THESE

présentée

pour obtenir le titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

École doctorale : Informatique et Télécommunications

Spécialité : Réseaux et Télécommunications

Par M. Ludovic **APVRILLE**

CONTRIBUTION A LA RECONFIGURATION DYNAMIQUE DE LOGICIELS EMBARQUES TEMPS-REEL : APPLICATION A UN ENVIRONNEMENT DE TELECOMMUNICATION PAR SATELLITE

SOUTENUE LE 11 JUIN 2002 DEVANT LE JURY COMPOSE DE :

PRESIDENT	**C. FRABOUL**
RAPPORTEURS	**R. CASTANET** **J.-P. THOMESSE**
EXAMINATEURS	**F. BOSSARD** **I. BURET** **J.-P. COURTIAT**
DIRECTEUR DE THESE	**M. DIAZ**
CO-DIRECTEUR	**P. SENAC**
CO-ENCADREMENT	**F. RICHARD**

Remerciements

Les travaux présentés dans ce mémoire ont été réalisés à l'Ecole Nationale Supérieure d'Ingénieurs de Constructions Aéronautiques (ENSICA), sur le site toulousain de la société Alcatel Space Industries et enfin, au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de Recherche Scientifique (CNRS).

En tout premier lieu, je tiens à exprimer mes sincères remerciements à Patrick Sénac, chef du département Mathématiques appliquées et Informatique de l'ENSICA. Il m'a fourni un soutien sans faille et un encadrement de très grande qualité pendant toute la durée de ces travaux.

Je souhaite aussi exprimer ma profonde gratitude à Michel Diaz, responsable du groupe Outils et Logiciels pour la Communication (OLC) du LAAS, pour avoir accepté de diriger ces travaux de recherches et pour ses nombreux conseils avisés.

Le travail présenté dans ce mémoire de thèse a été réalisé grâce au support financier d'Alcatel Space Industries. Je remercie tout particulièrement Christian Tabard qui a eu l'initiative de cette thèse et François Richard pour son encadrement.

Je suis également très reconnaissant à :

Monsieur F. Bosssard, ingénieur au CNES,
Madame I. Buret, ingénieur de recherche chez Alcatel Space Industries,
Monsieur R. Castanet, professeur des universités à l'ENSEIRB,
Monsieur J.-P. Courtiat, directeur de recherche au CNRS,
Monsieur C. Fraboul, professeur de l'Institut National Polytechnique de Toulouse,
Monsieur J.-P. Thomesse, professeur des universités à l'ENSEM,

pour l'honneur qu'ils ont fait en participant à mon jury de thèse. Je tiens à remercier en particulier Messieurs R. Castanet et J.-P. Thomesse pour le temps qu'ils ont accepté de consacrer à la relecture de ces travaux.

Je tiens aussi à exprimer mes sincères remerciements à toute l'équipe d'enseignants-chercheurs du département informatique de l'ENSICA : Yves Caumel, Laurent Dairaine , Fabrice Francès, Jérôme Lacan, Tanguy Pérennou et Pierre de Saqui-Sannes. Je remercie également les doctorants et plus particulièrement Emmanuel, Ernesto, Laurent, Luis et Romain avec qui j'ai partagé mon bureau, ainsi que les appelés scientifiques et stagiaires de DEA que j'ai pu côtoyer et apprécier pendant ces années. Je remercie enfin René Espourteau et Bernard Jarlan pour leur disponibilité et leur capacité à résoudre les 1001 tracas quotidiens. L'ambiance de travail et d'amitié au sein de ce département a grandement contribué à la réussite de cette thèse.

Je remercie aussi les membres du groupe OLC (LAAS) et tout particulièrement Jean-Pierre Courtiat et Christophe Lohr dont j'ai pu apprécier les grandes qualités humaines lors de nos fructueux échanges. Mes collègues d'Alcatel m'ont aussi apporté une précieuse aide. Je remercie plus précisément les membres de mon service, les membres de l'équipe logiciel vol, et enfin les membres des équipes travaillant sur les architectures des nouveaux systèmes de télécommunication par satellite.

Un grand merci aussi à tous mes amis qui m'ont soutenu et encouragé pendant ces années, et en particulier à Sophie et Nathalie qui ont pu se libérer et être à mes côtés le jour de ma soutenance. Merci aussi à Laurent qui, du premier jour de ma thèse jusqu'à la soutenance, n'a eu de cesse de faire avancer cette thèse en me communiquant son enthousiasme, son sérieux, et son investissement personnel dans son travail.

Mes derniers remerciements sont pour l'ensemble de ma famille. Tout d'abord pour ma sœur, Béatrice, qui quelques années avant moi a soutenu brillamment une thèse. Elle a alors créé en moi l'envie de suivre ses traces. Ensuite, pour mes parents, qui m'ont beaucoup encouragé. Je les remercie aussi pour leur important travail de relecture de ce manuscrit. Enfin, pour ma femme, Axelle, pour son soutien pendant les moments de désespoir, pour sa patience et pour ses nombreuses aides (relectures d'articles, etc.). Du fond du cœur, j'aimerais lui adresser le plus grand remerciement. Et même si elle pense encore que les "RdPFT-RD" sont des amis de R2-D2, c'est à elle que je dédie ce mémoire.

Table des matières

CHAPITRE 1 ..... **9**

INTRODUCTION **9**

1.1. POSITIONNEMENT DES SATELLITES DE TELECOMMUNICATION 9
1.2. L'EVOLUTIVITE NECESSAIRE DES SYSTEMES DE TELECOMMUNICATION PAR SATELLITE 10
1.3. PROBLEMATIQUE DE LA MISE A JOUR LOGICIELLE 10
1.4. CONTRIBUTIONS 11
1.5. ORGANISATION DU DOCUMENT 12

CHAPITRE 2 ..... **13**

LES ENVIRONNEMENTS SPATIAUX DE TELECOMMUNICATION **13**

2.1. INTRODUCTION 13
2.2. SYSTEMES SATELLITES DE TELECOMMUNICATION 13
2.2.1 Architecture générale d'un système satellite *13*
2.2.2 Charge-utile de télécommunication *14*
2.2.3 Besoins de reconfiguration dynamique des logiciels embarqués *18*
2.3. LOGICIELS EMBARQUES SUR SATELLITES 18
2.3.1 Architecture des calculateurs embarqués *18*
2.3.2 Environnement matériel *18*
2.3.3 Environnement logiciel *19*
2.4. CONCLUSION 20

CHAPITRE 3 ..... **23**

APPROCHES DE RECONFIGURATION DYNAMIQUE **23**

3.1. INTRODUCTION 23
3.2. MAINTENANCE D'APPLICATIONS : DEFINITION ET MECANISMES SUPPORTS 24
3.2.1 Définition de la maintenance *24*
3.2.2 Mécanismes supports à la maintenance *24*
3.3. LA RECONFIGURATION DYNAMIQUE : DEFINITIONS, MOTIVATIONS ET CARACTERISTIQUES 24
3.3.1 Terminologie associée à la reconfiguration dynamique *24*
3.3.2 Motivations des mécanismes de mise à jour logicielle sans arrêt de service *26*
3.3.3 Caractéristiques générales *27*
3.4. ANALYSE DES CONTRIBUTIONS 30
3.4.1 Introduction *30*
3.4.2 Besoin de reconfiguration dynamique *30*
3.4.3 Solutions matérielles *31*
3.4.4 Solutions logicielles *31*
3.4.5 Couche systèmes d'exploitation *31*
3.4.6 Couche Middleware *32*
3.4.7 Couche applicative : le patch logiciel *33*
3.4.8 Couche applicative : Architectures à composants *34*
3.5. COMPARAISON DES MECANISMES ET POSITIONNEMENT 39
3.5.1 Taxonomie des mécanismes *39*
3.5.2 Positionnement *40*
3.6. CONCLUSION 41

CHAPITRE 4 ..... **43**

UN CADRE FORMEL POUR LA MODELISATION DE RECONFIGURATION DYNAMIQUE 43

4.1. INTRODUCTION 43
4.2. UNE ARCHITECTURE LOGICIELLE INTRINSEQUEMENT RECONFIGURABLE 44

4.2.1 Introduction *44*
4.2.2 Définition de l'architecture logicielle *44*
4.2.3 Conclusion *47*
4.3. UN CADRE FORMEL DE MODELISATION LOGICIELLE BASE SUR UML 47
4.3.1 Besoin *47*
4.3.2 Introduction à TURTLE *48*
4.3.3 Présentation de l'environnement TURTLE *50*
4.4. TURTLE COMME ENVIRONNEMENT SUPPORT DE MODELISATION POUR LA RECONFIGURATION DYNAMIQUE 61
4.4.1 Introduction *61*
4.4.2 Principe général *61*
4.4.3 Modélisation des concepts architecturaux *62*
4.4.4 Méthodologie de vérification de propriétés sur la modélisation *67*
4.5. CONCLUSION 76

CHAPITRE 5 77

MECANISMES DE RECONFIGURATION DYNAMIQUE 77

SPECIFICATION, METHODOLOGIE ET MISE EN ŒUVRE 77

5.1. INTRODUCTION 77
5.2. PROCESSUS DE RECONFIGURATION DYNAMIQUE 78
5.2.1 Principe général *78*
5.2.2 Étapes du processus *78*
5.3. OPERATIONS DE RECONFIGURATION DYNAMIQUE 79
5.3.1 Notion d'opérations de reconfiguration dynamique *79*
5.3.2 Caractéristiques et terminologie des opérations *79*
5.3.3 Taxonomie des opérations *80*
5.4. LANGAGE DE RECONFIGURATION DYNAMIQUE 84
5.4.1 Besoin d'un langage *84*
5.4.2 Réseaux de Petri à Flux Temporels pour la Reconfiguration Dynamique *85*
5.4.3 Spécification des opérations de reconfiguration *89*
5.5. METHODOLOGIE 95
5.5.1 Introduction *95*
5.5.2 Principe général *95*
5.5.3 Description des étapes méthodologiques *96*
5.6. PRAGMATIQUE DE LA RECONFIGURATION 104
5.6.1 Introduction *104*
5.6.2 Architecture générale du système support *104*
5.6.3 Environnement support bord *106*
5.6.4 Environnement support sol *112*
5.7. CONCLUSION 115

CHAPITRE 6 117

ÉTUDE DE CAS : MAINTENANCE D'UN LOGICIEL OBP 117

6.1. INTRODUCTION 117
6.2. PRESENTATION D'UN LOGICIEL CHARGE-UTILE 118
6.3. MODELISATION ET VALIDATION FORMELLE 118
6.3.1 Modélisation logicielle *118*
6.3.2 Validation formelle de reconfiguration dynamique *120*
6.4. ÉVALUATION DES MECANISMES SUPPORTS 124
6.5. CONCLUSION 128

CHAPITRE 7 129

CONCLUSION GENERALE 129

7.1. RAPPEL DES CONTRIBUTIONS 129
7.2. PERSPECTIVES 131

ANNEXE A 133

OPERATEURS TURTLE DES DIAGRAMMES D'ACTIVITE 133

ANNEXE B 135

DEFINITION DES OPERATIONS DE RECONFIGURATION DYNAMIQUE 135

ACRONYMES 137

BIBLIOGRAPHIE DE L'AUTEUR 139

BIBLIOGRAPHIE 141

Chapitre 1

Introduction

Arthur Clarke soulignait dès 1945 l'intérêt des satellites géostationnaires comme support de réseaux aptes à couvrir toute la surface terrestre [Clarke 1945]. Les limites technologiques repoussèrent au début des années soixante les premiers essais de communication par satellite. Depuis, les satellites ont été largement utilisés pour les communications téléphoniques grandes distances et pour les retransmissions télévisées. Cependant, le positionnement des satellites vis-à-vis des architectures de communication élaborées depuis cette époque est à présent remis en cause par le coût de transmission de données via des infrastructures terrestres et par les améliorations spectaculaires des technologies terrestres optiques [Bigo 2000].

1.1. Positionnement des satellites de télécommunication

Face à la concurrence décrite précédemment, les systèmes de transmission de données par satellite doivent se recentrer sur des services utilisant au mieux les capacités intrinsèques de tels systèmes, à savoir, l'ubiquité d'accès, les communications longue distance et la capacité de diffusion. L'exploitation de ces capacités conduit à envisager l'utilisation des systèmes satellites dans trois contextes étroitement couplés au réseau Internet [Blineau 2001] : premièrement, un rôle d'accès des utilisateurs finaux aux frontières du réseau Internet (*last mile access*), avec une voie retour, soit filaire par adaptation des protocoles de routage [Duros 2001], soit satellitaire grâce à l'amélioration des technologies relatives aux antennes ; deuxièmement, un rôle de support aux réseaux d'acheminement de contenus (*Content Delivery Network*) : les données sont acheminées soit directement vers l'utilisateur final, soit vers des entités intermédiaires de stockage situées à la périphérie des réseaux (ce rôle permet d'exploiter au mieux la capacité de diffusion des satellites pour les communications de type point à multipoint) ; enfin, un rôle d'interconnexion entre points d'accès afin d'exploiter la capacité de grande distance des liaisons satellites.

Le renforcement de la concurrence provenant des réseaux terrestres amène non seulement à profiter au mieux des capacités intrinsèques des systèmes de télécommunication par satellite, mais aussi à optimiser au mieux la bande passante. Quatre niveaux d'optimisation ont ainsi été identifiés [Roullet 1999]. Au niveau radio, l'optimisation passe par des mécanismes de *réutilisation de fréquence* (charges utiles multi-faisceaux) et d'*amélioration des formes d'onde*. Au niveau cellule, cela passe par l'amélioration de l'occupation des slots des trames par utilisation d'un *multiplexage dynamique fréquentiel et temporel* sur les liens. Au niveau paquet, cela se traduit par l'utilisation d'un *commutateur embarqué de paquets* [Wittig 2000] et par la prise en compte des communications de type *point à multipoint*. Enfin, au niveau transport, cela se traduit par l'optimisation des mécanismes de bout en bout et plus particulièrement par l'optimisation des mécanismes relatifs au protocole dominant de l'Internet qu'est TCP [Balakrishnan 1996][Durst 1996][Allman 1999].

1.2. L'évolutivité nécessaire des systèmes de télécommunication par satellite

Le positionnement actuel des systèmes satellites, ainsi que l'optimisation accrue de la bande passante à plusieurs niveaux se traduit par un accroissement des fonctionnalités offertes par les charges utiles. Si le transport des données et certains algorithmes critiques du point de vue performance nécessitent des circuits dédiés, les autres fonctionnalités, pour des raisons de complexité et de flexibilité, doivent être implantées de façon logicielle. Par exemple, dans le cas de charges utiles régénératives multi-spots avec commutation de paquets, la commutation est réalisée de façon matérielle, mais son administration est implantée en partie de façon logicielle. Cela concerne par exemple l'interfaçage des protocoles des couches supérieures (par ex. ATM) aux mécanismes d'accès au canal partagé (par ex., DAMA, cf. [Roullet 1999]).

Cependant, les protocoles terrestres sont sujets à de nombreuses évolutions, tout comme la nature des flux de données transportés par les charges utiles régénératives multi-spots. La durée d'exploitation de quinze années de ces charges utiles au regard des évolutions précitées nous laisse penser que leurs fonctionnalités peuvent rapidement devenir obsolètes et donc non concurrentielles. La pérennité de telles charges utiles est donc conditionnée par leur capacité d'évolutivité logicielle. De plus, le besoin de continuité de service inhérent à ces charges utiles soulève le problème de la modification des fonctionnalités sans que le service offert aux utilisateurs soit perturbé.

Nous identifions là un besoin fondamental en termes d'évolutivité des fonctionnalités logicielles des charges utiles de télécommunication sans interruption de service.

1.3. Problématique de la mise à jour logicielle

Le patch logiciel, seule technique de mise à jour à des fins de maintenance effectivement utilisée dans le spatial [Garrido 1998][Stevens 2000], ne répond pas au besoin d'évolutivité précité en raison de la discontinuité du service qu'elle engendre. Si la technique classique de la reconfiguration dynamique répond potentiellement au problème de la continuité d'exécution de l'application, son adéquation aux logiciels spatiaux soulève des problèmes en termes de méthodologie de développement d'applications embarquées et intrinsèquement reconfigurables, et de mécanismes embarqués supports à la réalisation des reconfigurations.

En terme de méthodologie, les contraintes inhérentes aux applications spatiales (par exemple le respect d'un standard de codage) et la nature dynamique de la reconfiguration doivent être prises en considération. La nature dynamique se caractérise par le respect, lors de la reconfiguration, de deux types de contraintes relatives au logiciel considéré : les contraintes intrinsèques et les contraintes extrinsèques [Kramer 1985]. Les contraintes intrinsèques concernent les contraintes logiques et temps-réel (par exemple, respect d'échéances temporelles, cf. [Segal 1993]). Les contraintes extrinsèques font référence au service offert par un logiciel à son environnement extérieur et englobent à ce titre les contraintes de ressources offertes par le support d'exécution [Kramer 1985]. La nécessité de satisfaire ces deux types de contraintes quelle que soit la phase d'exécution du logiciel, y compris pendant les phases de reconfiguration, incite à la modélisation de ces contraintes à des fins de validation formelle.

Les environnements bords d'exécution des logiciels embarqués, qui sont fortement contraints en termes de ressources matérielles supports, n'ont pas été conçus pour exécuter des applications reconfigurables et pour piloter les reconfigurations d'applications.

En résumé, afin de répondre au problème de la reconfiguration dynamique d'applications spatiales embarquées, il est nécessaire de proposer :

- une méthodologie de développement de services logiciels de télécommunication vérifiant les deux critères suivants ;
  - une prise en compte des contraintes du logiciel embarqué temps-réel, ce qui se traduit par la nécessité d'un support formel de simulation, vérification et validation ;

  - une garantie en termes de continuité de service de télécommunication en particulier lors des phases de reconfiguration du logiciel ;
- un support à l'évolution et à la reconfiguration du logiciel spatial embarqué.

En fait, par ses objectifs, cette thèse s'inscrit tout naturellement dans une problématique plus large et générale, celle des systèmes logiciels s'exécutant dans un environnement contraint en termes de ressources, de disponibilité et de temps-réel. Ainsi, les solutions proposées en matière de reconfiguration dynamique de logiciels applicatifs devraient pouvoir être mises en œuvre sur tout autre système logiciel fonctionnant dans un environnement spatial ainsi que sur certains types de logiciel temps-réel embarqué.

1.4. Contributions

L'objectif de ce mémoire est de présenter une solution globale à la problématique de la reconfiguration dynamique des logiciels embarqués. Nous qualifions cette solution de globale dans la mesure où la contribution que nous proposons comprend non seulement une méthodologie innovante de développement de logiciels intrinsèquement reconfigurables mais aussi les mécanismes supports de la reconfiguration dynamique.

Au terme de ce travail, nous avons identifié cinq points qui caractérisent plus particulièrement notre contribution.

Définition d'une méthodologie logicielle

De nombreux travaux de recherche proposent des environnements de reconfiguration dynamique [Segal 1993], mais peu d'entre eux présentent une méthodologie logicielle applicable dans un environnement industriel de développement d'applications spatiales embarquées, et prenant en compte de façon formelle les contraintes de reconfiguration. La méthodologie proposée dans cette thèse s'appuie sur un cycle itératif basé sur le langage UML [Booch 1999]. Les mécanismes d'extension de ce langage sont utilisés afin d'enrichir l'expressivité temps-réel d'UML [Apvrille 2001b]. La dérivation des diagrammes UML d'une application en un langage formel intermédiaire, RT-LOTOS [Courtiat 2000] autorise la validation formelle de ces opérations de reconfiguration.

Modélisation et validation des contraintes applicatives et en particulier de la continuité de service

Notre deuxième contribution concerne la modélisation et la validation des contraintes intrinsèques et extrinsèques qui doivent être respectées par le logiciel lors de ses mises à jour. Des contributions mettent en évidence l'intérêt de vérifier de telles contraintes avant l'exécution d'une reconfiguration dynamique [Stewart 1997][Cailliau 2001] mais aucune solution satisfaisante n'a été proposée à ce jour. Nous proposons d'enrichir la méthodologie logicielle proposée par une modélisation explicite de ces contraintes à des fins de validation formelle d'opérations de reconfiguration dynamique. Cette modélisation s'appuie sur notre cadre formel basé sur UML.

Définition, modélisation et implantation d'une architecture logicielle

Notre troisième contribution concerne la définition, la modélisation, et l'implantation d'une architecture logicielle applicative et d'une architecture logicielle support des mécanismes de reconfiguration.

L'architecture logicielle applicative proposée est issue, d'une part de travaux précurseurs en termes de reconfiguration dynamique d'applications architecturées sous forme de composants [Kramer 1985][Purtilo 1991][Oreizy 1998] et d'autre part, de la structure inhérente aux logiciels spatiaux et plus généralement aux logiciels temps-réel embarqués. La définition de l'architecture logicielle s'accompagne de la définition d'opérations de reconfiguration dynamique applicables à cette architecture. Ensuite, notre méthodologie logicielle est enrichie par un processus de

développement des logiciels selon cette architecture. Ensuite, l'implantation de cette architecture est réalisée par utilisation d'un langage objet fonctionnant au-dessus d'un environnement logiciel et matériel du spatial. Enfin, une implantation montre la faisabilité des mécanismes proposés.

L'architecture logicielle support est décrite en termes de services d'exécution d'opérations logicielles de reconfiguration dynamiques et de services protocolaires utilisés par le processus de reconfiguration dynamique. L'implémentation réalisée consiste en un *middleware* s'intercalant entre l'application reconfigurable et le système d'exploitation sous-jacent.

Proposition et mise en œuvre d'un processus de reconfiguration dynamique

Le quatrième point de notre contribution concerne la proposition et la mise en œuvre d'un processus de reconfiguration d'applications embarquées. Ce dernier est mis en œuvre grâce à l'utilisation d'outils permettant de réaliser (1) la validation a priori [Courtiat 2000] ; (2) la génération d'un script de reconfiguration stipulant d'une part, les opérations à exécuter sur l'application pour la reconfigurer et d'autre part, les contraintes logiques et temporelles relatives à ces opérations. Ce script est décrit avec un formalisme basé sur les réseaux de Petri à flux temporels [Diaz 1993][Sénac 1996] ; (3) la mise en œuvre des mécanismes supportant la reconfiguration, et notamment, les mécanismes relatifs à l'intégration dynamique de code [Liang 1998]. Afin d'évaluer le processus de reconfiguration, une application de télécommunication a été modélisée et implémentée selon l'architecture précédemment définie.

Proposition d'une plate-forme expérimentale

Enfin, l'approche support a été expérimentée par l'utilisation d'une plate-forme d'émulation de systèmes de télécommunication par satellite [Apvrille 2001a]. Cette plate-forme est basée sur un réseau de stations informatiques représentant chacune une ou plusieurs entités du système satellite. Les communications entre ces entités transitent par une station spécifique [Rizzo 2000] dont le rôle est d'émuler les caractéristiques de transmission entre les entités du système satellite réel.

1.5. Organisation du document

Le chapitre 2 montre l'intérêt de la reconfiguration dynamique des logiciels au service des systèmes spatiaux. Les différentes approches de reconfiguration dynamique sont par la suite exposées et confrontées à nos besoins dans le chapitre 3. La mise en évidence du manque de prise en compte par les méthodologies utilisées à ce jour des contraintes de reconfiguration nous conduit à proposer, dans le chapitre 4, un environnement de modélisation et de validation adapté aux logiciels intrinsèquement reconfigurables. Pour cela, nous présentons une architecture logicielle de haut niveau, puis un cadre formel de modélisation des applications et de leurs contraintes. La validation formelle a priori du respect de ces contraintes lors des reconfigurations est explicitée. Le chapitre 5 expose les mécanismes supportant la reconfiguration dynamique d'applications développées selon l'architecture présentée au chapitre précédent. Tout d'abord, le processus de réalisation des reconfigurations est présenté. La suite du chapitre est consacrée d'une part, à la description de ce script de reconfiguration, et d'autre part, à sa mise en œuvre. Le chapitre 6 propose une étude de cas sur un logiciel embarqué au service d'une charge utile régénérative et multi-spots offrant des fonctions de télécommunication. La conclusion présente une synthèse de nos différentes contributions, puis expose les perspectives de nos travaux.

Chapitre 2

Les environnements spatiaux de télécommunication

Résumé. Ce chapitre décrit les environnements spatiaux de télécommunication en deux étapes. Dans un premier temps, nous mettons en évidence l'évolution récente des charges-utiles des satellites de télécommunication auxquelles sont conférées des fonctionnalités de commutation et de multiplexage dynamique. La mise en œuvre de telles charges utiles est particulièrement complexe et nécessite une implantation matérielle et logicielle de fonctionnalités avancées. Nous montrons que ces fonctionnalités peuvent être rapidement rendues obsolètes par l'évolution des protocoles et flux de données utilisateurs d'où le besoin d'évolution de ces fonctionnalités pendant leurs quinze années moyennes d'exploitation. Plus généralement, nous détaillons les motivations en faveur de la reconfigurabilité des applications spatiales embarquées à savoir la correction d'erreur, l'évolutivité et enfin la réutilisation. Dans un deuxième temps, nous mettons en évidence le manque de prise en compte des besoins de reconfiguration dynamique en termes de méthodologie logicielle et d'implantation des applications des systèmes satellites.

2.1. Introduction

L'objectif de ce chapitre est double. Le premier objectif est de montrer que les fonctionnalités des nouvelles charges utiles de télécommunication implantées de façon logicielles doivent bénéficier de mécanismes d'évolutivité permettant de garantir leur compétitivité pendant leurs années d'exploitation. Le deuxième objectif de ce chapitre est de montrer que les environnements logiciels actuellement utilisés dans le domaine du spatial ne tiennent pas compte de ce besoin de reconfigurabilité.

Ce chapitre est organisé comme suit. Tout d'abord, nous présentons pour les systèmes satellites de télécommunication : l'aspect plate-forme puis l'aspect charge utile. Nous décrivons l'architecture des nouvelles charges utiles de télécommunication et les fonctionnalités associées à ces charges utiles. Nous mettons en évidence le besoin de les implanter en partie de façon logicielle et le besoin d'évolutivité de ces fonctionnalités au cours de leur fonctionnement. Plus généralement, nous mettons en évidence les besoins de reconfigurabilité des logiciels du spatial. La deuxième section est consacrée aux logiciels embarqués sur satellites. Plus particulièrement, nous analysons ces environnements (implantation, méthodologie) par rapport au besoin de reconfiguration des fonctions logicielles des charges utiles de télécommunication. Enfin, nous concluons le chapitre.

2.2. Systèmes satellites de télécommunication

2.2.1 Architecture générale d'un système satellite

Un système satellite est constitué d'un segment spatial et d'un segment terrestre. Le segment spatial regroupe le satellite ainsi que le centre de contrôle du satellite qui est responsable du maintien en opération du satellite (orbite, gestion des équipements, etc.). Le segment terrestre concerne les stations sols de transmission de données des opérateurs des liens satellite et les équipements utilisateurs.

Un satellite est lui-même constitué de deux ensembles distincts : la plate-forme et la charge utile. La plate-forme assure la logistique nécessaire à la charge utile pour que celle-ci puisse réaliser sa mission. Ainsi, la plate-forme est en charge du système de propulsion du maintien en orbite (système de contrôle d'attitude), de l'alimentation électrique, du contrôle thermique, des télécommandes et télémesures et enfin de la structure mécanique de l'ensemble du satellite. La charge utile d'un satellite est responsable d'une mission. Dans le cas d'un satellite de télécommunication, la charge utile consiste par exemple en une antenne de réception, une antenne d'émission, et une chaîne de traitement entre ces deux antennes.

2.2.2 Charge-utile de télécommunication

2.2.2.1 *L'historique*

La Figure 2-1 met en évidence l'évolution des charges utiles de télécommunication depuis les années 1960. Les fonctionnalités logicielles mises en œuvre dans les charges utiles sont d'autant plus complexes que la couche de transmission impliquée est haute. Hormis les surfaces réfléchissantes, le répéteur constitue l'équipement le plus élémentaire en télécommunication spatiale. Son rôle est d'éliminer le bruit du signal reçu, d'amplifier ce signal en puissance, puis de l'émettre sur la fréquence du faisceau descendant via une antenne d'émission. Un satellite équipé de répéteurs est qualifié de satellite transparent (*bent-pipe*). Les fonctionnalités mises en œuvre de façon logicielle sur de tels équipements (répéteurs) concernent des fonctions classiques de gestion des équipements satellite : démarrage, diagnostic, remise à zéro, arrêt. Les années 1980 ont vu l'avènement de techniques de gestion de l'accès au canal. Les années 1990 ont été marquées par l'apparition de charges utiles qualifiées de numériques dans la mesure où le signal est démodulé à bord. Certains systèmes permettent de router des informations numériques entre desfaisceaux d'onde distincts (spots). Le routage est alors statique, la configuration du routage pouvant être éventuellement modifiée par télécommande. Récemment, des charges utiles capables de commuter des paquets entre spots et de gérer des qualités de service adaptées à des sémantiques de flux diversifiées ont vu le jour (systèmes dits *NGS – New Generation Satellite*) [Bem 2000][Farserotu 2000].

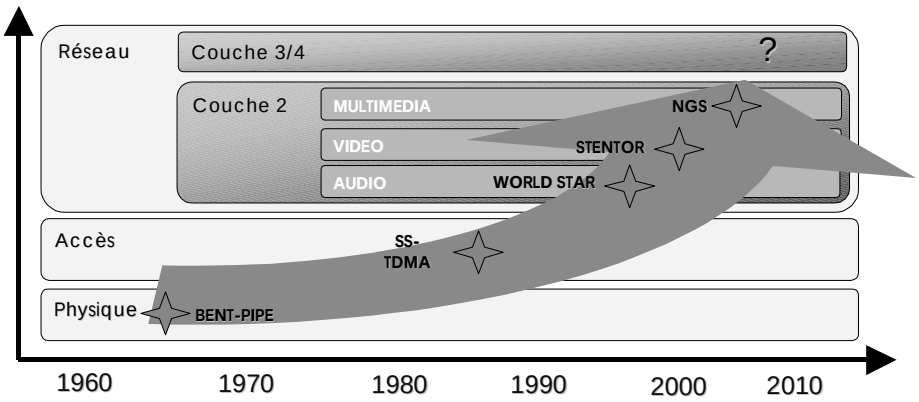


Figure 2-1. Évolution en fonction du temps des charges utiles des satellites de télécommunication.

2.2.2.2 *Charges utiles avec commutation embarquée de paquets*

Gains de bande passante

La concurrence du transport de données multimédia amène les constructeurs de charges utiles de télécommunication à optimiser la gestion de la bande passante et par-là même à proposer des architectures de charges utiles capables de gérer plusieurs spots, ce qui autorise la réutilisation de fréquences à l'instar des systèmes terrestres de téléphonie mobile. Afin d'assurer une connectivité totale entre les différents spots, les nouvelles charges utiles autorisent la commutation de données

d'un spot à l'autre. Trois types de charge utile sont en compétition : les charges utiles à commutation statique de circuit, à commutation de circuits programmée par télécommande et enfin, à commutation de paquets [Parise 2001]. Ces dernières permettent la commutation de cellules de type ATM ou DVB en garantissant la qualité de service associée aux flux gérés.

L'optimisation de la bande passante consiste aussi à multiplexer de façon dynamique les flux sur la trame montante et descendante. Dans le cas d'un système NGS, cela consiste notamment à accepter un trafic utilisateur supplémentaire et de façon temporaire lorsque des slots d'allocation sont laissés vacants par les flux acceptés dans le système.

Architecture système

Nous considérons par la suite un système satellite constitué des éléments listés ci-dessous. Cette architecture est directement inspirée du projet SAGAM [SAGAM 1998][Roullet 1999]. Les entités systèmes sont :

- un satellite géostationnaire multispot avec commutateur embarqué de cellules ATM ; ce commutateur possède une entrée/sortie par spot ;
- un centre de contrôle réseau dit *NCC* (*Network control Center*) ;
- des stations utilisateurs dites *UES* (*User Earth Station*) d'émission et réception de données. Ces stations peuvent être interconnectées à des sous-réseaux de type ATM ou IP, la station sol assurant l'interface avec le réseau satellite.

Architecture charge utile

L'architecture d'une charge utile à commutation de paquets est la suivante (cf. Figure 2-2) : une fois le signal reçu ramené en bande de base, il est démodulé. Les cellules obtenues sont aiguillées dans des buffers d'entrée de la matrice de commutation. Ensuite, elles sont routées et réparties, selon leur destination et leur qualité de service, dans des buffers de sortie. Enfin, les cellules sont codées puis émises via les antennes de sortie.

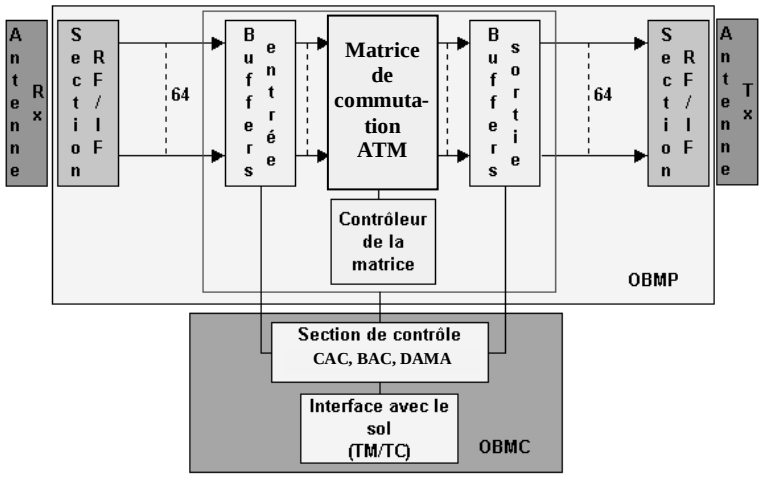


Figure 2-2. Architecture fonctionnelle d'une charge utile avec commutation.

Fonctionnalités mises en œuvre

Outre les classiques fonctionnalités d'administration des équipements, deux modules fonctionnels de support à la commutation sont présentés par la suite [Roullet 1999][Combes 2001](cf. Figure 2-2) :

- l'*OBMP* (*On Board Multimedia Processing*) est en charge du contrôle de la matrice, et de sa configuration ;

– l'*OBMC* (*On Board Multimedia Controler*) est en charge de la gestion des connexions (CAC) et de l'allocation de trafic sur les liens montants (DAMA) et descendants (BAC). Ces fonctionnalités sont réparties en une entité serveur placée dans le système satellite et une entité cliente placée dans les terminaux utilisateur (cf. Figure 2-3). On suppose par la suite que le CAC est placé au sol alors que le DAMA et le BAC sont placés à bord.

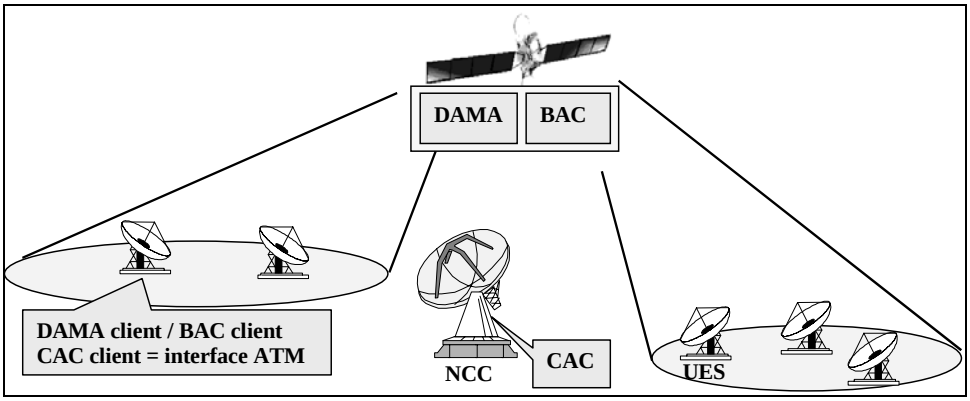


Figure 2-3. Répartition des entités protocolaires dans un système de type NGS. On suppose que le gestionnaire de connexion est implanté au sol.

Le CAC est en charge du contrôle préventif (acceptation de connexions ATM, demandes de modification de paramètres de QoS). Le calcul d'acceptation d'une nouvelle connexion tient compte des ressources aller et retour sur les liens montants et descendants. Le CAC est déclenché par émission d'une cellule SETUP depuis une UES.

Le BAC est en charge de l'allocation des slots sur la trame descendante. Ces slots sont réservés avant tout par le CAC lors de l'acceptation d'une connexion ATM. Le BAC peut aussi allouer des slots supplémentaires aux connexions de type UBR sur émission explicite d'un signal (appelé BAC-sig) depuis une UES. L'émission de ce signal est conditionnée par différents paramètres de l'UES et notamment par le taux de remplissage des buffers sols. Le BAC évalue l'ensemble des demandes associées aux BAC-sig et répartit la bande passante disponible i.e. non réservée par le CAC. L'allocation du BAC est alors propagée vers le DAMA..

Enfin, le DAMA est responsable de l'allocation des slots des trames montantes pour les différents spots. Les allocations sur les trames montantes sont périodiquement envoyées aux UES qui doivent émettre leurs cellules uniquement dans les slots qui leur sont attribués. Le DAMA reçoit des demandes d'allocation de la part du CAC, de la part des stations sols (DAMA-sig de demande de trafic supplémentaire pour les connexions de type VBR) ou enfin de la part des BAC (nouvelle allocation sur la trame descendante).

A titre d'exemple de l'utilisation des fonctionnalités précitées, considérons une demande ouverture de connexion (cf. Figure 2-4) : une station sol utilisateur fait suivre une cellule ATM d'ouverture de connexion (*SETUP*). Le CAC la reçoit, émet une cellule *CALL PROCEEDING* vers l'appelant, puis évalue la demande (CAC logique) ; en cas d'acceptation, la demande est propagée vers le nœud suivant. Lorsqu'en aval les nœuds ont accepté la connexion, une cellule *CONNECT* est propagée jusqu'au CAC qui effectue un calcul de CAC physique puis propage une cellule *CONNECT* en amont. Lorsque la connexion est définitivement acceptée (cellule *ACK_CONNECT* reçue de l'amont) le CAC met à jour la bande passante disponible au niveau du DAMA et du BAC, puis propage l'acceptation de la demande. Le plan de trame du spot correspondant à la nouvelle connexion reflète alors l'acceptation de la connexion.

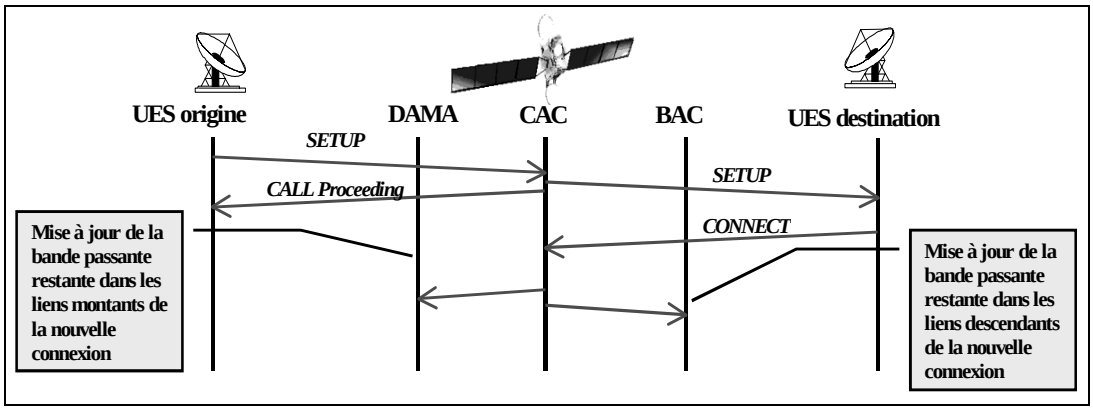


Figure 2-4. Scénario partiel de demande d'ouverture d'une connexion ATM.

L'analyse des fonctionnalités des charges utiles de nouvelle génération permet de mettre en évidence des fonctionnalités qui sont en étroite relation avec les flux utilisateurs. En effet, ces fonctionnalités sont directement dépendantes de la fréquence des ouvertures de connexion, des qualités de service des connexions, du profil du trafic UBR (Bac-sig) et du profil des flux de type VBR (Dama-sig), ce qui nous amène à dire que les modes d'allocation sont directement dépendants de la nature des trafics utilisateurs gérés.

Mise en œuvre des fonctionnalités

L'implantation des fonctionnalités décrites précédemment peut être répartie entre le sol et le bord et peut être réalisée de façon matérielle ou logicielle. La répartition bord / sol est une tâche complexe qui nécessite une étude approfondie de l'interactivité des fonctionnalités. En effet, placer une fonctionnalité au sol diminue l'interactivité (délai de propagation supplémentaire) et peut ainsi engendrer l'expiration de timers au niveau des protocoles (service dégradé). Réciproquement, l'implantation embarquée de la même fonctionnalité accroît l'interactivité mais présente un coût supplémentaire (consommation, masse, etc.). Toutefois, grâce à l'augmentation de la puissance des calculateurs embarqués et au regard de la complexité des fonctionnalités, les futures charges utiles avec commutateur embarqué posséderont un calculateur qui effectuera de façon logicielle un sous-ensemble des fonctionnalités précitées. A ce sujet, des études internes à Alcatel Space montrent la faisabilité d'une implantation logicielle bord des fonctionnalités associées au DAMA et au BAC.

Besoin d'évolutivité des fonctionnalités

Nous avons en effet vu précédemment que les fonctionnalités des OBPs sont intimement liées aux protocoles terrestres et aux données utilisateurs acheminées (profil de trafic). Les algorithmes de ces fonctionnalités sont conçus au regard de simulations effectuées pour un profil de trafic donné. Deux problèmes liés à cette approche peuvent rendre inadaptées ces fonctionnalités :

- la représentativité des simulations sols effectuées avec des outils tels que [NS 2002] ou [OPNET 2002] ;
- les quinze années d'exploitation au regard de l'évolution des protocoles et des profils de trafic. Nous avons vu précédemment que les fonctionnalités de télécommunication sont intimement liées aux flux utilisateurs. Ainsi, l'évolutivité des flux utilisateurs implique une évolutivité des fonctionnalités de télécommunication. Par exemple, outre les trafics gérés dans le projet SAGAM [Roulet 1999], [Combes 2001] propose de prendre en compte le trafic de type voix sur IP (VoIP). Cela se traduit par l'introduction d'un nouveau mode d'allocation qui consiste à allouer des slots fixes sur la trame afin de réduire la gigue. La prise en compte d'un nouveau type de trafic se traduit ainsi par la modification de la fonction d'allocation.

De plus, le besoin d'évolutivité associé au grand nombre d'utilisateurs de tels systèmes (potentiellement plusieurs centaines de milliers de stations sols) nous laisse penser que les

fonctionnalités devront être mises à jour alors que la charge utile est en exploitation : on parle alors de reconfiguration dynamique avec conservation du service offert aux utilisateurs.

2.2.3 Besoins de reconfiguration dynamique des logiciels embarqués

Trois principales motivations justifient l'emploi de techniques de reconfiguration des logiciels du spatial : la correction d'erreurs, l'évolutivité et enfin la réutilisation.

2.2.3.1 *Correction d'erreurs logicielles*

La correction d'erreurs logicielles consiste à rendre un logiciel en cours d'exploitation conforme à ses spécification initiales (correction de bogues).

2.2.3.2 *Évolutivité*

Le besoin d'évolutivité d'un engin spatial est lié à sa durée de vie et à sa mission. En effet, les besoins exprimés dans le cahier des charges sont susceptibles d'évoluer, par exemple dans le cas de missions scientifiques dans lesquelles un ajustement des algorithmes est parfois nécessaire [Folliot 2000] ou dans le cas de charges utiles de télécommunication [Boutry 2000]. Enfin, Les contraintes environnementales subies par un système spatial entraînent des comportements qui n'étaient pas prévus lors du lancement de l'engin. Notamment, l'engin spatial subit de nombreuses pannes matérielles dues aux vibrations, aux contraintes thermiques et aux rayonnements cosmiques [Martelli 1997]. L'impossibilité d'intervention humaine sur le matériel contraint à faire évoluer le logiciel. Par exemple, dans le cas du satellite Artémis qui a été placé sur une mauvaise orbite d'injection, l'agence spatiale européenne souligne le besoin de *« télécharger et de mettre en service à bord d'Artémis des instructions informatique permettant d'appliquer les modes de contrôle non prévus »* [ESA 2001].

2.2.3.3 *Réutilisation*

Lors de certaines missions notamment scientifiques, le satellite peut être réutilisé à d'autres fins une fois sa mission initiale achevée [Folliot 2000].

2.3. Logiciels embarqués sur satellites

2.3.1 Architecture des calculateurs embarqués

Un satellite est composé de plusieurs calculateurs :

- un calculateur plate-forme qui gère l'ensemble des équipements de la plate-forme et notamment, le sous-système de télécommande/télémesure. Ce calculateur est connecté avec les autres calculateurs par un bus temps-réel, typiquement un bus *1553*.
- Des calculateurs charge utile chacun responsable d'un ou plusieurs équipements.

Notons que sur certaines architectures un calculateur unique assure les fonctions relatives à la plate-forme et aux charges utiles.

2.3.2 Environnement matériel

La principale spécificité des environnements matériels du spatial est la limitation de puissance et de mémoire. En effet, l'exposition du matériel aux rayonnements cosmiques contraint les fabricants à durcir les circuits, et par-là même à produire un matériel spécifique et donc à la fois coûteux et en retard technologique de plusieurs années par rapport aux technologies grand public. Si des études sont actuellement réalisées quand à la spatialisation de processeurs commerciaux approchants les 200 Mips par utilisation conjointe de techniques de tolérance aux fautes, la puissance actuelle des processeurs est au mieux de 50 Mips, et plus généralement de 20 Mips (cf. les processeurs ERC-32 [ERC32 1999a] ou DSP 21020 [Suanno 1996]). Quant à la mémoire vive associée au calculateur, elle est généralement de 8 Mo et peut atteindre 32 Mo sur les dernières cartes à base d'ERC-32.

2.3.3 Environnement logiciel

L'environnement logiciel est constitué de mécanismes bas niveau de gestion des tâches applicatives (séquenceur, système d'exploitation) et de l'application. Ces dernières années ont été marquées par le développement de l'utilisation des systèmes d'exploitation temps-réel commerciaux au détriment des séquenceurs ou des systèmes d'exploitation « maison ». L'application en elle-même est généralement développée en C.

Le principe de fonctionnement d'un logiciel spatial est le suivant. Lors du démarrage, un logiciel spécifique dit *de boot* lance des fonctions minimales (par exemple, autotest des équipements) puis démarre le logiciel principal en recopiant le code du système d'exploitation et de l'application en mémoire vive et en invoquant le point d'entrée de l'application. En cas d'erreur critique lors du fonctionnement nominal, l'application passe en mode de survie (fonctions minimales).

Les logiciels charges utiles assurent des fonctions récurrentes à la plupart des missions et des fonctions spécifiques. Les fonctions classiques concernent la gestion des équipements (état de fonctionnement) ainsi que l'émission de rapport sur l'état des équipements et de l'application (*housekeeping*). Les fonctions spécifiques concernent la gestion même des équipements. On trouve des calculateurs en charge de mémoires de masse (lecture, écriture, compression de données), d'antennes (orientation), d'altimètres (données à acquérir) et plus récemment des fonctions liées à la gestion des flux de données utilisateurs (charge utile avec commutation).

En termes de contraintes, les logiciels plate-forme sont dits critiques (catégorie A) en raison des conséquences qu'un dysfonctionnement du logiciel pourrait entraîner sur l'exploitation du système satellite. Les logiciels charges utiles sont considérés comme moins critiques mais possèdent des contraintes temps-réel fortes pour mener à bien leur mission (contraintes à la millisecondes).

2.3.3.1 *Méthodologie de développement*

La méthodologie de développement mise en place chez Alcatel Space est basée sur un cycle UML itératif [Douglas 2001] enrichi par les points suivants :

- la traçabilité des exigences tout au long du cycle de développement [Debus 2001] ;
- la conformité aux standards de codage imposés à la fois par la catégorie du logiciel et par les procédures qualité en vigueur chez Alcatel ;
- la génération automatique de code natif ou cible ;
- l'utilisation de la simulation visuelle (animation des machines à états des classes UML) ;
- une vérification de fonctionnement sur hôte avec un simulateur (simulateur ERC-32 cf.) [ERC-32 1999b], etc.) ou sur cible avec récupération d'informations d'exécution au travers de sondes.

2.3.3.2 *Prise en compte du besoin de reconfigurabilité*

La seule technique actuellement mise en œuvre pour répondre au besoin de reconfigurabilité des logiciels bords satellite consiste en le téléchargement à bord d'un nouveau code applicatif. Lorsqu'un nouveau code est téléchargé, il est placé dans une EEPROM. La mémoire vive applicative est alors vidée, puis le nouveau code est transféré de l'EEPROM en mémoire vive, et enfin le point d'entrée de ce code est invoqué.

Cette technique implique un redémarrage applicatif complet et un arrêt de service conséquent. En effet, le redémarrage applicatif est long car il convient d'attendre que les équipements dirigés par ce logiciel soient dans un état compatible avec l'initialisation de ce logiciel. A ce jour, les techniques mises en oeuvre dans le domaine du spatial ne visent pas à réduire l'arrêt de service dû au redémarrage applicatif mais à diminuer le temps relatif au téléchargement du code et à l'écriture de ce code dans une EEPROM. En effet, [Garrido 1998] souligne que le temps d'écriture sur les EEPROM est conséquent : l'écriture sur les EEPROM se fait par bloc de 8 ko avec 10 millisecondes d'attente entre chaque écriture. [Garrido 1998] propose ainsi une technique à

base d'indirections de procédure permettant de mettre à jour non pas l'ensemble de l'EEPROM mais des sous-blocs de l'EEPROM.

[Stevens 2000] reprend l'idée de mise à jour partielle de l'EEPROM mais propose une technique qui permet de modifier non pas une procédure applicative mais un sous-ensemble du logiciel. Cette technique implique :

- la génération d'un logiciel augmenté, dont les nouvelles parties référencent les anciennes et réciproquement. Pour cela, il convient d'utiliser exactement le même compilateur que celui du logiciel original.
- L'exécution d'un code à bord permettant de mettre à jour les liens au niveau de l'ancien logiciel. Cette exécution ne tient pas compte des contraintes de continuité de service.

La Figure 2-5, extraite de [Stevens 2000], présente une implantation mémoire du logiciel avant et après modification. Elle met en évidence deux types de données en mémoire : des données présentes avant la mise à jour (données mémoires A et B) et les données mémoires ajoutées (C, D et E). Le code C sert à mettre à jour les liens (flèche en pointillés) du logiciel initial (code A) vers le logiciel modifié (code D). A son tour, l'ajout du code D permet de mettre en place automatiquement les liens (flèches pleines) du nouveau code vers le code initial ou vers les données initiales ou enfin vers les nouvelles données.

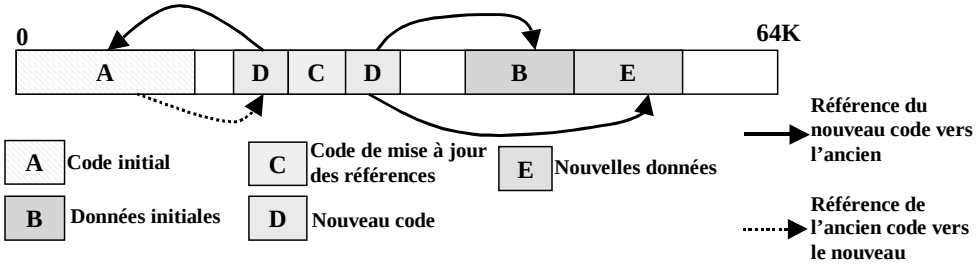


Figure 2-5. Intégration de code applicatif par utilisations de doubles indirections.

Afin de limiter l'investissement humain de production de la rustine logicielle et de simuler l'effet de la nouvelle rustine à bord, [Martelli 1997] propose une plate-forme simulant l'exécution au sol du logiciel bord. Lorsqu'une anomalie à bord est détectée, les traces bord d'exécution sont injectées sur cette plate-forme et analysées. La rustine pour corriger l'erreur est alors produite et son exécution à bord est simulée sur la plate-forme sol.

En résumé, les techniques de reconfigurabilité mises en œuvre dans le domaine du spatial :

- permettent des modifications en profondeur des applications.

Mais elles :

- engendrent un arrêt de service ;
- impliquent une intervention humaine conséquente ;
- nécessitent des outils et compilateurs spécifiques ;
- ne garantissent pas la bonne marche du système après reconfiguration (dans le meilleur des cas, simulation, cf. [Martelli 1997]).

2.4. Conclusion

L'objectif de ce chapitre était de montrer d'une part le besoin de reconfiguration dynamique des fonctionnalités logicielles de télécommunication à bord des satellites et d'autre part, l'absence de mécanismes élaborés et performants de reconfiguration dynamique mis en œuvre dans le spatial. Ce chapitre a montré cela au travers de quatre étapes :

- la description des fonctionnalités des nouvelles charges utiles de télécommunication qui sont implantées à bord et de façon logicielle pour des raisons d'interactivité et de complexité ;

- la mise en évidence du besoin d'évolutivité des fonctionnalités logicielles sans arrêt de service : cela justifie le recours à des mécanismes de reconfiguration dynamique ;
- l'analyse des besoins de reconfiguration des autres logiciels au service de systèmes spatiaux : la correction d'erreurs, l'évolutivité et la réutilisation ;
- la mise en évidence que les environnements logiciels de développement et d'exécution des logiciels du spatial n'ont pas été conçus pour supporter de façon intrinsèque des reconfigurations dynamiques.

Le chapitre suivant propose un panorama critique des approches de reconfiguration dynamique dans les autres domaines d'application et étudie leur adéquation avec notre besoin de reconfigurabilité.

Chapitre 3

Approches de reconfiguration dynamique

Résumé. Le chapitre précédent a montré qu'aucune technique mise en œuvre dans le domaine du spatial ne permet de répondre au besoin de mise à jour, avec continuité de service, des fonctionnalités logicielles de télécommunication embarquées dans les satellites. Fort de ce constat, ce chapitre étudie l'adéquation des contributions en termes de reconfiguration dynamique des autres domaines d'applications avec notre besoin de mise à jour. Après avoir introduit la terminologie, les motivations et les principales caractéristiques relatives à la reconfiguration dynamique, nous présentons les approches de reconfiguration dynamique en fonction des mécanismes supports sous-jacents à ces approches. Ces mécanismes peuvent être implantés dans quatre couches : matérielle (utilisation d'un support matériel redondant), systèmes d'exploitation, middleware (bus logiciels) et enfin applicative (techniques d'indirections et architectures dites à composants). Nous montrons qu'aucun environnement ne gère correctement les contraintes intrinsèques temp-réel et extrinsèques imposées par la reconfiguration dynamique d'applications temps-réel du spatial. Cependant, nous proposons d'utiliser l'approche à composants en raison de son adéquation avec d'autres critères de reconfiguration d'application spatiales (aspect implantation, degré de reconfiguration, etc.).

3.1. Introduction

Le chapitre précédent a mis en évidence le besoin d'évolutivité avec continuité de service des fonctionnalités logicielles des charges utiles de télécommunication. Nous avons aussi montré dans le chapitre précédent que les approches d'évolution logicielle actuellement utilisées dans le domaine du spatial sont inadaptées à la gestion de la continuité de service. L'objectif de ce chapitre est d'explorer les solutions en terme de mise à jour des logiciels dans les autres domaines d'applications et de confronter ces solutions à notre propre besoin. Deux principaux critères permettent d'évaluer ces solutions : leur capacité à gérer la continuité de service requise par nos charges utiles et leur capacité à être mise en œuvre dans les environnements matériels et logiciels du spatial.

Le chapitre est organisé comme suit. Nous définissons dans un premier temps la phase de maintenance des applications (terminologie et mécanismes support). Dans un deuxième temps, nous définissons la terminologie associée à la reconfiguration, puis nous détaillons les motivations d'utilisation de cette technique pour les différents domaines d'application et enfin, nous listons les caractéristiques relatives à la reconfiguration. La troisième partie de ce chapitre entend montrer les limitations des environnements de reconfiguration par rapport à nos besoins. Pour cela, nous présentons nos besoins que nous confrontons par la suite aux approches de reconfiguration dynamique. Enfin, nous concluons ce chapitre.

3.2. Maintenance d'applications : définition et mécanismes supports

3.2.1 Définition de la maintenance

La maintenance logicielle apparaît comme la dernière étape, et néanmoins la plus longue, du cycle de vie classique d'un logiciel. Cette phase commence lorsque la première version du logiciel est livrée au client. Trois raisons fondamentales peuvent justifier le recours à la maintenance logicielle : la maintenance corrective, la maintenance adaptative, et enfin la maintenance perfective.

- Maintenance corrective : elle consiste à modifier un logiciel afin de supprimer les non conformités de ce logiciel par rapport à ses spécifications.
- Maintenance adaptative : elle consiste à modifier un logiciel afin de répondre à de nouveaux besoins.
- Maintenance perfective : elle consiste à améliorer un logiciel, qui rendra les mêmes fonctionnalités, mais qui sera de meilleure qualité du point de vue de certains critères de génie logiciel (par exemple, la modularité, la performance, etc.).

3.2.2 Mécanismes supports à la maintenance

La mise à jour d'un logiciel consiste à remplacer la version *n* d'un logiciel par une nouvelle version *n+1*. La maintenance se divise en deux étapes : tout d'abord, la création de la version *n+1*, et ensuite, le remplacement de la version *n* par la version *n+1*. La création de la version *n+1* nécessite d'effectuer un cycle de développement logiciel réutilisant éventuellement le travail effectué pour la création de la version *n*. Le remplacement de la version *n* par la version *n+1* peut se réaliser au travers de deux mécanismes qui se distinguent par le caractère dynamique ou non de la mise à jour.

Le premier mécanisme est le plus basique. Il consiste à arrêter le logiciel *n*, à intégrer le logiciel *n+1*, à démarrer le logiciel *n+1*, puis à vérifier son bon comportement avant d'offrir ses services aux utilisateurs. Ce mécanisme est qualifié *d'application d'un patch*.

Le deuxième mécanisme consiste à générer la version *n+1* de l'application sans interrompre, pendant la mise à jour, le fonctionnement de la version *n*. De nombreux travaux de recherche s'intéressent aux applications pouvant être mise à jour avec un impact minimal auprès des utilisateurs. Ces applications sont dites *dynamiquement reconfigurables* [Kramer 85].

3.3. La reconfiguration dynamique : définitions, motivations et caractéristiques

3.3.1 Terminologie associée à la reconfiguration dynamique

3.3.1.1 *Définitions*

L'action qui consiste à modifier le code ou les données associées à un logiciel en cours de fonctionnement est appelée *reprogrammation à la volée* ou *modification en cours de fonctionnement*. Lorsque la modification concerne la structure du logiciel (ou configuration), on parle de *reconfiguration dynamique*. Par extension, le terme *reconfiguration dynamique* est associé à tout type de modification logicielle intervenant en cours de fonctionnement. En cas d'arrêt de fonctionnement du logiciel, on parle de *reconfiguration statique*.

Définition 1. Reconfiguration statique

On appelle reconfiguration statique d'un logiciel L le processus qui consiste à stopper totalement le logiciel L (i.e. la mémoire d'exécution de L est complètement réinitialisée), à effectuer des modifications sur le logiciel L, puis à redémarrer L [Kramer 85].

Définition 2. Reconfiguration dynamique

On appelle reconfiguration dynamique d'un logiciel L le processus qui consiste à modifier ce logiciel sans le stopper totalement.

On définit trois types de reconfigurations dynamiques : la reconfiguration dynamique sans garantie de continuité de service, la reconfiguration dynamique avec continuité partielle de service, et enfin la reconfiguration dynamique avec stricte continuité de service. Par la suite, on note : P_1 (respectivement P_2) l'ensemble des prédicats décrivant le service utilisateur *S* offert par le logiciel avant (resp. après) reconfiguration. Soit t_1 et t_2 respectivement les instants de début et de fin de reconfiguration.

Définition 3. Reconfiguration dynamique sans garantie de continuité de service

On appelle reconfiguration dynamique sans garantie de continuité de service une reconfiguration dynamique pour laquelle les deux assertions suivantes sont vraies :

1. $\forall t < t_1, \forall p \in P_1, p = vrai$
2. $\forall t > t_2, \forall p \in P_2, p = vrai$

La Figure 3-1 décrit les propriétés qui doivent être vraies en fonction du temps.

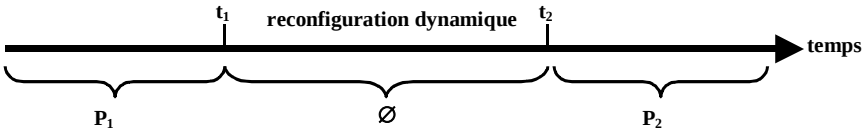


Figure 3-1. Chronogramme d'une reconfiguration dynamique sans garantie de continuité de service.

L'intervalle $[t_1, t_2]$ définit la période d'interruption de service.

Définition 4. Reconfiguration dynamique avec continuité partielle de service

On appelle reconfiguration dynamique avec continuité partielle de service une reconfiguration dynamique pour laquelle les trois assertions suivantes sont vraies :

1. $\forall t < t_1, \forall p \in P_1, p = vrai$
2. $\forall t > t_2, \forall p \in P_2, p = vrai$
3. $\forall t, \forall p \in P_1, (p_{21} \wedge p_{22} \wedge ... \wedge p_{2n} \Rightarrow p) \Rightarrow p \in P$ (seuls les services communs à P_1 et P_2 ont leur continuité assurée).

Ainsi, une reconfiguration dynamique sans arrêt de service du point de vue utilisateur se caractérise comme une reconfiguration dynamique pendant laquelle les règles qui régissent le service offert aux utilisateurs et qui ne sont pas affectées par la reconfiguration dynamique demeurent vraies (cf. Figure 3-2).

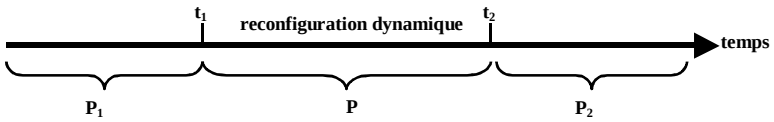


Figure 3-2. Chronogramme d'une reconfiguration dynamique avec continuité partielle de service.

