t4$, dont l'activation apparaît en (7), la création des autres brins de la réserve [$t5$ à $t7$] n'est pas représentée). Pour finir, les brins de la réserve appellent tous la méthode `ThreadPool::get(..)` qui les bloque en état d'attente sur une variable de condition (non représentée par soucis de concision). De manière concomitante, le brin receveur de connexions ($t3$) se bloque en attente de connexion [appel (6) à `accept` dont nous avons déjà parlé].

Les appels (9) à (21) représentent la réception puis la montée d'une requête de la couche de communication POSIX vers l'application. Après avoir reçu une demande de connexion [retour de l'appel (6)], le brin $t3$ crée un nouveau brin « récepteur » $t8$ [appel (10)] qui lit le contenu de la nouvelle requête sur la nouvelle connexion [appel POSIX (12) à `recv`]. Le brin $t8$ transporte ensuite cette requête jusqu'à la classe `ThreadPool`, où il l'ajoute à la queue

[11] La création de brin est dans la norme POSIX « asynchrone » : le brin $t1$ demande la création du brin $t3$ lors de la création de l'objet `GIOPServerStarterThreaded` [appel (2)], mais l'activité du nouveau brin $t3$ n'apparaît qu'en (5), après que $t1$ a eu le temps de continuer l'initialisation de l'ORB [appels (3) et (4)]. On note aussi l'absence du brin $t2$, le gestionnaire de brins de la bibliothèque de multitraitement, lequel a été « filtré » par abstraction comme nous l'avons expliqué page 99.



Légende :

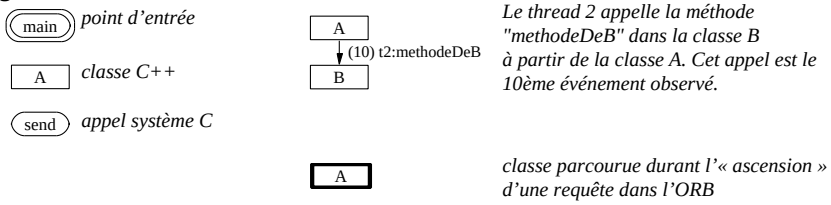


FIG. 5.19 : *Initialisation, réception et traitement d'une requête dans* ORBACUS

des requêtes en attente de traitement [appel (17) `t8:add`]. L'appel à `ThreadPool::add` débloque le brin de réserve $t4$, qui retourne de son appel à `ThreadPool::get`, et transporte la requête jusqu'au niveau applicatif [appel (21) à la méthode `say_hello`]. Le renvoi du résultat de cette requête est ensuite effectué par le brin $t4$ [appels (22) à (26), l'appel (26) appelant la primitive POSIX `send`].

Cette modélisation comportementale nous permet d'identifier les différents éléments du modèle générique de la figure 5.3 page 81. Les brins de réserve bloqués sur l'appel `ThreadPool::get` correspondent ainsi à la place T de la figure. Les appels POSIX `accept` et `recv` correspondent à la transition e. Les requêtes stockées dans l'instance de la classe `ThreadPool` correspondent à la place E. La place S en revanche n'est pas présente, le brin de la réserve qui traite la requête se chargeant de l'envoi de la réponse. La transition s est elle représentée par l'appel à `send`.

5.4.2 Principe de réalisation du prototype

La section précédente nous a permis de mettre en correspondance le modèle générique d'un ORB à réserve de brins proposé dans la section 5.2 page 78 avec la plate-forme concrète que nous avons choisie (GNU/LINUX et ORBACUS). Nous pouvons maintenant identifier dans ORBACUS et dans GNU/LINUX les points d'instrumentation que nous devons implémenter pour réaliser la méta-interface multi-niveaux de réplication `MetaRequestLifeCycle_V2` que nous avons proposée dans la section 5.2.5 page 93.

Pour des raisons de temps, nous avons choisi de n'implémenter que la seule maîtrise du non-déterminisme dans notre prototype, en nous limitant donc au sous-ensemble suivant de méthodes de la méta-interface :

```
requestBeforeApplication
requestAfterApplication
requestHasBeenReceived
replyHasBeenSent
requestBeforeContentionPoint
requestAfterContentionPoint
```

Dans cette partie très concrète de notre travail de thèse, notre souci a été de retarder le plus longtemps possible l'instrumentation propre du code de notre plate-forme expérimentale. En particulier, nous avons cherché à factoriser dans des bibliothèques génériques et réutilisables les mécanismes d'interception que nous avons utilisés. Dans ce but, nous avons appliqué une approche incrémentale, qui consiste d'abord à rendre réflexives les primitives de synchronisation et communication du niveau OS, puis, par agrégations successives, à obtenir les différentes méthodes recherchées de notre méta-interface. Ce travail a résulté en deux bibliothèques partagées en C++. La première bibliothèque, `libuspi.so` (2100 lignes de code), qui a été programmée sur la base d'une première implémentation en C par Ludovic Courtès, permet d'intercepter les primitives de communication et de synchronisation d'un

système POSIX. La seconde bibliothèque, `meta-corba.so` (1200 lignes de code), s'appuie sur les mécanismes de `libuspi.so` et implémente à proprement parler les méthodes de la méta-interface `MetaRequestLifeCycle_V2` que nous avons choisies de réaliser.

Le code de chacune de ces bibliothèques s'appuie exclusivement sur les standards CORBA et POSIX, et n'est donc propre ni à GNU/LINUX ni à ORBACUS. Le portage à un nouveau système d'exploitation s'opère par simple re–compilation (à condition cependant que celui-ci respecte la norme POSIX, ainsi que l'interface `dlopen`, disponible par exemple dans SUNOS et GNU/LINUX, et aujourd'hui largement répandue). L'instrumentation d'un nouveau bus à objet s'effectue par un « tissage » (en anglais *weaving*) manuel de points d'instrumentation dans le code de l'ORB, de façon extrêmement peu intrusive, comme nous le verrons.

Cette approche incrémentale, en privilégiant une instrumentation basée sur les interfaces du système (POSIX & CORBA), limite donc l'intrusion dans le code propre de la plate-forme concrète choisie (dans notre cas ORBACUS et GNU/LINUX), et garantit ainsi une très grande portabilité du canevas d'instrumentation ainsi formé.

5.4.3 La bibliothèque `libuspi.so` : réflexivité des primitives OS

Le sous ensemble de la méta-interface `MetaRequestLifeCycle_V2` que nous avons choisi d'implémenter cible essentiellement le cycle de vie des requêtes CORBA (réception, traitement, réponse), et la réification des activités de synchronisation de l'OS (prise et libération de verrous) liées à ce cycle de vie. Au niveau du système d'exploitation, le premier pas consiste donc à pouvoir intercepter et modifier le comportement des primitives de synchronisation et de communication du standard POSIX.

```
class MetaMutex {
public:
  int pthread_mutex_init      (pthread_mutex_t *mutex, ..) ;
  int pthread_mutex_lock      (..) ;
  int pthread_mutex_unlock    (..) ;
  [..]
};
void SetMetaMutexForMutex(MetaMutex*, pthread_mutex_t*);
```

FIG. 5.20 : *La classe `MetaMutex`*

Pour atteindre cet objectif, nous avons utilisé une méthode largement répandue dite par interposition de bibliothèque (*library shadowing*) [Levine 1999], et disponible sur la plupart des plates-formes POSIX. Cette méthode nous a permis de définir deux méta-classes, `MetaSocket` et `MetaMutex`. Ces méta-classes permettent de modifier de manière sélective le comportement de chaque verrou ou connexion d'un programme en lui associant une instance appropriée de l'une d'entre elles. L'extrait de code de la figure 5.20 donne un aperçu de l'interface de `MetaMutex` et de la méthode `SetMetaMutexForMutex(..)` qui

permet d'associer un verrou à un méta-verrou. De même une fonction `SetMetaSocketForSocket(..)` assure la même opération pour les `MetaSocket`.

Transcendance et contextes sémantiques

Les classes `MetaMutex` et `MetaSocket` permettent de sélectionner individuellement les connexions et les verrous d'un programme dont les appels seront réifiés et modifiés au méta-niveau. Cependant, comme nous l'avons expliqué dans la section 5.2.3 page 87, nous ne souhaitons réifier que l'activité de *certains* verrous, qui correspondent à des points de contention pour les requêtes de l'ORB. De même, nous ne sommes intéressés que par les connexions véhiculant des communications CORBA, alors qu'un programme peut utiliser des connexions pour de nombreuses autres activités. Il nous faut donc, pour pouvoir utiliser les classes `MetaMutex` et `MetaSocket`, trouver un moyen pour *sélectionner de manière portable et peu intrusive les connexions et les verrous d'un ORB qui sont pertinents pour la réplication* .

Pour répondre à ce problème nous introduisons les notions de *transcendance logicielle* et de *contexte sémantique*. Nous empruntons au vocabulaire de la philosophie la notion de transcendance qui représente le

> *« caractère de ce qui se situe au-delà d'un domaine pris comme référence, qui est au-dessus et d'une nature radicalement supérieure ».*
>
> *[ATILF 2003]*

La notion de *transcendance logicielle* renvoie au fait que, dans une organisation en couches, une opération à un niveau donné ne prend son sens que dans le contexte plus large de l'activité des niveaux supérieurs. L'ouverture d'une connexion peut par exemple être l'une des étapes de l'invocation d'une requête CORBA, mais elle peut aussi correspondre à l'envoi d'une requête vers un serveur graphique X-WINDOWS. Or ces deux situations sont complètement opaques pour quelqu'un qui n'observe le système qu'au niveau de sa couche réseau et n'a connaissance ni de la norme CORBA, ni du protocole X11. Le même phénomène se retrouve dans l'utilisation des verrous. Dit autrement, les couches les plus basses réalisent des actions dont elles ne peuvent comprendre le sens global pour le reste du système. La logique globale du système *transcende* les couches de bas niveau, dont l'activité prend dans le système *un sens plus large* que leur simple sémantique individuelle.

Le concept de transcendance va de pair avec ce que nous appellerons le *contexte sémantique* d'une opération. Ce « contexte sémantique » représente l'endroit dans le logiciel, souvent situé dans les niveaux supérieurs, où le sens pour le système d'une opération de bas niveau devient explicite pour un observateur extérieur. Pour pouvoir décider, lorsqu'une connexion ou un verrou est créé, si son comportement doit être réifié, nous avons besoin de connaître le contexte sémantique de cette création (la création d'une connexion s'opère par l'appel POSIX `socket(..)` ; celle d'un verrou par `pthread_mutex_init(..)`).

La figure 5.21 page ci-contre illustre cette situation pour l'appel POSIX `pthread_mutex_init()`, qui permet de créer un verrou exclusif. Le contexte sémantique **A** peut

par exemple correspondre à la création d'un verrou pour la gestion d'un compteur de références, et le contexte **B**, à la création d'un verrou pour la gestion des requêtes en attente de traitement. Le verrou créé dans le contexte **A** n'a pas besoin d'être réifié puisqu'il ne constitue pas ce que nous avons appelé un *point de contention* pour les requêtes. En revanche les opérations sur le verrou créé dans le contexte **B** doivent être réifiées au méta-niveau en utilisant une `MetaMutex` adaptée. La figure 5.21 montre que du fait de l'organisation interne d'un ORB, le contexte sémantique d'un appel peut se révéler relativement éloigné dans la pile d'invocations qui le provoque. ORBACUS par exemple utilise les primitives de synchronisation de la bibliothèque `JTC` dont nous avons parlé page 102 (voir notamment la figure 5.18 page 103). Or les deux verrous créés à partir des contextes **A** et **B** sont indistinguables dans cette bibliothèque (ceci est représenté sur la figure par le fait que les deux créations utilisent la même suite d'appels emboîtés). Il est donc impossible de décider *dans* la bibliothèque `JTC` si un verrou doit être réifié ou non.

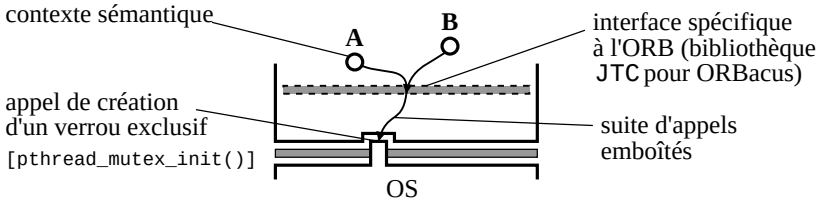


FIG. 5.21 : *Contextes sémantiques d'un appel de bas niveau (ici `pthread_mutex_init()`)*

Afin de pouvoir faire circuler le contexte sémantique d'un appel OS à travers les différentes couches qui séparent l'appel de son contexte, nous nous sommes inspirés des mécanismes de régulation qui existent en biologie végétale entre les racines et les feuilles d'une plante. Par exemple, en période de canicule,

> *« [l]es racines [d'une plante] sont capables de percevoir l'état de sécheresse du sol. Elles synthétisent alors une hormone de stress, l'acide abscissique, qui est véhiculée jusqu'aux feuilles par la sève et qui ferme les stomates, ce qui limite les pertes en eau. »*
>
> *Thierry Simonneau, cité dans [Galus 2003]*

Les brins d'exécution constituent la « sève » d'un système logiciel complexe puisqu'ils « transportent » données et contrôle d'un niveau d'abstraction à l'autre, et permettent à un programme de « prendre racine » dans son environnement d'exécution. Nous avons donc choisi, à la manière de l'acide abscissique chez les plantes, de faire circuler des *méta-marqueurs* porteurs de sens sur les brins de notre ORB[12]. Ces méta-marqueurs (représentés par la classe `MetaThreadInfo` page suivante) circulent en « contrebande » des paramètres du programme original, et permettent de fournir de manière portable et non-intrusive des

[12] L'ajout transparent de données à un brin d'exécution est une fonctionnalité standard des modèles de programmation à activités multiples. Cette fonctionnalité est par exemple assurée dans la norme POSIX par la fonction `pthread_setspecific()`.

informations aux couches de bas niveaux sur le contexte dans lequel elles sont utilisées. Un tel ajout d'informations est très proche du traçage de causalité par *talonnage* (*piggybacking*) sur les messages d'un système réparti, très utilisé dans les algorithmes distribués (par exemple dans [Baldoni et al. 1997, Baldoni et al. 1998b]).

```
class MetaThreadInfoType ;
class MetaThreadInfo {
public:
  MetaThreadInfo(MetaThreadInfoType* someType) ;
  void attachToSelfThread() ;
  void detachFromSelfThread() ;
};
```

FIG. 5.22 : *La classe* `MetaThreadInfo`

Usines à méta-verrous et à méta-connexions

Nous utilisons la classe `MetaThreadInfo` que nous venons d'introduie pour ajouter aux classes `MetaMutex` et `MetaSocket` la capacité de se fixer sur un brin d'exécution. Par héritage multiple nous obtenons alors deux nouvelles « méta-classes », `ThreadMetaSocket` et `ThreadMetaMutex` (figure 5.23 ci-dessous).

