Université de Savoie Polytech'Savoie

ÉCOLE DOCTORALE SISEO

Année 2008

Analyse de performances des systèmes basés composants

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE SAVOIE

Discipline : Informatique

présentée et soutenue publiquement par

Nabila SALMI

Date de soutenance : 18 Septembre 2008

à Polytech'Savoie

devant le jury ci-dessous

Pr. Yamine AIT-AMEUR, Professeur	ENSAM. Poitiers
Pr. Kamel BARKAOUI, Professeur	CNAM. Paris
Bruno DILLENSEGER, Ingénieur de Recherches	Orange Labs. France Telecom
Pr. Giuliana FRANCESCHINIS, Professeur	Universita del Piemonte Orientale. Alessandria
Pr. Malika IOUALALEN, Professeur	USTHB. Alger
Pr. Patrice MOREAUX, Professeur	Polytech'Savoie. Annecy
Dr. Hervé VERJUS, Maître de Conférences	Polytech'Savoie. Annecy

Directeur de thèse : Pr. Patrice MOREAUX
Laboratoire : LISTIC

N° ED 0366-XXX

ANALYSE DE PERFORMANCES DES SYSTÈMES BASÉS COMPOSANTS

Performance analysis of Component-Based Systems

Nabila SALMI

Université de Savoie - Polytech'Savoie

Le commencement de toutes les sciences, c'est l'étonnement de ce que les choses sont ce qu'elles sont.
Aristote, Métaphysique, I, 2.

Ce document a été préparé avec LaTeX2ε et la classe `these-IRIN` version 0.92 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-IRIN` est disponible à l'adresse :

`http://login.irin.sciences.univ-nantes.fr/`

Impression : principalthesis.tex – 26/9/2008 – 15:57

Révision pour la classe : $Id: these-IRIN.cls,v 1.3 2000/11/19 18:30:42 fred Exp

Remerciements

Avant tout, un grand Merci à Dieu de m'avoir donné la force d'achever cette thèse.

Je tiens d'abord à remercier vivement les membres du Jury qui me font l'honneur de juger ce travail.

Je remercie M. Yamine AIT-AMEUR, Professeur d'Université à l'École Nationale Supérieure de Mécanique et d'Aérotechnique (ENSMA) de Poitiers et M. Kamel BARKAOUI, Professeur d'Université au CNAM de Paris d'avoir accepté d'être rapporteurs de cette thèse et d'effectuer une évaluation approfondie de mes travaux.

Je remercie vivement aussi Mme Giuliana FRANCESCHINIS, Professeur à l'Université del Piemonte Orientale, Alessandria, Italie, de me faire l'honneur de faire partie de mon jury de thèse.

Mes remerciements vont également M. Hervé VERJUS, Maître de Conférences à l'École Polytech'Savoie d'Annecy, Université de Savoie, ainsi que M. Bruno Dillenseger, Ingénieur de recherches à Orange Labs de France Telecom, pour avoir bien voulu examiner mon travail d'un oeil attentif.

Je ne saurais assez remercier mon directeur de thèse M. Patrice Moreaux, Professeur d'Université à l'École Polytech'Savoie d'Annecy, ainsi que Mme Malika Ioualalen, Professeur à l'Université des Sciences et Technologie de Houari Boumediene (USTHB) d'Alger : M. Patrice Moreaux, pour sa vision et sa rigueur scientifique, ainsi que ses remarques et conseils précieux qui m'ont guidé tout au long de ce travail de thèse; Mme Malika Ioualalen, pour ses encouragements, son appui, et également pour ses remarques et conseils utiles. Je tiens à leur exprimer ma très vive gratitude et reconnaissance.

Je ne manquerais pas de remercier M. Philippe Bolon, Professeur d'Université à l'École Polytech'Savoie d'Annecy de m'avoir accueilli dans le laboratoire LISTIC dont il est directeur, ainsi que tous les membres et personnel du laboratoire qui m'ont témoigné leur sympathie et apporté leur aide.

Merci également à mes collègues de travail à l'Université des Sciences et Technologie de Houari Boumediene (USTHB) d'Alger, département Informatique, pour leurs encouragements, leur appui et leur sympathie, en particulier, Souad, Samia, Baya et Djaouida, ainsi que mes amies Wahiba, Louisa, Karima et bien d'autres.

Merci enfin aux doctorants qui m'ont témoigné leur sympathie et leur aide, en particulier Karim pour sa précieuse aide et appui et Nacima. Je n'oublierais pas ma mère, mes soeurs et mon amie Ihcene d'Annecy qui m'ont vivement soutenu durant cette dernière année.

Table des matières

Partie II — Analyse structurée des CBS

Table des figures

Partie I — État de l'art

Partie II — Analyse structurée des CBS

Partie III — Outils et étude de cas

Liste des tableaux

Partie I — État de l'art

Partie II — Analyse structurée des CBS

Partie III — Outils et étude de cas

Introduction générale

Depuis de nombreuses années, l'analyse des systèmes a suscité l'intérêt des chercheurs. L'objectif principal était de construire des systèmes corrects, fiables, répondant aux exigences initialement tracées, et éviter des pertes importantes ou encore quelquefois la survenue de catastrophes. De telles situations se sont réellement produites à travers l'histoire, lorsque certaines erreurs critiques n'ont pas été vérifiées et prévues . On en cite la panne du réseau téléphonique des États Unis d'Amérique en 1989, ou encore la destruction de la fusée Ariane 5 en 1996. En conséquence, il s'avère primordial d'analyser un système (logiciel ou matériel), notamment avant sa mise en route.

Dans le contexte des systèmes logiciels, il est d'autant plus important d'analyser un système. Ces systèmes requièrent une certaine correction, en ce qui concerne les aspects fonctionnel et temporel, en particulier lorsqu'il s'agit d'applications critiques, réactives et temps réel telles que les applications de contrôle avionique, de gestion industrielle, ou encore les applications embarquées. En fait, la complexité d'analyse d'un système s'est accentuée au fil du temps.

Conception de systèmes logiciels, orientation vers le paradigme composant

A l'origine, la conception de systèmes logiciels a commencé par de petits programmes monolithiques réalisant une tâche donnée. Puis, est venue l'époque de programmes conçus sous forme d'un ensemble de modules ou fonctions appelées à partir d'un corps principal, initiant ainsi la réutilisation de code. Toutefois, ce n'était pas suffisant, et l'on cherchait après à structurer les applications autour d'objets, plutôt que de les structurer autour d'actions. Ceci a donné naissance à l'approche *orientée objets*.

Le principe de l'orienté objets consiste principalement à organiser les programmes comme un ensemble d'objets coopérants, appartenant à la réalité physique. Ces objets sont construits à l'aide d'un *moule* ou *classe* comprenant les propriétés essentielles d'une *instance* d'objet. Avec cette approche, la réutilisation de code s'est accentuée de plus en plus, et nombreuses sont les applications qui sont aujourd'hui conçues autour d'objets.

Ainsi, l"approche orientée objets a eu beaucoup de succès et a permis de mettre à l'avant le développement du logiciel. Néanmoins, des limites ont été ressenties, liées notamment au faible degré de réutilisation de code et au passage à l'échelle. Le souci était d'améliorer le coût et le temps de développement par réutilisation de code. Certains chercheurs ont alors pensé à réutiliser des fragments de code précompilés et les assembler pour former une nouvelle application, à l'image des systèmes matériels, conçus par assemblage de composants électroniques préfabriqués. Les fragments de code sont dits *composants*. Ainsi, naquit l'approche *basée composants*.

Systèmes basés composants, nouvelle démarche de conception

La dernière décennie a connu une orientation de l'industrie du logiciel vers la conception de systèmes sous la forme d'assemblage de composants. L'objectif de ce type de conception est de produire

rapidement des applications de qualité par réutilisation de composants précompilés et de faciliter la maintenance et la dynamicité des applications. Un *composant* est une unité de composition, munie d'interfaces spécifiées contractuellement, et de dépendances de contexte explicites [205; 206]. Une *interface* est un point d'accès au composant, définissant des services offerts ou requis. Les composants sont assemblés en connectant leurs interfaces, formant un *système basé composant* (CBS). Un composant peut lui-même contenir un nombre fini de *sous-composants*, permettant un emboîtement hiérarchique (composant *composite*). Les composants du plus bas niveau de la hiérarchie sont dits *primitifs*.

Afin de permettre la construction d'applications basées composants, des *modèles de composant* ont été définis. Il existe de nombreux modèles de composants tant dans le domaine académique que dans le domaine industriel : EJB, CCM, .net, FRACTAL, PECOS, Koala, [204; 166; 167; 149; 159; 43], etc. Pour la plupart de ces modèles, un *langage de description d'architecture (ADL)* permet de décrire un assemblage de composants définissant une application. À partir d'une telle description, un outil de compilation génère le code de l'application en réalisant les vérifications usuelles correspondantes.

Au début, les concepteurs se sont intéressés à vérifier des propriétés de compatibilité entre les composants à interconnecter, pouvant être de différents concepteurs. Cette vérification de compatibilité est principalement basée sur la comparaison de types de composants. Cependant, ce type de vérification s'est avéré insuffisant. Le besoin de pousser la vérification à des propriétés plus critiques a été ressenti, comme par exemple vérifier la propriété d'absence de blocage, la terminaison d'un service ou d'une application, ou encore calculer le temps de réponse à une requête, etc. Globalement, les concepteurs doivent avoir l'assurance que le système résultant d'un assemblage de composants est correct et peut atteindre les objectifs pour lesquels il est construit et les performances attendues de lui. Il est donc préconisé de développer des méthodes et outils qui permettent d'effectuer une analyse qualitative et quantitative des CBS, pour fournir aux concepteurs un support dans leurs activités.

Analyse de performances, contexte de notre travail

L'analyse d'un système revêt deux aspects :
(i) L'aspect qualitatif qui consiste à rechercher les propriétés du système considéré, comme l'existence d'une situation de blocage, l'accessibilité à un certain état, l'existence de conflits, etc.
(ii) L'aspect quantitatif, dynamique des systèmes, où sont calculées les valeurs d'un ensemble d'indices de performance, qui rendent compte des performances du système. Nous pouvons citer, par exemple le temps de réponse à une requête dans un système multi-utilisateurs, le nombre moyen d'utilisateurs connectés à un site Internet de E-commerce, le débit moyen des transactions dans un système bancaire, le taux de perte des messages dans un réseau de communication, etc. Dans le contexte des CBS, plusieurs indices de performances sont recherchés, à savoir le temps de réponse à une requête traitée par un composant, le temps de réponse du CBS, le taux d'erreurs générées, ...