Définition 5. Reconfiguration dynamique avec stricte continuité de service

Soit $P_2 = \{p_{21}, p_{22}, ..., p_{2n}\}$.

On appelle reconfiguration dynamique avec stricte continuité de service une reconfiguration dynamique pour laquelle les trois assertions suivantes sont vraies :

1. $\forall t < t2, \forall p \in P_1, p = vrai$
2. $\forall t > t2, \forall p \in P_2, p = vrai$
3. $\forall p \in P_1, p_{21} \wedge p_{22} \wedge ... \wedge p_{2n} \Rightarrow p$

Ainsi, une reconfiguration dynamique avec stricte continuité de service se caractérise comme une reconfiguration dynamique pendant laquelle les règles, qui régissent le service offert aux utilisateurs demeurent vraies pendant et après la reconfiguration dynamique (cf. Figure 3-3).

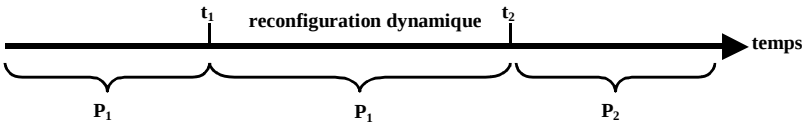


Figure 3-3. Chronogramme d'une reconfiguration dynamique avec stricte continuité de service.

3.3.1.2 *Notion d'opérations sur un logiciel*

Les différents besoins en termes de reconfiguration de logiciels embarqués sont l'ajout de fonctionnalités, la suppression de fonctionnalités et enfin la modification de fonctionnalités.

Soit P1 l'ensemble des prédicats définissant le logiciel avant reconfiguration, et P2, l'ensemble des prédicats définissant le service offert par le logiciel après reconfiguration, avec $P_1 = \{p_{11}, p_{12}, ..., p_{1n_1}\}$ et $P_2 = \{p_{21}, p_{22}, ..., p_{2n_2}\}$.

L'ajout de fonctionnalités doit permettre au logiciel modifié d'offrir plus de fonctionnalités que précédemment. Cela se traduit par $\forall p \in P_1, p_{21} \wedge p_{22} \wedge ... \wedge p_{2n_2} \Rightarrow p$.

La suppression de fonctionnalités consiste à rendre inaccessible après reconfiguration au moins un service de P1. Ces services supprimés ne sont ainsi pas dans P2. Cela se traduit par :

$\exists p_1 \in P_1 / p_1 \notin P_2$.

Enfin, la modification de fonctionnalités se traduit, du point de vue description du service offert, comme le retrait et l'ajout de fonctionnalités.

3.3.1.3 *Cycle de fonctionnement d'un logiciel et processus de reconfiguration dynamique*

Le cycle de fonctionnement d'un logiciel est l'ensemble du temps compris entre la date de la première exécution du logiciel et sa date de dernière exécution (cf. Figure 3-4).

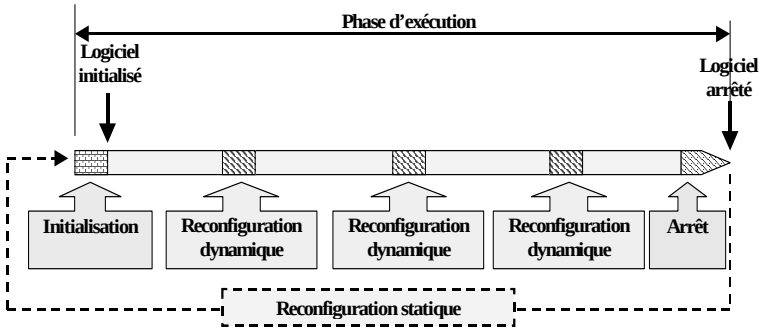


Figure 3-4. Cycle de fonctionnement d'un logiciel reconfigurable.

Une phase d'exécution d'un logiciel est elle même composée de sous-phases : une phase d'initialisation du logiciel, des phases intermédiaires dites de fonctionnement nominal, des phases pendant lesquelles le logiciel est dit en phase de reconfiguration dynamique, et enfin, la phase d'arrêt du logiciel. Certaines de ces phases ont été identifiées dans [Pellegrini 1999].

Par la suite, nous appelons *processus de reconfiguration dynamique* l'ensemble des étapes qui permettent la reconfiguration d'une application en cours de fonctionnement.

3.3.2 Motivations des mécanismes de mise à jour logicielle sans arrêt de service

La plupart des applications logicielles supportent un arrêt de service lors de leur mise à jour à l'instar des applications grand public dont la phase de maintenance s'effectue en général avec arrêt de l'application, et même parfois avec redémarrage du système d'exploitation sous-jacent. De

nombreux types d'application ne peuvent se satisfaire d'une telle approche et nécessitent par là même des mécanismes garantissant la conservation d'un service utilisateur pendant la reprogrammation. Par la suite, nous énumérons ces domaines d'application et leurs motivations respectives pour de tels mécanismes.

Les motivations du domaine du spatial ont été décrites dans le chapitre précédent. Pour mémoire, ces trois motivations principales sont : la *correction d'erreur logicielle*, *l'évolutivité* de la mission et donc du logiciel servant cette mission, et enfin la *réutilisation* d'un logiciel charge utile afin de conférer une nouvelle mission à l'engin spatial correspondant.

De nombreux autres domaines d'applications nécessitent la mise en place de techniques permettant la reprogrammation à la volée. Le domaine des télécommunications terrestres est l'un des précurseurs dans ce domaine : le service utilisateur doit être dans le cas le plus défavorable, faiblement dégradé [Boute 1981]. En effet, les équipements de télécommunication, depuis leur automatisation, gèrent plusieurs milliers de communications simultanément. De plus, ils sont confrontés à des architectures de communication qui évoluent très rapidement. Ainsi, depuis le début des années 1980, des techniques permettant de mettre à jour de façon transparente pour les utilisateurs les équipements réseaux, tant du point de vue matériel que logiciels ont été proposées [Boute 1981]. Ces techniques ont par la suite été enrichies [Engram 1983][Segal 1989][Segal 1993]. Actuellement, outre les mécanismes permettant la mise à jour pour des raisons de *maintenance,* des contributions s'intéressent à la personnalisation du traitement effectué par les équipements réseaux pour chaque flux de données. Ce paradigme, appelé *réseau actif*, vise à modifier le comportement habituel des nœuds des réseaux « stockage, transmission » en « stockage, exécution personnalisée, transmission » [Tennenhouse 1997][Chen 2000][Patel 2001].

[Stewart 1996] met en évidence le besoin de continuité de service de certaines applications temps-réels embarquées. Les contributions de ce domaine se concentrent sur les applications ayant des contraintes fortes en termes de temps-réel [Pasetti 1999][Stewart 1996] ou en termes de criticité [Garrido 1998].

Enfin, la principale motivation de reconfiguration des applications distribuées concerne le redéploiement applicatif à des fins de *répartition de la charge* des applications, de *tolérance aux fautes* [Assis 1997][Shrivastava 1998], d'a*dministration* optimisée et autonome des réseaux et des bases de données [Hartman 1996][Fuggeta 1998] et enfin de *prototypage plus rapide* (flexibilité logicielle accrue) [Assis 1997]. Si les besoins de continuité de service dans ce type d'applications sont généralement moins forts que dans les systèmes temps-réel, le redéploiement doit par contre se réaliser avec conservation de la consistance logique applicative [Kramer 1985].

3.3.3 Caractéristiques générales

Les techniques mises en œuvre afin de reconfigurer dynamiquement les applications sont très dépendantes du domaine considéré [Segal 1993]. Toutefois, il est possible d'extraire des caractéristiques d'évaluation des environnements de reconfiguration dynamique.

3.3.3.1 *Degré de reconfiguration*

Le degré de reconfiguration se caractérise par la capacité de l'environnement de reconfiguration dynamique à modifier en profondeur l'application. [Segal 1993] propose, en ce qui concerne les architectures logicielles modulaires, de qualifier les modifications inter-modules de superficielles, et les modifications intra-modules, par exemple les changements d'interface d'un module, de modification profonde. [Okamoto 1994] note par ailleurs que la reconfiguration dynamique et totale d'une application paraît difficile.

3.3.3.2 *Transparence*

Le processus de reconfiguration dynamique doit être le plus transparent possible du point de vue de l'utilisateur de l'application, mais aussi du point de vue du programmeur. La transparence du point de vue utilisateur est équivalente à la *préservation de service au regard d'un contrat*. A noter qu'un

contrat peut supposer un arrêt de service, à l'instar des offres de bande passante des opérateurs de télécommunication qui incluent une interruption de service (notion de disponibilité à x %). Du point de vue programmeur, la transparence de la reconfiguration se caractérise par l'impact de cette reconfigurabilité sur le développement de l'application [Kramer 1985][Segal 1993]. Par exemple la programmation explicite de points de reconfiguration (cf. [Purtilo 1994]) nuit à la transparence de la reconfiguration du point de vue développeur. Ainsi, si l'on note P_c, le temps de programmation d'un logiciel L, et P_r, le temps de mise en œuvre au sein de L des mécanismes permettant de rendre L intrinsèquement reconfigurable, la transparence de la reconfiguration du point de vue programmation est d'autant plus grande que le rapport P_c/P_r est grand.

3.3.3.3 *Manipulation humaine*

Une manipulation humaine est généralement nécessaire afin d'effectuer une modification dynamique sur une application, mais doit être minimisée [Okamoto 1994]. La modification d'une application est initiée par la demande d'exécution d'un *script de reconfiguration*, qui décrit dans un langage spécifique l'ensemble des opérations à effectuer sur l'application à reconfigurer [Kramer 1985][Purtilo 1991][Okamoto 1994][Oreizy 1998]. Ce script est généré manuellement ou automatiquement, à partir d'une description de l'application, qui peut aller d'une simple trace mémoire à une description architecturale applicative décrite dans un langage de type ADL [Magee 1984][Medvidovic 2000]. Le langage associé au script doit proposer un haut niveau d'abstraction [Oreizy 1998], ainsi qu'une interface simple [Okamoto 1994]. De nombreuses contributions mentionnent le besoin de pouvoir effectuer la mise à jour à distance, notamment dans le domaine des télécommunications [Okamoto 1994] et des systèmes embarqués [Stewart 1997]. Certains environnements vont encore plus loin et ont comme objectifs d'autoriser les mises à jour d'applications distribuées [Segal 1993]. La reconfiguration en elle-même doit pouvoir être contrôlée (contrôle d'exécution et de synchronisation [Stockenberg 1993]), et un rapport de configuration doit être envoyé à la fin du processus [Frieder 1989][Purtilo 1991]. Enfin, l'ensemble du processus peut-être piloté à partir d'un gestionnaire de l'application, à l'exemple du *manager* introduit dans [Okamoto 1994].

3.3.3.4 *Couches impliquées*

Les mécanismes supports à la reconfiguration dynamique doivent de préférence être implantés dans une couche du système qui soit la plus haute possible, pour des raisons de portabilité [Shrivastava 1998]. Notamment, il apparaît important de ne pas s'appuyer sur des couches matérielles [Segal 1993][Stokenberg 1993], sauf éventuellement en ce qui concerne les systèmes embarqués, pour des raisons de performance [Stewart 1996]. Enfin, il est de plus préférable pour un environnement de reconfiguration dynamique de n'offrir aucune contrainte en termes de langages de développement [Segal 1993][Shrivastava 1998].

3.3.3.5 *Préservation des contraintes applicatives*

Deux types de contraintes applicatives doivent être prises en compte lors des mises à jour : les contraintes intrinsèques et extrinsèques.

Contraintes intrinsèques

Pour [Kramer 1985], les contraintes intrinsèques se regroupent sous le terme de consistance logique. La consistance logique concerne (1) les ressources applicatives, par exemple les ressources en termes de mémoire, mais aussi en termes de ressources temps-réel et (2) la consistance logique d'interconnexion, qui se traduit, dans les applications développées sous forme de procédures, par une consistance des références entre procédures (notamment, format et typage des paramètres), et dans les applications développées selon une approche à composants, par une consistance logique dans l'interconnexion entre les ports de communication des modules.

Pour notre part, il nous paraît judicieux de séparer les contraintes intrinsèques logiques et temps-réel. Nous regroupons donc sous le terme *contrainte logique intrinsèque* les contraintes de

ressources logiques applicatives (par ex., mémoire) et les contraintes syntaxiques (par ex., format des données échangées entre modules). Nous y ajoutons aussi les contraintes de bonnes propriétés comportementales (absence d'état puits). Les contraintes intrinsèques temps-réel concernent toutes les contraintes logicielles internes qui ne peuvent être explicitées sans recours à des opérateurs temporels. Par exemple, la périodicité d'une tâche est une contrainte intrinsèque temporelle.

Nous notons par la suite P_{il} et P_{it}, l'ensemble des prédicats relatifs aux contraintes intrinsèques respectivement logiques et temporelles d'une application.

Le respect des contraintes intrinsèques applicatives se traduit par :

$\forall t, \forall p_l \in P_{il}, \forall p_t \in C_{it}, p_l \wedge p_t = vrai$

Contraintes extrinsèques

La préservation du service se distingue de la préservation des propriétés internes applicatives dans la mesure où il est possible de générer une application dont certaines contraintes logiques internes sont erronées tout en offrant un bon service utilisateur, et réciproquement, il est tout à fait possible de conserver les propriétés internes au logiciel (propriétés temps-réel par exemple), tout en ne respectant pas la continuité de service. Les contraintes extrinsèques sont souvent mentionnées par les contributions, notamment dans [Frieder 1989], mais pour la plupart des contributions, assurer la continuité de service se borne à assurer les contraintes intrinsèques regroupées sous le terme de *consistance logique* [Kramer 1985][Segal 1993][Shrivastava 1998][Palma 1999].

Nous représentons une contrainte extrinsèque *S* (c'est-à-dire un service devant être rendu, cf. 3.3.1.1) comme un ensemble de prédicats de contraintes logiques P_{sl}, de contraintes temporelles P_{st}, et de contraintes d'environnement P_{se}. P_{sl} représente des contraintes logiques devant être respectées par l'application (par exemple, tel module logiciel est interconnecté avec tel autre), C_{st} représente des contraintes temporelles, par exemple, que tel module logiciel doit traiter telle requête en moins de *t* unités de temps, et enfin C_{se} représente les contraintes devant être rendues par l'environnement au logiciel, par exemple, que tel équipement répond à telle requête en moins de *t* unités de temps.

De plus, certaines contraintes extrinsèques impliquent des contraintes intrinsèques. Par exemple, si une contrainte extrinsèque impose que deux modules soient interconnectés, cela implique que les modules soient correctement interconnectés, ce qui est une contrainte intrinsèque. Ainsi, la préservation d'un service se traduit comme la préservation des propriétés extrinsèques logiques, temporelles et d'environnement de ce service, et de toutes les contraintes intrinsèques qui sont impliquées par les propriétés extrinsèques de ce service.

Soit $S = P_{sl} \cup P_{st} \cup P_{se}$. Soit P_{il} et P_{it}, respectivement les contraintes intrinsèques respectivement logiques et temps-réel d'une application fournissant le service S. Soit $P_l \subset P_{il} / (P_{sl} \cup P_{st} \cup P_{se}) \Rightarrow P_l$. De même, soit $P_t \subset P_{it} / (P_{sl} \cup P_{st} \cup P_{se}) \Rightarrow P_t$. La préservation de tous les services de S pendant une reconfiguration implique que :

$\forall t, \forall p_{sl} \in P_{sl}, \forall p_{st} \in P_{st}, \forall p_{se} \in P_{se}, \forall p_{sil} \in P_l, \forall p_{sit} \in P_t$, on a :

$p_{sl} \wedge p_{st} \wedge p_{se} \wedge p_{sil} \wedge p_{sit} = vrai$

Gestion des contraintes

La préservation des deux types de contraintes nécessite :

- d'effectuer la reconfiguration, à un instant précis et dans un état applicatif particulier dans lequel les opérations de reconfiguration peuvent être effectuées en respectant les contraintes intrinsèques [Segal 1993]. Cet instant de reconfiguration est appelé *point de reconfiguration*. Ce point de reconfiguration, à l'instar de l'environnement Polylith [Purtilo 1994], est mis en œuvre grâce à la définition d'opérations de reconfiguration permettant de requérir puis d'attendre qu'un sous-ensemble du logiciel rejoigne son point de reconfiguration.

- Une atomicité du processus de modification : l'ensemble de la reconfiguration dynamique doit être mené à son terme i.e. toutes les opérations correspondant à une reconfiguration doivent être appliquées avec succès au logiciel à reconfigurer. Si tel n'est pas le cas, le processus doit être annulé [Shrivastava 1998]. [Okamoto 1994] souligne de plus le besoin de retour en arrière en cas de risque de perturbation du service.

3.4. Analyse des contributions

3.4.1 Introduction

Dans cette section, après avoir listé nos besoins en termes de reconfiguration dynamique selon les caractéristiques précitées, nous confrontons les travaux précurseurs sur la reconfiguration dynamique par rapport à ces besoins. Ces travaux s'appuient sur deux approches : une approche matérielle et une approche logicielle. L'approche matérielle consiste par exemple à offrir un mécanisme de *redondance* permettant la mise à jour logicielle sur un équipement qui ne soit pas en cours d'utilisation. L'approche logicielle se divise en trois sous-ensembles, suivant que le support des mécanismes de mise à jour est fourni par le système d'exploitation, par un mécanisme implanté entre le système d'exploitation et l'application (bus logiciel par exemple), ou par des mécanismes de niveau applicatif.

3.4.2 Besoin de reconfiguration dynamique

Nous énumérons dans ce paragraphe les différents besoins concernant la reconfiguration d'applications temps-réel embarquées du spatial. Ces besoins sont exprimés au regard des caractéristiques de la reconfiguration dynamique (paragraphe 3.3.3).

- *Degré de reconfiguration*. En raison de la diversité des logiciels des environnements spatiaux (protocoles de télécommunication, logiciels scientifiques, etc.) et donc de la nature des modifications, nous pensons qu'un haut degré de reconfiguration est souhaitable : la solution adoptée doit autoriser la modification de fonctionnalités majeures et non la simple personnalisation d'un service à l'instar des réseaux actifs.
- *Transparence de la reconfiguration*. Du point de vue programmeur, elle doit être élevée. En effet, la méthodologie logicielle proposée ne doit pas remettre profondément en cause le cycle de développement actuel (cf. paragraphe 2.3.3.1) : notamment en termes de langages de conception, d'implémentation et d'outils supports.
- *Manipulation humaine*. Idéalement, le script de reconfiguration dynamique doit pouvoir être généré de façon automatique par l'équipe d'ingénieurs responsable de la maintenance du logiciel, puis être transmis au centre de contrôle du satellite. Un opérateur doit alors pouvoir appliquer ce script à bord et obtenir le rapport d'exécution à partir d'une interface de haut niveau.
- *Couches impliquées*. La solution doit être implantable à un coût modéré dans les environnements spatiaux. Ainsi, la solution doit être peu gourmande en termes de ressources (CPU, mémoire, alimentation, poids) en raison des contraintes des environnements spatiaux (cf. paragraphes 2.3.2).
- *Préservation des contraintes applicatives*. Le besoin de gestion des contraintes applicatives est très fort, en raison notamment des besoins de continuité de service (cf. paragraphe 2.2.2.2) et de la potentielle criticité du logiciel considéré (cf paragraphe 2.3.3). La solution proposée doit donner des garanties formelles du respect des contraintes intrinsèques et extrinsèques. Les contraintes intrinsèques englobent tout particulièrement les contraintes de ressources qui sont partiellement spartiates dans les environnements satellites. Les contraintes extrinsèques concernent en priorité les

services dit critiques devant être rendus à l'environnement et dans un deuxième temps les contraintes des services utilisateur.
Par la suite, nous confrontons les travaux du domaine avec les cinq besoins enumérés.

3.4.3 Solutions matérielles

La principale approche repose sur la *redondance* matérielle qui consiste à offrir un matériel supplémentaire prenant le relais du matériel nominal pendant la mise à jour [Rey 1986]. Une fois la mise à jour terminée, le matériel nominal reprend alors la charge de l'application. Cette approche présente deux limites. Premièrement, la technique du basculement d'un matériel à un autre nécessite une gestion en partie logicielle de la cohérence des données entre les deux calculateurs. [Segal 1989] propose pour cela de définir des points de reconfiguration qui assurent le basculement d'un matériel à un autre en garantissant au maximum la continuité de service et qui minimisent le temps d'exécution des procédures logicielles de recopie d'état d'un calculateur à l'autre. Mais les difficultés de gestion de cohérence d'état sont encore plus critiques dans le cas de mises à jour distribuées, ce qui conduit [Frieder 1989] à affirmer que l'approche par redondance n'est pas adaptée aux équipements de télécommunication. Par là même, des solutions purement logicielles sont proposées [Okamoto 1994]. Deuxièmement, le dédoublement matériel est très coûteux, ce qui limite son utilisation à des mises à jour logicielles de systèmes nécessitant une redondance matérielle pour des raisons de fiabilité (tolérance aux fautes) d'où son emploi pour les équipements de télécommunication [Engram 1983][Segal 1989]. Pour des questions d'encombrement, de masse et de consommation, l'idée de redondance matérielle n'est pas transposable aux environnements spatiaux.

Une approche plus récente consiste à implanter les fonctionnalités sur des circuits électroniques reprogrammables dits FPGA (*Field Programmable Gate Array*), notamment dans le spatial pour implanter les techniques de modulation (*Software Radio*, cf. [Kenington 1997]). Toutefois, ces circuits ne sont reprogrammables qu'en totalité ce qui rend les reconfigurations statiques. De plus, le temps de reprogrammation est relativement important (plusieurs dizaines de millisecondes).

3.4.4 Solutions logicielles

[Segal 1993] définit quatre types de solutions logicielles permettant la modification d'applications. (1) Les changements de type abstrait concernent la modification des types de données manipulées par les applications. (2) Les changements de serveur dans les systèmes de reconfiguration implémentant le paradigme client/serveur. (3) Les changements dans les systèmes dont la communication entre entité logicielle est réalisée par passage de message. Enfin, (4), les changements de procédure dans les applications programmées dans des langages fonctionnels. Ces quatre types de changement sont mis en œuvre soit au niveau système d'exploitation, soit au niveau applicatif, soit enfin dans des environnements intermédiaires entre l'application et le système d'exploitation. Ainsi, l'approche de [Segal 1993] consiste à différentier les techniques de reconfiguration selon la structuration logicielle de l'applicatif. Pour notre part, il nous paraît plus intéressant, pour des raisons de clarté de mise en œuvre, de classifier les techniques et les environnements de reconfiguration dynamique selon la couche logicielle incriminée, à savoir la couche du système d'exploitation, la couche intermédiaire, et enfin la couche applicative.

3.4.5 Couche systèmes d'exploitation

De par leur couplage fort avec le matériel, les systèmes d'exploitation se positionnent comme des supports à des mécanismes de bas niveau, à des mécanismes nécessitant des performances élevées, et enfin à des mécanismes visant à la gestion des ressources systèmes (mémoire partagée par exemple) et matérielles.

Les systèmes d'exploitation les plus usuels utilisent la technique des librairies dynamiques [Franz 1997] afin de diminuer la trace des données exécutables. Ces dernières comportent alors des liens vers ces librairies. Ces liens sont créés lors de la compilation des données ou lors de l'exécution de ces données. Toutes les applications référençant une librairie modifiée bénéficient de la mise à jour. L'arrêt partiel de service lors des mises à jour des librairies et le faible degré de reconfiguration de cette technique (modification limitée au code des librairies) écartent l'utilisation de cette technique pour les applications spatiales.

Le deuxième mécanisme support des systèmes d'exploitation concerne les mécanismes relatifs aux tâches et aux processus. Par exemple, une technique consiste à démarrer une tâche d'une application, et à la synchroniser avec la fin d'une tâche qu'elle doit remplacer. Certains systèmes d'exploitation temps-réel proposent en outre des mécanismes permettant de contrôler les flux d'exécution temps-réel assez finement du point de vue démarrage et contraintes temporelles. Ainsi, certaines sondes de la NASA ont été reconfigurées par utilisation des mécanismes de gestion de tâche. Mais l'approche ne paraît pas assez générale et globale pour satisfaire les contraintes de la mise à jour de services de télécommunication.

Enfin, le domaine des réseaux actifs utilise massivement des mécanismes des systèmes d'exploitation. En effet, si l'intégration logicielle du code utilisateur ne pose aucun problème par exécution d'un nouveau processus en mode utilisateur [Chen 2000] ou par un mécanisme de *plug-in*, elle soulève des problèmes en termes d'équité et de disponibilité de ressources qui sont résolus par des mécanismes propriétaires de niveau système d'exploitation [Neumann 2001][Yan 2001]. Ces mécanismes concernent les ressources CPU (performance), mémoire, ou encore les ressources physiques du routeur telles que les buffers d'émission [Calvert 1998][Patel 2001]. Mais les approches proposées ne font que garantir des ressources à un code et non le service qui pourra être rendu par un code accepté (gestion des contraintes intrinsèques du code et non des contraintes extrinsèques). De plus, la reconfiguration se limite à une particularisation d'un traitement de niveau applicatif, sans refonte en profondeur de l'application globale du routeur (degré de reconfiguration faible). Cette limitation empêche la transposition à un calculateur embarqué sur satellite potentiellement reconfiguré sur des fonctionnalités majeures.

3.4.6 Couche Middleware

Une couche logicielle de type middleware s'insère entre la couche applicative et la couche système, et offre des services de haut niveau permettant aux applications de s'abstraire, pour certains services, du système d'exploitation sous-jacent. Ces services sont de deux types. Premièrement, des services de communication assurant la *transparence de la localisation*. Le middleware est alors qualifié de *bus logiciel*. Dans ce sens, l'OMG a définit une architecture standard, CORBA [OMG 1996], offrant des services de communication basés sur le paradigme client/serveur. Son objectif est d'offrir une unification des services permettant aux applications distribuées orientées objets d'interagir aux travers d'objets hétérogènes. Deuxièmement, des services de migration permettant le déplacement physique d'entités applicatives.

De nombreuses contributions, principalement du domaine des applications distribuées, s'appuient sur des couches de type middleware à des fins de reconfiguration d'application (notamment pour faciliter les redéploiements applicatifs). Certaines contributions définissent leur bus logiciel [Purtilo 1994] alors que d'autres proposent des extensions à l'environnement CORBA [Pellegrini 1999].

Le bus logiciel *Polylith* [Purtilo 1994] propose des services de communication et de reconfiguration dynamique. La déclaration de la structure applicative en termes de modules, d'interfaces des modules et de déploiement de ces modules est réalisée grâce à un langage d'interconnexion (MIL, cf. [Remer 1976]). Une fois l'application démarrée, le bus logiciel prend en charge la réception et l'exécution des ordres de reconfiguration dynamique, avec gestion de la consistance applicative. Cette gestion repose notamment sur l'isolation des canaux de communication des modules impliqués par les ordres de reconfiguration. Le regroupement, au sein

d'une même entité, des fonctionnalités en termes de communication et de reconfiguration facilite la préservation des propriétés internes [Hofmeister 1993].

[Pellegrini 1999] propose d'étendre l'environnement CORBA par des services de mobilité qui sont utilisés à des fins de reconfiguration dynamique. Le processus de changement de site d'exécution d'un objet *o1* est classique : encodage de l'état de *o1*, création de *o2* (même type que *o1*) sur le nouveau site, recopie de l'état de *o1* dans *o2* puis destruction de *o1*. Une des difficultés réside dans la gestion des références vers l'objet redéployé. En effet, les appels à l'ancien objet doivent être empêchés dès le que le processus de migration a débuté (encodage de l'état). De même, les références vers le nouvel objet doivent être mises à jour uniquement une fois le nouvel objet correctement créé (création et recopie de l'état). [Pellegrini 1999] propose pour cela d'étendre d'une part le service de noms, et d'autre part les mécanismes de communication entre objets, en introduisant le concept de *stub* et *skeleton* étendus (cf. Figure 3-5). Le rôle du *stub étendu* est de détecter les migrations d'objets lors des invocations. En cas d'erreurs lors d'une invocation (défaut d'objets), le service de noms du bus identifie de façon transparente le nouveau site d'exécution de l'objet. De même, le *skeleton étendu* retourne un message d'erreur si l'objet invoqué est en cours de reconfiguration.

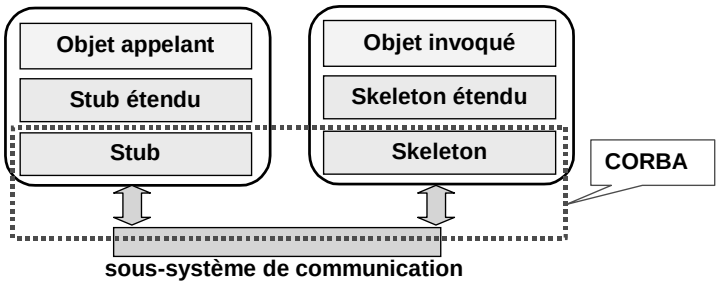


Figure 3-5. Intégration de services étendus dans le bus logiciel CORBA.

En conclusion, les couches middleware offrent des services indépendants des systèmes d'exploitation et des applications développées. Des travaux de recherche ont de plus montré la possibilité de leur adjoindre des services de reconfiguration dynamique capables de gérer la consistance logique applicative. La limite de cette approche réside d'une part dans la mise en œuvre (aspects mémoire et performance) et d'autre part, dans l'absence de gestion explicite de contraintes temps-réel et de continuité de service.

3.4.7 Couche applicative : le patch logiciel

Deux mécanismes fondamentaux ont été proposés au niveau applicatif. Le premier concerne la possibilité de modifier la mémoire de l'application (technique du patch). Le deuxième consiste en la définition d'une architecture logicielle spécifique qui, une fois déployée, peut être modifiée par le biais d'opérateurs. A noter que dans les deux cas, ces mécanismes peuvent être épaulés par des mécanismes des couches sous-jacentes (matériel, système d'exploitation, bus logiciel). Nous examinons dans un premier temps le mécanisme du patch logiciel.

Le patch de mémoire de masse consiste à modifier le code applicatif stocké sur cette mémoire (disque dur, EEPROM, etc.) [Garrido 1998][Stevens 2000]. Cette modification peut être réalisée indépendamment du fonctionnement de l'application tant que celle-ci n'accède pas à ses données de mémoire de masse. Comme nous l'avons vu dans le paragraphe 2.3.3.2, cette technique, qui autorise tout type de modification sur le logiciel, présente deux désavantages majeurs : une intervention humaine conséquente et un arrêt de service. Le patch de la mémoire vive applicative entend résoudre le problème de la discontinuité de service inhérente au patch de mémoire morte.

Le domaine des télécommunications est un des précurseurs de ces techniques de patch de mémoire applicative. Dans un premier temps, les contributions concernent la gestion de la consistance logique applicative par évolution progressive des anciens services vers les nouveaux

services : ces derniers prennent en charge toutes les nouvelles communications utilisateurs [Boute 1981]. A l'instar d'autres contributions, [Boute 1981] propose de réaliser la mise à jour par évolution progressive en introduisant, pour chaque type abstrait modifié, un type abstrait équivalent : les deux types évoluent en parallèle, jusqu'à ce que le nouveau type prenne totalement la relève. [Engram 1983] complète cette technique par la notion d'état de passage d'une configuration à une autre « clear state », et le support de la redondance matérielle synchronisée avec la mise à jour. La notion de point de reconfiguration permet de passer directement d'une configuration à une autre, rendant obsolète la technique de l'évolution progressive. Toutefois, [Engram 1983] reconfigure les différents services applicatifs les uns après les autres, et non la configuration totale.

Par la suite, l'environnement PODUS (Procedure-Oriented Dynamic Updating System) [Frieder 1989][Segal 1993] propose un mécanisme de remplacement d'une procédure par une autre pour les applications développées avec un langage fonctionnel et selon une approche descendante. La première étape de la modification consiste en l'identification des dépendances entre procédures car les procédures modifiées ne doivent pas être en cours d'exécution. Si les dépendances syntaxiques peuvent être détectées par analyse du code applicatif et de la pile d'appel, les dépendances sémantiques c'est-à-dire les interactions non détectables par une analyse syntaxique mais imposées par le bon fonctionnement applicatif doivent être nécessairement décrites par le programmeur. A partir de l'analyse syntaxique et sémantique, les auteurs proposent un algorithme qui garantit la cohérence logique applicative lors de la mise à jour. La deuxième étape consiste à modifier les anciennes procédures par des nouvelles. Pour cela, les auteurs proposent l'introduction d'*interprocédures* (cf. Figure 3-6) qui interceptent les appels à l'ancienne procédure, et appellent à la place, en modifiant le cas échéant le nombre d'arguments, la nouvelle procédure. De même, le concept de *mprocédure* est introduit. Leur but est de convertir les arguments d'appel d'une ancienne procédure vers une nouvelle. Les limites de l'approche concernent l'investissement du programmeur (déclaration des dépendances sémantiques), l'impossibilité de gérer des applications multitâches et la prise en compte du service utilisateur uniquement au travers de la consistance logique applicative.

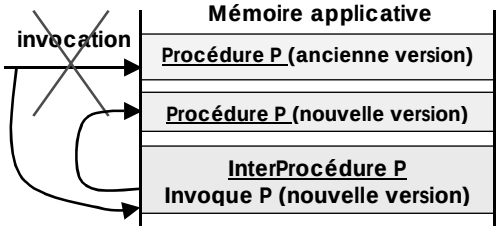


Figure 3-6. Rôle des interprocédures dans [Ségal 1993].

En conclusion, les approches par patch logiciel de mémoire vive :

- ont un faible degré de reconfigurabilité (modification de procédure) tout particulièrement dans le cas d'applications multitâches ;
- ne gèrent la continuité de service qu'au travers du maintien de la consistance logique ;
- impliquent une intervention humaine conséquente (dépendances sémantiques, création du patch).

3.4.8 Couche applicative : Architectures à composants

La préservation de la consistance logique applicative lors des reconfigurations est une des raisons qui ont poussé à l'introduction d'*architectures à composants*. [Stewart 1996] souligne en outre que ce type d'architecture permet de répondre au mieux aux critères de génie logiciel tels que la rapidité du cycle de développement, la flexibilité et la réutilisabilité.

Nous analysons par la suite des environnements à architecture à composants, et plus particulièrement les environnements CONIC [Kramer 1985], Polylith [Purtilo 1994], Argus [Liskov 1985], Port-Based Objet [Stewart 1997], l'environnement proposé par [Cailliau 2001] et enfin les environnements à code mobile [Fuggetta 1998][Keramane 1998]]['Tagant 1998][Ismail 1998].

3.4.8.1 *Eléments architecturaux*

[Oreizy 1998] définit un *composant* comme une boîte noire qui encapsule des fonctionnalités. Un composant possède un état interne manipulé par plusieurs flots d'exécutions distincts.

Un composant ne communique avec d'autres composants qu'au travers de *points de connexion* d'entrée et de sortie d'information. [Shrivastava 1998] étend la notion de port d'entrée en introduisant deux concepts : la notion d'entrée multiple et la notion de priorité. Une donné *d* reçue sur un point de connexion avec entrée multiple est la concaténation des données de toutes les entrées : la réception de *d* est donc conditionnée par la présence d'une donnée sur chacune des entrées. Le rôle d'une priorité d'un point de connexion d'entrée est de spécifier en cas de compétition entre deux points de connexion la donnée à lire en priorité par le composant. Nous considérons par la suite que les points de connexion des composants font partie intégrante des composants (internes aux composants, mais visibles de l'extérieur).

Les points de connexion sont reliés entre eux par des *connecteurs*. Un connecteur relie généralement deux points de connexion. Cependant, [Shrivastava 1998] autorise la connexion d'un point de connexion d'entrée p1 vers un autre point de connexion d'entrée p2 : toute donnée émise vers p1 est ainsi transmise à p1 et à p2. La manipulation de ces connecteurs, indépendamment des composants, permet de limiter le couplage entre les modules et ainsi de manipuler les connexions sans modification des composants [Purtilo 1994][Assis 1997].

3.4.8.2 *Mise en œuvre de l'architecture*

La mise en œuvre de l'architecture concerne l'implantation des éléments architecturaux : composants, points de connexion et communication entre points de communication.

Les composants sont associés soit à des grappes d'objets ou à des tâches logicielles. Dans ARGUS [Liskov 1985] les composants, appelés *Guardians* [Liskov 1983], sont l'agrégation d'un flot d'exécution, d'objets persistants et d'objets non persistants. [Shrivastava 1998] reprend cette approche en la justifiant par le modèle de tâche inhérent à la modélisation logicielle des systèmes transactionnels. L'environnement *Chimera* [Stewart 1997] reprend aussi cette approche pour les applications embarquées temps-réel en introduisant des composants appelés PBO – Port-Based Objects – (cf. Figure 3-7) qui sont des unités logicielles indépendantes qui correspondent à des tâches logicielles. A l'image des objets classiques, les PBOs offrent en plus à leur environnement des méthodes aux travers de leurs ports de communication. Un port spécifique appelé *port des ressources* donne aux PBOs une interface vers des drivers. De plus, il est possible de spécifier des contraintes temps-réel au niveau des PBOs (par exemple, pour un PBO dont la tâche est périodique, la valeur de la période). Enfin, le dernier type de composant identifié est l'*agent*. Ce dernier possède les trois propriétés suivantes [Keramane 1998]['Tagant 1998] : l'autonomie, l'intelligence, et la mobilité. Les agents sont constitués d'unités d'exécution EU (*Execution Unit*, en général une tâche du système) et de ressources (fichiers, pointeurs, etc.). Une machine virtuelle appelée CE (*Computational Environnement*) capable d'exécuter les EU doit être déployée sur tous les sites d'accueil des agents. Elle sert d'interface entre le noyau et/ou la couche réseau, et les composants.

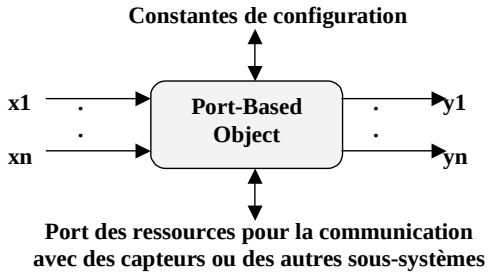


Figure 3-7. Port-Based Object introduit dans [Stewart 1997].

Les communications entre composants sont réalisées soit par passage de message avec support du mécanisme RPC (*Remote Procedure Call*) [Kramer 1985] ou d'un bus logiciel [Purtilo 1994][Pellegrini 1999], soit par mémoire partagée [Stewart 1997] pour des questions de performance.

Finalement, l'approche de [Shrivastava 1998] d'assimiler les tâches logicielles et les données de ces tâches à des composants constitue un découpage architectural adaptable aux logiciels du spatial (cf. paragraphe 2.3.3). C'est par ailleurs l'approche qui a été suivie par [Cailliau 2001]. En termes de mise en œuvre de la communication, les bus logiciels souffrent d'un manque de performance et d'une trace mémoire importante, mais offre une gestion de l'hétérogénéité nécessaire dans le domaine du spatial en raison de la diversité des environnements matériels et des systèmes d'exploitation supports.

3.4.8.3 *Processus de reconfiguration dynamique*

Le processus de reconfiguration consiste à élaborer un script de reconfiguration dynamique, puis à exécuter ce script sur l'application à reconfigurer. Ce script comporte une description des opérations de reconfiguration dynamique à effectuer sur l'application ou alors une description de l'architecture à déployer. Dans ce dernier cas, l'architecture est décrite, à l'aide de langages dits *ADL* (*Application Description Language,* cf. [Medvidovic 2000]), en termes de composants, d'interconnexions entres composants, des services offerts par ces composants et du déploiement de ces composants [Kramer 1985].

Une fois élaboré, le script est interprété par un gestionnaire de configuration, à l'image du *configurateur* de CONIC qui possède une vue structurelle applicative afin d'exécuter le script.

3.4.8.4 *Opérations de reconfiguration sur les architectures à composants*

Les applications à architecture à composants supportent deux types de reconfiguration : les modifications du schéma de la configuration de l'architecture [Assis 1997], qui sont indépendantes du comportement interne des composants, et les reconfigurations internes aux composants. Nous décrivons par la suite trois changements externes aux composants (changement de structure, changement de mise en œuvre de l'architecture, changement de géométrie) et trois changements internes aux composants (changement de d'interfaces, changement de comportement, et changement d'état).

Changements externes aux composants

- Un changement de structure consiste en la modification de la configuration structurelle d'une application. Les changements structuraux de base sont l'ajout d'un composant, le retrait d'un composant et enfin la modification des liaisons entre composants.
- Un changement de mise en œuvre de l'architecture consiste en la modification de la gestion de l'architecture logicielle, et plus particulièrement de la transmission de l'information entre les points de connexion (format de transport des données).

- Un changement de géométrie consiste en la migration de site d'exécution d'au moins un composant avec conservation de l'état des composants [Assis 1997].

Changements internes aux composants

- Un changement d'interface consiste en la modification des services offerts par un composant, c'est-à-dire l'ajout ou le retrait d'une interface de communication de ce composant
- Un changement interne d'un composant consiste en la modification du comportement du composant ou de ses données.
- Un changement d'état d'un composant consiste à modifier l'état dans lequel ce composant se trouve. Par exemple, dans l'environnement CONIC [Kramer 1985], un composant impliqué par une reconfiguration doit préalablement passer dans l'état *passif* ou *gelé*.

Nature des opérations de reconfiguration

L'environnement Polylith se distingue par le nombre et le type des changements supportés : les changements de structure, de géométrie, d'interface et enfin d'implantation. Les primitives de reconfiguration sont de trois types [Purtilo 1991] :

1. les primitives de gestion de la synchronisation lors des reconfigurations. Cela correspond à la gestion de l'arrêt d'interfaces ou de composants.
2. Les primitives de modification des composants : ajout/retrait de modules, modification des interfaces des composants, modification d'attributs d'interfaces. Des primitives permettent par ailleurs d'obtenir le droit de modification sur un composant.
3. Enfin, les primitives de gestion de liens entre composants : création ou destruction d'un lien entre deux. Les changements de géométrie sont obtenus par combinaison des trois types de primitive.

Si le but de l'environnement ARGUS est d'offrir aux applications un environnement distribué tolérant aux fautes, certaines opérations proposées permettent des changements dynamiques de configuration. Ainsi, l'injection d'une erreur au niveau d'un *guardian* permet son arrêt *guardians*, puis sa modification. De plus, ARGUS supporte des opérations d'ajout et de retrait de *guardians*.

On peut regretter dans les environnements étudiés l'absence de support pour des opérations intra-composants qui permettraient de d'éviter des ajouts/ retraits de composants. En effet, si les opérations de reconfiguration des architectures à composants offrent un haut degré de reconfigurabilité, leur limitation à la structure architecturale de base va à l'encontre de la continuité de service. Considérons l'exemple de la modification d'un composant. Les opérations de modification d'architecture contraignent, afin de modifier le comportement interne d'un composant c1, à remplacer c1 par un composant c2. Cela consiste à ajouter le composant c2 à l'application, à amener c1 à son point de reconfiguration (et donc à isoler c1), à générer l'état de c2 en fonction de c1, à mettre à jour les connexions de c1 et c2 afin que c2 remplace c1, et enfin à démarrer c2. Les opérations logicielles de recopie d'état sont particulièrement coûteuses en termes de mémoire et de puissance CPU, d'où une probabilité plus élevée d'induire un arrêt de service.

3.4.8.5 *Gestion des contraintes applicatives*

Les contraintes applicatives ont été définies au paragraphe 3.3.3.5. Trois approches des environnements à composants compatibles entre elles permettent de gérer ces contraintes.

La première approche consiste à modéliser l'architecture logicielle et ses modifications avec un langage ADL. Par exemple, [Allen 1998] met en évidence la modélisation de contraintes intrinsèques au sein de l'ADL *Wright*. La modélisation conjointe de l'architecture, des différentes configurations de l'architecture et des contraintes relatives à cette architecture permet d'obtenir des

garanties en termes de contraintes intrinsèques syntaxiques et de bon fonctionnement (deadlock). Ces garanties sont obtenues en donnant une sémantique formelle à cet ADL qui est basée sur CSP [Hoare 1985]. En termes d'expression des contraintes intrinsèques temps-réel, [Stewart 1997] propose de les modéliser au niveau des PBO, mais elles ne sont pas prises en considération lors de la reconfiguration.

La deuxième approche consiste à exécuter certaines opérations de reconfiguration à des *points de reconfiguration*. Ainsi, [Kramer 1985] propose de gérer la cohérence logique et temporelle des messages en transit en introduisant la notion d'état aux composants : état *actif*, état *passif* et état *gelé*. Dans l'état *passif*, un composant ne peut envoyer de messages et dans l'état *gelé*, il ne peut pas recevoir, traiter ou envoyer de message. Un composants *actif* ne peut pas changer d'état s'il est en cours de traitement d'une requête. Pendant une reconfiguration, le *configurateur* gèle les composants directement concernés, et rend « *passif* » ceux émettant des requêtes vers les composants gelés. Dans ARGUS, les modifications de configuration ne peuvent être effectuées qu'entre les *transactions* qui sont des parties de code atomiques. Ainsi, une reconfiguration suppose préalablement d'arrêter ou d'annuler les *transactions* en cours, puis de reconfigurer les *guardians*, et enfin de redémarrer les *guardians*. Cette notion de *transaction* assure la consistance logique des *guardians*. Dans le cas de changement de géométrie applicative, la gestion de la cohérence logique applicative nécessite en outre des mécanismes *d'encodage des états des composants*. [Hofmeister 1993][Purtilo 1994]. Par exemple, dans le cas où un agent s'apparente à une grappe d'objets, l'algorithme de migration de code est le suivant [Tagant 1998] : 1) définition de l'agent à faire migrer ; 2) clonage à distance des objets invariants de la grappe d'objets de l'agent par transmission de leur organisation et de leurs valeurs ; 3) clonage à distance des objets dépendants du contexte local de la grappe. La création est dite « intelligente » car elle doit s'effectuer en fonction du contexte local ; 4) transmission des informations pour remettre à jour les liens sortants de la grappe ; 5) transmission des informations d'exécution ; 6) Destruction de la grappe au niveau du site initial. Cet algorithme soulève le problème de la mobilité des ressources. En effet, si certaines ressources sont facilement « transportables » (i.e. peuvent être encodées), certaines sont très difficilement voire pas transférables (par ex., un lien sur une ressource matérielle locale non partagée) : on parle alors d'un système à mobilité faible.

Enfin, la dernière approche repose sur des techniques a priori. Par exemple, [Cailliau 2001] propose une simulation a priori de l'exécution du script de reconfiguration sur l'application spatiale embarquée. Une fois la reconfiguration simulée avec succès, le script est envoyé à bord puis exécuté. Cette technique peut-être utilisée conjointement avec les techniques précédentes, à l'instar de la méthodologie proposée dans [Cailliau 2001].

Nous notons dans les deux premières approches un manque de prise en compte des contraintes intrinsèques temps-réel et extrinsèques (cf. [Gupta 1996]). Les approches consistant à définir des points de reconfiguration permettent de garantir la consistance logique applicative à peu de frais pour le programmeur. L'idée d'obtention de garantie a priori nous paraît intéressante (troisième approche) et a été soulignée comme telle par [Gupta 1996]. Mais l'approche par simulation a priori proposée dans [Cailliau 2001] nous paraît inadaptée dans la mesure où elle n'offre pas de garantie formelle, mais une simple information sur un déroulement possible de la reconfiguration. [Gupta 1996] souligne par ailleurs ce besoin de validation formelle (et non de simulation).

3.4.8.6 *Conclusion*

En résumé, les approches à composants présentent les avantages suivants :

- une implantation simple (architecture, communication par mémoire partagée, etc.), adaptable aux environnement spatiaux ;
- une diversité de modification ;

Mais :

- la gestion des contraintes applicatives est insuffisante [Gupta 1996].

3.5. Comparaison des mécanismes et positionnement

3.5.1 Taxonomie des mécanismes

Le graphique de la Figure 3-8 présente les mécanismes supports de la reconfiguration dynamique en fonction du degré de dynamicité (axe des abscisses) et en fonction de la couche de l'environnement impliquée dans le mécanisme (axe des ordonnées). La reconfiguration est d'autant plus statique que le temps de mise à jour du logiciel est élevé (temps entre l'arrêt et le redémarrage). La reconfiguration est d'autant plus dynamique que les contraintes intrinsèques et extrinsèques sont prises en compte.

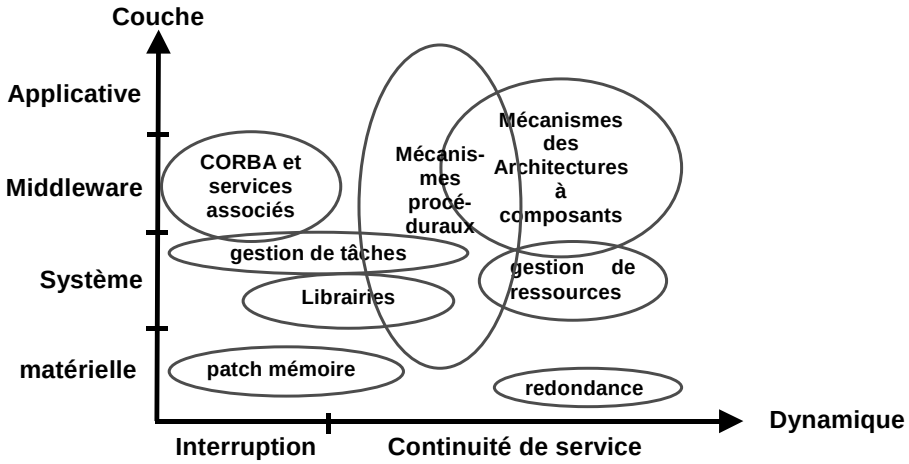


Figure 3-8. Classement des mécanismes selon la dynamique de la reconfiguration et la couche impliquée.

Nos besoins en termes de continuité de service et de mise en œuvre nous incite à nous pencher vers les systèmes les plus dynamiques et dont l'implantation des mécanismes est réalisée dans les couches les plus hautes. La redondance matérielle est exclue pour des questions d'adéquation avec les contraintes du spatial. Les mécanismes purement basés sur les systèmes d'exploitation sont trop limitatifs en terme de degré de reconfigurabilité (cf. paragraphe 3.4.5). Les mécanismes dits procéduraux et les mécanismes relatifs aux architectures à composants semblent donc les plus à même de répondre à nos besoins. Nous comparons par la suite les deux approches en termes de gestion des contraintes applicatives.

Dans les architectures logicielles procédurales, la préservation de la consistance applicative est assurée par des mécanismes d'indirections sur les procédures ou les données [Boute 1981][Engram 1983]. La charge du programmeur consiste alors en la déclaration des dépendances sémantiques entre les différentes procédures et en la mise en place d'un code effectuant les modifications de procédures sur l'application [Shrivastava 1998]. Dans les architectures modulaires, la charge du programmeur consiste d'une part à prévoir dans chaque composant le passage du fonctionnement nominal de ce composant à son point de reconfiguration et d'autre part, à générer un script de reconfiguration modélisant les modifications désirées. En conclusion, dans les deux cas, la préservation des propriétés intrinsèques est assurée par l'identification des sous-ensembles applicatifs impactés par la reconfiguration (procédures impactées ou composants impactés) et par l'application d'opérations de reconfiguration uniquement à des points d'exécution précis. Nous voyons toutefois une limite à l'approche procédurale : la mise à jour d'applications multitâches procédurales paraît complexe car les procédures modifiées doivent être inutilisées par l'ensemble des tâches applicatives. De plus, les procédures modifiées ne peuvent être que des procédures de bas niveau i.e. en bas de l'arbre d'appel [Segal 1993], ce qui réduit le degré de reconfigurabilité. A contrario, la diversité des modifications offertes par les opérations de reconfiguration sur les architectures à composants ainsi que le support des applications multitâches nous incitent à nous tourner vers ces architectures.

3.5.2 Positionnement

La principale limite des approches à composants réside dans la gestion des contraintes applicatives [Gupta 1996]. Le processus de reconfiguration des applications développées selon une architecture à composants est le suivant : (1) description de la nouvelle configuration, (2) simulation éventuelle du passage de la configuration précédente à la nouvelle, (3) exécution par un gestionnaire de configuration d'opérations de reconfiguration afin de faire évoluer la configuration applicative. Nous proposons d'enrichir ce processus par une prise en compte, dans les trois étapes de ce processus, des contraintes intrinsèques et extrinsèques applicatives.

Au niveau de la première étape, il s'agit d'exprimer la nouvelle configuration et ses contraintes (intrinsèques et extrinsèques) par modélisation (1) de l'exécution du script de reconfiguration et (2) des contraintes devant être respectées pendant le passage de l'ancienne configuration à la nouvelle. A cette fin, l'utilisation d'un langage de type ADL paraît mal adaptée pour des questions d'expressivité de ce langage (faible capacité d'expression des contraintes temps-réel ou de service, cf. [Medvidovic 2000]) et pour des questions de contrainte industrielle (cf. paragraphe 3.4.2). En particulier, afin de se conformer au cycle méthodologique en vigueur chez Alcatel Space, nous proposons d'utiliser le langage UML. Toutefois, nous devons enrichir ce langage en raison de ses lacunes en termes d'expressivité structurelle et comportementale [Apvrille 2001a] et proposer un canevas de modélisation UML de l'architecture à composants choisie.

En ce qui concerne la deuxième étape, il s'agit non pas de simuler l'exécution du script de reconfiguration sur l'application [Cailliau 2001], mais de la valider formellement. Pour cela, nous proposons de dériver la modélisation UML construite à la première étape en une spécification dans un langage formel puis de la valider formellement. La validation formelle peut amener à refuser le script de reconfiguration ou à le modifier en rajoutant des contraintes concernant l'exécution des opérations. Ces contraintes entre opérations sont de quatre types : contraintes de synchronisation (une opération o1 doit être effectuée après une opération o2), contraintes relatives à l'exécution d'une opération (exécution au plus en *n* unités de temps et sans générer d'erreur), contraintes temporelles entre opérations (par exemple, les opérations de réactivation des composants doivent être effectuées au plus tard *n* unités de temps après l'opération d'arrêt du composant considéré, ce qui nous conduit à introduire la notion de *point de reconfiguration temporellement contraint*), et enfin, contraintes logiques entre opérations (par ex., tel buffer ne doit pas dépasser tel seuil pendant que tel module est arrêté). Afin de répondre au besoin de modélisation dans un script de ces contraintes, nous proposons d'utiliser les réseaux de Petri à flux temporels introduits dans [Diaz 1993][Sénac 1996] et de les étendre avec des jetons colorés [Jensen 1997] afin de gérer le code de retour des opérations (la bonne exécution ou non de l'opération). Enfin, en ce qui concerne les opérations de reconfiguration, nous avons vu dans le paragraphe 3.4.8.4 que les opérations inter-composants peuvent avoir un impact sur la continuité de service : il apparaît intéressant de proposer en plus des opérations de haut niveau (opérations inter-composants), des opérations de bas niveau (opérations intra-composants).

Enfin, en ce qui concerne la troisième étape, il s'agit de proposer un support à l'architecture à composants et un support à l'exécution du script de reconfiguration. L'introduction de la notion de composants au sein d'un système embarqué nous laisse penser à l'utilisation de technologies logicielles à base d'objets (1). Le support aux opérations de reconfiguration nécessite un mécanisme d'intégration dynamique de code (2). Enfin, le besoin d'indépendance vis-à-vis des couches inférieures nous amène à considérer les machines virtuelles d'exécution de code (3). Les points (1), (2) et (3) nous conduisent à évaluer l'adéquation des environnements Java avec les contraintes du spatial. En effet, ces environnements logiciels à base d'objets offrent une indépendance d'exécution par rapport au système logiciel et matériel sous-jacent (machine virtuelle Java) et intègrent un mécanisme d'intégration dynamique de classes Java [Malabarba 2000].

3.6. Conclusion

Ce chapitre a mis en évidence les différentes approches relatives à la reconfiguration dynamique du logiciel. Tout d'abord, nous avons présenté la reconfiguration dynamique comme une technique support à la maintenance logicielle. Ensuite, nous avons exposé les motivations de la reconfiguration dynamique : correction d'erreurs, évolutivité, réutilisation, personnalisation d'un service, redéploiement applicatif, etc.. Enfin, nous avons analysé les travaux du domaine de la reconfiguration dynamique. Cette analyse a permis de mettre en évidence une adéquation des architectures à composants avec nos propres besoins en termes de mise en œuvre et de degré de reconfigurabilité. Mais les approches à composants se limitent à la gestion des contraintes intrinsèques logiques applicatives et oublient totalement les contraintes temps-réel et extrinsèques [Gupta 1996]. Nous proposons de prendre en compte l'ensemble de ces contraintes par un cycle de développement logiciel enrichi par une phase de validation formelle a priori et par un support embarqué adapté.

Finalement, notre contribution se décline en trois points :

- *La modélisation des contraintes applicatives* avec le langage UML.
- *La conception et la gestion des contraintes intrinsèques et extrinsèques* au niveau méthodologique et au niveau support.
- *L'implantation* par la mise en œuvre dans le spatial des mécanismes architecturaux et de reconfiguration dynamique avec un environnement à base d'objets Java.

Les deux chapitres suivants présentent nos travaux relatifs aux points énumérés ci-dessus. Le Chapitre 4 présente la modélisation et la validation formelle du respect des contraintes applicatives lors des reconfigurations dynamiques. Le Chapitre 5 met en évidence l'ensemble des mécanismes (méthodologie, mécanismes supports, implantation) nécessaires à la gestion des contraintes applicatives.

Chapitre 4

Un cadre formel pour la modélisation de reconfiguration dynamique

Résumé. Dans le chapitre précédent, l'analyse des contributions dans le domaine de la reconfiguration dynamique d'applications a permis de mettre en évidence la supériorité des architectures à composants pour répondre au problème de la reconfigurabilité des logiciels du spatial. Mais, à l'instar des autres approches de reconfiguration, les architectures à composants ne prennent que partiellement en compte la gestion des contraintes applicatives intrinsèques et extrinsèques et n'offrent donc qu'une réponse partielle au problème de la reconfiguration des logiciels de télécommunication par satellites. Nous proposons dans ce chapitre de prouver le respect de ces contraintes par une technique de validation formelle a priori. A cette fin, nous proposons une architecture à composants adaptée au milieu du spatial, puis un cadre formel de modélisation basé sur le langage UML (contrainte industrielle) et permettant la preuve formelle de propriétés sur une modélisation logicielle. Par la suite, nous montrons comment, par utilisation de cet environnement et en s'inspirant des contributions effectuées dans les langages de modélisation d'architecture, il est possible de modéliser l'architecture logicielle précédemment introduite et de modéliser les contraintes d'exécution associées à l'application (contraintes internes et externes). Enfin, en accord avec l'approche proposée par certains langages de modélisation d'architectures (ADL), nous introduisons la notion de gestionnaire de configuration d'une application, qui, modélisé et validé conjointement avec l'application, nous permet de déterminer si un changement de configuration respecte les contraintes d'exécution de l'application.

4.1. Introduction

Le chapitre 2 a mis en évidence le besoin de reconfigurabilité avec continuité de service des logiciels de télécommunication du spatial. Dans le chapitre précédent, l'analyse des contributions en termes de reconfiguration dynamique a montré l'absence de prise en compte de façon formelle des contraintes applicatives intrinsèques temps-réel et extrinsèques lors des reconfigurations dynamiques. Pourtant, ce besoin est mis en évidence par de nombreuses contributions, notamment par [Kramer 1985][Hofmeister 1993][Gupta 1996]. [Gupta 1996] souligne par ailleurs l'intérêt de prendre en compte l'ensemble de ces contraintes lors d'un processus de validation formelle a priori mais n'apporte pas de solution à ce sujet. Nous proposons ainsi, dans ce chapitre, de répondre au besoin de validation formelle a priori du respect des contraintes applicatives lors des reconfigurations.

De nombreux environnements de modélisation des applications à composants ont été proposés par le passé. Ces environnements reposent sur un langage de type ADL [Medvidovic 2000]. Si les contraintes industrielles, et tout particulièrement celles d'Alcatel, nous incitent à utiliser le langage UML (cf. paragraphe 2.3.3.1, page 19), son utilisation soulève des problèmes en termes de modélisation et de validation. En termes de modélisation, il s'agit de pouvoir représenter l'architecture applicative, ses contraintes intrinsèques et extrinsèques applicatives et les configurations de ces architectures. En termes de validation, il convient de prouver a priori le respect des contraintes précédemment modélisées au regard d'une reconfiguration. Mais si certains

ADLs ont été conçus pour répondre en partie à ce besoin [Allen 1997], UML a été conçu pour la conception de systèmes et notamment de systèmes logiciels orientés objets. En effet, en termes d'expressivité de systèmes temps-réel, UML possède de nombreuses lacunes en termes d'expression de la structuration de systèmes temps-réel (parallélisme, synchronisation) et du comportement temps-réel [Terrier 2000][Saqui 2001]. En ce qui concerne la validation, UML n'a pas de sémantique formelle. Aussi, par utilisation de mécanismes d'extension de ce langage, nous proposons un cadre formel basé sur UML, inspiré des contributions du domaine des ADLs et permettant la modélisation des architectures à composants, des contraintes intrinsèques et extrinsèques associées à ces architectures, et la validation formelle a priori du respect de ces contraintes.

Le chapitre est organisé comme suit. Premièrement, nous proposons une architecture de haut niveau basée sur la notion de composants logiciels en adéquation avec la structure des logiciels du spatial. Deuxièmement, nous définissons notre cadre formel basé sur UML. Cette présentation comprend trois parties principales. Tout d'abord, la définition d'opérateurs permettant de donner une sémantique formelle à la structuration sous forme de classes. Ensuite, le présentation de nouveaux opérateurs logiques et temporels innovants permettant la description comportementale de classes logicielles. Enfin, la description du processus de validation associé au cadre formel. Ce chapitre se poursuit ensuite sur la modélisation, par utilisation du cadre formel introduit, des applications temps-réel embarquées et de leurs contraintes. Nous montrons notamment comment modéliser une application développée selon l'architecture de haut niveau introduite précédemment. Nous montrons comment par utilisation de ce cadre formel il est possible de modéliser des contraintes applicatives intrinsèques (y compris temps-réel) et extrinsèques, et de valider ces contraintes au regard d'opérations de reconfiguration dynamique. Enfin, nous concluons ce chapitre.