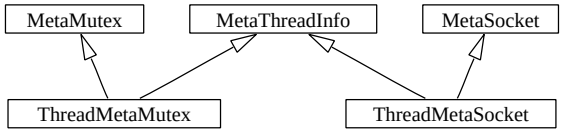


FIG. 5.23 : *Diagramme de classes des méta-objets du niveau POSIX*

`ThreadMetaSocket` et `ThreadMetaMutex` combinent les caractéristiques de `MetaThreadInfo` (pouvoir se fixer sur un brin) à celle de `MetaSocket` et `MetaMutex` (pouvoir intercepter et modifier les appels de l'OS relatifs à la gestion des verrous et à la gestion des connexions). `ThreadMetaMutex` par exemple est implémentée de telle sorte qu'elle intercepte toutes les créations de verrous effectuées par le brin auquel elle est attachée. Par défaut `ThreadMetaMutex` ne modifie pas la sémantique de création des verrous POSIX (la création est transmise à l'appel `pthread_mutex_init` d'origine), mais en modifiant par héritage le comportement de `ThreadMetaMutex` (figure 5.24 page suivante), il devient possible d'imposer, par exemple, que tous les verrous créés par un brin donné durant une section particulière de code soient associés à une `MetaMutex` spécifique au moment de leur création (sur la figure 5.24, tous les verrous créés sont associés à une instance de la classe `SpecificMetaMutex`, qui interceptera à l'avenir leurs opérations).

```
class ModifiedThreadMetaMutex : public ThreadMetaMutex {
public:
  pthread_mutex_init( pthread_mutex_t *mutex, ..)
  {
    // A- Comportement par défaut : l'appel
    //    pthread_mutex_init d'origine
    int result = ThreadMetaMutex::pthread_mutex_init( mutex, ..) ;

    // B - Les opérations sur le verrou nouvellement créé (mutex) seront
    //     dorénavant interceptées par une instance de SpecificMetaMutex.
    SetMetaMutexForMutex ( new SpecificMetaMutex, mutex ) ;

    return result ;
  }
} ;
```

FIG. 5.24 : *Principe de fonctionnement d'une usine à méta-verrous*

La classe `ModifiedThreadMetaMutex` de la figure 5.24 crée des méta-verrous sur les nouveaux verrous créés par un brin. C'est donc *une usine à méta-verrous*. De la même manière, `ThreadMetaSocket` peut être utilisée pour définir des *usines à méta-connexions*, qui associent des méta-connexions aux nouvelles connexions créées par un brin.

La période durant laquelle un brin est contrôlé par *une usine à méta-verrous* ou *une usine à méta-connexions* est modulable, en utilisant les capacités d'attachement et de détachement de `MetaThreadInfo` (figure 5.22 page ci-contre). `ThreadMetaSocket` et `ThreadMetaMutex` forment donc un canevas particulièrement puissant pour contrôler quels verrous et quelles connexions intercepter dans le système (ce que Kiczales appelle le contrôle d'impact, dont nous avons parlé page 28). Ce canevas forme la base de notre méta-interface multi-niveaux, que nous détaillons dans la section suivante.

5.4.4 La bibliothèque `meta-corba.so` : la méta-interface multi-niveaux

Comme nous l'avons expliqué, la seconde bibliothèque d'instrumentation, `meta-corba.so`, utilise les méta-classes de `libuspi.so` pour implémenter à proprement parler le méta-modèle multi-niveaux pour la réplication que nous avons développé dans la section 5.2 page 78. Un élément essentiel de ce méta-modèle sont les points de contention des requêtes (voir page 88), qui permettent de relier l'activité de synchronisation de l'OS (prise et libération de verrous), avec le cycle de vie des requêtes CORBA. La réification de ces points de contention requiert de pouvoir sélectionner les verrous dans l'ORB qui déterminent l'ordre de traitement des requêtes. Nous utilisons pour cela la classe `ThreadMetaMutex` de la bibliothèque `libuspi.so` que nous venons de présenter.

Comme nous venons de l'expliquer, lorsqu'une instance de `ThreadMetaMutex` est associée à un brin, elle intercepte toutes les opérations de création de verrou que peut faire ce brin. Pour ne sélectionner que les verrous qui représentent des points de contention pour les requêtes, il suffit donc de spécialiser `ThreadMetaMutex` en une « usine à méta-verrous » qui redéfinisse `pthread_mutex_init(..)` (nous avons appelé cette usine `ContentionPointFactory`).

```
int ContentionPointFactory::pthread_mutex_init( pthread_mutex_t *mutex,..){
  int result = ThreadMetaMutex::pthread_mutex_init( mutex, ..) ;
  RequestContentionPoint *
    newContentionPoint = new RequestContentionPoint(myMetaLifeCyle) ;
  SetMetaMutexForMutex ( newContentionPoint, mutex ) ;
  return result ;
}
```

FIG. 5.25 : *Comment les verrous devant être réifiés sont sélectionnés*

La redéfinition de `pthread_mutex_init(..)` dans la classe `ContentionPointFactory` permet d'associer une `MetaMutex` particulière à chaque nouveau verrou que crée le brin contrôlé par `ContentionPointFactory` (figure 5.25). En instanciant à chaque fois une spécialisation de `MetaMutex` (que nous appellerons `RequestContentionPoint`) qui réifie les opérations d'allocation de verrou vers un objet `MetaRequestLifeCycle`, nous réalisons de cette manière les méthodes `requestBeforeContentionPoint` et `requestAfterContentionPoint` de la méta-interface que nous cherchions à implémenter. Le diagramme de classe final obtenu est représenté sur la figure 5.26.

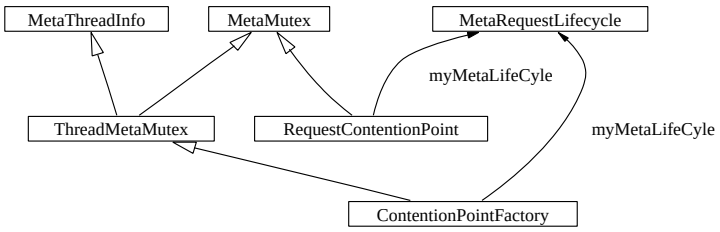


FIG. 5.26 : *Diagramme d'héritage de `RequestContentionPoint`*

Exemple d'utilisation de `meta-corba.so` sur ORBACUS

Munis de la classe `ContentionPointFactory` nous pouvons maintenant instrumenter le code d'ORBACUS en identifiant dans les sources l'endroit où sont initialisés les verrous qui protègent les opérations `add` et `get` de la classe `ThreadPool` que nous avions présentée lors de notre explication du fonctionnement d'ORBACUS page 104. Le morceau de code instrumenté est le suivant :

```
aContentionPointFactory.attachToSelfThread();   // instrumentation
pools_[i] = new ThreadPool(i, nthreads);        // code original
aContentionPointFactory.detachFromSelfThread(); // instrumentation
```

Autres points d'instrumentation

Nous ne pouvons, faute de place, détailler l'implémentation par la bibliothèque `meta-corba.so` des méthodes restantes de la méta-interface `MetaRequestLifeCycle_V2`. Pour la réification des entrées et sorties de requêtes, nous avons utilisé la même méthode que pour les points de contention, en utilisant un objet « usine » (*factory*) qui s'accroche temporairement au brin d'initialisation. Cette interception nous permet de rajouter par talonnage des identifiants de requête uniformes sur toutes les répliques (nous utilisons un canevas de talonnage générique qui ne dépend pas du protocole GIOP utilisé par CORBA). Nous utilisons de nouveau la classe `MetaThreadInfo` pour permettre à ces identifiants de voyager sur les brins d'exécution au travers des différentes couches de l'ORB. Seule la queue des requêtes en attente gérée par la classe `ThreadPool` pose ici problème, puisque les requêtes de cette queue ne sont portées par aucun brin. Nous avons donc dû légèrement modifier les structures chaînées utilisées dans ORBACUS pour permettre la transmission des identifiants de requête du brin récepteur ($t8$ sur la figure 5.19 page 105) au brin de réserve ($t4$ sur la même figure). L'utilisation d'identifiants uniformes sur toutes les répliques est essentielle à notre approche, puisqu'elle permet de s'abstraire des brins sous-jacents. Il n'est plus nécessaire de forcer toutes les répliques à utiliser les mêmes brins pour les mêmes requêtes. En tout, nos modifications ajoutent 24 nouvelles lignes au code d'ORBACUS, ce qui est extrêmement peu intrusif (0,02% du code original de la bibliothèque partagée d'ORBACUS).

La réalisation des méthodes `requestBeforeApplication` et `requestAfterApplication` a été la plus délicate, en nous obligeant à modifier le compilateur IDL de l'ORB pour directement instrumenter le scion (*skeleton*) utilisé par ORBACUS (la classe `POA_Hello` dans notre exemple page 105). Mais ces modifications n'excèdent pas 15 lignes (0,016% du code original du compilateur IDL d'ORBACUS)).

5.4.5 Résultats et conclusion

La trace des activités du serveur CORBA obtenue par notre prototype de la méta-interface multi-niveaux `MetaRequestLifeCycle_V2` est représentée sur la figure 5.27 page suivante. On y reconnaît l'interception de la création d'une connexion (la connexion 8), à laquelle une méta-connexion appropriée est adjointe. Les identifiants 6439 et 6427 sont ceux respectivement du brin récepteur (6439) et de la réserve (6427) qui traitent la requête.

La figure 5.27 fait apparaître deux points de contention (le point de contention `1`, et le point de contention `2`). L'existence de deux points de contention tient à l'implémentation

```
Initialization OK.
(meta-corba) New meta-socket for socket 8
(meta-corba) In LAAS::Generic_Piggybacker::accept
(meta-corba) (6439) Received Request with ID: 1:6436
(meta-corba) (6439) Before Contention Point 1 with RequestID: 1:6436
(meta-corba) (6439) After  Contention Point 1 with RequestID: 1:6436
(meta-corba) (6439) Before Contention Point 2 with RequestID: 1:6436
(meta-corba) (6439) After  Contention Point 2 with RequestID: 1:6436
(meta-corba) (6427) Before Contention Point 1 with RequestID: 1:6436
(meta-corba) (6427) After  Contention Point 1 with RequestID: 1:6436
(meta-corba) (6427) Before application with RequestID: 1:6436
Hello World!: 1
(meta-corba) (6427) After application with RequestID: 1:6436
(meta-corba) (6427) Sending Reply for Request with ID: 1:6436
(meta-corba) Destroying a meta-socket Generic_Piggybacker
```

FIG. 5.27 : *Traçage de l'activité d'*ORBACUS *par notre prototype*

particulière de la bibliothèque de synchronisation d'ORBACUS (JTC, voir page 102). Le point 1 est touché deux fois par la requête en cours de traitement : une fois alors qu'elle est portée par le brin récepteur, une autre fois lorsqu'elle est prise en charge par le brin de la réserve qui la traite. Le point 2 est lui touché une fois lors du traitement de la requête, par le brin récepteur. En tout, il s'agit donc de **trois** décisions de synchronisation durant le traitement d'une requête qui doivent être répliquées pour assurer le déterminisme. Notre analyse garantit par ailleurs que ces décisions sont suffisantes pour assurer le contrôle du non-déterminisme dans ORBACUS. Si nous rapportons ces trois décisions aux 203 opérations de synchronisations effectuées par ORBACUS pour chaque requête, nous obtenons donc **un gain d'un rapport de 67** entre le nombre d'interceptions dans une approche mono-niveau ciblant uniquement le système d'exploitation et l'approche multi-niveaux implémentée par notre prototype (ce gain est à mettre en parallèle avec les performances reportées par Napper *et al.* dans [Napper et al. 2003], que nous avons mentionnées page 86).

En plus de ce gain très important, cette implémentation illustre comment le caractère multi-niveaux des informations capturées par la méta-interface se retrouve dans la méthode de réalisation incrémentale que nous avons proposée. Cette première réalisation n'est qu'un prototype, mais valide notre approche, en montrant comment de manière portable, générique et extrêmement peu intrusive, une méta-interface multi-niveaux POSIX/CORBA peut être réalisée pour maîtriser le non-déterminisme dû au multitraitement.

5.5 Conclusion

Dans ce dernier chapitre, nous avons illustré, en partant d'un ensemble de mécanismes de réplication bien connus, comment la démarche de développement d'une méta-interface multi-niveaux proposée à la fin du chapitre 4 pouvait être mise en œuvre. Nous avons ainsi pu montrer sur une architecture concrète CORBA/POSIX l'intérêt de la notion d'*empreinte réflexive*, et l'importance de l'étude des *liens inter-niveaux* pour le développement d'une méta-interface multi-niveaux.

Nous avons aussi abordé les aspects liés à la réalisation d'un prototype de cette méta-interface. Nous avons notamment présenté une méthode générale d'extraction de méta-modèle à partir des sources d'un composant. Cette méthode d'extraction nous a permis d'implémenter notre prototype de manière portable, générique, et extrêmement peu intrusive sur l'OS GNU/LINUX et le bus à objets commercial ORBACUS. Ce prototype valide la pertinence de notre approche sur des implémentations utilisées dans l'industrie.

Conclusion et perspectives

A good lesson when you fly COTS stuff - make sure you know how it works.

Glenn E. Reeves,
MARS PATHFINDER *Flight Software Cognizant* Engineer
[Reeves 1997]

LE 4 juillet 1997 la sonde MARS PATHFINDER de la NASA se posait sur la planète rouge. Quelques jours plus tard le système de contrôle de la sonde commençait à présenter des redémarrages incontrôlés, ralentissant gravement la mission [Reeves 1997]. Le problème se trouva être une inversion de priorité[13] causée par un verrou *interne* à l'OS utilisé (VXWORKS de la société WIND RIVER). Voici ce qu'écrit Glenn E. Reeves (le responsable du logiciel embarqué sur PATHFINDER) à propos de la solution trouvée [Reeves 1997] :

> *Once we understood the problem the fix appeared obvious : [...] enable the priority inheritance. [*WIND RIVER*] suppl[ies] global configuration variables for [such customization] (although this is not documented and those who do not have* VXWORKS *source code or have not studied the source code might be unaware of this feature).*

Dans notre travail de thèse nous nous sommes particulièrement intéressés à la mise en œuvre de la tolérance aux fautes dans les systèmes logiciels complexes de façon transparente et adaptable. Nous avons montré la nécessité d'aborder un système complexe selon l'ensemble des niveaux d'abstraction qui le composent, et en particulier, nous avons conclu à l'importance *d'une ouverture combinée* des différents composants d'un tel système.

L'anecdote par laquelle nous avons ouvert cette conclusion, l'inversion de priorité survenue dans la sonde PATHFINDER, montre que la nécessité d'une ouverture partielle des composants d'un système complexe va au delà de la seule tolérance aux fautes, et touche

[13] Une inversion de priorité se produit lorsque, dans un système temps réel, une tâche (similaire à un brin d'exécution) de haute priorité se trouve bloquée par un verrou que possède une tâche de basse priorité. Le retard de la tâche de haute priorité entraîne alors une violation des échéances temps réel du système. L'inversion de priorité se résout en utilisant un mécanisme dit d'*héritage de priorité* (*priority inheritance*).

d'autres aspects non-fonctionnels, comme la maîtrise du temps-réel. La cause de l'inversion de priorité dans la sonde PATHFINDER — le partage d'un verrou entre différentes primitives de communication à *l'intérieur* de VXWORKS — et la solution choisie par la NASA — l'utilisation d'options de configuration non–documentées — illustrent l'importance d'une connaissance interne des composants utilisés dans un système pour une maîtrise globale de ses aspects non-fonctionnels.