En particulier, l'analyse des CBS a fait l'objet de plusieurs travaux. La plupart de ces travaux ont essentiellement adressé l'analyse qualitative comportementale, alors que très peu ont étudié l'évaluation des performances des CBS.

Nous nous intéressons dans nos travaux à l'analyse des performances des CBS. Dans ce cadre, trois catégories de méthodes sont généralement utilisées :

1. Les méthodes de mesure qui mesurent le système à analyser durant son opération ou activité. Ces techniques sont essentielles pour le contrôle des systèmes en fonctionnement, afin de procéder à des réglages ou équilibrage de charge.

2. Les méthodes de simulation [156; 192; 187] qui génèrent les résultats par un programme de simulation qui imite le comportement du système, comme par exemple la génération aléatoire d'événements discrets répartis dans le temps. Ces techniques restent encore importantes pour l'estimation d'indices de performances de modèles trop complexes à analyser par les méthodes analytiques, à cause généralement de la taille de l'espace d'états généré.
3. Les méthodes analytiques qui consistent à résoudre un modèle mathématique englobant les caractéristiques essentielles du système à analyser.

Le développement des méthodes analytiques a pris de l'ampleur depuis les années 1970. Pour calculer des indices de performance, elles se basent sur des outils mathématiques, consistant essentiellement en : la théorie des probabilités et particulièrement des processus stochastiques markoviens [120; 54; 82], l'analyse opérationnelle [69] ou encore la théorie des algèbres dites (max,+) [15] en liaison avec la théorie des processus ponctuels; ces deux dernières ne sont pas fondées sur des propriétés probabilistes.

Les méthodes stochastiques markoviennes s'intéressent à la résolution à long terme (à l'équilibre) ou à un instant donné (analyse transitoire) de la chaîne de Markov du modèle, permettant ainsi de calculer les probabilités pour que le système se trouve dans chacun des états de la chaîne. Elles s'appuient sur l'utilisation d'un modèle formel pouvant être de bas niveau tel que les chaînes de Markov [120; 54], ou bien de haut niveau comme les réseaux de files d'attente [120; 121; 79], les algèbres de processus stochastiques et les réseaux de Petri Stochastiques [161; 157; 81].

Modèles de performances et motivation du travail de thèse

Le modèle fondamental de l'analyse des performances des systèmes informatiques (et d'autres encore) consiste en le modèle de *réseau de files d'attente* [120; 121; 79]. Avec ce modèle et pour des cas « relativement simples », il est possible de calculer une solution analytique des probabilités à l'équilibre. En plus des files d'attente, de nombreux modèles plus ou moins spécialisés ont été développés pour l'étude des systèmes informatiques : réseaux de Petri, files d'attente particulières, réseaux d'automates stochastiques, algèbres de processus stochastiques, etc. Le développement de ces modèles avait pour objectif de répondre à la diversité et à la complexité croissante des systèmes étudiés.

Dans le contexte des CBS, les quelques travaux d'analyse de performance proposés se différencient par le modèle de performance utilisé. Parmi ces travaux, Rugina et al. [188] ont proposé la traduction d'une description en langage AADL d'un système vers un modèle *Réseau de Petri Stochastique Généralisé* (GSPN) pour des fins de mesure de fiabilité (dependability). D'autres travaux ont suivi l'approche de prédiction des performances [62], proposée pour le domaine du génie logiciel. Cette approche part des spécifications d'architecture, exprimées généralement en langage UML [34], et les traduit en modèles adéquats pour l'analyse de performance. Suivant cette méthode, [222; 94] ont proposé de construire un modèle de *files d'attente* particulier dit *à couches* (*Layered Queuing Networks*) [175] pour calculer les performances d'un système.

Ces approches sont intéressantes. Toutefois, elles sont limitées par le manque de généricité et la pertinence du modèle de performance utilisé. De plus, lorsque la taille du système considéré est importante, il devient difficile, voire impossible, d'effectuer une analyse, en raison de l'explosion combinatoire de l'espace d'états générés. Il convient alors de penser à développer de nouvelles méthodes et outils d'analyse de CBS, qui puissent répondre à ces exigences.

Dans cet objectif, il est d'abord impératif de choisir le modèle de performance adéquat à utiliser. À ce sujet, nous notons un inconvénient majeur du modèle des réseaux de files d'attente, notamment le

manque d'expressivité : en effet, il n'est pas possible avec ce modèle d'exprimer certaines propriétés liées aux systèmes complexes de nos jours, telles que la synchronisation, le parallélisme et les situations de conflits. De même, les modèles d'automates et d'algèbres de processus stochastiques souffrent du même désavantage.

À l'encontre, les réseaux de Petri stochastiques [161; 157; 81] se caractérisent par un pouvoir d'expression puissant des propriétés des systèmes complexes. Ceci a poussé les concepteurs à utiliser largement ce formalisme dans le domaine de l'évaluation de performances des systèmes informatiques [14]. Par ailleurs, lorsque la taille du système est considérable, il est important de manipuler un modèle permettant l'agrégation et l'exploitation des symétries d'un système. Dans ce sens, des modèles réseaux de Petri de haut niveau (colorés) ont été développés. Parmi ces modèles de haut niveau, le modèle *réseau stochastique bien formé* (Stochastic Well formed Net, SWN) est intéressant pour la large panoplie de résultats numériques, d'algorithmes et d'outils d'analyse qu'il offre. Il est particulièrement adapté aux systèmes caractérisés par des symétries de comportement. Ce qui appuie le choix de ce modèle de performance.

Apports de la thèse

Cette thèse s'inscrit dans le cadre de l'évaluation des performances des systèmes basés composants. Notre objectif est de développer une méthode générique d'analyse des performances des systèmes à composants, qui part de la description d'architecture et des comportements des composants, pour aboutir au calcul d'indices de performances à l'équilibre de ces systèmes. Nous cherchons à calculer ces indices par des méthodes exactes. Ce travail propose de pallier aux insuffisances en terme d'analyse et d'évaluation des performances des CBS. De plus, nous visons à analyser des systèmes complexes de grande taille.

L'idée clé de notre méthode est de tirer parti de l'architecture compositionnelle des systèmes à composants, pour réduire la complexité d'analyse en termes de temps de calcul et d'occupation mémoire et permettre d'analyser des systèmes à espace d'états important.

Pour ce faire, nous introduisons une nouvelle méthode d'analyse, basée sur le modèle *réseau stochastique bien formé* (SWN). Cette méthode résume deux apports essentiels de notre travail :

1. Le premier apport consiste en la proposition d'une modélisation d'un CBS, appropriée à l'analyse structurée de performances. Notre méthode part de l'architecture à composants et modélise le CBS à étudier par conversion en modèles SWNs, à l'aide de règles systématiques que nous avons définies. Cette modélisation dépend fortement du modèle de composant et des modes de communication entre les composants. Globalement, nous distinguons deux modes d'interaction entre composants : les interactions *synchrones* (invocation de service ou méthode) et les interactions *asynchrones* (envoi/réception d'événements). Pour chaque type d'interaction, nous proposons une modélisation systématique, appropriée aux interfaces de composants.
2. Le second apport réside dans l'extension de travaux précédents ayant défini une approche structurée d'analyse des performances d'une décomposition synchrone ou asynchrone de SWNs, pour être exploitable dans le contexte d'un CBS. Le but est de réduire la complexité d'analyse (temps de calcul et espace mémoire nécessaire à l'analyse). Pour cela, l'approche se fonde sur une description tensorielle du générateur de la chaîne de Markov agrégée sous-jacente. Nous avons étendu cette méthode à des compositions mixtes de SWNs et l'avons adaptée aux CBS. Ceci a nécessité l'étude de trois problèmes :
   (1) composer les modèles de composants au lieu de décomposer un SWN global;

(2) ramener une interconnexion de composants en une composition synchrone ou asynchrone de SWNs, afin de pouvoir appliquer la méthode structurée, et

(3) étudier les problèmes soulevés par la présence simultanée de compositions *mixtes* synchrones et asynchrones au sein du SWN global.

Enfin, pour illustrer notre méthode d'analyse des performances, nous l'instancions aux CBS basés sur le modèle Fractal, ainsi qu'aux CBS basés sur le modèle CCM.

Plan de thèse

Ce document est divisé en trois parties.

La première partie présente les différents concepts de base sur lesquels s'appuie notre travail, ainsi qu'un état de l'art sur l'analyse des CBS.

Dans le chapitre 1, nous passons en revue les définitions inhérentes aux systèmes basés composants en les illustrant par des exemples concrets. Le chapitre 2 présente une analyse et classification des différents modèles de composant décrits dans la littérature. Nous énonçons un ensemble de critères d'analyse, nous guidant dans la classification. Puis, nous faisons une synthèse de ces différents modèles. Nous détaillons dans le chapitre 3, les méthodes d'analyse qualitative et quantitative des systèmes, tout en décrivant les formalismes de description et les modèles de performances utilisés à cet effet. Nous nous focalisons ensuite sur l'analyse des systèmes basés composants et dressons un état de l'art des travaux proposés dans ce cadre là. Nous terminons cette partie par le chapitre 4 qui décrit les principaux concepts liés aux réseaux de Petri bien formés, ainsi que les détails des réseaux stochastiques bien formés. Nous décrivons après la méthode de décomposition structurée proposée pour cette classe de réseaux, qui se base sur la communication synchrone ou asynchrone entre sous-réseaux [97; 98].

La deuxième partie de cette thèse décrit l'essentiel de nos travaux en cinq chapitres.

Le chapitre 5 présente le étapes principales de notre approche d'analyse des CBS. Puis, il situe les problèmes étudiés pour développer notre méthode et détaille la première étape consistant en la construction d'un modèle global du CBS à analyser. Le chapitre 6 détaille la partie analyse de performances proprement dite du modèle SWN d'un CBS. Nous étudions dans ce chapitre l'applicabilité des résultats d'analyse structurée pour l'évaluation des performances des CBS, et nous en dégageons les éléments nécessaires pour mener efficacement cette analyse. Nous discutons après dans le chapitre 7 le choix du niveau de granularité de la composition de modèles SWNs, pour des fins d'amélioration des temps de calcul engendrés à l'analyse. Nous présentons cette étude à travers deux exemples de systèmes basés composant. Les chapitres 8 et 9 présentent deux instanciations de notre méthode d'analyse des performances à des systèmes basés sur des composants FRACTAL, et d'autres basés sur des composants CCM. Un exemple d'application est présenté pour chacune de ces instanciations pour illustrer les différentes étapes.

La dernière partie de ce manuscrit présente des applications numériques.

Le chapitre 10 décrit l'ensemble des outils impliqués dans notre travail, à savoir GreatSPN, *compSWN* *PERFSWN* et Scilab. Les deux derniers chapitres 11 et 12 de ce manuscrit appuient notre méthode d'analyse, en présentant les calculs numériques d'indices de performance des exemples d'applications données dans les chapitres 8 et 9.