4.2. Une architecture logicielle intrinsèquement reconfigurable

4.2.1 Introduction

Cette section entend définir une architecture logicielle de haut niveau intrinsèquement reconfigurable. Le chapitre 3 a permis de mettre en évidence la supériorité des applications architecturées sous forme de composants en termes de reconfigurabilité (cf. paragraphe 3.4.8.4). Le choix de ces architectures pour les applications spatiales soulève le problème de leur capacité à être mises en œuvre dans un environnement spatial. Cela concerne la mise en œuvre de la structuration en termes de composants, des mécanismes support à l'architecture (par ex. mécanismes de communication entre les composants) et enfin des mécanismes relatifs à la reconfiguration (par ex., les points de reconfiguration, cf. [Hofmeister 1993]).

Par la suite nous définissons une architecture de haut niveau à composants qui tienne compte des problèmes précités. Pour cela, nous nous appuyons sur les architectures à composants déjà définies dans le milieu des systèmes distribués [Kramer 1985], des systèmes transactionnels [Shrivastava 1998], et des systèmes embarqués [Stewart 1997] et plus particulièrement des systèmes spatiaux [Cailliau 2001].

4.2.2 Définition de l'architecture logicielle

4.2.2.1 *Concepts fondamentaux de l'architecture*

Une architecture à base de composants logiciels encapsule en leur sein des fonctionnalités de même nature et autorise des communications entre ces composants logiciels [Kramer 1985]. Dans les applications spatiales, les fonctionnalités de même nature sont généralement associées à une tâche du système applicatif. C'est pourquoi nous reprenons l'idée de [Liskov 1985] et plus tard de [Shrivastata 1998] d'associer les tâches applicatives aux composants logiciels. Cette idée a par

ailleurs aussi été reprise dans les contributions des systèmes embarqués [Stewart 1997] et plus particulièrement pour ceux du spatial [Cailliau 2001] (cf. paragraphe 3.4.8.2).

La méthodologie de développement logiciel en vigueur chez Alcatel Space (cf. paragraphe 2.3.3.1) s'appuie notamment sur l'outil *Rhapsody* permettant la génération automatique de code [Douglas 1999]. Les fonctionnalités applicatives sont organisées sous forme de tâches logicielles qui communiquent de façon asynchrone au travers de canaux de communication. Notre souci de ne pas rompre avec la méthodologie en vigueur nous incite à proposer une communication asynchrone entre ces composants, à l'instar de nombreuses architectures à composants [Kramer 1985][Purtilo 1994][Stewart 1997][Palma 1999][Pellegrini 1999].

Finalement, les éléments architecturaux de notre approche sont :

- les composants logiciels, appelé *modules* ;
- les points de communication des composants, appelés *ports*. Ces ports se déclinent *en ports de sortie*, et *ports d'entrée*, selon qu'ils sont destinés à émettre ou à recevoir des données. Un port appartient à un et un seul module.
- les connecteurs, appelés *liens*. Un lien L assure le transfert de données entre un port de sortie et un port d'entrée de façon asynchrone.

4.2.2.2 *Concepts logiciels de l'architecture*

Cette section présente les concepts logiciels relatifs aux modules et aux communications entre ces modules. La mise en oeuvre de ces concepts sera traitée dans le chapitre 5, paragraphe 5.6.

Implantation des modules

[Liskov 1985] et plus récemment les environnements à base d'objets mobiles [Tagant 1998] ou les environnements des systèmes temps-réel [Stewart 1997] soulignent l'intérêt des architectures objets pour mettre en œuvre les architectures à composants. Les composants représentent alors, dans le cas de [Liskov 1985], un flot d'exécution ainsi qu'un agrégat d'objets. Cette approche nous paraît pertinente dans la mesure où une architecture objets regroupe, à l'instar des architectures à composants, les fonctionnalités de même nature au sein de classes architecturales. De plus, cette approche reprend le découpage objet imposé par la méthodologie Alcatel en matière de développement d'applications spatiales (cf. 2.3.3.1). Ainsi, un module apparaît comme un ensemble d'objets dont un et un seul représente une tâche logicielle.

États des modules

On distingue deux types d'états : l'état *interne* des modules et l'état *externe*.

L'état interne d'un module représente l'ensemble des états de tous les composants logiciels de ce module. Un module étant un agrégat d'objets, l'état interne d'un module est défini comme l'union des états de tous les objets de ce module.

L'état externe d'un module représente son stade d'exécution dans son cycle de vie. Le cycle de vie d'un processus exécutable est généralement « créé – actif – détruit ». Les contributions des architectures à composants ont mis en évidence le besoin d'introduire en plus des états relatifs à la gestion de la consistance logique applicatives lors des reconfigurations dynamiques [Kramer 1985][Hofmeister 1993][Palma 1999]. Dans l'optique de simplifier la mise en œuvre et de limiter la charge du programmeur, nous ne reprenons pas l'approche de passivité et de gel des modules introduite dans [Kramer 1985] : nous n'introduisons qu'un seul état, *suspendu*, correspondant au point de reconfiguration d'un module et qui implique de la part du module une absence totale de traitement (contrairement à l'état passif de [Kramer 1985]. Finalement, au regard de l'implantation objet des modules et de l'état de reconfiguration, nous pouvons définir cinq états :

- *chargé* : l'ensemble des classes des objets relatifs au module considéré sont présentes dans la mémoire vive de l'application.

- *créé* : l'ensemble des objets initiaux du module considéré sont présents dans la mémoire vive de l'application. Ces objets ne peuvent être créés que si l'ensemble des classes nécessaires à l'instanciation de ces objets sont en mémoire vive i.e. le module est dans l'état *chargé*.
- *actif* : le module est en phase nominale d'exécution.
- *suspendu* : le module est arrêté (point de reconfiguration).
- *détruit* : le module est détruit : les objets du module sont éliminés de la mémoire vive.

Les transitions possibles entre ces états sont présentées à la Figure 4-1:

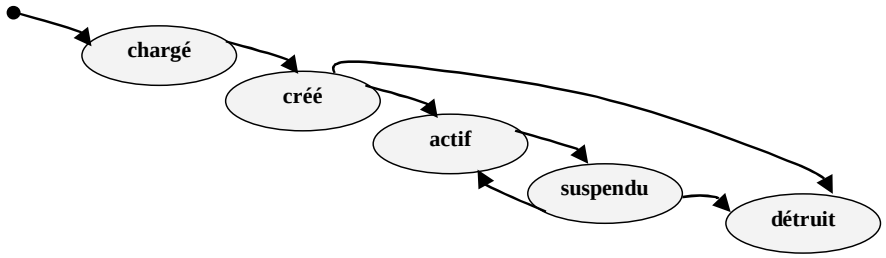


Figure 4-1. Etats d'exécution des modules.

Finalement, la notion d'état externe des modules est introduite (1) à des fins de gestion des ressources mémoires (état *chargé*, état *créé*, état *détruit*), et (2) à des fins de préservation de propriétés applicatives (état *suspendu*).

Taxonomie des modules

Nous proposons de classifier les modules en fonction de leur comportement interne. En effet, les modules peuvent être différenciés selon qu'ils utilisent ou non leurs ports d'entrée et de sortie, et selon qu'ils utilisent des interfaces externes de communication (appels systèmes). Les modules peuvent réaliser quatre interactions avec leur milieu extérieur (cf. le modèle PBO de [Stewart 1997]) : la lecture de données sur un port d'entrée, l'écriture de données sur un port de sortie et la lecture ou l'écriture de données via un appel système. Nous nous inspirons de la classification effectuée par [Stewart 1997] pour les systèmes temps-réel pour exhiber, au regard des spécifications logicielles des applications spatiales, quatre types de modules :

- les modules obtenant des données sur leur port d'entrée, et manipulant ces données puis produisant des données sur un ou plusieurs ports de sortie ; nous qualifions ces modules de *modules de traitement asynchrones*.
- Les modules ayant un comportement périodique : ces modules, qualifiés de *modules de traitement périodique*, lisent ou produisent périodiquement des données.
- Les modules récupérant des données sur des interfaces systèmes et les émettant sur des ports de sortie, ou les modules récupérant des données sur des ports d'entrée, et écrivant ces données sur des interfaces systèmes. On distingue ces modules des autres modules car ces modules lisent ou écrivent des données indépendamment des autres modules. On qualifie respectivement ces types de modules de *modules de lecture système* et de *modules d'écriture système*.

Communications entre modules

Nous avons défini une architecture à composants pour laquelle les liens (connecteurs) assurent des transmissions asynchrones. Nous considérons par la suite les mécanismes logiciels sous-jacents à cette transmission.

En considérant l'approche utilisée dans la méthodologie d'Alcatel, le générateur automatique de code de l'outil [Douglass 1999] implante une communication asynchrone entre les tâches du système. La tâche possède un seul *buffer* de réception de tous les messages relatifs à cette tâche. La tâche possède donc un port d'entrée. Si le concept de communication par passage de

messages nous paraît adapté, en revanche, l'unicité du buffer de réception nuit à la diversité du comportement de réception des messages. En effet, il n'est pas possible à une tâche logicielle de privilégier la réception de tel type de message : nous pensons qu'il est souhaitable pour un module de pouvoir offrir plusieurs points d'entrée et d'associer un buffer à chaque point d'entrée. Mais, nous ne reprenons pas l'idée de [Shirivastava 1998] concernant les ports à entrées multiples et la duplication de données au niveau des ports d'entrée (cf. 3.4.8.1). Si l'approche se justifie par la complexité des interconnexions dans les systèmes transactionnels, dans le cas des systèmes embarqués, les approches consistent à opter pour des ports de connexion peu complexes [Stewart 1997]. En conséquence, dans notre approche, un port est impliqué dans au plus un lien.

Nous l'avons vu dans le chapitre 3, la reconfiguration dynamique soulève le problème de la consistance logique d'interconnexion des modules. Afin de faciliter l'analyse de cette consistance, nous reprenons l'approche proposée dans maintes contributions concernant le typage des ports [Stewart 1997]. L'approche objet de notre modèle nous incite à introduire un typage sous forme de classes. L'analyse de la consistance d'interconnexion consiste à vérifier que le type d'un port de sortie d'un lien *l* est *compatible* avec le type du port d'entrée de *l*. Cette compatibilité se définit comme suit :

Soit l un lien, p_s un port de sortie, t_s le type associé à ce port, p_e un port d'entrée et t_e le type associé à ce port d'entrée.

Si $l = \langle p_s, p_e \rangle$, alors la contrainte de compatibilité implique que $(t_s = t_e) \vee (t_s$ hérite de $t_e)$.

En effet, pour qu'un message de type t_s puisse être lu par p_e, il faut que le format des données définies par t_e soit identique au format du message reçu sur p_e. Du point de vue objet, cela se traduit par une présence d'identificateurs identiques dans la classe du message reçu, ce qui se traduit par une égalité de classes ou par un héritage de classes.

4.2.3 Conclusion

Dans cette partie, nous avons défini une architecture à composants en prenant en considération deux points essentiels : la compatibilité de cette architecture avec la méthodologie logicielle employée chez Alcatel Space et l'aspect intrinsèquement reconfigurable de cette architecture. Cette dernière est directement inspirée des contributions des autres domaines d'application et spécialisée au regard de nos contraintes.

Il convient à présent de proposer un cadre formel de modélisation et de validation des applications construites selon l'architecture que nous venons de présenter. Par la suite, nous présentons un cadre formel (section 4.3) puis nous montrons son aptitude à modéliser notre architecture (section 4.4).

4.3. Un cadre formel de modélisation logicielle basé sur UML

4.3.1 Besoin

Notre objectif est de proposer un cadre formel permettant la modélisation d'une application (architecture et comportement) et de ses propriétés dans le but de valider formellement la préservation de propriétés applicatives lors de reconfiguration dynamique. En termes de capacité de modélisation, ce cadre formel doit offrir des outils permettant de décrire l'architecture présentée précédemment (structure objet de l'architecture, mécanismes de communication inter-modules) ainsi que les contraintes associées à cette architecture : contraintes intrinsèques logiques et temps-réel et contraintes extrinsèques.

Pour des raisons d'adéquation avec la méthodologie de développement logiciel en vigueur chez Alcatel Space (paragraphe 2.3.3.1), nous développerons un cadre formel basé sur le langage UML [OMG 2001]. Mais l'incapacité des environnements UML à répondre à nos besoins de modélisation et de validation formelle nous conduit à introduire un nouveau profil UML (TURTLE) dont la sémantique formelle est donnée par une traduction en un langage formel.

4.3.2 Introduction à TURTLE

4.3.2.1 *Les limites d'UML*

Forte du support de *l'Object Management Group* (OMG), la notation UML est de plus en plus utilisée par les concepteurs de systèmes temps réel. Dans les faits, ce n'est pas un, mais plusieurs UML temps réel qui sont en compétition pour répondre aux exigences du Request For Proposal [OMG 1999b] de l'OMG.

- UML-RT est supporté par l'outil *Rose RT* et il inclut dans UML des concepts importés de ROOM [Selic 1998] ;
- RT-UML est supporté par l'outil *Rhapsody* et utilise autant que possible les constructions natives d'UML 1.3 [Douglas 1999] ;
- UML en amont de SDL est proposé par Telelogic (outil *TAU* [Bjorkander 2000]) ;
- L'outil d'Artisan *Software Real-Time Studio* [Artisan 1999].

Au-delà de leurs différences, les solutions implantées par ces outils industriels présentent des limitations importantes en termes d'expression de structuration, d'expression de comportement et enfin de validation.

En termes d'expression de structuration, le diagramme de classes UML permet la description de classes applicatives, et de relations entre ces classes. La principale limite concerne l'expressivité des associations entre classes dont la sémantique informelle est donnée à titre de documentation. De plus, la structuration des diagrammes de classes ne met pas en évidence la structure en termes de tâches applicatives, et les relations de parallélisme ou de synchronisation entre ces tâches.

En termes d'expression de comportement, les limites concernent principalement l'expression de contraintes temporelles. Ainsi, les opérateurs temporels sont limités à des temporisateurs à durée fixe. L'absence d'opérateurs pour gérer des intervalles rend impossible la description de la gigue et plus généralement de l'asynchronisme inhérent aux systèmes de télécommunication [Sénac 1996]. Enfin, il manque à UML un opérateur apte à exprimer une offre de synchronisation limitée dans le temps (modélisation des erreurs temporelles).

Enfin, en termes de validation, les outils sur le marché proposent deux solutions. La première consiste à réaliser des simulations visuelles par animation des modèles de comportement La deuxième solution consiste à utiliser les diagrammes de séquences : la violation des spécifications par une application se traduit par une non conformité entre un scénario d'exécution de l'application et le diagramme de séquences construit en phase d'analyse. Dans les deux cas, seul un sous-ensemble des fonctionnements applicatifs est étudié (simulation visuelle d'un comportement, analyse de quelques scénarios). Ainsi, les deux solutions proposées en termes de validation de propriétés se limitent à une exploration limitée du comportement applicatif.

4.3.2.2 *Les différentes approches*

Les limitations mentionnées ci-dessus ont été contournées en utilisant des primitives systèmes dépourvues de sémantique formelle, ce qui empêche de valider des modèles UML par la confrontation a priori à des exigences temporelles. Des actions concertées ont vu le jour pour remédier à cette situation et donner une sémantique précise à UML [Bruel 1998][Bruel 1999][Evans 1999]. Les travaux de recherche portent sur l'utilisation conjointe d'UML et d'une méthode formelle : Systèmes de Transitions Etiquetées [Guennec 2000][Jard 1998], réseaux de Petri [Delatour 1998], Z [Dupuy 2000], langages synchrones [Andre 2001][Dupuy 2001], PVS [Traore 2000] et E-LOTOS [Clark 2000]. Cependant, ces diverses solutions n'envisagent qu'une utilisation conjointe entre UML et le langage formel c'est-à-dire que le langage formel est utilisé directement au niveau des modèles UML.

4.3.2.3 *Notre proposition*

La solution que nous proposons est non pas l'utilisation conjointe d'un langage formel et du langage UML, mais l'utilisation transparente d'un langage formel, c'est-à-dire que les diagrammes UML sont étendus par des opérateurs UML ayant une sémantique formelle par traduction dans un langage formel. La validation formelle d'un tel modèle UML consiste alors en la traduction transparente de ce modèle UML en une spécification dans un langage formel, puis en une analyse d'accessibilité de cette spécification.

Notion de profil UML

Le cadre formel de modélisation basé sur UML que nous proposons est un *profil* UML pour les applications temps-réel embarquées. Un profil UML est une spécialisation du méta-modèle UML en un méta-modèle spécifique à une classe d'applications. Un profil UML est constitué d'éléments du méta-modèle, de nouveaux éléments construits à partir des mécanismes d'extension d'UML, d'une description de la sémantique du profil, de notations additionnelles et enfin de règles de traduction, validation et de présentation.

Le profil UML proposé par la suite fournit des outils destinés à améliorer la description logique et temps-réel des relations entre classes, et les comportements internes de ces mêmes classes. Cela se traduit, d'une part, au niveau du diagramme des classes, par l'introduction d'un nouveau stéréotype[1] UML au méta-modèle UML, et d'autre part, au niveau de la description comportementale des classes (diagramme d'activités), par l'introduction, entre autre, de trois opérateurs temporels. La sémantique du profil est donnée par traduction des modèles UML enrichis par nos outils en une spécification dans le support formel RT-LOTOS, algèbre de processus temporisée [Courtiat 2000]. Un point important est que la syntaxe RT-LOTOS demeure transparente aux utilisateurs d'UML qui peuvent ainsi valider leurs modèles UML sans apprendre RT-LOTOS. Le profil est nommé TURTLE, acronyme anglais de ***T****imed* ***U****ML and* ***RT-L****otos* ***E****nvironment*. Notre choix pour le langage RT-LOTOS se justifie par la disponibilité d'un outil de validation formelle [Courtiat 2000]. Mais le principe de dérivation vers un langage formel est générique et pourrait s'appliquer à d'autres supports de spécification formelle offrant des opérateurs temporels à l'instar des Réseaux de Petri à flux Temporels (RdPFT) [Diaz 1993][Sénac 1996].

Notons que la traduction des modèles UML vers RT-LOTOS ne doit pas être interprétée comme une tentative pour donner une syntaxe graphique à LOTOS, expérience déjà tentée dans le passé [Cheung 1990] mais qui a subi un échec en termes d'utilisation. Notre objectif est bien de proposer des opérateurs de haut niveau de description des systèmes temps-réel indépendamment du langage formel RT-LOTOS mais compatibles avec une méthodologie de conception du logiciel. La complexité de la traduction de certains de ces opérateurs dans le langage RT-LOTOS [Lohr 2002] témoigne de notre abstraction par rapport à ce langage formel.

RT-LOTOS

LOTOS [Bolognesi 1987] est une Technique de Description Formelle pour la spécification et la conception de systèmes de traitement distribués. Une spécification LOTOS se présente sous la forme d'un processus structuré en d'autres processus. Un processus LOTOS est une boite noire qui communique avec son environnement au travers de portes et sur le principe d'une offre de rendez-vous. Des échanges mono- ou bi-directionnels de valeurs sont autorisés lors de la synchronisation.

Le parallélisme et la synchronisation entre processus s'expriment par des opérateurs de composition : mise en séquence, synchronisation sur toutes ou certaines portes, choix non déterministe et entrelacement (composition parallèle sans synchronisation). Les opérateurs de composition sont identifiés par leurs symboles (cf. Tableau 4.1).

[1] Un stéréotype est un ajout indirect au méta-modèle UML. Le stéréotype TURTLE ainsi que les types abstraits sont identifiés graphiquement par un symbole « tortue» situé dans le coin supérieur droit de la classe considérée.

Tableau 4.1. Opérateurs LOTOS.

Opérateur	***Description***	***Exemple***
[]	Choix.	P[a,b,c,d] = P1[a,b] [] P2[c,d]
|||	Composition parallèle sans synchronisation.	P[a,b,c,d] = P1[a,b] ||| P2[c,d]
|[b,c,d]|	Composition parallèle avec synchronisation sur plusieurs portes (b,c,d).	P[a,b,c,d,e] = P1[a,b,c,d] |[b,c,d]| P2[b,c,d,e]
hide b in |[b]|	Composition parallèle avec synchronisation sur la porte b qui est de plus cachée.	P[a,c] = hide b in P1[a,b] |[b]| P2[b,c]
>>	Composition séquentielle.	P[a,b,c,d] =P1[a,b] >> P2[c,d]
[>	Préemption (*disrupt*).	P[a,b,c,d] = P1[a,b] [> P2[c,d]
;	Prefixage d'un processus par l'action a.	a ; P
stop	Processus qui ne peut communiquer avec aucun autre processus.	
Exit	Processus qui peut se terminer et se transformer ensuite en stop.	

RT-LOTOS étend LOTOS avec trois opérateurs temporels (Tableau 4.2). La combinaison d'un délai déterministe et d'un délai non déterministe permet de traiter les intervalles temporels. Le langage ainsi étendu conserve la partie « contrôle » de LOTOS, mais remplace les types de données algébriques par des implantations en C++ ou Java [Courtiat 2000].

Tableau 4.2. Opérateurs temporels de RT-LOTOS.

Opérateurs Temporel	***Description***
a{t}	Offre limitée dans le temps.
delay(t1)	Délai déterministe.
Latency(t2)	Délai non déterministe.

4.3.3 Présentation de l'environnement TURTLE

4.3.3.1 *Modélisation structurelle*

Principe

Un diagramme de classes UML permet la description d'une architecture logicielle statique en termes de classes et de relations. Ces relations sont de cinq types :

- *L'association* exprime une connexion sémantique bidirectionnelle entre deux classes. Cette relation est généralement enrichie par une définition textuelle de cette sémantique. Cette définition est fournie à titre de documentation.
- *L'agrégation* est une relation plus forte que l'association : il s'agit d'une association non symétrique, qui exprime un couplage fort et une relation de subordination.
- La *composition* est une relation d'agrégation forte qui traduit une appartenance d'une classe à une autre classe. La classe composée est responsable du cycle de vie de ses classes composantes (i.e. création, destruction).
- La *dépendance* est une relation unidirectionnelle d'obsolescence : la modification d'un élément e1 dont dépend un élément e2 peut induire la mise à jour de e2.

– Enfin, la relation d'*héritage* traduit une spécialisation, une généralisation ou une classification entre classes.

Ainsi, la relation d'association est la seule ayant une sémantique textuelle informelle. Si l'on reprend l'approche de [Douglas 1999] qui consiste à associer à chaque tâche du système une classe dite *active*[2], alors les relations d'association entre les tâches ne peuvent être qu'informelles ce qui va à l'encontre du besoin de description d'un système logiciel temps-réel en termes de contraintes relationnelles entre tâches. Ainsi, nous proposons d'enrichir la sémantique de l'association : toute relation d'association est attribuée par une classe spécifique précisant la sémantique de cette association. A cet effet, nous définissons le type abstrait *Composer* sous forme d'une classe dont les héritiers peuvent attribuer les associations. Nous définissons sept héritiers à la classe *Composer*, qui sont appelés *opérateurs de composition*. Ces opérateurs sont *Parallel*, *Synchro*, *Invocation*, *Sequence*, *Preemption*, *Suspend* et *Periodic*. Ils sont présentés et justifiés plus en détail par la suite.

Attribution d'une association par un opérateur de composition

Toute classe mise en relation d'association attribuée par un opérateur de composition est appelée *Tclass*. Ainsi, au sein du même diagramme de classes cohabitent des *Tclass* et des classes « normales ». La Figure 4-2 met en évidence deux *Tclass* mises en relation d'association, cette dernière étant attribuée par un opérateur de composition : la Figure 4-2-(a) met en évidence une relation de *parallélisme* entre les *Tclass* T1 et T2.

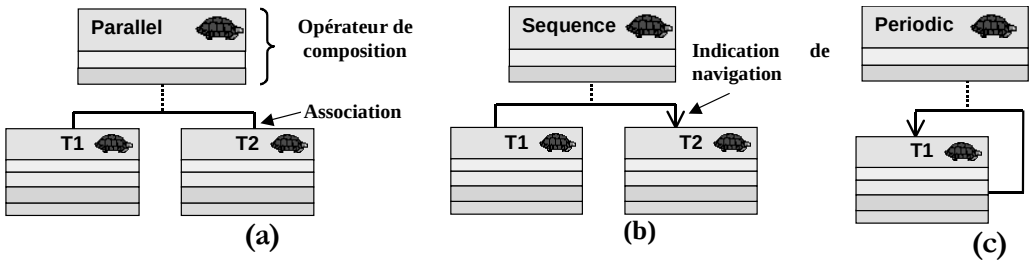


Figure 4-2. Exemple d'utilisation d'opérateurs de composition : (a) Parallel. (b) Sequence. (c) Periodic.

La Figure 4-2-(b) met en évidence une relation de *séquence* entre les *Tclass* T1 et T2. L'association entre ces deux *Tclass* comporte une indication de navigation. Cette indication de navigation est utilisée pour toute association attribuée par un opérateur de composition ayant une sémantique mono-directionnelle. Enfin, une association reliant une classe à elle-même peut être attribuée par un opérateur de composition (cf. Figure 4-2-(c) : relation de périodicité d'une *Tclass* sur elle-même).

Le type abstrait Gate

En plus des procédés classiques dont disposent habituellement les classes pour communiquer (appels de méthode, modifications d'attributs publics, etc.), les classes stéréotypées *Tclass* peuvent communiquer via des portes, un concept pour lequel nous introduisons le type abstrait *Gate* (Figure 4-3-(a)). Par défaut, une *Gate* permet des échanges d'information bidirectionnels. Deux types abstraits sont ajoutés pour rendre ces échanges mono-directionels (Figure 4-3-(b)) : on distingue les portes sur lesquelles des données peuvent être émises (*OutGate*) et les portes sur lesquelles des données peuvent être reçues (*InGate*). Par la suite, nous utilisons l'expression « une *Tclass* réalise un appel sur une *Gate* g » pour exprimer qu'une *Tclass* désire communiquer sur la porte *g* de type *Gate*.

[2] Une classe est dite *active* lorsqu'elle représente un flux d'exécution (tâche). Un objet est dit *actif* s'il est une instance d'une classe active ou lorsqu'une de ses méthodes est en cours d'appel par un objet actif.

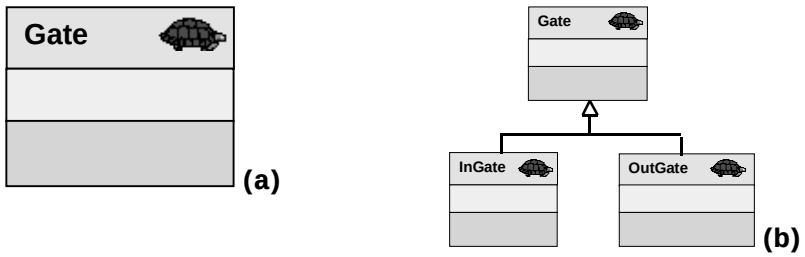


Figure 4-3. (a) Le type abstrait Gate *et (b) distinction entre* InGate *et* OutGate.

Le stéréotype Tclass

La Figure 4-4 décrit la structure générale d'un stéréotype *Tclass*. L'objectif de ce stéréotype est de distinguer les classes qui entreront en compte lors de la validation (les *Tclass*) et les classes qui seront ignorées lors de la validation (les classes normales). Cette différenciation nous conduit à imposer des propriétés d'utilisation des classes, notamment en termes de communication. Ces propriétés sont les suivantes :

- Prop1 : La validation nécessite d'avoir une visibilité totale sur les communications effectuées par les Tclass. Pour cela, nous restreignons ces communications à des transmissions synchrones au travers des portes de communication (*Gate*). Une *Tclass* peut communiquer avec des classes « normales » par appel de méthode, modification d'attributs, signaux, etc., mais ces communications sont ignorées pour la validation.

***Tclass* Id**	Identificateur de la classe de stéréotype *Tclass*.
Attributs	Liste d'attributs, excepté les attributs de type *Gate*.
Portes (*Gates*)	Liste d'attributs de type *Gate*. Ils peuvent être déclarés publics (+), privés (-), ou protégés (#).
Méthodes	Liste de méthodes, incluant un constructeur.
Description du comportement	Diagramme d'activités qui peut faire référence aux attributs, portes et méthodes précédemment définis ou hérités.

Figure 4-4. Structure d'une classe stéréotypée « Tclass ».

- Prop2 : Afin d'appréhender totalement les communications entre les Tclass, il convient aussi de restreindre les communications des classes normales vers les Tclass. A cette fin, tous les attributs des Tclass, sauf ceux de type *Gate*, doivent être déclarés privés ou protégés. De même, toutes les méthodes, sauf les constructeurs, doivent être déclarées privées (-) ou protégées (#). Seuls les constructeurs peuvent être déclarés publics.
- Prop3 : des attributs de type *Gate* peuvent être déclarés dans tout type de classe. Néanmoins, seules les *Tclass* peuvent les utiliser à des fins de communication.
- Prop4 : une *Tclass* T peut hériter d'une classe C si et seulement si C et tous ses ascendants satisfont la propriété *Prop2* : une *Tclass* ne peut donc hériter de moyens de communications autres que des portes (*Gate*).
- Prop5 : afin de valider les comportements des *Tclass*, un diagramme d'activités[3] doit accompagner chaque *Tclass*. Ce diagramme peut utiliser tout attribut ou méthode déclarés dans la *Tclass* ou hérités.
- Prop6 : afin de limiter la sémantique relationnelle entre deux *Tclass* il ne peut y avoir qu'une seule relation entre deux *Tclass*. Si cette relation est une association alors elle doit être attribuée avec une classe associative de type *Composer*. En effet, une simple

[3] Seuls les diagrammes d'activité sont pris en considération dans ce mémoire.

association n'ayant pas de sémantique formelle, la relation ne pourrait être prise en compte lors de la validation.

- Prop7 : les relations d'agrégation et de dépendance ne sont pas supportées dès lors qu'une *Tclass* est impliquée dans la relation. Ces relations nuiraient en effet à l'idée de communication restreinte aux travers des *Gate*.
- Prop8 : une classe ou une *Tclass* peut-être composée de *Tclass* ou de classes. Nous expliquerons en effet lors de la présentation du *Composer* de synchronisation que la relation de composition s'intègre dans le modèle de communication utilisant les *Gate*.

Structuration en termes de tâches logicielles

Outre un diagramme de classes explicitant les relations entre classes et le comportement de ces mêmes classes, il convient de modéliser les tâches actives au démarrage de l'application. Les tâches étant des objets à part entière, nous proposons de modéliser explicitement tous ces objets (tâches actives au démarrage) au sein d'un diagramme d'objets UML.

Opérateur de parallélisme

La modélisation du parallélisme et la synchronisation inhérente à tout système temps-réel n'est pas explicite dans UML. Cette considération nous conduit à introduire deux opérateurs de composition : *Parallel* et *Synchro*.

Deux *Tclass* actives, mises en relation par une association attribuée par l'opérateur *Parallel* sont exécutées en parallèle et sans synchronisation.

Opérateur de synchronisation

Deux *Tclass*, mises en relation par une association attribuée par l'opérateur *Synchro* réalisent des synchronisations entre elles dans deux flux d'exécution séparés. Cette synchronisation peut donner lieu à un échange de données dont le format est précisé lors de l'appel aux portes de synchronisation dans le diagramme d'activités. Si l'association entre les deux *Tclass* comporte un sens de navigation, alors l'échange de données ne peut se réaliser que dans le sens indiqué par la navigation.

Deux *Tclass* doivent se synchroniser sur deux *Gate*, qui doivent être listées dans une formule OCL (*Object Constraint Language*, cf. [OMG 2001]). Par exemple, supposons que les portes (attributs de type *Gate*) g1 et g2 de la *Tclass* T1 se synchronisent respectivement avec les portes g3 et g4 de la *Tclass* T2. Dans ce cas, la formule OCL qui accompagne la relation d'association entre T1 et T2 doit être {T1.g1 = T2.g3 et T1.g2 = T2.g4} (cf. Figure 4-5-(a)). A chaque fois qu'une activité de T1 réalise une action sur g1, cette activité doit attendre que la classe T2 réalise une action sur g3, et réciproquement. Lorsque l'action est enfin réalisée par les deux classes, l'échange de données a lieu, et les deux classes passent à l'instruction suivante de l'activité respective. A noter, que dans le cas où la porte de synchronisation porte le même identificateur dans les deux *Tclass*, alors la formule OCL peut être réduite aux seuls identificateurs de porte. Par exemple si les *Tclass* T1 et T2 se synchronisent sur la portes g1, avec g1 porte de T1 et T2 (la porte est distincte mais porte le même identificateur), alors la formule OCL peut être réduite *à* {g1}. De même, dans toute formule OCL de type $\{T_i.g_{ti} = T_j.g_{tj}\}$, si aucune porte de T_j ne porte l'identificateur « g_{ti} » alors T_i peut être supprimé de la formule OCL.

La Prop8 sur les *Tclass* précise que la relation de composition est possible entre deux *Tclass*. Nous allons à présent expliciter comment une Tclass peut communiquer avec les composants d'une autre *Tclass*. Supposons que la *Tclass* T1 soit composée de la *Tclass* T2 et que T1 possède une porte publique g1 et T2 une porte publique g2. Une *Tclass* T3 qui veut communiquer via sa porte g3 avec T2 via g2 a deux solutions : soit T3 communique directement avec T2 sur la porte g2, soit T3 communique avec T1 sur la porte g1, à la condition qu'une formule OCL associe g1 et g2. Cette formule doit être attachée à la relation de composition entre T1 et T2. Elle doit alors être de la forme : {T1.g1 = T2.g2} (cf. Figure 4-5-(b)).

Enfin, précisons qu'une porte de synchronisation d'une *Tclass* T peut-être utilisée dans plusieurs relations de synchronisation concernant T.

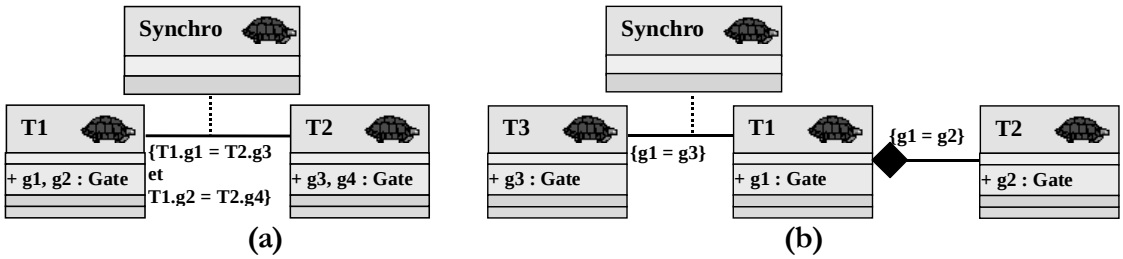


Figure 4-5. Utilisation de l'opérateur Synchro.*(a) Relation simple. (b) Combiné avec une relation de composition.*

Opérateur d'appel de méthode (Invocation)

L'opérateur *Synchro* permet de modéliser un échange de données entre deux flux d'exécution distincts mais ne permet pas de modéliser une communication dans un même flux d'exécution. A cette fin, nous proposons d'introduire l'opérateur *Invocation*.

Considérons deux *Tclass*, T1 et T2, mises en relation par une association dirigée de T1 à T2 et attribuée par la classe associative *Invocation* (cf. Figure 4-6). On dit alors que T2 peut-être invoquée par T1.

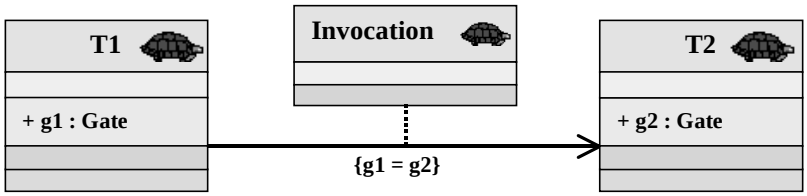


Figure 4-6. Utilisation de l'opérateur Invocation.

Tout comme dans la relation de synchronisation, une porte de chaque *Tclass* doit être impliquée dans chaque invocation (une *porte d'invocation* et une *porte invoquée*). Soit g1 (resp. g2) une porte de T1 (resp. T2). Considérons que la formule OCL suivante est associée à la relation : {*T1.g1 = T2.g2*}. Alors, quand une activité de T1 réalise un appel sur g1, cette activité doit attendre que T2 réalise un appel sur g2. Quand T2 réalise cet appel, les données sont échangées uniquement dans le sens de la navigation (i.e. de T1 à T2). L'activité de T1 réalisant l'appel est bloquée jusqu'à ce que T2 réalise de nouveau un appel sur g2. Dans ce deuxième cas, un échange de données peut avoir lieu dans le sens contraire à celui indiqué par la navigation. Ainsi, le code de T2 qui est invoqué par T1 est exécuté dans le flux d'exécution de T1. Enfin, précisons qu'une porte ne peut être impliquée que dans une et une seule invocation.

Opérateur de séquence

Les trois opérateurs précédents permettent de décrire la structure de communication entre les tâches mais en aucun cas l'aspect dynamique des tâches : démarrage et arrêt. Les opérateurs suivants permettent de décrire cela.

Les deux *Tclass,* mises en relation par une association à laquelle est attribué l'opérateur *Sequence*, sont exécutées l'une après l'autre, dans le sens donné par la navigation de l'association. Dans la relation (T1 Sequence T2), T1 doit se terminer[4] avant que T2 ne démarre. T2 est exécutée dans un nouveau flux d'exécution si T2 est une classe active.

[4] On dit qu'une *Tclass* se termine lorsque toutes les activités associées à cette classe ont atteint leur point de terminaison.

Opérateur de préemption

L'opérateur de séquence nécessite la terminaison d'une tâche avant d'exécuter en séquence une nouvelle tâche. L'opérateur de *Preemption* entend résoudre cette limite.

La *Tclass* T2, désignée par la navigation de l'association reliant deux *Tclass* T1 et T2, et dont l'opérateur de composition associé est *Preemption*, peut interrompre définitivement et à n'importe quel instant l'autre *Tclass*. Cette interruption se produit lorsque la première instruction de T2 est réalisable (synchronisation sur une porte par exemple). Les deux *Tclass* doivent être des classes actives.

Opérateur de suspension

L'opérateur *Preemption* interrompt définitivement le comportement d'une tâche ce qui nuit à son utilisation pour modéliser un ordonnancement de tâche. En effet, un ordonnanceur réalise une élection de tâche qui a pour effet de suspendre ou activer provisoirement des tâches, sans perte de leur environnement d'exécution. Nous résolvons ce problème par la définition d'un opérateur de suspension des *Tclass*.

La *Tclass* T2, désignée par la navigation de l'association reliant deux *Tclass* T1 et T2, et dont l'opérateur de composition associé est *Suspend*, peut être suspendue par l'autre *Tclass T1,* à condition que T2 soit une classe active (cf. Figure 4-7).

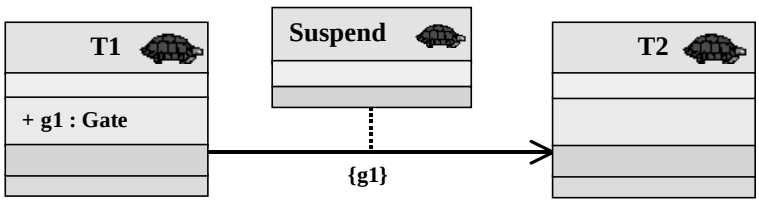


Figure 4-7. Utilisation de l'opérateur Suspend.

Cette suspension se produit suite à la réalisation par T1 d'un appel sur une porte de synchronisation précisée dans la formule OCL accompagnant l'association. Soit g cette porte de synchronisation : T2 peut être suspendue à n'importe quel instant, sauf lorsque T2 est en train de réaliser un appel sur une porte d'invocation : l'appel sur g est alors bloquant pour T1 tant que la suspension n'est pas effective i.e. tant que l'invocation réalisée par T2 est active. En effet, il ne paraît pas souhaitable d'interrompre une *Tclass* réalisant une invocation car cette dernière correspond à un appel de méthode : afin de respecter un modèle objet réaliste, la suspension d'une classe doit suspendre l'ensemble de la pile d'appel relative à cette classe. Si un opérateur de suspension désigne une classe particulière, cet opérateur ne s'applique qu'à la classe concernée et ne peut donc suspendre que la classe impliquée dans la relation : pour suspendre l'ensemble d'une pile d'appel, il convient de suspendre explicitement l'ensemble des objets impliqués, et non uniquement un objet impliqué dans la pile d'appel.

Opérateur de périodicité

Les opérateurs de composition précédents font totalement abstraction de l'aspect temps-réel inhérent aux tâches des systèmes temps-réel. Si par la suite, nous introduisons des opérateurs de comportement temps-réel au niveau des diagrammes d'activité, nous pensons que la périodicité des tâches fait partie des comportements de haut niveau du système. Ainsi, nous proposons un opérateur de composition de périodicité.

Une relation d'association entre deux *Tclass* actives T1 et T2 et attribuée par l'opérateur *Periodic* (cf. Figure 4-8) signifie que pour chaque instance de T1, une nouvelle instance de T2 est démarrée périodiquement au moins une fois, selon la période précisée dans une formule OCL (mot clé *period*). Si le mot clé *deadline* est présent dans la formule OCL, alors la nouvelle instance de T2 devra avoir terminé son exécution avant la valeur associée à *deadline*. Si l'opérateur *Periodic* est

attribué à une association d'une classe sur elle-même (cf. Figure 4-2, page 51), alors cette classe voit son comportement (décrit par son diagramme d'activité, cf. 4.3.3.2) exécuté de façon périodique.

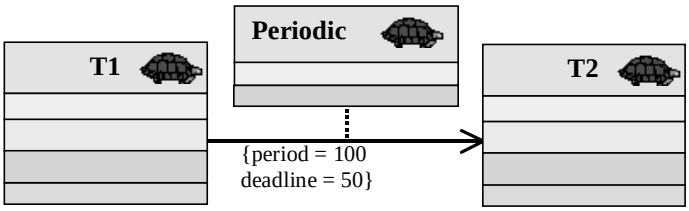


Figure 4-8. Opérateur de composition Periodic.

4.3.3.2 *Modélisation comportementale*

Deux diagrammes UML permettent la description du comportement d'une classe : le diagramme d'états (*state diagram*) et le diagramme d'activités (*activity diagram*). A ce jour, l'environnement TURTLE ne permet la description du comportement interne des *Tclass* qu'au travers des diagrammes d'activités. Notre choix de n'aborder dans un premier temps que ce diagramme est motivé par le fait que la plupart des outils "UML temps-réel" à l'exemple de *Rhapsody* [Douglass 1999] autorise la description du comportement des classes avec les deux types de diagrammes.

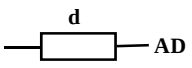
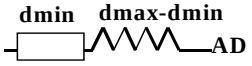
Opérateurs logiques

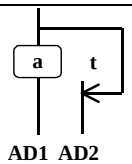
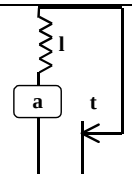
Pour toutes les constructions des diagrammes d'activités [Douglas 1999], nous avons proposé une sémantique formelle donnée par la traduction en RT-LOTOS (cf. Tableau 8.1, Annexe A, page 133) : nous gérons ainsi la traduction des boucles, du choix, du parallélisme, de la synchronisation et des états de démarrage et de fin des activités. Pour les communications avec les autres *Tclass*, le pictogramme correspondant aux actions a été utilisé pour décrire les appels sur les attributs de type *Gate* (communications avec les autres *Tclass*).

Opérateurs temporels

L'impossibilité de modéliser avec les diagrammes de comportement UML des contraintes temporelles non fixes nous incite à proposer une approche plus générale pour la gestion du temps par l'introduction de trois opérateurs temporels fondamentaux : un délai déterministe, un délai non déterministe et une offre limitée dans le temps. Le Tableau 4.3 présente les trois pictogrammes associés à ces trois opérateurs. A ces trois opérateurs s'ajoute un opérateur de délai applicable à un intervalle temporel, cumulant ainsi les effets d'une durée fixe égale à la borne inférieure de l'intervalle et une latence égale à la différence entre les bornes supérieure et inférieure de l'intervalle. La sémantique de ces opérateurs est donnée par transposition en RT-LOTOS [Courtiat 2000]. Soit AD la dénomination pour un diagramme d'activités, et τ(AD) le processus RT-LOTOS correspondant à la transcription de AD.

Tableau 4.3.Opérateurs temporels de TURTLE.

Opérateurs temporels TURTLE	***Description***	***Traduction RT-LOTOS***
	*Retard déterministe. AD est interprété après d unités de temps.*	***delay****(d)* τ*(AD)*
	*Retard non déterministe. AD est interprété après au plus t unités de temps.*	***latency****(t)* τ*(AD)*
	*Délai non déterministe. AD est interprété après au moins dmin et au plus dmax unités de temps.*	***delay****(dmin,dmax)* τ*(AD)*

	*Offre limitée dans le temps. L'offre sur la* Gate ***a*** *est offerte pendant une période inférieure ou égale à* ***t****. Si l'offre est réalisée, alors AD1 est interprété, sinon AD2 est interprété.*	***a****{t, τ(AD2)} ; τ(AD1)*
	*Offre limitée dans le temps. L'offre sur la* Gate ***a*** *est offert pendant une période inférieure ou égale à* ***t****. A noter que la latence commence en même temps que la limitation sur l'offre. Si l'offre est réalisée, alors AD1 est interprété, sinon AD2 est interprété.*	***latency****(l)* ***a****{t, τ(AD2)} ; τ(AD1)*

Les opérateurs temporels précédents représentent une durée qui s'écoule indépendamment du fait que la tâche à laquelle ils appartiennent soit en cours d'exécution ou non. Il nous paraît intéressant dans les systèmes temps-réel d'offrir des opérateurs temporels qui soient stoppés lorsque la tâche à laquelle ils appartiennent n'est pas en cours d'exécution afin par exemple de modéliser un temps de traitement algorithmique. A cette fin, nous proposons des opérateurs temporels dit *interruptibles* c'est-à-dire dont la valeur temporelle est suspendue lorsque la *Tclass* relative à cet opérateur est suspendue (opérateur de composition *Suspend*). Graphiquement, ces opérateurs se distinguent des opérateurs temporels classiques par l'introduction d'un sablier. La Figure 4-9 met en évidence la représentation graphique (a) du délai déterministe, (b) du délai non déterministe et (c), de l'offre limitée dans le temps.

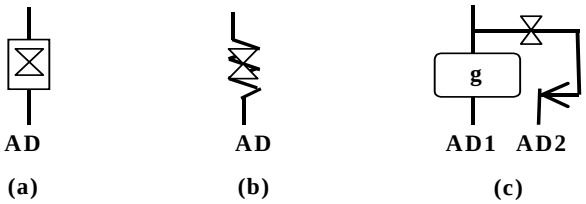


Figure 4-9. Représentation graphique des opérateurs temporels interruptibles TURTLE. (a) Délai déterministe. (b) Délai non déterministe. (c) Offre limitée dans le temps.

4.3.3.3 *Processus de validation formelle*

Ce paragraphe décrit les étapes du processus de validation TURTLE (cf. Figure 4-10).

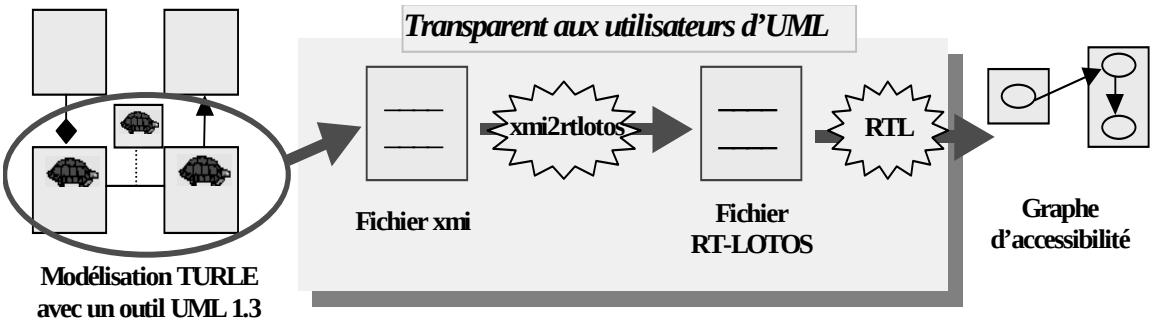


Figure 4-10. Processus de validation : du modèle TURTLE au graphe d'accessibilité.

Tout d'abord, les *Tclass* et les relations qui les lient sont extraites des diagrammes UML et sauvegardées dans un format d'échange de modèle UML, par exemple au format XMI [OMG 2002a]. Ce format intermédiaire est ensuite converti en RT-LOTOS par un processus de traduction que nous décrivons au paragraphe 4.3.3.4. A des fins d'analyse, un graphe d'accessibilité est obtenu par utilisation de l'outil RTL, outil développé au LAAS [Courtiat 2000]. L'exploitation de ce graphe

est décrite au paragraphe 4.3.3.5. Bien entendu, ce graphe est généré si et seulement si le système considéré est fini. Sinon, l'analyse se limite à une simulation c'est-à-dire à une exploration partielle des états du système.

Il convient de noter que les modèles TURTLE peuvent être réalisés à partir des ateliers logiciel supportant UML 1.3 et la sauvegarde des diagrammes UML au format XMI [Saqui 2001].

4.3.3.4 *Processus de traduction d'une modélisation TURTLE en une spécification RT-LOTOS*

Principe général de la traduction

Le processus de traduction d'un modèle UML en RT-LOTOS s'applique uniquement aux *Tclass*, aux relations entre les *Tclass* et aux comportements des *Tclass*.

Le profil TURTLE présenté dans ce mémoire correspond en fait à un profil TURTLE *standard* spécialisé pour la modélisation des systèmes temps-réel (cf. Figure 4-11). Cette spécialisation se traduit par l'introduction d'opérateurs de composition (*Suspend*, *Periodic, Invocation*) et de comportement (opérateurs temporels interruptibles). D'autres profils TURTLE adaptés à la modélisation de protocoles ou de documents multimédias sont actuellement à l'étude. Les algorithmes de traduction proposés tiennent compte de cette organisation sous forme de profils : chaque profil spécialisé intègre des algorithmes de traduction vers une modélisation TURTLE standard. Il existe en outre des algorithmes de traduction d'une modélisation TURTLE standard vers une spécification RT-LOTOS. Ces choix de traduction permettent de réutiliser les algorithmes élaborés pour le profil TURTLE standard, et implémentés dans une maquette logicielle.

Comme il n'est pas possible d'exposer les algorithmes de traduction, nous présentons par la suite l'approche suivie et invitons le lecteur à consulter [Lohr 2002].

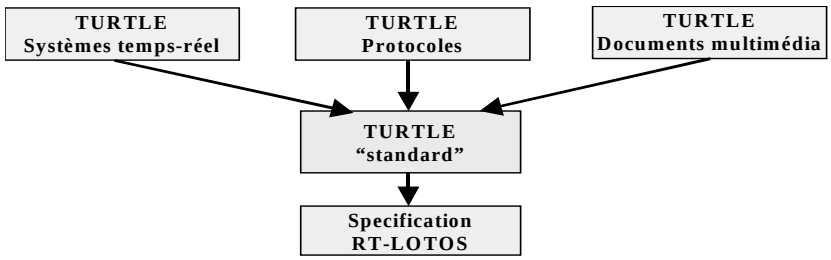


Figure 4-11. Les profils TURTLE.

Traduction du profil TURTLE temps-réel en TURTLE standard

L'algorithme de traduction consiste à générer une nouvelle modélisation TURTLE qui ne comprend ni les opérateurs de composition *Suspend*, *Periodic, Invocation* ni les opérateurs de comportement suivants : les portes d'invocation et les opérateurs temporels interruptibles (cf. Figure 4-13, étape (1)).

- l'opérateur *Suspend* se traduit par une synchronisation sur une porte spécifique. Le comportement de la *Tclass* soumise à la suspension est modifié de telle sorte que lorsque cette synchronisation est effectuée, le comportement de la classe est interrompu, sauf dans le cas d'opérateurs temporels non interruptibles. Les opérateurs interruptibles sont alors traduits par des opérateurs non interruptibles mais pendant lesquels la synchronisation sur la porte de suspension est offerte.
- l'opérateur *Periodic* se traduit par la modification du comportement de la Tclass dont l'opérateur de périodicité est issu : son comportement modélise le démarrage périodique de la *Tclass* pointée par la relation de périodicité.
- Enfin, l'opérateur d'*Invocation* est remplacé par une relation de synchronisation et chaque porte d'invocation *g !x ?y* (cf. Annexe A) est remplacée par l'enchaînement en série des portes de synchronisation *g !x* et *g ?y*. En effet, l'invocation étant assimilée à

un appel de méthode du paradigme objet, l'appel de méthode s'accompagne d'une émission de données (*g !x*) puis d'une attente de retour de paramètres (*g ?y*).

Traduction de TURTLE standard en RT-LOTOS

Une spécification RT-LOTOS est organisée en deux parties distinctes : une déclaration des relations entre processus et une description du corps des processus. Ces derniers processus correspondent soit aux processus listés dans la partie de déclaration soit aux processus qui sont déclenchés dans le corps d'autres processus. L'idée générale est de traduire les relations entre les *Tclass* dans la partie de déclaration des relations entre processus, en associant à chaque *Tclass* T_i un processus RT-LOTOS p_i. Ensuite, le comportement de chaque processus p_i représente le comportement de la Tclass T_i. Cependant, nous allons le voir par la suite, ce modèle simpliste n'est pas directement applicable au regard des opérateurs de composition de *Tclass*.

Si la traduction des éléments graphiques de description du comportement ne pose que peu de problème en RT-LOTOS (cf. Tableau 4.3 et Tableau 9.1), en revanche, la traduction de la structure (relations entre classes) est une tâche beaucoup plus complexe, dans la mesure où LOTOS ne propose que des relations binaires entre processus, alors qu'une *Tclass* peut-être en relation directe avec un nombre quelconque de *Tclass*. Considérons à titre d'exemple une *Tclass* T1 en relation de séquence avec une *Tclass* T2, et en relation de préemption avec une *Tclass* T3 (cf. Figure 4-12).

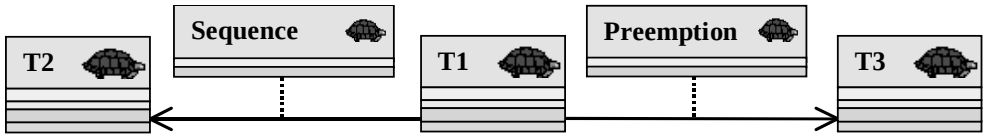


Figure 4-12. Relation de Séquence et de Preemption respectivement entre T1 et T2, et entre T1 et T3.

Soit p1 (resp. p2 et p3) le processus RT-LOTOS associé à T1 (resp. T2 et T3). Cette modélisation signifie que si T1 se termine, alors T2 est exécutée et que si T3 démarre alors que T1 n'a pas fini son exécution alors T1 est arrêtée et T3 est exécutée. En utilisant les relations binaires RT-LOTOS, en mappant la relation *Sequence* sur la séquence LOTOS (>>) et la relation *Preemption* sur le *disrupt* LOTOS ([>), et enfin en associant un processus LOTOS à chaque Tclass (p_i associé à T_i), il existe trois traductions possibles :

(1) *(p1 >> p2) [> p3*. Cette traduction signifie que p2 est exécuté en séquence après p1, que p1 peut être préempté par p3, mais aussi que p2 peut être préempté par p3 ce qui est faux.

(2) *(p1 [> p3) >> p2* . Cette traduction signifie que p2 est exécuté en séquence après p1, mais aussi après p3, ce qui est faux.

(3) *(p1 >> p2) [] (p1 [> p3)*. Cette traduction signifie soit que p2 est exécuté en séquence après p1, soit que p1 est préempté par p3. Cette traduction semble correcte à ceci près que le choix entre la séquence et la préemption est déterminé non pas lors de l'exécution des processus, mais par un choix non déterministe avant l'exécution de p1.

Cet exemple met ainsi en évidence qu'il n'est pas possible d'associer dans le cas général un opérateur de composition TURTLE à un opérateur de composition de processus RT-LOTOS.

Afin de résoudre le problème de la traduction simultanée des opérateurs de composition, nous proposons un algorithme basé sur la notion de priorité opératoire. Si l'on note *Pri*(oc), la priorité de l'opérateur de composition oc, avec oc ∈ {*Parallel*, *Synchro*, *Sequence*, *Preemption*}, on pose :

Pri(Preemption) > *Pri*(Sequence) > *Pri*(Synchro) > *Pri*(Parallel).

Finalement, l'algorithme qui tient compte de ces priorités est identifié par les étapes (2) et (3) de la Figure 4-13. L'étape (2) consiste à construire le corps des processus LOTOS à partir des opérateurs composition et du comportement des *Tclass* (cf. [Lohr 2002]), et l'étape (3) de la Figure 4-13 qui consiste à générer la déclaration des processus LOTOS. Cette déclaration des processus

correspond aux processus initiaux du système. Par conséquent, ces processus correspondent aux instances initiales des *Tclass* qui peuvent être obtenues dans le diagramme d'objets de la modélisation TURTLE.

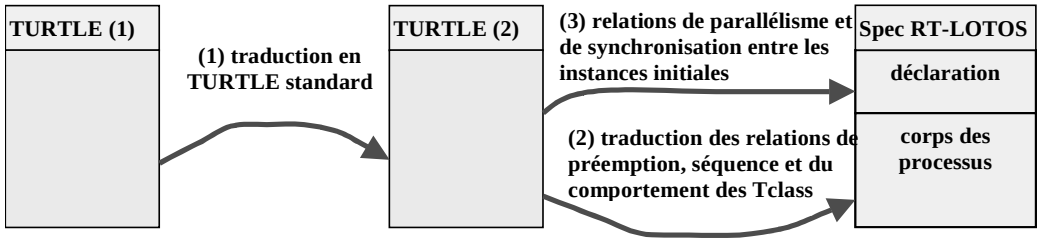


Figure 4-13. Etapes de l'algorithme de traduction des diagrammes TURTLE en RT-LOTOS.

4.3.3.5 *Graphes d'accessibilité*

Une fois la spécification RT-LOTOS obtenue suite à l'application des algorithmes de traductions sur les diagrammes TURTLE, l'outil RTL permet de générer un graphe d'accessibilité (cf. 4.3.3.3) à condition que le système considéré soit fini.

Description

Un graphe d'accessibilité est un système d'états et de transitions étiquetées comportant au moins un état initial et des états qui sont tous accessibles depuis l'état initial par un chemin fini. Ces graphes représentent l'évolution logique d'un système c'est-à-dire tous les comportements logiques possibles d'un système depuis un état initial donné. Un graphe d'accessibilité est en général construit de telle façon que son nombre d'états et de transitions est minimal. La Figure 4-14-(a) met en évidence un graphe d'accessibilité comportant un état initial (état 0) et deux états (état 1 et état 2) accessibles depuis l'état 0 respectivement à partir des transitions étiquetées par *a* et *b*. Un tel graphe permet de mettre en évidence des transitions logiques entre états, mais ne met pas en évidence le comportement temporel du système. Ce problème est résolu par utilisation d'automates temporisés dont les états de contrôle sont constitués de sous-états temporels mettant en évidence les régions temporelles depuis lesquelles les transitions logiques peuvent être tirées. Considérons par exemple le graphe d'accessibilité de la Figure 4-14-(b). Ce graphe possède trois états de contrôle (états 0 qui est l'état initial, état 1 et état 2). Depuis l'état initial, il est possible, lors des 50 premières unités de temps, de tirer la transition étiquetée « a », ce qui conduit le système dans l'état 1. Dans l'état 0, lorsque 50 unités de temps se sont écoulées, une transition « t » amène le système dans une nouvelle région temporelle (même état logique), depuis laquelle la transition « a » ne peut plus être tirée, alors que la transition « b » qui amène le système dans l'état 2 peut être tirée.

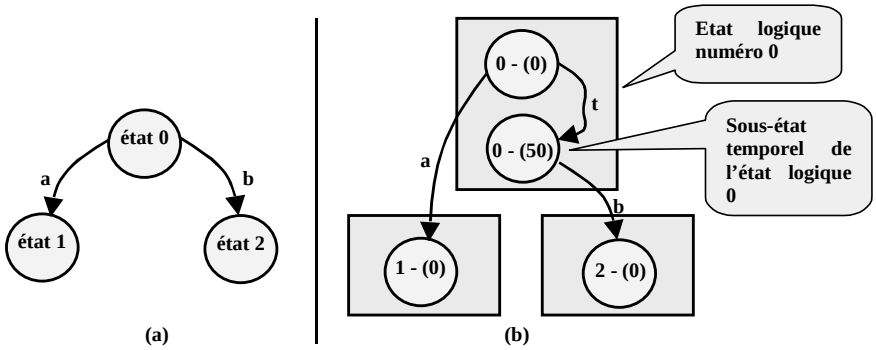


Figure 4-14. Exemple de graphe d'accessibilité. (a) Graphe non temporisé. (b) Graphe temporisé.

L'outil RTL permet d'obtenir l'automate temporisé associé à une spécification RT-LOTOS. Au niveau de chaque état de l'automate figure l'identificateur de l'état ainsi qu'un vecteur représentant la valeur des horloges internes de tous les processus RT-LOTOS actifs. Dans

l'exemple considéré (cf. Figure 4-12-(b)), il n'y a qu'un processus par état de contrôle et donc qu'une horloge.

Analyse

Une transition logique sur un graphe généré à partir de l'outil RTL correspond à un appel sur une porte de synchronisation. Une transition temporelle sur ce même graphe correspond à une évolution du temps (opérateur temporel). L'analyse du graphe d'accessibilité s'apparente donc à l'identification de transitions spécifiques et à l'identification des états puits. La remontée au modèle TURTLE est facilitée par une table de correspondance générée par les algorithmes de traduction et qui associe à chaque étiquette d'une transition logique du graphe une porte TURTLE [Lohr 2002].