Aujourd'hui, dans les cas les plus critiques, des ingénieurs du fournisseur d'un composant sont « loués » à l'intégrateur, et servent ainsi de *vecteur implicite* de méta-informations relatives à l'implémentation du composant. Lors du développement de PATHFINDER, la NASA a ainsi pu travailler très étroitement avec WIND RIVER pour adapter l'OS VXWORKS à ses besoins. Cette démarche n'est pas envisageable pour des projets plus petits, ou lorsque les contraintes sont moins fortes et les ressources disponibles plus limitées. Pour rendre cet objectif largement accessible, même aux organisations qui n'ont ni les moyens ni le prestige de la NASA, nous pensons que les composants d'un système logiciel complexe ne peuvent plus être utilisés comme de simples boîtes noires, fondées une séparation stricte entre interface fonctionnelle et implémentation.

Aujourd'hui, dans les systèmes informatiques complexes, une ouverture partielle des composants logiciels utilisés, réalisée de façon cohérente et coordonnée, est nécessaire pour allier maîtrise des aspects non-fonctionnels et maîtrise de la complexité.

Résumé de nos travaux de thèse

Réalisé sans garde-fous, cet abandon (partiel) du principe d'encapsulation est, toutefois, porteur de dangers. En rendant visibles certains aspects d'implémentation des composants d'un système, l'on casse la séparation nette entre interface et réalisation. De nouvelles interdépendances, de nouveaux canaux de propagation du changement peuvent potentiellement être introduits, neutralisant par là même les capacités d'évolution de l'application. Mal maîtrisées, ces nouvelles dépendances peuvent même mettre en danger l'intégrité du système (l'on pensera au vol inaugural d'Ariane 5, dont nous avons parlé dans la section 1.2.1 page 9).

Dans notre travail de thèse, nous avons choisi d'utiliser la réflexivité pour permettre cette ouverture partielle des composants d'un système complexe (chapitre 2 page 19). En explicitant, de façon méthodique, par le biais d'une méta-interface et d'un méta-modèle, les aspects d'un composant qui sont rendus visibles, l'approche réflexive permet de contrôler cette ouverture, et d'en éviter les écueils. La réflexivité avait déjà fait ses preuves pour la réalisation transparente de la tolérance aux fautes sur de *petits* systèmes. Nous avons donc étudié dans quelle mesure, et par quels moyens ces résultats pouvaient être étendus à des systèmes logiciels complexes, multi-composants et multi-couches (chapitre 3 page 35).

Pour cela, nous avons commencé par constater la complémentarité des différents niveaux d'abstraction dans lesquels s'organisent les composants d'un système complexe, et avons ainsi justifié pourquoi l'ouverture des composants d'un tel système nécessitait d'appréhender l'ensemble des niveaux d'abstraction rencontrés (section 3.3 page 50) :

- Les composants des hauts niveaux manquent en effet d'informations sur l'activité des couches basses du système. Leur seule ouverture ne permet donc pas la réalisation de mécanismes de tolérance aux fautes puissants et complets.
- De façon symétrique, les composants des bas niveaux ne fournissent pas suffisamment de sémantique sur l'activité du système. Les approches se limitant à l'instrumentation de ces bas niveaux résultent, dans les systèmes complexes, en des mises en œuvre particulièrement inefficaces de la tolérance aux fautes, allant jusqu'à remettre en question leur intérêt.

Pour utiliser la réflexivité tout en prenant en compte l'organisation stratifiée des composants d'un système complexe, nous avons donc proposé un nouveau cadre conceptuel, la *réflexivité multi-niveaux* (chapitre 4 page 53). Les différentes notions que nous avons présentées, les *liens inter-niveaux* (section 4.1.3 page 56), les *empreintes réflexives* (section 4.2.1 page 59), la *transcendance logicielle* (section 5.4.3 page 108), sont un premier pas pour permettre de combiner les méta-modèles exportés par les différents composants d'un système en une perception globale de ce système. Dans le chapitre 5, nous avons validé la pertinence pratique de notre proposition en abordant la réplication d'une plate-forme constituée de composants utilisés couramment dans l'industrie : le système d'exploitation libre GNU/LINUX et l'ORB commercial ORBACUS.

Cette validation pratique nous a permis de montrer qu'en explicitant en une *empreinte réflexive* suffisamment large les besoins opératoires d'un ensemble de stratégies de tolérance aux fautes, nous permettions à *tout* système implémentant cette *empreinte* d'adapter ses mécanismes de tolérance aux fautes. Pour cela, nous avons commencé par présenter l'*empreinte réflexive* des mécanismes de réplication que nous avions choisis. Nous avons ensuite proposé un modèle générique d'un bus à objet CORBA à réserve de brins (figure 5.3 page 81). Ce modèle générique nous a permis de construire une *méta-interface multi-niveaux* des interactions entre l'ORB et le système d'exploitation correspondant à l'*empreinte réflexive* que nous avions obtenue (section 5.2.5 page 93). Un caractère essentiel de la méta-interface multi-niveaux ainsi construite tient à sa *généricité* : elle ne s'appuie que sur les standards CORBA et POSIX et ne présuppose pas d'implémentation particulière d'ORB (TAO, OMNIORB, ORBACUS) ou d'OS (GNU/LINUX, SOLARIS). Cette méta-interface pour la réplication englobe donc à la fois une famille de mécanismes de réplication, mais aussi toute une famille de plates-formes, dès que celles-ci utilisent les interfaces CORBA et POSIX qui sont à sa base.

Nous avons reflété ce caractère générique dans l'implémentation de cette méta-interface, en la réalisant au travers de deux bibliothèques C++ d'interception réutilisables sur différentes plates-formes (page 106).

Nous avons utilisé ces deux bibliothèques pour instrumenter notre plate-forme expérimentale GNU/LINUX–ORBACUS, en commençant par extraire des modèles structurels et

comportementaux des composants utilisés (section 5.4.1 page 102), pour en déduire les points pertinents d'instrumentation. Nous avons, pour cette activité de rétro-conception, utilisé l'outil COSMOPEN, spécifiquement développé dans le cadre de notre thèse, que nous décrivons plus en détail dans l'annexe B page 127.

Cette instrumentation nous a permis de valider le bien-fondé de notre proposition, puisque tout en restant très peu intrusive (un peu moins de 0,02% du code original), elle nous a permis dans le cas du bus à objets ORBACUS de diviser par 67 le nombre d'interceptions nécessaires à la maîtrise du non-déterminisme, comparativement à une approche mono-niveau qui n'aurait ciblé que le seul système d'exploitation (page 112 et suivantes).

Leçons et perspectives

L'extraction de méta-modèles, à la base de notre cas d'étude, en plus d'exiger un effort certain de rétro-conception, n'aurait cependant pas été possible si nous n'avions pas eu accès aux sources des composants en question. Dans la perspective de systèmes réalisés par assemblage de composants préexistants, se pose donc la question de savoir comment éviter cette étape de rétro-conception afin de rendre utilisable en pratique la démarche ébauchée dans cette thèse :

Nous pensons que la piste la plus pertinente pour la maîtrise des aspects non-fonctionnels des systèmes complexes consistera à l'avenir à inclure dans tout composant un méta-modèle « transversal » pour donner accès aux détails pertinents de son implémentation de façon *standardisée, disciplinée, et cohérente*.

Ce type de modélisation comportementale de haut niveau, livrée avec le composant, existe déjà dans d'autres domaines connexes à l'informatique, comme la micro-électronique, où l'intégration système représente une étape critique du développement [Goldstein 2002]. Cette avance de la synthèse de circuits sur le génie logiciel en matière de méta-modélisation s'explique par : (1) l'importance grandissante de la modularité et de la réutilisation dans ce domaine (des entreprises comme ARM LTD. ne vendent que des morceaux de plan de circuit sur étagère, appelés « *intelectual property blocks* » ou plus couramment « *IP blocks* » en anglais) ; et (2) l'impossibilité de « modifier » une conception silicium défaillante une fois le circuit produit. Aujourd'hui, à l'image des intégrateurs silicium, et alors que l'informatique est sur le point de se fondre, à devenir invisible, dans notre environnement quotidien, l'heure est venue pour les intégrateurs de systèmes logiciels complexes de sauter le pas de la « composabilité transversale ».

Le méta-modèle destiné à la réplication que nous avons proposé dans le chapitre 5 page 75, ainsi que les bibliothèques partagées qui l'implémentent, s'appliquent à une famille de mécanismes de réplication (active, passive, semi-active), ainsi qu'à une famille de plates-formes concrètes (celles basées sur CORBA et POSIX). La forme d'implémentation

choisie, par bibliothèques partagées, permet de retarder le plus tard possible la prise en compte des particularités d'implémentation d'un composant (section 5.4.2 page 106).

La vision que nous proposons ici revient à extrapoler cette approche en supposant *des composants rendus réflexifs dès leur réalisation* par la publication, par leur fournisseur, sous une forme standardisée, d'un méta-modèle de leur fonctionnement. La présentation par le composant de son propre méta-modèle dans un formalisme standardisé permet d'assouplir l'alternative du tout ou rien en matière d'adaptation (le consultant en détachement chez l'intégrateur, ou aucune information) que nous dénoncions au début de cette conclusion, en automatisant l'échange d'informations entre organisations et entre équipes de développement comme le montre la figure 5.28 page suivante. En choisissant un canevas de description de méta-modèles suffisamment expressif, la réflexivité peut continuer à assurer un découplage puissant entre les choix spécifiques d'une implémentation et la réalisation de mécanismes transversaux tels que la tolérance aux fautes. La méta-interface réflexive assure alors au moment de l'exécution la découverte « à la volée » des propriétés internes du niveau de base par le méta-niveau, libérant les développeurs de ce fardeau. De cette façon, le paradigme réflexif permet de combiner ouverture et encapsulation, et assure des fondations solides à la démarche que nous proposons.

Cependant les exigences de généricité imposées sur les méta-modèles exportés gagne dans ce contexte une nouvelle acuité : comment choisir en effet les besoins qui doivent être pris en charge par ce méta-modèle ? La question posée est alors celle de l'échange d'informations entre fournisseurs et utilisateurs : (1) comment évaluer les besoins réflexifs ? (2) dans quel formalisme standardisé décrire les méta-modèles ?

Nous avons dans ce mémoire proposé la notion d'empreinte réflexive pour aborder la première question. La seconde question reste ouverte, et constituera sans doutes la clef de voûte du modèle à composants dont nous suggérons le développement. Idéalement, ce formalisme devrait permettre la construction de méta-modèles exhaustifs et concis, pour pouvoir répondre de manière complète et simple à toute question posée sur le composant modélisé. Paradoxalement, cette complétude implique de contenir au moins autant d'information que le composant lui-même, entraînant un effort poussé de structuration et d'organisation pour éviter d'obtenir un résultat aussi complexe et difficile à manipuler que le composant original. Un double écueil devra donc être évité dans le développement de ce nouveau méta-formalisme : un formalisme dont l'expressivité est trop restreinte risque de pêcher par manque d'informations nécessaires à la compréhension du système, et de limiter l'utilité des méta-modèles ainsi décrits. Un formalisme trop riche devient aussi trop complexe, trop difficile à utiliser, et source d'erreurs. Cette problématique rejoint les objectifs de canevas tels FRACTAL [Bruneton et al. 2002] ou le calcul formel KELL [Bidinger & Stefani 2003] qui ont déjà été proposés pour aborder de manière réflexive et récursive la composition dans les architectures à base de composants. Ces approches pionnières sont sans doute un excellent point de départ pour le changement de paradigme que nous défendons.

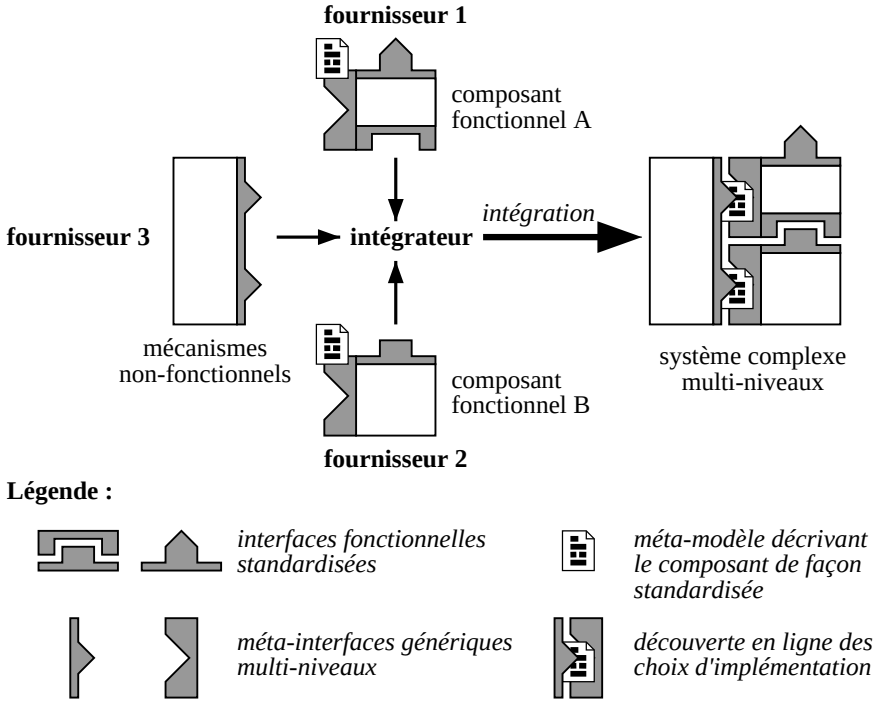


FIG. 5.28 : *Un nouveau modèle à composants réflexifs*

Annexe A

Le théorème d'incomplétude de Gödel

Une des premières et des plus importantes utilisations de la réflexivité dans le domaine de la logique et de ce qui à l'époque n'était pas encore l'informatique théorique se trouve dans la démonstration faite par Kurt Gödel (1906–1978) de son théorème d'incomplétude. Il nous a semblé très intéressant de dire quelques mots à son sujet pour montrer comment les notions présentées au chapitre 2 page 19 dans le contexte des langues naturelles et des langages de programmation s'appliquent aussi à des systèmes mathématiques formels.

En 1931, Gödel montrait qu'il n'était pas possible de construire un système axiomatique formel[1] pour raisonner sur l'arithmétique des nombres naturels qui permette de démontrer toutes les propositions vraies de l'arithmétique[2] [Gödel 1931]. Par manque de place, nous nous contenterons de quelques commentaires sur les aspects « réflexifs » de ce résultat fondamental, mais le lecteur intéressé trouvera dans [Nagel et al. 1989] une présentation très didactique de ce théorème ainsi qu'une traduction française de l'article original de Kurt Gödel.

Pour comprendre les travaux de Gödel, il est nécessaire de bien distinguer entre un système axiomatique formel (le signifiant) et son interprétation (le signifié). Un système axiomatique est une sorte de « jeu de construction » qui permet de générer un ensemble de chaînes de symboles selon des règles combinatoires bien définies. Le processus de construction de ces chaînes est purement formel, et à aucun moment il n'est nécessaire de donner une signification quelconque aux chaînes ainsi manipulées. Les chaînes de départ sont les

[1] C'est-à-dire un système dont les théorèmes puissent être énumérés de manière « mécanique » par une machine de Turing, par un « ordinateur ».

[2] Plus précisément, l'existence d'un tel système formel impliquerait la non–cohérence de l'arithmétique, toute proposition arithmétique devenant à la fois vraie et fausse.

axiomes, et toute chaîne qui peut être dérivée de ces axiomes en appliquant les règles du système de manière récursive sont appelées les *théorèmes* du système.