Enfin, nous terminons par une conclusion résumant l'ensemble de nos travaux, ainsi que les perspectives de notre travail.

PARTIE I

État de l'art

CHAPITRE 1

Systèmes basés composant

1.1 Historique : de la fonction C au composant

A l'aube de la programmation, les programmes étaient écrits pour des machines disposant de peu de ressources de calcul et de mémorisation. Des sous-programmes pouvant exécuter plusieurs fois un même segment de code avec des paramètres différents ont alors été utilisés, afin de réduire l'espace mémoire physique nécessaire pour stocker le code d'un programme. De là a commencé la notion de *réutilisation* de logiciel. Plus tard, les sous-programmes ont été très vite réutilisés dans plusieurs programmes, dans le but de faciliter le développement de logiciels. Des programmeurs ont même tenté d'écrire des sous-programmes génériques de telle sorte qu'ils puissent être groupés dans des librairies et réutilisés de manière systématique.

La réutilisation du logiciel n'était pas l'unique souci. La qualité et le coût préoccupaient encore plus les concepteurs et développeurs. A l'origine, les logiciels étaient conçus pour une durée de vie limitée et des objectifs modérés. Cette conception était telle que l'entretien, la maintenance, l'extension des fonctionnalités, et même la réutilisation de telles applications étaient difficiles et très coûteux à réaliser. Ceci était dû à la difficulté de modifier les programmes existants, et à la dépendance excessive vis-à-vis de la structure physique des données. Il a été estimé que 70 % du coût du logiciel était consacré à la maintenance. Quelquefois, il était même impossible de procéder à cette maintenance. Il était alors plus simple de remplacer les applications par des logiciels entièrement nouveaux. Ceci était inadmissible pour des gros logiciels ayant été l'objet d'un investissement majeur. Il reste certain que la qualité d'un logiciel ne peut être satisfaisante si elle néglige cet aspect.

Le défi des concepteurs s'était alors orienté vers le développement de logiciels de qualité industrielle et ayant une longue durée de vie. L'idée était de rechercher des méthodologies rigoureuses permettant de maîtriser la complexité du logiciel, tout en donnant l'illusion de la simplicité et en réutilisant les architectures et codes existants.

Afin d'atteindre ces propriétés de *réutilisabilité*, d'*extensibilité* et de *compatibilité*, les techniques développées devaient être *architecturales*, produisant des programmes souples, décentralisés et *composés de modules cohérents connectés par des interfaces bien définies*.

Historiquement, en 1968, Dijkstra a été le premier à souligner l'importance de la structuration du code pour l'écriture de programmes corrects, et a introduit la notion de séparation des préoccupations, en démontrant qu'un programme pouvait être composé de plusieurs segments de code implantés de manière indépendante [73]. A la même date, lors d'un colloque de l'OTAN, D. McIlroy [141] présentait les « *composants logiciels produits en masse*« comme étant les briques de base pour la production de masse dans le cadre de l'industrialisation, et prononçait déjà l'idée de l'assemblage à partir de boîtes noires

bien spécifiées sans citer de solution technique. Dans les années 70, Parnas [171] a présenté le concept de la programmation face aux interfaces pour concevoir des logiciels avec des modules interchangeables, en ignorant leurs détails d'implantation. Puis, DeRemer [70] a introduit le concept de « *programmation à gros grain* » (*programming in the large*) qui consiste à programmer des composants logiciels fournissant des abstractions bien définies par des groupes de travail différents. C'est ainsi qu'est apparue la conception *objet*.

La technique *Orientée Objets* repose sur une idée élémentaire : Les systèmes informatiques réalisent des actions sur des objets. Le logiciel est alors structuré autour d'objets plutôt qu'autour d'actions. Ainsi, la programmation par objets consiste essentiellement à mettre en oeuvre des programmes organisés sous-forme d'ensembles d'objets coopérants. Les objets appartiennent à la réalité physique : dans un environnement graphique, il peut s'agir de points, de lignes, d'angles, de surfaces, etc; dans un environnement administratif, il s'agit d'employés, de fiches de paies, d'échelles de salaires, ...etc. Afin de programmer en "objets", la notion de *classe* d'objets est utilisée. Une classe est un « moule » groupant toutes les propriétés essentielles de l'objet et permettant de créer des *instances*. Un objet représente une instance de la classe. Toutes les classes sont des membres d'une hiérarchie de classes unifiées par des relations d'héritage [33].

L'approche objet est élégante et a eu beaucoup de succès. La plupart des applications d'aujourd'hui sont conçues autour d'objets. Toutefois, elle présente des limites importantes en termes de granularité et de passage à l'échelle [56]:

- Seul un faible niveau de réutilisation a été atteint, en raison du couplage fort entre les objets. En effet, ces derniers peuvent communiquer directement sans passer par leurs interfaces.
- La structure des applications objets est peu lisible (consistant en général en un ensemble de fichiers).
- La plupart des mécanismes objets sont gérés manuellement : création d'instances, gestion des dépendances entre classes, appels explicites de méthodes, ...etc.
- Les architectures objets spécifient uniquement les services fournis par les composants d'implantation, sans définir les services ou besoins requis par ces composants.
- Les propriétés non fonctionnelles ne sont pas directement supportées, et peu de solutions sont proposées pour faciliter l'adaptation, l'évolution et l'assemblage d'objets.
- Enfin, les architectures objets ne sont pas adaptées à la description de schémas complexes de communication et de coordination, et la recherche des interactions dans une architecture objet n'est pas facile.

Toutes ces raisons ont fait naître un besoin de concevoir des composants ayant un niveau d'abstraction plus élevé que celui des objets et étant mieux adaptés à une industrie de réutilisation : C'est l'approche *par composants*.

1.2 L'approche basée composants

Les technologies objets ont certes profondément modifié l'ingénierie des systèmes logiciels, améliorant ainsi leur analyse, leur conception et leur développement. Elles sont devenues rapidement très populaires, grâce au gain considérable en termes d'organisation, de réutilisation et de maintenabilité qu'elles apportent. Néanmoins, plusieurs soucis persistaient au sein de la communauté de recherche « Architectures logicielles », notamment, *comment assembler le tout à partir de briques de base* et *comment atteindre un haut degré d'abstraction et de réutilisation* afin d'améliorer la qualité du logiciel.

Une homologie a été alors faite avec la conception de systèmes matériels à partir de composants matériels. Les gens se posaient plusieurs questions: Pourquoi le logiciel n'est pas comme le matériel? Pourquoi tout nouveau développement doit-il repartir à zéro ? N'est-il pas plus intéressant pour construire un nouveau système de commander des composants catalogués dans des catalogues de modules logiciels et les réutiliser directement, comme le font les concepteurs de systèmes matériels qui utilisent des catalogues de puces de circuits intégrés, plutôt que de réinventer la roue à chaque fois ?

Toutes ces questions et bien d'autres ont suscité le développement de l'approche par *composants*. Bien que cette tendance soit ancienne, il a fallu attendre les années 90 pour qu'elle émerge à nouveau. Toutefois, une première tentative en 1986, considérée comme l'ancêtre des composants logiciels tels que définis aujourd'hui, a abouti à la mise en oeuvre d'*ObjectiveC* par Brad Cox [61]. Les composants logiciels ont depuis évolué, en mettant *l'architecture* au coeur du processus de conception du logiciel.

1.2.1 Composants et systèmes à composants

L'expérience gagnée dans la discipline objets a largement contribué à développer les idées de composants et de réutilisation. Nous définissons ici précisément qu'est-ce qu'un composant et un système basé composant.

1.2.1.1 Composant

Selon le Petit Robert, un composant est un « élément qui entre dans la composition de quelque chose et qui remplit une fonction particulière ». Dans le vocabulaire informatique, et précisément dans celui du génie logiciel, un composant a d'abord désigné des fragments de code, puis est passé vers une définition plus générale, englobant toute unité de réutilisation [141]. C'est une brique logicielle de base permettant la construction d'applications logicielles.

Définition 1.1 (Composant)**.** Un *composant* logiciel est une unité de composition, munie d'*interfaces* spécifiées contractuellement, et de *dépendances* de contexte explicites [206]. Il peut alors être déployé indépendamment, et est sujet à la composition par des parties tierces.

D'autres définitions sont données dans la littérature: Meyer [148] a défini un composant logiciel comme un élément de programme pouvant être utilisé par d'autres éléments de programmes dits *clients*. Les clients et leurs auteurs n'ont pas besoin d'être connus par les auteurs de composants.

D'autre part, Harris, de la communauté « architecture logicielle » a décrit un composant comme un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support [103]. De plus, il est doté d'interfaces standards pour pouvoir interopérer.

En pratique, un composant peut être une unité de calcul ou une unité de stockage à laquelle est associée une unité d'implantation. Il peut être une unité source consommée au moment du développement ou de la conception de l'architecture, ou une unité de déploiement (unité exécutable), ou encore une unité de remplacement ou de *versioning*. Les clients, serveurs et bases de données en sont des exemples types. Ces définitions sont associées à différents niveaux de raisonnement sur la nature du composant.

En fait, la nature d'un composant peut évoluer en fonction de la phase du processus de développement considérée :

- Pendant la phase de conception, un composant est vu comme une entité de modélisation de l'architecture.

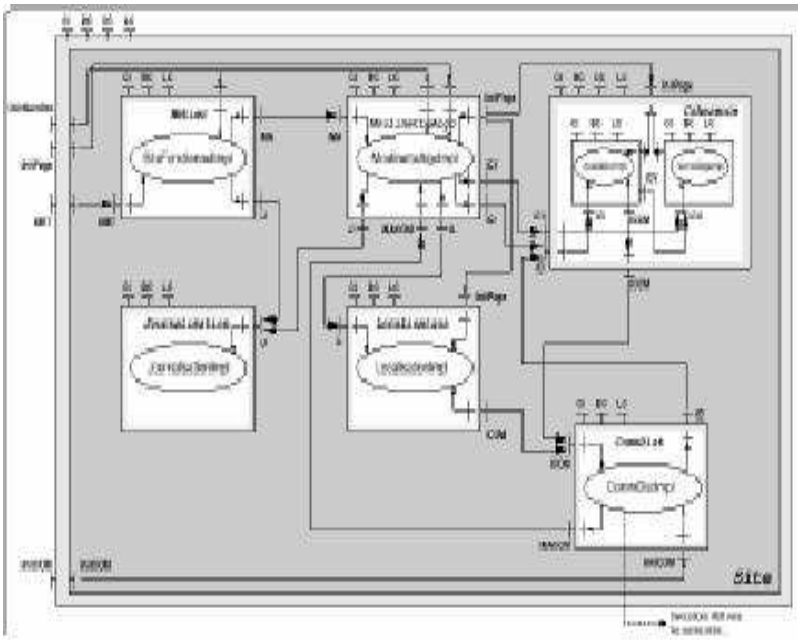


Figure 1.1: Un exemple de CBS

- Pendant la phase de programmation, il est considéré comme une unité de code source, vu sous le moule d'un type.
- Pendant les phases qui suivent, il est vu comme une unité de déploiement et d'exécution.