4.4. TURTLE comme environnement support de modélisation pour la reconfiguration dynamique

4.4.1 Introduction

Ce chapitre entend proposer une solution au problème de la validation a priori de la reconfiguration dynamique d'une application avec respect de contraintes applicatives, notamment de contraintes de continuité de service [Gupta 1996]. Dans un premier temps, nous avons défini une architecture intrinsèquement reconfigurable et compatible avec les environnements spatiaux. Dans un deuxième temps, nous avons proposé un cadre formel de modélisation basé sur le langage UML afin de se conformer à la méthodologie Alcatel. L'objectif de cette section est de montrer comment par utilisation de ce cadre formel, il est possible de modéliser une application développée selon l'architecture introduite dans la première section et de valider formellement l'exécution d'ordres de reconfiguration dynamique sur une application au regard du respect de contraintes inhérentes à cette application.

A cette fin, nous proposons de nous inspirer des contributions en termes de modélisation et de validation du domaine des ADLs, et tout particulièrement de l'ADL Wright [Allen 1997] qui permet la modélisation d'une architecture de haut niveau et la validation d'une reconfiguration au regard de contraintes applicatives. Cependant, nous ne pouvons que nous inspirer de la philosophie de modélisation et de validation et non réutiliser directement ce langage (et plus généralement les autres ADLs) pour trois raisons :

- une modélisation TURTLE peut être réalisée avec un modeleur UML [Saqui 2001] et plus particulièrement avec l'atelier UML en vigueur chez Alcatel (cf. paragraphe 2.3.3.1), contrairement à l'ADL *Wright*.
- l'absence de prise en compte dans les ADLs des contraintes intrinsèques temps-réel et extrinsèques ;
- le processus de validation formelle est limité à l'examen de la consistance logique d'interconnexion (cf. [Allen 1998]).

Par la suite, nous montrons dans un premier temps le principe général de l'utilisation de TURTLE comme un ADL. Puis, dans un deuxième temps, nous montrons l'adéquation de TURTLE avec la modélisation de l'architecture de haut niveau présentée dans la section 4.2. Enfin, dans un troisième temps, nous exposons la validation a priori d'ordres de reconfiguration dynamique.

4.4.2 Principe général

Nous présentons dans cette partie le principe général de notre approche d'utilisation de TURTLE comme un ADL en termes de modélisation et de validation.

4.4.2.1 *Modélisation*

Nous décrivons dans ce paragraphe le principe général d'utilisation de TURTLE comme un ADL. [Allen 1997] souligne l'importance pour un langage de type ADL de permettre la modélisation :

- des composants logiciels. Pour [Allen 1997], cette modélisation comprend la description du comportement des composants et de leurs points de reconfiguration ;
- du schéma d'interconnexion. A ce sujet, [Allen 1998] précise que le schéma d'interconnexion doit être mis en évidence de façon claire par l'ADL ;
- des contraintes intrinsèques et notamment la consistance logique d'interconnexion et les ressources applicatives ;
- et enfin, de la configuration logicielle et éventuellement les aspects dynamiques de la configuration à l'instar du *Configuror* [Allen 1998].

Nous montrons par la suite comment l'environnement TURTLE est apte à modéliser ces quatre points, et comment nous remédions à la limite de modélisation des ADL en termes de modélisation et de validation de contraintes.

En ce qui concerne la modélisation de l'architecture, nous proposons d'associer une *Tclass* par module du système et de modéliser le comportement des modules (ce qui inclut le comportement des points de reconfiguration) par description du comportement des *Tclass*. En effet, une *Tclass* correspond au découpage modulaire de notre environnement TURTLE. De plus, le modèle de communication des *Tclass* permet de modéliser le schéma de communication de l'architecture en associant les ports de sortie des modules à des attributs de type *OutGate* et les ports d'entrée des modules à des attributs de type *InGate*. Nous proposons de modéliser les liens entre modules par une *Tclass* de *Routage* et par une *Tclass* représentant le buffer du port d'entrée du lien. Nous le verrons par la suite, la classe de routage permet de ne pas figer la description du schéma d'interconnexion et de modéliser les caractéristiques des liens de communication (par ex., délai de transmission).

En termes de modélisation des contraintes applicatives, nous proposons l'utilisation d'observateurs [Jard 1988]. Nos observateurs sont des classes externes dont nous assurons par construction la non intrusivité sur l'architecture modélisée.

Enfin, reprenant l'approche de [Allen 1997] d'introduire un *Configurator* qui décrit la configuration applicative et les modifications concernant cette configuration, nous proposons d'introduire une *Tclass* appelée *GestConf* qui gère la configuration applicative initiale et l'exécution d'un script de reconfiguration sur cette configuration initiale.

4.4.2.2 *Validation*

L'ADL Wright est basé sur CSP (*Communicating Sequential Processes*, cf. [Hoare 1985]) afin de valider formellement les contraintes applicatives [Allen 1997]. Selon la même approche, nous proposons de nous appuyer à des fins de validation formelle sur le processus de validation formelle de l'environnement TURTLE (cf. 4.3.3.3). Pour cela, nous proposons de regrouper dans la même spécification TURTLE les Tclass d'architectures, les Tclass d'observation de propriétés et enfin le gestionnaire de configuration, puis d'appliquer à l'ensemble de la modélisation des Tclass le processus de validation formelle de TURTLE.

4.4.3 Modélisation des concepts architecturaux

Cette section met en évidence la modélisation TURTLE de l'architecture logicielle (modules, ports, liens).

4.4.3.1 *Modélisation des modules*

Les concepts associés aux modules sont la notion de tâche logicielle, d'agrégat d'objets, de port de communication, d'état interne et externe, et de comportement (cf. 4.2.2). Nous avons

expliqué dans le paragraphe 4.4.2.1 l'intérêt de modéliser un module par une *Tclass*. Par la suite, nous montrons comment les concepts associés à un module et que nous venons de rappeler sont modélisés au sein de la *Tclass* de ce module (cf. Figure 4-15) :

- Tâche logicielle : la *Tclass* est définie comme une classe active du système ;
- Agrégat d'objets : toute classe non active d'un module ne peut pas être une *Tclass*. D'après les propriétés relatives aux *Tclass* (cf. 4.3.3), cela restreint de façon naturelle les communications entre modules aux seuls attributs de type *Gate*.
- Ports de communication : les ports de communication d'entrée et de sortie sont modélisés par des attributs respectivement de type *InGate* et *OutGate* (cf. 4.4.2.1)
- Etat interne : l'état interne d'un module est représenté par l'état des attributs de la Tclass représentant le module, et de tous les états internes des objets en relation de composition avec cette *Tclass*.
- Etat externe : nous reprenons l'approche d'[Allen 1997] qui consiste à modéliser les points de reconfiguration. Pour cela, nous proposons d'introduire une classe abstraite *Module* dont héritent les *Tclass* des modules et qui définit deux attributs de type *Gate* (*arrêt* et *active*). L'attribut *arrêt* représente une demande de l'environnement pour que le module rejoigne son point de reconfiguration. Dans le même esprit, l'attribut *active* représente une demande d'activation. Nous utilisons en outre cet attribut pour représenter le passage de l'état *créé* à l'état *actif* (cf. 4.2.2.2).
- Comportement du module : un module étant modélisé par une *Tclass*, nous proposons de modéliser le comportement du module au niveau du diagramme d'activités de cette *Tclass*. La modélisation du comportement doit décrire comment les demandes de la part de l'environnement de passer dans l'état *suspendu* (point de reconfiguration) sont traitées. Ces demandes se traduisent nécessairement par une offre sur la porte *active*. Une modélisation générique de ce comportement est présentée à la Figure 4-15. Nous reviendrons ultérieurement (cf.4.4.3.4) sur le principe de modélisation des différents types de comportement des modules (cf. paragraphe 4.2.2.2).

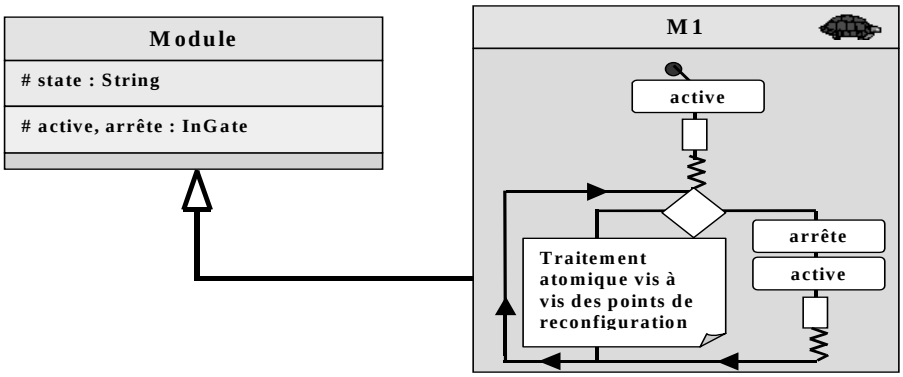


Figure 4-15. Modélisation générique d'un module en TURTLE.

4.4.3.2 *Modélisations des ports*

Les ports des modules sont modélisés à l'aide du type abstrait *Gate* défini dans l'environnement TURTLE. Nous distinguons par la suite la modélisation des ports de sortie et des ports d'entrée. La Tclass d'un module doit alors déclarer autant d'attributs de type *InGate* (resp. *OutGate*) qu'elle possède de ports d'entrée (resp. de ports de sortie).

Modélisation des buffers

Il convient de modéliser le buffer de réception associé aux ports d'entrée (cf. 4.2.2.2). Nous introduisons pour cela une classe *BufferPort*. La classe *BufferPort* ne représente pas une activité (tâche) du système logiciel. Nous utilisons donc pour représenter les trois opérations de cette classe des

portes d'invocation TURTLE (cf. Figure 4-16) : la porte *depot* (opération de dépôt d'un message dans le buffer), la porte *lecture* (opération de lecture d'un message dans le buffer), et enfin, la porte *nombre* (opération de récupération du nombre de messages actuellement dans le buffer). Dans le paradigme objet, les appels de méthode sur un objet donné depuis plusieurs activités distinctes peuvent se réaliser en parallèle. Toutefois, sur un objet de type BufferPort, une seule méthode doit pouvoir être appelée simultanément pour assurer la cohérence logique des données (problème de la synchronisation dans un système producteur / consommateur). Nous modélisons cette synchronisation d'appel par l'utilisation d'opérateurs de choix exclusif entre les différentes portes des trois opérations fondamentales du buffer.

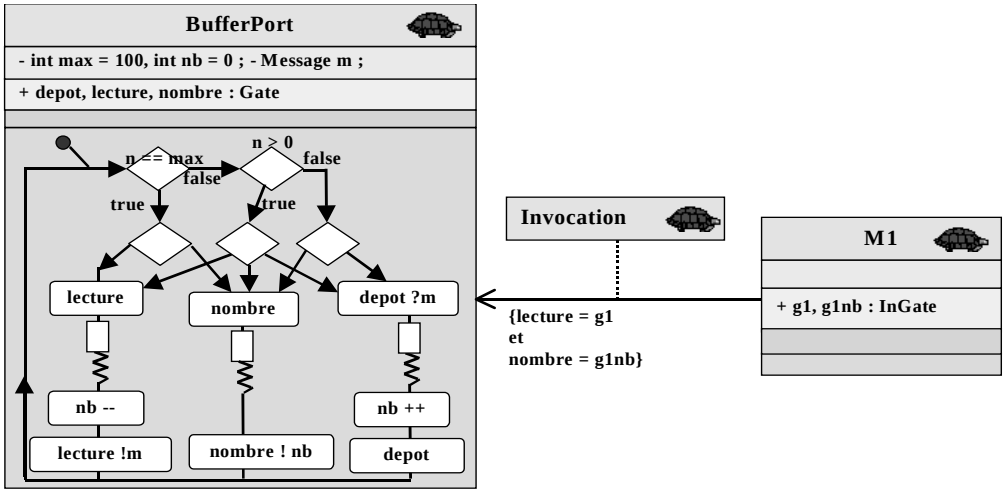


Figure 4-16. Modélisation TURTLE d'un buffer et de l'association entre le port d'entrée d'un module et le buffer de ce port d'entrée.

Le cas nominal de traitement effectué par le buffer correspond au cas où le buffer n'est ni plein ni vide. Lorsque ces deux conditions sont réalisées, les trois opérations sont accessibles. Leur temps de traitement est modélisé pour chacune d'elle par des intervalles temporels (valeurs obtenues en simulation). Les trois portes étant des portes invoquées (au sens TURTLE), la classe *BufferPort* utilise un premier appel à ces portes (avec réception éventuelle des données), modélise le temps de traitement, puis rappelle la même porte avec émission potentielle de données (cas de la lecture du nombre de message et de la lecture d'un message). Dans le cas particulier où le buffer est plein (cas où n == max vaut *true*), l'opération de dépôt d'un message est inaccessible. De même, lorsque le buffer est vide (cas où n > 0 vaut *false*), l'opération de lecture d'une donnée est inaccessible.

La modélisation proposée ne tient pas compte du type de message transmis et des valeurs contenues dans le message. Si les valeurs des messages sont pertinentes au regard du fonctionnement du système logiciel, elles doivent être ajoutées à la modélisation.

Association entre port d'entrée et buffer

A chaque port d'entrée p est associé un buffer b (cf. 4.2.2.2). Un buffer n'étant pas une classe active du système, nous modélisons la relation entre p et b par une relation d'invocation. Le port d'entrée étant modélisé par un attribut de type *Gate* d'une *Tclass* et le *buffer* étant aussi une *Tclass* qui offre ses opérations de lecture au travers de deux attributs de type *Gate*, la relation d'*Invocation* entre p et b associe la porte du port d'entrée de la *Tclass* du module avec l'opération de lecture d'un message. La Figure 4-16 met en évidence la relation d'invocation entre la *Tclass* d'un module et un buffer. Les portes du côté modules sont nommées *g1* et *g1nb* et sont en relation respectivement avec *lecture* et *nombre* de la classe *BufferPort*. Chaque appel dans *M1* sur *g1* déclenche une invocation sur la porte *lecture* de *BufferPort*. De même chaque appel sur *g1nb* déclenche une invocation sur la porte *nombre* de *BufferPort*.

4.4.3.3 *Modélisation des liens*

Les ADL, à l'exemple de *Wright* [Allen 1997], permettent de modéliser les connecteurs i.e. l'interconnexion entre les points de connexion des composants (cf. 3.4.8.1). Il s'agit donc pour nous de modéliser les *liens* de notre architecture avec l'environnement TURTLE c'est-à-dire la communication des messages entre le port de sortie d'un lien et le buffer du port d'entrée du même lien. A l'image de la solution de modélisation proposée pour la communication entre un port d'entrée et son buffer, il pourrait être intéressant d'utiliser une relation *d'Invocation* entre l'attribut de type *OutGate* représentant le port de sortie et l'attribut *depot* du buffer correspondant. Cependant, une telle modélisation ne permettrait pas de modéliser les caractéristiques du lien (par ex., un délai de transmission) et présenterait une architecture figée. Au contraire, la modélisation explicite du comportement des liens par une *Tclass* nous paraît plus adaptée.

Ainsi, nous introduisons une classe *Routage* qui modélise le routage des messages entre les ports de sortie et les buffers des ports d'entrée. Pour les mêmes raisons qui nous ont incités à utiliser une relation d'*Invocation* entre un port d'entrée et son buffer, nous modélisons la transmission entre un port de sortie et un buffer par deux relations d'invocation : une entre le module du port de sortie et la classe de routage, et une autre entre la classe de routage et le buffer du port d'entrée correspondant. La modélisation proposée met en évidence l'aspect asynchrone de la transmission d'un message d'un module à un autre. En effet, l'ensemble du processus de dépôt du message est effectué dans le flot d'exécution du module émetteur, alors que le processus de réception du message est effectué dans le flot d'exécution du module récepteur.

A titre d'exemple, la Figure 4-17 met en évidence la modélisation d'un lien entre le module *M1* et le module *M3*, et d'un lien entre le module *M2* et le module *M4*. Les deux traitements associés à l'acheminement des messages des deux liens sont mis en parallèle dans la classe *Routage*. Chacun de ces traitements consiste en (1) l'attente sur une porte invoquée (par ex. *g1 ?m1*) ; (2) la modélisation du comportement des liens par un temps de traitement ; (3) le dépôt du message dans le *BufferPort* correspondant au lien considéré via une porte d'invocation (par ex. *g1buf !m1*). La porte d'invocation est liée par une relation *d'Invocation* avec une porte du buffer correspondant (par ex. *depot = g1buf*). (4) Enfin, l'appel à la porte initiale (par ex. *g1*) libère l'appel effectué du côté du module émetteur du message (par ex. le module *M1*).

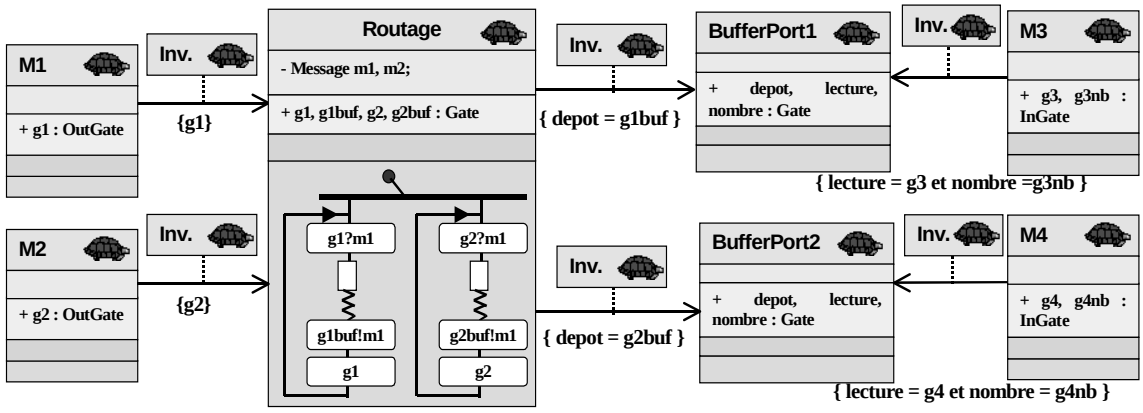


Figure 4-17. Modélisation TURTLE de la notion de lien.

4.4.3.4 *Modélisation du comportement interne des modules*

Cette section propose de répondre au besoin de modélisation des différents comportements des modules, à savoir (1) la réception de message sur un port et le traitement associé à cette réception, (2) la lecture de données sur une interface matérielle ou système, (3) l'émission de messages sur un port de sortie, (4) l'écriture de données sur une interface matérielle ou système, et enfin (5), la gestion de comportement périodique (cf. 4.2.2.2).

Dans de nombreux cas, l'abstraction d'un comportement se traduit par la modélisation temporelle de ce comportement, c'est-à-dire par la modélisation d'un temps de traitement obtenu par simulation ou par évaluation du nombre d'instructions (cf. paragraphe 2.3.3.1).

Réception de message

Un port d'entrée étant modélisé par un attribut de type *InGate*, un module doit réaliser un appel sur cet attribut pour lire une donnée en entrée sur ce port.

Si un module possède plusieurs ports d'entrée, le comportement de réception des messages sur les ports peut être modélisé en affectant une priorité distincte par port qui est d'autant plus élevée que les messages arrivant sur le port sont urgents (cf. [Shrivastava 1998]). Dans ce cas, il convient de tester la disponibilité de message sur les ports par ordre de priorité décroissante. Une offre sur les attributs de type *InGate* et limitée à une durée nulle permet une telle modélisation.

La lecture d'un message sur un port d'entrée doit être suivie par la modélisation relative au traitement de ce message. Afin de ne modéliser en TURTLE qu'un sous-système logiciel pertinent, il peut apparaître judicieux de modéliser les algorithmes de traitement des messages par les temps de traitement de ces algorithmes, ce qui permet de s'affranchir des détails algorithmiques. Pour modéliser ces temps de traitement, nous suggérons l'utilisation des opérateurs temporels TURTLE interruptibles (en effet, un algorithme est interrompu quand la tâche l'exécutant est suspendue) et de déporter les algorithmes eux-mêmes dans des classes ne faisant pas l'objet du processus de validation (classes non TURTLE). Notons enfin que les appels de méthode du système logiciel réel correspondant au traitement du message et réalisés sur les classes auxiliaires peuvent être insérés librement dans la modélisation TURTLE. En effet, ces appels de méthode sont ignorés lors de la traduction en RT-LOTOS (cf. Annexe A).

A titre d'exemple d'une telle modélisation, la Figure 4-18 présente un module M1 qui écoute les messages arrivant sur sa porte *g1*. En cas de réception d'un message sur cette porte, le module exécute la méthode *algo*() sur un objet o de la classe *AlgoM*, objet référencé par un attribut de M1. L'appel à cet algorithme est modélisé en TURTLE par un intervalle temporel.

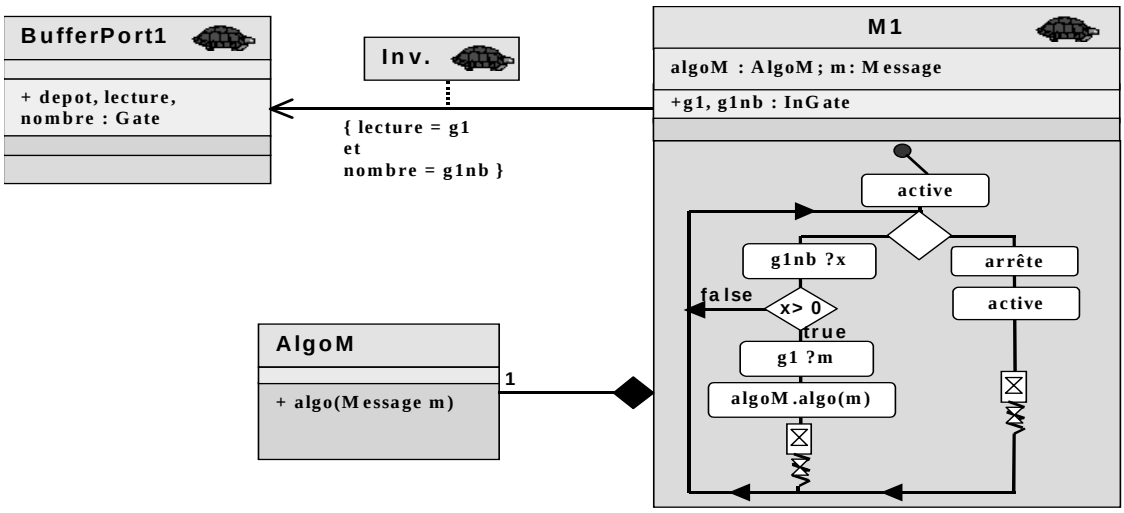


Figure 4-18. Exemple de modélisation d'un algorithme associé à la réception d'un message.

Lecture / écriture sur des interfaces matérielles

Le comportement temporel d'un appel système (lecture ou l'écriture sur une interface matérielle) est modélisé par un temps d'attente non interruptible : en effet, le traitement n'est pas effectué au sein de la tâche appelante, mais au sein du système d'exploitation.

Émission d'un message

En raison du choix d'associer un port de sortie à un attribut de type *OutGate*, l'émission d'un message est représentée par un appel sur un attribut de type *OutGate*. Selon la modélisation du module de routage, le message émis est ignoré (absence de lien) ou transmis au buffer d'un port d'entrée. [Allen 1998] souligne l'importance de ne pas modéliser les comportements liés à l'architecture au sein des modules. Nous suggérons donc de modéliser le comportement d'acheminement du message au sein du module de routage.

Gestion d'un comportement périodique

La modélisation des comportements périodiques inhérents aux systèmes logiciels temps-réel incite à utiliser l'opérateur de composition *Periodic* (cf. 4.3.3). Le comportement périodique d'un module se traduit alors par une association attribuée par l'opérateur *Periodic* et reliant la *Tclass* du module à elle-même (cf. Figure 4-2-(c)).

4.4.3.5 *Visualisation de l'architecture*

[Allen 1998] et [Medvidovic 2000] soulignent l'importance pour l'utilisateur d'un ADL de pouvoir appréhender l'architecture modélisée. La perception du schéma architectural d'une application modélisée avec TURTLE selon les directives présentées précédemment n'est pas triviale. C'est pourquoi nous proposons l'introduction d'un diagramme d'architecture qui reflète les modules et l'interconnexion de ces modules. Ce diagramme est construit à partir du diagramme de classes et d'objets et comprend :

1. toutes les instances des *Tclass* qui héritent de *Module* (prise en compte des modules de l'architecture, cf. 4.4.3.1);
2. des relations entre les objets du (1). Ces relations représentent les liens architecturaux entre les modules. Nous proposons d'orienter ces relations du module émetteur vers le module récepteur.
3. une formule OCL à chaque extrémité des relations du (2). Cette formule liste le port du module impliqué dans la relation. Pour un port de sortie, nous proposons de lister dans cette formule l'attribut de type *OutGate* impliqué dans la relation d'*Invocation* avec la classe de routage. Pour un port d'entrée, nous proposons de lister l'attribut de type *InGate* impliqué dans le relation d'invocation avec le buffer.

A titre d'exemple d'un diagramme d'architecture, nous reprenons les modules et liens du diagramme de classes illustré à la Figure 4-17. Nous considérons que le diagramme d'objets relatifs à ce diagramme de classes comporte une instance de chaque module de l'architecture. Le diagramme d'architecture résultant est donné à la Figure 4-19. Ce dernier comprend quatre instances (m1, m2, m3 et m4) des quatre modules M1, M2, M3, et M4 de la Figure 4-17. Le lien entre M1 et M3 est représenté par une relation orientée de m1 à m3 à laquelle sont associées deux relations OCL {g1} et {g3} représentant les ports impliqués dans le lien modélisé.

Figure 4-19. Exemple d'un diagramme d'architecture.

4.4.4 Méthodologie de vérification de propriétés sur la modélisation

La section précédente met en évidence la modélisation de l'architecture logicielle, introduite à la section 4.2 par utilisation du cadre formel TURTLE. Afin de répondre au besoin de validation formelle a priori du respect de contraintes applicatives lors des reconfigurations, nous proposons :

- de montrer comment ces contraintes peuvent être modélisées avec TURTLE conjointement à la modélisation de l'architecture applicative ;
- de mettre en évidence la modélisation des configurations applicatives et des changements de configuration, modélisation qui s'inspire du travail présenté dans [Allen 1998] ;
- enfin, de présenter le processus de validation formelle de l'ensemble de la modélisation (architecture, contraintes, configuration).

4.4.4.1 *Modélisation des contraintes applicatives*

Nous avons vu au chapitre 3, section 3.3.3.5, que la reconfiguration dynamique d'une application soulève le problème du respect de contraintes applicatives pendant la reconfiguration : les contraintes intrinsèques et les contraintes extrinsèques. Nous rappelons que les contraintes intrinsèques regroupent les contraintes syntaxiques, les contraintes de ressources physiques (mémoire, CPU), les contraintes logiques de bon fonctionnement (par ex. absence de *deadlock*) et enfin les contraintes temps-réel intrinsèques à l'application. Les contraintes extrinsèques concernent l'ensemble des contraintes relatives aux services devant être fournis par l'application à son environnement. L'ensemble de ces contraintes a été mis en évidence par [Kramer 1985][Gupta 1996] comme devant être pris en considération lors des reconfigurations d'application.

Modélisation directe de propriété

Certaines contraintes intrinsèques peuvent être modélisées directement au niveau de la description du comportement des modules ou au niveau de la description du comportement de l'architecture :

- les contraintes syntaxiques sont modélisées par utilisation des contraintes syntaxiques de TURTLE. Par exemple, le type des données doit être le même sur deux portes se synchronisant ;
- les contraintes de bon fonctionnement sont modélisées de façon implicite par la description du comportement du système ;
- certaines contraintes intrinsèques temps-réel telles que la périodicité des modules peuvent être modélisées par utilisation des opérateurs temporels de TURTLE ;
- enfin, les contraintes de ressources d'ordonnancement peuvent être modélisées conjointement avec la modélisation du comportement applicatif. En effet, le comportement interne temps-réel des modules est décrit par utilisation des deux types d'opérateurs temps-réel de description de comportement de TURTLE, à savoir les opérateurs temps-réel interruptibles, et non interruptibles. (cf. Figure 4-9). L'utilisation de ces opérateurs combinée à l'utilisation de l'opérateur de composition *Suspend* nous permet de décrire un comportement d'activation et de suspension des modules et donc l'ordonnancement de ces modules. Pour cela, il convient de modéliser l'ordonnanceur relatif aux modules, et d'appliquer l'opérateur de composition *Suspend* entre l'ordonnanceur et entre chaque module. La Figure 4-20 met en évidence cette composition, en supposant que la classe *Ordonnanceur* représente le comportement d'ordonnancement du système. Tout appel sur une des portes mentionnées conjointement à une relation de suspension (*Suspend*) active ou suspend alternativement le module impliqué dans la relation. Les autres *Tclass* de la modélisation TURTLE (buffer, routage) ne représentent pas des tâches du système : elles ne sont pas en relation directe avec l'ordonnanceur et ont donc été écartées de ce schéma. A noter que les modules utilisent le mécanisme d'invocation pour recevoir ou émettre des messages. Pendant ces invocations, les modules ne peuvent être suspendus immédiatement (cf. paragraphe 4.3.3, opérateur *Suspend*).

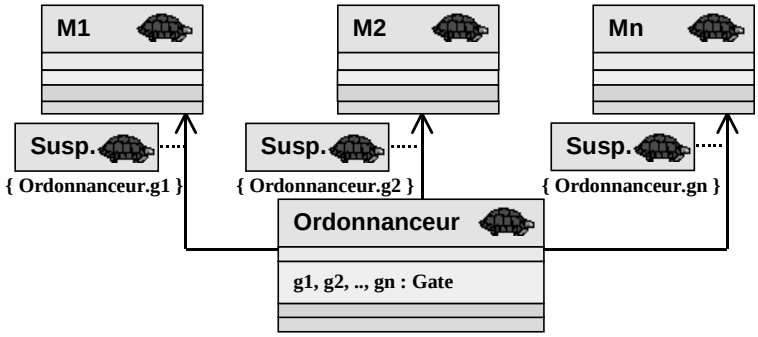


Figure 4-20. Relation de composition entre l'ordonnanceur et les modules applicatifs.

4.4.4.2 *Modélisation par observation*

Les contraintes intrinsèques non prises en compte dans la description du comportement des modules et les contraintes extrinsèques applicatives sont modélisées au moyen d'observateurs décrits en TURTLE, concept expérimenté avec succès sur d'autres langages de modélisation (voir par exemple [Jard 1988] pour Estelle). A l'instar de la modélisation proposée dans [Jard 1988], nos observateurs sont des classes externes à l'application logicielle intégrées au modèle TURTLE de cette dernière.

Ces observateurs doivent analyser le comportement applicatif de façon non intrusive [Jard 1988] et s'abstenir en particulier de perturber (1) l'ordonnancement du système et (2) les modules (classes applicatives) auprès desquels ils récupèrent les données. Supposons ici qu'une *Tclass* O (Observateur) désire accéder aux informations d'un module M. Pour mettre en œuvre cette récupération de données non intrusive, l'observateur est vu comme une *Tclass* active indépendante de l'ordonnanceur et en relation de synchronisation avec M, en s'assurant que cette synchronisation avec O n'introduira pas de retard d'exécution chez M (O accepte toujours de se synchroniser dès que M est prêt à le faire).

L'observateur ainsi construit doit également rendre compte des violations de propriété identifiées au cours de la validation. La réaction à ces violations sous forme d'une synchronisation sur une porte nommée *erreur* facilite leur repérage dans le graphe d'accessibilité (transition *erreur*). Il appartient au concepteur de l'application de choisir de la bloquer ou non lorsqu'une erreur se produit.

La Figure 4-21 illustre la possibilité pour un observateur O d'analyser une contrainte logique et temporelle d'un module M.

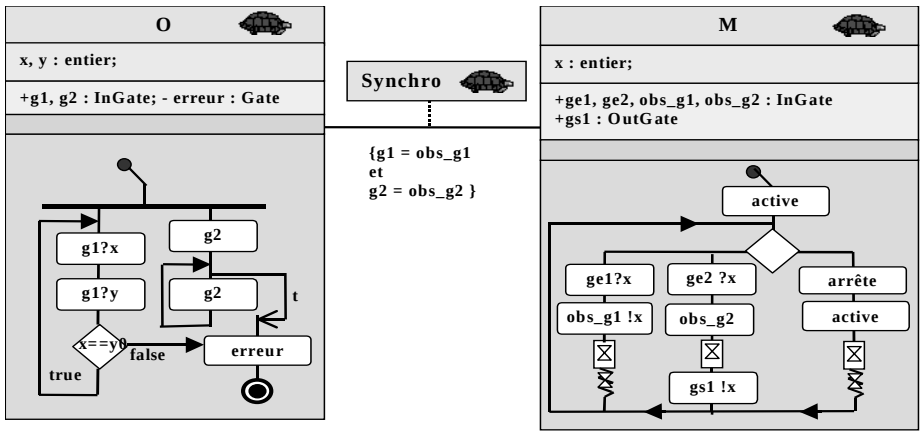


Figure 4-21. Observation de propriétés logiques et temporelles sur un module.

L'observation se traduit par une synchronisation entre O et M sur les portes obs_g1 et obs_g2 de M respectivement connectées aux portes g1 et g2 de O. La porte g1 permet d'observer une contrainte logique : les valeurs entières *2k+1*ième et *2(k+1)*ième reçues sur la porte ge1 de M

doivent être identiques. La porte g2 de O permet d'observer une contrainte temporelle : entre la réception de deux messages sur la porte ge2, il ne s'écoule pas plus de t unités de temps. Au niveau de l'observateur, les deux contraintes sont examinées en parallèle sur les parties gauche et droite du diagramme, respectivement. La violation de l'une ou l'autre de ces propriétés déclenche une synchronisation sur la porte *erreur* et l'arrêt de l'observateur, ce qui a pour effet de bloquer partiellement M (les synchronisations sur g1 et g2 ne sont plus possibles).

Nous proposons d'utiliser la technique de l'observation à des fins de modélisation des ressources mémoire applicatives. Pour cela, nous suggérons l'introduction d'un observateur dédié à la gestion de la mémoire. Cet observateur se synchronise avec toutes les classes TURTLE représentant des modules applicatifs. L'observateur offre deux portes de synchronisation correspondant respectivement à une allocation mémoire et à une désallocation. Toute classe TURTLE réalisant une de ces deux opérations mémoire doit alors offrir la synchronisation sur la porte correspondant à l'opération.

4.4.4.3 *Validation formelle d'un modèle de reconfiguration dynamique*

Principe général

La validation formelle demeure un problème ouvert dans le domaine de la reconfiguration dynamique du logiciel [Gupta 1996]. Son objectif est de prouver le respect a priori de contraintes intrinsèques et extrinsèques au regard d'une modification de la configuration d'une application. La solution explorée dans ce mémoire consiste à appliquer des techniques éprouvées de simulation et d'analyse d'accessibilité de modèles temporisés [Courtiat 2000] sur la spécification RT-LOTOS dérivée de la modélisation TURTLE de la reconfiguration dynamique d'une application.

Nous avons vu précédemment qu'une reconfiguration sur une architecture à composants consiste en l'application d'opérations de reconfiguration sur la configuration de cette application (cf. 3.4.8.4). De plus, selon la nature de la reconfiguration, certaines propriétés applicatives peuvent être maintenues pendant la phase de reconfiguration (cf. 3.3). De ces deux considérations, nous en déduisons un schéma générique de reconfiguration dynamique (cf. Figure 4-22) :

- l'application est exécutée de t0 à t1 dans une première configuration (configuration 1). Les propriétés extrinsèques (P1) sont respectées de t0 à t1.
- L'exécution d'opérations de reconfiguration de t1 à t2 modifie progressivement la configuration de l'application. Pendant cette phase, l'application rend les services P à son environnement.
- Enfin, à partir de t2, l'application est dans sa deuxième configuration (configuration 2), les propriétés extrinsèques P2 sont respectées.

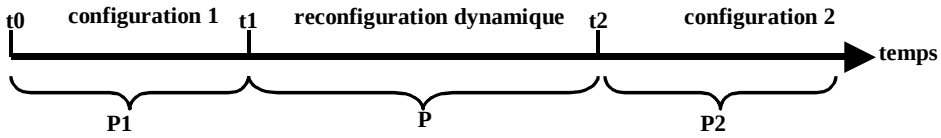


Figure 4-22. Chronogramme générique d'une reconfiguration dynamique.

Ainsi, valider formellement le respect de contraintes se caractérise par la validation formelle du respect :

- des propriétés P1 en configuration 1 ;
- des propriétés P pendant la reconfiguration ;
- des propriétés P2 après reconfiguration.

Si nous avons vu comment modéliser une configuration logicielle et ses propriétés dans le paragraphe 4.4.3, la validation formelle du respect de propriétés dynamiques soulève le problème de la modélisation des changements de configuration et des changements de propriétés devant être observées. Par la suite, nous mettons en évidence comment peuvent être modélisés ces deux

aspects (changement de configuration, changement de propriétés observées) par utilisation de TURTLE.

Modélisation du changement de configuration

Selon l'exemple du *configurator* introduit dans [Allen 1997], nous proposons de modéliser les opérations de reconfiguration dans une classe spécifique apte à piloter les composants architecturaux afin de réaliser les opérations de reconfiguration.

Pour cela, nous proposons d'utiliser une classe TURTLE appelée *GestConf* qui gère la configuration logicielle initiale (*configuration 1*, cf. Figure 4-22), la durée de fonctionnement de l'application dans cette configuration (cette durée vaut *t1 – t0*, cf. Figure 4-22) et l'exécution du script de reconfiguration dynamique qui regroupe les opérations de reconfiguration dynamique (de t1 à t2, cf. Figure 4-22).

GestConf doit tout d'abord modéliser la première configuration logicielle. Cela consiste à activer simultanément les modules de la première configuration ainsi que les liens et les observateurs de cette configuration. Nous proposons de modéliser l'activation des ces entités par synchronisation sur une porte *active* située en entête du comportement de chacune des *Tclass* impliquées dans cette configuration.

Ensuite, *Gestconf* doit aussi modéliser le fonctionnement de l'application dans cette première configuration pendant une durée de *t1 – t0*. Nous représentons cela au niveau de *GestConf* par un opérateur temporel de comportement.

Enfin, *GestConf* doit modéliser l'exécution des opérations de reconfiguration dynamique. Cela consiste à modéliser (1) l'arrêt des observateurs de P1 (cf. Figure 4-22) et l'activation des observateurs de P, (2) l'exécution des opérations de reconfiguration dynamique comme spécifiées dans le script de reconfiguration, et (3) l'arrêt des observateurs de P et l'activation des observateurs de P2. Si l'activation ou l'arrêt d'observateurs peut se traduire par une synchronisation sur une porte d'activation ou d'arrêt (procédé choisi pour modéliser la suspension et l'activation des modules, cf. 4.4.3.1), la modélisation des opérations de reconfiguration est une tâche plus complexe que nous explicitons par la suite.

Nous souhaitons modéliser les changements de configuration relatifs aux architectures à composants (cf. 3.4.8.4), c'est-à-dire :

- la modification d'une interconnexion : il faut pouvoir modéliser la suppression d'une connexion entre deux modules, ou la création d'une connexion entre deux modules ;
- la modification de la structure en termes de modules : ajout ou retrait d'un module ;
- enfin, la modification interne d'un module i.e. la modification du comportement de ce module.

Afin de ne pas perturber le fonctionnement de l'application, la classe *GestConf* doit être non intrusive. A l'instar des observateurs (cf. 4.4.4.2), nous proposons de modéliser les opérations de reconfiguration (par ex.., l'arrêt d'un module) par l'utilisation de portes de synchronisation spécifiques et d'opérateurs de composition *Synchro*. Nous montrons par la suite comment ce choix de modélisation nous permet de mettre en œuvre la modélisation de la modification des interconnexions, de la structure modulaire et enfin du comportement des modules : pour chacun de ces types de modification, nous proposons un principe de modélisation.

Modélisation d'une modification d'interconnexion

La modification d'une interconnexion consiste en la destruction ou la création d'un lien.

La destruction d'un lien se caractérise par l'absence d'acheminement des messages entre le port de sortie de ce lien et le port d'entrée du lien. L'acheminement des messages étant modélisé au sein de la *Tclass* de routage, nous proposons d'ajouter pour chaque lien devant être détruit une porte

de synchronisation modélisant la désactivation du lien. L'appel par *GestConf* sur cette porte produit l'arrêt du lien.

De même, pour modéliser la création d'un lien, nous proposons l'utilisation d'une porte de synchronisation spécifique et pilotée par *GestConf*. L'exécution de l'activité de ce lien dans la classe de routage est conditionnée par la synchronisation préalable sur cette porte.

A titre d'exemple, la Figure 4-23 met en évidence deux liens qui sont arrêtés, et un autre qui est lancé. Par souci de clarté, nous n'avons pas fait figurer les synchronisations entre le gestionnaire de configuration et les modules. Ces synchronisations sont nécessaires pour l'activation des modules (configuration 1).

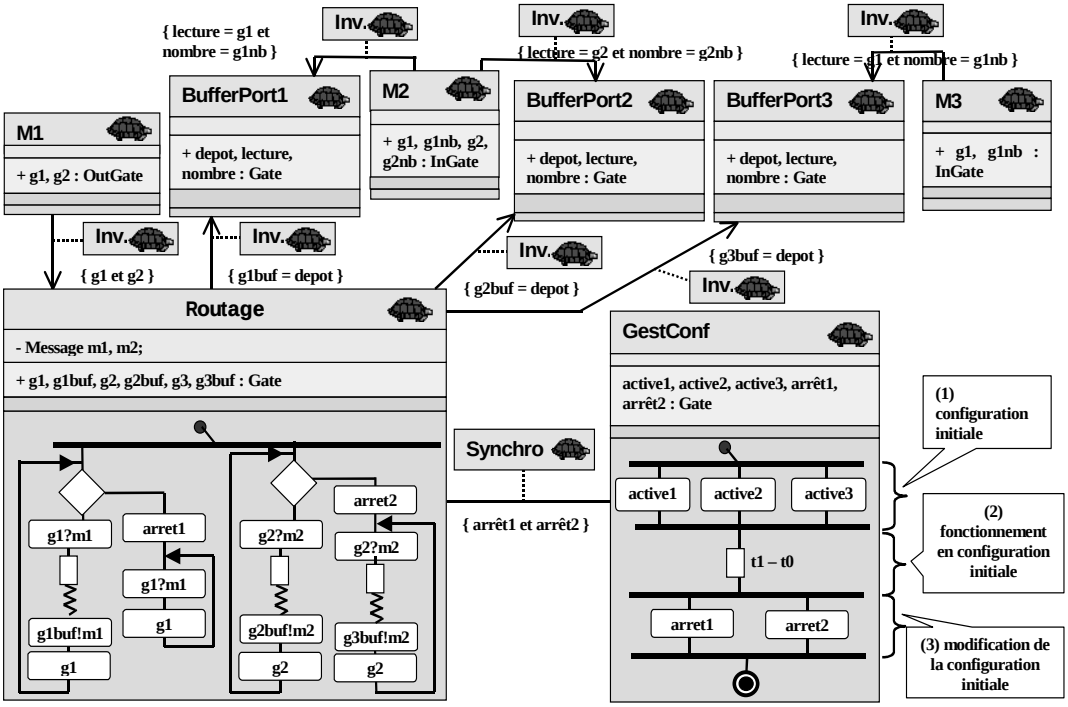


Figure 4-23. Modélisation TURTLE d'une suppression de lien et d'une modification de lien.

Dans la configuration initiale (cf. Figure 4-23), la porte de sortie *g1* du module *M1* est connectée à la porte d'entrée *g1* du module *M2*. De même, la porte de sortie *g2* du module *M1* est connectée à la porte d'entrée *g2* du module M2. Ces liens modélisés sont mis en place avant le démarrage applicatif. Le démarrage de l'application s'accompagne de l'activation des trois modules *M1*, *M2* et *M3* (1). Puis l'application fonctionne en configuration 1 pendant au moins *d* unités de temps, et pendant au plus *d+ l* unités de temps (2). La reconfiguration est ensuite initiée (3). Elle consiste en l'exécution en parallèle des appels sur les portes *arrêt1* et *arrêt2*. Lors de l'exécution de la synchronisation entre *Routage* et *GestConf* sur *arrêt1*, la connexion entre le module de routage et *BufferPort1* est rompue (si *M1* était en train d'émettre un message sur *g1*, la fin de l'invocation a lieu avant que le lien soit modifié). En effet, au sein de la classe *Routage*, la synchronisation sur *arrêt1* supprime l'appel à *g1buf* lors de la réception des messages sur *g1*. De même, l'exécution de la synchronisation sur *arrêt2* modifie le comportement du lien entre *M1.g2* et *M2.g2* : au niveau de la classe *Routage*, tout message reçu sur la porte *g2* est émis non plus sur la porte *g2buf* mais sur la porte *g3buf* d'où la suppression du lien entre *M1.g1* et *M2.g2* et la création d'un lien entre *M1.g2* et *M3.g1*. A noter que la suppression du lien n'est réalisée que lorsque la synchronisation sur *arrêt2* devient effective c'est-à-dire que plus aucun message n'est en transit sur le lien de communication. Lorsque les deux synchronisations sur *arrêt1* et *arrêt2* sont terminées, l'application s'exécute dans une nouvelle configuration.

Modélisation d'une modification de la structure en termes de modules

La modification de la structure en termes de modules se traduit par l'ajout de modules ou le retrait de modules. Un module est représenté par une *Tclass* qui peut-être activée ou désactivée par appel sur des portes de synchronisation (cf. 4.4.3.1). Nous proposons donc de modéliser l'ajout d'un module par la première activation de ce module et le retrait d'un module par l'arrêt de ce module. Ainsi, le gestionnaire de configuration, par utilisation de portes de synchronisation, peut piloter la structure modulaire.

La Figure 4-24 met en évidence l'arrêt du module M1, et l'activation synchronisée du module M2. Nous avons supposé que l'arrêt du module M1 doit se faire uniquement au point de reconfiguration de M1.

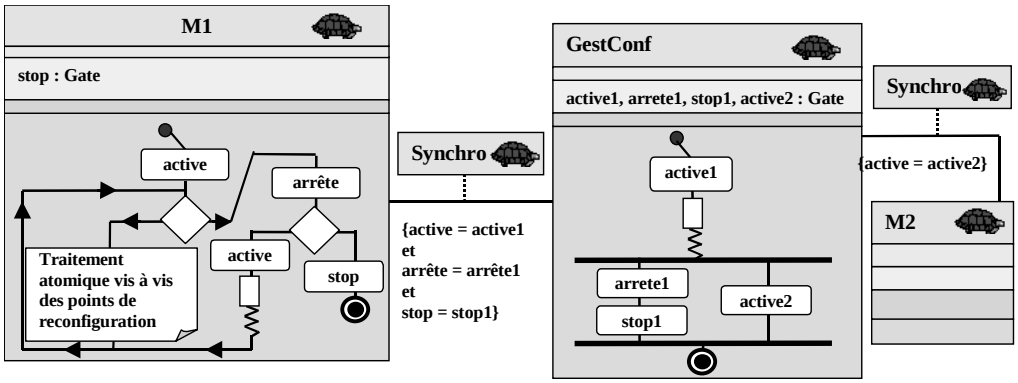


Figure 4-24. Modélisation d'un ajout et d'un retrait de module.

Modélisation des modifications internes des modules

Nous proposons deux solutions à la modélisation du changement du comportement interne d'un module. Tout d'abord, cette modification peut être modélisée par un changement de module : un module M1 est remplacé par un module M2 dont le comportement est le nouveau comportement désiré pour M1. La deuxième solution consiste à insérer dans le comportement du module une porte de synchronisation qui permet de choisir le numéro de version du comportement du module : le gestionnaire de configuration, via cette porte de synchronisation, envoie lorsque nécessaire le nouveau numéro de version du comportement.

A titre d'exemple, la Figure 4-25 met en évidence la modification de comportement d'un module M1.

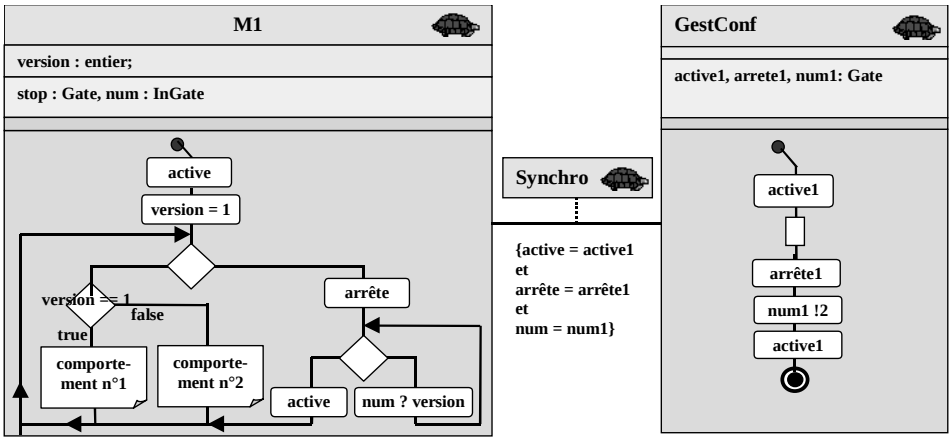


Figure 4-25. Modélisation de la modification interne du comportement d'un module.

La modification interne se fait en trois étapes : le module rejoint son point de reconfiguration (synchronisation avec le gestionnaire de reconfiguration), puis par synchronisation,

le gestionnaire de reconfiguration modifie la version de comportement du module. Enfin, le module est réactivé par synchronisation d'activation avec le gestionnaire de configuration. A noter que cela ne représente qu'une modélisation qui ne préjuge en rien de l'implantation d'une telle modification qui, dans le cas réel, ne peut avoir été prévue systématiquement lors de la première version d'un module alors en cours de fonctionnement.

Modélisation des propriétés de reconfiguration

Nous avons vu précédemment que les propriétés internes et externes applicatives vérifiées par les observateurs ne sont valables que pour une configuration donnée (cf. Figure 4-22).

Soit une application A, soit O1 l'ensemble des observateurs de la configuration applicative n°1, soit O2 l'ensemble des observateurs relatifs à la reconfiguration dynamique, et soit O3 l'ensemble des observateurs de la configuration applicative n°2. Un observateur est dit actif à un instant t si la propriété qu'il observe doit être respectée par le système à l'instant t. Il faut :

- à t0 : activer les observateurs de O1;
- à t1 : activer les observateurs de O2 qui ne sont pas dans O1 et désactiver les observateurs de O1 qui ne sont pas dans O2;
- à t2 : activer les observateurs de O3 qui ne sont pas dans O2 et désactiver les observateurs de O2 qui ne sont pas dans O3.

Pour modéliser cela, nous proposons de laisser les observateurs gérer l'examen des propriétés en permanence, mais nous associons aux observateurs une valeur booléenne qui indique si les cas d'erreurs doivent être traités ou pas. En effet, afin d'observer les classes applicatives les observateurs utilisent des portes de synchronisation : la désactivation des observateurs entraînerait un blocage des classes applicatives sur ces portes de synchronisation. Le positionnement de cette valeur booléenne est effectué par le gestionnaire de reconfiguration lors du début ou de la fin d'une reconfiguration (à t1 ou t2, cf. Figure 4-22).

La Figure 4-26 met en évidence une telle modélisation : le module *M1* est observé par *O* (nous reprenons l'exemple de la Figure 4-21, page 69).

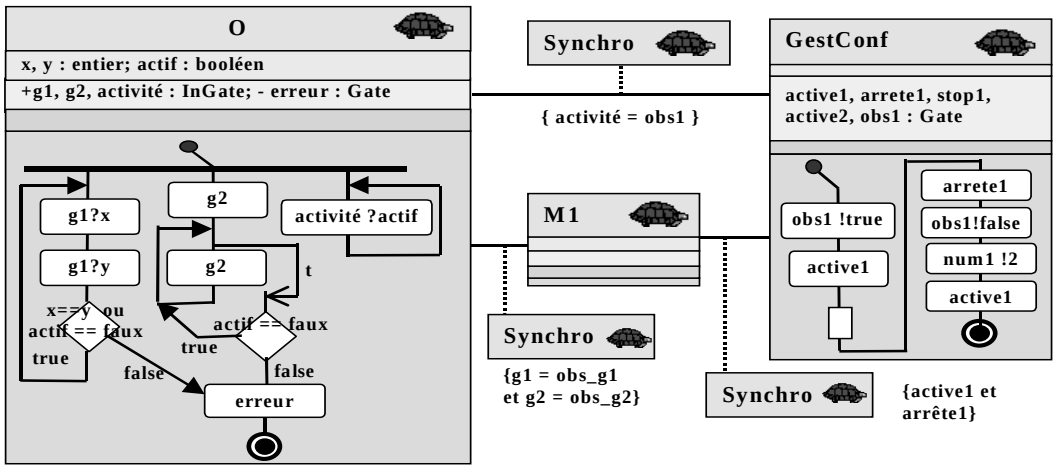


Figure 4-26. Modélisation de l'activation / désactivation d'observateurs.

Nous avons ajouté à la classe *O* un attribut de type booléen (*actif*) et un test sur cette valeur booléenne avant l'entrée dans l'état d'erreur (appel sur *erreur*) : l'observateur s'arrête si et seulement si une propriété observée est fausse et si l'observateur est actif. Dans un premier temps, le gestionnaire de configuration active l'observateur, puis le module M1. Après un certain temps de fonctionnement nominal, le gestionnaire arrête le module *M1* (*arrête1*), puis l'observateur (*obs1 !false*, on suppose ici qu'il n'y a pas de continuité de service du point de vue des propriétés observées par *O*) puis modifie le comportement du module *M1* (*num1 !2*), avant de le réactiver (active1).

Exploitation du graphe d'accessibilité

La phase de validation formelle consiste à générer le graphe d'accessibilité de l'ensemble de la modélisation (classes applicatives, classes de gestion de la configuration, observateurs) par utilisation du processus de validation formelle de l'environnement TURTLE (cf. 4.3.3.3). Si le système est fini, un graphe d'accessibilité est obtenu. Par la suite, nous détaillons l'exploitation de ce graphe.

L'analyse du graphe d'accessibilité permet de mettre en évidence les appels aux portes de synchronisation d'erreurs des observateurs (portes *erreur*) et les situations d'interblocage. Si ces cas sont observés pendant la configuration logicielle originale (i.e. entre t0 et t1, cf. Figure 4-22), alors cela remet en cause non pas la reconfiguration dynamique de l'application mais la configuration initiale de l'application. Si ces cas sont observés après démarrage de la reconfiguration dynamique (i.e. après t1, cf. Figure 4-22), cela signifie que la reconfiguration ne peut pas être effectuée avec certitude en respectant les propriétés modélisées. Deux cas se présentent alors : (1) la reconfiguration conduit systématiquement l'application dans des cas d'erreurs et (2) la reconfiguration conduit parfois à des cas d'erreur. Deux solutions sont alors envisageables : une modification du script de reconfiguration ou un assouplissement des propriétés analysées par les observateurs (reconfiguration dynamique avec respect de moins de services).

Deux solutions visent à modifier le script de reconfiguration. La première consiste à générer un nouveau script amenant l'application à une configuration logicielle différente de celle souhaitée auparavant : comme la reconfiguration ne peut pas être menée, on décide au lieu d'introduire par exemple deux nouveaux services, de n'en introduire qu'un seul. La deuxième solution est préconisée lorsque la reconfiguration conduit parfois à des cas d'erreurs (cas numéro (2), voir paragraphe précédent). Les opérations de reconfiguration dynamique sont modélisées par des synchronisations sur des portes. Ces synchronisations apparaissent donc au niveau du graphe d'accessibilité si ces opérations ont pu être réalisées. L'analyse du graphe permet donc de mettre en évidence les contraintes de synchronisation (transitions logiques) et temporelles (transitions temporelles) qui existent entre les opérations de reconfiguration pour que la reconfiguration puisse être menée à bien. A titre d'exemple, supposons par exemple que deux opérations de reconfiguration o1 et o2 soient modélisées par des appels sur les portes g1 et g2. Supposons que le graphe d'accessibilité correspondant à l'exécution de ces deux opérations soit celui représenté à la Figure 4-27 : la transition étiquetée « g1 » met en évidence l'exécution de l'opération o1 depuis l'état 0. Ensuite, soit l'opération o2 (porte g2) est exécutée et conduit l'application dans l'état 2, soit 50 unités de temps s'écoulent au bout desquelles la transition « erreur » conduit l'application dans l'état 3 : ce graphe met en évidence que l'opération o2 doit être réalisée dans un délai de 50 unités de temps après la réalisation de o1.

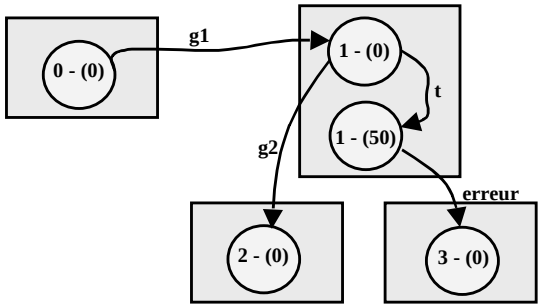


Figure 4-27. Exemple de mise en évidence sur un graphe d'accessibilité d'une contrainte temporelle entre deux opérations.

Finalement, l'analyse du graphe d'accessibilité peut permettre l'identification de contraintes logiques et temporelles concernant les opérations de reconfiguration dynamique.

4.5. Conclusion

L'objectif de ce chapitre était de répondre au problème de la preuve formelle en terme de respect des contraintes applicatives lors de la reconfiguration d'une application [Gupta 1996].

Pour répondre à ce besoin de validation, nous avons introduit une architecture applicative à composants compatible avec les environnements spatiaux et intrinsèquement reconfigurable (section 4.2). Face aux contraintes industrielles d'Alcatel en termes de langage de conception, nous avons proposé un cadre formel basé sur UML et autorisant la validation formelle de propriétés à partir des diagrammes UML (section 4.3). Enfin, par utilisation de cet environnement, nous avons montré comment modéliser les concepts architecturaux nécessaires à la validation formelle du respect de contraintes applicatives lors des reconfigurations (section 4.4). Nous nous sommes pour cela inspirés notamment de l'ADL Wright [Allen 1997]. Enfin, nous avons mis en évidence le besoin de gestion de contraintes logiques et temporelles au sein des scripts de reconfiguration.

Le chapitre suivant présente l'intégration de ce processus de modélisation et de validation formelle au sein des environnements spatiaux selon trois étapes :

- la définition d'opérations de reconfiguration dynamique et d'un script de reconfiguration dynamique apte à exprimer ces opérations et les contraintes relatives à ces opérations ;
- la proposition d'une méthodologie logicielle adaptée à la reconfiguration dynamique intégrant le cadre formel proposé et permettant la génération du script de reconfiguration ;
- enfin, la définition de mécanismes supports à l'architecture et à la reconfiguration dynamique i.e. supports à l'exécution du script de reconfiguration.

Chapitre 5

Mécanismes de reconfiguration dynamique Spécification, méthodologie et mise en œuvre

Résumé. Un cadre formel de validation a priori de propriétés internes et externes des applications intrinsèquement reconfigurables a été proposé au chapitre précédent. Nous proposons d'explorer dans ce chapitre les mécanismes supports pour la reconfiguration dynamique. Ces mécanismes reposent sur une méthodologie logicielle permettant l'expression et la validation formelle des scénarios de reconfiguration et sur une architecture logicielle support à la réalisation des scénarios de reconfiguration. Ainsi, nous proposons d'introduire un formalisme basé sur les réseaux de Petri dont le rôle est de spécifier les opérations à effectuer sur une application donnée afin de la faire évoluer vers la configuration souhaitée tout en respectant les contraintes intrinsèques et extrinsèques de cette application (scénario de reconfiguration). Cette spécification permet de décrire des contraintes associées à ces opérations, contraintes qui s'attachent à la préservation des propriétés applicatives pendant la mise à jour : contraintes temporelles et logiques associées à l'exécution des opérations, possibilité de retour en arrière en cas de violation de propriétés. Nous proposons par la suite une méthodologie basée sur UML et permettant de valider formellement à l'aide du cadre formel proposé dans le chapitre précédent que l'exécution de cette spécification sur l'application ne remet pas en cause les propriétés applicatives (propriétés de continuité de service par exemple). Enfin, nous décrivons un environnement support à l'exécution de cette spécification.

5.1. Introduction

La reconfiguration dynamique d'un logiciel est généralement réalisée par l'exécution sur ce logiciel d'opérations logicielles dites de reconfiguration. Le cadre formel présenté au chapitre précédent et dont le rôle est de valider formellement l'application de ces opérations ne met pas en exergue quelles opérations doivent être appliquées pour conduire un logiciel dans une nouvelle configuration. Le problème ne se limite d'ailleurs pas à la définition de ces opérations, mais il concerne aussi la prise en compte des contraintes logiques et temporelles liées à l'application de ces opérations sur un logiciel donné. Notre objectif est de montrer que le respect de ces contraintes logiques et temporelles lors de l'application des opérations de reconfiguration dynamique assure le respect des propriétés applicatives et notamment de la continuité de service.

Nous proposons de présenter dans ce chapitre l'ensemble des mécanismes qui vont servir à définir puis appliquer ces opérations tout en respectant les contraintes associées à ces opérations. Dans un premier temps, nous énumérons les étapes constituant l'ensemble du processus permettant de mettre à jour une application spatiale embarquée. Ces étapes concernent notamment la spécification des opérations de reconfiguration dynamique à appliquer. Deuxièmement, nous définissons les opérations de reconfiguration dynamique dont il serait souhaitable de disposer. Troisièmement, nous introduisons un formalisme basé sur les réseaux de Petri afin de décrire l'ordonnancement logique et temporel entre les opérations devant être appliquées. Afin de spécifier les scénarios valides de reconfiguration, nous proposons un nouveau formalisme basé sur les réseaux de Petri à flux temporels [Diaz 1993][Sénac 1996]. Cette extension consiste en l'utilisation de jetons colorés et de gardes typées [Jensen 1997]. Nous montrons par la suite que ce formalisme

est particulièrement bien adapté à la modélisation des caractéristiques des opérations de reconfiguration. Enfin, nous proposons une architecture sol et bord support à la reconfiguration d'applications spatiales. Cette architecture est décrite en termes de blocs fonctionnels, de mécanismes logiciels et de langage support, et enfin en termes de protocole d'échanges entre le bord et le sol.