Une famille de tels systèmes existe telle qu'il est possible d'associer à chacune des chaînes générées une proposition arithmétique des nombres naturels (les chaînes du système sont alors appelées des « formules »). C'est l'interprétation de la chaîne. Si le système est bien construit, toutes les chaînes générées formellement par le système axiomatique s'interprètent en des propositions qui sont vraies (c'est ce que l'on appelle *la cohérence* du système). La question sur laquelle s'est penché Kurt Gödel était de savoir s'il existait un tel système qui permette de générer les chaînes correspondant à *toutes* les propositions vraies de l'arithmétique, en donc par le biais de l'interprétation, de générer *toutes* les propositions vraies de l'arithmétique. Gödel a montré qu'un tel système n'existait pas. Comment ?

Le trait de génie de Gödel a été de rajouter un second niveau d'interprétation, en projetant les formules du système axiomatique dans l'ensemble des nombres naturels. Il devenait alors possible de parler du « nombre x associé à la formule X », et de construire une autre formule Y, contenant x (le représentant de X), et donc parlant indirectement de X. En se basant sur son mécanisme de projection, Gödel arrive à construire un prédicat $NonDémontrable$, tel que la formule $NonDémontrable(\alpha)$ s'interprète comme « La formule associée au nombre α n'est pas démontrable dans le système axiomatique ». En utilisant ce prédicat, Gödel construit un nombre ω tel que le nombre associé à la formule $G \equiv NonDémontrable(\omega)$ soit justement ... ω ! En parlant de la formule associée à ω, G parle d'elle même. C'est justement cette formule G qui permet de démontrer le théorème d'incomplétude de Gödel : supposer G démontrable (c'est-à-dire dérivable formellement à l'intérieur du système axiomatique) implique que la proposition associée à G soit vraie, hors cette proposition dit justement que G n'est pas démontrable. En supposant l'arithmétique cohérente, l'on déduit que G n'est pas démontrable, donc que la proposition arithmétique associée est vraie, donc qu'il existe des formules associées à des propositions vraies qui ne sont pas démontrables.

Pour notre propos, il est très important de noter que G contient un nombre qui la représente, mais ne se contient pas elle même ! Le système axiomatique manipule des nombres, et certains nombres renvoient aux éléments structurels du système axiomatique, de la même manière que le français manipule des mots, et que certains mots du français renvoient aux éléments structurels de la langue française. La figure A.1 illustre ce processus de projection.

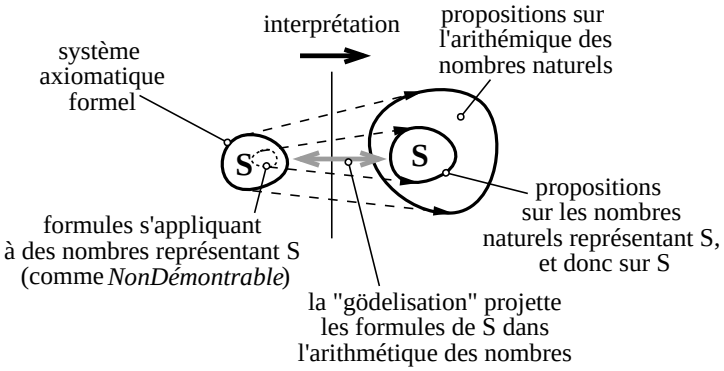


FIG. A.1 : *Projection d'un système axiomatique dans l'espace de son discours*

Annexe B

COSMOPEN : un outil de rétro-conception pour les logiciels multi-niveaux

COMME nous l'avons souligné dans notre conclusion, les composants logiciels du marché ne possèdent pas aujourd'hui les méta-interfaces nécessaires à l'implémentation orthogonale de la tolérance aux fautes, telle que nous la proposons dans ce mémoire. Lorsqu'une modification d'un composant s'avère inévitable pour des raisons de sûreté de fonctionnement, elle passe aujourd'hui par une collaboration étroite entre intégrateur et fournisseur, comme dans le cas de la NASA avec l'OS temps réel VXWORKS [Reeves 1997], ou de la SNCF avec le bus à objets ILOG BROKER [RIS 2002]. Pour la plupart des projets, cependant, ce type de collaborations reste inenvisageable en pratique. Pour pallier cet obstacle, nous avons, dans le cadre de nos travaux de thèse, développé une suite d'outils de rétro-conception, baptisée COMSOPEN[1], dédiée à l'extraction de méta-modèles structurels et comportementaux dans le contexte d'architectures complexes multi-niveaux.

COSMOPEN permet d'analyser la structure et le comportement d'un ensemble de composants utilisés dans une architecture en couches, puis d'en extraire des modèles de haut niveau par des opérations de filtrage dédiées. COSMOPEN facilite ainsi la mise en correspondance des activités de bas niveau d'une plate-forme avec les *comportements émergeants* des couches supérieures du système. COSMOPEN nous a notamment permis de construire les modèles présentés dans le chapitre 5, notamment ceux des figures 5.18 et 5.19, pages 103 et 105. Dans cette annexe, en complément aux modèles déjà présentés dans le reste de ce mémoire, nous rappelons brièvement l'organisation générale de COSMOPEN, avant de nous

[1] *Comprehensive Open Source MOdeling & Patternizing ENvironment*, COSMOPEN est disponible en source libre sous licence *GNU General Public Licence*.

étendre plus en détail sur les fonctionnalités de son composant d'analyse et d'abstraction, appelé OPSBROWSER (*OPen Source Browser*).

B.1 Architecture de l'outil

COSMOPEN suit les principes architecturaux proposés par [Chen et al. 1995] pour la manipulation de codes source de grande taille (figure B.1) :

1. des données d'analyse brutes sont extraites à partir de l'observation (structurelle ou dynamique) d'un programme ;
2. ces informations brutes sont ensuite traduites dans un format de référence basé sur le langage XML, pour être facilement manipulables de manière automatisée ;
3. un outil d'abstraction interactif (abstracteur) est utilisé pour construire des modèles structurels ou comportementaux abstraits à partir des données d'analyse formatées en XML ;
4. l'information obtenue est mise sous forme graphique en utilisant un outil de visualisation externe (dans notre cas, le programme `dot`[2] d'AT&T)

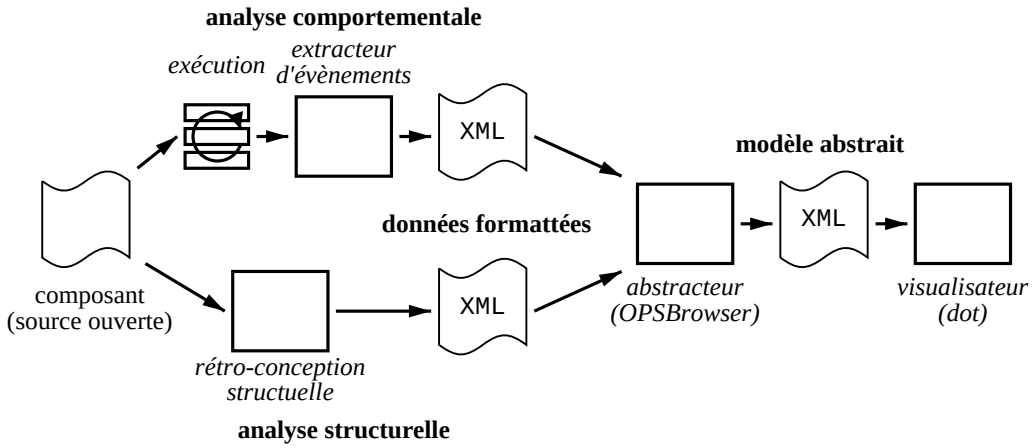


FIG. B.1 : *Architecture générale de l'outil* COSMOPEN

Nous avons expliqué dans la section 5.3 page 95 les principes mis en œuvre pour extraire les données d'analyse brutes (aussi bien structurelles que comportementales) à partir de l'observation d'un programme, et pour transformer ces données en un format XML. Dans la suite de cette annexe, nous nous concentrons sur l'outil d'abstraction fourni par la suite COSMOPEN : le manipulateur de graphes OPSBROWSER.

[2]http ://www.research.att.com/sw/tools/graphviz/

B.2 OPSBROWSER, l'abstracteur de COSMOPEN

B.2.1 Aperçu

OPSBROWSER est le composant central de COSMOPEN. Grâce à lui, il est possible de maîtriser la profusion des informations obtenues par l'observation directe d'un système, en construisant des modèles « abstraits » adaptés aux possibilités humaines de synthèse et d'analyse. Le besoin de tels outils d'abstraction a été reconnu depuis longtemps par les chercheurs en génie logiciel [Chen et al. 1995], et OPSBROWSER n'est, à cet égard, pas nouveau. L'intérêt d'OPSBROWSER tient à la spécificité de certains de ses opérateurs, qui ciblent directement l'analyse des interactions entre niveaux d'un système complexe.

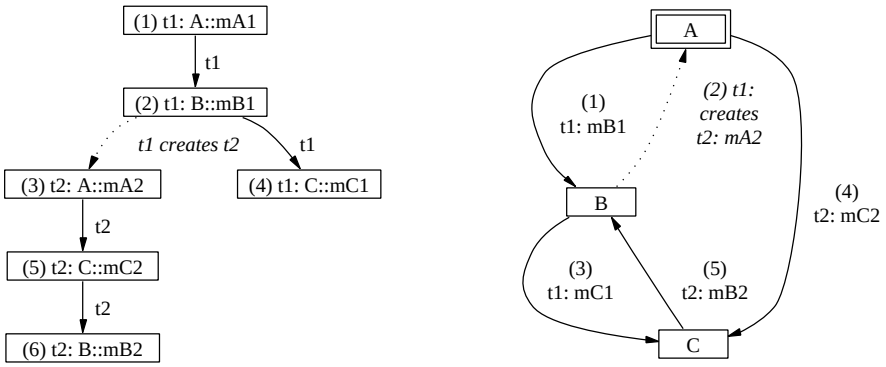
OPSBROWSER se présente comme un mini–interpréteur de calculs sur des graphes issus de l'observation (structurelle ou dynamique) de programmes orientés objet ou procéduraux. L'utilisation de « variables de graphe » lui permet de combiner de manière élaborée plusieurs opérateurs élémentaires en des filtres arbitrairement complexes. Par exemple, le code suivant charge dans la variable `A` le graphe contenu dans le fichier `jtc.xml`, dans la variable `B` le graphe du fichier `orbacus.xml`, puis assigne le contenu de `A` à la variable `C`, avant d'ajouter le contenu de `B` à `C` (`C` ← `A`; `C` ← `C` ∪ `B`). Le graphe résultant est ensuite sauvegardé dans un fichier `orbacus-jtc.xml`.

```
load jtc.xml     A
load orbacus.xml B
assign A C
add    B C
save C orbacus-jtc.xml
```

B.2.2 Graphes structurels et graphes comportementaux

Les graphes manipulés par OPSBROWSER sont de deux sortes : structurels et comportementaux. Un graphe structurel, comme celui de la figure 5.18 page 103, reprend les informations d'un diagramme de classe UML (*Unified Modeling Language* [OMG 1999]) : les nœuds représentent des classes ou des types du programme analysé, les arêtes des liens d'héritage ou des associations. Un graphe comportemental capture quant à lui un ensemble d'invocations, obtenues à partir d'une observation *in vivo* de l'exécution d'un programme. Il peut être représenté de deux manières : sous forme d'arbre d'appels (représentation étendue) ou sous forme de diagramme d'interaction (représentation condensée). La figure B.2 page suivante donne un exemple d'un même graphe comportemental représenté selon ces deux façons.

L'arbre d'appels traduit la suite des invocations et leur emboîtement telles qu'elles ont été observées à l'exécution du programme. Ainsi sur la figure B.2-a, l'on observe que le brin $t1$ a invoqué la méthode `mB1` de la classe `B` depuis la méthode `mA1` de la classe `A`.



(a) arbre d'appels (b) diagramme d'interaction

FIG. B.2 : *Deux représentations d'un même graphe comportemental*

Dans `B::mB1`, le brin $t1$ a ensuite créé le brin $t2$ dans la méthode `mA2` de la classe `A`, puis a invoqué la méthode `mC1` de la classe `C`. Les chiffres entre parenthèses sont des « estampilles temporelles » qui indiquent l'ordre global d'observation de chacune des invocations (une sorte de temps logique de l'observation). L'on voit ainsi que l'invocation à `C::mC2` par $t2$ se produit après l'appel à `C::mC1` par $t1$. Ces estampilles temporelles permettent de rendre compte de l'entrelacement des activités parallèles des différents brins.

La représentation sous forme d'arbre d'appels est facile à lire, mais peu compacte, et ne traduit pas la structure orientée objet du programme observé. C'est pourquoi nous avons implémenté dans OPSBROWSER une deuxième représentation, inspirée des diagrammes d'interaction d'UML. Nos diagrammes d'interaction ne suivent cependant pas scrupuleusement la norme UML, pour des raisons pratiques de compacité, au sens où nous ne représentons pas les objets interagissant, mais les classes correspondantes (figure B.2-b)[3]. Il est important de noter que cette seconde représentation, plus condensée, d'un graphe comportemental n'est qu'un mode de présentation, les structures manipulées en interne par OPSBROWSER restant celle d'un ensemble d'arbres d'appels. En particulier, tous les opérateurs d'OPSBROWSER travaillent directement sur l'arbre d'appels et non pas sur le diagramme d'interaction. Cette précision est importante pour comprendre l'effet d'opérations de manipulation, puisque les *arêtes* d'un diagramme d'interaction correspondent en fait aux *nœuds* d'un arbre d'appels.

B.2.3 Les opérateurs proposés par OPSBROWSER

Les différents opérateurs de graphe, accompagnés de la version courte de leur description telle qu'elle est fournie par l'aide en ligne d'OPSBROWSER, sont indiqués sur la table B.1 page suivante. Ces opérateurs recouvrent des opérations classiques d'algèbre booléenne, comme

[3] OPSBROWSER gère cependant les informations relatives aux identités des objets, et il ne serait pas difficile d'étendre le moteur actuel pour produire des diagrammes d'interaction représentant explicitement chaque instance de classe.

manipulation des variables

`clear`	removes all nodes from a graph variable.
`delete`	deletes a graph variable.
`assign`	assigns the graph of a given variable to a second.
`print`	prints the nodes of a graph possibly using a pattern.

opérateurs booléens

`add`	adds a graph to another.
`exclude`	excludes the nodes of a given graph from a second.
`abstract`	abstracts away the nodes present in one graph from another graph.
`envlop`	computes the edge envelop of a graph.
`remAlone`	removes the standalone nodes of a graph.

opérateurs basés sur les correspondances de noms

`put`	puts nodes from one graph to another w.r.t. a pattern.
`remove`	removes the nodes of a graph whose names match a pattern.
`absPatern`	abstracts away the nodes that match a pattern.
`absSelf`	abstracts away internal calls for classes matching a pattern.

opérateurs de traversées récursives

`backward`	computes the backward closure of a graph.
`forward`	computes the forward closure of a graph.
`spread`	union of the forward and backward closures of a graph.
`backN`	computes the backward set of a graph within a given depth.
`forwN`	computes the forward set of a graph within a given depth.
`spreadN`	computes the spread set of a graph within a given depth.

opérateurs d'analyse structurelle

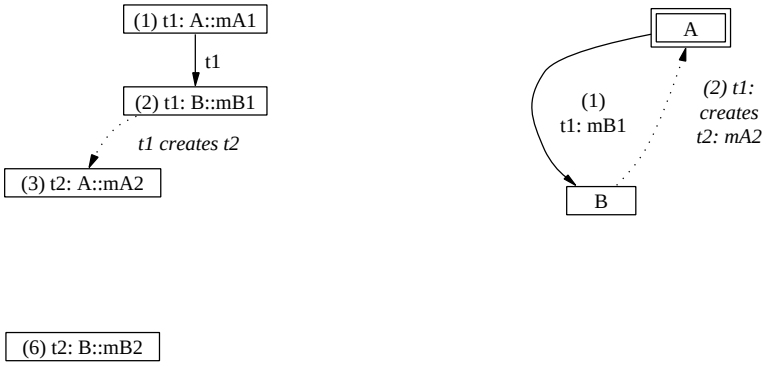
`remHin`	removes the link of kind « superclass » from a graph.
`remUse`	removes the link of kind « usage » from a graph.

analyse comportementale (utilisation du temps logique)

`remAfter`	removes the event of a call graph that are subsequent to a given time.
`remBefore`	removes the event of a call graph that are prior to a given time.
`slice`	puts nodes from one graph to another w.r.t. a time interval.
`leapOver`	leaps over calls based on temporal ordering

TAB. B.1 : *Les principales opérations de manipulation fournies par* OPSBROWSER

`add` (union) ou `exclude` (complément)[4], que nous avons déjà rencontrées dans l'exemple donné page 129. À ces opérateurs qui travaillent directement sur deux variables de graphes, correspondent des opérateurs basés sur la reconnaissance de motifs dans les noms de nœuds (*pattern matching*). L'utilisation des noms de nœuds se révèle en effet très utile pour sélectionner la partie d'un modèle ayant trait à une classe, un brin ou un espace de nom particulier. Ainsi, si sur le graphe de la figure B.2 page 130, nous appliquons l'opération « `remove C::*` » (retrait de toutes les invocations faites sur la classe `C`), nous obtenons le nouveau graphe représenté sur la figure B.3 (de nouveau dans les deux représentations). L'on notera comment l'opération « `remove C::*` » s'applique aux nœuds de l'arbre d'appels, et comment elle se traduit sur le diagramme d'interaction.