En parallèle, la nature de l'architecture logicielle évolue d'une forme abstraite vers une forme exécutable.

Dans la suite, nous adoptons une définition à deux niveaux du composant : Un composant est défini d'une manière abstraite par son *type*. A un niveau plus bas, le type est instancié pour créer une *instance* de composant.

Type de composant Le type d'un composant est une définition abstraite d'une entité logicielle. Il présente sa vue extérieure et donne les informations nécessaires qui permettent de présenter ses fonctions, ses dépendances et ses propriétés configurables.

Instance de composant Une instance de composant est une entité s'exécutant dans une application, ce qui correspond à un objet en programmation orientée objet. Chaque instance est caractérisée par un identifiant unique, un type de composant et une implantation particulière de ce type. Elle possède également une valeur pour chaque propriété configurable définie au niveau de son type.

Les composants logiciels sont agencés et assemblés afin de construire une application. Les systèmes ainsi construits sont dits *Systèmes Basés Composants* ou *Component Based Systems (CBS)*.

1.2.1.2 Système basé composants (CBS)

L'approche composant est une technique qui consiste à concevoir et développer des systèmes par assemblage de composants réutilisables (Voir figure 1.1), à l'image des composants électroniques ou mécaniques [56].

Définition 1.2 (Système basé Composant)**.** Un CBS (Component Based System) est un système composé d'un ensemble de composants préfabriqués, préconçus et « prétestés », interconnectés par des contrats, ou interfaces, requis et fournis.

Pour construire un CBS, les développeurs utilisent des *composants logiciels sur étagère (COTS, Commercial Off the Shelf)*, plutôt que de les créer à partir de rien. Ces composants sont préfabriqués, pré-testés, s'auto-contiennent et disposent de documentations appropriées et d'un statut de réutilisation bien défini. La construction d'applications peut ainsi se faire d'une manière rapide [56] et est plus facile à maintenir et faire évoluer.

De nos jours, l'approche composant est plus une façon de concevoir un système logiciel. Elle fait aujourd'hui l'objet de standardisation. Il est de plus en plus apparent qu'elle sera à la base de la conception des générations futures de logiciels, et constituera sans doute une réponse crédible à l'évolutivité nécessaire et permanente des architectures logicielles de demain.

1.2.2 Objectifs et motivations

L'objectif principal de la conception par composants est double :

- D'une part réduire les coûts de l'ingénierie logicielle et des délais de développement, grâce à la réutilisation de fragments ou composants existants. En effet, on prendrait moins de temps à acheter et réutiliser un composant réutilisable qu'à le concevoir, le coder, le tester, le mettre au point et le documenter. Un composant réutilisable (composant *sur étagère, COTS*) est un code compilé ou binaire, qui doit renfermer toutes les informations nécessaires à son déploiement dans un environnement donné.
- D'autre part, produire des applications de qualité, plus fiables et aisément maintenables, et améliorer la productivité et l'industrialisation de la production de logiciels. On résoudrait ainsi en partie plusieurs problèmes tels que les coûts élevés, le dépassement du budget, le manque de fiabilité, ...etc.

D'autres objectifs sont liés au désir de faire partager des concepts communs aux utilisateurs, et construire des systèmes hétérogènes à base de composants réutilisables sur étagère.

1.2.3 Modèles de composant

Afin de concrétiser l'approche composants, de nombreuses propositions de *modèles de composants* ont émergé. Un modèle de composant décrit les concepts à partir desquels sont définis les composants logiciels, ainsi que différents mécanismes basés sur ces concepts [172] :

- Un mécanisme de *composition* permet de mettre en relation au moins deux composants afin qu'ils réalisent une ou plusieurs fonctionnalités d'une application.
- Un mécanisme d'*exécution* interprète le code des composants, afin que ceux-ci réalisent leurs fonctionnalités.
- Un mécanisme de *déploiement* permet de rendre des composants prêts à être exécutés.
- Un mécanisme de *communication* permet à au moins deux composants d'échanger des messages. Ces échanges sont définis en fonction de la composition des composants et de leur déploiement. Ils sont réalisés pendant leur exécution.

La plupart de ces modèles sont orientés fortement sur l'aspect implantation. On trouve des modèles issus tant de milieux académiques et d'autres de milieux industriels. Nous citons les modèles EJB, CCM

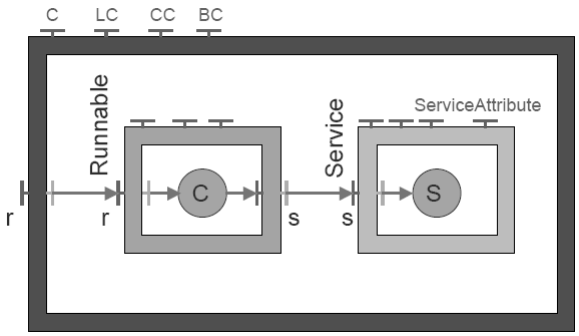


Figure 1.2: Modèle de Composant FRACTAL

et CORBA, COM+/.NET, Fractal,... [204; 166; 167; 169; 149]. Chaque modèle de composants est sous-tendu (explicitement ou implicitement) par une plate-forme qui implante les mécanismes qu'il définit. Dans le monde des composants distribués, cette plate-forme est appelée *intergiciel* (*middleware*).

Sur le plan industriel, des bibliothèques de composants (classes C++, Java, etc...) sont offertes , ainsi que des architectures logicielles pour déployer et intégrer des composants logiciels [1; 77; 32]. Un véritable marché du composant a émergé sur le Web avec des sociétés qui jouent le rôle de *brokers*, constituant un réseau de distribution à travers lequel des composants peuvent être échangés [21; 209].

Exemple 1.1. *Le modèle* FRACTAL *[43] est un modèle de composant général académique, développé au sein du consortium ObjectWeb par France Telecom R&D et l'INRIA (pour plus de détails, voir section 8.2).*
Un exemple d'application simple basée composant Fractal [38] est donné dans la figure 1.2. Telle montrée sur la figure, l'application est constituée de deux composants C (Client) et S (Serveur).

1.3 Caractéristiques des composants

Un composant est caractérisé par plusieurs caractéristiques [56] :

1. Il est autodescriptif, capable, pour certains modèles de composant, de disposer d'un mécanisme d'introspection permettant de connaître et modifier dynamiquement ses caractéristiques.
2. Il est composable ou connectable avec d'autres composants. C'est une unité de composition.
3. Il est configurable, paramétrable via ses propriétés, selon un contexte d'exécution particulier.
4. Il est réutilisable (unité de réutilisation), nécessitant éventuellement une étape d'adaptation au contexte d'exécution.
5. Il est autonome, pouvant être déployé et exécuté indépendamment des autres composants.

Ces caractéristiques sont réalisées grâce à des propriétés concrètes propres à chaque modèle de composant. Généralement, un modèle de composant possède plusieurs propriétés dont : ses *interfaces*, son type, sa sémantique et ses contraintes.

1.3.1 Interfaces

Un composant propose un ensemble de services fonctionnels appelés *services fournis*. Il utilise éventuellement un ensemble de services, appelés *services requis*. Dans la programmation orientée composant, seuls ces services sont considérés. L'ensemble de ces services constitue ce que l'on appelle communément *interface*. Le composant est alors vu comme une boîte noire que les utilisateurs n'ont pas besoin de connaître. Seule la partie *interface* reste visible.

1.3.1.1 Définition d'une interface

Une *interface*, appelée également *port de connexion*, est un point d'accès au composant à partir du monde extérieur. Elle permet de représenter explicitement les relations entre composants et facilite la substitution d'un composant par un autre. Elle spécifie les services (messages, opérations, variables) que le composant peut offrir (interface *fournie*) ou requérir (interface *requise*). Un service est tantôt défini comme une opération, tantôt comme un ensemble d'opérations, modélisant ainsi des services pouvant être composés d'un ensemble de sous-services.

Par exemple, un service de gestion de compte peut être décomposé en un service de consultation de compte, un service de retrait d'argent et un service de dépôt d'argent.

Exemple 1.2. *En reprenant l'exemple 1.1, deux composants Fractal C et S sont représentés dans la figure 1.2. Le composant C fournit une interface nommée "r" et une autre nommée "s". Ces deux interfaces correspondent respectivement à un service fourni (r) et un service requis (s). De même, le composant S offre une interface nommée "s" de type service fourni. D'autres interfaces sont également apparentes sur la figure pour chacun des deux composants. Ces interfaces sont explicitées dans la suite, ainsi que le détail des interfaces fournies et requises.*

La représentation concrète d'une interface fournie ou requise consiste en un ensemble d'opérations pouvant prendre plusieurs formes correspondant aux modèles (patterns) d'interaction : (i) un appel synchrone de méthode ou procédure, avec des valeurs de paramètres; (ii) un appel asynchrone de procédure vu sous forme d'une source ou puits d'événements,

Un nombre de notations, connues comme les *Langages de Description d'Interface* (*Interface Description Languages (IDL)*), ont été désignées pour décrire les interfaces. À ce jour, il n'y a pa de modèle commun unique d'IDL, mais la syntaxe de la plupart des IDLs existants s'inspire de celle d'un langage de programmation procédurale [122]. Certains langages de programmation actuels (comme Java, C#) incluent la notion d'interface et définissent donc leur propre IDL. Une définition typique d'une interface spécifie la signature de chaque opération, i.e. son nom, son type et le mode de transmission des paramètres et des valeurs de retour, et les exceptions qu'elle peut déclencher durant l'exécution (le « requêteur » est sensé fournir les fonctions de manipulation (handlers) de ces exceptions).

Selon le mode d'interaction, plusieurs types d'interfaces sont utilisés.

1.3.1.2 Types d'interface

De nombreuses applications actuelles exécutent des activités qui coopèrent et interagissent entre elles, afin d'atteindre un objectif commun. Les types d'interaction diffèrent entre des interactions synchrones et asynchrones. Comme l'interface d'un composant est un moyen d'interaction avec les autres composants d'une application, plusieurs types d'interface découlent des types d'interaction.