5.2. Processus de reconfiguration dynamique

5.2.1 Principe général

Le processus de reconfiguration dynamique est le processus qui permet à une application spatiale en cours de fonctionnement d'être reprogrammée. Le processus doit permettre à une équipe sol de proposer une nouvelle version du logiciel, de valider non seulement la nouvelle version du logiciel mais aussi le passage de la version précédente à la nouvelle, puis d'appliquer une procédure de reconfiguration à l'application embarquée afin de la faire évoluer.

5.2.2 Étapes du processus

Lors d'un processus de reconfiguration dynamique, on peut distinguer les étapes suivantes (cf. Figure 5-1):

1. un besoin de reconfiguration dynamique concernant l'application est exprimé. Ce besoin peut être exprimé pour deux raisons principales. La première concerne le constat d'une non conformité du satellite avec la spécification du service devant être rendu. Ce défaut peut être constaté soit par les exploitants des services satellites soit par le service de contrôle du satellite. La deuxième raison concerne l'évolutivité du système satellite mis en œuvre : un client ou un exploitant du système satellite considéré exprime un nouveau besoin.
2. Une équipe en charge de la maintenance du système satellite considéré analyse le besoin de reconfiguration dynamique, et propose une nouvelle architecture bord dont la mise en œuvre se traduit par la modification d'entités matérielles et logicielles (notre contribution aborde seulement la modification d'entités logicielles). Si une des entités logicielles doit être modifiée, le besoin de reconfiguration est transmis à l'équipe logicielle responsable.
3. L'équipe logicielle doit proposer, au regard des nouveaux besoins, une nouvelle application. Cette nouvelle application est issue d'un nouveau cycle de développement (spécification, conception, codage, phase de test). La modélisation logicielle est réalisée par utilisation de l'environnement TURTLE.
4. Une fois la nouvelle application terminée, il convient de déterminer la procédure permettant de faire évoluer l'application précédente vers la nouvelle application, avec conservation des propriétés spécifiées. Pour cela, nous proposons de modéliser les opérations à effectuer à bord : nous introduisons à cette fin un langage de reconfiguration dynamique qui autorise d'une part la description d'un ensemble d'opérations permettant de faire évoluer une application 1 vers une application 2, et d'autre part la modélisation des contraintes liées à l'exécution des opérations décrites. En effet, l'exécution des opérations est susceptible d'entraîner la violation de contraintes internes et externes à l'application. La spécification du scénario d'exécution des opérations de reconfiguration dynamique est appelée SRD (Spécification de Reconfiguration Dynamique).
5. Les opérations étant exprimées avec la SRD, il convient de déterminer formellement le respect des propriétés internes et externes de l'application lors de l'exécution de la SRD sur cette application. Pour cela, nous proposons de modéliser, avec l'environnement TURTLE, d'une part l'exécution de la SRD, et d'autre part les propriétés devant être

respectées par l'application. La validation formelle de la modélisation ainsi obtenue doit permettre de vérifier que l'exécution à bord de la SRD ne perturbera pas le respect des propriétés applicatives. Si certaines de ces contraintes ne sont pas respectées, il convient de modifier la SRD, et de recommencer le cycle de validation.

6. En cas de succès du cycle de validation, la SRD est envoyée au centre de gestion de la mission du système satellite (opérateur sol). De plus, les nouvelles entités logicielles nécessaires à la génération de la nouvelle application (par exemple, de nouvelles classes applicatives) sont placées si besoin dans un serveur de classes, situé au sol.
7. La SRD est envoyée à bord, où elle est traitée par un environnement support permettant l'exécution de toutes les opérations pouvant être spécifiées. L'exécution de la spécification est réalisée au regard des contraintes spécifiées dans ce modèle. En cas de violation de ces contraintes, un processus de retour en arrière est activé. L'exécution de certaines des opérations décrites dans la SRD peut déclencher le téléchargement de classes applicatives depuis le serveur de classes situé au sol, au travers de la liaison de communication bord/sol (liaison TM/TC).
8. Enfin, une fois l'exécution du modèle de reconfiguration dynamique achevé, un rapport est envoyé à l'opérateur sol.

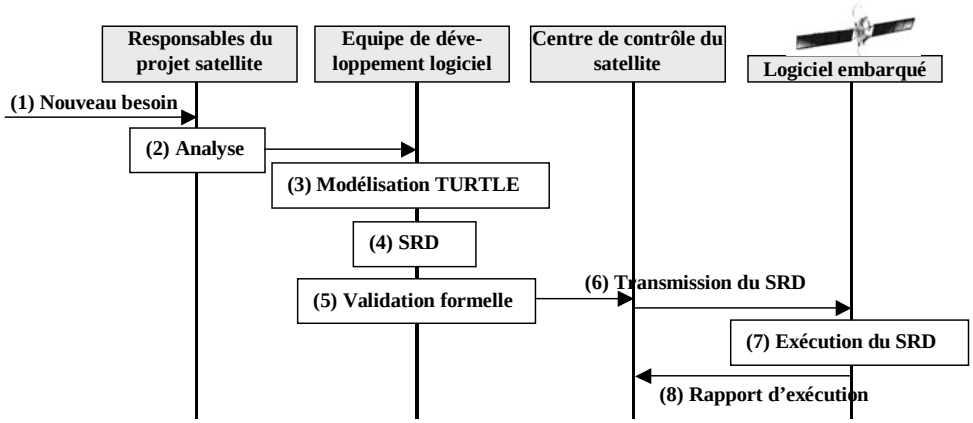


Figure 5-1. Processus de reconfiguration dynamique.

5.3. Opérations de reconfiguration dynamique

5.3.1 Notion d'opérations de reconfiguration dynamique

Une opération de reconfiguration dynamique représente une opération logicielle pouvant être appliquée à une application en cours de fonctionnement, et dont le rôle est de participer à la conduite d'une évolution de la configuration d'une application. Ces opérations apparaissent comme des services logiciels supports au processus de reconfiguration dynamique.

Dans un premier temps, nous présentons les caractéristiques de ces opérations de reconfiguration dynamique. Par la suite, nous proposons une description de ces opérations, selon que ces opérations sont relatives à la configuration de l'architecture applicative (opérations dite *inter-modules*) ou relatives à la configuration interne des modules (opérations dites *intra-modules*).

5.3.2 Caractéristiques et terminologie des opérations

Les opérations de reconfiguration dynamique possèdent les caractéristiques suivantes :

– l'exécution d'une opération de reconfiguration est conditionnée par le respect d'une précondition. La *précondition* d'une opération o est une assertion représentant l'ensemble des conditions devant être respectées par le système applicatif A pour pouvoir lui appliquer l'opération o. La précondition d'une opération o au regard d'une

application A est notée *Pré*(o, A). De même, la *postcondition* d'une opération o est une assertion représentant l'ensemble des conditions devant être respectées par le système applicatif A après application de l'opération o. La postcondition d'une opération o est notée *Post*(o, A).

- Une opération o de reconfiguration peut admettre une opération inverse. Si tel est le cas, l'opération o est qualifiée d'opération réversible et l'opération inverse est notée $\bar{o}$.
- Une opération est dite de haut niveau lorsqu'elle concerne des éléments logiciels de notre architecture à composants. Sinon, elle est qualifiée de bas niveau.

5.3.3 Taxonomie des opérations

Les opérations de reconfiguration dynamique des architectures à composant sont généralement limitées à des modifications sur l'architecture logicielle (modifications inter-composants, cf. 3.4.8.4). Nous avons montré dans le chapitre 3 l'intérêt d'introduire des opérations appelées intra-composants et permettant de réaliser des modifications internes aux composants. Par la suite, nous définissons des opérations de reconfiguration dynamique pour les deux types de modifications (inter et intra composants).

5.3.3.1 *Opérations inter-modules*

Les opérations inter-modules concernent soit la modification des interconnexions entre modules, soit la modification de l'état d'exécution des modules. Leur rôle est donc de piloter des changements de la structure applicative.

Opérations de gestion des interconnexions

- Création d'un lien : il s'agit de créer une connexion entre un port de sortie et un port d'entrée. Lors de la création, il convient de s'assurer que le typage des deux ports concernés est compatible.
- Destruction d'un lien : il s'agit de détruire une connexion entre un port de sortie et un port d'entrée. La principale difficulté de cette opération est qu'il faut s'assurer qu'aucun message n'est en transit sur le lien considéré.

Opérations de gestion des modules

- Ajout d'un module : cette opération consiste à ajouter à l'application un module. L'ajout nécessite le chargement éventuel d'un nouveau code (classes) au sein de l'application, l'instanciation de nouveaux objets, et enfin le démarrage d'un nouveau flux d'exécution (celui du nouveau module).
- Retrait d'un module : cette opération consiste à retirer d'une application un module c'est-à-dire à arrêter le flux d'exécution du module, à détruire ses objets, et enfin à retirer de la mémoire applicative toutes les classes uniquement relatives à ce module.

5.3.3.2 *Opérations intra-modules*

Les opérations intra-modules autorisent tous les types de changements internes au module c'est-à-dire les changements d'interfaces, les changements de comportement, et enfin les changements de données.

Opération de gestion des ports

- Ajout d'un port : l'ajout d'un port consiste à ajouter un port d'entrée ou de sortie à un module. S'il s'agit d'un port d'entrée, un buffer correspondant à ce port est créé.
- Destruction d'un port : la destruction d'un port consiste à retirer d'un module un port qui n'est pas connecté par un lien à un autre port.

Opérations de modification du comportement interne

La modification du comportement interne revient non plus à effectuer des opérations sur une architecture logicielle spécifique (architecture à composants), mais à effectuer des opérations sur une organisation d'objets quelconque (la grappe d'objets du module). Ainsi, l'application d'opérations de modification du comportement interne des composants soulève des problèmes similaires à la modification d'appels de procédures lors de la modification d'applications développées selon un modèle de programmation fonctionnel. En effet, une architecture d'appels de méthodes s'apparente à une architecture d'appels de procédures. Ainsi, des problèmes de cohérence logique de l'interconnexion des objets internes à un module (appels de méthodes par exemple) se posent. En effet, supposons par exemple qu'un module M possède un objet o_1 et un objet o_2, et que l'objet o_1 réalise un appel sur la méthode m de l'objet o_2. Si la signature de la méthode m est modifiée, et que l'appel dans o_1 n'est pas modifié, alors cette modification de méthode introduit une incohérence au sein de l'architecture objets.

Afin de gérer ces problèmes d'incohérence logique lors de la mise à jour de la structure interne d'un module, nous proposons deux procédés : un procédé général, qui autorise tout type de modification interne d'un module, mais qui engendre un nombre élevé d'opérations logicielles de reconfiguration, d'où un arrêt de service potentiellement élevé. Le deuxième procédé diminue fortement le nombre d'opérations mais limite la mise à jour interne à des cas précis.

Le procédé général consiste, afin de modifier le comportement interne d'un module m_1, à introduire dans l'application un nouveau module m2 dont le comportement interne est le comportement interne de m1 modifié (nouvelle version de m_1), et dont l'état interne lors de son lancement est initialisé au regard de l'état de m_1 lors de son arrêt. L'opération est la suivante :

- Remplacement d'un module : l'algorithme de remplacement de m_1 par m_2 est le suivant :
  1. chargement des nouvelles classes de m_2 ;
  2. instanciation de m_2 ;
  3. suspension de m_1 ;
  4. encodage de l'état de m_1 ;
  5. positionnement de l'état de m_2 en fonction de l'état de m_1 ;
  6. démarrage de m_2 ;
  7. destruction de m_1.

Si l'avantage de cette méthode est d'autoriser tout type de modification sur le module m1 dans la mesure où l'ensemble de l'état interne de m1 est utilisé pour initialiser m2, elle présente trois inconvénients majeurs :

1. entre la suspension de m_1 et le démarrage de m_2, de nombreuses opérations logicielles sont réalisées : encodage de l'état de m_1, positionnement de l'état de m_2. Ces opérations sont coûteuses du point de vue temps de calcul, ce qui peut se traduire par un arrêt de service du point de vue utilisateur.
2. Deux modules cohabitent simultanément, entre les opérations 1 et 6, ce qui induit un gaspillage des ressources. On pourrait éventuellement détruire m_1 après avoir encodé son état, puis créer m_2, instancier les objets de m_2, positionner l'état de m_2 en fonction de celui de m_1 et enfin démarrer m_2. Cet algorithme optimise les ressources mais accroît le temps d'arrêt applicatif d'où une probabilité plus élevée de non continuité de service.
3. L'intervention du programmeur, qui est obligé de prévoir, au sein de m_2, une méthode permettant d'initialiser l'état de m_2 en fonction de celui de m_1.

Nous proposons un procédé de modification interne des modules qui supprime en partie les trois inconvénients précités au prix d'une certaine réduction de la capacité de modification. Nous proposons en effet d'introduire deux opérations :

- Modification de tous les objets d'une classe donnée : cette opération permet de modifier tous les objets d'une classe donnée en objets d'une nouvelle classe, au sein d'un module donné. L'algorithme associé à cette opération est le suivant :

On suppose que l'ancienne classe est C, la nouvelle C'. Soit $O = \{o_i, i \in [1..n]\}$ l'ensemble des objets du module m concerné par l'opération, tel que $\forall\ i \in [1..n]$, classe(o_i) = C. Soit O' l'ensemble des nouveaux objets.

1. chargement si besoin de la nouvelle classe (chargement de C') ;
2. suspension de m ;
3. les objets $o_{i'}$ de la classe C' sont instanciés et leur état interne est positionné selon l'état interne de o_i (méthode spécifique à C' à la charge du programmeur) ;
4. mise à jour de tous les attributs de m ayant pour valeur o_i : l'attribut vaut alors o'_i ;
5. réactivation de m ;
6. destruction des objets o_i ;
7. destruction de C si C n'est plus utilisée.

Cet algorithme corrige certains des inconvénients précédents. Tout d'abord, la trace mémoire est diminuée car l'encodage des états ne concerne pas la totalité du module, mais un sous-ensemble. Ensuite, l'arrêt de service est diminué car le positionnement des états des objets est moins complexe que le positionnement des états internes de tous les objets d'un module. Enfin, la charge du programmeur est réduite à la recopie de l'état des objets modifiés et non à tous les objets du module.

Toutefois, cet algorithme présente deux inconvénients que l'on retrouve dans les opérations de modification des applications développées selon une approche fonctionnelle. Tout d'abord, l'impossibilité de modifier la pile d'appel d'un module implique qu'aucun objet modifié ne soit en cours d'appel lorsque le module est dans l'état « suspendu ». De plus, à l'instar des modifications dans les applications développées selon le modèle fonctionnel, il n'est pas possible de modifier l'objet de plus haut niveau d'un module. Ensuite, le maintien de la cohérence logique de l'architecture objets implique que la signature des attributs et méthodes de la classe C' et C soient identiques [Malabarba 2000]. Cette restriction se traduit par :

- C hérite de C'. Considérons l'exemple illustré par la Figure 5-2-(a) : un objet de classe C1 référence un objet de classe C2. Pour modifier le comportement de C2, alors il convient d'introduire une nouvelle classe C3 qui hérite de C2 : l'objet o2 instance de C2 est détruit une fois que l'objet o3 instance de C3 est créé. Le lien entre o1 et o2 est alors remplacé par un lien entre o1 et o3 (modification d'attribut).
- Si les objets du module impliqué dans la modification ne référencent jamais les objets de la classe C mais uniquement une surclasse de C (appelée D), alors C peut être remplacée par tout héritier de D. Dans le cas (b) de la Figure 5-2, la classe C1 référence une classe C2. En supposant que dans un premier temps, l'instance de C1 référence un objet o3 de la classe C3 qui hérite de C2, alors il est possible de remplacer o3 par un objet o4 à condition que la classe de o4 hérite de C2. Une fois le remplacement effectué, la classe C3 peut-être supprimée car C4 ne dépend que de C2 (et des classes dont C2 hérite).

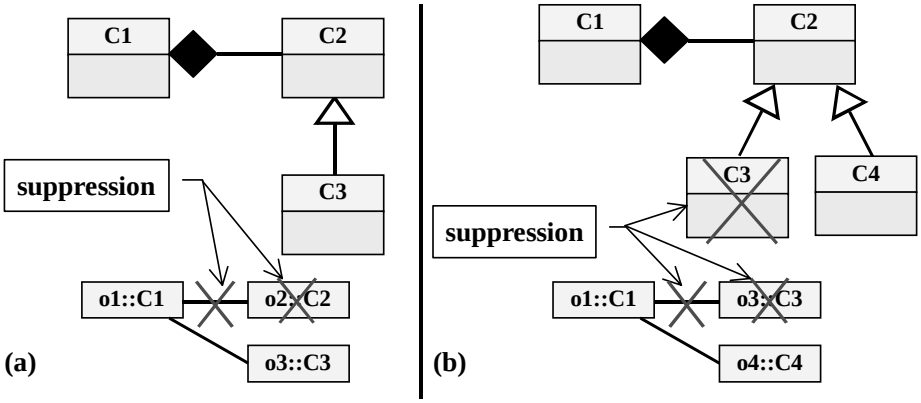


Figure 5-2. Modélisation d'un changement interne de classe. (a) Par héritage direct. (b) Par classe intermédiaire.

- Modification d'un objet o d'une classe c : l'objet o instance de c est transformé en un objet d'une nouvelle classe. L'algorithme est identique au précédent en prenant O = {o}.

5.3.3.3 *Opérations supports*

Les opérations suivantes sont des opérations dont le rôle est d'offrir un support aux autres opérations de reconfiguration : il s'agit donc d'opérations de bas niveau.

Opération de gestion de la mémoire applicative

- Chargement d'une classe : le chargement d'une classe consiste à ajouter à la mémoire applicative une classe non présente dans cette mémoire.
- Destruction d'une classe : la destruction d'une classe consiste à retirer de la mémoire applicative une classe. Celle-ci ne peut être supprimée que si aucune instance de cette classe n'est présente dans l'application.
- Instanciation d'un objet : l'instanciation d'un objet consiste à ajouter à la mémoire applicative une nouvelle instance d'une classe. La classe de cet objet doit bien entendu figurer dans la mémoire applicative.
- Destruction d'un objet : la destruction d'un objet consiste à retirer de la mémoire applicative une instance de classe.
- Lecture de la valeur d'un attribut d'un objet : cette opération consiste à récupérer la valeur de l'attribut d'un objet.
- Modification d'un attribut d'un objet : la modification d'un attribut d'un objet consiste à modifier la valeur d'un attribut d'un objet d'un module.

Opérations de gestion de l'état d'exécution des modules

Ces opérations représentent les opérations supports des *changements d'état* (cf. 3.4.8.4) d'un système applicatif.

- Chargement d'un module : le chargement d'un module consiste à charger toutes les classes de ce module qui ne sont pas présentes dans la mémoire applicative.
- Instanciation d'un module : l'instanciation d'un module consiste à instancier tous les objets de ce module. Les modules communiquant uniquement par échange d'informations au travers des ports, tous les objets propres à un module doivent être crées lors de l'instanciation de la tâche logicielle du module. En effet, le mode de communication entre modules contraint les modules à ne pas partager d'objets.
- Lancement d'un module : le lancement d'un module consiste à démarrer le flux d'exécution d'un module dont les objets sont déjà instanciés.

- Suspension d'un module : la suspension d'un module consiste à passer un module en état nominal de fonctionnement dans l'état « suspendu ».
- Réactivation d'un module : la réactivation d'un module consiste à passer un module dont l'état courant est « suspendu » dans l'état « actif ».

Opération de gestion de l'état interne des modules

- Encodage de l'état d'un module : l'opération d'encodage de l'état d'un module m consiste à générer un objet représentant l'état interne d'exécution de m, i.e. l'ensemble des états internes des objets de m, et la valeur de tous les attributs des objets de ce module m. L'opération d'encodage ayant une durée non nulle, il convient que le module ne soit pas en cours de fonctionnement (état « actif ») lors de la réalisation de cette opération.
- Positionnement de l'état d'un module : l'opération de positionnement de l'état d'un module consiste à passer en paramètre à un module l'encodage de l'état d'un autre module. Il est alors à la charge du module d'utiliser cet état pour modifier son état interne. Cette opération ne peut se réaliser si le module est dans l'état « actif ».
- Encodage de l'état d'un objet d'un module : l'opération d'encodage de l'état interne d'un objet d'un module consiste à générer un objet représentant l'état interne de cet objet c'est-à-dire la valeur de tous les attributs de cet objet. L'opération d'encodage ayant une durée non nulle, cela suppose qu'aucun appel ne soit en cours ou ne soit réalisé sur cet objet pendant l'encodage.
- Positionnement de l'état d'un objet d'un module : l'opération de positionnement de l'état d'un objet d'un module consiste à passer en paramètre à un objet o_1 un objet o_2 qui représente l'encodage d'un objet o_3. Il est alors à la charge du programmeur de l'objet o_1 de savoir utiliser l'information stockée dans o_2 pour générer un nouvel état à l'objet o_1.

5.3.3.4 *Tableau synoptique des opérations*

Un tableau synoptique des opérations de reconfiguration est proposé en annexe de ce mémoire (cf. Tableau 9.1, page 135).

5.4. Langage de reconfiguration dynamique

5.4.1 Besoin d'un langage

La reconfiguration d'une application nécessite l'exécution sur cette application d'un ensemble d'opérations de reconfiguration dynamique. Ces opérations possèdent des contraintes d'ordonnancement logique et temporel [Tyrrell 1992], ce qui nous conduit à proposer un langage permettant l'expression des scénarios valides (du point de vue logique et temporel) d'exécution des opérations de reconfiguration.

5.4.1.1 *Spécification informelle des besoins en termes de modélisation des opérations de reconfiguration*

L'analyse des opérations de reconfiguration des autres contributions [Kramer 1985][Tyrrell 1992][Purtilo 1994] nous incite à proposer la spécification :

- de l'exécution des opérations de reconfiguration. Cela concerne le succès de la réalisation d'une opération sur un système logiciel dans un intervalle temporel rigoureusement défini ;
- de l'ordre partiel associé à l'exécution des opérations de reconfiguration dynamique. En effet, l'exécution de ces opérations est contrainte par des préconditions qui se traduisent par des dépendances logiques entre opérations. Ces dépendances se traduisent par des relations série / parallèle entre l'exécution de ces opérations.

- Des contraintes logicielles identifiées lors de la phase de validation formelle de l'exécution du scénario de reconfiguration. Nous avons en effet montré dans le paragraphe 4.4.4.3 que l'identification d'erreurs lors de la phase de validation formelle peut conduire à l'introduction dans le scénario de reconfiguration de nouvelles contraintes logiques et temporelles.
- De la réversibilité du scénario de reconfiguration. La violation d'une contrainte logique ou temporelle d'exécution d'une opération se traduit par l'arrêt de l'exécution du scénario puis par l'exécution d'un nouveau scénario permettant de ramener l'application dans une configuration identique à celle avant le démarrage de l'exécution du scénario nominal de reconfiguration.

5.4.1.2 *Choix d'un formalisme de spécification*

De nombreuses contributions du domaine de la reconfiguration dynamique choisissent de spécifier les opérations de reconfiguration dynamique et leurs contraintes par utilisation d'un langage de script propriétaire [Kramer 1985][Purtilo 1994][Cailliau 2001]. Cette approche présente trois inconvénients majeurs :

- l'aspect propriétaire du formalisme introduit ;
- l'interprétation embarquée d'un tel script, dont l'implémentation nécessite de mettre en place un analyseur syntaxique ;
- l'absence de technique de validation formelle de l'exécution du scénario. En effet, nous désirons que l'exécution de la spécification des opérations puisse être validée formellement à partir du modèle TURTLE de l'application. Mais il apparaît assez complexe d'écrire en TURTLE un analyseur syntaxique de script. Nous préférons donc, par souci de simplicité, dériver le script directement sous forme d'une modélisation TURTLE. Pour cela, il convient d'avoir des relations assez directes entre le formalisme TURTLE et le formalisme de la spécification de reconfiguration.

Pour remédier aux inconvénients énumérés ci-dessus, nous proposons l'utilisation d'un formalisme éprouvé, permettant la modélisation de l'ordre partiel entre des opérations et des contraintes logiques et temporelles associées à ces opérations. Malgré son pouvoir d'expression, le formalisme RT-LOTOS [Courtiat 1996] nous paraît peu adapté en raison de la complexité que représenterait l'intégration embarquée à bord d'un satellite d'un analyseur RT-LOTOS. A contrario, les interpréteurs de *réseau de Petri* sont assez simples à mettre en œuvre (un tel interpréteur a été implémenté en Java, cf. [Apvrille 2000]). De plus, le modèle réseau de Petri est un formalisme permettant de spécifier de façon explicite des dépendances entre opérations (expression de l'ordre partiel, cf. [Rojas 2000]). Toutefois, il convient d'utiliser des réseaux de Petri permettant la modélisation explicite de contraintes temporelles (notamment, les intervalles temporels d'exécution des opérations). Pour cela, nous proposons d'utiliser le modèle *réseau de Petri à flux Temporels* (RdPFT) introduit dans [Diaz 1993] et [Sénac 1996], et qui a été utilisé avec succès pour la modélisation de contraintes de présentation d'applications multimédias et pour la modélisation d'applications temps-réel [Diaz 1994][Owezarski 1996][Rojas 2000].

5.4.2 Réseaux de Petri à Flux Temporels pour la Reconfiguration Dynamique

5.4.2.1 *Réseaux de Petri à Flux Temporels*

Le modèle réseaux de Petri à flux temporels RdPFT est une extension du modèle des réseaux de Petri, dont le rôle est de définir l'expression de contraintes temporelles et de synchronisation dans un système faiblement synchrone. L'indéterminisme temporel est modélisé, au niveau des arcs reliant une place p à une transition t, par l'introduction d'un 3-uple qui représente respectivement la durée minimale x, nominale n et maximale y d'attente d'un jeton j au niveau de la

place p avant que la transition t soit tirée (cf. Figure 5-3). Soit τ l'instant d'arrivée du jeton *j* sur une place p. On appelle intervalle de validité temporelle (IVT) d'un arc reliant la place p à une transition t_k l'intervalle temporel [τ+*min*, τ+*max*]. Cet intervalle modélise l'ensemble des instants pour lesquels la transition t peut-être tirée.

Il a été montré qu'il n'est pas possible, dans le cas général, de satisfaire l'ensemble des contraintes temporelles associées aux arcs reliant des places à une même transition. Le problème est résolu par l'introduction d'un typage au niveau des transitions, typage qui représente une nouvelle sémantique de synchronisation fondamentale. Les trois plus importantes sont *et*, *ou* et *maître*. Elles permettent de privilégier respectivement le traitement le plus en retard, en avance ou un traitement donné.

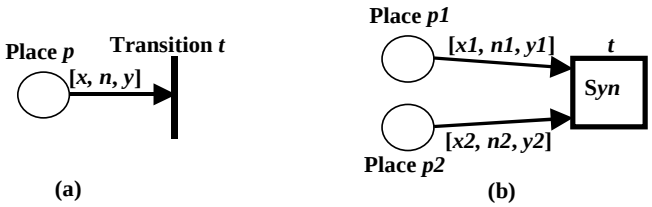


Figure 5-3. Modélisation de schémas de synchronisation : (a) simple, et (b) avec typage de la transition.

5.4.2.2 *Utilisation ad-hoc des RdPFT pour modéliser des opérations de reconfiguration dynamique*

Nous désirons utiliser les RdPFT afin de modéliser les scénarios de reconfiguration dynamique.

L'idée générale est d'associer aux places des RdPFT une opération de reconfiguration dynamique et de construire un RdPFT dont la structure de synchronisation modélise les dépendances entre opérations. De plus, les contraintes temporelles statiques peuvent être modélisées par des intervalles temporels qui sont associés aux arcs.

Cependant, le modèle RdPFT présente des limites en termes de pouvoir de modélisation du succès ou de l'échec de l'exécution d'une opération. En effet, il convient d'examiner la valeur de retour de chaque opération o avant d'exécuter les opérations qui dépendent de o : une transition (qui représente une dépendance) ne doit donc être tirée qu'en cas de code de retour satisfaisant de la part de toutes les opérations reliées à cette transition. Pour cela, nous proposons d'étendre les RdPFT en modélisant les codes de retour des opérations directement au sein du modèle, par utilisation de transitions typées et de jetons colorés [Jensen 1997]. Ainsi, en supposant le réseau de Petri sain, une transition ne peut-être tirée que si les places initiant le tirage possèdent un jeton dont la valeur est compatible avec la garde de cette transition.

5.4.2.3 *Définition d'un RdPFT pour la modélisation d'opérations de reconfiguration dynamique*

Nous définissons par la suite un Réseau de Pétri à Flux Temporels pour la modélisation d'opérations de Reconfiguration Dynamique (RdPFT-RD). Les RdPFT-RD enrichissent les RdPFT comme suit :

- à chaque place est associée une fonction f ;
- l'arrivée d'un jeton au niveau d'une place déclenche l'exécution de la fonction associée à cette place ;
- une valeur entière (ou couleur) est attribué à chaque jeton (par défaut, la valeur 0) ;
- tout nouveau jeton est généré avec la valeur 0 ;
- la fonction déclenchée par l'arrivée d'un jeton dans une place modifie la valeur de ce jeton comme suit : en cas de succès de la fonction, la valeur 1 est attribuée au jeton. Sinon, la valeur 2 lui est attribuée.

– A toute transition est attribuée une *garde* constituée d'une liste de valeurs entières : une transition n'est tirable que si elle est sensibilisée par des jetons dont la valeur est incluse dans la garde de la transition.

Un RdPFT-RD se définit formellement comme suit :

Définition 6. RdPFT-RD

Un RdPFT-RD est un n-uplet (P, T, *Pré*, *Post*, M_0, J, *IVT*, *Syn*, *V*, *G*) avec :

– P l'ensemble des places
– T l'ensemble des transitions
– *Pré* représente la fonction associée à un arc reliant une place à une transition. Cette fonction est évaluée dynamiquement c'est-à-dire avant chaque tir de transition. On a : $\forall t \in T$ / *Syn*(t) = et, $\forall p \in {}^\circ t$, on a *Pré*(p, t) = $\{j_i\}$

Nous ne considérons que les transitions de type *et* dans la mesure où les besoins de modélisation des opérations de reconfiguration ne font pas apparaître un besoin clair d'utilisation des autres types de transition.

– *Post* représente la fonction associée à un arc reliant une transition à une place. Cette fonction est évaluée dynamiquement c'est-à-dire après chaque tir de transition. On a : $\forall t \in T \; \forall p \in t^\circ$, on a *Post*(p, t) = $\{j_0\}$.
– M_0 est le marquage initial.
– J est l'ensemble des jetons de valeur 0, 1 et 2. On a J = $\{j_0, j_1$ et $j_2\}$.
– *IVT* est la fonction qui associe à tout arc reliant une place à une transition un intervalle de validité temporelle. *IVT* est définie de la manière suivante :
Soit A^+ l'ensemble des arcs qui relient les places et les transitions d'un RdPFT-RD :
$A^+ = \{(p,t) \in P \times T / Pré(p,t) \neq \emptyset\}$

$$IVT : \; A+ \rightarrow \mathbb{Q}^+ \times \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{+\infty\})$$
$$ai \mapsto (xi, ni, yi) \mid xi \leq ni \leq yi$$

– *Syn* est une fonction qui prend en paramètre une transition, et retourne le typage de cette transition (*et*). Nous ignorons les 8 autres types de transitions introduits dans [Sénac 1996]. La fonction *Syn* est définie comme suit :

$$Syn : T \rightarrow \{et\}$$

– *V* est une fonction qui retourne les valeurs entières associées au marquage d'une place p. Si le marquage de la place p est l'ensemble vide, alors la fonction retourne un ensemble vide. La fonction *V* se définie comme suit :

$$V : \; P \rightarrow \{0, 1, 2\}$$
$$p \mapsto E \subset \{0, 1, 2\}$$

– G est une fonction qui retourne l'ensemble des valeurs associées à une transition. Cet ensemble est non vide et est inclus dans {0, 1, 2}. Il représente la *garde* associée à cette transition.

$$G : \; T \rightarrow \{0, 1, 2\}$$
$$t \mapsto E \subset \{0, 1, 2\} \text{ et } E \neq \emptyset$$

La représentation graphique d'une transition possédant une garde entière est identique à celle d'une transition classique à laquelle est ajoutée le texte suivant : [v] avec v représentant la valeur associée à la transition, si cette valeur est unique. En cas de valeur multiple $\{i_1, i_2, .., i_n\}$, la représentation graphique est la suivante : $[i_1, i_2, .., i_n]$. Enfin, la représentation graphique de l'ensemble {0, 1, 2} est [*]. Cette dernière représentation graphique est la représentation graphique par défaut et est par là même facultative.

5.4.2.4 *Sémantique formelle d'un RdPFT-RD*

La sémantique formelle d'un RdPFT-RD est donnée d'une part par l'aspect statique du RdP (l'état du RdP c'est-à-dire son marquage et la valeur des intervalles temporels sensibilisés) et d'autre part par l'aspect dynamique du RdP c'est-à-dire par la description de l'évolution du RdP d'un état à un autre.

Etat d'un RdPFT-RD

L'état d'un RdPFT-RD est donné à tout instant par un couple S = (M, I) où

- M représente le marquage courant du RdPFT-RD,
- I est la liste des intervalles de validité temporelle des arcs sensibilisés par le marquage courant. Un arc a = (p, t) est sensibilisé si $j \in M(p)$.

Règles d'évolution entre états

L'état du réseau évolue dans deux cas : la terminaison d'une fonction associée à une place et le tir d'une transition.

La terminaison de la fonction f de la place p se traduit par la modification de la valeur du jeton j ayant initiée cette fonction. Le jeton se voit attribué la valeur du code de retour de la fonction. Le marquage M' se définit formellement comme suit : $M'(p) = M(p) + \{j_i\} - \{j_0\}$ avec i représentant le code de retour de la fonction (i=1 représente la cas nominal, i=2 représente un cas d'erreur).

Le tir d'une transition t dans l'état S = (M, I) à l'instant relatif θ par rapport à la date d'occurrence de cet état se traduit par un nouvel état S' = (M', I') calculé de la façon suivante :

- le nouveau marquage est calculé ainsi : un jeton de valeur compatible avec la valeur de la garde de la transition t est retiré de toute place en amont de la transition t, toute fonction associée aux jetons retirés est immédiatement stoppée et un jeton dont la valeur est 0 est généré pour toute place p en aval de la transition. Le marquage M' se définit formellement comme suit :

$$\forall p \in P, M'(p) = M(p) - Pré(p, t) + Post(p, t)$$

- L'ensemble I' est calculé par application des trois étapes suivantes :
  1. Retirer de l'ensemble I tous les intervalles dynamiques (tous les triplets) correspondant aux arcs qui étaient sensibilisés dans l'état S et qui ne le sont plus suite au tir de la transition t.
  2. Réduire d'une valeur θ vers l'origine des temps les bornes (i.e. bornes minimum, maximum et durée nominale) de tous les intervalles dynamiques de validité temporelle restant dans l'ensemble I suite à l'étape précédente. Borner par zéro les valeurs des nouvelles bornes minimum et maximum ainsi obtenues. Ainsi l'intervalle dynamique (x, n, y) est remplacé dans I par $(\max(0, x - \theta), n - \theta, \max(0, y - \theta))$.
  3. Introduire dans l'ensemble I les intervalles de validité temporelle des arcs nouvellement sensibilisés ou resensibilisés suite au tir de t.

Règles de tir des transitions « et » des RdPFT-RD

Nous reprenons la sémantique de tir introduite dans [Sénac 1996] qui spécifie que l'intervalle de synchronisation d'une transition *et* est égal à l'intersection des intervalles lorsque celle-ci est non vide, et égal à un instant de synchronisation égal au maximum des bornes inférieures des intervalles absolus de validité temporelle lorsque cette même intersection est vide. Dans le cas des RdPFT-RD, nous introduisons, pour sensibiliser une transition, le fait que le marquage des places p en aval de la transition considérée doit être compatible avec la garde de cette transition, c'est-à-dire que la valeur du marquage est incluse dans l'ensemble des valeurs associées à la garde.

Définition 7. Tir d'une transition « et »

Une transition t_k telle que $Syn(t_k)$ = et telle que $\forall a_i \in A^+_k$, $IDVT(a_i) = (\alpha_i, d_i, \beta_i)$ est tirable à l'instant θ à partir de l'état S = (M, I) si et seulement si les deux conditions suivantes sont satisfaites :

- la transition t_k est sensibilisée i.e. le marquage des places en amont est non vide et la couleur des jetons sensibilisant la transition est incluse dans les couleurs de la garde de la transition :
  $\forall p \in {}^\circ t_k, M(p) \neq \varnothing \wedge V(p) \subset G(t_k)$

- l'instant relatif θ satisfait les inégalités suivantes :
  $MIN_k = \max_i(\alpha_i) \leq \theta \leq \max_i(\min_i(\beta_i), \max_i(\alpha_i)) = MAX_k$

5.4.3 Spécification des opérations de reconfiguration

5.4.3.1 *Introduction*

Par la suite, nous présentons la spécification des scénarios de reconfiguration dynamique. Cette spécification doit pouvoir exprimer l'ordonnancement logique et temporel des opérations de reconfiguration dynamique. Pour cela, nous montrons la spécification :

- de l'exécution d'une opération de reconfiguration dynamique : modélisation de la nature de l'opération et de son profil temporel d'exécution ;
- de l'ordonnancement logique des opérations : cela se traduit par la spécification des dépendances logiques entre les opérations ;
- des contraintes applicatives à respecter lors de l'application des opérations de reconfiguration sur cette application ;
- des retours en arrière en cas de violation des contraintes listées ci-dessus.

5.4.3.2 *Spécification de l'exécution des opérations*

Soit O = $\{o_i, i \in [1..n]\}$, un ensemble d'opérations de reconfiguration dynamique. Il convient de modéliser l'exécution de ces opérations sur une application A. Chaque opération o_i est modélisée par une place p_i. L'exécution d'une opération est représentée par l'arrivée d'un jeton sur la place de l'opération. La valeur associée à la place est tout d'abord 0, puis lorsque l'opération est achevée, la valeur de la place représente le code de retour de l'opération : 1 en cas de succès, 2 en cas d'échec. Ainsi, dans le cas où l'opération échouerait, il convient de dérouter le RdP dans un cas spécifique de gestion d'erreur. Prenons l'exemple de la modélisation d'une opération o_1 (cf. Figure 5-4).

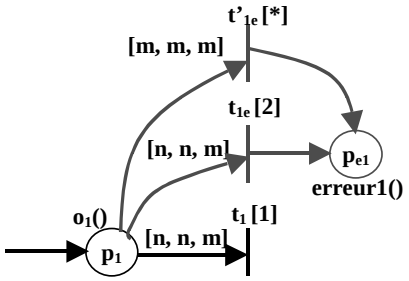


Figure 5-4. Spécification de l'exécution d'une opération o1() avec un RdPFT-RD.

La place associée à cette opération est p_1, la transition associée à p_1 est nommée t_1, à laquelle est associée la garde [1]. Pour traiter le cas d'échec de l'opération (code de retour = 2), nous modélisons de plus une transition t_{1e} dont la garde vaut « [2] ». Cette transition conduit par la suite à un cas d'erreur (place p_{e1}, l'opération *erreur1*() étant attachée à la place).

Afin de spécifier le temps d'exécution d'une opération, nous utilisons les intervalles temporels. Soit n le temps d'exécution minimal d'une opération, soit m le temps maximal toléré pour l'exécution de cette opération. L'intervalle temporel associé à l'opération est donc [n, n, m]. Cet intervalle temporel est valable quel que soit le code de retour de l'opération. Afin de modéliser le traitement d'erreur lorsque l'opération n'a ni échoué ni réussi au bout de m unités de temps, nous insérons une transition supplémentaire (t'_{1e} sur la Figure 5-4) qui permet de faire évoluer le RdP une fois m unités de temps écoulées : dans ce cas, le même traitement d'erreur que lorsque l'opération retourne une code d'erreur est appliqué (place p_{1e}). L'arc allant de p_1 à t'_{1e} modélise donc un *timer*, l'intervalle de validité temporel associé étant [m, m, m].

5.4.3.3 *Spécification de l'ordonnancement logique entre opérations*

Les dépendances entre opérations, pour un ensemble d'opérations données, s'expriment par utilisation des relations parallèle et série. Les relations *parallèle* sont utilisées pour spécifier des relations de non dépendance entre opérations, alors que les relations de dépendance s'expriment au travers de relation série.

Soit $O = \{o_i, i \in [1..n_o]\}$, un ensemble d'opérations de reconfiguration dynamique et soit $D = \{d_i, i \in [1..n_d]\}$, l'ensemble des dépendances entre ces opérations, avec $d_i = (o_j, o_k)$ signifie que o_k dépend de o_j. Nous supposons qu'il n'existe pas de cycle de dépendances.

Le RdPFT-RD associé à O et D est construit comme suit :

1. Initialisation des places du RdP : $P = P \cup (p_0, \textit{debut()})$: on ajoute à P une place p_0 à laquelle est associée une opération *début*() qui retourne instantanément la valeur 1. De plus, $P = P \cup (p_{fin}, \textit{fin()})$, la dernière place du réseau de Petri, à laquelle l'opération *fin*() est associée : cette opération ne se termine jamais i.e. ne retourne jamais de valeur. De plus, pour toute opération o_i, on génère une place p_i : $P = P \cup \bigcup_{i=1}^{i=n} (p_i, o_i)$.
2. Initialisation des transitions du RdP : $T = T \cup t_0 \cup t_{fin}$ avec t_0 qui représente la première transition, et t_{fin} la dernière. De plus, $A^+ = A^+ \cup (p_0, t_0)$ et $A^- = A^- \cup (t_{fin}, p_{fin})$.
3. Elimination des dépendances par transitivité c'est-à-dire pour tout i_1, i_2, i_3, j, k, l où l'on a $d_{i1} = (o_j, o_k)$ et $d_{i2} = (o_k, o_l)$ et $d_{i3} = (o_j, o_l)$ alors on supprime d_{i3} de D.
4. A toute place p_i, nous associons un arc vers une transition t_i, sauf s'il n'existe pas d'opération o_k qui dépende de o_i : dans ce cas : on construit un arc de la place p_i directement vers la transition t_{fin}. Cela se traduit par : $\forall i \in [1..n_o]$, si $\exists k\ i \in [1..n_o]$, $k \neq i$ / $(oi, ok) \in D$ alors $T = T \cup t_i$ et $A^+ = A^+ \cup (p_i, t_i)$ sinon $A^+ = A^+ \cup (p_i, t_{fin})$.
5. Pour toute opération qui ne dépend d'aucune opération, on construit un arc entre la transition de début, et la place de l'opération i.e. pour toute opération i telle que : $\forall j \in [1..n_o]$, $(o_j, o_i) \notin D$, $A^- = A^- \cup (t_0, p_i)$.
6. Cette étape consiste à traduire les relations de dépendances logiques entre les opérations par des relations de synchronisation (transitions). Lorsqu'une opération o_i ne dépend que d'une seule autre opération o_j, il suffit de construire un arc entre la transition t_j et la place p_i. Cela se traduit par $\forall\ i \in [1..n_o]$, $\exists!\ j \in [1..n_o]$ / $(o_j, o_i) \in D \Rightarrow A^- = A^- \cup (t_j, p_i)$. Par contre, si l'opération o_i dépend de plus d'une opération, alors pour toute opération o_k dont elle dépend, on construit une place p_{ki} à laquelle l'opération *id*(1) est associée (l'opération *id*(x) est une opération qui retourne instantanément la valeur passée en paramètre, c'est-à-dire x).et une transition t^*_i et un arc entre t_k et p_{ki}, et un arc entre p_{ki} et t_{ki}. Cela se traduit par : $\forall (i,k) \in [1..n_o]^2$, $(o_k, o_i) \in D \Rightarrow P = P \cup (p_{ki}, id(1)) \wedge T = T \cup t_{ki} \wedge A^- = A^- \cup (t_k, p_{ki}) \wedge$ si $(p_{ki}, t^*_i) \notin A^+$ alors $A^+ = A^+ \cup (p_{ki}, t^*_i)$.

A noter aussi que les IVT associés à chaque arc sortant d'une place sont de type [0, *, +∞], car les contraintes temporelles ne sont pas prises en compte dans cet algorithme. De plus, nous ne mettons pas en évidence la gestion des cas d'erreurs (code de retour des opérations).

Considérons un exemple permettant d'illustrer l'algorithme de génération du RdPFT-RD en ce qui concerne les dépendances entre opérations. Considérons O = {o_1, o_2, o_3, o_4, o_5, o_6} et D = {(o_1, o_2), (o_2, o_3), (o_2, o_4), (o_1, o_4), (o_5, o_6), (o_5, o_3)}. Les trois premières étapes consistent à construire le RdP initial (cf. Figure 5-5), comprenant les places d'initialisation et de fin, ainsi que les deux transitions associées (t_0 et t_{fin}), et les 6 places correspondant aux 6 opérations o_i. L'étape 3 consiste à retirer de D l'élément {(o_1, o_4)} qui est inutile dans la mesure où o_4 dépend de o_2 qui elle même dépend de o_1. Lors de l'étape 4, pour chaque opération o_i qui admet une opération o_k / (o_i, o_k) ∈ D, on crée une transition t_i, et un arc de la place de o_i vers t_i : o_1, o_2 et o_5 respectent ces conditions : nous créons donc trois transitions t_1, t_2 et t_5, et nous créons un arc respectivement entre p_1 et t_1, p_2 et t_2, et enfin p_5 et t_5. A contrario, o_3, o_4 et o_6 ne respectent pas ces conditions : nous créons donc un arc entre les places p_3, p_4 et p_6 et t_{fin}. A l'étape 5, il s'agit d'ajouter un arc entre t_0 et la place de toute opération qui ne dépend d'aucune autre. C'est le cas de o_1 et o_5 : un arc est donc crée entre t_0 et p_1 et un autre entre t_0 et p_5. A l'étape 6, il s'agit de considérer les relations de dépendances l'une après l'autre, et de générer pour chaque relation de dépendance multiple (o_i, o_j) une place intermédiaire p_{ij} (opération id(1)) et de créer un arc entre t_i et p_{ij}, entre p_{ij} et t^*_j et enfin entre t^*_j et p_j (sous réserve que ce dernier n'existe pas). Prenons le cas de (o_2, o_3) et (o_5, o_3). On crée une place p_{23} et une place p_{53} auxquelles est associée l'opération id(1). Pour (o_2, o_3), on crée un arc entre t_2 et p_{23}, on crée la transition t^*_3, on crée un arc entre p_{23} et t^*_3, et un arc entre t^*_3 et p_3. Ensuite, pour (o_5, o_3), on crée un arc entre t_5 et p_{53} et un arc entre p_{53} et t^*_3.

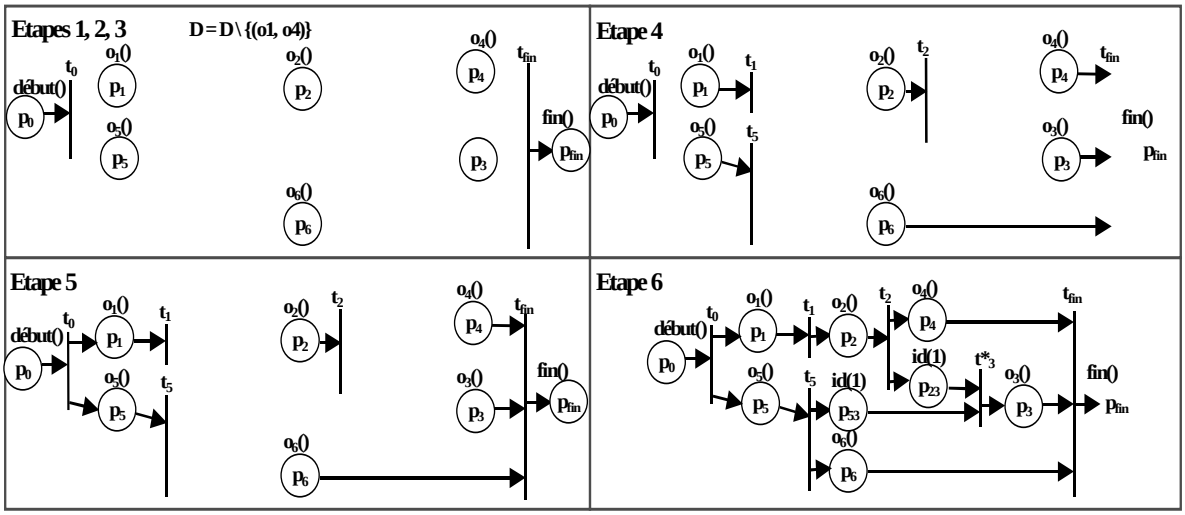


Figure 5-5. Modélisation des dépendances entre 6 opérations avec un RdPFT-RD.

5.4.3.4 *Spécification de contraintes logicielles*

Nous avons vu précédemment que l'exécution d'un scénario de reconfiguration sur une application soulève le problème du respect de contraintes intrinsèques et extrinsèques applicatives (notamment, la continuité de service). Le paragraphe 4.4.4.3 a montré que la validation formelle de l'exécution d'un scénario de reconfiguration peut amener à ajouter au scénario des contraintes logiques et temporelles. Nous montrons par la suite comment il est possible par utilisation des RdPFT-RD de modéliser ces contraintes logicielles.

Nous proposons de spécifier les contraintes logicielles par ajout au RdPFT-RD d'une nouvelle place par contrainte logicielle. A cette place, nous associons une fonction *f* représentant l'examen de cette contrainte sur le logiciel et qui retourne la valeur entière 2 en cas de violation de cette contrainte. Cette nouvelle place est intégrée à la spécification selon sa relation logique et temporelle avec les opérations de reconfiguration dynamique. Par exemple, considérons le cas où une contrainte logicielle doit être examinée entre l'exécution d'une opération o_1 et d'une opération o_2. Soit p_1 la place associée à l'opération o_1, et soit p_2 la place associée à l'opération o_2. Il s'agit de

tester la valeur de $f_1()$ (f_1 représente la contrainte que nous voulons examiner) tant que o_2 n'a pas terminé son exécution, et ce, depuis le début de l'exécution de o_1. Soit t^*_1 la transition telle qu'il existe un arc entre t^*_1 et p_1 et soit t_2 la transition avec une garde valant 1 et telle qu'il existe un arc entre p_2 et t_2. L'examen de $f_1()$ se traduit par l'ajout de la place p_{f1} au modèle (cf. Figure 5-6-(a). De plus, on ajoute un arc entre t^*_1 et p_{f1}, et un arc entre pf_1 et t_2 : cela correspond au cas où la contrainte est vraie. Pour modéliser le cas où la contrainte devient fausse, nous ajoutons une transition t_{f1} et un arc entre p_{f1} et t_{f1}. La garde associée à t_{f1} est [2], ce qui correspond au cas où $f_1()$ est faux : si la contrainte devient fausse, t_{f1} est tirée et le RdP passe dans un mode de gestion des erreurs (place p_e). A noter que les arcs de sortie de la place p_{f1} ne sont pas temporisés (intervalle [0, 0, +∞]) dans la mesure où aucune contrainte temporelle statique n'est associée à l'examen de la contrainte représentée par f_1.

L'expérience acquise lors des validations formelles de scénario de reconfiguration dynamique a montré l'importance d'introduire dans le scénario de reconfiguration le temps d'arrêt maximal d'un module logiciel pendant sa reconfiguration dynamique. En effet, de nombreuses opérations de reconfiguration dynamique supposent l'arrêt d'un certain nombre de modules impliqués par ces opérations. Or, ces modules sont directement en charge de services utilisateurs, ce qui implique une contrainte forte quant au temps de leur arrêt. S'il est possible de modéliser la contrainte « temps d'arrêt maximal entre l'arrêt et le redémarrage d'un module » par la méthode présentée ci-dessus, la capacité intrinsèque des RdPFT de modélisation d'intervalles temporels statiques nous incite à proposer un schéma de spécification simplifié pour cette contrainte particulière.

Soit *m* la valeur maximale du nombre d'unités de temps pouvant s'écouler entre le début de l'exécution d'une opération o_1, et la fin de l'exécution d'une opération o_2. o_1 est modélisée par une place p_1, o_2 par une place p_2 (cf. Figure 5-6-(b)). Soit t^*_1 la transition telle qu'il existe un arc entre t^*_1 et p_1. Soit t_2 la transition nominale de p_2 dont la garde vaut 1. Pour modéliser un *timer* de valeur *m*, nous créons une nouvelle place p_t (à laquelle l'opération id() est associée, cf. 5.4.3.2), un arc entre t^*_1 et p_t, et un arc entre p_t et t_2. Nous associons un intervalle temporel à l'arc entre p_t et t_2 qui est [0, 0, m]. Pour modéliser le cas où le *timer* expire (la transition t_2 n'est plus tirable), nous ajoutons au modèle une place p_e (place d'erreur), une transition t_{pt} : un arc relie p_t à t_{pt} et un autre arc relie t_{pt} à p_e. Le *timer* expirant à m unités de temps, nous associons l'intervalle temporel [m, m, m] à l'arc reliant p_t à t_{pt}.

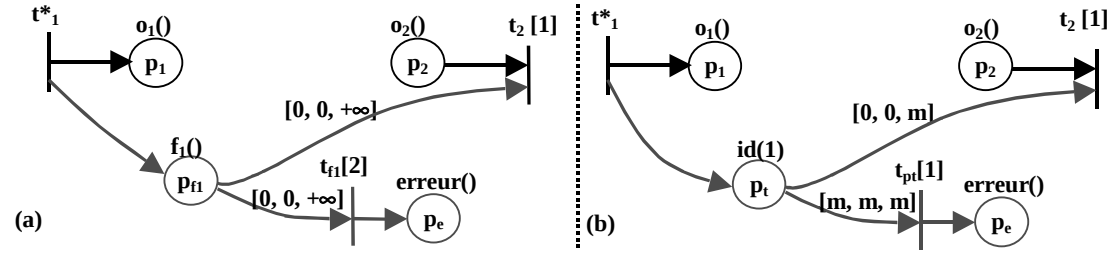


Figure 5-6. Spécification d'une contrainte entre deux opérations. (a) Cas général. (b) Cas d'une contrainte temporelle statique.

5.4.3.5 *Spécification des retours en arrière*

Principe général

Lorsqu'un cas d'erreur se produit lors de l'exécution du RdPFT-RD, il convient de modéliser les opérations à effectuer pour ramener l'application dans son état initial. Le principe est d'invoquer un autre RdPFT-RD qui modélise le retour en arrière. Il s'agit donc dans un premier temps d'expliciter ce RdP inverse, puis de générer des jetons dans ce RdP inverse pour que son exécution ramène l'application depuis le cas d'erreur rencontré dans le RdP nominal dans son état initial. Nous proposons pour cela la démarche suivante :

1. spécification des opérations de reconfiguration dynamique dans un RdPFT-RD dit nominal ;
2. construction et spécification d'un RdPFT-RD dit inverse, modélisant le retour en arrière dans le cas général ;
3. exécution du RdPFT-RD nominal ;
4. en cas d'erreur, arrêt de l'exécution du RdPFT-RD nominal (absorption des jetons), génération de jetons au niveau du RdPFT-RD inverse ;
5. exécution du RdPFT-RD inverse ;
6. en cas d'erreur lors de l'exécution du retour en arrière, on arrête l'exécution du RdPFT-RD inverse (absorption de tous les jetons).

Par la suite, nous détaillons chaque mécanisme de l'algorithme ci-dessus non décrit précédemment.

Construction du RdPFT-RD inverse

Le RdPFT-RD inverse représente l'ensemble des opérations à effectuer pour ramener une application ayant subi des opérations de reconfiguration dynamique dans son état initial. Pour cela, nous utilisons la notion d'*opération inverse* (cf. paragraphe 5.3.2). Soit R le réseau de Petri nominal et R' le réseau de Petri inverse. Soit $O = \{o_i, i \in [1..n_o]\}$, un ensemble d'opérations de reconfiguration dynamique.

R' est construit comme suit :

1. Pour toute place p associée soit à une opération de O, soit à une opération *id*() non associée à un contrainte logique ou temporelle, on génère une place p' dont l'opération associée est l'opération inverse de celle de la place p : si o est l'opération associée à p, alors $\overline{o}$ est l'opération de p'. Si $\overline{o}$ n'existe pas, l'opération n'est pas inversible, on insère l'opération *stop*() qui arrête toute exécution du RdP : il n'est pas possible d'inverser le processus.
2. On ajoute ensuite p' à l'ensemble P' des places de R' : $P' = P' \cup \{p'\}$.
3. Pour toute transition t de T, on génère une transition identique t' : $T' = T' \cup \{t'\}$.
4. Tout arc de R est repris dans R' en « l'inversant » : $\forall a_{ij} \in A^+$, $a_{ij}=(p_i, t_j)=$, on fait : $A'^- = A'^- \cup (t_j, p_i)$ et : $\forall a_{ij} \in A^-$, $a_{ij}=(t_j, p_i)=$, on fait : $A'^+ = A'^+ \cup (p_i, t_j)$.

Considérons l'exemple suivant : la Figure 5-7 met en évidence le RdPFT-RD inverse de celui représenté à la Figure 5-5 – étape 6. Toutes les opérations o_i ont été remplacées par leur opération inverse $\overline{o_i}$, p_i par p'_i et tous les arcs ont été inversés. L'opération début() admet comme inverse fin() et réciproquement, d'où le changement au niveau de l'état final et initial.

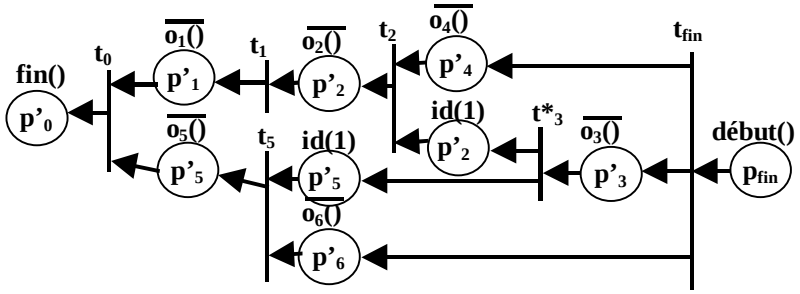


Figure 5-7. Exemple d'un RdPFT-RD inverse.

En ce qui concerne les contraintes logicielles (cf. 5.4.3.4), la spécification de ces contraintes dans le RdPFT-RD ne peut être automatisée car il n'y pas forcément de relation entre une

contrainte modélisée dans le RdPFT-RD nominal et une autre devant être modélisée dans le RdPFT-RD inverse. Par exemple, il n'existe par de relation entre l'intervalle temporel d'exécution associé à une opération o et l'intervalle temporelle d'exécution associé à $\overline{o}$.

Spécification de la relation entre un RdPFT-RD nominal et un RdPFT-RD inverse

Nous détaillons dans ce paragraphe la spécification de la gestion des cas d'erreurs qui se produisent lors de l'interprétation du RdPFT-RD nominal. Il s'agit, lorsqu'un cas d'erreur se produit (place *erreur* rencontrée) de modéliser l'absorption des jetons au niveau du RdPFT-RD nominal, et la génération de jetons au niveau du RdPFT-RD inverse. L'algorithme général est le suivant :

1. si l'erreur est provoquée depuis une place p_i représentant une opération de reconfiguration dynamique (mauvais code de retour, timer expiré), il convient de générer un jeton au niveau de la place p'_i, et de générer un jeton à toutes places p'_j pour laquelle p_j a un jeton. Cela est valable aussi pour les places de type *id* utilisées à des fins de modélisation de dépendance entre opérations (cf. paragraphe 5.4.3.3).
2. Tous les jetons des places modélisant des contraintes logicielles temporelles ou logiques sont tout simplement absorbés car il n'existe pas de place équivalente dans le RdPFT-RD inverse. En effet, l'existence d'une contrainte entre deux opérations n'implique pas qu'il existe une contrainte sur l'exécution de l'inverse de ces deux opérations.
3. Soit p' une place du RdPFT-RD inverse dans laquelle un jeton a été généré (phase 1 de l'algorithme). Il faut aussi générer un jeton pour toute place représentant une contrainte logique ou temporelle associée à l'opération représentée par p'. Ces contraintes sont identifiées lors de la validation formelle de l'exécution du scénario inverse de reconfiguration dynamique.

Considérons l'exemple suivant (cf. Figure 5-8) : un RdPFT-RD est constitué de trois places (p_1, p_2, p_3) représentant respectivement trois opérations o_1, o_2 et o_3.

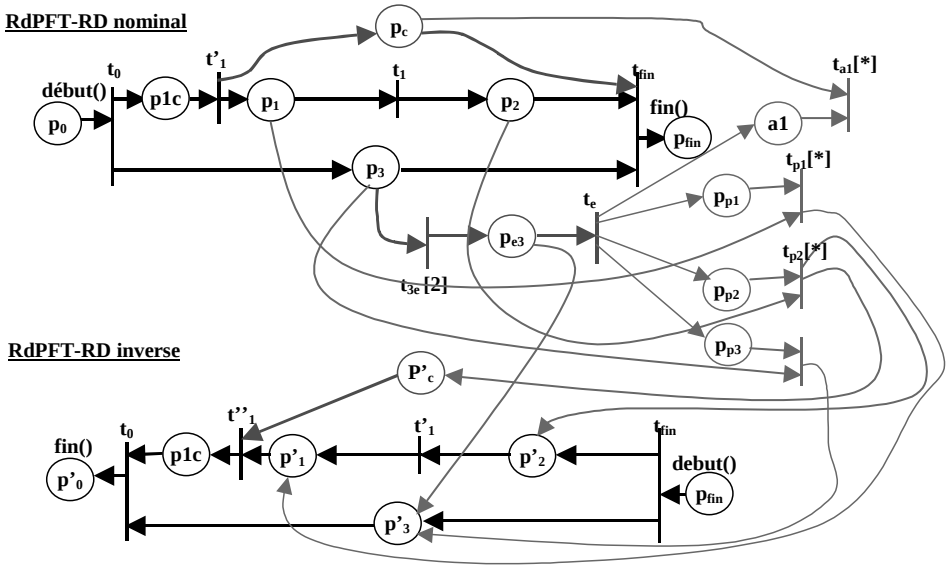


Figure 5-8. Modélisation de la gestion du transfert des jetons entre un RdP nominal et un RdP inverse.