(a) arbre d'appels (b) diagramme d'interaction

Figure B.3 : *Graphe de la figure B.2 après l'opération "`remove C::*`"*

Nous ne pouvons, faute de place, détailler l'ensemble des opérateurs restants, dont la signification, pour la plupart, reste facilement compréhensible, au moins à qui est habitué à la manipulation de graphes (opérateurs de fermeture, opérateurs de retrait d'arêtes). Afin d'illustrer l'intérêt d'OPSBROWSER pour l'analyse de programmes multi-niveaux, nous nous concentrons dans la section qui suit sur deux types d'opérations qui lui sont spécifiques, le *saut* (opérateur `leapOver` dans la table B.1), et *l'abstraction* (opérateurs `abstract`, `absPatern`, et `absSelf`), en reprenant l'exemple des figures 5.16 et 5.17 de la page 101.

B.3 OPSBROWSER en action

B.3.1 Le programme étudié

Dans la section 5.3.2 page 99 du chapitre 5, nous avons étudié le fonctionnement de la bibliothèque `libpthread.so` de multitraitement de LINUX, et indiqué que nous pouvions,

[4]On notera à ce propos l'absence d'une opération d'intersection, celle-ci ne s'étant pas révélée utile lors des étapes de notre travail. Ceci souligne l'étroite relation entre les opérateurs d'OPSBROWSER et les besoins ressentis au cours de notre travail de rétro-conception.

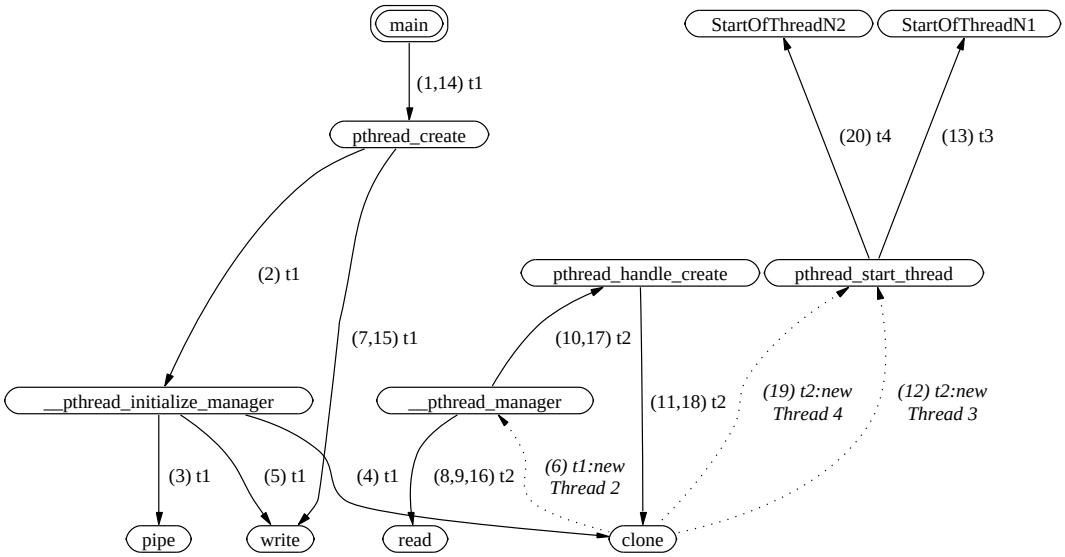


FIG. B.4 : *Création de deux brins par la bibliothèque de multitraitement de* LINUX 2.4

avec COSMOPEN, ne retenir que la sémantique de haut niveau contenue dans les traces d'observation comportementale d'un programme. Nous revenons ici sur cet exemple pour illustrer pas-à-pas comment s'opère cette extraction sémantique. La figure B.4 (qui reprend la figure 5.16 page 101) résulte de l'observation du petit programme qui suit, lorsque celui-ci s'exécute sur le noyau LINUX 2.4 (seul le déroulement des appels à `pthread_create` est représenté). `pipe`, `write`, `read`, `clone` correspondent à des appels système LINUX, et sont invoqués en interne pas la bibliothèque de multitraitement suite aux appels à `pthread_create` par le programme.

```
int main () {
   pthread_t threadN1, threadN2 ;
   pthread_create(&threadN1, NULL, StartOfThreadN1, NULL) ;
   pthread_create(&threadN2, NULL, StartOfThreadN2, NULL) ;
   pthread_join(threadN1, NULL) ;
   pthread_join(threadN2, NULL) ;
}
```

B.3.2 Causalités cachées et « sauts » temporels

Notre objectif est de transformer le graphe de la figure B.4 pour ne plus faire ressortir que la logique applicative du petit programme précédent, c'est-à-dire la création de deux brins par le brin principal. Mais nous nous heurtons à une difficulté : un graphe comportemental ne fait ressortir que les flux de contrôle d'un programme, et ne capture pas les

interactions entre brins réalisées par échange de données au travers de structures partagées (flux de données). Sur le graphe B.4, les demandes de création par le brin principal $t1$ de deux nouveaux brins (appels (1) et (14)) se traduisent par l'écriture d'une requête de création sur un tube de communication (*pipe*) entre le brin $t1$ et le gestionnaire de brins (*thread manager*) $t2$, qui est interne à `libpthread.so` (appels (7) et (15) à `write`). Les informations écrites sur le pipe sont lues lors des appels correspondants (9) et (16) à `read`. Le lien de causalité qui existe entre un appel à `write` et un appel à `read` n'est, sur la figure, pas visible. Comment alors relier la création du brin $t3$ par $t2$ (appels (11) et (12)) à la première invocation de `pthread_create` par $t1$ (appel (1)) ? et celle de $t4$ (appels (18) et (19)) à la seconde invocation de `pthread_create` (appel (14)) ? Nous avons choisi pour cela de mettre à profit les informations temporelles capturées par un graphe comportemental. Si nous considérons en effet l'appel (11) $t2$: `clone`, c'est le premier appel à `clone` (en dehors de la création du brin gestionnaire $t2$) à se produire après l'appel (1) $t1$: `pthread_create`. Ces deux appels sont donc nécessairement reliés. De la même façon, si nous considérons qu'à chaque invocation à `pthread_create` ne peut correspondre qu'une seule invocation à `clone` (en dehors, toujours, de la création du brin gestionnaire $t2$), nous relions immédiatement l'appel (18) $t2$: `clone` à l'appel (14) $t1$: `pthread_create`. Le seul cas où ce type d'inférence basée sur les estampilles temporelles ne fonctionne pas est lorsque les séquences $<\texttt{pthread_create}^i \rightarrow \texttt{clone}^i>$ s'entrelacent (par exemple $[\texttt{pthread_create}^1; \texttt{pthread_create}^2; \texttt{clone}^1; \texttt{clone}^2]$). Il est cependant possible de détecter ce type d'entrelacements (très rares en pratique) de façon automatique, et de s'interdire ainsi toute conclusion erronée.

Pour réaliser cette inférence basée sur les estampilles temporelles de façon automatique, nous avons développé dans OPSBROWSER un opérateur de « saut » (*leap* en anglais) appelé `leapOver`. `leapOver` utilisent trois opérandes, de la forme

```
leapOver <motifDeDépart> <motifDeSaut> <graphe>.
```

`<graphe>` est la variable de graphe à laquelle s'applique l'opérateur.

`<motifDeDépart>` est un motif sur les noms de nœuds pour sélectionner les premiers éléments des paires $<\texttt{appelCause}^i \rightarrow \texttt{appelEffet}^i>$. Dans notre exemple, nous voulons sélectionner les appels à `pthread_create`. Notre motif de départ sera donc « `::pthread_create'*` » [5].

`<motifDeSaut>` est un motif sur les noms de nœuds pour sélectionner les seconds éléments des paires $<\texttt{appelCause}^i \rightarrow \texttt{appelEffet}^i>$. Dans notre exemple, ce second motif sera « `::clone'*` ».

Munis de ces trois opérandes, `leapOver` fonctionne alors de la manière suivante : pour chacune des invocations $\texttt{appelEffet}^i$ sélectionnées par le motif de saut (pour nous chacune des invocations à `clone`), `leapOver` recherche la plus récente invocation $\texttt{appelCause}^i$

[5] Les « `::` » en début de motif indiquent que `pthread_create` est un symbole global, qui ne fait notamment partie d'aucune classe. Le « `'*` » est une syntaxe particulière à OPSBROWSER qui permet de sélectionner toutes les invocations à `pthread_create`, quel que soit le brin qui les effectue. Il serait aussi possible, par une autre syntaxe, de ne sélectionner que celles faites par un brin précis.

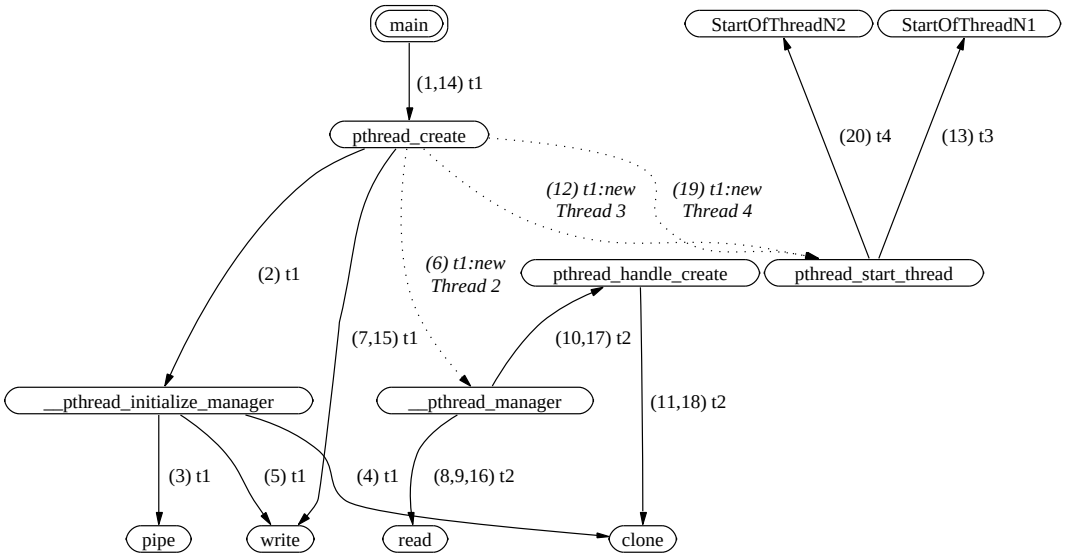


FIG. B.5 : *Graphe B.4 après une opération de saut « `leapOver` »*

correspondant au motif d'arrivée qui se soit produite avant `appelEffet`^{i} (pour nous la plus récente invocation à `pthread_create` précédent le `clone` considéré). Ainsi à (11) $t2$: `clone`, `leapOver` fera correspondre (1) $t1$: `pthread_create`, et à (18) $t2$: `clone`, (14) $t1$: `pthread_create`.

Ensuite, pour chaque paire <`appelCause`^{i} $\rightarrow$ `appelEffet`^{i}> ainsi formée, `leapOver` opère un « *saut* » depuis `appelCause`^{i} « par-dessus » `appelEffet`^{i}, en déplaçant toutes les arêtes qui sortent de `appelEffet`^{i} vers `appelCause`^{i} (pour nous, donc, les arêtes sortant de `clone`, c'est-à-dire les arêtes de création de brin). Si par exemple une arête existe dans le graphe original entre `appelEffet`^{i} et l'invocation `unAutreAppel`, alors cette arête est supprimée et remplacée par une arête équivalente de `appelCause`^{i} vers `unAutreAppel`.

L'exécution de

```
leapOver ::pthread_create'* ::clone'* GrapheGlobal
```

sur le graphe B.4 page 133 produit le nouveau graphe représenté sur la figure B.5. On notera sur ce nouveau graphe comment les arêtes de création de brin (en pointillés) ne partent plus de `clone` mais de `pthread_create`.

B.3.3 S'abstraire des détails tout en gardant leur trace

Il nous reste maintenant à éliminer du graphe toutes les invocations des brins $t1$ et $t2$ qui correspondent au fonctionnement interne de la bibliothèque `libpthread.so`. Pour cela nous sélectionnons ces invocations en utilisant la suite d'opérateurs suivants :

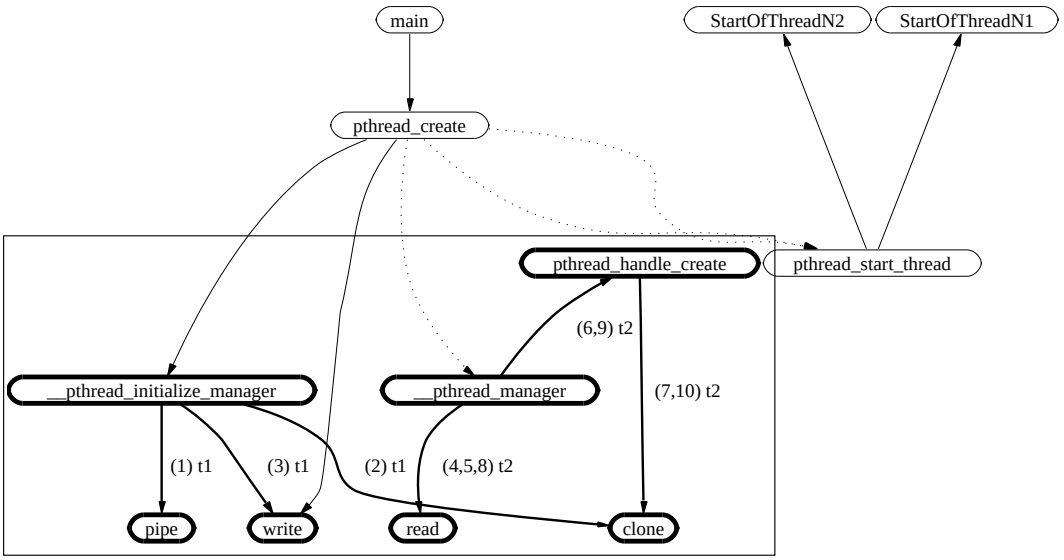


FIG. B.6 : *Sélection des actions internes à la bibliothèque de multitraitement*

```
put    ::pthread_create'* GrapheGlobal P
forwN  1 P GrapheGlobal
remove ::pthread_start_thread'* P
remove ::pthread_create'*       P
forward P GrapheGlobal
```

Ce « mini–programme » utilise les opérateurs d'addition et de retrait basés sur les motifs (`put` et `remove`), et les opérateurs de traversée récursive avant (`forwN` et `forward`) pour sélectionner la fermeture transitive issue de `pthread_create`, à l'exception de la branche que démarre `pthread_start_thread`. Le résultat (le graphe `P`) est représenté sur la figure B.6.