Ainsi, à travers les modèles de composant définis dans la littérature, on peut trouver globalement trois types d'interfaces :

1. Interface de type invocation de service,
2. Interface de type événement, et
3. Interface de type variable partagée.

Invocation de service Dans la programmation orientée objets, un objet est accédé en faisant une invocation de ses méthodes ou services. L'interface d'un objet constitue alors l'ensemble des constituants (méthodes et attributs) publics.

De manière similaire, un composant peut exposer une interface équivalente à la notion d'interface dans la programmation orientée objet. Elle est dite de type invocation de service. Dans ce cas, l'interface peut être de type *fourni* ou *requis*:

- Une interface *fournie* ou *serveur* permet de décrire un service fourni implanté par le composant, et permettant de réagir aux invocations reçues.
- Une interface *requise* ou *cliente* sert à décrire un service requis d'un autre composant, afin de pouvoir invoquer ses opérations.

En termes d'architecture, l'introduction des interfaces clientes présente une avancée considérable par rapport aux objets, en explicitant les références extérieures d'un module logiciel, qui étaient cachées dans le code des objets. Ceci permet de connaître les dépendances d'un composant avec son environnement en regardant son type, et de modifier le fournisseur du service sans modifier le code d'implantation.

Un même composant peut déclarer un ensemble d'interfaces fournies et un ensemble d'interfaces requises. Comme pour les objets, le type d'une interface est défini par l'ensemble des signatures des opérations qu'il contient.

Exemple 1.3. *Dans l'exemple 1.1, figure 1.2, le composant C fournit une interface serveur nommée "r" de type Main, définissant une méthode* Main*, appelée lorsque l'application est lancée. La deuxième interface nommée "s" est cliente, de type Service. Le composant S offre une interface serveur nommée "s" de type Service, fournissant une méthode* Print*.*
Les interfaces sont programmées normalement en utilisant un langage de programmation. Voici le code Java pour la définition de ces interfaces :

```
public interface Service
{
  void print (String msg);
  }

public interface Main
{
  void main (String[] args);
  }
```

Le code des classes et interfaces associées aux deux composants est donné plus tard en section. 8.2.

Interface de type événement Ce type d'interface a été défini pour réaliser une interaction asynchrone entre les composants par la notion d'*événement*.

Un événement est défini comme étant une transition d'état détectable qui peut avoir lieu à un point défini en temps et dans un environnement spécifique [122]. On peut citer des exemples d'événements apparaissant dans l'environnement d'un processus : Le changement de la valeur d'une variable (contenu

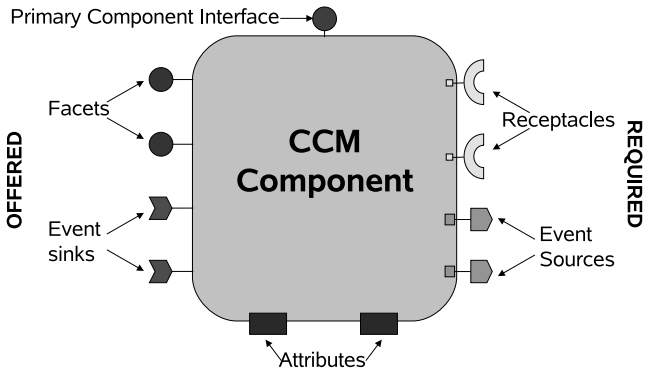


Figure 1.3: Le modèle CCM (CORBA Component Model)

d'une case mémoire); le déclenchement d'une interruption comme par exemple un signal d'expiration de timeout ou un clic de bouton de la souris.

Dans le contexte d'un système distribué, l'événement a un sens plus général: C'est une notification d'une transition d'état locale, envoyée vers un ensemble d'entités réceptrices souvent distantes. C'est également une instance de message, où un message désigne une information transmise par un émetteur à un ou plusieurs récepteurs. En fait, on parle d'événement lorsqu'il s'agit de l'occurrence de la transition initiale (événement local), et de message lorsqu'on sous-entend le contenu en informations étant transmises.

L'entité au sein de laquelle le changement d'état opère est appelée *producteur d'événement* ou *source d'événement*. L'entité réceptrice de la notification est un *consommateur d'événement* ou *puits d'événement*.

En s'inspirant de ces deux entités émettrices et réceptrices d'événement, deux types d'interfaces événement ont été utilisés dans les modèles à composant définissant une interaction événementielle :

- Interfaces *sources d'événement*: points de connexion nommés pouvant émettre vers un ou plusieurs composants des messages asynchrones ou *événements*.
- Interfaces *puits d'événement*: points de connexion qui reçoivent des notifications d'événement à partir d'une ou plusieurs sources.

Généralement, une notification d'événement peut être initiée par l'entité source (on parle alors de mode *push*), ou par l'entité puits (mode *pull*). Une combinaison de ces deux modes peut être aussi utilisée.

La réception d'une notification d'événement par une entité puits déclenche l'exécution d'une réaction spécifique au sein de cette entité. Cette réaction est dite *gestionnaire d'événement* ou *handler*.

Dans certains modèles, les notifications d'événements peuvent être acheminées à travers une entité intermédiaire appelée *canal d'événement*. Celui-ci reçoit des événements des sources et les délivre aux puits d'événements. Le canal d'événement peut également être responsable d'autres opérations telles que le filtrage d'événements selon des définitions de classes d'événements, l'enregistrement des souscriptions des puits aux événements dans le modèle Publish/subscribe (voir le modèle CCM en section 9.2), etc.

Exemple 1.4. *Le modèle CCM (CORBA Component Model) [169] est un modèle de composant indépendant des systèmes d'exploitation et des langages de programmation. Il permet le déploiement de composants au sein d'un environnement distribué.*
Un composant CCM est défini avec deux types d'interfaces (voir figure 1.3) : des interfaces de type

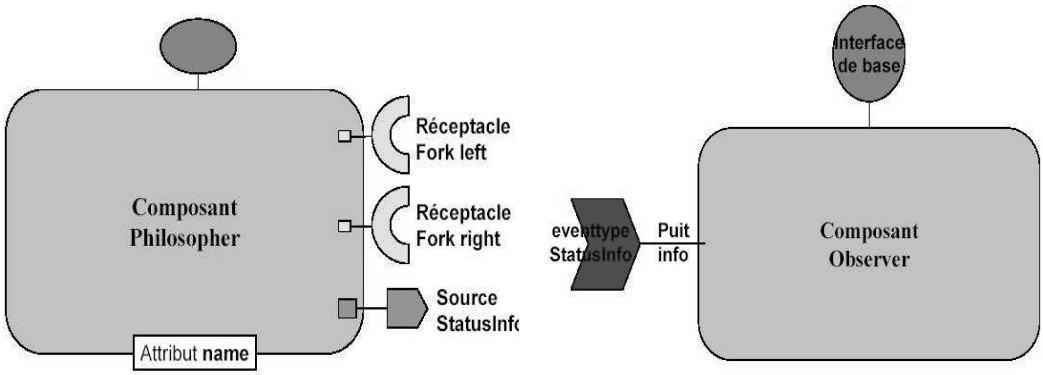


Figure 1.4: Exemples de composants CCM

invocation de service, données par les facettes *(service offert) et* réceptacles *(service fourni), et des interfaces de type événement source et puits. La figure 1.4 présente un exemple de deux composants CCM* Philosopher *et* Observer*qui font partie d'une application simple Le dîner des philosophes. Nous donnons ci-après la description des interfaces associées à ces deux composants en utilisant la notation OMG IDL 3.0.*

```
component Philosopher
{
  attribute string name;
  // La fourchette de gauche
  uses Fork left;
  // La fourchette de droite
  uses Fork right;
  // La source d'événements StatusInfo
  publishes StatusInfo info;
  };

component Observer
{
  // Le puits de réception de StatusInfo
  consumes StatusInfo info;
  };
```

Interface de type variable partagée Certains modèles de composant se sont basés sur la communication par variable partagée pour faire communiquer des composants. A cet effet, un type d'interface adapté a été défini : Tel que son nom l'indique, une telle interface consiste en une variable qui sera partagée avec tout composant qui lui sera connecté.

Exemple 1.5. *Le modèle PECOS [219] définit des composants communiquant par des interfaces ou ports sous forme de variable partagée. Ces interfaces sont caractérisées par un nom, un type, une plage de valeurs et une direction (in, out ou inout). Un exemple de définition d'un composant PECOS est donné*

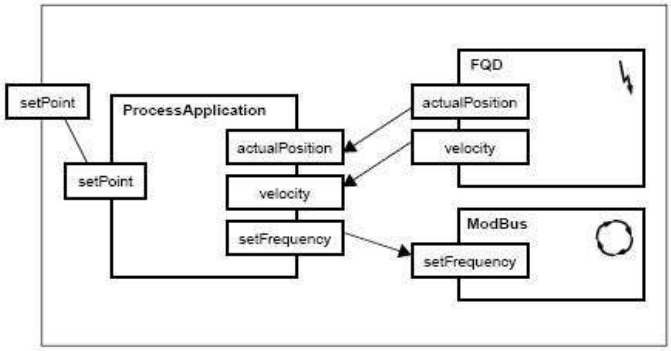


Figure 1.5: Exemple de composants PECOS

ci-après avec le langage CoCo, introduit spécifiquement pour ce modèle de composant. Le composant défini fait partie d'une application CBS donné par la figure 1.5.

```
component ProcessApplication
{
  in float setPoint;
  in float actualPosition;
  in float velocity;
  property cycleTime =100;
  property execTime =20;
  out float setFrequency
  }
```

En fait, ce type d'interface est un cas particulier de l'interface invocation de service, puisqu'une variable partagée est manipulée soit par opérations de lecture ou écriture. Ces deux opérations peuvent être considérées comme des méthodes particulières.

1.3.1.3 Modes de connexion d'interface

Afin de faire communiquer deux composants, on connecte leurs interfaces. La communication entre composants se fait alors selon trois modes possibles [101] dépendant du type d'interfaces connectées :

- *Mode de connexion synchrone*: Ce mode est réalisé à l'aide de l'invocation de service : le composant client demandeur de service appelle une opération de l'interface fournie par le composant serveur, et reste bloqué (synchronisé) jusqu'à la fin de l'opération. Ce mode est utilisé pour les services qui ne peuvent être rendus que sous certaines conditions (temporelles ou applicatives), et qui nécessitent une synchronisation entre le composant offrant le service et celui le recevant, comme le cas des producteurs/consommateurs.
- *Mode de connexion asynchrone*: réalisé par les interfaces de type événement, ce mode est utilisé pour les services sans contraintes de synchronisation entre les deux composants. Ainsi, le composant client demandeur de service ne se bloque pas après avoir invoqué une opération de l'interface serveur requise.