On spécifie une contrainte logique entre o_1 et o_2, ce qui se traduit par la mise en place d'une place p_c associée à cette contrainte. Seule la gestion d'erreur de la place p_3 est représentée. Lorsqu'une erreur survient au niveau de la place p_3 (l'opération $o_3()$ génère un code de retour valant 2), un jeton est généré au niveau de la place p_{e3}. Un jeton est alors généré au niveau de la place p'_3 (RdPFT-RD inverse), puis un jeton est généré vers la spécification de l'examen du RdP nominal (algorithme décrit précédemment) : chaque place du RdPFT-RD nominal est examinée (nous n'avons pas modélisé l'examen de la place p_{1c} pour des questions de clarté) : chaque place p_{pi}

examine si un jeton est présent au niveau de la place p_i : si cela s'avère vrai, alors le jeton de la place p_i est absorbé (garde de type « * » au niveau de la transition t_{pi}) et un jeton est généré sur la place p'_i. En cas de contrainte associée à une opération du RdPFT-RD inverse (c'est le cas de p'_2), un jeton est généré par contrainte si un jeton est généré pour la place associée à cette contrainte. Ainsi, la transition t_{p2} génère deux jetons, un vers la place p'_2, et un vers la place p'_c qui représente une contrainte associée à la place p'_2. . Pour toutes les contraintes du RdP nominal, le jeton associé est absorbé, sans nouvelle génération : c'est le cas du jeton de la place p_c qui se retrouve absorbé via une synchronisation sur t_{a1} depuis la place a_1. La transition est tirée dans tous les cas car t_{a1} possède une garde de type *.

Spécification de la reprise d'erreur dans un RdPFT-RD inverse

Dans le cas d'erreur survenant dans le RdPFT-RD inverse, l'algorithme est tout autre : il s'agit simplement d'absorber l'ensemble des jetons des places du RdPFT-RD : cela correspond à l'arrêt du processus de retour en arrière. La processus d'absorption est identique, et se modélise de la même façon que le processus d'absorption de jetons associé à des contraintes temporelles et logiques modélisées dans le RdP nominal.

5.5. Méthodologie

5.5.1 Introduction

Nous proposons par la suite une méthodologie logicielle prenant en compte, dans l'ensemble du cycle de vie d'un logiciel, la notion de reconfigurabilité logicielle. Cette méthodologie propose d'unifier le cadre formel de modélisation d'une application logicielle proposé dans le Chapitre 4, et la spécification des mécanismes de reconfiguration dynamique (spécification des opérations de reconfiguration). Pour cela, nous nous appuyons sur un cycle de développement UML itératif permettant de modéliser une application par utilisation de l'environnement TURTLE, puis, lorsqu'un nouveau besoin se fait sentir sur cette application, de spécifier les changements à l'aide d'un RdPFT-RD, puis de valider formellement sur le modèle logiciel que l'exécution de ce RdPFT-RD se fait avec respect des propriétés applicatives modélisées avec l'environnement TURTLE.

Par la suite, nous présentons le principe général de cette méthodologie, puis nous détaillons chacune des étapes méthodologiques identifiées.

5.5.2 Principe général

La vision globale de notre méthodologie se découpe en cinq étapes (cf. Figure 5-9) :

1. une première version du logiciel (*logiciel 1*) est générée suite à un processus itératif de développement logiciel basé sur UML. Ce cycle utilise la méthodologie TURTLE à des fins de modélisation et de validation : un premier modèle logiciel est construit (*modèle 1*) puis validé formellement. Le cycle s'achève par l'implantation du logiciel ainsi construit à bord du satellite.
2. Une fois le logiciel opérationnel, considérons qu'un nouveau besoin est exprimé. Il convient alors de recommencer un nouveau cycle logiciel tenant compte des nouveaux besoins. Cela mène à générer un nouveau modèle logiciel (*modèle 2*) et un nouveau logiciel (*logiciel 2*).
3. A partir de la différence entre les deux modélisations (*modèle1* et *modèle2*), il est possible de générer un ensemble d'opérations de reconfiguration dynamique dont l'exécution permet de faire évoluer le premier logiciel (*logiciel1*) vers le second (*logiciel2*). L'ensemble d'opérations est dérivé en une spécification de l'exécution de ces opérations, sous forme d'un RdPFT-RD.

4. La quatrième étape consiste à valider formellement que l'exécution à bord du RdPFT-RD respectera les contraintes modélisées en TURTLE. Pour cela, l'exécution à bord du RdPFT-RD est modélisée en TURTLE, puis cette modélisation est validée formellement. L'analyse du graphe d'accessibilité conduit soit à accepter le RdPFT-RD, soit à lui ajouter des contraintes d'exécution (contraintes logiques et temporelles), soit enfin à refuser son exécution.
5. Le RdPFT-RD, éventuellement modifié lors de la phase de validation formelle, est envoyé à bord pour exécution. Le logiciel *logiciel1* est alors reconfiguré pour donner le logiciel *logiciel2*.

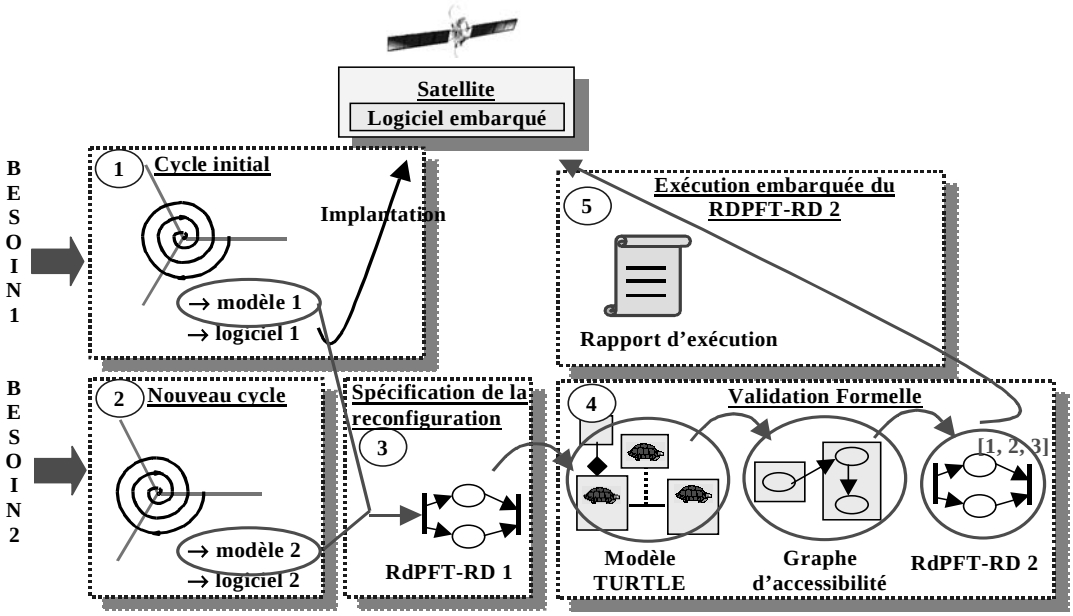


Figure 5-9. Méthodologie logicielle associée à la reconfiguration dynamique.

5.5.3 Description des étapes méthodologiques

5.5.3.1 *Cycle méthodologique de développement logiciel*

Le cycle itératif UML que nous proposons pour le développement logiciel est un cycle classique, qui peut être calqué, sauf en ce qui concerne la phase de modélisation et de validation formelle, sur les cycles itératifs UML implantés dans les outils industriels [RUP 2001][Douglass 1999]. Ce cycle est divisé en trois grandes phases (cf. Figure 5-10) : une phase d'analyse, une phase de conception, et une phase de test.

La phase d'analyse consiste principalement à identifier, à partir du cahier des charges du système, les cas d'utilisation du système (diagramme UML des cas d'utilisation, (1)). Par la suite, l'analyse du comportement du système se traduit par la réalisation de diagrammes de séquences (2).

La phase de conception consiste à réaliser une conception orientée objet du logiciel. La structure de l'architecture logicielle est tout d'abord modélisée à l'aide de la méthodologie TURTLE (modules, ports). Par la suite, les reste des classes logicielles sont identifiées, et intégrées au modèle TURTLE sous forme de classes non TURTLE (3). Le comportement des toutes les classes est ensuite décrit (4). Pour les classes TURTLE, il est décrit à l'aide des outils graphiques des diagrammes d'activité TURTLE. Le comportement des classes TURTLE représente une modélisation de leur exécution : par exemple, les appels de méthodes entre une classe TURTLE et une classe non TURTLE sont modélisés au niveau de la classe TURTLE par un opérateur temporel qui représente le temps de traitement associé à cette méthode (cf. 4.3.3.2). La troisième et dernière étape de la phase de conception consiste à modéliser la configuration logicielle (5) : cela se traduit par la modélisation de la classe de routage et du configurateur (cf. 4.4.4). L'ensemble du

diagramme de classe UML est appelé par la suite *modèle logiciel*. La sous-partie de ce modèle réalisé avec la méthode TURTLE est appelé *modèle logiciel TURTLE*. Le *modèle logiciel TURTLE* est donc une partie du *modèle logiciel*.

Outre les phases de test classique (tests unitaires, tests d'intégration), la phase de test est enrichie par un processus de validation formelle. Tout d'abord, il s'agit d'extraire des propriétés critiques du logiciel au niveau des spécifications logicielles, et d'intégrer ces propriétés au modèle TURTLE, soit par modélisation directement dans les classes concernées par ces propriétés, soit par ajout au modèle de classes de type observateurs de propriétés (6). Dans un deuxième temps, le modèle TURTLE est validé (7) : un graphe d'accessibilité est généré, et est exploité.

Une fois le cycle terminé, il peut être réitéré : il s'agit alors de raffiner progressivement les diagrammes élaborés lors du cycle précédent..

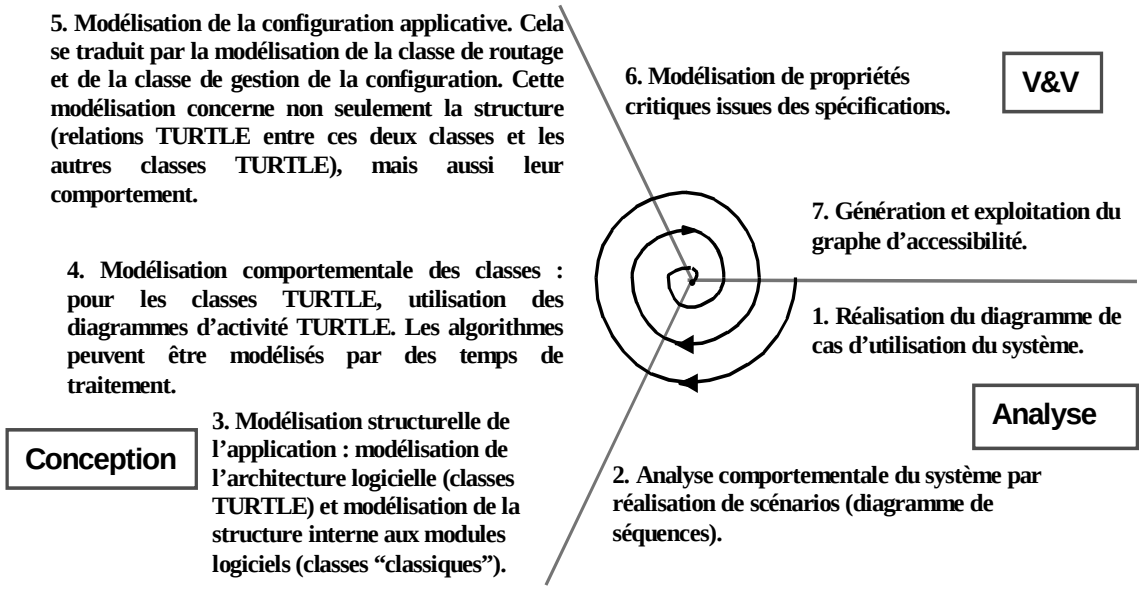


Figure 5-10. Cycle itératif de développement logiciel utilisant l'environnement TURTLE.

5.5.3.2 *Cycle méthodologique associé au développement d'une nouvelle version d'un logiciel*

Il s'agit de proposer un cycle méthodologique dans le cas où un nouveau besoin concernant un logiciel en cours de fonctionnement est exprimé. Ce cycle comporte toujours trois phases : une phase d'analyse, une phase de conception, et enfin une phase de validation.

- La phase d'analyse est identique à celle du cycle classique, sauf que les diagrammes UML de cas d'utilisation et de séquences peuvent être réutilisés si les parties du système qu'ils représentent sont inchangées.

- La phase de conception est identique à celle du cycle classique. Le modèle UML de diagramme de classes est repris, et modifié au regard des nouvelles spécifications. Par exemple, dans le cas où le nouveau besoin se traduit par l'ajout d'un module, les nouvelles classes correspondant à ce nouveau module sont ajoutées au diagramme de classes. Par la suite, la mise en place de la configuration du logiciel est décrite indépendamment de la version précédente du logiciel (on considère dans cette phase que le logiciel ne remplacera pas un autre logiciel).

- La phase de validation consiste à modéliser les propriétés issues des nouvelles spécifications sous forme de propriétés internes aux classes, ou sous forme d'observateurs de propriétés. Ces propriétés représentent les propriétés qui doivent être respectées lors du fonctionnement nominal du nouveau logiciel. Par la suite, la

génération et l'analyse du graphe d'accessibilité permettent de valider le comportement modélisé.

5.5.3.3 *Spécification de la reconfiguration*

Principe

Une fois le nouveau modèle logiciel créé (cf. phase méthodologique précédente), il convient de spécifier, à l'aide d'un RdPFT-RD, les opérations de reconfiguration dynamique qui doivent être appliquées au *logiciel 1* afin de le faire évoluer vers le *logiciel 2*. Pour cela, nous proposons un algorithme, qui, au regard du *modèle logiciel 1* et du *modèle logiciel 2* génère un RdPFT-RD représentant les ordres à appliquer pour passer du *logiciel 1* au *logiciel 2*. L'algorithme comporte trois étapes fondamentales :

1. la génération d'un ensemble O d'opérations dont l'application sur le *logiciel 1* génère le *logiciel 2*.
2. Le raffinage des opérations de reconfiguration dynamique. C'est-à-dire que l'on dérive les opérations de reconfiguration de sorte à obtenir dans la mesure du possible des opérations de reconfiguration de bas niveau.
3. La génération du RdPFT-RD qui est construit selon l'algorithme de génération d'un RdPFT-RD à partir d'un ensemble d'opérations (cf. paragraphes 5.4.3.2 et 5.4.3.3). Pour chaque opération, il convient de spécifier le temps minimal, nominal et maximal d'exécution des opérations. Le RdPFT-RD inverse est lui aussi généré et intégré au RdPFT-RD nominal selon l'algorithme du paragraphe 5.4.3.5.

Les étapes 1 et 3 de cet algorithme sont détaillées ci-dessous.

1ère étape : génération de l'ensemble O d'opérations.

La comparaison entre *modèle 1* et *modèle 2* permet de déterminer l'ensemble des opérations de reconfiguration (opérations de haut niveau) : Soit O l'ensemble de ces opérations. O est construit comme suit :

- pour tout lien l = (m1, ps, m2, pe) supprimé, on fait : O = O ∪ {supprimerLien(m1, ps)} ;
- pour tout lien l = (m1, ps, m2, pe) créé, on fait : O = O ∪ {créerLien(m1, ps)} ;
- pour tout module m supprimé, on fait : O = O ∪ {supprimerModule(m)} ;
- pour tout nouveau module m, on fait : O = O ∪ {ajouterModule(m)} ;
- pour tout port p d'un module m supprimé, on fait : O = O ∪ {supprimerPort(p, m)} ;
- pour tout port d'entrée pe ajouté à un module m, on fait O = O ∪ {ajouterPortEntrée(pe, m)} ;
- pour tout port de sortie ps ajouté à un module m, on fait O = O ∪ {ajouterPortsortie(ps, m)} ;
- enfin, pour toute modification interne (hors modification des ports) d'un module m, il convient d'examiner la modification interne de ce module. Si une modification en termes de classes ou d'objets est possible (cf. paragraphe 5.3.3.2), alors, pour toute nouvelle classe C_i introduite, on applique les deux étapes algorithmiques suivantes : 1) si les p objets de type C_k sont remplacés par des objets de type C_i, alors on fait : O = O ∪ {modifierClasse(C_k, C_i, m)}. 2) Si seulement un sous-ensemble des p objets sont modifiés en une nouvelle classe C_i, pour chaque objet p_j modifié, on fait : O = O ∪ {modifierObjet(o_j, C_i, m)}. Si la modification interne au module m_1 ne peut s'exprimer avec ce type d'opérations, il convient d'utiliser une opération de remplacement : O = O ∪ {remplacerModule(m_1, m_2)} avec m_2 représentant le nouveau module.

3ème étape : génération du RdPFT-RD

Le RdPFT-RD est généré au regard de l'ensemble O d'opérations : les dépendances entre opérations sont obtenues à partir des préconditions des opérations, puis traduites sous forme de relations de synchronisation (cf. paragraphe 5.4.3.3, Spécification de l'ordonnancement logique entre opérations). De plus, le RdPFT-RD inverse est généré, ainsi que la gestion des cas d'erreurs. Cependant, deux limites apparaissent lors de la génération d'un tel RdPFT-RD : au niveau de la gestion des ressources mémoire, et au niveau de la gestion de la réversibilité.

Génération du RdPFT-RD : gestion des ressources mémoires

De nombreuses opérations manipulent des ressources mémoires : il convient, afin d'optimiser ces ressources, d'effectuer les opérations de suppression d'entités logicielles (libération mémoire) avant les opérations d'ajout d'entités logicielles, si ces opérations n'ont pas déjà des relations de dépendance qui les contraignent à s'exécuter dans un ordre défavorable (allocation – désallocation). La gestion optimale des ressources se traduit donc par l'introduction de nouvelles relations de dépendances entre opérations. Ces relations de dépendance sont construites comme suit : soit O_a l'ensemble des opérations qui augmentent les ressources mémoires, et soit O_d l'ensemble des opérations qui diminuent les ressources mémoire. On a $(O_a \cap O_d = \varnothing) \wedge (O_a \subset O) \wedge (O_d \subset O)$. Soit D l'ensemble des dépendances des opérations de O. $\forall(o_1, o_2) \in O_d \times O_a$, (dépend(o1, o2) = faux $\wedge$ dépend(o2, o1) = faux) $\Rightarrow D = D \cup \{(o_1, o_2)\}$.

Génération du RdPFT-RD : gestion de la réversibilité

Certaines opérations de reconfiguration ne sont pas réversibles, notamment les opérations qui visent à supprimer des entités logicielles de l'application : par exemple, une opération qui supprime de l'application un objet logiciel n'est pas réversible (perte de l'état de l'objet lors de sa suppression). Or, il peut apparaître souhaitable de pouvoir retourner à l'application précédente en cas d'erreurs lors de l'exécution de la reconfiguration. Pour cela, nous proposons d'effectuer les opérations de reconfiguration dynamique non réversibles après exécution explicite d'une opération *val*() dont le code de retour *1* n'est généré que lorsque l'opérateur du système satellite a validé les opérations de reconfiguration déjà effectuées. De cette manière, tant que l'opérateur ne donne pas son accord pour franchir l'étape *val*, la reconfiguration demeure réversible.

Ainsi, l'algorithme correspondant à la modification du RdPFT-RD nominal est le suivant :

1. tout arc reliant une place p_i (dont l'opération associée est non réversible) vers t_i, est modifié en arc de p_j vers t_{fin} ;
2. tout arc dont l'extrémité est une p_i et dont l'opération associée est non réversible est modifié en arc de t_{val} vers p_i ;
3. tout arc de p_i vers t_{fin} est modifié en arc de p_i vers t_{rev}.

Cet algorithme s'applique lorsque le RdPFT-RD est déjà construit. Il est aussi possible de générer directement le RdPFT-RD qui optimise la réversibilité. Pour cela, il convient d'ajouter des relations de dépendance au sein de l'ensemble D des dépendances du RdPFT-RD : pour toute opération o_i non réversible, on ajoute la dépendance : $D = D \cup \{(val, o_i)\}$. Le RdPFT-RD peut alors être généré selon l'algorithme du paragraphe 5.4.3.3.

Reprenons l'exemple du paragraphe 5.4.3.3. Nous modifions le RdPFT-RD de la Figure 5-5, p. 91, comme suit : nous intégrons une place nommée p_{val}() (opération de validation) avant l'exécution de la transition de fin, ce qui permet de revenir en arrière tant que cette transition n'a pas été franchie, et qu'aucune opération non réversible n'a été exécutée : la Figure 5-11 met en évidence la nouvelle place p_{val} de validation de la reconfiguration. En cas de succès de la validation, la place

p_{fin} est atteinte. A contrario, si l'opérateur ne valide pas la reconfiguration, l'opération *val*() retourne le code *2* et le RdPFT-RD inverse est interprété.

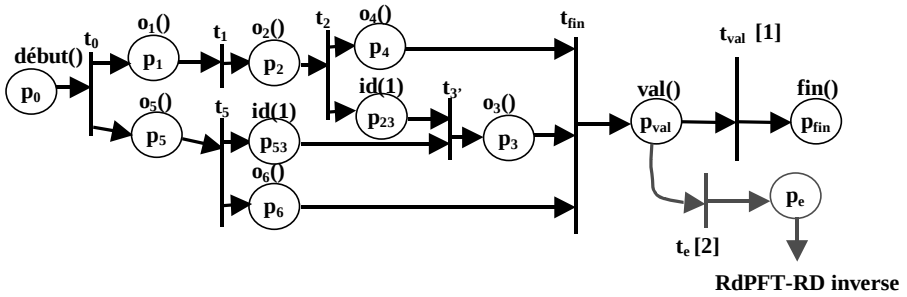


Figure 5-11. Modélisation d'une place de validation de l'exécution d'une reconfiguration.

Examinons à présent la modification devant être apportée au RdP afin de prendre en compte la non réversibilité de certaines opérations. Supposons par exemple que les opérations o3 et o6 de la Figure 5-11 ne soient pas réversibles. Le nouveau RdPFT-RD que l'on peut construire par application de l'algorithme précédent est représenté à la Figure 5-12 : les opérations o3 et o6 ne sont réalisées qu'après l'opération *val*. Cette modélisation est possible car aucune opération non réversible ne dépend de o3 et o6. Ainsi, il n'est pas possible dans le cas général de modéliser la réversibilité. Par la suite, nous définissons les RdPFT-RD selon leur degré de réversibilité.

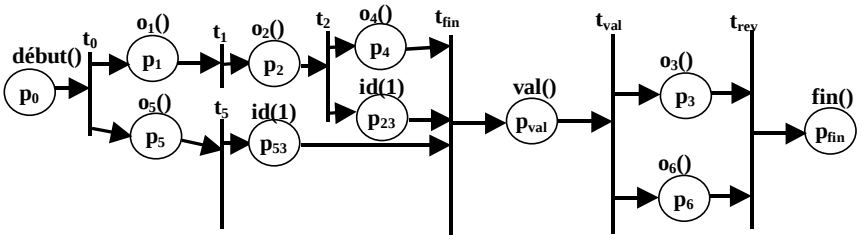


Figure 5-12. Modélisation de deux opérations o_3 et o_6 non réversibles.

Soit un RdPFT-RD. Soit $P_a \subset P \ / \ \forall \ p_i \in P_a$, l'opération o_i associée à p_i est non réversible et aucune opération réversible ne dépend de o_i.

Soit $P_b \subset P \ / \ \forall \ pi \in P_b$, l'opération o_i associée à p_i est non réversible et il existe au moins une opération réversible dépendant de o_i.

Par construction, on a $P_a \cap P_b = \varnothing$.

Définition 8. RdPFT-RD totalement réversible

On qualifie de *totalement réversible* un RdPFT-RD pour lequel $P_a \cup P_b = \varnothing$.

En effet, $P_a \cup P_b = \varnothing$ implique que toutes les opérations soient réversibles.

Définition 9. RdPFT-RD partiellement réversible

On qualifie de *partiellement réversible* un RdPFT-RD pour lequel les deux assertions suivantes sont vraies :

- $P_a \neq \varnothing$
- $P_b = \varnothing$

En effet, dans ce cas, certaines opérations sont non réversibles, mais aucune opération réversible ne dépend d'une opération non réversible, ce qui permet de réaliser dans un premier temps toutes les opérations réversibles, de valider leur exécution, puis de réaliser les opérations non réversibles.

Définition 10. RdPFT-RD non réversible

- On appelle *RdPFT-RD non réversible* un RdPFT-RD pour lequel $P_b \neq \emptyset$.

En effet, dans cas, il n'est pas possible d'exécuter toutes les opérations réversibles sans exécuter au moins une opération non réversible : l'exécution du scénario n'est donc pas réversible.

Génération du RdPFT-RD : compromis ressources / réversibilité

Les contraintes d'optimisation des ressources imposent d'effectuer au plus tôt les opérations de libération de ressources mémoire, qui sont des opérations non réversibles. A contrario, l'optimisation de la gestion de la réversibilité conduit à les repousser après exécution de toutes les opérations réversibles. Il est du choix du concepteur de choisir le meilleur compromis entre l'une ou l'autre des optimisations. La phase de validation formelle qui suit la phase de génération du RdPFT-RD peut aussi inciter à choisir un RdP qui optimise les ressources, ou au contraire, un RdPFT-RD qui améliore la réversibilité.

5.5.3.4 *Processus de validation formelle*

La phase méthodologique de validation formelle a pour but de valider que la spécification de reconfiguration dynamique peut-être appliquée au modèle TURTLE avec respect des propriétés spécifiées. Cette phase se découpe en deux sous-phases : la modélisation en TURTLE de la spécification de reconfiguration dynamique (SRD) et l'analyse du graphe d'accessibilité du modèle TURTLE.

Modélisation TURTLE de la spécification de reconfiguration dynamique

L'étape précédente du processus méthodologique a permis de générer la spécification de la reconfiguration dynamique. Il convient à présent, à des fins de validation, de modéliser en TURTLE l'exécution de cette spécification à bord. Le *modèle TURTLE de reconfiguration dynamique* est l'union des modélisations suivantes (cf. Figure 5-13) :

1. modélisation TURTLE du logiciel 1 (architecture et observateurs) ;
2. modélisation TURTLE des différences entre le logiciel 2 et le logiciel 1 (architecture et observateurs) ;
3. modélisation d'un module de routage compatible avec le logiciel 1 et 2, et piloté par le gestionnaire de configuration ;
4. modélisation des propriétés devant être vérifiées *pendant la reconfiguration* (observateurs dits *observateurs de reconfiguration*).
5. modélisation TURTLE du gestionnaire de configuration qui démarre le logiciel 1, attend un temps à déterminer, puis exécute la spécification de reconfiguration tout en pilotant les observateurs : désactivation des observateurs du logiciel 1 et activation des observateurs de reconfiguration au début de l'exécution de la reconfiguration, désactivation des observateurs de reconfiguration et activation des observateurs du logiciel 2 à la fin de la reconfiguration.

Le gestionnaire de reconfiguration est issu du RdPFT-RD spécifiant les opérations de reconfiguration. Le modèle TURTLE issu de cette spécification représente une modélisation de l'exécution de la spécification. En effet, la spécification représentée par le RdPFT-RD est exhaustive au regard du logiciel à reconfigurer, alors que le modèle TURTLE ne représente qu'une modélisation partielle des fonctionnalités du logiciel : il n'est donc pas possible dans le cas général d'extraire automatiquement les opérations du RdPFT-RD et de générer une modélisation en TURTLE de l'exécution de ces opérations, cette génération exigeant des capacités d'abstraction des mécanismes exprimés par la spécification, cette abstraction devant en outre être réalisée au regard de la modélisation logicielle TURTLE. En effet, il convient d'extraire de la spécification les opérations pertinentes au regard du sous-système critique modélisé en TURTLE. A noter que la

modélisation TURTLE des opérations de reconfiguration dynamique a été présentée au paragraphe 4.4.4.3.

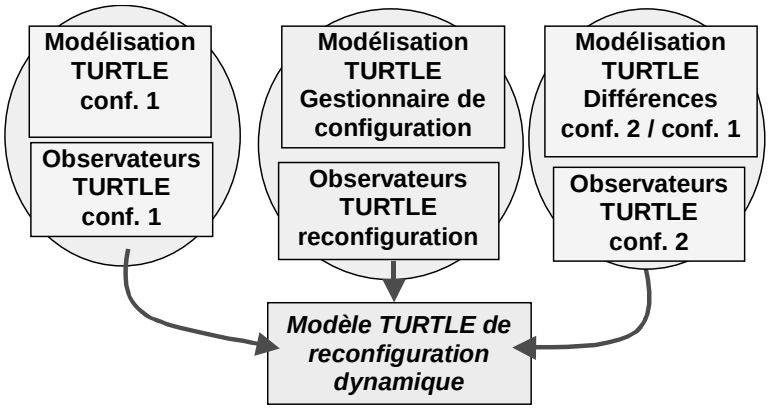


Figure 5-13. Processus d'obtention du modèle TURTLE de reconfiguration dynamique.

Nous présentons par la suite une modélisation TURTLE de quelques spécifications de reconfiguration dynamique : la modélisation de relation série / parallèle, la modélisation des contraintes de temps d'exécution, la modélisation de contraintes de temps entre opérations, et enfin, la modélisation de contraintes logiques entre opérations.

Un RdPFT-RD exprime avant tout un ordre partiel par l'intermédiaire d'opérateurs série / parallèle entre des opérations de reconfiguration. Supposons que nous puissions traduire chaque opération de reconfiguration par une offre de synchronisation sur une porte. Si les relations séries entre des opérations s'expriment de façon naturelle en TURTLE par un enchaînement de relations de synchronisation, les relations de parallélisme doivent être explicitement modélisées par l'élément graphique permettant de modéliser des sous-activités TURTLE parallèles.

La Figure 5-14 présente des relations séries et parallèles entre 5 opérations o_1,…,o_5. Ces 5 opérations sont traduites en TURTLE par cinq synchronisations sur cinq portes différentes g_1, …, g_5 associées respectivement à o_1, …, o_5. A noter que la modélisation TURTLE d'un RdPFT-RD inverse s'effectue de la même façon. A noter aussi que la modélisation TURTLE (cf. Figure 5-14-(b)) doit être insérée au niveau du gestionnaire de configuration.

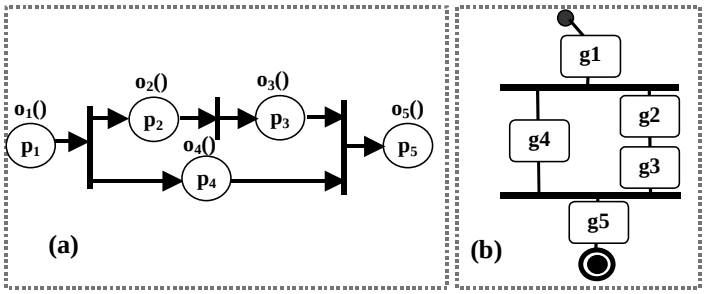


Figure 5-14. Expression des contraintes de synchronisation. (a) Spécification RdPFT-RD. (b) Transcription en TURTLE.

La modélisation TURTLE des temps d'exécution des opérations s'effectue comme suit : il s'agit de modéliser un temps minimal d'exécution et un temps maximal d'exécution : l'opération doit s'exécuter dans l'intervalle temporel $[t_{min}, t_{max}]$. En cas de violation de la contrainte maximale d'exécution, il convient de modéliser en TURTLE un cas d'erreur. De plus, en cas de retour anticipé de l'opération ($t_{exec} < t_{min}$), il faut générer une attente artificielle afin de continuer l'interprétation du diagramme d'activité uniquement lorsque $t_{exec} \geq t_{min}$. La Figure 5-15 met en évidence la modélisation TURTLE associée à la spécification sous forme d'un RdPFT-RD de l'exécution d'une opération o.

La porte TURTLE associée à cette opération est *g*. Afin de modéliser la durée minimale d'exécution, nous introduisons une synchronisation entre deux sous-activités : une qui représente l'exécution de *g*, et une autre qui après t_{min} unités de temps, offre une synchronisation sur la porte *min*. Au niveau de l'activité de *g*, la porte min n'est offerte qu'une fois la synchronisation sur g effectuée : si la synchronisation sur *g* se déroule avant t_{min} unités de temps, la synchronisation sur *min* reste bloquée jusqu'à t_{min} unités de temps. Dans le cas contraire, la synchronisation sur min est offerte immédiatement. Afin de modéliser la durée t_{max}, nous introduisons, au niveau de la synchronisation sur *g*, une offre limitée sur t_{max}. En cas d'échec de la synchronisation sur g pendant t_{max} unité de temps, c'est le cas de gestion d'erreur qui est ensuite interprété.

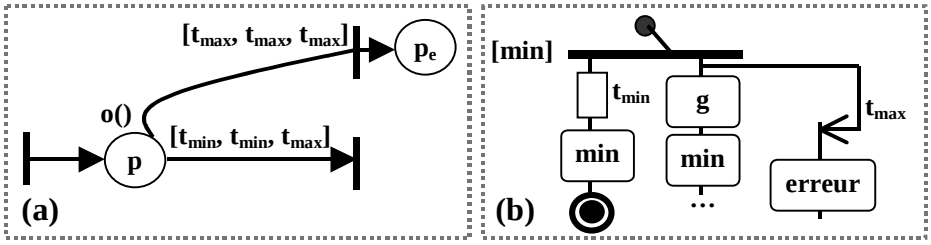


Figure 5-15. Expression des contraintes d'exécution. (a) Spécification avec un RdPFT-RD. (b) Transcription en TURTLE.

Par la suite, nous montrons comment il est possible de modéliser en TURTLE des contraintes temporelles entre deux opérations. La Figure 5-16 met en évidence une telle modélisation. Supposons qu'il existe une contrainte de temps imposant qu'il ne s'écoule pas plus de *m* unités de temps entre le début de l'exécution de o_1 et la fin de l'exécution de o_2. Nous introduisons deux portes TURTLE *g1* et *g2* qui représentent respectivement o_1 et o_2. Lorsque la synchronisation sur *g1* est proposée, en parallèle est proposée une synchronisation en temps limité (temps limité = *m*) sur la porte *max*. Cette synchronisation sur *max* est par ailleurs offerte après l'exécution sur la porte *g2*. Ainsi, soit la synchronisation sur *max* située après *g2* est offerte avant *m* unités de temps, et dans ce cas la contrainte temporelle est respectée, soit l'offre limitée expire avant, et dans ce cas, c'est le cas d'erreur qui est traité. Si la contrainte entre les deux opérations o_1 et o_2 est une contrainte logique, dans ce cas, l'offre sur *max* n'est offerte que si la contrainte logique est respectée : dès qu'elle ne l'est plus et que la synchronisation sur *max* n'a pas été offerte, alors le cas d'erreur est interprété.

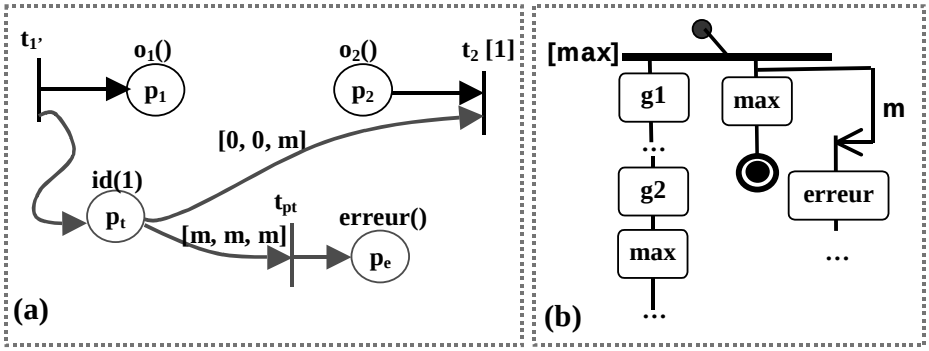


Figure 5-16. Contraintes temporelles d'exécution entre deux opérations. (a) Spécification RdPFT-RD. (b) Modélisation TURTLE.

Analyse du graphe d'accessibilité

Le graphe d'accessibilité considéré est celui obtenu par le processus de validation formelle TURTLE appliqué au modèle TURTLE de reconfiguration dynamique (cf. paragraphe précédent). Son examen permet de mettre en évidence deux aspects : la présence éventuelle d'état puits ou de violation de propriétés modélisées. Si c'est l'exécution de la reconfiguration qui génère des erreurs, alors il convient de modifier le RdPFT-RD modélisé en TURTLE comme suit :

1. si des erreurs sont générées lorsque la reconfiguration est lancée dans un certain état applicatif, alors il convient d'ajouter en tête du RdPFT-RD des opérations qui examinent l'état applicatif, et qui font que l'interprétation des opérations de reconfiguration n'a lieu que si l'état applicatif courant est compatible avec l'état applicatif analysé sur le graphe d'accessibilité.
2. Si des erreurs sont générées à certains stades de l'interprétation du RdPFT-RD et dans un certain état applicatif, alors il convient d'ajouter dans le RdPFT-RD des contraintes entre toutes les opérations pouvant être exécutées à ce stade applicatif. S'il s'agit d'un état logique applicatif, on insère des contraintes logiques entre les opérations concernées c'est-à-dire une contrainte qui va s'assurer que l'état applicatif n'est pas celui qui a été identifié comme conduisant au cas d'erreur identifié. S'il s'agit d'un état temporel, alors des contraintes temporelles entre opérations sont ajoutées au RdPFT-RD.
3. Si la reconfiguration ne peut-être effectuée sans erreurs dans aucun cas, ou ne peut être réalisée qu'à partir d'un état applicatif peu probable, alors il convient ou d'assouplir les propriétés devant être respectées pendant la reconfiguration (la continuité de service peut souffrir d'une telle modification), ou de reconfigurer l'application dans une moindre mesure.

5.5.3.5 *Exécution embarquée de la spécification des ordres de reconfiguration*

L'exécution à bord du RdPFT-RD est conditionnée par la démonstration qu'aucun cas d'erreur ne se produit lors de la validation formelle de modélisation TURTLE du RdPFT-RD. L'exécution embarquée du modèle peut conduire à une demande explicite auprès de l'opérateur de passer à l'étape suivante (demande de continuation d'exécution du scénario, opération *val*()). De plus, l'exécution de la dernière opération du RdPFT-RD entraîne l'envoi à l'opérateur d'un rapport d'exécution.

5.6. Pragmatique de la reconfiguration

5.6.1 Introduction

Cette section est consacrée à la mise en œuvre de la reconfiguration dynamique au sein d'un système satellite. Cette mise en œuvre regroupe d'une part les aspects sols de la mise en œuvre et d'autre part, les aspects bords. Il s'agit de proposer une répartition sol / bord permettant de mettre en œuvre dans un système satellite l'architecture logicielle bord présentée au chapitre 2 et l'exécution d'un modèle de reconfiguration dynamique.

Tout d'abord, nous présentons l'architecture générale du système support : cette présentation se concentre notamment sur la répartition bord / sol des différentes fonctionnalités inhérentes aux mécanismes de reconfiguration. Par la suite, nous détaillons les fonctionnalités embarquées, puis nous explicitons leur mise en œuvre. Enfin, nous terminons par un passage en revue des fonctionnalités au niveau du sol.

5.6.2 Architecture générale du système support

5.6.2.1 *Besoins*

Le principal besoin de mise en œuvre concerne l'exécution d'une spécification de reconfiguration dynamique sur un logiciel embarqué dans un engin spatial. Cette exécution présente plusieurs aspects : l'envoi à bord de la spécification, la lecture embarquée de la spécification et l'exécution des opérations spécifiées dans le modèle, et enfin l'émission du rapport d'exécution de la spécification.

5.6.2.2 *Proposition*

L'architecture de mise en œuvre proposée est décrite à la Figure 5-17. Cette architecture comporte trois composants principaux : le module logiciel embarqué du satellite et deux composants sols : le module opérateur et le serveur de classes. Par la suite, nous détaillons ces trois composants.

Architecture globale du système embarqué

Le système embarqué du satellite est responsable de trois tâches principales :

- l'*exécution de l'application* embarquée elle-même. L'exécution de l'application se traduit par l'exécution des modules applicatifs et par la réalisation du routage de messages entre les modules.
- Un *middleware de reconfiguration dynamique embarqué*. Ce middleware doit pouvoir gérer la spécification de reconfiguration dynamique envoyée par l'opérateur.
- Enfin, la *communication avec les autres composants* systèmes (opérateur et serveur de classes) est supportée via une interface de communication. Nous supposerons que cette interface est de type TM/TC au-dessus de laquelle un protocole de transport est fourni, par exemple une implémentation du protocole SCPS-TP [SCPS 1997], particulièrement bien adapté à ce type de liaison satellite.

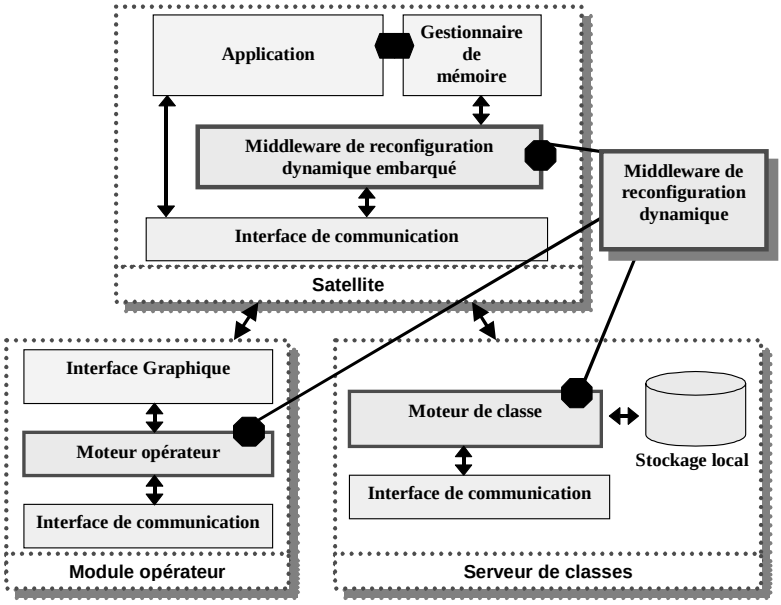


Figure 5-17. Architecture globale du système support à la reconfiguration dynamique.

Architecture globale du module opérateur

Le module opérateur comprend trois blocs fonctionnels principaux :

- l'*interface graphique*, qui permet de saisir directement des ordres de reconfiguration dynamique, d'envoyer une spécification, de recevoir un rapport ou une demande de validation de l'exécution de la spécification (dernière étape avant la non réversibilité), ou d'envoyer des ordres pour examiner l'état de l'application ;
- le *moteur opérateur*, entité qui formate les ordres et les spécifications, et extrait les rapports de reconfiguration dynamique ;
- l'*interface de communication* (cf. l'interface de communication du logiciel embarqué).

Architecture globale du serveur de classes

Le serveur de classes est constitué :

- d'un *stockage local* permettant au serveur de classes de stocker des classes applicatives. Ce stockage est typiquement constitué d'un disque dur et de son architecture matérielle et logicielle associée.
- D'un *moteur de classes*, qui est l'implémentation d'un protocole permettant au bord de demander l'envoi du code d'une classe, ou permettant à l'opérateur d'ajouter ou de retirer une classe au stockage local.
- D'une interface de communication (cf. l'interface de communication du logiciel embarqué).

A noter que l'ensemble {Middleware de reconfiguration dynamique embarqué, moteur opérateur, moteur de classes} est appelé par la *suite Middleware de Reconfiguration Dynamique* (MRD). La partie embarquée est appelée suite *Middleware de Reconfiguration Dynamique embarqué* (MRD.e), la partie opérateur est appelée suite *Middleware de Reconfiguration Dynamique opérateur* (MRD.o) et la partie du serveur de classes est appelée par la suite *Middleware de Reconfiguration Dynamique du serveur de classes* (MRD.sc)

5.6.3 Environnement support bord

5.6.3.1 *Principe général*

Nous proposons d'implémenter l'architecture logicielle et le middleware de reconfiguration dynamique dans le langage orienté objet Java. Dans un premier temps, nous mettons en évidence pour un logiciel (application et middleware) programmé dans ce langage une implantation possible dans un système embarqué dans un satellite. Par la suite, nous explicitons un certain nombre de fonctionnalités du support de l'architecture et du middleware et montrons la mise en œuvre de ces fonctionnalités. Nous montrons notamment l'intérêt du langage Java pour le support des opérations de reconfiguration nécessitant une intégration dynamique de code.

5.6.3.2 *Architecture d'implantation*

Nous présentons dans ce paragraphe une implantation possible pour un système support à la reconfiguration d'une application développée en langage Java. La Figure 5-18 met en évidence une telle implantation. Nous supposons que le logiciel dit plate-forme (DHU) et le logiciel charge utile (PMU) soient implantés sur la même carte. La carte comprend quatre entités différentes :

- une PROM de démarrage, qui comprend un logiciel de démarrage (dit logiciel de boot) qui assure notamment l'autotest des périphériques, et un logiciel écrit en C ou en Ada assurant des fonctions minimales telles que la gestion des périphériques et les interfaces de communication (bus, TM/TC, etc.).
- Deux EEPROMS. L'EEPROM 1 comprend un exécutable qui assure le lancement du système d'exploitation, du middleware de reconfiguration dynamique, et enfin du logiciel applicatif. La présence de L'EEPROM 2 se justifie de trois manières. Tout d'abord, nous conservons la possibilité d'effectuer des modifications concernant le système d'exploitation et le middleware, ce qui n'est pas faisable avec l'architecture de reconfiguration choisie qui limite les modifications à la partie applicative du logiciel bord. Ensuite, il peut être utile, en cas d'erreur critique, de rédémarrer sur une nouvelle version du logiciel bord, plutôt que de relancer l'ancienne version et de la reconfigurer à nouveau pour qu'elle soit conforme à la version en fonctionnement avant l'erreur critique : il pourrait ainsi être souhaitable de reprogrammer l'EEPROM 2 avec la nouvelle version logicielle dès qu'une opération de reconfiguration a eu lieu. Enfin, l'EEPROM 2 accroît la tolérance aux fautes de l'ensemble du système (cas de défaillance de l'EEPROM 1).
- la RAM qui sert de mémoire d'exécution au système d'exploitation, au middleware de reconfiguration, et enfin à la partie applicative du logiciel.

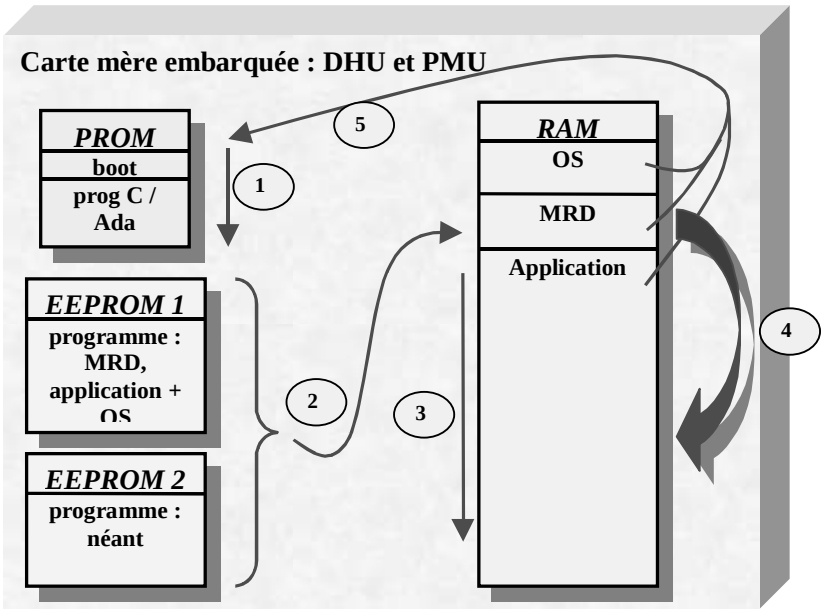


Figure 5-18. Implantation embarquée d'un logiciel applicatif intrinsèquement reconfigurable.

Le processus de fonctionnement de la carte est le suivant (cf. Figure 5-18). (1) Le logiciel de boot est lancé, puis un programme minimal démarre. (2) Le code du logiciel principal est recopié en RAM, puis est exécuté : le système d'exploitation est initialisé, puis ce dernier exécute une première tâche logicielle qui lance les autres tâches applicatives. (3) L'application est en phase nominale d'exécution. (4) Un ordre d'exécution d'une spécification de reconfiguration est envoyé au middleware de reconfiguration dynamique embarqué. L'interprétation de la spécification conduit à modifier le code applicatif. (5) En cas d'erreur critique (système d'exploitation, middleware, logiciel applicatif), la seule solution est de revenir au programme minimal de gestion. La mémoire RAM est vidée, puis le processus recommence (phase 2).

Par la suite, nous décrivons la partie logicielle de l'environnement support bord : tout d'abord, les fonctionnalités offertes par le middleware de reconfiguration dynamique embarqué sont mises en évidence. Ensuite, nous montrons la mise en œuvre de la partie applicative.

5.6.3.3 *Middleware de reconfiguration embarqué*

Le middleware de reconfiguration dynamique est implanté entre le système d'exploitation, et l'applicatif. Il comprend cinq modules fondamentaux (cf. Figure 5-19) :

- un moteur de reconfiguration dynamique ;
- une librairie d'opérations ;
- un gestionnaire de mémoire ;
- un protocole opérateur ;
- un protocole de classes.

Les trois premiers modules sont traités dans cette section, alors que les deux protocoles seront abordés dans les sections suivantes. Pour mémoire, seule l'application peut être reconfigurée par l'approche proposée : le MRD ne peut en aucun cas être modifié.

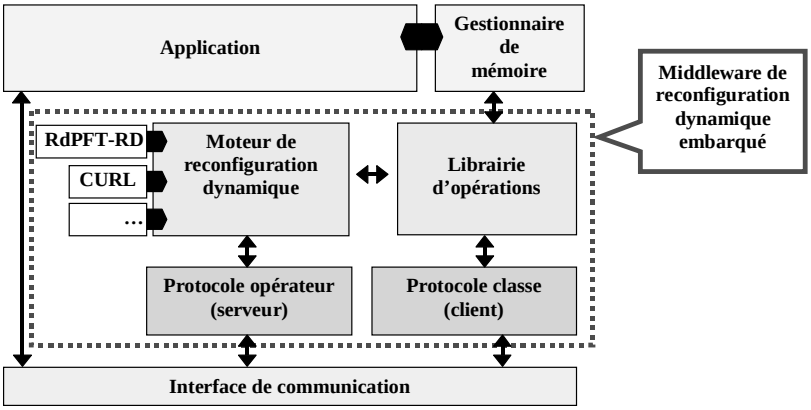


Figure 5-19. Middleware de reconfiguration dynamique embarqué.

Moteur de reconfiguration dynamique

Le but du moteur de reconfiguration dynamique est d'interpréter une spécification d'opérations à exécuter sur le logiciel embarqué. La spécification lui est transmise par le protocole opérateur, puis elle est interprétée. Pour chaque opération contenue dans la spécification, le moteur de reconfiguration dynamique s'appuie sur la librairie d'opérations.

Un mécanisme de *plug-in* permet d'utiliser les fonctionnalités du moteur de reconfiguration dynamique tout en offrant la possibilité d'utiliser son propre langage de reconfiguration dynamique. Nous donnons dans le schéma de la Figure 5-19 l'exemple de l'utilisation de notre propre langage de reconfiguration (RdPFT-RD), mais aussi l'exemple de l'utilisation du langage CURL (*Corot Update Reconfiguration Language*), un langage développé dans le cadre de l'étude de la reconfiguration dynamique du logiciel embarqué du satellite scientifique COROT [Cailliau 2001]. Afin d'être utilisé avec notre middleware, tout environnement d'un langage de reconfiguration doit comprendre les éléments suivants : (1) un mécanisme pour encoder / décoder la spécification des opérations à des fins de transmission de la spécification (par exemple, transmission d'une spécification entre le sol et le bord), (2) des mécanismes permettant d'interpréter la spécification, et enfin éventuellement (3), des mécanismes permettant d'encoder / décoder le rapport issu de l'interprétation de la spécification. L'environnement associé à notre langage de reconfiguration dynamique comporte des fonctions d'encodage et de décodage du modèle, ainsi qu'un interpréteur de RdPFT-RD.

L'algorithme réalisé en boucle par le moteur de reconfiguration est le suivant :

1. chargement du modèle ;
2. décodage du modèle ;
3. interprétation du modèle : utilisation en support de la librairie d'opérations ;
4. encodage éventuel d'un rapport d'interprétation.

Librairie d'opérations

La librairie d'opérations offre une interface permettant d'exécuter sur l'application les opérations de reconfiguration de bas niveau. Pour toutes les manipulations mémoires, elle s'appuie sur le gestionnaire de mémoire. Un code de retour est renvoyé par la librairie suite à une demande d'exécution d'une opération.

Gestionnaire de mémoire

Le gestionnaire de mémoire doit être capable de gérer toutes les opérations qui consistent à écrire ou lire un élément d'architecture ou un élément relatif aux applications orientées objets, à savoir les objets et les classes. Ces opérations représentent des opérations de bas niveau du point de vue logiciel au regard des opérations fournies par la librairie d'opérations. Le gestionnaire de mémoire est intimement lié à l'application dans la mesure où les opérations mémoires conduites par l'application doivent être connues du gestionnaire d'application (création d'objets, etc.).

5.6.3.4 *Mise en œuvre de l'architecture logicielle*

Nous abordons dans cette partie l'implémentation d'une application logicielle embarquée intrinsèquement reconfigurable.

Critères de sélection d'un langage support

L'augmentation de la taille des logiciels du spatial associée à une réduction du temps de développement et du coût de production conduit les acteurs industriels du spatial à se tourner vers des solutions logicielles innovantes. L'utilisation de noyaux temps-réel commerciaux est une de ces solutions. Toutefois, afin d'aller encore plus loin, il pourrait être intéressant de se tourner vers des langages de plus haut niveau offrant une meilleure indépendance vis-à-vis du matériel, et permettant par-là même d'améliorer la réutilisation des composants logiciels développés dans le cadre d'un projet. De plus, les contraintes de reconfiguration dynamique nous conduisent à penser qu'un tel langage de haut niveau devra offrir des mécanismes supports permettant l'intégration non prévue et dynamique d'un nouveau code logiciel pendant le fonctionnement de l'application.

Ces trois considérations en terme d'indépendance vis-à-vis du matériel, de modularité, et d'intégration dynamique de code nous conduisent à nous pencher vers le langage Java qui répond à ces trois critères. En effet, une machine virtuelle Java (JVM pour *Java Virtual Machine*) chargée d'exécuter le code issu de la compilation de programme Java assure une indépendance vis-à-vis du matériel sous-jacent. Ce code est de plus portable, c'est-à-dire qu'il peut migrer d'une plate-forme à une autre sans recompilation. Ensuite, Java est un langage orienté objets qui offre donc une modularité intrinsèque. Enfin, les environnements d'exécution de Java sont pourvus d'un mécanisme d'intégration dynamique de code. Mais si Java semble répondre favorablement aux trois critères précités, il s'agit cependant de valider son utilisation pour le développement d'applications temps-réel embarquées.

Java à l'épreuve des applications temps-réel embarquées

Le langage Java possède un certain nombre d'atouts pour le développement d'applications temps-réel embarquées [Nilsen 2001]. Premièrement, Java est un langage sans pointeurs, ce qui accroît grandement la fiabilité du système. De plus, un ramasse-miettes s'occupe périodiquement de nettoyer la mémoire des objets non référencés et de défragmenter la mémoire. Deuxièmement, Java supporte l'intégration de code natif (en C par exemple) afin par exemple d'autoriser le pilotage de périphériques [Bunnel 1998]. Troisièmement, le langage intègre la notion de tâches. Ces tâches sont soit gérées par un ordonnanceur interne à la machine virtuelle Java (*green threads*), soit par l'ordonnanceur du système d'exploitation sous-jacent (*native threads*).

Toutefois, le langage Java possède de sérieuses limites pour les applications temps-réel embarquées. Tout d'abord, les performances d'exécution sont inférieures à celles du langage C [Prechelt 2000]. Les performances des premières machines virtuelles Java étaient encore plus médiocres (interprétation du code), mais l'introduction des compilateurs à la volée (compilateurs dits *jit*) a permis de résoudre en partie le problème. Toutefois, l'introduction de tels compilateurs nuit au déterminisme d'exécution (la première exécution d'un bout de code nécessite une compilation) et augmente fortement la trace mémoire de l'environnement d'exécution en raison notamment du maintien d'une table des symboles en mémoire. De plus, les techniques d'optimisation sont particulièrement complexes à mettre en œuvre lors des compilations à la volée. Le module *Hotspot* apporte quelques solutions à ce problème [HotSpot 1999]. Deuxièmement, le support intrinsèque des tâches dans Java amène à se poser la question de la performance de l'ordonnancement de tâches Java, en termes notamment de temps de commutation de tâche, de déterminisme (respect des priorités par exemple), et de portabilité. Troisièmement, un gestionnaire de mémoire de type ramasse-miettes est-il à même d'assurer un parfait déterminisme quant au temps d'accès mémoire ? Enfin, l'intégration de code natif à des fins de pilotage de périphérique nuit à la portabilité applicative, à la réutilisabilité, et à l'aspect tout orienté objet des applications.

Evaluation du langage Java pour les applications spatiales embarquées

Afin de répondre aux limitations précitées en ce qui concerne la capacité du langage Java à être utilisé pour les applications spatiales embarquées, nous avons évalué ce dernier [Apvrille 1998][Apvrille 1999]. Notre évaluation a porté tout d'abord sur l'ordonnancement. Par la suite, nous avons développé un prototype d'application spatiale afin d'évaluer les autres limitations, et plus particulièrement la gestion mémoire et le pilotage de périphérique par utilisation de code natif.

Notre environnement d'évaluation est le suivant (cf Figure 5-20) :

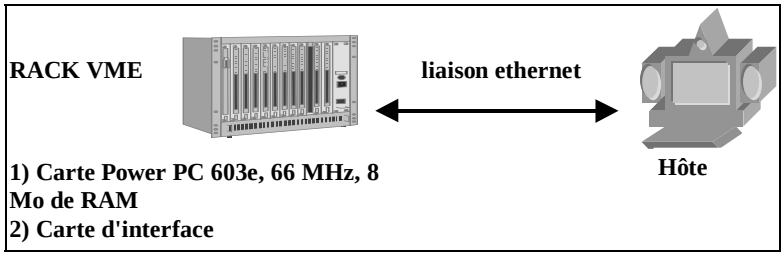


Figure 5-20. Environnement d'évaluation du langage Java pour les applications logicielles spatiales.

- un Rack VME constitué notamment d'une carte mère potentiellement spatialisable (Power PC). Un système d'exploitation temps-réel commercial est implanté dans l'EEPROM de démarrage de la carte. Une machine virtuelle Java est incluse dans ce système d'exploitation.
- Un hôte permettant l'implantation du logiciel via liaison Ethernet. Le code Java implanté est instrumenté afin de récupérer les données de simulation.

Résultats d'évaluation du langage Java

Nous résumons dans ce paragraphe les principaux résultats d'évaluation. L'ensemble des résultats peut-être obtenu dans [Apvrille 1998]. En ce qui concerne l'ordonnancement, le fait de lier les tâches Java à l'ordonnanceur d'un système d'exploitation temps-réel permet d'obtenir des performances compatibles avec celles des applications spatiales. En ce qui concerne la gestion de la mémoire, le mécanisme de ramasse-miettes s'est avéré peu performant et très intrusif (il est non préemptible pendant plusieurs dizaines de millisecondes). Nous avons toutefois expérimenté deux solutions. La première consiste à intégrer le fonctionnement du ramasse-miettes dans le profil d'exécution des tâches applicatives. Pour cela, nous avons borné son temps de fonctionnement, et nous le déclenchons au moment opportun. La deuxième solution consiste à interdire les allocations dynamiques en dehors de la période d'initialisation de l'application. Pour cela, nous avons implanté un système de pool d'objets. Cette solution s'est avérée très efficace et peu contraignante, dans la mesure où les allocations dynamiques en cours de fonctionnement de l'application sont proscrites au niveau des standards de codage des environnements spatiaux. Enfin, les autres points (code natif, etc.) ont montré de bons résultats.

Ces résultats encourageants nous ont amené à privilégier l'utilisation de ce langage. De nombreux travaux visant à améliorer les environnements d'exécution Java pour les applications embarquées [Bossu 1999][Clohessy 2001] associés au processus de spécification d'une version temps-réel du langage Java [RTJWG 2000][RTJ 2001] nous ont confortés dans notre choix. Nous avons donc décidé de réaliser nos maquettes de logiciels spatiaux reconfigurables et du middleware de reconfiguration en Java.

Architecture logicielle

L'architecture logicielle présentée au paragraphe 4.2 a été implantée en Java. Cela concerne l'implantation des modules, des ports, et des liens de communication.

Les modules ne possédant qu'un seul flux d'exécution, nous les assimilons à une tâche Java. L'architecture UML de notre implantation est présentée à la Figure 5-21. Un *Module* hérite de la

classe *java.lang.Thread* (un module est une tâche du système) et est constitué par défaut de 25 ports d'entrées (*InGate*) et de 25 ports de sortie (*OutGate*), qui héritent tous les deux de la notion de porte (*Gate*). Une port d'entrée possède un buffer de réception des messages (classe *ModuleBuffer*). Les messages pouvant être reçus dans un tel buffer sont de type *Message*, ou d'un type hérité. Par défaut, un *ModuleBuffer* peut stocker 100 messages. Afin de respecter les contraintes d'allocations dynamiques précédemment évoquées, tous les messages d'un *ModuleBuffer* sont instanciés lors de la création de la porte d'entrée correspondante, et sont placés dans une réserve de messages. De plus, afin de garantir la cohérence des données transmises dans ces messages, la lecture ou l'écriture de messages se fait au niveau du buffer au sein d'une section critique (mot clé Java *synchronized*). Enfin, nous associons à tout module une classe de type *BufferGroup* dont le rôle est d'offrir une méthode d'attente de réception d'un message sur un buffer d'entrée quelconque. En effet, en l'absence d'une telle classe, un *Module* possédant plusieurs ports d'entrée ne pourrait se mettre en attente de l'arrivée d'un message que sur l'un des ports.