En « retirant » `P` du graphe principal (c'est-à-dire en ne gardant que la partie en grisé de la figure B.6) nous obtenons le graphe de la figure B.7 page ci-contre. Ce nouveau graphe fait directement ressortir la création par le brin $t1$ de deux nouveaux brins $t3$ et $t4$. Cependant, nous pourrions souhaiter nous *abstraire* complètement de la bibliothèque de multitraitement en supprimant toutes les invocations qui lui correspondent (`pthread_create` et `pthread_start_thread` sur le graphe), tout en gardant trace des relations d'emboîtement entre les invocations restantes. C'est ce que permettent de faire les opérateurs d'*abstraction* d'OPSBROWSER, qui suppriment un ou plusieurs nœuds d'un graphe, et relient tous les voisins « amonts » des noeuds supprimés à leurs voisins « avals ». Ainsi, si un nœud `a` est relié par une arête à `b`, lui-même relié à `c`, alors l'abstraction du nœud `b` provoque la suppression de `b`, et des deux arêtes depuis `a` et vers `c`, et la création d'une arête entre `a` et `c`, en « souvenir » du nœud `b` disparu. Nous pouvons donc appliquer

```
absPatern ::pthread_* GrapheGlobal
```

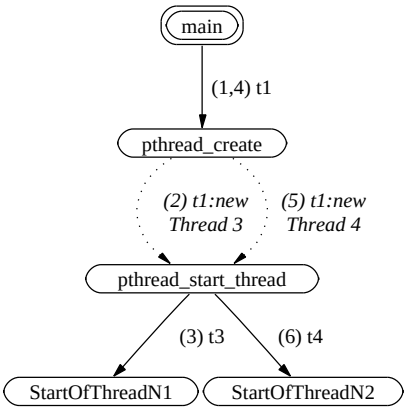


FIG. B.7 : *Le graphe résultant du retrait (`exclude`) du graphe B.6*

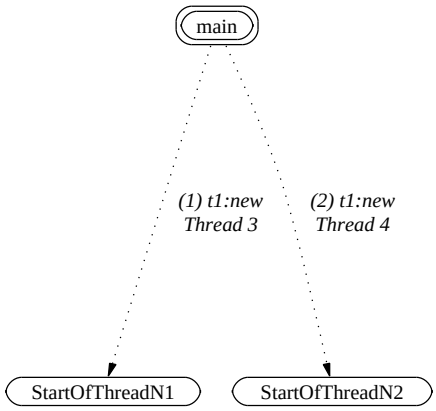


FIG. B.8 : *Après abstraction des méthodes POSIX `pthread_`... du graphe B.7*

au graphe B.7. Nous obtenons alors le graphe B.8, où nous reconnaissons le graphe 5.17 page 101 du chapitre 5.

B.3.4 En résumé

En partant du graphe relativement complexe de la figure B.4 page 133, nous avons obtenu en utilisant la suite d'opérations résumées sur la figure B.9 page suivante le graphe extrêmement simple de la figure B.8.

Ce filtre, constitué d'une suite d'opérateurs élémentaires d'OPSBROWSER, permet donc à partir d'une observation directe d'un programme, d'isoler l'action d'une couche particulière (ici la couche de multitraitement), pour ensuite s'en abstraire et pouvoir travailler, à un plus haut niveau d'abstraction, sur la logique applicative du programme étudié. Il est essentiel de souligner ici que le filtre de la figure B.9 n'est en rien spécifique au programme de la page 133 que nous avons étudié. Ce filtre s'applique à tout logiciel qui utilise la bibliothèque de multitraitement `libpthread.so` de LINUX 2.4, et ne dépend donc que de cette bibliothèque. Dans cette généricité réside la grande force, selon nous, de notre « abstracteur » OPSBROWSER. Sans lui la construction du modèle d'ORBACUS que nous avons présenté dans la section 5.4.1 page 102 de notre dernier chapitre aurait été impossible.

B.4 Conclusion

Nous avons présenté, dans cette annexe, comment notre outil COSMOPEN, et plus particulièrement son composant d'abstraction, OPSBROWSER, nous avait permis de construire le méta-modèle du bus à objets ORBACUS que nous avons présenté dans notre chapitre 5. Ce méta-modèle est lui-même à la base du prototype de méta-interface multi-niveaux pour la

```
leapOver  ::pthread_create'* ::clone'* GrapheGlobal
put       ::pthread_create'* GrapheGlobal P
forwN     1 P GrapheGlobal
remove    ::pthread_start_thread'* P
remove    ::pthread_create'*       P
forward   P GrapheGlobal
exclude   P GrapheGlobal
absPatern ::pthread_* GrapheGlobal
```

FIG. B.9 : *Le filtre final d'abstraction permettant d'obtenir le graphe B.8*

tolérance aux fautes que nous avons développé pour valider l'approche défendue dans cette thèse. Nous espérons avoir ainsi montré que COSMOPEN se révélait un moyen *léger*, *peu intrusif*, *portable*, et à l'usage *extrêmement puissant* pour analyser un système complexe architecturé en couches. Par ses opérateurs spécifiques d'abstraction, le composant d'abstraction de COSMOPEN, OPSBROWSER, autorise l'étude séparée de différentes couches d'un système, tout en éclairant leurs interactions, et en soulignant la place de l'action de chacune dans la réalisation du *service émergent* fourni pas le système.

COSMOPEN a cependant ses limites, dont nous sommes pleinement conscients, et nous ne pensons pas qu'il permette à lui seul de construire les méta-modèles génériques pour composants dont nous nous sommes faits les avocats dans notre chapitre de conclusion page 117. La première limite de COSMOPEN, qui fonde aussi la plupart de ses qualités, tient au caractère ponctuel des observations comportementales qu'il fournit : COSMOPEN, par la méthode de capture d'évènements que nous avons choisie (section 5.3.2 page 97), ne fournit la trace que d'*une seule* exécution parmi d'autres. Le modèle obtenu ne recouvre donc jamais toutes les exécutions possibles, et il convient de garder fermement à l'esprit cette limitation pour ne pas en tirer des conclusions erronées. Une autre faiblesse de COSMOPEN tient à son caractère interactif. Ce n'est pas un outil « automatique » : il aide le concepteur à comprendre le système analysé, mais ne se substitue pas à l'intelligence humaine.

Ces limitations (incomplétude et interactivité) doivent toutefois être replacées selon nous dans la perspective des limites inhérentes aux outils d'extraction de modèles, qui sont directement liées aux résultats fondamentaux d'impossibilité de l'informatique (dont le théorème d'incomplétude de Gödel dont nous avons dit quelques mots dans l'annexe A page 123 fonde la base) :

Un modèle d'un composant obtenu de manière *automatisée* ne peut permettre de répondre de manière *elle aussi automatisée* à des questions sur le composant qui sont *indécidables* pour le composant en question (sinon ces questions seraient décidables).

En n'étudiant de manière interactive qu'une seule exécution d'un programme, CosmOpen ne peut être comparé aux approches par interprétation abstraite ou par « slicing » [Cousot & Cousot 1977, Dwyer et al. 2000], qui calculent une enveloppe de *toutes* les exécutions possibles d'un système (et donc sur–approximent en quelque sorte son comportement). Mais de par sa simplicité, nous pensons que CosmOpen peut s'avérer un complément particulièrement utile à ce type d'analyse formelle, en permettant rapidement une compréhension intuitive d'un système, pour pouvoir orienter et piloter ces analyses successives.

CosmOpen, bien que simple dans son principe, libère le développeur de méta-interfaces des tâches d'analyse de bas niveaux qui sont automatisables pour lui permettre de se concentrer sur les problèmes d'analyse sémantique les plus difficiles.

Bibliographie

[AEEC] AEEC. *ARINC-429 Digital Information Transfer System Parts 1, 2, 3*.

[Agha et al. 1992] Agha, G., Frolund, S., Panwar, R., & Sturman, D. (1992). A Linguistic Framework for Dynamic Composition of Dependability Protocols. In Transactions, I., editor, *the IFIP Conference on Dependable Computing for Critical Applications (DCCA-3)*, pages 197–207, Palermo (Sicily), Italy. Elsevier.

[Amir et al. 1992] Amir, Y., Dolev, D., Kramer, S., & Malki, D. (1992). Transis : A Communication Subsystem for High Availability. In *FTCS-22 : 22nd International Symposium on Fault Tolerant Computing*, pages 76–84, Boston, Massachusetts. IEEE Computer Society Press.

[Arlat et al. 2002] Arlat, J., Fabre, J.-C., Rodriguez, M., & Salles, F. (2002). Dependability of COTS microkernel-based Systems. *IEEE Transactions on Computers*, 51(2) :138–163.

[ATILF 2003] ATILF (2003). Le trèsor de la langue française informatisé (TLFi). http ://zeus.inalf.cnrs.fr/. Conception et réalisation informatiques : Jacques Dendien.

[Atkinson et al. 1983] Atkinson, M., Bailey, P., Chisholm, K., Cockshott, W., & Morrison, R. (1983). PS-algol : A Language for Persistent Programming. In *Proc. 10th Australian National Computer Conference, Melbourne, Australia*, pages 70–79.

[Avizienis & Kelly 1984] Avizienis, A. & Kelly, J. P. J. (1984). Fault Tolerance by Design Diversity : Concepts and Experiments. *IEEE Computer*, 17(8) :67–80.

[Baldoni et al. 1998a] Baldoni, R., Helary, J., & Raynal, M. (1998a). Consistent Records in Asynchronous Computations. *Acta Informatica*, 35(6) :441–455.

[Baldoni et al. 1997] Baldoni, R., Hélary, J.-M., Mostefaoui, A., & Raynal, M. (1997). A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability. In *International Symposium on Faut-Tolerant Computing FTCS-27*, pages 68–77, Seattle, Washington.

[Baldoni et al. 1998b] Baldoni, R., Hélary, J.-M., & Raynal, M. (1998b). Rollback-Dependency Trackability : A Minimal Characterization and its Protocol. Publication Interne PI-1173, IRISA.

[Balkrishna Ramkumar 1997] Balkrishna Ramkumar, V. S. (1997). Portable Checkpointing for Heterogeneous Architectures. In *27th International Symposium on Fault-Tolerant Computing (FTCS'97) - Digest of Papers*, pages 58–67, Seattle, WA (USA).

[Basile et al. 2003] Basile, C., Kalbarczyk, Z., & Iyer, R. (2003). A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 149–158, San Francisco, CA. IEEE Computer Society.

[Beck 1999] Beck, K. (1999). *Extreme Programming Explained : Embrace Change*. Addison-Wesley.

[Bidinger & Stefani 2003] Bidinger, P. & Stefani, J.-B. (2003). The Kell calculus : operational semantics and type system. In *Proceedings 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 03)*, Paris, France.

[Birman 1985] Birman, K. (1985). Replication and Fault-Tolerance in the ISIS System. In *10th ACM Symposium on Operating Systems Principles, Operating System Review 19(5)*, pages 79–86, Orcas Island. Washington, USA. ACM, New York.

[Blair et al. 1998] Blair, G. S., Coulson, G., Robin, P., & Papathomas, M. (1998). An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England. Springer-Verlag.

[Breton 1997] Breton, D. L. (1997). *Du silence*. Collection traversées. Métailié, Diffusion Seuil, Paris.

[Briot et al. 1998] Briot, J.-P., Guerraoui, R., & Lohr, K.-P. (1998). Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3) :291–329.

[Bruneton et al. 2002] Bruneton, E., Coupaye, T., & Stefani, J.-B. (2002). Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain). http ://fractal.objectweb.org/.

[Chandra & Toueg 1996] Chandra, T. D. & Toueg, S. (1996). Unreliable Failure Detectors for Reliable Distributed Systems. *Journal for the Association for Computing Machinery (JACM)*, 43(2) :225–267.

[Chandy & Lamport 1985] Chandy, K. M. & Lamport, L. (1985). Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1 (February)) :63–75.

[Chen et al. 1995] Chen, Y.-F., Fowler, G., Koutsofios, E., & Wallach, R. (1995). Ciao : a graphical navigator for software and document repositories. In *International Conference on Software Maintenance*, pages 66–75, Opio (Nice), France.

[Chiba 1995] Chiba, S. (1995). A Metaobject Protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA.

[Cointe 1999] Cointe, P., editor (1999). *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99, Saint-Malo, France, July 19-21, 1999, Proceedings*, volume 1616 of *Lecture Notes in Computer Science*. Springer.

[Colin & Puaut 2000] Colin, A. & Puaut, I. (2000). Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, 18(2) :249–274. Special issue on worst-case execution time analysis.

[Costa et al. 1998] Costa, F. M., Blair, G. S., & Coulson, G. (1998). Experiments with Reflective Middleware. In *Proceedings of the ECOOP Workshop on Reflective Object-Oriented Programming and Systems (ROOPS '98)*, volume 1543 of *Lecture Notes in Computer Science (LNCS)*, pages 390–391, Brussels, Belgium. Springer Verlag.

[Courtès 2003] Courtès, L. (2003). Systèmes tolérant les fautes à base de support d'exécution réflexifs Capture en ligne de l'état d'applications. Stage de DEA - Université de Franche-Comté, LAAS-CNRS, Toulouse.

[Cousot & Cousot 1977] Cousot, P. & Cousot, R. (1977). Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. ACM Press, New York, NY.

[Dahl & Nygaard 1966] Dahl, O.-J. & Nygaard, K. (1966). SIMULA–an ALGOL-Based Simulation Language. *Comunications of the ACM (CACM)*, 9(9) :671–678.

[Dieter & Jr. 1999] Dieter, W. R. & Jr., J. E. L. (1999). A User-level Checkpointing Library for POSIX Threads Programs. In *29th International Symposium on Fault-Tolerant Computing*, pages 224–227, Madison, Wisconsin USA.

[Dieter & Jr. 2001] Dieter, W. R. & Jr., J. E. L. (2001). User-level Checkpointing for LinuxThreads Programs. In *2001 USENIX Technical Conference*, Boston, Massachusetts, USA.

[Dumant et al. 1999] Dumant, B., Horn, F., Tran, F. D., & Stefani, J.-B. (1999). Jonathan : an open distributed processing environment in Java. *Distributed Systems Engineering*, 6(1) :3–12.

[Dwyer et al. 2000] Dwyer, M. B., Hatcliff, J., & Zheng, H. (2000). Slicing Software for Model Construction. *Journal of Higher-order and Symbolic Computation*, 13(4) :315–353. Special Issue on the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1999).