- *Mode diffusion continu* : Ce mode décrit le cas où un protocole est utilisé pour créer un pont de communication continu entre le composant client et le composant serveur.

1.3.2 Attributs

Un composant peut définir un ensemble de *propriétés* ou *attributs*. Un attribut est vu comme un paramètre accessible qui permet de configurer le comportement du composant. Les attributs permettent en général la réutilisation d'un même type de composant dans des contextes différents en permettant d'ajuster les propriétés configurables au début ou au cours du cycle de vie d'une instance de composant. Ils sont généralement définis par des doublets (*nom, type*) et sont accessibles via des opérations de lecture (*get*) et de mise à jour (*set*) fournies par le composant.

Exemple 1.6. *Dans l'exemple 1.1, figure 1.2, le composant S offre une interface nommée "Attribute-Controller" de type ServiceAttributes, qui fournit quatre méthodes pour lire et modifier deux attributs du composant.*

```
public interface ServiceAttributes extends AttributeController
{
  String getHeader ();
  void setHeader (String header);
  int getCount ();
  void setCount (int count);
  }
```

1.3.3 Services non fonctionnels

Outre les services fonctionnels applicatifs (fournis et requis), un composant peut requérir des services *non fonctionnels* pour son exécution. Il peut demander à exécuter une fonctionnalité du système d'exploitation (telle que l'accès à un périphérique, création de fichier,...etc) ou d'un intergiciel (middleware) comme accéder au service de gestion de transactions, de sécurité, de persistance, ou autre.

Les fonctionnalités offertes par un système d'exploitation ou par un intergiciel sont dites *services non fonctionnels*. Un service non fonctionnel peut également désigner un service lié au cycle de vie des composants tel que les opérations de création, suppression, modification des composants ou leurs interconnexions. En effet, un composant est d'abord créé en instanciant un type défini de composant, connecté à d'autres instances de composant, puis activé. Durant son cycle de vie, une application peut être sujette à des reconfigurations éventuellement dynamiques, telles que désactiver des composants, supprimer une instance de composant, supprimer des liaisons entre composants, réaliser de nouvelles liaisons, etc.

Afin de permettre une configuration la plus simple possible des *services non-fonctionnels*, deux concepts ont été utilisés dans les modèles à composants : le concept de *conteneur* et le concept de *contrôleur*.

Définition 1.3 (Conteneur)**.** Un *conteneur* est un environnement d'exécution qui contient un ensemble de composants et leur permet un accès et une utilisation simplifiée des services non-fonctionnels.

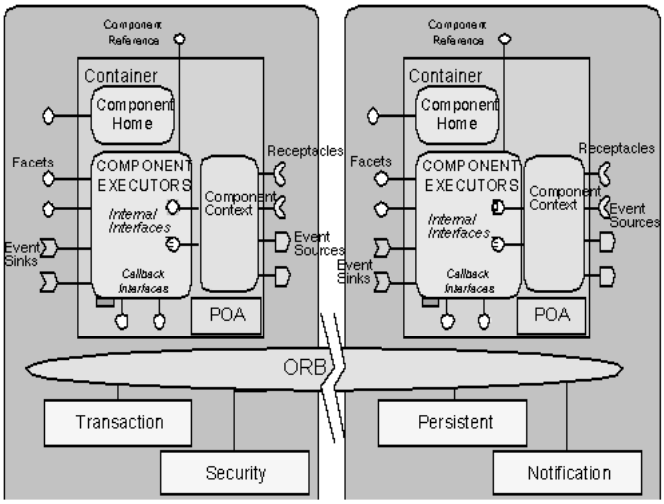


Figure 1.6: Structure de conteneurs et composants CCM

Un conteneur agit comme un gestionnaire pour les composants. Il est responsable de leur création, destruction et réalisation des opérations de reconfiguration. Il permet également de faire appel aux fonctionnalités du système d'exploitation. De plus, comme un composant peut recevoir des requêtes de service ou faire appel à des services fonctionnels de composants appartenant à d'autres conteneurs, le conteneur s'occupe d'acheminer ces invocations depuis ou vers l'extérieur aux composants cibles.

Les invocations sont acheminées par le conteneur à travers deux formes d'objets associés à chacun des composants contenus:
(i) une interface externe (la même que celle du composant), qui agit comme un intercepteur pour tous les appels entrants vers le composant et qui est visible de l'extérieur.
(ii) de manière optionnelle, un intercepteur pour les appels sortants, interne au conteneur. Celui-ci peut également fournir une interface locale afin de réduire le coût de communication entre les composants qu'il contient.

Exemple 1.7. *La figure 1.6 expose le contenu d'un conteneur dand le modèle CCM et sa relation avec les composants qu'il gère. Comme montré dans la figure, un conteneur définit diverses opérations d'accès aux services du middleware tels que les services de persistence, notification d'événements, gestion de transactions, réplication et sécurité. Des stratégies et politiques d'exécution sont également offertes telles que les politiques POA, catégories d'usage des composants, etc.*

La notion de conteneur a été notamment utilisée dans les modèles de composants EJB [204] et CCM [169]. Un notion plus large, le *contrôleur*, a été proposée dans un autre modèle de composant à savoir le modèle FRACTAL [43]. L'intérêt du *contrôleur* est de permettre la personnalisation du contrôle offert aux composants.

Définition 1.4 (Contrôleur)**.** Un *contrôleur* est un méta-objet, faisant partie d'un composant et permettant de gérer les aspects non-fonctionnels requis par le composant.

Un contrôleur a un rôle particulier. Par exemple, il peut être chargé de fournir une représentation causalement connectée de la structure d'un composant (en termes de sous-composants). Il peut contrôler le comportement d'un composant et/ou de ses sous-composants, comme permettre de suspendre/reprendre l'exécution d'un composant. Plus encore, il peut connecter/déconnecter des sous-composants, etc.

Exemple 1.8. *Dans l'exemple 1.1, figure 1.2, le composant C expose une interface non fonctionnelle appelée "BindingController", offrant des méthodes pour lier le composant avec d'autres et gérer ces liaisons (supprimer, lister, rechercher, ...).*

Par rapport au *conteneur*, un *contrôleur* ne fournit pas nécessairement un même flot d'exécution pour les composants qu'il contient (comme dans le cas où les composants sont distribués). Inversement, le *conteneur* ne constitue pas une partie d'un composant, comme c'est le cas pour un *contrôleur*. Le concept de *contrôleur* est plus large puisqu'il intègre la gestion de cycle de vie proposée par les modèles EJB et CCM, à l'intérieur d'un composant lui-même.

1.3.4 Granularité de composant

Un composant est caractérisé par une granularité précise. Il peut être aussi petit qu'une procédure simple, ou bien aussi large qu'une application entière. Pour une meilleure modularité des applications, il est plus intéressant de définir un système comme une hiérarchie de composants emboîtés les uns dans les autres. De ce fait, la hiérarchisation nécessite de définir des unités atomiques dites *composants primitifs* telles que les structures de données et les fonctions mathématiques, et des structures complexes appelées *composants composites*, composées d'éléments ou sous-composants interconnectés par des liens.

1.3.4.1 Composant primitif

Lorsqu'un composant représente une unité atomique non décomposable, il est dit *primitif* ou *de base*. Généralement, il est considéré appartenir au plus bas niveau de l'architecture d'un système, et encapsule du code implantant les fonctionnalités métier lui étant attribuées.

Exemple 1.9. *Considérons l'exemple 1.1, figure 1.2. Nous avons déjà présenté la définition des interfaces des deux composants C et S. Ces deux composants sont primitifs et sont créés en définissant d'abord leur type, puis en instanciant ces types, tel recommandé dans [38].*

```
// type du composant C
ComponentType cType = tf.createFcType(new InterfaceType[]
{
  tf.createFcItfType("m", "Main", false, false, false),
  tf.createFcItfType("s", "Service", true, false, false)
  });
// type du composant S
ComponentType sType = tf.createFcType(new InterfaceType[]
{
  tf.createFcItfType("s", "Service", false, false, false),
  tf.createFcItfType("attribute-controller",
  "ServiceAttributes", false, false, false)
  });
```

Dans ce code, le composant C a un type cType *constitué d'une interface serveur m dont la signature est* Main*, et d'une interface cliente s dont la signature est de type* Service*. Le type client ou serveur de l'interface est décrit par le troisième paramètre (true or false) de l'appel* createFcItfType*. Le composant S a un type* sType *constitué d'une interface serveur s dont la signature est* Service*, et une interface attribute-controller dont la signature est* ServiceAttributes*.*

Les instances (les templates dans cet exemple) des composants C, S sont créés comme suit :

```
// Template du composant C
Component cTmpl = gf.newFcInstance(
   cType, "primitiveTemplate", new Object[] {"primitive", "CImpl"});
// template du composant S
Component sTmpl = gf.newFcInstance(
   sType, "parametricPrimitiveTemplate", new Object[]
   {"parametricPrimitive", "SImpl"});
```

1.3.4.2 Composant composite

Un composant peut lui-même contenir un nombre fini d'autres composants interagissant entre eux, appelés *sous-composants* ou *composants internes*, permettant ainsi aux composants d'être emboîtés à un niveau arbitraire. Dans ce cas, ce composant est dit *composite*.

La notion de composant composite permet de manipuler des assemblages de composants formant une architecture logicielle, comme une seule unité réutilisable de modélisation (comme dans les modèles *Wright* et *Fractal*), ou de programmation (comme le modèle *ArchJava*) (voir aussi section 2.3.3.1). Il permet également de décrire une architecture à différents niveaux de granularité, ce qui facilite la lecture globale d'une architecture.

La déclaration d'un composite consiste alors en la déclaration des instances de sous-composants primitifs et composites qui le composent, et la description des interconnexions de ces instances. Un composant composite a ses propres interfaces. Il est alors nécessaire de lier ces interfaces à celles de ses sous-composants afin de pouvoir exécuter les services du composant composite. Cependant, il est toujours possible de fournir les services par des implantations propres au composant composite.

Exemple 1.10. *En reprenant l'exemple 1.9 et la figure 1.2, le composant parent contenant les deux composants C et S est composite. Il est créé de même en définissant d'abord son type, puis en instanciant ce type comme suit :*

```
// type du composant rComp
ComponentType sType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("s", "Service", false, false, false),
  tf.createFcItfType("attribute-controller", "ServiceAttributes",
                   false, false, false)
});
// Template du composant rComp
Component rTmpl = gf.newFcInstance(
   rType, "compositeTemplate", new Object[] {"composite", null});
// Instance du composant parent rComp (application)
Component rComp=((Factory)rTmpl.getFcInterface("factory")).newFcInstance();
```

1.3.5 Sémantique de composant et contraintes

La sémantique d'un composant décrit le comportement attendu à l'exécution. Le comportement « réel« est donné avec un formalisme ou langage de programmation, qui permet de définir complètement comment le composant s'exécute. De façon générique, la description des comportements de composants est susceptible d'évoluer tout au long du processus de développement, d'une version abstraite (comme par exemple signatures de méthodes et diagrammes UML) vers une version concrète (code des méthodes).