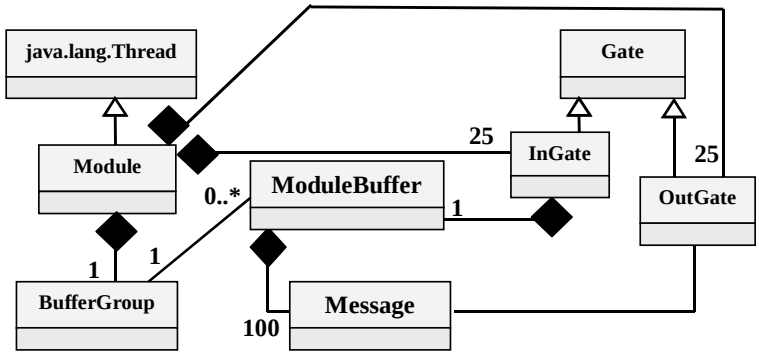


Figure 5-21. Diagramme des classes d'architecture de l'implantation.

Support des opérations de reconfiguration dynamique

Dans cette sous-section, nous examinons la mise en œuvre de certaines opérations supports à la reconfiguration dynamique. Il s'agit en effet de mettre en œuvre toutes les opérations logicielles de reconfiguration dynamique de bas niveaux (cf. 5.3.3.3). Ce paragraphe n'aborde que la mise en œuvre des opérations nécessitant des mécanismes logiciels avancés. En effet, deux types d'opérations semblent devoir bénéficier de mécanismes spécifiques : les opérations de manipulation des objets (création, destruction), et les opérations de manipulation des classes (chargement, destruction).

En ce qui concerne les objets, il convient de pouvoir spécifier, à l'exécution, l'instanciation d'un objet d'un nom de classe non prévu lors de la création de l'application. Par exemple, il s'agit de créer un objet instance de la classe « classe1 ». Pour cela, nous proposons d'utiliser les mécanismes de *Reflection* Java autorisant la création de tels objets à partir d'un nom de classe. Le rôle de cette API spécifique est d'offrir des mécanismes introspectifs dans les classes et les objets d'une application Java.

En ce qui concerne les classes, il s'agit de pouvoir ajouter à la mémoire applicative une classe, ou réciproquement, il s'agit de pouvoir supprimer de la mémoire applicative une classe. De nombreux travaux de recherche s'intéressent à l'intégration dynamique de classes applicatives dans les langages objets. Des mécanismes ont été proposés dans les langages tels que *Smalltalk* [Goldberg 1983] ou *C++* [Hjalmtysson 1998]. De son côté, Java intègre un mécanisme de téléchargement de classes, le *classloader* Java [Liang 1998], possédant les quatre caractéristiques suivantes :

- le chargement des classes à la demande et donc au plus tard ;
- la cohérence du typage lors de l'intégration de nouvelles classes ;
- la possibilité de reprogrammer la politique de chargement ;

– enfin, la gestion de plusieurs espaces de noms : chaque *classloader* possède son propre espace de noms, les classes Java ne sont pas référencées uniquement par leur nom, mais aussi par l'identificateur de leur classloader les ayant chargées.

Historiquement, le *classloader* a été mis en place pour répondre au besoin des *applets* en ce qui concerne le téléchargement et l'exécution de code distant. A présent, le *classloader* est utilisé pour toute intégration de classes au sein de la machine virtuelle Java. Il est notamment utilisé en support pour les *servlets* et les *Java beans*. Le *classloader* fonctionne de la façon suivante [Liang 1998][Malabarba 2000] : lorsque la machine virtuelle Java rencontre une référence vers une classe absente de la mémoire applicative lors de l'instanciation d'un nouvel objet, le *classloader* associé à l'objet ayant initié l'instanciation du nouvel objet est invoqué. Par défaut, il s'agit du *classloader* de la machine virtuelle Java qui ne sait récupérer les classes que depuis le système de fichier local. Si la classe est présente sur le système de fichier local, alors elle est intégrée à la mémoire applicative après passage dans le gestionnaire de sécurité, puis le nouvel objet peut être instancié. Cependant, la politique de chargement depuis le système de fichier local peut ne pas satisfaire toutes les applications. Ainsi, par extension de la classe *ClassLoader* et par redéfinition de certaines méthodes de cette classe, il est possible de modifier la politique de chargement des classes, et par-là même il est possible d'offrir par exemple un chargement de classes depuis un site distant. A noter que les classes systèmes sont toujours téléchargées par le *classloader* local.

Nous avons redéfini notre propre *classloader* afin d'autoriser l'intégration dynamique de classes Java offertes par le serveur de classes. Cette approche de configuration applicative par utilisation du *classloader* a été tout d'abord proposée par [Little 1998]. [Malabarba 2000] va plus loin en mettant en évidence la possibilité de définir un mécanisme supplémentaire permettant de remplacer directement, au sein de la machine virtuelle Java, une classe C1 par une classe C2 en supposant que C2 respecte un certain nombre de restrictions. Nos opérations de remplacement de tous les objets d'un module par des objets d'une nouvelle classe pourraient s'appuyer sur de tels mécanismes. Toutefois, cette approche nécessite une modification de la gestion des classes Java sur la pile de la machine virtuelle Java.

Enfin, en ce qui concerne la suppression de classes de la mémoire applicative, les deux mécanismes logiciels supports concernent le *classloader* et le ramasse-miettes. Le ramasse-miettes libère de la mémoire applicative toutes les classes qui ne sont plus utilisées. Cependant, dans certaines implémentations, le *classloader* gardant une référence sur l'ensemble des classes qu'il a téléchargé, il convient pour supprimer une classe de supprimer de la mémoire le *classloader* ayant chargé cette classe.

5.6.4 Environnement support sol

L'environnement support sol est présenté en deux étapes : l'environnement support sol associé à l'opérateur et l'environnement support sol associé au serveur de classes.

5.6.4.1 *Environnement support sol opérateur*

Description du middleware opérateur

Le middleware opérateur est présenté à la Figure 5-22. Ses principales fonctionnalités sont les suivantes :

– L'encodage des spécifications de reconfiguration (gestionnaire d'ordre).
– La récupération des rapports d'exécution de spécification.
– L'envoi d'ordre spécifique en réponse à une opération support de reconfiguration dynamique (demande de validation de l'exécution d'une spécification, par exemple, avant d'exécuter des opérations non réversibles).

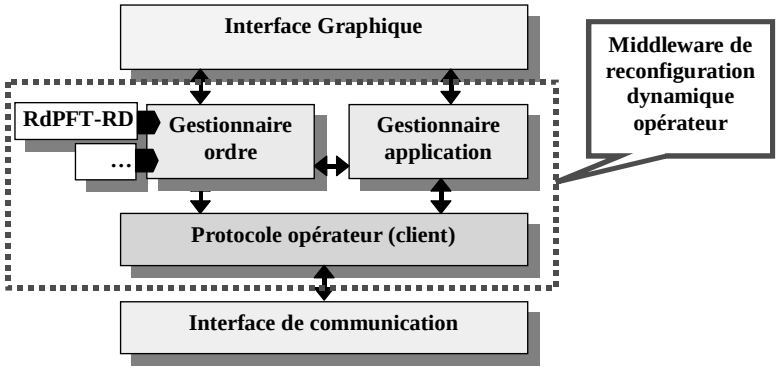


Figure 5-22. Architecture fonctionnelle du Middleware opérateur.

Gestionnaire d'ordre

Le gestionnaire d'ordre permet d'encoder une spécification d'opérations de reconfiguration dynamique. Tout comme le moteur embarqué de reconfiguration dynamique, ce gestionnaire d'ordre possède un mécanisme de type *plug-in*, qui permet de configurer l'encodeur associé à un langage de spécification. La Figure 5-22 donne l'exemple de l'utilisation d'un encodeur de RdPFT-RD. Cet encodage nécessite éventuellement le recours à un gestionnaire d'application dont le rôle est de maintenir à jour une image sol de l'architecture logicielle courante du logiciel embarqué : l'encodage peut-être réalisé au regard de la configuration applicative : par exemple, les identificateurs UML des modules peuvent être transformés en adresse mémoire (référence) du module.

Protocole opérateur

Le protocole opérateur permet au segment sol de communiquer avec le middleware embarqué afin d'échanger des informations de type *spécification de reconfiguration* ou de réponse à une demande de validation (cf. 5.5.3.3) dans le sens sol/bord, et de type rapport ou demande de validation, dans le sens bord/sol.

L'architecture de communication suit le paradigme client/serveur. L'entité serveur est placée à bord du satellite, alors que l'entité cliente est incluse dans la partie opérateur. Le protocole doit s'appuyer sur un protocole de transport offrant un mode connecté. En effet, le protocole d'échange de modèles (protocole opérateur) n'offre pas de signalisation permettant l'établissement de connexion au niveau de ce protocole. On dit par la suite qu'un client et un serveur sont connectés si le protocole de transport sous-jacent a pu établir une connexion entre les deux entités.

Le protocole de transport sous-jacent doit aussi offrir un mode de transmission de données totalement fiable et ordonné.

Nous ne présentons pas l'ensemble du protocole dont les détails peuvent être trouvés dans [Apvrille 2001c]. Nous nous contentons de présenter l'échange nominal de données entre l'entité *opérateur* et l'entité *embarqué* des middlewares (cf. Figure 5-23). Cet échange est le suivant : (1) l'envoi de la spécification à exécuter. Cette spécification a été préalablement encodée. (2) et (3) Le middleware bord décode la spécification, l'exécute puis envoi un rapport partiel, et une demande de validation lorsque l'opération de reconfiguration *validation*() est rencontrée. (4) Le middleware opérateur répond *ok* à la demande de validation. L'opération support de reconfiguration exécutée à bord retourne alors un code de retour favorable, la suite de la spécification est interprétée. Lorsque l'opération *fin*() est rencontrée par l'interpréteur, un rapport définitif est envoyé (5).

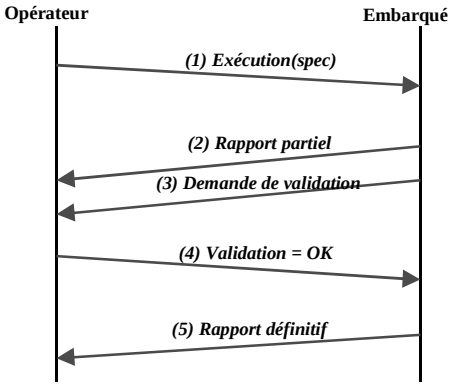


Figure 5-23. Echange nominal de données entre le middleware opérateur et le middleware embarqué.

5.6.4.2 *Middleware de reconfiguration dynamique : serveur de classes*

Description du middleware serveur de classes

L'architecture fonctionnelle du middleware de reconfiguration dynamique au niveau du serveur de classes (MRD.sc) est décrite à la Figure 5-24. Il est constitué :

- d'un gestionnaire de classes dont le rôle est de gérer l'écriture et la lecture de classes sur le stockage local ;
- d'une entité protocolaire : le protocole classes.

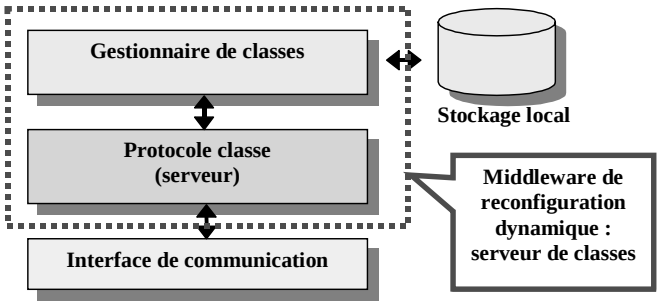


Figure 5-24. Architecture fonctionnelle du Middleware de reconfiguration dynamique : serveur de classes.

Protocole de classes

Le protocole de classes comprend deux entités : une entité serveur et une entité client. L'entité serveur gère localement, via le gestionnaire de classes, l'accès à des fichiers de type *.class* (bytecode Java). L'entité serveur est typiquement placée au niveau du segment sol, alors que l'entité cliente est placée soit au sol, pour permettre la gestion des classes par un opérateur, soit au niveau du satellite, pour permettre au satellite de télécharger les classes requises.

Tous les transferts s'effectuent en mode connecté. Le protocole repose sur un protocole de transport totalement fiable et ordonné, et offrant un mode connecté (typiquement TCP).

La Figure 5-25-(a) présente une suite typique d'échanges de données entre une entité cliente de type opérateur, et l'entité serveur de classes. Le client désire ajouter une classe, il spécifie le nom et la taille au serveur. (2) Le serveur répond *ok*. (3) Le client envoie les données de la classe. La Figure 5-25-(b) met en évidence un échange typique de données entre un client de type middleware embarqué et le serveur de classes. (1) Le client demande si le serveur possède la classe de nom *maClasse*. Si c'est le cas, le serveur émet (2) la taille de la classe, puis (3) les données de la classe.

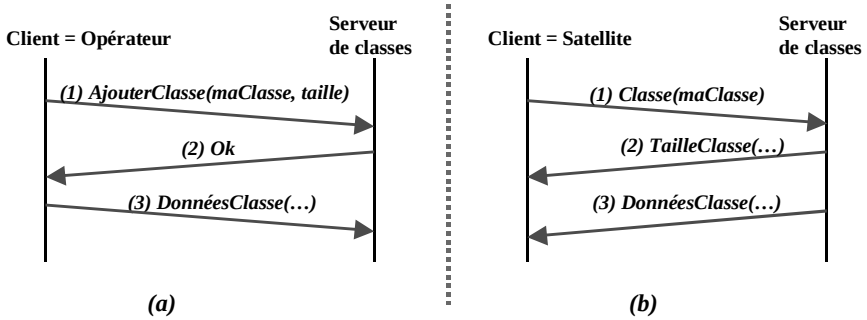


Figure 5-25. Echange nominal de données entre le serveur de classes et (a) un client de type opérateur, et (b) un client de type satellite.

5.7. Conclusion

L'objectif de ce chapitre était de mettre en évidence l'ensemble des mécanismes supports nécessaires à la reconfiguration dynamique d'applications spatiales embarquées. Une des contraintes fortes de cette reconfiguration est qu'elle doit garantir le respect de contraintes applicatives et tout particulièrement la continuité de service.

A cette fin, nous avons proposé :

- la spécification des scénarios de reconfiguration dynamique avec des réseaux de Petri étendus dont le pouvoir d'expression autorise la description des contraintes relatives aux opérations de reconfiguration ;
- une méthodologie logicielle enrichie par un cycle de validation formelle de la spécification du scénario de reconfiguration. Ce processus de validation formelle s'appuie sur la méthodologie TURTLE présentée au chapitre 4 et permet de valider formellement et a priori le respect des contraintes applicatives lors de l'exécution embarquée du scénario de reconfiguration. Le processus de validation formelle peut conduire soit au refus ou à la modification du scénario de reconfiguration (non respect de certaines contraintes applicatives), soit à l'acceptation du scénario (respect total des contraintes applicatives).
- Des mécanismes supports à déployer dans le système satellite. Ces mécanismes concernent tout particulièrement l'exécution du scénario de reconfiguration avec respect des contraintes associées à ces scénarios.

Le chapitre suivant est consacré à l'évaluation de notre approche de reconfiguration pour des logiciels responsables de charges utiles de télécommunication.

Chapitre 6

Étude de cas : maintenance d'un logiciel OBP

Résumé. Dans ce chapitre, nous proposons d'évaluer l'approche proposée dans les chapitres 4 et 5 en réponse au besoin de reconfiguration dynamique. Pour cela, nous considérons la reconfiguration dynamique d'un logiciel au service d'une charge utile de télécommunication. Nous présentons les fonctionnalités de ce logiciel, puis nous appliquons la méthodologie proposée au regard de différentes évolutions possibles : nous mettons en évidence cette méthodologie dans le cas d'une modification intra-composant et d'une modification inter-composant. Nous montrons dans les deux cas l'intérêt de la validation formelle a priori. Par la suite, nous évaluons l'implantation proposée en termes de mécanismes supports d'architecture à composant sur une plate-forme d'émulation du système satellite. Nous montrons la capacité de notre mise en oeuvre à modifier un logiciel de télécommunication en assurant la continuité des services utilisateurs.

6.1. Introduction

L'apparition de fonctionnalités logicielles dans les charges utiles de télécommunication soulève de nouveaux problèmes en termes de reconfiguration dynamique (respect des contraintes intrinsèques et extrinsèques) qui n'ont été qu'imparfaitement résolus à ce jour (cf. Chapitre 2 et Chapitre 3). Nous avons proposé dans le Chapitre 4 et le Chapitre 5 une méthodologie globale de reconfiguration dynamique reposant notamment sur un cadre de modélisation à des fins de validation a priori du respect des contraintes applicatives lors des mises à jour et sur des mécanismes supports à la mise à jour. Nous proposons dans ce chapitre d'évaluer ce cadre et les mécanismes supports sur le logiciel d'une charge utile de télécommunication de nouvelle génération i.e. avec commutation embarquée.

L'étude de cas présentée dans ce chapitre est divisée en deux parties : une partie concerne la méthodologie et une partie les mécanismes supports. En termes de méthodologie, il s'agit de montrer que le cadre formel présenté au chapitre 4 est d'une part apte à modéliser le logiciel considéré ainsi que les contraintes intrinsèques et extrinsèques relatives à ce logiciel et d'autre part que la validation formelle d'un script de reconfiguration amène des informations pertinentes concernant le respect des contraintes intrinsèques et extrinsèques. En termes de mécanismes supports, il convient de vérifier que l'architecture logicielle et le middleware support à la reconfiguration sont implantables dans un contexte spatial et que le processus de reconfiguration peut être mis en œuvre avec succès.

Le plan du chapitre est le suivant. Dans la première section, nous présentons le logiciel considéré. La deuxième section est consacrée à la validation formelle d'ordres de reconfiguration dynamique sur ce logiciel. Pour cela, nous mettons en évidence la modélisation de ce logiciel (architecture et contraintes). Puis nous mettons en évidence la construction et la validation de scripts de reconfiguration issus de nouveaux besoins concernant ce logiciel. Dans la troisième section, nous présentons l'évaluation des mécanismes supports au travers d'une plate-forme d'émulation. Enfin, nous concluons ce chapitre.

6.2. Présentation d'un logiciel charge-utile

La concurrence avec les technologies terrestres de transmission de données à haut-débit [Bigo 2000] amène les fabricants de systèmes de télécommunication par satellite à optimiser au mieux la bande passante par réutilisation de fréquences (satellites multi-spots avec commutation dynamique de paquets à bord des satellites) et par multiplexage dynamique temporel et fréquentiel sur les liens montants et descendants (cf. chapitre 2, paragraphe 2.2.2.2). Le projet SAGAM adresse tout particulièrement les fonctions mises en œuvre pour un tel système satellite gérant du trafic de type ATM [Roullet 1999]. Outre les fonctions d'administration de la matrice de commutation, des algorithmes assurent la gestion d'accès au canal satellite : l'attribution des slots sur les liens montants et descendants est réalisée au regard des connexions ATM acceptées dans le système. Dans cette étude de cas, nous supposons que les fonctionnalités sont toutes implantées de façon logicielle et à bord du satellite. Nous considérons par la suite les fonctionnalités suivantes :

- l'émission par le satellite d'un plan de trame toutes les 50 ms. Ce plan de trame indique à chaque utilisateur ses slots sur le lien montant (allocation de type MF-TDMA).
- Les fonctionnalités de type CAC, DAMA et BAC (cf. paragraphe 2.2.2.2). Pour mémoire, le CAC est en charge de l'acceptation des connexions ATM dans le système, le DAMA et le BAC sont respectivement responsables du multiplexage dynamique sur les liens montants et descendants. Les utilisateurs peuvent signifier une demande d'augmentation de trafic UBR ou VBR en émettant un bac-sig et un dama-sig en direction respectivement du BAC et du DAMA. De plus, nous considérons que la demande d'une nouvelle connexion est initiée par un message de type cac-sig.

6.3. Modélisation et validation formelle

Dans cette section, nous proposons une modélisation logicielle des fonctionnalités décrites dans la section précédente. Puis nous mettons en évidence la validation formelle de l'application d'opérations de reconfiguration dynamique sur ces fonctionnalités.

6.3.1 Modélisation logicielle

6.3.1.1 *Modélisation architecturale*

Il ne nous est pas possible dans ce document d'exposer l'ensemble des diagrammes TURTLE correspondant à la modélisation des fonctionnalités logicielles décrites précédemment. Nous donnons par la suite une description informelle des modules et une représentation du diagramme d'architecture.

Modules architecturaux

Le modules associés aux fonctionnalités logicielles décrites à la section 6.2 sont :

- un module *Bacsig* de réception des messages bac-sig, qui sont dirigés vers le *Bac*.
- Un module *Bac* qui traite les messages de type bac-sig, en déduit une allocation et envoie le résultat au module *Dama*.
- Un module *Cacsig* qui reçoit les cac-sig et les dirige vers le module *Cac*.
- Un module *Cac* qui traite les messages de type cac-sig et en déduit si la connexion est acceptée une allocation qui est envoyée au *Bac* et au *Dama*.
- Un module *Damasig* reçoit les signaux de type Dama-sig et les redirige vers le *Dama*.
- Un module *Dama* tient compte des messages en provenance du *Bac* et du *Cac* et traite les Dama-sig. Si le traitement d'un de ces messages amène à modifier les allocations du plan de trame, un message est envoyé au module *FarSender*.
- Un module *FarSender* modélise l'émission du plan de trame. La réception d'un message sur son premier port d'entrée correspond à une nouvelle allocation stipulée par le

Dama alors que la réception d'un message sur son deuxième port d'entrée correspond à un ordre d'émission immédiate du plan de trame.

- Enfin, un module *Clock* émet toutes les 50 ms un message sur le port d'entrée n°2 du module *FarSender*.

La modélisation de la réception des signaux (*Bacsig*, *Cacsig*, *Damasig*) exprime le comportement d'arrivée de ces signaux dans le système logiciel réel. Les comportements algorithmiques des modules sont principalement représentés sous forme de temps de traitement obtenus en simulation lors d'une étude interne d'Alcatel sur la charge algorithmique de ces fonctionnalités.

Modélisation TURTLE et Diagramme d'architecture

Le diagramme de classes TURTLE du système logiciel comprend 8 *Tclass* (une par module), une classe de routage, 7 classes de Buffer soit en tout 16 *Tclass*.

La Figure 6-1 présente le diagramme d'architecture de la modélisation TURTLE.

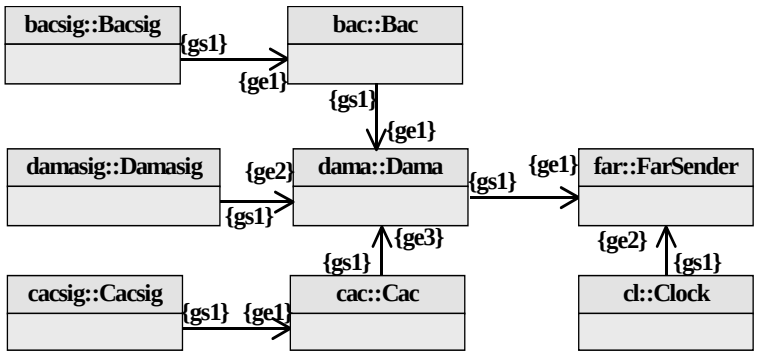


Figure 6-1. Diagramme d'architecture du logiciel étudié.

6.3.1.2 *Contraintes applicatives*

Nous proposons de valider formellement le respect de deux contraintes extrinsèques :

- l'émission périodique (période = 50 ms) d'un plan de trame ;
- tous les dama-sig et les bac-sig reçus après l'émission d'un plan de trame sont traités avant l'émission du plan de trame suivant.

Nous modélisons un observateur synchronisé avec *FarSender* pour chacune des contraintes. Le premier observateur analyse la date d'émission du premier plan de trame émis par *FarSender*, puis s'assure que les suivants sont émis exactement 50 ms après le précédent. Le deuxième note pour chaque période de 50 ms le nombre de dama-sig et de bac-sig traité par *FarSender* et vérifie que ce nombre est identique au nombre de dama-sig et bac-sig à traiter dans une période. Lorsque l'une des deux contraintes n'est pas respectée, l'observateur relatif à cette propriété effectue une synchronisation sur la porte « erreur » ce qui entraîne par-là même l'arrêt de *FarSender*.

A titre d'exemple de modélisation TURTLE, la Figure 6-2 illustre le comportement de *FarSender* et de ses deux observateurs.

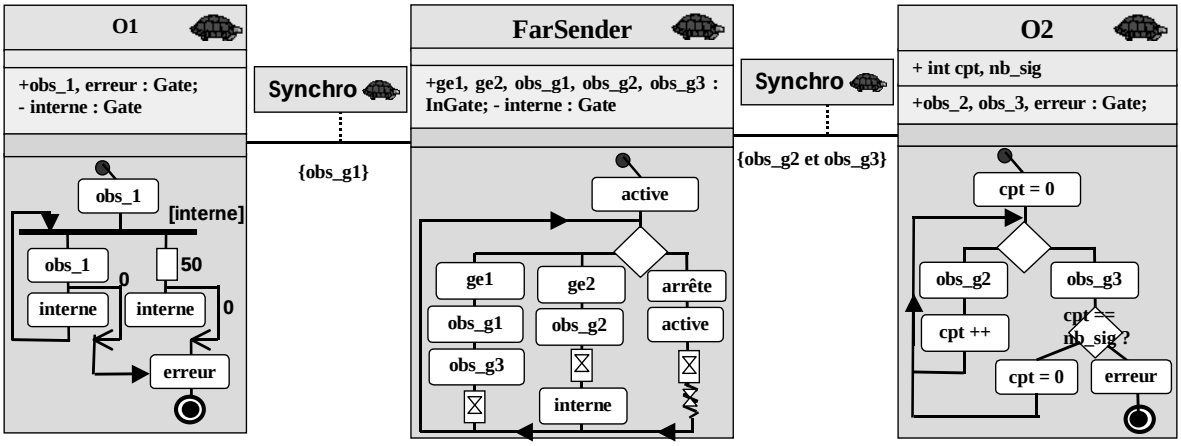


Figure 6-2. Modélisation TURTLE de l'observation de deux contraintes applicatives.

6.3.2 Validation formelle de reconfiguration dynamique

Les algorithmes relatifs aux nouvelles charges utiles sont très évolutifs (cf. paragraphe 2.2.2.2). Par la suite, nous considérons une modification intra-composant et une modification inter-composant justifiées par l'évolution de la nature du trafic utilisateur dans le système satellite.

6.3.2.1 *Modification intra-composant*

Dans cette partie, nous considérons la mise en place d'une nouvelle qualité de service de gestion des flux IP dans le système satellite. Le trafic IP est encapsulé dans des connexions UBR. Les slots sur les trames sont alloués de façon juste entre les différentes connexions UBR au regard de certains paramètres stipulés dans les messages bac-sig. Nous proposons afin d'introduire cette qualité de service de modifier les services d'allocation des slots. Cette modification se traduit par un nouvel algorithme de traitement des bac-sig au niveau du module *Bac*. Afin de réaliser cette modification de façon dynamique, il convient de spécifier le script de reconfiguration inhérent à cette modification, puis de valider formellement que l'exécution de ce script sur l'application respecte les deux contraintes applicatives précitées (paragraphe 6.3.1.2).

La modélisation UML du BAC est la suivante (cf. Figure 6-3-(a)). Le module BAC est constitué d'une classe TURTLE *Bac* (tâche logicielle du module) qui est composée d'un objet instance de *BacAlgo*, lui-même composé d'objets instances de *BacData*. Les algorithmes du Bac étant implantés dans la classe *BacAlgo*, la modification souhaitée peut se réaliser en introduisant une nouvelle classe *BacAlgo2* héritant de *BacAlgo* (modification intra-composant, cf. 5.3.3.2). D'après le paragraphe 5.5.3.3, la modification de cette classe se traduit par l'opération de reconfiguration *modifierClasse(BacAlgo, BacAlgo2, Bac)*.

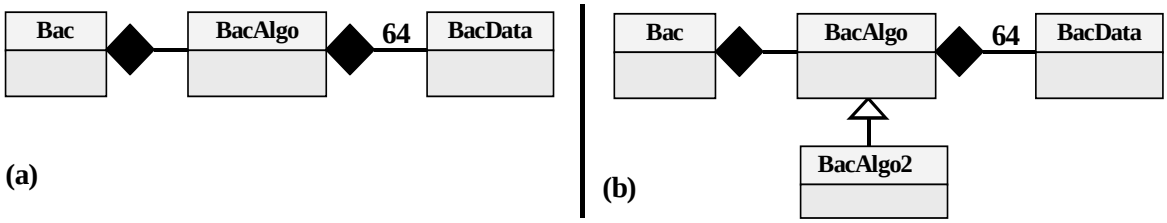


Figure 6-3. Modélisation UML du module BAC. (a) : Avant reconfiguration. (b) : Après reconfiguration.

La spécification au sein d'un RdPFT-RD des opérations de reconfiguration est obtenue par raffinage successif des opérations initiales de reconfiguration dynamique (cf. 5.5.3.3). De plus, il est possible de modifier les dépendances entre les opérations de reconfiguration dynamique afin de privilégier la réversibilité ou la gestion des ressources. Par la suite, nous mettons en évidence la construction de la spécification des opérations de reconfiguration.

Le RdPFT-RD spécifiant les opérations de reconfiguration dynamique issues de *modifierClasse(BacAlgo, BacAlgo2, Bac)* est présenté à la Figure 6-4. La place p1 modélise le chargement de la classe *BacAlgo2*, précondition de la création de l'objet *o*, instance de *BacAlgo2*. Une fois *o* créé, il convient de mettre à jour le lien de l'objet de type Bac vers l'objet de type *BacAlgo* (l'attribut *algo* de la classe Bac doit pointer vers *o* : cela se traduit par l'opération *écrireAttribut(o, algo, bac))* et il convient de recopier dans *o* l'état de l'objet instance de *BacAlgo* (appelé *ba*). Cela se traduit par les opérations état = *encoderObjet(ba)* et *positionnerEtat(état, o)*. Ces trois opérations de manipulation des objets internes au module *Bac* ne peuvent être réalisées que si ce module a rejoint son point de reconfiguration, d'où la suspension (place p3) et l'activation (place p7) du *Bac* respectivement avant et après ces opérations. Enfin, l'objet *ba* peut être détruit après l'encodage de son état (place p8). Notons que pour optimiser les ressources (cf. 5.5.3.3), il conviendrait de détruire l'objet *ba* avant de charger la classe *BacAlgo2* et de créer *o*. Notons aussi que l'opération *détruireObjet(ba)* rend l'exécution de la spécification non réversible : il conviendrait donc de déporter cette opération après *activer(Bac)* (cf. 5.5.3.3).

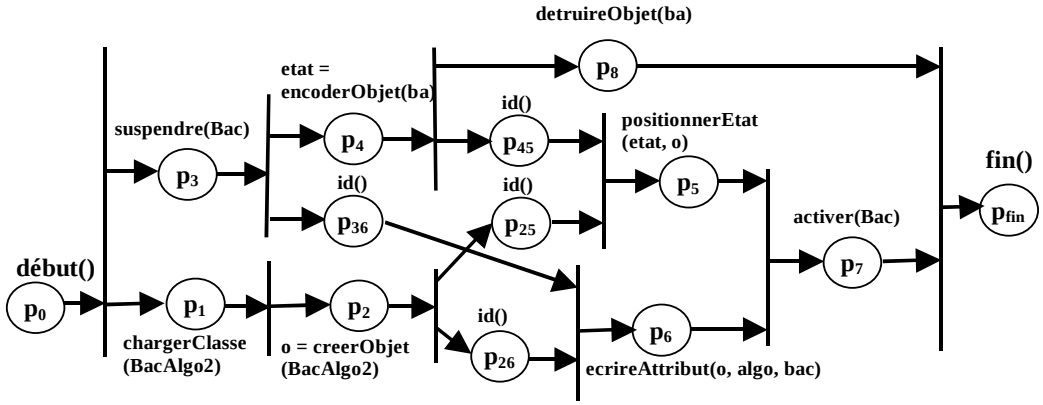


Figure 6-4. RdPFT-RD n°1 relatif à la modification de l'algorithme du Bac.

Nous avons modélisé cette spécification dans une classe *GestConf* intégrée à la modélisation TURTLE de l'application. La suspension et l'activation se modélisent comme des synchronisations avec la *Tclass* Bac (cf. 4.4.4.3). Le changement d'algorithme du Bac est modélisé comme un nouveau comportement au sein de la *Tclass* Bac, comportement qui est activé par synchronisation entre le gestionnaire de configuration et la *Tclass* Bac (cf. cf. 4.4.4.3). Enfin, les autres opérations sont modélisées par des délais.

L'analyse d'accessibilité met en évidence que l'exécution de cette spécification ne peut se réaliser avec respect de la contrainte de traitement des bacsig (il s'agit de la deuxième contrainte observée dans le modèle TURTLE). En effet, l'analyse met en évidence que le temps de suspension du *Bac* est tel qu'il est supérieur à une période (i.e. 50 ms). Afin de résoudre ce problème, nous proposons d'ajouter une contrainte de dépendance entre la suspension du *Bac* et la création de l'objet *o*. Ainsi, la suspension du *Bac* se fait « au plus tard », ce qui pourrait permettre de limiter la suspension du Bac. Cette contrainte de dépendance nous amène à modifier la spécification de la reconfiguration (cf. Figure 6-6) afin de prendre en compte cette nouvelle contrainte : *suspendre(Bac)* est réalisée après *creerObjet(BacAlgo2)*. Reprenant le même principe de modélisation de *GestConf* (cf. 4.4.4.3), la nouvelle analyse d'accessibilité met en évidence le caractère périodique du logiciel (période = 50 ms) et montre la possibilité d'effectuer cette reconfiguration si et seulement si elle est démarrée en milieu de cycle c'est-à-dire, si t0 représente la date de début d'exécution de l'application alors la reconfiguration peut-être démarrée à l'instant t si $\exists k$ / $t \geq t0 + k * \text{période} + 17000$ µs et $t \leq t0 + k * \text{période} + 43000$ µs. A titre d'exemple d'analyse d'accessibilité, la Figure 6-5 met en évidence un extrait du graphe d'accessibilité lorsque la reconfiguration dynamique commence à t0 + 5000 µs. La synchronisation sur la porte *g4* traduit la demande de la part du module *Clock* d'Émission d'un plan de trame par *FarSender*. Cette demande d'émission d'un plan de trame est

suivie par la synchronisation entre *FarSender* et un observateur sur la porte d'observation *go3* correspondant à l'analyse par l'observateur du nombre de messages traités avant émission de ce plan de trame (contrainte applicative décrite au paragraphe 6.3.1.2) Le nombre de messages traités étant incorrect, l'observateur réalise une synchronisation sur la porte *erreur3*, ce qui traduit une violation de la contrainte applicative et donc une absence de continuité de service.

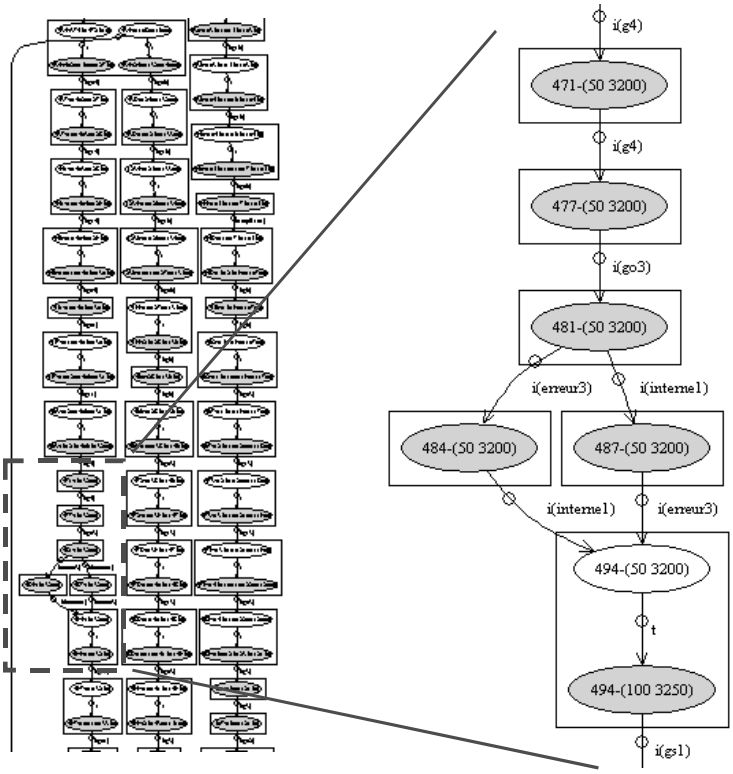


Figure 6-5. Extrait du graphe d'accessibilité de la reconfiguration dynamique du Bac.

Ce résultat d'analyse nous conduit à modifier la spécification de reconfiguration : une première opération attend que l'application se trouve dans un intervalle temporel favorable avant de passer à l'opération de chargement de *BacAlgo2* (cf. Figure 6-6).

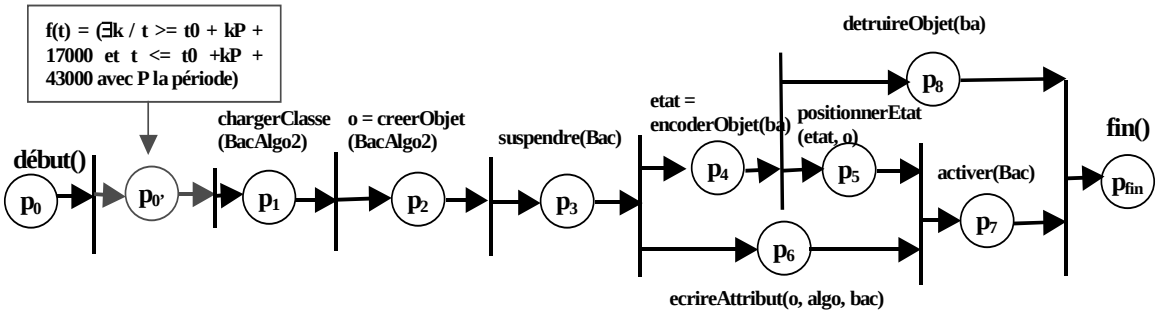


Figure 6-6. RdPFT-RD n°2 relatif à la modification de l'algorithme du Bac.

6.3.2.2 *Modification inter-composant*

Dans cette partie, nous mettons en évidence l'intérêt des opérations inter-composants pour intégrer de nouveaux services au sein d'une application en cours de fonctionnement. Nous souhaitons à présent gérer le trafic de type VoIP (*Voice Over IP*) dans le système satellite. Par défaut, ce type de trafic est transmis au sein d'une connexion CBR. Dans le système actuel, les connexions CBR se voient attribuer un nombre de slots constant dans la trame montante. Mais ces slots n'ont pas une position fixe dans la trame ce qui induit une gigue perturbatrice pour les flux téléphoniques.

Actuellement, l'allocation des slots sur la trame montante pour les connexions CBR est réalisée par le *Dama*. Ainsi, nous proposons de modifier l'algorithme d'allocation du Dama. De plus, afin qu'une station utilisateur puisse indiquer au système satellite ses connexions CBR qui transportent un flux de type VoIP, nous introduisons un nouveau signal voip-sig émit par les stations sols et reçu par le système satellite.

La modélisation de ces nouvelles fonctionnalités se traduit par l'introduction de deux nouveaux modules :

- un module *Voipsig* assure la réception des messages voip-sig qui sont dirigés vers le module Voip ;
- un module *Voip* déduit une allocation des messages voip-sig. Cette allocation est ensuite transmise au module Dama.

Le module Dama doit être modifié comme suit : un nouveau port doit être ajouté à ce module (port de réception des allocations transmises par le module *Voip*). De plus, le comportement interne de réception de ces nouveaux messages doit être ajouté au module. Mais contrairement à la modification applicative précédente (cf. 6.3.2.2), nous supposons ici que la modification induite ne peut-être supportée par des opérations intra-composants, ce qui nous conduit à remplacer le module *Dama* par un nouveau module *Dama2*. Finalement, le diagramme d'architecture du nouveau système est illustré à la Figure 6-7. Ce diagramme met en évidence l'ajout de trois nouveaux modules et la suppression du module *Dama*.

Par la suite, Nous mettons en évidence la construction de la spécification de la reconfiguration permettant de passer de l'ancien au nouveau système. Tout d'abord, il s'agit de générer l'ensemble des opérations dont l'application sur l'ancien système logiciel permet de générer le nouveau système logiciel. Selon l'algorithme de génération de l'ensemble O des opérations (cf. 5.5.3.3), on a (on suppose qu'initialement O = ∅) :

- quatre liens sont supprimés : O = O ∪ {détruireLien(bac, gs1), détruireLien(damasig, gs1), détruireLien(cac, gs1), détruireLien(dama, gs1)} ;
- six liens sont créés : O = O ∪ {créerLien(voipsig, gs1, voip, ge1), créerLien(voip, gs1, dama2, ge2), créerLien(bac, gs1, dama2, ge1), créerLien(damasig, gs1, dama2, ge2), créerLien(cac, gs1, dama2, ge3), créerLien(dama2, gs1, far, ge1)} ;
- deux modules sont créés : O = O ∪ {ajouterModule(vopip), ajouterModule}.
- le module Dama2 remplace le module Dama : O = O ∪ {remplacerModule(dama)}.

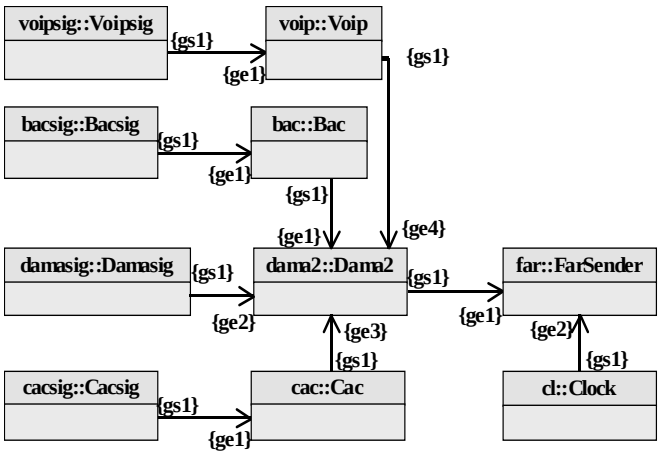


Figure 6-7. Diagramme d'architecture du nouveau système logiciel.

Le raffinage des opérations de l'ensemble O a conduit à la spécification des opérations de reconfiguration (cf. Figure 6-8). Cette spécification suppose que la fonction *id*() est associée au place « p » (cf. 5.4.3.3). Cette spécification a été modélisée en TURTLE puis validée. La validation formelle a permis de mettre en évidence l'impossibilité de réaliser cette reconfiguration avec respect

des contraintes applicatives. En effet, les modules *bac* et *damasig* sont suspendus pendant toute la durée du téléchargement de la classe *Dama2*, qui est très longue au regard de la période du logiciel (plusieurs centaines de périodes). Pour corriger ce problème, nous avons modifié la spécification comme suit : la suspension des modules (places p1, p2, p3 et p6) n'est réalisée qu'après chargement et création des nouveaux modules (places p4, p7, p8, p12, p13, et p14 de la Figure 6-8). La validation de cette nouvelle spécification permet de mettre en évidence un intervalle temporel pendant lequel la reconfiguration peut être menée avec succès.

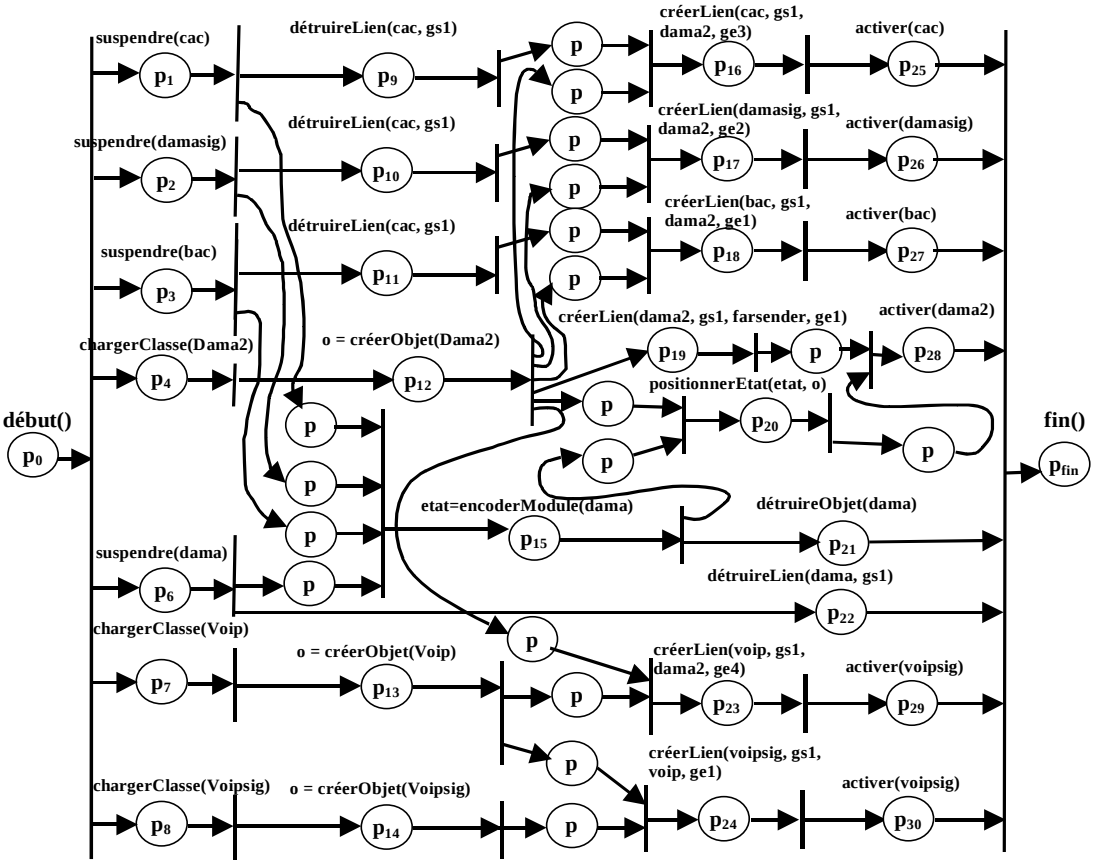


Figure 6-8. Spécification des opérations de reconfiguration (reconfiguration n°2).

6.4. Évaluation des mécanismes supports

6.4.1.1 *Besoin d'une plate-forme d'évaluation*

L'évaluation des mécanismes supports de la reconfiguration dynamique concerne :

- la faisabilité de l'intégration dynamique de code sur une application satellite développée en Java ;
- la performance de l'architecture support (middleware de reconfiguration dynamique, cf. 5.6.3.3) et notamment des protocoles qui entrent en jeu dans le système.

Le contexte particulier de l'étude (reconfiguration dynamique de logiciels de télécommunication par satellite) nous incite à proposer une plate-forme d'expérimentation tenant compte des particularités du système satellite. Ces particularités sont les suivantes :

- les mécanismes supports sont répartis et utilisent des liens satellites à des fins de communication entre les différentes entités, par exemple, pour la communication entre le middleware opérateur et le middleware embarqué. Ce lien satellite de type télécommande / télémesure possède des contraintes fortes en termes de bande

passante et de taux de perte qui ont une répercussion importante sur la performance du système de reconfiguration (protocoles d'échanges de spécification et de téléchargement de code).

- Les services sont offerts aux utilisateurs via un lien satellite qui là encore possède des caractéristiques influant directement sur la performance du système. En effet, le délai de communication entre un utilisateur et le satellite est de 125 ms.

Finalement, afin d'évaluer les mécanismes supports, nous avons mis en évidence le besoin d'un environnement reproduisant le caractère particulier de transmission inhérent à un système satellite géostationnaire.

6.4.1.2 *Choix d'une technique d'évaluation*

Trois techniques ont été proposées à des fins d'évaluation d'architectures logicielles réparties et de protocoles : la *mesure directe*, la *simulation*, et l'*émulation*.

La mesure directe consiste à utiliser un système réel, ce qui peut se révéler particulièrement difficile à mettre en œuvre et coûteux dans le cas d'un système satellite. De plus, l'expérimentation dans des conditions particulières (congestions, taux d'erreur élevé, etc.) n'est pas directement contrôlable car certains facteurs sont dépendants de l'environnement donc non maîtrisables (conditions météorologique, congestion des nœuds du réseau Internet, etc.).

La simulation est largement utilisée quand les conditions réelles sont difficiles à obtenir et à contrôler. Plusieurs outils répondent à ce besoin. OPNET [OPNET 2002] est particulièrement utilisé dans le milieu industriel. NS [NS 2002] (*Network simulator*) est un outil *freeware* développé à l'université de Berkeley et largement utilisé dans le milieu académique et au sein de l'IETF. Le problème de ces outils réside dans le fait qu'ils ne permettent pas de tester les implémentations réelles des protocoles.

A contrario, avec l'émulation, les implémentations réelles des protocoles peuvent être évaluées au-dessus d'une couche réseau virtuelle qui offre un service défini par l'utilisateur. La qualité de service peut être contrôlée de façon fine (en termes de délai de transmission, de distribution des pertes, etc.) afin d'émuler par exemple les caractéristiques des liaisons satellites. Enfin, l'émulation peut être facilement mise en œuvre sur un réseau local de stations de travail.

6.4.1.3 *La plate-forme NINE*

NINE (*Nine Is a Network Emulator*) est une plate-forme d'émulation développée au département informatique de l'ENSICA et dont le but est d'offrir un service au niveau IP dont la qualité de service peut être contrôlée de manière dynamique. L'avantage de ce système est que l'ensemble des applications et protocoles développés de façon native au-dessus d'IP peuvent être évalués directement sur la plate-forme.

La plate-forme est constituée de deux principaux éléments (cf. Figure 6-9) : des stations de travail et des routeurs. Les stations de travail sont identiques à celles utilisées dans les situations réelles ; mais elles sont connectées à un réseau physique capable d'offrir au moins la capacité du réseau cible (l'émulation est notamment basée sur un surdimensionnement). L'émulation est réalisée au sein de routeurs IP qui permettent le contrôle de la qualité de service des paquets traités. Ces routeurs sont basés sur le module *dummynet* de *FreeBSD* [Rizzo 2000] qui permet le contrôle du trafic, en transit entre les différentes interfaces réseau de la machine, en termes de bande passante, de taille de file d'attente, de délai et de taux de perte. Une interface graphique développée par nos soins permet le contrôle dynamique et à distance de ces paramètres.

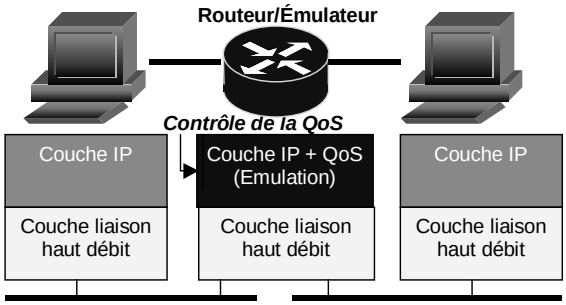


Figure 6-9. Architecture de NINE.

La qualité de service entre les stations applications clientes peut être contrôlée de façon fine au moyen de *pipes* : à un *pipe* correspond une qualité de service définie selon les quatre paramètres décrits dans le paragraphe précédent. Des règles (*rules*) associées aux *pipes* décrivent des contraintes en termes de protocoles, port adresses IP, etc. Tout paquet IP transitant par le routeur est classifié selon la règle adéquate et la politique du *pipe* correspondant à cette règle lui est appliquée. L'algorithme de type WFQ (*Weight Fair Queuing*) implémenté dans *dummynet* et appelé WF2Q+ admet une complexité en O(log N) (N = nombre de flux actifs) ce qui permet la gestion efficace de milliers de flux.

La plate-forme est actuellement composée de 3 routeurs (PC Pentium II and III) et d'une douzaine de stations de travail (PC et stations Sun) utilisées pour exécuter les applications utilisateurs et les protocoles à évaluer. Toutes ces machines sont interconnectées par des sous-réseaux Ethernet reliés via des *hubs* ou des *switch* ce qui permet une configuration rapide de l'architecture.

Nous définissons trois vues permettant de décrire une expérimentation sur la plate-forme NINE (cf. Figure 6-10). La *vue système* définit le système réel et donc les caractéristiques devant être émulées. La *vue émulation* propose une modélisation sous forme de *pipes* des caractéristiques devant être émulées : à chaque *pipe* est ainsi associé un scénario de qualité de service. Enfin, la vue physique représente l'architecture réseau en termes d'équipement, de table de routage, etc. permettant la réalisation de cette émulation.

Évaluation des mécanismes supports

Grâce à la plate-forme d'expérimentation NINE, nous avons évalué les mécanismes supports à la reconfiguration sur une émulation du système SAGAM. En effet, notre expérimentation nécessite l'émulation de la QoS entre les UES et les services utilisateurs de SAGAM, entre le middleware de reconfiguration embarqué (MDRe) et le middleware opérateur (MDRo) et enfin, entre le middleware de reconfiguration embarqué et le middleware serveur de classes (MDRs).

La Figure 6-10 décrit les trois vues de NINE répondant à notre besoin d'émulation. Le système réel est composé d'un satellite à bord duquel sont embarqués des services logiciels et le MDRe. Un spot comprend les utilisateurs (trois UES). Enfin, le centre de contrôle du satellite comprend le MDRo et le MDRs. Le vue émulation présente les deux *pipes* correspondant à deux qualités de service distinctes. Le *pipe1* représente le lien satellite entre les UES et le satellite. Les paramètres de qualité de service choisis sont un délai de 125 ms et une bande passante de 2 Mb/s. Le *pipe2* émule le lien satellite entre les entités du centre de contrôle et le satellite. La qualité de service doit être celle d'un lien TM/TC. Nous avons choisi les paramètres de qualité de service suivants : un délai de 125 ms, un taux de perte de 5% et un débit de 3Kb/s. Enfin, la vue physique met en évidence l'utilisation de trois sous-réseaux : un dédié au satellite, un dédié aux stations utilisateurs et enfin le dernier dédié au centre de contrôle. Notons que seuls deux sous-réseaux sont nécessaires car les UES et le centre de contrôle peuvent se situer dans le même sous-réseau puisque aucune communication directe n'est réalisée entre eux : la différentiation entre les deux *pipes* est en effet réalisée de façon logicielle au niveau du routeur. Cependant, pour des raisons de

surdimensionnement et de disponibilité d'équipements, nous avons préféré utiliser trois sous-réseaux.

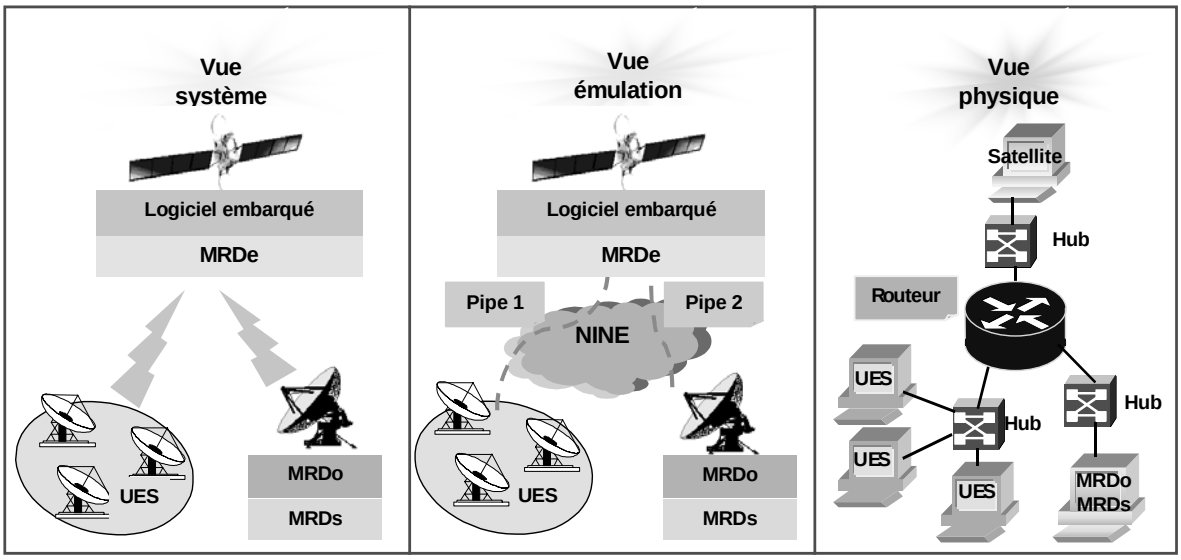


Figure 6-10. Configuration NINE du système SAGAM.

Le logiciel embarqué implémente les fonctionnalités décrites dans la section 6.2, à savoir le CAC, le BAC, et le DAMA, et effectue l'émission d'un plan de trame toutes les 50 ms. Du côté client, le logiciel de l'UES génère du trafic de type Internet : afin de réaliser ce générateur de trafic, nous nous sommes inspirés de traces relatives à des transmissions de données de type HTTP et FTP : cela nous conduit à générer de façon apériodique des rafales de données et de façon continue un flux constant de données. Une fois générés, les paquets IP sont encapsulés dans des cellules ATM. Ces cellules sont stockées dans un buffer local. Le logiciel client écoute les plans de trame et insère ses cellules dans les slots de la trame montante au regard des indications fournies dans le plan de trame. Au niveau de l'opérateur, une interface graphique permet de demander l'exécution d'opérations de reconfiguration dynamique et d'obtenir un rapport cette exécution. Enfin, un serveur de classes est en attente des demandes de téléchargement de classes Java.

L'expérimentation menée est la suivante.

Pour chaque UES, toutes les 10 cellules, nous enregistrons le délai d'attente des cellules dans le buffer sol (CTD – *Cell transfer Delay*), le nombre de cellules en attente dans les buffers et enfin le nombre de cellules détruites suite à un débordement de buffer. Nous démarrons successivement trois stations sols, puis nous lançons une reconfiguration dynamique qui consiste à offrir un service différencié à une station sol spécifique de telle sorte que cette station sol ne perde plus de cellules à condition bien sur que son trafic UBR n'excède pas la bande passante restante après les allocations CBR et VBR. Pour offrir un tel service différentié, nous modifions le comportement interne du BAC (algorithme d'allocation).

Les résultats de cette expérimentation sont présentés à la Figure 6-11 qui décrit le CTD des stations sols en fonction du temps. Tout d'abord, les stations démarrent successivement : (1), (2) et (3)). Tant que la première station est seule à émettre, son CTD est compris entre 0 et 200ms. Mais le démarrage des autres stations induit un CTD important pour toutes les stations (parfois plus de 1000 ms), ce qui traduit un partage des ressources. La reconfiguration dynamique est démarrée (4). Cela se traduit par l'envoi au MRDe d'un réseau de Petri décrivant les modifications à effectuer sur le système. L'interprétation de ce réseau de Petri induit notamment le téléchargement de classes Java via le serveur de classes et l'arrêt de modules qui est matérialisé par la barre verticale (5). Cet arrêt de modules correspond à un arrêt interne au logiciel (point de reconfiguration) et non à un arrêt des services offerts à l'environnement. Une fois la reconfiguration achevée, la station démarrée

en premier se voit offrir un service différencié (6) : elle ne perd plus de cellules. En outre, son CTD demeure inférieur à 500 ms : c'était la seule station qui était temporellement contrainte. Le service offert par l'application embarquée a donc été reconfiguré sans altération des autres services offerts aux stations sols.

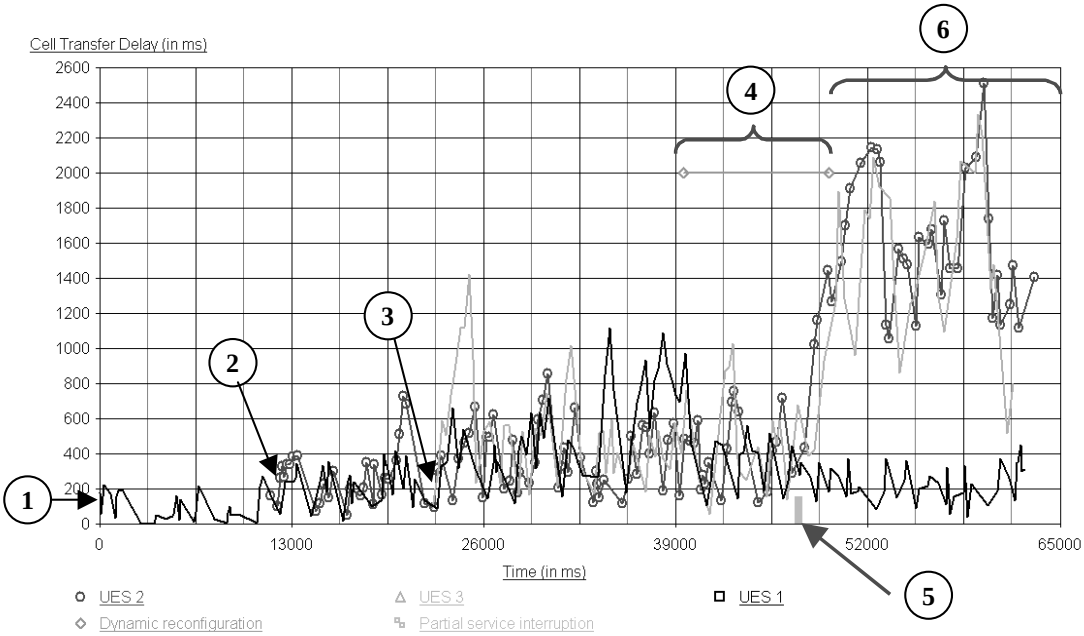


Figure 6-11. CTD des UES en fonction du temps.

6.5. Conclusion

Dans ce chapitre, nous avons évalué notre approche de reconfiguration dynamique sur un logiciel responsable du multiplexage dynamique sur une charge utile de nouvelle génération embarquée dans un satellite.

Dans un premier temps, nous avons présenté notre démarche méthodologique de validation formelle de scénarios de reconfiguration dynamique dans le cas d'une modification intra-composant puis d'une modification inter-composant. Le processus de validation formelle nous a amené dans les deux cas à modifier le scénario de reconfiguration dynamique dont l'exécution embarquée aurait pu mener à la violation de propriétés extrinsèques applicatives.