[Elnozahy et al. 2002] Elnozahy, E., Alvisi, L., Johnson, D. B., & Wang, Y.-M. (2002). A Survey of Rollback-Recovery Protocols in Message Passing Systems. *ACM Computing Surveys*, 34(3 (September)) :375–408.

[Engler et al. 1995] Engler, D. R., Kaashoek, M. F., & O'Toole, J. (1995). Exokernel : An Operating System Architecture for Application-Level Resource Management. In *Fifteenth ACM Symposium on Operating System Principles (SOSP'95)*, pages 251–266, Cooper Mountain Resort, Colorado.

[ESA 1996] ESA (1996). Ariane 5, Flight 501 Failure, Report by the Inquiry Board. Technical Report, ESA, Paris. *Jacques-Louis Lions et al.*

[Espen Skoglund 2000] Espen Skoglund, Christian Ceelen, J. L. (2000). Transparent Orthogonal Checkpointing Through User-Level Pagers. In *9th International Workshop on*

Persistent Object Systems (POS9), number 2135 in Lecture Notes in Computer Science, pages 201–214, Lillehammer, Norway. System Architecture Group, University of Karlsruhe, Springer Verlag.

[Fassino et al. 2002] Fassino, J.-P., Stefani, J.-B., Lawall, J., & Muller, G. (2002). THINK : A Software Framework for Component-based Operating System Kernels. In *Usenix Annual Technical Conference*, Monterey (USA).

[Felber 1998] Felber, P. (1998). *A Service Approach to Object Groups in CORBA*. Thèse de Doctorat, Département d'informatique de l'École Polytechnique Fédérale de Lausanne, Lausanne (Suisse).

[Felber et al. 1996] Felber, P., Garbinato, B., & Guerraoui, R. (1996). The Design of a CORBA Group Communication Service. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS-15)*, pages 150–159, Niagara-on-the-Lake, Canada.

[Ferrari et al. 1997] Ferrari, A. J., Chapin, S. J., & Grimshaw, A. S. (1997). Process Introspection : A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification. Technical Report, University of Virginia (USA). http ://www.cs.virginia.edu/ ajf2j/introspect/.

[Fetzer & Cristian 1995] Fetzer, C. & Cristian, F. (1995). On the Possibility of Consensus in Asynchronous Systems. In *1995 Pacific Rim Int'l Symp. on Fault-Tolerant Systems*, pages 86–91, Newport Beach, CA.

[Fischer et al. 1985] Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985). Impossibility of Distributed Concensus with One Faulty Process. *Journal of the Association for Computing Machinery*, 32(2 (April)) :374–382.

[Galus 2003] Galus, C. (2003). Comment plantes et arbres luttent contre sécheresse et canicule. *Le Monde*.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.

[Goldstein 2002] Goldstein, H. (2002). Checking the Play in Plug-and-Play. *IEEE Spectrum*, (June) :50–55.

[Goscinny & Uderzo 1965] Goscinny, R. & Uderzo, A. (1965). *Astérix et Cléopâtre*, volume 6. Dargaud.

[Grace et al. 2003] Grace, P., Blair, G., & Samuel, S. (2003). ReMMoC : A Reflective Middleware to support Mobile Client Interoperability. In *International Symposium on Distributed Objects and Applications (DOA 2003)*, number 2519 in Lecture Notes in Computer Science (LNCS), Catania, Sicily (Italy). Springer Verlag.

[Guerraoui et al. 1997] Guerraoui, R., Garbinato, B., & Mazouni, K. R. (1997). GARF : A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 5(4) :32–39.

[Gödel 1931] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, (38) :173–198.

[Hyde 2003] Hyde, R. (2003). *The Art of Assembly Language*. No Starch Press.

[ISO 1995] ISO (1995). *Open Distributed Processing (ODP) Reference Model*. ISO (International Organization for Standardization). ISO/IEC Recommendations X.901-904, International Standard 10746-1-4.

[Jiménez-Peris et al. 2000] Jiménez-Peris, R., Mart'inez, M. P., & Arévalo, S. (2000). Deterministic Scheduling for Transactional Multithreaded Replicas. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 164–173, Nürmberg, Germany.

[Johnson & Zwaenepoel 1987] Johnson, D. & Zwaenepoel, W. (1987). Sender-Based Message Logging. In *Seventeenth International Symposium on Fault-Tolerant Computing (FTCS)*, pages 14–19.

[Joy et al. 2000] Joy, B., Steele, G., Gosling, J., & Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley Pub Co, 2nd edition.

[Karablieh & Bazzi 2002] Karablieh, F. & Bazzi, R. A. (2002). Heterogeneous Checkpointing for Multithreaded Applications. In *21st Symposium on Reliable Distributed Systems (SRDS'02)*, pages 140–149, Osaka, Japan.

[Kasbekar et al. 1999] Kasbekar, M., Narayanan, C., & Das, C. (1999). Selective Checkpointing and Rollbacks in Multithreaded Object-Oriented Environment. *IEEE Transactions on Reliability*, 48(4) :325–337.

[Kiczales et al. 1991] Kiczales, G., des Rivieres, J., & Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

[Kiczales & Lamping 1993] Kiczales, G. & Lamping, J. (1993). Operating Systems : Why Object-Oriented ? In Hutchinson, L.-F. C. & Norman, editors, *the Third International Workshop on Object-Orientation in Operating Systems*, pages 25–30, Asheville, North Carolina. IEEE Computer Society Press.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-Oriented Programming. In Springer, editor, *European Conference on Object-Oriented Programming (ECOOP'97)*, volume LNCS 1241, pages 220–242, Jyväskylä, Finland.

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

[Killijian et al. 1998] Killijian, M., Fabre, J., Ruiz-García, J., & Shiba, S. (1998). A metaobject protocol for fault-tolerant CORBA applications. In *17th IEEE Symposium on Reliable Distributed Systems (SRDS-17)*, pages 127–134, West Lafayette (USA).

[Killijian & Fabre 2000] Killijian, M.-O. & Fabre, J.-C. (2000). Implementing a Reflective Fault-Tolerance CORBA System. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 154–163, Nürnberg, Germany.

[Killijian et al. 1999] Killijian, M.-O., Ruiz-García, J.-C., & Fabre, J.-C. (1999). Using Compile-Time Reflection for Object State Capture. In [Cointe 1999], pages 150–152.

[Killijian et al. 2002] Killijian, M.-O., Ruiz-García, J.-C., & Fabre, J.-C. (2002). Portable serialization of CORBA objects : a reflective approach. In *18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002)*, pages 68–82, Seattle, Washington, USA. ACM Press.

[Kon et al. 2002] Kon, F., Costa, F., Blair, G., & Campbell, R. H. (2002). The case for reflective middleware. *Communications of the ACM*, 45(6 (June)) :33–38.

[Kon et al. 2000] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L. C., & Campbell, R. H. (2000). Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York. Springer-Verlag.

[Koo & Toueg 1987] Koo, R. & Toueg, S. (1987). Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1) :23–31.

[Kopetz & Grünsteidl 1994] Kopetz, H. & Grünsteidl, G. (1994). TTP - A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1) :14–23.

[Lamport et al. 1982] Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3) :382–401.

[Laprie 1996] Laprie, J.-C., editor (1996). *Le guide de la sûreté de fonctionnement*. Cepadues.

[Ledoux 1999] Ledoux, T. (1999). OpenCorba : A Reflective Open Broker. In [Cointe 1999], pages 197–214.

[Levine 1999] Levine, J. R. (1999). *Linkers and Loaders*. Morgan Kaufmann Publishers, 1st edition edition.

[Li et al. 1994] Li, K., Naughton, J. F., & Plank, J. S. (1994). Low-Latency, Concurrent Checkpointing for Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8) :874–879.

[Liskov 1987] Liskov, B. (1987). Keynote address - Data Abstraction and Hierarchy. In *Addendum to the proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'87)*, pages 17–34, Orlando, Florida, United States. ACM. also published in ACM SIGPLAN Notices Volume 23 , Issue 5 (May 1988).

[Maes 1987] Maes, P. (1987). Concepts and Experiments in Computational Reflection. volume 22 of *SIGPLAN Notices*, pages 147–155, Orlando, Florida. ACM.

[Maffeis 1995] Maffeis, S. (1995). Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 135–146, Monterey, CA. USENIX.

[Mercury 1992] Mercury (1992). American Airlines software development woes. *San Jose Mercury News*. cité dans le volume 13, issue 64 of the Risks Forum of ACM Committee on Computers and Public Policy, *http ://catless.ncl.ac.uk/Risks/13.67.html*.

[Meyer 1997] Meyer, B. (1997). *Object Oriented Software Construction*. Prentice Hall PTR, seconde edition.

[Mishra et al. 1993] Mishra, S., Peterson, L. L., , & Schlichting, R. D. (1993). Modularity in the Design and Implementation of Consul. In *Proceedings of the First IEEE Symposium on Autonomous Decentralized Systems (ISADS'93)*, pages 376–382, Kawasaki, Japan.

[MITRE 2003] MITRE (2003). Use of Free and Open-Source Software (FOSS) in the U.S. Department of Defense. Technical Report MP 02 W0000101 (Version 1.2.04), The MITRE Corporation.

[Nagel et al. 1989] Nagel, E., Newman, J. R., Gödel, K., & Girard, Y. (1989). *Le théorème de Gödel*. Editions du Seuil, Paris (F).

[Napper et al. 2003] Napper, J., Alvisi, L., & Vin, H. (2003). A Fault-Tolerant Java Virtual Machine. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434.

[Narasimhan et al. 1997] Narasimhan, P., Moser, L. E., & Melliar-Smith, P. M. (1997). The Interception Approach to reliable distributed CORBA Objects. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 245–248, Portland, OR, USA. USENIX.

[Netzer & Xu 1995] Netzer, R. H. B. & Xu, J. (1995). Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2) :165–169.

[OMG 1999] OMG (1999). OMG Unified Modeling Language V. 1.3. http ://www.omg.org/cgi-bin/doc ?ad/99-06-08.pdf.

[OMG 2002a] OMG (2002a). *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*. Needham, MA, U.S.A.

[OMG 2002b] OMG (2002b). *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*, chapter 23. Fault Tolerant CORBA. In [OMG 2002a].

[OMG 2002c] OMG (2002c). *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*, chapter 21. Portable Interceptors. In [OMG 2002a].

[OMG 2002d] OMG (2002d). *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*, chapter 8. Dynamic Skeleton Interface. In [OMG 2002a].

[OMG 2002e] OMG (2002e). *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*, chapter 15. General Inter-ORB Protocol. In [OMG 2002a].

[OMG 2002f] OMG (2002f). The CORBA Persistent State Service. Spécifications.

[OMG 2002g] OMG (2002g). *Minimum CORBA Specification (Version 1.0 - formal/02-08-01)*. Needham, MA, U.S.A.

[Or 1983] Or, M. B. (1983). Another advantage of free-choice : Completely asynchronous agreement protocols. In *Proceedings of the 2nd annual ACM symposium on Principles of Distributed Computing (PODC 83)*, pages 27–30, Montreal, Canada. ACM.

[Ousterhout et al. 1988] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., & Welch, B. B. (1988). The Sprite Network Operating System. *Computer*, 21(2) :23–36.

[Parnas 1972] Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15 :1053–1058.

[Peterson et al. 1989] Peterson, L. L., Buchholz, N. C., & Schlichting, R. D. (1989). Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3) :217–246.

[Powell 1991] Powell, D., editor (1991). *Delta-4 : A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag.

[Pérennou & Fabre 1998] Pérennou, T. & Fabre, J.-C. (1998). A Metaobject Architecture for Fault-Tolerant Distributed Systems : the FRIENDS Approach. *IEEE Trans. on Computer, Special Issue on Dependability of Computing Systems*, 47 :78–95.

[Randell 1975] Randell, B. (1975). System structures for software fault tolerance. *IEEE Transaction on Software Engineering 1, 2 (June 1975), 220-232.*, 1(2) :220–232.

[Raymond 1995] Raymond, K. (1995). Reference Model of Open Distributed Processing (RM-ODP) : Introduction.

[Reeves 1997] Reeves, G. E. (1997). What really happened on Mars ? volume 19, issue 54 of the Risks Forum of ACM Committee on Computers and Public Policy, *http ://catless.ncl.ac.uk/Risks/19.54.html*.

[Reiss & Renieris 2000] Reiss, S. P. & Renieris, M. (2000). Generating Java trace data. In *Java Grande Conference (ACM 2000 conference on Java Grande)*, pages 71–77, San Francisco, CA, USA. ACM.

[RIS 2002] RIS (2002). Compte rendu de l'atelier thématique n.3 : «Intergiciels et sûreté de fonctionnement». Compte Rendu d'Atelier, RIS (Réseau d'Ingénierie de la Sûreté de fonctionnement).

[Rodriguez 2002] Rodriguez, M. (2002). *Technologie d'empaquetage pour la sûreté de fonctionnement des systèmes temps-réel*. Thèse de doctorat, Institut National Polytechnique, Toulouse (France).

[Rodriguez et al. 2000] Rodriguez, M., Fabre, J.-C., & Arlat, J. (2000). Formal Specification for Building Robust Real-time Microkernels. In *21st IEEE Real-Time Systems Symposium*, Orlando, Florida, USA.

[Russinovich et al. 1993] Russinovich, M., Segall, Z., & Siewiorek, D. P. (1993). Application Transparent Fault Management in Fault Tolerant Mach. In *Twenty-Third Annual International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 10–19, Toulouse, France.

[Schmidt & Cleeland 1999] Schmidt, D. & Cleeland, C. (1999). Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, 16(4). Special Issue on Design Patterns.

[Schmidt 1998] Schmidt, D. C. (1998). An Architectural Overview of the ACE Framework : A Case-study of Successful Cross-platform Systems Software Reuse. *USENIX login magazine*. Tools special issue.

[Schmidt 2002] Schmidt, D. C. (2002). Middleware for Real-Time and Embedded Systems. *Communications of the ACM*, 45(6 (June)) :43–48.

[Silberschatz et al. 2002] Silberschatz, A., Gagne, G., & Galvin, P. B. (2002). *Operating System Concepts*. Wiley Text Books, 6th edition.

[Smith 1982] Smith, B. C. (1982). *Procedural Reflection in Programming Languages*. Ph.D. thesis, MIT.

[Stallman & Pesch 2002] Stallman, R. & Pesch, R. H. (2002). *Debugging with GDB*. The Free Software Foundation, Boston, MA, USA, 9th edition.

[Stolper 1999] Stolper, S. A. (1999). Streamlined Design Approach Lands Mars Pathfinder. *IEEE Software*, 16(5 (September/October)) :52–62.

[Strom et al. 1988] Strom, R., Bacon, D., & Yemini, S. (1988). Volatile Logging in n-Fault-Tolerant Distributed Systems. In *Eighteenth International Symposium on Fault Tolerant Computing (FTCS)*, pages 44–48.

[Systä 2000] Systä, T. (2000). *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. Ph.D. Thesis, Tampere University.

[Tanenbaum & Woodhull 1997] Tanenbaum, A. S. & Woodhull, A. S. (1997). *Operating Systems : Design and Implementation*. Prentice Hall, 2nd edition.

[Tatsubori et al. 2000] Tatsubori, M., Chiba, S., Killijian, M.-O., & Itano, K. (2000). OpenJava : A Class-based Macro System for Java. In Cazzola, W., Stroud, R. J., & Tisato, F., editors, *Reflection and Software Engineering*, LNCS 1826, pages 119–135. Springer-Verlag.