Pour décrire les comportements des composants, deux approches ont été proposées :

- La première approche consiste à utiliser une notation formelle permettant de décrire la *sémantique comportementale* d'un composant ou d'une interface. La sémantique comportementale est la façon dont les activités sont organisées et la façon dont les messages sont reçus et émis en conséquence. Elle décrit le comportement « nominal » d'un composant (globalement ou localement à une de ses interfaces).
- La seconde approche consiste à décrire un ensemble de *contraintes* sur un composant (globalement ou localement à ses interfaces). Une contrainte exprime une règle que le composant doit respecter quant à sa composition et son exécution au sein du système, afin de rester valide durant sa durée de vie. Les contraintes peuvent inclure des restrictions sur les valeurs permises de certaines propriétés, sur l'utilisation d'un service offert par un composant ou autre, refus de certains échanges de messages...etc.

Certaines parties d'un comportement sont plus intuitivement décrites via une sémantique comportementale (telles que le patron global des activités du composant), et d'autres sont plus facilement décrites par des contraintes (telles que les états d'erreur, acceptation de messages,...).

1.4 Autres concepts

1.4.1 Implantation d'un composant

L'*implantation* d'un composant correspond à l'association d'un comportement à un type de composant. En analogie avec la programmation orientée objets, l'implantation correspond à une classe dans un langage à base de classes, ou à un prototype dans un langage à base de prototypes. Elle regroupe une *implantation fonctionnelle* et une *implantation non fonctionnelle*:

- L'implantation fonctionnelle d'un composant représente sa mise en oeuvre d'un point de vue métier, indépendamment des conditions d'exécution. Elle consiste à réaliser l'ensemble des interfaces d'un composant en utilisant ses interfaces clientes si elles existent, et en prenant en compte ses propriétés configurables définies au niveau du type.
- L'implantation non fonctionnelle représente l'adaptation de ce code métier aux conditions d'exécution, en procédant généralement à l'implantation d'autres aspects liés au contrôle du composant dans son environnement d'exécution. On peut citer par exemple l'implantation des propriétés réflexives des composants, l'établissement des services requis à travers des liaisons, la gestion du cycle de vie des composants, etc...

1.4.2 Paquetage d'un composant

Un paquetage d'un composant est une entité « déployable » (souvent une archive) contenant son type, au moins une de ses implantations, ainsi que les contraintes techniques associées à chaque implantation

(langage de programmation utilisé, système d'exploitation cible, etc...).

1.4.3 Déploiement

Le *déploiement* est la phase qui consiste à diffuser, installer et paramétrer des composants sur les sites d'une infrastructure matérielle et système, afin de rendre une application prête à être utilisée [172].

Le site sur lequel est déployé un composant doit offrir un ensemble de *services non-fonctionnels* et un ensemble de *caractéristiques techniques* qui lui sont propres. Les services non fonctionnels ont été explicités dans la section 1.3.3. Quant aux caractéristiques techniques, ce sont des propriétés du site de déploiement, pouvant être utilisées pour vérifier la validité du déploiement. Nous citons par exemple le type du système d'exploitation, la fréquence du processeur, les langages supportés, etc.

1.5 Interconnexion de composants

L'assemblage des composants permet de construire des applications complexes à partir de composants simples. On parle également de *composition* de composants. Celle-ci est effectuée en interconnectant les composants demandant des services à d'autres composants offrant ces services.

1.5.1 Assemblage de composants

Le mécanisme d'assemblage de composants repose sur le principe de *connexion* entre *interfaces* de composants logiciels. La connexion est assurée par des *protocoles* ou *modes* de communication élémentaires (appels de procédures, envoi/réception d'événements, utilisation de messages ou pipes, etc). Elle peut être également réalisée par des protocoles complexes en utilisant la notion de *connecteurs*. L'interconnexion des composants donne lieu à une *configuration* réalisant l'application souhaitée. Ces différents concepts sont définis ci-après.

1.5.2 Modes de communication

Comme introduit dans la section 1.3.1.2, divers types ou modes d'interactions se distinguent dans la littérature. Principalement, on trouve le mode de communication par invocation de service et la communication basée événements.

1.5.2.1 Invocation de service

Les composants offrent des services fonctionnels à leur environnement. Un service est vu comme un comportement défini contractuellement pouvant être implanté pour être utilisé par un composant quelconque, basé uniquement sur un contrat.

Un service élémentaire est défini par une interface qui est, comme vu précédemment, est une description concrète de l'interaction entre le requêteur et le fournisseur du service (voir figure 1.7). Un service complexe peut être défini par plusieurs interfaces, chacune d'elles représente un aspect particulier du service.

L'invocation de méthode suit le modèle du client/serveur pour l'envoi et le traitement de la requête. La caractéristique principale de ce mode de communication est le blocage du requêteur de service après l'envoi de sa requête jusqu'à la réception de la réponse à sa requête.

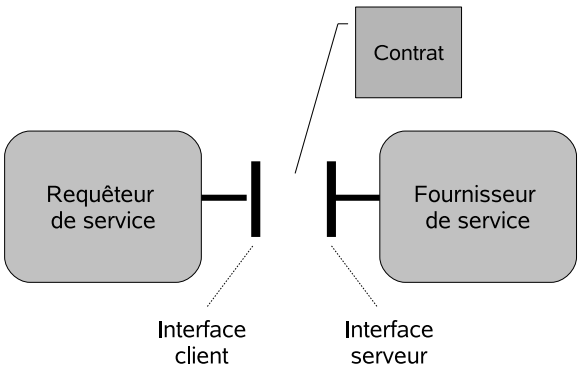


Figure 1.7: Invocation de service

1.5.2.2 Communication basée événements

L'interaction basée événements intervient entre une entité émettrice appelée *source*, et une entité réceptrice dite *puits*. Par opposition à l'invocation de service, ce mode d'interaction libère la source d'événements après l'émission d'un signal d'événement. La source peut ainsi continuer ses tâches en parallèle au traitement de l'événement qu'elle vient d'émettre.

Pour implanter ce mode de communication, deux modèles de communication sont généralement utilisés [122] : le modèle *observateur* et le modèle *publish/subscribe*.

Le modèle observateur Dans sa forme primitive [83], ce modèle implique un objet « observé » et un nombre non spécifié d'objets « observateurs » indépendants. Le problème pour les observateurs est de rester au courant de tout changement se produisant sur l'objet observé.

Cette situation survient typiquement lorsqu'une entité abstraite donnée (comme le contenu d'un tableur) dispose de plusieurs représentations ou images (table, histogramme, camembert, etc). Quand l'entité est modifiée, toutes ses images requièrent une mise à jour pour refléter ces changements. Ces images et les processus qui les maintiennent sont indépendants les uns des autres. De plus, de nouvelles formes peuvent être introduites à tout moment.

Dans ce contexte, la solution du modèle observateur (figure 1.8) propose que les observateurs enregistrent leur intérêt à l'objet observé, qui garde une liste de ces observateurs. Lorsqu'un changement se manifeste, l'objet notifie d'une manière asynchrone tous les observateurs inscrits. Chaque observateur peut alors envoyer une requête à l'objet observé pour récupérer les changements produits. Ce pattern utilise donc les deux modes *push* et *pull* explicités avant.

Cette solution a plusieurs avantages : d'une part, les observateurs peuvent ne pas se connaître entre eux, ni connaître les objets observés, et d'autre part, un observateur informé d'un changement peut demander une information spécifique de l'observé au moment qu'il choisit. Toutefois, ce mode a certaines limitations, à savoir :
(i) L'entité observée peut être surchargée, puisqu'elle fournit et implante une interface pour l'enregistrement, garde trace des observateurs et répond aux requêtes. Ceci est au détriment des performances et du passage à l'échelle (*scalability*).
(ii) Le processus à deux phases de notification et de requête n'est pas sélectif. Il peut entraîner des échanges inutiles, comme dans le cas où un observateur n'est pas intéressé par un changement particulier, mais doit quand même envoyer une requête à l'objet observé.

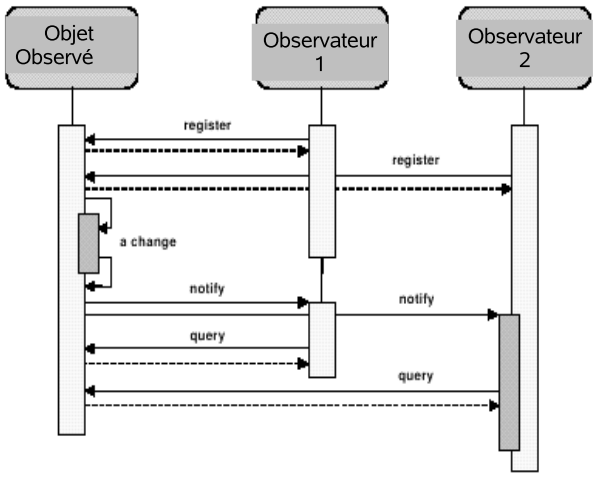


Figure 1.8: Modèle de l'observateur

Le modèle suivant remédie à ces limitations.

Le modèle publish/subscribe Le modèle *Publish-Subscribe* est une forme plus générale de celui de l'observateur. Il définit deux rôles : un *publisher* comme source d'événement et un *subscriber* comme puits d'événement. Une entité donnée peut jouer les deux rôles. Les événements générés par les publishers sont sélectivement notifiés aux subscribers. La sélection est effectuée sur la base de la définition de classes d'événements (ensemble d'événements satisfaisant certaines conditions). Les subscribers enregistrent leur intérêt pour une classe d'événements en *souscrivant* auprès du *publisher* de cette classe. Lorsqu'un événement E est généré par un publisher, tous les subscribers ayant souscrit à la classe à laquelle E appartient reçoivent une notification asynchrone de l'occurrence de E. La réaction à l'événement se fait alors localement à chaque subscriber.

Une forme plus abstraite de ce modèle a été également définie, en introduisant un médiateur entre les publishers et les subscribers : les *canaux d'événements*. Un canal d'événement est une entité responsable de l'enregistrement des subscribers, de la réception d'événement, de leur filtrage selon des classes et de leur routage vers les subscribers intéressés (voir la figure 1.9). Selon les cas, le médiateur peut être implanté en différentes façons : par un serveur centralisé, par des serveurs coopérants distribués, ou éventuellement par les publishers comme le cas des observateurs.