Dans un deuxième temps, nous avons proposé une plate-forme d'évaluation (nommée NINE) des mécanismes supports à la reconfiguration dynamique. Cette plate-forme repose sur le principe de l'émulation qui a pour avantage d'autoriser des évaluations de performance sur des implantations de protocoles et non sur des modélisations de ces protocoles à l'exemple de la technique de simulation.

Au-dessus de cette plate-forme, nous avons implémenté les différentes entités logicielles des mécanismes supports. Puis, nous avons montré la capacité de cet environnement à reconfigurer, de façon transparente et en cours de fonctionnement, un logiciel embarqué.

Chapitre 7

Conclusion Générale

Dans le domaine des télécommunications, la grande évolutivité des flux utilisateurs soulève le problème de la reconfiguration dynamique des applications logicielles en charge de ces flux et embarquées dans les satellites. L'étude bibliographique des travaux du domaine a montré que les mises à jour logicielles actuellement pratiquées dans le domaine du spatial ne peuvent répondre aux contraintes de continuité de service requises par de telles fonctions de télécommunication.

Fort de ce constat, nous avons analysé les contributions en terme de reconfiguration dynamique des autres domaines d'applications. Une fois exclues les solutions matérielles pour des questions de mise en œuvre, nous nous sommes tourné vers les environnements à base d'architecture à composants en raison de leur haut degré de reconfigurabilité et de la possibilité de les mettre en œuvre dans les calculateurs embarqués sur satellite. Cependant, à l'instar de l'ensemble des techniques de reconfiguration proposées à ce jour, les environnements à base de composants souffrent d'un déficit de prise en compte des contraintes applicatives temps-réel intrinsèques et extrinsèques Ainsi, dans ce mémoire, nous avons proposé un environnement de reconfiguration dynamique d'applications spatiales (et plus généralement d'applications embarquées) qui, dans un cadre formel, tienne compte des contraintes applicatives précitées.

7.1. Rappel des contributions

La gestion du caractère dynamique de la reconfiguration a conduit les concepteurs des environnements à base de composants logiciels à introduire la notion de point de reconfiguration, d'encodage des états des composants et d'expression des contraintes intrinsèques au sein des langages de description des architectures (langages dits ADL). Au mieux, ces approches permettent de gérer la consistance logique applicative : à ce titre, les contraintes temporelles intrinsèques et extrinsèques sont totalement ignorées.

Dans le but de gérer à la fois les contraintes logiques et temporelles, nous proposons une approche basée sur une validation formelle a priori du respect des contraintes précitées et sur des mécanismes supports à la reconfiguration dynamique adaptés à la gestion de ces contraintes.

Validation formelle a priori

La première contribution proposée par cette thèse (Chapitre 4) aborde le problème de la validation formelle a priori des reconfigurations. A cette fin, nous avons défini une architecture à composants (paragraphe 4.2) dans laquelle ces derniers sont assimilés à des tâches logicielles.

Nous avons ensuite introduit un cadre formel (paragraphe 4.3) de modélisation à la fois de cette architecture, des contraintes intrinsèques et extrinsèques et enfin des opérations de reconfiguration (modélisation du gestionnaire de configuration). Si les architectures logicielles à composants sont habituellement décrites à l'aide d'un langage spécifique de type ADL, pour ce qui

nous concerne, nous nous sommes appuyés sur le langage UML afin de répondre aux contraintes industrielles en termes de langage de conception logicielle.

De plus, face aux insuffisances d'UML en termes de structuration et d'expression du comportement, nous avons proposé un profil UML (TURTLE) qui enrichit UML par des opérateurs de structuration (expression explicite du parallélisme et de la synchronisation) et des opérateurs temporels (description du comportement). Ce profil intègre de plus un processus de validation formelle qui consiste à dériver de la modélisation UML une spécification en RT-LOTOS, langage formel supporté par un outil de validation.

Nous avons appliqué avec succès ce profil TURTLE à la modélisation de l'architecture à composants précédemment définie (paragraphe 4.4) : les propriétés intrinsèques et extrinsèques sont modélisées sous forme de classes UML d'observation et les opérations de reconfiguration dynamique sont également modélisées par une classe (gestionnaire de configuration).

Mécanismes supports à la reconfiguration

Notre deuxième contribution (Chapitre 5) consiste à intégrer notre cadre de validation formelle au sein d'un processus de reconfiguration dynamique d'une application spatiale. Ce processus de reconfiguration dynamique comporte trois étapes fondamentales : la spécification des opérations de reconfiguration dynamique à effectuer sur une application pour répondre à un nouveau besoin, une méthodologie logicielle intégrant une phase de validation formelle de la spécification des opérations et enfin des mécanismes embarqués support à l'exécution de la spécification des opérations.

Pour la première étape, nous avons proposé un formalisme à base de réseaux de Petri à flux temporels afin d'exprimer les contraintes d'exécution relatives aux opérations de reconfiguration : contraintes logiques d'exécution d'une opération, contrainte temporelle d'exécution d'une opération, contraintes logiques et temporelles entre l'exécution de deux opérations.

Pour la deuxième étape, nous avons proposé un cycle itératif UML s'appuyant sur le profil UML TURTLE et enrichi par une phase de validation formelle : la spécification des opérations de reconfiguration dynamique est modélisée en TURTLE conjointement avec la modélisation applicative puis validée formellement.

Enfin, la dernière étape nous a amenés à proposer un support embarqué à notre architecture à composants et à l'exécution de la spécification des opérations de reconfiguration. Un environnement logiciel basé sur le langage Java combiné à un noyau temps-réel a été choisi en raison notamment de la modularité de ce langage objet (mise en œuvre de l'architecture à composants) et des mécanismes d'intégration dynamique de code de la machine virtuelle Java. Nous avons montré par la suite l'adéquation de cet environnement avec les contraintes des logiciels spatiaux et sa capacité à mettre en œuvre les opérations de reconfiguration que nous avons définies.

Étude de cas : reconfiguration de services logiciels de télécommunication embarqués dans un satellite

Afin d'évaluer notre réponse au problème de la reconfiguration dynamique, le Chapitre 6 propose une étude expérimentale des mécanismes proposés sur une application spatiale. Pour cela, nous avons évalué la reconfiguration dynamique d'un logiciel embarqué dans un satellite et assurant des fonctions de télécommunication telles que le multiplexage temporel dynamique sur les liens montants et descendants.

Tout d'abord, nous avons mis en évidence la pertinence de notre cadre formel basé sur UML pour modéliser puis valider formellement le respect de contraintes applicatives lors d'une reconfiguration. Pour cela, nous avons réalisé une modélisation TURTLE du logiciel pour laquelle nous avons validé formellement le respect de contraintes logiques et temps-réel au regard de modifications intra et inter composants. Les informations obtenues grâce à ces validations garantissent la continuité de service de l'application lors de la mise à jour.

Par la suite, nous avons évalué la mise en œuvre des mécanismes supports que nous avons implémentés. Pour effectuer cette évaluation dans des conditions réalistes, nous avons développé

une plate-forme (*NINE*) d'émulation des caractéristiques de transmission du système satellite. Grâce à cette plate-forme, nous avons notamment mis en évidence la capacité de notre environnement support à intégrer de façon dynamique des nouveaux services au sein de l'application en cours de fonctionnement tout en respectant les contraintes de continuité de service.

7.2. Perspectives

Modélisation et validation formelle pour la reconfiguration dynamique

Le cadre formel TURTLE que nous avons proposé à des fins de garanties du respect des contraintes intrinsèques et extrinsèques applicatives peut être amélioré en termes de modélisation et de validation.

En termes de modélisation, trois enrichissements seraient souhaitables.

Tout d'abord, l'introduction d'opérateurs de comportement décrivant de façon explicite les points de préemption des composants architecturaux faciliterait la modélisation de l'ordonnancement. Cette action doit être menée en conformité avec les spécifications temps-réel d'UML concernant l'ordonnançabilité [OMG 2002b]. Nous proposons aussi d'enrichir la description du comportement interne des classes de l'environnement TURTLE (dites *Tclass*) par des opérateurs de comportement abstrait. Ces opérateurs pourraient être choisis au regard des comportements génériques des composants logiciels des applications embarquées identifiés par [Stewart 1997] et plus spécifiquement des applications spatiales.

Dans un deuxième temps, une meilleure adéquation de l'environnement TURTLE avec le paradigme objet ouvrirait la voie à la génération automatique de code et à la validation formelle depuis les même diagrammes UML. En effet, la modélisation d'un comportement abstrait dans les *Tclass* rend impossible la génération de code telle qu'elle est pratiquée dans les ateliers logiciels UML

Enfin, nous avons étudié avec notre cadre formel la problématique de la modélisation d'une architecture à composants d'une application centralisée. Mais de nombreuses applications distribuées pourraient bénéficier de la validation formelle de leur mise à jour. Une telle validation suppose la modélisation des contraintes relatives à la distribution (caractéristiques de communication entre les sites d'exécution, etc.). Les réseaux actifs pourraient alors bénéficier de techniques de validation a priori du bon comportement global lors de l'intégration de code utilisateur dans les équipements réseaux.

En termes de validation, deux améliorations peuvent être apportées. Tout d'abord, l'explosion combinatoire pourrait être mieux maîtrisée par la mise en place de techniques innovantes de traduction de TURTLE à RT-LOTOS à l'exemple des techniques développées pour d'autres cadres formels [Castanet 1998][Ribet 2002]. Ensuite, une meilleure intégration du processus de validation formelle au sein d'un outil de modélisation UML faciliterait la remontée au modèle depuis le graphe d'accessibilité.

Enfin, d'autres domaines d'applications pourraient bénéficier d'un tel cadre formel basé sur UML, notamment le domaine des systèmes distribués, des protocoles et des documents multimédias, domaines pour lesquels les techniques de modélisation et de validation ont été déjà largement utilisées [Sénac 1996] [Toussaint 1997]. Mais chaque domaine d'application ayant ses spécificités, il serait souhaitable de proposer de nouveaux opérateurs TURTLE de haut niveau à l'instar de ce que nous avons fait pour les applications temps-réel.

Mécanismes supports

Trois améliorations sont envisageables au niveau des mécanismes supports. Tout d'abord, le degré de reconfigurabilité de notre environnement est supérieur à celui des autres environnements à base de composants logiciels grâce à la définition d'opérations de reconfiguration intra-composants. Mais ces opérations sont assez restrictives : elles ne peuvent s'appliquer qu'à des cas bien particuliers (cf. 5.3.3.3). Il pourrait ainsi être intéressant de proposer des opérations intra-composant plus générales.

De plus, notre environnement n'intègre pas d'opération de changement de géométrie d'architecture (changement de site d'exécution) contrairement par exemple à celui proposé dans [Purtilo 1994].

Ensuite, la méthodologie de développement logiciel ne fournit pas d'idée précise quant à l'identification des contraintes intrinsèques et extrinsèques. Nous suggérons à ce sujet une traçabilité des exigences effectuée conjointement avec l'identification et la modélisation des contraintes applicatives.

Enfin, certains mécanismes supports à la reconfiguration dynamique gagneraient à être intégrés à la machine virtuelle Java : protocoles de téléchargement de code et vérification de disponibilité de ressource étendue.

Évaluation de l'approche pour d'autres types ou domaines d'application

Nous avons évalué notre approche de reconfiguration dynamique sur un logiciel charge-utile de télécommunications. D'autres applications du domaine du spatial pourraient bénéficier de cette approche. Tout d'abord, nous avons vu dans le chapitre 6 que le système satellite considéré comprend des entités logicielles sols (utilisateurs et centre de contrôle du réseau) et bords. Si ce mémoire ne traite que de la mise à jour des entités bords, il pourrait être intéressant d'étendre nos travaux afin de proposer une reconfiguration dynamique synchronisée entre toutes les entités logicielles d'un système satellite (constellations de satellite par exemple). Ensuite, l'introduction dans le spatial de composants matériels reconfigurables (type FPGA, *Field Programmable Gate Array*) notamment à des fins d'implantation de fonctions de télécommunication [Kenington 1997] amène deux nouvelles problématiques : la définition d'un support logiciel à la mise à jour de ces composants matériels et la reconfiguration synchronisée entre les fonctionnalités matérielles et logicielles. Enfin, les études sur l'autonomie des engins spatiaux [Séguela 2001] soulèvent le problème de la reconfiguration automatisée et autonome du logiciel à des fins de tolérance aux fautes.

Utilisation de la plate-forme NINE à d'autres fins

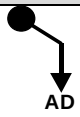
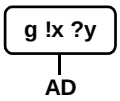
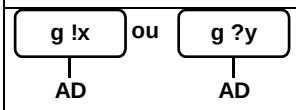
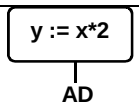
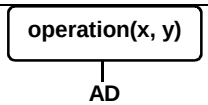
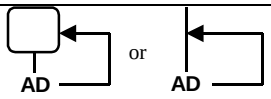
La plate-forme d'émulation NINE a déjà été expérimentée avec succès pour d'autres travaux [Rojas 2000][Sénac 2001]. Il serait à présent souhaitable d'améliorer le processus de routage afin d'autoriser (1) la réalisation de traitements particuliers pour chaque paquet et (2) la réalisation de scénarios dynamiques issus de traces réelles combinant ainsi les avantages du faible coût de l'émulation à celui de la représentativité de la mesure directe. La plate-forme pourrait aussi bénéficier d'un outil qui, depuis une description de la vue système (cf. 6.4.1.3) déduirait automatiquement la configuration de la plate-forme en termes d'interconnexion de stations de travail (création par exemple de VLAN) et en termes de configuration du routeur (règles de traitement appliquées aux paquets).

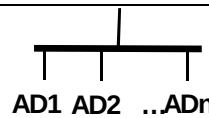
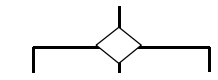
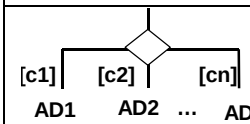
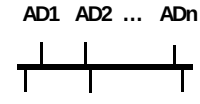
Annexe A

Opérateurs TURTLE des diagrammes d'activité

Le Tableau 8.1 illustre la sémantique associée par transposition en LOTOS aux constructions des diagrammes d'activités [Douglas 1999]. Soit AD la dénomination pour un diagramme d'activités, et **τ**(AD) le processus RT-LOTOS correspondant à la transcription de AD. Par défaut, les portes utilisées à titre d'exemple sont des portes de synchronisation.

Tableau 8.1.Opérateurs non temporels de TURTLE.

Diagramme d'activités TURTLE	***Description***	***Traduction LOTOS***
	*Début d'un diagramme d'activités. En d'autres termes, début de la traduction.*	*τ(AD)*
	*Appel sur la* Gate *g avec émission de la valeur x et réception de la valeur y. AD est ensuite interprété.*	*si g porte de synchronisation :* *g !x ?y :nat ; τ(AD)* *si g porte d'invocation :* *g !x ; g ?y :nat ; τ(AD)*
	*Appel sur la* Gate *g (g, porte invoquée) avec émission de la valeur x ou réception de la valeur y. AD est ensuite interprété.*	*g !x ; τ(AD) ou g ?y :nat ; τ(AD)*
	*Assignation d'une valeur à un attribut. AD est ensuite interprété.*	*Let y : Ytype = x*2 in* *τ(AD)*
	*Appel de méthode. AD est ensuite interprété.*	*τ(AD)* *(appel de méthode non traduit)*
	*Structure de type boucle. AD est interprété à chaque fois que l'interpréteur entre dans la boucle.*	*Process LabelX[g1,…gn]* *: noexit := τ(AD)* *>>LabelX[g1,…gn]* *end proc ;*
	Parallélisme entre n sous-activités décrites par AD1, AD2, …, Adn. A des fins de documentation, les sous-	*τ(AD1) ||| τ(AD2) |||* *… ||| τ(Adn)*

	*activités peuvent être séparées par des « swinlanes ».*	*(« swinlanes » non traduites)*
	*Synchronisation sur les portes g1, g2, …, gm entre n sous-activités décrites par AD1, AD2, …, Adn.*	*τ(AD1) |[g1,…gm]| τ(AD2) |[g1,…gm]| … |[g1,…gm]| τ(Adn)*
	*Sélection de la première activité prête à s'exécuter parmi les n activités décrites par AD1, AD2, … , Adn.*	*τ(AD1) [] τ(AD2) [] … [] τ(Adn)*
	*Sélection de la première activité prête à s'exécuter et dont la garde est vraie, parmi les n activités décrites par AD1, AD2, … , Adn.*	*[c1] -> τ(AD1) [] [c2] -> τ(AD2) [] … [] [cn] -> τ(Adn)*
	*Une fois que toutes les activités AD1, AD2, …, Adn ont terminé leur exécution, les m activités décrites par AD'1, AD'2, …, AD'm sont exécutées en parallèle (entrelacement).*[5]	*(τ(AD1) ||| τ(AD2) ||| … τ(Adn)) >> (τ(AD'1) ||| τ(AD'2) ||| … τ(AD'm))*
	*Une fois que toutes les activités AD1, AD2, …, Adn ont terminé leur exécution, les m activités décrites par AD'1, AD'2, …, AD'm sont exécutées en synchronisation sur les k portes g1, g2, …, gk.*	*(τ(AD1) ||| τ(AD2) ||| … τ(Adn)) >> (τ(AD'1) |[g1, …gk]| τ(AD'2) |[g1, …gk]| … τ(AD'm))*
	*Terminaison d'une activité*	*exit*

[5] Si le cardinal des ADi vaut 1 et que le cardinal des AD'j vaut aussi 1, alors une simple flèche est utilisée entre AD1 et AD'1.

Annexe B

Définition des opérations de reconfiguration dynamique

Cette annexe présente une vue synthétique des opérations de reconfiguration dynamique.

Le Tableau 9.1 met en évidence la signature, l'inverse et les préconditions des opérations de reconfiguration. On suppose qu'une application A est définie comme suit : $A = M \cup L \cup C$ avec M l'ensemble des modules, L l'ensemble des liens ⟨PortSortie, PortEntree⟩.et C l'ensemble des classes. De plus, on a :

L = ⟨PortSortie, PortEntree⟩ et M = ⟨ensPortEntrée, ensPortSortie, état, ensObjets, modèleUML⟩, ensPortEntrée un ensemble de ports d'entrée, ensPortSortie un ensemble de ports de sortie, état ∈ {*chargé*, *créé*, *exécutable*, *suspendu*, *détruit*}, ensObjets un ensemble d'objets, modèleUML, et une représentation analytique du modèle UML du module, à savoir un modèle analytique des classes, relations entre classes, des objets et des relations entre les objets. On note m_i.ensPortEntrée l'ensemble des ports d'entrée du module m_i (la notation est étendue à tout élément des modules ou des liens).

Tableau 9.1.Classification des opérations : signature, opération inverse et précondition.

Signature de o	Opération inverse de o	Précondition
Opération d'architecture		
créerLien (m1, ps, m2, pe)	détruireLien (m1, ps)	*Pré*(créerLien(m1, ps, m2, pe)) = m1 ∈ M ∧ m2 ∈ M ∧ ps ∈ m1.ensPortsortie ∧ pe ∈ m2.ensPortEntrée ∧ ∀ pe' port d'entrée de l'application {L∈ L / L = ⟨ps, pe'⟩} = ∅ ∧ ∀ ps' port de sortie de l'application {L∈ L / L = ⟨ps', pe⟩} = ∅ et compatible(ps.type, pe.type) = vrai.
détruireLien(m, ps)	créerLien(m1, ps, m2, pe) avec m1 = m ∧ pe ∈ m2.ensPortEntrée ∧ avant application de détruireLien, on a ⟨ps, pe⟩ ∈ L.	*Pré*(détruireLien(m, ps)) = m ∈ M ∧ ps ∈ m.ensPortsortie ∧ ∀ pe port d'entrée de l'application card {L ∈ L / L = ⟨ps, pe⟩} = 1.
ajouterModule(m)	détruireModule(m)	*Pré*(ajouterModule(m))) = ∃ place mémoire pour classes du module c et pour les objets de m.
détruireModule(m)	∅	*Pré*(détruireModule(m)) = m ∈ M.
Opérations internes aux modules		
ajouterPortEntrée(pe, m)	supprimerPort(pe, m)	*Pré*(ajouterPortEntrée(pe, m) = ∀ m ∈ M, pe ∉ m.ensPortEntrée.
ajouterPortSortie(ps, m)	supprimerPort(pe, m)	*Pré*(ajouterPortsortie(ps, m) = ∀ m ∈ M, ps

		∉ m.ensPortSortie.
supprimerPort(p, m)	∅ si p = port d'entrée ajouterPortSortie(p, m) si p est un port de sortie.	*Pré*(supprimerPortsortie(p, m)) = (p ∈ m.ensPortEntree ∨ p ∈ m.ensPortsortie) ∧ {L = ⟨ps, pe⟩ /L∈ *L* ∧ (p = ps ∨ p = pe) } = ∅.
remplacerModule(m1, m2)	∅	*Pré*(remplacerModule(m1, m2)) = ∃ place mémoire pour classes du module c et pour les objets de m2.
modifierObjet(o, c, m)	∅	*Pré*(modifierObjet(o, c, m)) = ∃ place mémoire pour la classe c et une instance de c ∧ o ∈ m ∧ ∀ o' ∈ m, o' possède un attribut a pointant vers o ⇒ c hérite de classe(a).
modifierClasse(c1, c2, m)	∅	*Pré*(modifierClasse(c1, c2, m)) = ∃ place mémoire pour la classe c2 ∧ si n = card{o_i / classe(o_i) = c1 ∧ o_i ∈ m} alors ∃ place mémoire n instances de c2 ∧ ∀ o' ∈ m, o' possède un attribut a pointant vers o_i ⇒ c hérite de classe(a).
Opérations supports		
chargerClasse(c)	détruireClasse(c)	*Pré*(chargerClasse(c)) = ∃ place mémoire pour c.
détruireClasse(c)	chargerClasse(c)	*Pré*(détruireClasse(c)) = c ∈ Appli.ensClasses.
créerObjet(c)	Soit o un objet tel que classe(o) = c et si on note A'=exe(créerObjet(c), A) alors o est tel que o ∉ A ∧ o ∈A'. L'opération inverse est alors : détruireObjet(o)	*Pré*(créerObjet(c)) = c ∈ Appli.ensClasses ∧ ∃ place mémoire pour o.
détruireObjet(o)	∅	*Pré*(détruireObjet(o)) = ∃ m ∈ *M* / o ∈ m.ensObjets.
lireAttribut(a, o)	∅	*Pré*(lireAttribut(a, o)) = ∃ m ∈ *M* / o ∈ m.ensObjets ∧ a attribut de o.
écrireAttribut(valeur, a, o)	∅	*Pré*(écrireAttribut(valeur, a, o)) = ∃ m ∈ *M* / o ∈ m.ensObjets ∧ a attribut de o.
chargerModule(m)	∀ c ∈ m.modèleUML.diagclasses, ∀ m' ∈ *M*,, m' ≠ m ∧ c ∉ m.modèleUML.diagclasses ⇒ détruireClasse(c)	Soit C = {c classes / c ∈ m.modèleUML.diagclasses ∧ m ∉ Appli.ensClasses}. *Pré*(chargreModule(m)) = ∃ place mémoire pour classes de C.
instancierModule(m)	détruireObjet(o) avec o / o ∈ m.ensObjets ∧ classe(o) hérite de Module	*Pré*(instancierModule(m)) = m ∈ *M* ∧ m.état = chargé ∧ ∃ place mémoire pour instancier les objets de m.
démarrer(m)	suspendre(m)	*Pré*(démarrer(m)) = m ∈ *M* ∧ m.état = créé ∧ ∀o ∈ m.modèleUML, o ∈ m.ensObjets.
suspendre(m)	activer(m)	*Pré*(suspendre(m)) = m ∈ *M* ∧ m.état = actif.
Activer(m)	suspendre(m)	*Pré*(activer(m)) = m ∈ *M* ∧ m.état = suspendu.
encoderModule(m)	∅	*Pré*(encoderModule(m)) = m ∈ *M* ∧ m.état = suspendu.
positionnerEtat(état, m)	∅	*Pré*(positionnerEtat(m)) = m ∈ *M* ∧ (m.état = suspendu ∨ m.état = créé).
encoderObjet(o)	∅	Soit m ∈ *M* / o ∈ m.ensObjets. *Pré*(encoderObjet(o)) = (m.état = suspendu ∨ m.état = créé).
positionnerEtat(état, o)	∅	Soit m ∈ *M* / o ∈ m.ensObjets. *Pré*(positionnerEtat(m)) = (m.état = suspendu ∨ m.état = créé).

Acronymes

ADL, *Application Description Language.*
ATM, *Asynchronous Transfer Mode.*
BAC, *Block Admission Control.*
CAC, *Connection Admission Control.*
CBR, *Constant Bit Rate.*
CDN, *Content Delivery Network.*
CORBA, *Common Object Request Broker Architecture.*
CPU, *Central Processing Unit.*
CSP, *Communicating Sequential Processes.*
CTD, *Cell transfer Delay.*
CURL, *Corot Update Reconfiguration Language.*
DAMA, *Demand Assignment Multiple Access.*
DSP, *Digital Signal Processing.*
DVB, *Digital Video Broadcast.*
EEPROM, *Electrically Erasable Programmable Read-Only Memory.*
FPGA, *Field Programmable Gate Array.*
FTP, *File Transfer Protocol.*
HTTP, *Hyper Text Transfer Protocol.*
IETF, *Internet engineering Task Force.*
IP, *Internet Protocol.*
IVT, *Intervalle de Validité Temporelle.*
JVM, *Java Virtual Machine.*
LOTOS, *Language Of Temporal Ordering Specification.*
MF-TDMA, *Multi-Frequency Time Division Multiple Access.*
MIL, *Module Interconnection Language.*
MRD, *Middleware de Reconfiguration Dynamique.*
MRD.e, *Middleware de Reconfiguration Dynamique embarqué.*
MRD.o, *Middleware de Reconfiguration Dynamique opérateur.*
MRD.sc, *Middleware de Reconfiguration Dynamique du serveur de classes.*
NGS, *New Generation Satellite.*
NINE, *Nine Is a Network Emulator.*
OBMC, *On Board Multimedia Controler.*
OBMP, *On Board Multimedia Processing.*
OBP, *On-board Processing.*
OCL, *Object Constraint Language.*
OMG, *Object Management Group.*
QoS, *Quality of Service.*
RAM, *Random Access Memory.*
RPC, *Remote Procedure Call.*
RdP, *Réseau de Petri.*
RdPFT, *Réseau de Petri à Flux Temporel.*
RdPFT-RD, *Réseau de Petri à Flux Temporel de Reconfiguration Dynamique.*
RT-LOTOS, *Real-Time LOTOS.*
SAGAM, *SAtellite Géostationnaire pour Accès Multimédia.*
SRD, *Spécification de Reconfiguration Dynamique.*
TCP, *Transport Control Protocol.*
TDMA, *Time Division Multiple Access.*
TM/TC, *Telemetry / Telecommand.*
TURTLE, *Timed UML and RT-Lotos Environment.*
UES, *User Earth Station.*
UBR, *Unspecified Bit Rate.*
UML, *Unified Modeling Language.*
VBR, *Variable Bit Rate.*
VLAN, *Virtual Local Area Network.*
VME, *Versa Modular Eurocard (IEEE 1014).*
VoIP, *Voice over IP.*
WFQ, *Weight Fair Queuing.*
XMI, *XML Metadata Interchange.*

XML, *Extensible Markup Language*.

Bibliographie de l'auteur

Revue internationale

DE SAQUI-SANNES P., APVRILLE L., LOHR C., SÉNAC P., COURTIAT J.-P., « UML and RT-LOTOS : An Integration for Real-Time System Validation », *European Journal of Automation (JESA)*, Ed. Hermès, 2002 (to appear).

Conférences avec comité de lecture et actes

APVRILLE L., DE SAQUI-SANNES P., SENAC P., LOHR C., « Reconfiguration dynamique de protocoles embarqués à bord de satellites », *Actes du Colloque Francophone sur l'Ingéniérie des Protocoles* (*CFIP'2002*), Montréal, 27-30 mai 2002.

APVRILLE L., DAIRAINE L., SÉNAC P., DIAZ M., «Satellite Telecom Software Dynamic Upgrade with QoS Continuity », *Proceedings of the 20th International AIAA Communications Satellite Systems Conference* (*ICSSC'2002*), May 12 – 15, Montréal, Canada, 2002.

DE SAQUI-SANNES P., APVRILLE L., LOHR C., SENAC P., COURTIAT J.-P., « UML et RT-LOTOS : Vers une intégration informel/formel au service de la validation de systèmes temps réel », *Actes du colloque francophone sur la Modélisation des Systèmes Réactifs* (*MSR'2001*), Toulouse, France, Octobre 2001.

APVRILLE L., DE SAQUI-SANNES P., LOHR C., SÉNAC P., COURTIAT J.-P., « A New UML Profile for Real-time System Formal Design and Validation », *Proceedings of the Fourth International Conference on the Unified Modeling Language* (*UML'2001*), Toronto, Canada, October 2001.

APVRILLE L., DE SAQUI-SANNES P., SÉNAC P., DIAZ M., « Formal Modeling of Space-Based Software in the Context of Dynamic Reconfiguration », *Proceedings of DAta Systems In Aerospace* (*DASIA*'2001), May 28 – June 1st, Nice, France 2001.

APVRILLE L., DAIRAINE L., SÉNAC P., DIAZ M., « Dynamic Reconfiguration Architecture of Satellite Network Software Services », *Proceedings of the 19th International AIAA Communications Satellite Systems Conference* (*ICSSC'2001*), Toulouse, France, April 17-20, 2001.

APVRILLE L., DAIRAINE L., ROJAS-CARDENAS L., SÉNAC P., DIAZ M., « Implementing a User Level Multimedia Transport Protocol in Java », *The Fifth IEEE International Symposium on Computers and Communication* (*ISCC'2000*), Antibes-Juan les Pins, France, July 4-6, 2000.

APVRILLE L., SÉNAC P., RICHARD F., DIAZ M., « Java Face to Small Satellites », The 5th International Symposium on Small Satellite systems and Services, La Baule, France, June 19-23, 2000.

APVRILLE L., SÉNAC P., DIAZ M., « A Java Framework for Spatial Embedded Systems », *Proceedings of the 13th Annual AIAA/USU International Conference on Small Satellites*, Logan, Utah, USA, August 23-26, 1999.

Rapports internes et rapport de contrat

APVRILLE L., « Reconfiguration dynamique de modules logiciels au sein d'un système à commutation embarquée : définition de l'architecture logicielle embarquée », *rapport contractuel ENSICA/Alcatel Space*, août 2001.

APVRILLE L., « Reconfiguration dynamique de modules logiciels au sein d'un système à commutation embarquée : modélisation et analyse des performances », *rapport contractuel ENSICA/Alcatel Space*, avril 2001.

APVRILLE L., « Reconfiguration dynamique de modules logiciels au sein d'un système à commutation embarquée : modélisation et analyse des performances », *rapport contractuel ENSICA/Alcatel Space*, octobre 2000.

APVRILLE L., « Reconfiguration dynamique de modules logiciels au sein d'un système à commutation embarquée : définition et justification des mécanismes logiciels », *rapport contractuel ENSICA/Alcatel Space*, avril 2000.

APVRILLE L., « Les technologies Objet et la distribution d'applications logicielles sur les constellations de satellites », *rapport contractuel ENSICA/Alcatel Space*, avril 2000.

APVRILLE L., « Evaluation du langage Java pour les applications spatiales embarquées », *Mémoire de DEA*, Institut National Polytechnique de Toulouse, Septembre 1998.

Bibliographie

[Allen 1997] ALLEN R., « A formal Approach to Software Architecture », *Ph.D. Thesis*, Carnegie Mellon University, School of computer Science, available as TR#CMU-CS-97-144, May 1997.

[Allen 1998] ALLEN R., DOUENCE R., GARLAN D., « Specifying and analyzing Dynamic Software Architectures », *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, Lisbon, Portugal, March 1998.

[Allman 1999] ALLMAN M., GLOVER D., SANCHEZ L., « Enhancing TCP Over Satellite Channels using Standard Mechanisms », *IETF RFC 2488*, January 1999.

[Andre 2001] ANDRE C., « Paradigmes objets et synchrones dans les systèmes temps-réel », *journée Objets Temps Réel du Club SEE (Systèmes Informatiques de Confiance)*, Paris, 18 janvier 2001. http://www.cert.fr/francais/deri/seguin/SEE/01.01.18/annonce.html.

[Apvrille 1998] APVRILLE L., « Evaluation du langage Java pour les applications spatiales embarquées », *Mémoire de DEA*, Institut National Polytechnique de Toulouse, Septembre 1998.

[Apvrille 1999] APVRILLE L., SÉNAC P., DIAZ M., « A Java Framework for Spatial Embedded Systems », *Proceedings of the 13th Annual AIAA/USU Conference on Small Satellites*, Logan, Utah, USA, August 23-26, 1999.

[Apvrille 2000] APVRILLE L, DAIRAINE L., ROJAS-CARDENAS L., SÉNAC P., DIAZ M., « Implementing a User Level Multimedia Transport Protocol in Java », *Proceedings of the Fifth IEEE Symposium on Computers and Communication (ISCC'2000)*, Antibes-Juan les Pins, France, July 4-6, 2000.

[Apvrille 2001a] APVRILLE L., DAIRAINE L., SÉNAC P., DIAZ M., « Dynamic Reconfiguration Architecture of Satellite Network software Services », *Proceedings of the 19th International Communications Satellite Systems Conference*, vol. 2, Toulouse, France, 17-20 April 2001.

[Apvrille 2001b] APVRILLE L., DE SAQUI-SANNES P., LOHR C., SÉNAC P., COURTIAT J.-P., « A New UML Profile for Real-time System : Formal Design and Validation », *Proceedings of the Fourth International Conference on the Unified Modeling Language (UML'2001)*, Toronto, Canada, October 1 – 5, 2001.

[Apvrille 2001c] APVRILLE L., « Reconfiguration dynamique de modules logiciels au sein d'un système à commutation embarquée : définition de l'architecture logicielle embarquée », *rapport contractuel ENSICA/Alcatel Space*, août 2001.

[Artisan 1999] ARTISAN SOFTWARE TOOLS,: http://www.artisan-software.com, 1999.

[Assis 1997] ASSIS ROSA F., SILVA A.R., « Component Configurer: A Design Pattern for Component-Based Configuration », *Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97)*, Siemens Technical Report 120/SW1/FB. Munich, Germany, 1997.

[Balakrishnan 1996] BALAKRISHNAN H., PADMANABHAN V. N., SESHAN S., KATZ R., « A comparison of Mechanisms for Improving TCP Performance over Wireless Links », *ACM SIGCOMM*, Août 1996.

[Bem 2000] BEM D., ET AL., « Broadband Satellite systems », *IEEE Communications Surveys and Tutorials*, vol. 3, n° 1, First Quater 2000.

[Bigo 2000] BIGO S., IDLER W., « Des térabits par seconde sur une fibre Teralight d'Alcatel », *Revue des télécommunications d'Alcatel*, p. 288-296, 4ème trimestre 2000.

[Bjorkander 2000] BJORKANDER M., « Real-Time Systems in UML and SDL », *Embedded System Engineering*, October/November 2000 (http://www.telelogic.com).

[Blineau 2001] BLINEAU J., CASTELLANET M., CHEVAL P., VERHULST D., « La contribution des satellites à l'internet », *Revue des Télécommunicatiosn d'Alcatel*, p. 243-248, 4ème trimestre 2001.

[Bolognesi 1987] BOLOGNESI T., BRINKSMA E., « Introduction to the ISO specification Language LOTOS », *Computer Networks and ISDN Systems*, vol. 14, n° 1, 1987.

[Booch 1999] BOOCH, G., RUMBAUGH, J. & JACOBSON, I., « The Unified Modeling Language User Guide », *Addison-Wesley*, Reading MA, USA, 1999.

[Bossu 1999] BOSSU Y., « Java embarqué », *EYROLLES – 02/1999*, ISBN: 2-212-09053-6.

[Boute 1981] BOUTE, R.T., DE MAN, J., PEETERS, H., « Secure on-the-fly Software Modification », *Proceedings of the Fourth International Conference on Software Engineering for Telecommunication Switching systems*, p. 49-53, 1981.

[Boutry 2000] BOUTRY L., « L'évolution des réseaux », *Memento technique du CNET*, n° 15, http://www.cent.fr/sas/mento15/chap5.html, mars 2000.

[Bruel 1998] BRUEL, J.-M. FRANCE R.B., « Transforming UML Models to Formal Specifications », *Proceedings of the International Conference on Object Oriented Programming Systems Language and Applications (OOPSLA'98)*, Vancouver, Canada, 1998.

[Bruel 1999] BRUEL J.-M., « Integrating Formal and Informal Specification Techniques. Why? How? », *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, Boca Raton, Florida, USA, IEEE Computer Press, p. 50-57, 1999.

[Bunnel 1998] BUNNELL M., « Mixing Java and C in Embedded Systems », *Real-time Magazine*, Internet Embedded – 98/1

[Cailliau 2001] CAILLIAU D., « Maintenance du logiciel de vol de véhicules spatiaux scientifiques par le biais de la reconfiguration dynamique », *thèse de doctorat de l'université Paris VI*, octobre 2001.

[Calvert 1998] CALVERT K., BHATTACHARJEE S., ZEGURA E., ET STERBENZ J., « Directions in Active Networks », *IEEE Communications Magazine*, 1998.

[Castanet 1998] CASTANET R., KONE O., LAURENCOT L., « On the Fly Test Generation for real-time protocols », *Proceedings of the 7th International Conference on Computer Communications and Networks*, p. 378-385, Lafayette, USA, October 12-15, 1998.

[Chen 2000] CHEN T.M., « Evolution to the Programmable Internet », *IEEE Communications Magazine*, vol. 38, issue 3, p. 124 –128, March 2000.

[Cheung 1990] CHEUNG T.Y., YE Y.C., YE X., WANG G.Q., « UO-GLOTOS: A Syntax/System for Representing, Editing and Translating Graphical LOTOS », *Formal Description Techniques*, II, Elsevier Science Publishers B.V., ed. S.T. Vuong, 1990.

[Clark 2000] CLARK, R.G., MOREIRA A.M.D., « Use of E-LOTOS in Adding Formality to UML », *Journal of Universal Computer Science*, vol. 6, n° 11, p. 1071-1087, 2000.

[Clarke 1945] CLARKE C., « Extra-Terrestrial Relays: Can Rocket Stations Give World-wide Radio Coverage », *Wireless World*, , pages 305-308, October 1945.

[Clohessy 2001] CLOHESSY K., « Development Tools for Embedding Java », *Dedicated Systems Magazine*, p. 80-83, January 2001.

[Combes 2001] COMBES S., FOUQUET C., RENAT V., « Packet-Based DAMA Protocols for New Generation Satellite Networks », *Proceedings of the 19th AIAA International Communications Satellite Systems Conference*, Toulouse, France, 17-21 Avril 2001.

[Courtiat 2000] COURTIAT J.-P., SANTOS C.A.S., LOHR C., OUTTAJ B., « Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique », *Computer Communications*, Vol. 23, No. 12, p. 1104-1123, 2000.

[Debus 2001] DEBUS V., « Requirements Management : Practical Approach », *Proceedings of Data Systems In Aerospace (DASIA)*, Nice, France, 28 May – 1st June, 2001.

[Delatour 1998] DELATOUR J., PALUDETTO M., « UML/PNO, a way to merge UML and Petri net objects for the analysis of real-time systems », *OO Technology and Real Time Systems Workshop (ECOOP'98)*, Bruxelles, Belgium, 1998.

[Diaz 1993] DIAZ M., SÉNAC P., « Time Stream Petri Nets: a Model for Multimedia Streams synchronization », *1st International Conference on Multimedia Modeling (MMM'93)*, Singapore, p. 257-273, novembre 1993.

[Diaz 1994] DIAZ M., LOZES A., CHASSOT C., AMER P., « Partial Order Connections: a new Concept for High Speed and Multimedia Services and Protocols », *Annals of telecommunication*, vol. 49, n°5-6, 1994.

[Douglas 1999] DOUGLASS B.P., « Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns », *Addison-Wesley Longman*, 1999 (http://www.ilogix.com).

[Douglas 2001] DOUGLAS B.P., « ROPES : Rapid Object-Oriented Process for Embedded Systems », http://www.ilogix.com/quick_links/white_papers/, 2001.

[Dupuy 2000] DUPUY S., LEDRU Y., CHABRE-PECCOUD M., « Vers une intégration utile de notations semi-formelles et formelles : une expérience en UML et Z », vol.6, n°1, p. 9-32, Hermès, Paris, 2000.

[Dupuy 2001] DUPUY S., DU BOUQUET L., « A Multi-formalism Approach for the Validation of UML Models », *Formal Aspects of Computing*, No.12, p. 228-230, 2001.

[Duros 2001] DUROS E., DABBOUS W., IZUMIYAMA H., ZHANG Y., « A Link-Layer Tunneling Mechanism for Unidirectional Links », *Internet Request for Comments*, 3077, Mars 2001.

[Durst 1996] DURST R., MILLER G., and Travis E., « TCP Extensions for Space Communications », *ACM MOBICOM*, Novembre 1996.

[Engram 1983] ENGRAM, R.L., SHANNON P.A., WEBER S.S., « Transparent Software and Hardware Changes in a Telecommunication System », *Proceedings of the Fifth International Conference on Software Engineering for Telecommunication Switching Systems*, p. 157-162, 1983.

[ERC32 1999a] ERC32 Home page at ESTEC, http://www.estec.esa.nl/wsmwww/erc32/erc32.html

[ERC32 1999b] Free software for ERC-32, at ESTEC: http://www.estec.esa.nl/wsmwww/erc32/freesoft.html

[ESA 2001] ESA, « Les opérations de sauvetage d'Artémis », communiqué de presse n°44-2001, Agence Spatiale Européenne, 2001.

[Evans 1999] EVANS A.S., COOK S., MELLOR S., WARMER J., WILLS A., « Advanced Methods and Tools for a Precise UML », *Proceedings of the 2nd International Conference on the Unified Modeling Language, UML'99*, Colorado, USA, LNCS 1723, 1999.

[Farserotu 2000] FARSEROTU J., PRADAS R., « A Survey of Future Broadband Multimedia Satellite Systems, Issues and Trends », *IEEE Communications Magazine*, p. 128-133, June 2000.

[Folliot 2000] FOLLIOT B, CAILLIAU D., PIUMARTA I., BELLENGER R., « PLERS : Plateforme Logiciel Embarqué Reconfigurable pour Satellite – Application au Satellite Corot », *Actes de RenPar'2000*, Besançon, 19 juin 2000.

[Franz 1997] FRANZ M., « Dynamic Linking of software components », *IEEE Computer*, Volume: 30 Issue: 3, Page(s): 74 –81, March 1997.

[Frieder 1989] FRIEDER O., HERMAN G.E., MANSFIELD W.H., JR. SEGAL, M.E., « Dynamic program modification in telecommunications systems », *Proceedings of the Seventh International Conference on Software Engineering for Telecommunication Switching Systems SETSS 89*, p. 168 –172, 1989.

[Fuggetta 1998] FUGGETTA A.; PICCO G.P.; VIGNA G., « Understanding Code Mobility », *IEEE Transactions on Software Engineering*, Vol. 24 Issue 5, p. 342 –361, May 1998.

[Garrido 1998] GARRIDO B. ET AL., « MINISAT01 On-board Software Maintenance », *Proceedings of Data Systems In Aerospace (DASIA'98)*, p. 65-69, Athens, Greece, 25-28 May, 1998.

[Goldberg 1983] GOLDBERG A., ROBSON D., « Smalltalk-80: The Language and Its Implementation », *Addison-Wesley*, ISBN 0-201-11371-6, 1983.

[Guennec 2000] LE GUENNEC A., « Méthodes formelles avec UML : modélisation, validation et génération de tests », *Actes du 8è Colloque Francophone sur l'Ingénierie des Protocoles CFIP'2000*, Toulouse, Editions Hermès, Paris, p. 151-166, 17-20 octobre 2000.

[Gupta 1996] GUPTA D., JALOTE P., BARUA G., « A Formal Framework or On-line Software Version Change », *IEEE Transactions on Software Engineering*, vol. 22, No 2, p. 120-131, Feb 1996.

[Hartman 1996] HARTMAN J., MANBER U., « Liquid Software : A new Paradigm for Networked Systems », *Technical Report 96-11*, Department of Computer Science, University of Arizona, June 1996.

[Hjalmtysson 1998] HJÁLMTÝSSON G., GRAY R., « Dynamic C++ classes – A Lightweight mechanism to update code in a running program », *Proceedings of the USENIX Annual Technical Conference*, p. 65-76, June, 1998.

[Hoare 1985] HOARE C.A.R., « Communicating Sequential Processes », *Prentice Hall*, 1985.

[Hofmeister 1993] HOFMEISTER C., PURTILO J., « Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for Replacement », *Proceedings of the 13th International Conference in Distributed Computing systems (ICDCS'93)*, IEEE Computer Society Press, p. 101-110, 1993.

[HotSpot 1999] Sun Microsystems Inc., HotSpot: The Java HotSpot virtual machine architecture, http://java.sun.com/products/hotspot/whitepaper.html, April 1999.

[Ismail 1998] ISMAIL 1998, HAGIMONT D., « Spécialisation de serveurs par des agents mobiles », *Actes de NOTERE'98*, Montréal, Canada, Octobre 1998.

[Jard 1988] JARD C., MONIN J.-F., GROZ R., « Development of Véda, A Prototyping Tool for Distributed Algorithms », *IEEE Transactions on Software Engineering*, vol. 14, No .3, March 1988.

[Jard 1998] JARD C., JEZEQUEL J.-M., PENNANEAC'H F., « Vers l'utilisation d'outils de validation de protocoles dans UML », *Technique et Science Informatique*, vol. 17, n°9, p. 1083-1098, Hermès, Paris, 1998.

[Jensen 1997] JENSEN K., « Coloured Petri Nets: Basic Concepts, Analysis Methods and Pratical Use », Volume 1, Springer 1997.

[Kenington 1997] KENNINGTON P.B., ET AL., « The Application of Software Radios to Integrated Satellite/Terrestrial Communications », *Integration of Satellite and Terrestrial OCS (Digest No: 1997/152)*, IEE Colloquium on, 1997.

[Keramane 1998] KERAMANE C., KARMOUCH A., « Une Architecture à agents pour l'inter-opérabilité de sites distribués », *Actes de NOTERE'98*, Montréal, Québec, Canada, Octobre 1998.

[Kramer 1985] KRAMER, J., MAGEE. J., « Dynamic Configuration for Distributed Systems », *IEEE Transactions on Software Engineering*, Vol. SE-11 No.4, p. 424-436, April 1985.

[Liang 1998] LIANG S., BRACHA G., « Dynamic Class Loading in the Java Virtual Machine », *Object-Oriented Programming systems, Languages and applications Conference*, in special Issue of SIGPLAN notice, C chambers, number 10, Vancouver, October 1998.

[Liskov 1983] LISKOV, B., SCHEIFLER, R., « Guardians and Actions: Linguistic Support for Robust, Distributed Programs », *ACM Transactions on Programming Languages and Systems*, volume 5, number 3, pages 381-404, July 1983.

[Liskov 1985] LISKOV B.H., « The Argus Language and System », *in Distributed Systems: Methods and Tools for Specifications*, pp. 343-430, Lecture Notes in Computer Science no. 190, Springer-Verlag, 1985.

[Little 1998] LITTLE M., WHEATER M., « Building configurable Applications in Java », *Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems*, May 1998.

[Lohr 2002] LOHR C., APVRILLE L., COURTIAT J.P., DE SAQUI-SANNES P., « Algorithmes de traduction des diagrammes TURTLE en RT-LOTOS », *Rapport technique LAAS*, en préparation, 2002.

[Magee 1994] MAGEE J., DULAY N. AND KRAMER J., « A Constructive Development for Parallel and Distributed Programs », *Proceedings of the international Workshop on Configurable Distributed Systems*, p. 4-14, Pittsburgh, March 1994.

[Malabarba 2000] MALABARBA S. ET AL., « Runtime Support for Dynamic Java Classes », *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, E. Bertino (Ed.), Sophia Antipolis and Cannes, France, June 2000.

[Martelli 1997] MARTELLI A., TORCHIA F., « The Software Maintenance in the BeppoSax Scientific Mission », *Proceedings of DAta Systems In Aerospace (DASIA)*, p.369-374, Sevilla, Spain, 26-29 May 1997..

[Medvidovic 2000] MEDVIDOVIC N., TAYLOR R., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, no. 1, January 2000.

[Neumann 2001] NEUMANN F., « VIPeR : une architecture logicielle flexible pour les routeurs », *Revue des télécommunications d'Alcatel*, p. 189-190, 3ème trimestre 2001.

[Nilsen 2001] NILSEN K., « Java Simplifies Embedded Programming, Reducing Errors », *Dedicated Systems Magazine*, p. 84-88, January 2001.

[NS 2002] The Network Simulator – ns-2, http://www.isi.edu/nsnam/ns

[Okamoto 1994] OKAMOTO A.; SUNAGA H., KOYANAGI K., « Dynamic program modification in the non-stop software extensible system (NOSES) », *IEEE International Conference on Communications (ICC '94)*, Conference Record, 'Serving Humanity Through Communications.', p. 1779 –1783 vol.3, 1994.

[OMG 1996] OMG (Object Management Group), « The Common Object Request Broker : architecture and specification, Revision 2.0 », *Document 97-02-25*, July 1996.

[OMG 2001] OMG, « Unified Modeling Language Specification », Version 1.4, Object Management Group http://www.omg.org/technology/documents/formal/uml.htm

[OMG 2002a] OMG, « XML Metadata Interchange (XMI) Specification », Version 1.2, January, 1st, 2002. http://www.omg.org/technology/documents/formal/xmi.htm

[OMG 2002b] OMG, « UML Profile for Schedulability, Performance and Time », January 28, 2002. http://www.omg.org/techprocess/meetings/schedule/UML_Profile_for_Scheduling_FTF

[OPNET 2002] OPNET Modeler, http://www.mil3.com/products/modeler/home.html

[Oreizy 1998] OREIZY P., MEDVIDOVIC N., TAYLOR R. N., « Architecture-Based Runtime Software Evolution », *Proceedings ot the International conference on software Engineering (ICSE'1998)*, Kyoto, Japan, April 19-25, 1998.

[Owezarski 1996] OWEZARSKI P., « Conception et formalisation d'une application de visioconférence coopérative. Application et extension pour la téléformation », rapport LAAS n°96478, *thèse de doctorat*, décembre 1996.

[Palma 1999] DE PALMA N., BELLISSARD L., RIVEILL M., « Dynamic Reconfiguration of Agent-Based Applications », *Third European Research Seminar on Advances in Distributed Systems* (*ERSADS'99*), Madeira Island – Portugal, 23-28 April 1999.

[Parise 2001] PARISE D., MEROUR J.-M., SOULERES E., « Evolution de l'architecture et de la technologie des charges utiles », *Revue des Télécommunications d'Alcatel*, p. 259-263, 4ème trimestre 2001.

[Pasetti 1999] PASETTI T., PREE W., « The Component Software Challenge for Real-Time Systems », *Proceedings of the First International Workshop on Real-Time Mission-Critical Systems*, Scottsdale, Arizona, 30 November – 1 December 1999.

[Patel 2001] PATEL A., « Active Network Technology », *IEEE Potentials*, Volume: 20 Issue: 1, Page(s): 5 –10, Feb-March 2001.

[Pellegrini 1999] PELLEGRINI M.-C., « Reconfiguration d'applications réparties : application au bus logiciel CORBA », *mémoire de thèse*, Institut National Polytechnique de Grenoble, 1999.

[Prechelt 2000] PRECHELT, L., « An empirical comparison of seven programming languages », *IEEE Computer*, vol. 33 , issue 10, p. 23 –29, Oct. 2000.

[Purtilo 1991] PURTILO J. M., HOFMEISTER C. R., « Dynamic reconfiguration of distributed programs », *Proceedings of the 11th International Conference on Distributed Computing Systems*, p. 560—571, IEEE Computer Society Press, Arlington TX, May 1991.

[Purtilo 1994] PURTILO J. M., « The Polylith Software Bus », *ACM Transactions on Programming Languages and Systems*, vol. 16, n° 1, p. 151-174, January 1994.

[Remer 1976] DE REMER F., KRON H., « Programming-in-the-Large Versus Programming-in-the-Small »,

IEEE Transactions on Software Engineering, vol. SE-2, n° 2, June 1976.

[Rey 1986] REY R., « Engineering and operations in the Bell system (second edition) », AT&T Bell Laboratories, Murray Hill, NJ, 1986.

[Ribet 2002] RIBET P. O., VERNADAT F., BERTHOMIEU B., « Maîtrise de l'explosion combinatoire pour la vérification formelle », *Rapport LAAS n°02064*, Février 2002.

[Rizzo 2000] RIZZO L., « Dummynet Emulator », Universita di Pisa, http://www.iet.unipi.it/~luigi/ip_dummynet/.

[Rojas 2000] ROJAS-CARDENAS L., « Architecture de transport à ordre et fiabilité partiels pour les applications multimédias adaptatives à temps contraint », Rapport LAAS n°00531, *thèse de doctorat*, novembre 2000.

[Roullet 1999] ROULLET L., ET AL., « SAGAM demonstrator of a G.E.O. Satellite Multimedia Access System: Architecture & Integrated Resource Manager », *Proceedings of the European Conference on Satellite Communication*, Toulouse, France, November 3-5, 1999.

[RTJ 2001] The Real-Time for Java Expert Group, « the Real-Time Specification for Java », available as rtsj-V1.0.pdf at http://www.rtj.org, December 11, 2001.

[RTJWG 2000] The Real-Time Java™ Working Group, « Revision 1.0.14 of Real-Time Core Extensions », Sept. 2, 2000. Available at http://www.j-consortium.org.

[RUP 2001] Rational Unified Process, http ://www.rational.com/products/rup/, 2001.

[SAGAM 1998] « SAtellite Géostationnaire pour Accès Multimédia », *Projet RNRT* (Réseau National de la Recherche en Télécommunication), France, 1998. http://www.sagam-satellite.com.

[Saqui 2001] DE SAQUI-SANNES P., « Diagramming TURTLE classes using Rhapsody », ENSICA internal report (2001).

[SCPS 1997] CONSULTATIVE COMMITEE FOR SPACEDATA SYSTEMS, « Space Communications Protocol Specification – Transport Protocol (SCPS-TP) », CCSDS 714.0-R-3, Red Book, http://www.scps.org/scps/, September 1997.

[Ségal 1989] SEGAL M. E., FRIEDER O., « Dynamically Updating Distributed Software: Supporting Change in Uncertain and Mistrustful Environments », *Proceedings of the Conference on Software Maintenance*, p. 254 –261, 1989.

[Segal 1993] SEGAL M. E., FRIEDER O., « On-the-fly program modification: systems for dynamic updating », IEEE Software , Vol. 10, Issue 2 , p. 53 –65, March 1993.

[Séguéla 2001] SÉGUÉLA D., CHARMEAU M.C., BOSSARD F., « CNES studies on satellite autonomy », *Proceedings of Data Systems In Aerospace (DASIA)*, Nice, France, 28 May – 1st June, 2001.

[Selic 1998] SELIC B., RUMBAUGH J., « Using UML for Modeling Complex Real-Time Systems », http://www.rational.com, 1998.

[Sénac 1996] SÉNAC P., DIAZ M., DE SAQUI-SANNES P., LÉGER A., « Modeling Logical and Temporal Synchronization in Hypermedia Systems », *IEEE Journal on Selected Areas in Communication*, special issue on multimedia synchronization, 1996.

[Sénac 2001] SÉNAC P., EXPOSITO E., DIAZ M., « Towards a New Generation of Generic Transport Protocols », *IWDC 2001*: *International Workshop on Digital Communications*, Taormina, Italy, September 17-20, 2001.

[Shrivastava 1998] SHRIVASTAVA S. K., WHEATER S.M., « Architectural Support for Dynamic Reconfiguration of Distributed workflow Applications », *IEE Proc-Software*, Vol 145, n°5, October 1998.

[Stevens 2000] STEVENS J.S., JOHNSON G. L. R., « Updating the SOHO AOCS ACU On-board Software », *Proceedings of DAta Systems In Aerospace (DASIA)*, p. 347-352, Montreal, Canada, 26-28 May, 2000.

[Stewart 1996] STEWART D., ARORA G., « Dynamically Reconfigurable Embedded Software – Does It make Sense », *Proceedings of the International IEEE Conference on Engineering of Complex Computer Systems (ICECCS) and Real-Time Applications Workshop (RTAW)*, Montreal, Canada, p. 217-220, Oct 21-

25, 1996.

[Stewart 1997] STEWART D., VOLPE R., KHOSLA P., « Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects », *IEEE Transactions on Software Engineering*, vol. 23, no. 12, December 1997.

[Stockenberg 1993] STOCKENBERG J.E., « A Dynamic Integration Architecture for High Availability Real-Time Systems », *Proceedings of the Conference on Software Maintenance (CSM'93)*, p. 51 –60, 1993.

[Suanno 1996] SUANNO G., VITTA M., « A DSP Approach to Space Scientific Experiments », *Proceedings of the 7th International Conference on Signal Processing applications and Technology (ICSPAT'96)*, Boston, 7-10 October, 1996.

[Tagant 1998] Tagant A.M., « Agents mobiles, objets mobiles : quels problèmes faut-il résoudre ? », *Actes de NOTERE'98*, Montréal, Québec, Canada, Octobre 1998.

[Tennenhouse 1997] TENNENHOUSE D., ET AL., « A Survey of Active Network Research », *IEEE Communication Magazine*, vol. 35, p. 80-86, January 1997.

[Terrier 2000] TERRIER F., GÉRARD S., « Real Time System Modeling with UML: Current Status and Some Prospects », *Proceedings of the 2nd Workshop of the SDL Forum society on SDL and MSC*, SAM 2000, Grenoble, France, 2000.

[Toussaint 1997] TOUSSAINT J., SIMONOT-LION F., THOMESSE J.-P., « Time constraints verification method based on time Petri nets », *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends of Distributed Systems*, p. 262-267, Tunis, Tunisia, 29-31 October 1997.

[Traore 2000] TRAORÉ I., « An Outline of PVS Semantics for UML Statecharts », *Journal of Universal Computer Science*, Vol. 6, No. 11, p. 1088-1108, 2000.

[Tyrrell 1992] TYRRELL S., URIEL M. R., « Dynamic Software Configuration Management in Broadband Telecommunications Environments », *Proceedings of the 14th International Switching Symposium (ISS'92)*, vol. 2, Yokohama, Japan, October 25-30, 1992.

[Wittig 2000] WITTIG M., « Satellite Onboard Processing for Multimedia Applications », IEEE Communications Magazine, p. 134-140, June 2000.

[Yan 2001] YAN D.K.Y., CHEN X., « Resource Management in Software-Programmable Router Operating Systems », *IEEE Journal on Selected Areas in Communications*, vol. 19, issue 3, p. 488 –500, March 2001.

Résumé : Autrefois confinées à un rôle de traitement de signal, les charges utiles des satellites de télécommunication assurent à présent des services réseaux qui nécessitent la mise en œuvre de moyens logiciels complexes. Le caractère critique de ces fonctions logicielles et l'évolution de la nature des flux de données transférés sur une longue période d'exploitation soulèvent la problématique de la mise à jour de ces fonctions tout en assurant la continuité du service offert aux utilisateurs.

Les approches de reconfiguration des applications spatiales reposent actuellement sur la technique de rustine logicielle. Or, celle-ci induit un arrêt de service conséquent. De plus, les autres approches ne gèrent la continuité de service qu'au travers du maintien de la cohérence logique applicative et ignorent ainsi les contraintes temps-réel. Ce double constat nous amène à proposer une nouvelle méthodologie de reconfiguration dynamique reposant sur une validation formelle du respect des contraintes applicatives intrinsèques et extrinsèques en préalable à l'exécution de la reconfiguration. Pour cela, nous avons introduit un profil UML temps-réel dont les forces sont l'expression explicite des synchronisations et du comportement temps-réel des composants logiciels. La sémantique de ce profil est obtenue par traduction vers le langage formel RT-LOTOS supporté par un outil de validation. A l'aide de ce profil, la modélisation conjointe de l'application et d'un scénario exprimant l'ordonnancement des opérations de reconfiguration permet de valider formellement le respect des contraintes applicatives et de continuité de service. Un environnement support a été développé afin d'interpréter à bord les scénarios de reconfiguration après validation. Cette interprétation s'appuie sur des mécanismes logiciels tels que l'intégration dynamique de code.

La méthodologie a été appliquée avec succès sur un logiciel charge utile de télécommunication avec commutation embarquée de paquets.

Mots-clef : Reconfiguration, Temps-réel, Vérification, Validation, Architecture objets

Abstract: Satellite telecommunication payloads are now in charge of software-implemented advanced network services. Transferred data streams' quality of service constantly evolving, over long periods, an issue is raised concerning payload's software functionalities upgrade. Moreover, service continuity constraints require these upgrades to be dynamically performed.

Space-based embedded software's usual reconfiguration techniques rely on the patch technique which induces a consequent service interruption. Additionally, existing reconfiguration techniques for other kind of applications manage service continuity only through the application's logical consistency management: therefore, they totally omit real-time constraints. These two facts lead us to propose an innovative dynamic reconfiguration methodology. Before they are performed, dynamic reconfigurations are formally validated i.e. the dynamic reconfiguration is proved to respect intrinsic and extrinsic software constraints. To do so, we have introduced a real-time UML profile. It includes the definition of composition operators – particularly parallelism and synchronization - that extend UML's structural design capacity, and the definition of real-time behavior operators. The profile semantics is obtained through a translation to RT-LOTOS, a formal language supported by a validation tool. Using this profile, the modeling of both the application and a scenario which describes dynamic reconfiguration operations' temporal and ordering constraints makes it possible to formally validate this model against the violation of intrinsic and extrinsic software constraints. An onboard software environment has been implemented to interpret validated scenario. The interpretation relies on software mechanisms such as dynamic code integration.

The proposed methodology has been successfully evaluated over a space-based embedded software in charge of a satellite telecommunication packet switching payload.

Keywords: Reconfiguration, Real-time, Verification, Validation, Object-oriented architecture