[Taylor & Wilson 1989] Taylor, D. & Wilson, G. (1989). *In Dependability of Resilient Computers*, chapter The Stratus System Architecture. Blackwell Scientific Publications, Oxford.

[Taylor & Wright 1986] Taylor, D. J. & Wright, M. L. (1986). Backward Error Recovery in a UNIX Environment. In *16th Anual International Symposium on Fault- Tolerant Computing (FTCS)*, pages 118–123, Vienna (Austria).

[Taïani 2003] Taïani, F. (2003). CosmOpen : A Reverse-Engineering Tool for Complex Open-Source Architectures. In *DSN-03 supplemental volume, Student Forum of DSN'03, The International Conference on Dependable Systems and Networks, San Francisco, CA, June 22nd-25th, 2003, (3 p.)*, San Francisco, CA. IEEE Computer Society.

[Taïani et al. 2002] Taïani, F., Fabre, J.-C., & Killijian, M.-O. (2002). Principles of Multi-Level Reflection for Fault-Tolerant Architectures. In *2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*, pages 59–66, Tsukuba (Japan).

[Taïani et al. 2003] Taïani, F., Fabre, J.-C., & Killijian, M.-O. (2003). Towards Implementing Multi-Layer Reflection for Fault-Tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, San Francisco, CA. IEEE Computer Society.

[Theimer & Hayes 1991] Theimer, M. M. & Hayes, B. (1991). Heterogeneous Process Migration by Recompilation. In *the 11th International Conference on Distributed Computing Systems*, pages 18–25.

[Turing 1936] Turing, A. (1936). On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42).

[van Renesse et al. 1998] van Renesse, R., Birman, K. P., Hayden, M., Vaysburd, A., & Karr, D. (1998). Building Adaptive Systems Using Ensemble. *Software–Practice and Experience*, 28(9) :963–979.

[van Renesse et al. 1996] van Renesse, R., Birman, K. P., & Maffeis, S. (1996). Horus : A Flexible Group Communication System. *Communications of the ACM (CACM)*, 39(4) :76–83.

[Wirth 1985] Wirth, N. (1985). *Programming in Modula-2*. Springer Verlag, Berlin, 3ème edition.

[Yokote 1992] Yokote, Y. (1992). The Apertos reflective operating system : The concept and its implementation. In Paepcke, A., editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 414–434, New York, NY. ACM Press.

[Yokote et al. 1989] Yokote, Y., Teraoka, F., & Tokoro, M. (1989). A Reflective Architecture for an Object-Oriented Distributed Operating System. In Cook, S., editor, *Third European Conference on Object-Oriented Programming (ECOOP'89)*, pages 89–106, Nottingham, UK. Cambridge University Press.

[Young et al. 1987] Young, M., Tevanian, A., Rashid, R. F., Golub, D. B., Eppinger, J. L., Chew, J., Bolosky, W. J., Black, D. L., & Baron, R. V. (1987). The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *the Eleventh ACM Symposium on Operating System Principles (SOSP 1987)*, pages 63–76, Austin, Texas. ACM Press.

Index

— A —

abstraction, 10, 12, 13, 15, 20, 35, 50–51, 53, 59–62, 72, 77, 95, 99, 101, 104, 128–137
ACE, 90, 96
ADA, 13, 83
AEGIS, 29
agrégation, 57
AIBO, 29
algorithme, 17, 35, 53, 59–72
APERTOS/ APERIOS, 29
appel système, 11, 12, 16, 40, 78, 98
approximation, 64–70
Ariane 5, 4–5, 9–10
asynchronie, 7, 45, 63, 65, 77, 83, 104
attribut (d'un objet), 13, 22, 42, 49
auto–référencement, 20

— B —

bibliothèque système, 12, 78, 97, *voir* libpthread.so
binding, *voir* liaison
BOA (Basic Object Adapter), 43
brin d'exécution, 49, 50, 57, 58, 68–70, 82–93, 102, 109–114, 117, 129, 132–135, *voir* multitraitement, *voir* réserve de brins
bus à objets, 11–12, 14–15, 29–33, *voir* CORBA

— C —

C++, 12, 27, 49, 51, 62–63, 68, 97, 102, 106, 119
capacité réflexive, 24–27, 59–72, 77
capture d'état, 17, 36–38, 42–47, 63, 76–77, 89–95
  ∼ distribué, 35, 37–38, 65–71
  ∼ symbolique, 48, 91–92
checkpointing, *voir* capture d'état
checkpointing protocols, *voir* capture d'état
checkpoints, *voir* capture d'état
CHORUS, 50
classe, 13–14
cohérence, 36–38, 45–47, 66–71, 76, 79, 83–89, 124
Common Data Representation, 49
communication, 107–111, *voir* connexion
communication de groupe, 43–44
compensation (recouvrement par), 6
compilateur IDL, 80, 113
complexité, xi–xii, 1, 8–17, 47, 51, 53–54, 62, 96, 109, 117–121, 127, 129, 138
composant, xi–xiii, 5, 8–9, 17, 30, 32, 77, 95–98, 117–121, 127, 138
  ∼ autotestable, 6
computational reflection, *voir* réflexivité
connexion, 88, 104, 107–111, 113
Consul, 41
contexte sémantique, *voir* sémantique
contrôle d'impact, 28, 29, 111
contrat, 6, 13

CORBA, 15, 80–82, *voir* ORBACUS
CORBA Persistent State Service, 42, 43
COSMOPEN, 97, 99, 127–139
COTS, *voir* composant
couche, xi, 16–17, 29, 47, 50–51, 54–58, 62, 72, 78, 90, 95–97, 100, 102–104, 108–110, 113, 119, 127, 137, 138
création de brin, 104, 135

— D —

découplage, 9–12, 17, 23, 26, 39, 91, 121
défaillance, 2–4, 7, 36, 38, 76
  ∼ byzantine, 5
  ∼ incontrôlée, 5
  silence sur ∼, 5, 50, 77
DELTA-4, 75
détection d'erreur, 6, 36
déterminisme, 57, 58, 63, 76–77, 79, 82–90, 93, 114
  ∼ par morceau, 83
diagnostic, 6, 32, 50
diagramme comportemental, *voir* modèle
diagramme structurel, *voir* modèle
distribution, *voir* système distribué
diversité, 4–5, 50
DOXYGEN, 97, 102
DYNAMICTAO, 32
Dynamic Interface Repository, 33
Dynamic Invocation Interface, 40, 44
Dynamic Skeleton Interface, 40, 44

— E —

ELECTRA, 43–44
empreinte réflexive, 59–64, 72, 75–78, 119–121
ENSEMBLE, 41, 42
erreur, 2–3
  détection d'∼, 6, 17, 28, 36, 77
  recouvrement d'∼, 6
espace de nommage, 29
état caché, 57
ETERNAL, 43
exo-noyau, 29
extraction de modèle, 95–100, 120, 127
extreme programming, 6

— F —

famille de mécanismes, 61, 72, 75, 119, 120
faute, 2–4, 58, 60
  ∼ persistante, 6
  modèle de ∼s, 5
filtrage, 99, 100, 127, 129, 132–138
flexibilité, 7, 26, 39–40, 47, 82
fournisseur, 15, 16, 118–121, 127
FRANCE TÉLÉCOM, 31
FRIENDS, 27, 49–50

— G —

GARF, 48–49
GNU/LINUX, xiii, 8, 47, 57, 72, 78, 98, 102, 104, 106
graphe, *voir* modèle

— H —

HORUS, 41, 43

— I —

IDL, 41, 44, 113
IIOP, 104
ILOG BROKER, 8, 127
incrémentalité, 28, 106–107
intégrateur, 16, 118–121, 127
intégration, 40, 43–47, 120
interception, 40, 43–49, 62, 79, 84, 87, 106–114, 119
intercession, 25, 26, 49, 61, 92, 93
interdépendance, 8–10, 118

interface, 9, 11, 16, 27, 28, 54–58, 96–98, 107, 118–119
  ∼ d'un objet, 13, 29–33
intergiciel, *voir* bus à objets
interopérabilité, 9–11, 15, 28, 104
interprétation abstraite, 97, 139
interpréteur méta-circulaire, 24, 29
intervalle de recouvrement, *voir* recouvrement
introspection, 25, 26, 49, 61, 91
IONA, 102
IOR, 80
ISIS, 41–42, 48

— J —

JAVA, 12, 15, 24–27, 30, 31, 55–56, 86, 97, 102
JAVA RMI, 15, 31, 43
JONATHAN, 31–32

— L —

liaison, 29–32, *voir* objet de liaison
libpthread.so, 57, 98–100, 103, 104, 132–137
lien inter-niveaux, 56–58, 96, 98
light weight processus, 98
LINUX, *voir* GNU/LINUX
LISP, 22–23
lockstep processors, 7
logged-based recovery, 90
LYNXOS, 50

— M —

MACH, 28
marshalling, *voir* mise à plat
MAUD, 48–49
message en transit, 38, 69
méta-calcul, 25
méta-classe, 32, 107–108, 111
méta-connexion, 107–113
  usine à ∼s, 110–111
méta-donnée, 27
méta-espace, 30
méta-hiérarchie, 29
méta-interface, 21, 24, 25, 32, 40, 72, 118, 121, *voir* interface
  ∼ multi-niveaux, 72, 75, 88–95, 102–114, 119, 137
méta-marqueur, 109
méta-modèle, 20–21, 23–27, 54, 72, 78, 93–100, 102–106, 118–121, 127, 137–138
méta-niveau, 25–26, 30, 49, 88–93, 108, 109, 121
méta-noyau, 50
méta-objet, 26, 29, 30
  méta-méta-objet, 29
méta-programme, 22–23, 28, 54
méta-verrou, 107–108
  usine à ∼s, 110–111
MetaMutex, *voir* méta-verrou
MetaSocket, *voir* méta-connexion
méthode (d'un objet), 13–14, 22, 25, 27, 30, 49, 69, 80, 91, 92, 97
micro–protocole, 42
micro-noyau, 50
MICROSOFT, 11
middleware, *voir* bus à objets
mise à plat, 42, 79, 80
MMU (Memory Management Unit), 11, 57
modèle, *voir* méta-modèle
  ∼ comportemental, 96–100, 104–106, 129–130, 132–137
  ∼ de programmation, 11, 12, 14–16, 23, 29, 40, 41, 47, 51, 54–56, 80–82, 86, 87, 95, 97, 102
  ∼ structurel, 96–97, 102, 110, 129–130
  extraction de ∼, *voir* extraction
modèle de fautes, *voir* faute
modèle de programmation, *voir* modèle
mode noyau, 11

mode utilisateur, 11, 12, 47, 57
MODULA, 13
modularité, 6, 9–10, 13, 14, 16, 17, 26, 29, 31, 33, 44, 48, 120
module, *voir* modularité
MOP, *voir* protocole à méta-objets
multiplexage, 11, 57
multitraitement, 47, 58, 82–95, 98–100, 102, 104, 133, 136, 137
MUSE, 29
mutex, *voir* verrou

— N —

N-version programming, 5
NEOCLASSTALK, 32
niveaux de privilège, 11
niveau d'abstraction, xii, 12, 14–17, 24, 47, 48, 50–51, 54–58, 64, 71, 109, 137
niveau de base, 25–26, 28, 121
nommage, 31
non-déterminisme, *voir* déterminisme
noyau d'un OS, 8, 11–12, 47, 49, 58, 78, 98–101, 133, *voir* exo-noyau, *voir* méta-noyau

— O —

object adapter, 81
OBJECT GROUP SERVICE, 44
objet de base, 26
objet de liaison, 30–32
OMG (Object Management Group), 15
OMNIORB, 85, 90
opacification, 16, 17
OPENC++, 26, 27, 49
OPENCORBA, 32–33
OPENJAVA, 26
OPENORB, 29–31
OPSBROWSER, 129–137
ORB, *voir* bus à objets
ORBACUS, 78, 82, 84, 85, 90, 96, 97, 102–107, 109, 112–114
orienté objet, 12–14, 22, 26, 29, 43, 48, 68, 96, 129
OS, *voir* système d'exploitation

— P —

piecewise determinism, *voir* déterminisme par morceaux
piggybacking, *voir* talonnage
pile, 49, 57, 70, 89–94, 98, 109
pipelining, 58
point de contention, 88–92, 94, 108, 109, 111–113
polymorphisme, 13
portabilité, 39
Portable Object Adapter, 43, 81
POSIX, 11, 12, 51, 55, 58, 62, 78, 80, 85, 88, 89, 95, 98–110, 137
poursuite (recouvrement par), 6
protection, 10, 11, 29, 44–45, 54
protocole à méta-objets, 26, 48, 49
prototype, 102–115
PS–ALGOL, 42
PSYNC, 41
PYTHON, 30

— R —

RATIONAL ROSE, 97
recouvrement
  ∼ par compensation, 7
  intervalle de ∼, 66–67
recouvrement d'erreur, 6
redondance, 4–7, 75
référence (d'un objet), 13, 14, 31, 32, 80, 109
réflexivité, 19–33, 40, 47–50
  ∼ multi-niveaux, 87, 93–95
réification, 24–26, 49, 61, 88, 91, 107–113
réplication, 5, 7, 36, 44, 48, 75–95

~ active, 75–76
~ passive, 75–76
~ semi-active, 75–77
reprise, 37–38, 45–46, 61, 63, 71, 76
reprise (recouvrement par), 6, 35
requête, 43, 55, 80–95
réserve de brins, 81–82
rétro-conception, 95–100, 120, 127–132
réutilisabilité, 35, 39, 106, 119
reverse-engineering, *voir* rétro-conception
RM-ODP, 29, 31

— S —

scions, 32
sémantique, 9, 15, 16, 23, 24, 27, 47, 51, 57, 71, 86, 87, 95, 99, 110, 133, 139
contexte ~, 108–110
séparation des préoccupations, 39
separation of concerns, *voir* séparation des préoccupations
signifié, 20–21, 23, 26, 55, 123
signifiant, 20–21, 23, 26, 123
silence sur défaillance, *voir* défaillance
SIMULA, 12
skeleton, 32, 81, 113
SMALLTALK, 12, 32, 48
socket, *voir* connexion
SOLARIS, 47
souche, 32, 80
sous-classes, 14
stub, 32
sûreté de fonctionnement, xi–xii, 1–8
surrogates, 31
synchronie, 7, 36
synchronisation, 7, 36–38, 41, 43–44, 48, 49, 76, 79, 82–95, 102, 106–111, 114, *voir* verrou
syscall, *voir* appel système
systèmes d'exploitation, 50
système complexe, *voir* complexité
système d'exploitation, 10–12, 107–111
système distribué, 7–8

— T —

talonnage, 61, 110
TAO, 82, 85, 90, 96, 97
THINK, 29, 31
thread pool, *voir* réserve de brins
tolérance aux fautes, xii, 4–8, 16–17, 35–51, 53, 59, 117–121, 127, 138
transcendance logicielle, 108–110
TRANSIS, 41
translation, 57
transparence, 10, 14–16, 30, 39–40, 47, 50

— U —

UNIX, 11
usable knowledge, 65
usine
~ à liaisons, 31
~ à méta-connexions, 110–111
~ à méta-verrous, 110–111
utilisateur, 2, 10, 11, 16, 51, 60, 104, 121

— V —

validité d'un algorithme, 59–71
verrou, 57, 82–89, 107–111
VXWORKS, 8, 117–118, 127

— W —

WIN32, 11
WINDOWS, 11, 15

— X —

X-WINDOWS, 108