Plusieurs avantages découlent du modèle Publish-Subscribe, à savoir la réduction du nombre de notifications puisque seuls les subscribers intéressés par un événement sont notifiés, et l'anonymat des publishers et des subscribers qui n'interagissent pas directement et peuvent disparaître du système à tout moment sans aucune incidence. En fait, ces bénéfices dépendent fortement de l'implantation du modèle. Par exemple, une implantation centralisée peut réduire l'indépendance mutuelle entre les activités.

1.5.3 Notion de connecteur

Lorsqu'il y a incompatibilité dans la spécification des interactions entre des composants à connecter, il est nécessaire d'utiliser des adaptateurs dits *connecteurs* visant à régler ces incompatibilités.

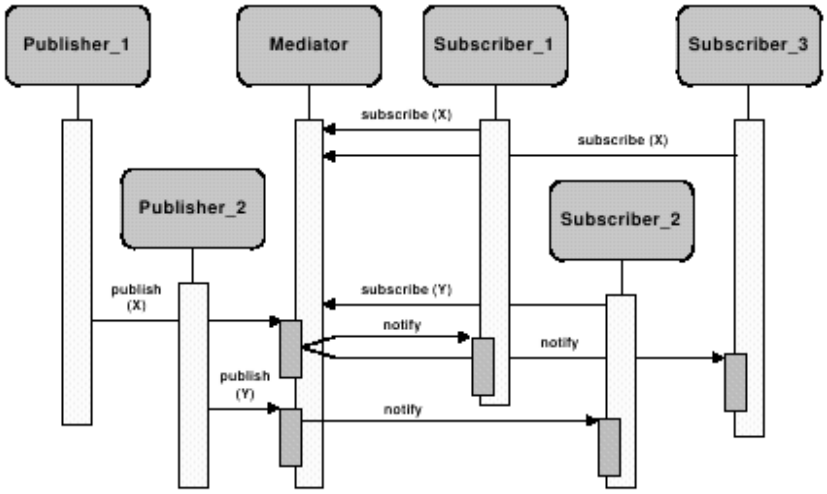


Figure 1.9: Modèle Publish/Subscribe

Définition 1.5 (Connecteur)**.** Un connecteur est une entité de modélisation, représentant un protocole d'interaction entre composants et les règles qui régissent cette interaction.

Souvent, le connecteur est une entité de première classe, décrite, manipulée et réutilisée explicitement au même titre qu'un composant. Il peut également être caractérisé par des propriétés similaires à celles des composants, à savoir le type, les interfaces qui définissent les rôles joués dans l'interaction, les contraintes et les propriétés non fonctionnelles [145].

1.5.4 Notion de Configuration

Une configuration décrit un assemblage de composants interconnectés. Elle l'agrège de telle manière à pouvoir le manipuler comme si c'était un composant. Elle possède également comme propriétés un type et des interfaces au même titre que les composants.

La structure globale d'un assemblage ou une configuration de composants est donnée par l'*architecture logicielle*.

1.6 Architecture logicielle

L'architecture logicielle d'une application est une vision d'ensemble, pouvant être assimilée à un plan de construction. Elle exprime la dimension organisationnelle d'un système ou d'une application logiciel(le) à un haut niveau d'abstraction, l'exposant comme une collection de composants communiquant les uns avec les autres. Elle est définie comme suit:

Définition 1.6 (Architecture logicielle [197])**.** L'architecture d'un système logiciel définit ce système en termes de composants. Elle spécifie la structure et la topologie du système et montre la correspondance attendue entre les besoins du système et les éléments du système construit. Elle peut aussi adresser des

propriétés de niveau système, telles que la capacité, le débit, la consistance et la « composabilité » des composants.

La définition de cette structure est motivée par le fait de fournir un plan précis ou métamodèle approprié pour guider le développement du système et prédire son comportement avant de le construire. C'est une sorte de documentation qui permet de mieux comprendre l'application développée.

Un autre travail de Medvidovic [144] définit l'architecture logicielle comme étant un niveau de conception qui va au-delà des algorithmes et des structures de données d'un calcul : concevoir et spécifier la structure globale du système émerge comme un nouveau type de problème. Les aspects structurels incluent l'organisation et la structure de contrôle globales, les protocoles de communication, la synchronisation et l'accès aux données, l'affectation de la fonctionnalité des éléments de conception, la distribution physique, la composition des éléments de conception, la performance et le passage à l'échelle et la sélection parmi les alternatives de conception.

Afin de décrire l'architecture logicielle d'un système, des langages spécifiques ont été développés. Ce sont les *Langages de description d'architecture*.

1.7 Langages de description d'architecture (ADL)

L'architecture logicielle d'un système constitue son épine dorsale. Afin d'asseoir le développement d'une architecture basée composants, il était nécessaire de disposer de notations formelles et d'outils d'analyse des spécifications architecturales.

Une approche pour décrire les systèmes basés composants est d'utiliser les notations de l'approche objet. Chaque composant peut être représenté par une classe, ses interfaces par des interfaces de classe, et les interactions entre composants peuvent être définies en termes d'associations. Toutefois, cette approche a présenté des limites importantes en termes de granularité et de passage à l'échelle. C'est alors que le monde académique [2; 84] s'est intéressé à formaliser la notion d'architecture logicielle à base de composants, en proposant des langages de description d'architecture (*Architecture Description Language, ADL*).

1.7.1 Langage ADL

Les ADLs trouvent leurs racines dans les langages d'interconnexion de modules (*Module Interconnection Languages*) des années 70 [183]. Un langage de description d'architectures (ADL) est une manière semi-formelle de spécifier les architectures logicielles et systèmes. C'est un langage informatique formel, utilisé pour décrire les composants logiciels (et éventuellement matériels) d'un système et les interactions entre ces composants. Il fournit une syntaxe concrète (notations et sémantiques) et un cadre conceptuel pour la caractérisation d'architectures logicielles décrites par des assemblages de composants logiciels (et éventuellement matériels).

1.7.2 Éléments décrits

La description d'architecture inclut la description des éléments essentiels d'une architecture logicielle [56] :

- Les composants que contient le système,
- les spécifications comportementales des composants,

- les modèles et mécanismes utilisés dans les interactions, à savoir les connecteurs, et
- enfin un modèle définissant une configuration architecturale du système.

De plus, un ADL doit fournir les moyens pour exprimer les interfaces fonctionnelles des composants (services requis et offerts et interfaces à événements). Á un plus haut niveau, la spécification d'une architecture décrit comment ces composants sont combinés en sous-systèmes, comment ils interagissent (structure des flots de données et de contrôles), et comment ils sont alloués sur les composants matériels (si des composants matériels sont utilisés). Ces concepts de base d'une description architecturale sont communs à l'ensemble des travaux menés sur les ADLs.

Par ailleurs, d'autres aspects du paradigme composant peuvent être utiles à définir dans un ADL, mais sont non nécessaires, tels que les contraintes qui servent à faire respecter les utilisations prévues d'un composant, les styles architecturaux et les propriétés non fonctionnelles qui permettent la traçabilité des composants, leur gestion et analyse.

Notons enfin que la description de tant d'aspects résulte en une architecture bien définie, ce qui permet à un concepteur de raisonner sur les propriétés du système décrit, telles que les propriétés de compatibilité entre composants, la conformité aux standards, la performance, la sûreté et la fiabilité.

1.7.3 Catégories d'ADLs

Les ADLs ont été classifiés par Medvidovic et Taylor [144] en trois catégories :

1. Les ADLs de configuration, comme C2 [143], qui permettent de décrire des architectures contenant les informations nécessaires pour générer le code squelette de leurs composants et connecteurs. Certains ADLs permettent de manipuler des informations de déploiement et d'administration d'applications instances de ces architectures.
2. Les ADLs d'intégration, comme Acme [86; 87], qui permettent d'intégrer et de partager des éléments d'architecture définis via des ADLs hétérogènes.
3. Les ADLs formels reposant sur des formalismes de modélisation, qui permettent de décrire les propriétés des composants, des connecteurs et des configurations (leur comportement, leurs contraintes d'utilisation, leurs propriétés non fonctionnelles,...etc).

Dans cette thèse, nous nous intéressons à cette troisième catégorie d'ADLs. En effet, l'utilisation de notations formelles facilite le développement et l'utilisation d'outils d'analyse, afin de déduire à partir de la description d'éventuelles erreurs de conception, des propriétés qualitatives et quantitatives, ...etc. La validation, la vérification et l'analyse des spécifications architecturales est centrale dans ces langages. Dans cet esprit de vérification et validation, un ADL englobe typiquement une *théorie sémantique formelle*. Cette théorie caractérise les architectures; elle influence la convenance d'un ADL pour un type particulier de systèmes (comme les systèmes à haute concurrence), où certains aspects d'un système (tels que ses propriétés). On peut citer comme exemples de théories formelles les réseaux de Petri [174], les diagrammes d'états [102], les ensembles d'événements partiellement ordonnés [134], les processus séquentiels communicants (CSP) [106], etc.

1.7.4 Exemples d'ADLs

On trouve parmi les ADLs certains étudiés par l'école américaine, ainsi que d'autres proposés par l'école européenne.

- L'école américaine se retrouve en UniCon [198; 199], C2 [143], Wright [10], Acme [86; 87] et ArchJava [8].

- L'école européenne a défini Darwin [135; 137], SOFA [180] et Fractal [43].

Plus de détails sur ces ADLs et leurs modèles de composants sont donnés dans le chapitre suivant. Nous donnons toutefois un exemple simple de description ADL.

Exemple 1.11. *L'exemple 1.1, figure 1.2, décrit une application simple nommée* HelloWOrld*. Nous donnons ci-après sa description Fractal ADL.*

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature=
  "java.lang.Runnable"/>
  <component name="client">
    <interface name="r" role="server" signature=
    "java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

1.8 Conclusion

Le souci de réutilisabilité de code et de réduction des coûts et délais de développement d'applications a conduit à l'approche composant. Dans cette approche, une application est construite en assemblant des composants existants et préconçus à l'image de l'assemblage de composants matériels. Un composant est vu comme une boîte noire, dotée de points d'accès dits *interfaces*. Les interfaces permettent de relier les composants afin d'atteindre un objectif global.

Afin de construire des systèmes basés composants, des modèles de composant ont été proposés dans la littérature, ainsi que des outils appropriés tels que les langages de description d'architecture.

Pour mieux cerner les caractéristiques globales communes aux modèles de composant, nous présentons dans le chapitre suivant les modèles les plus répandus dans la littérature, à savoir aussi bien dans le monde académique que dans le monde industriel.

CHAPITRE 2

Classification des modèles de composant

2.1 Introduction

De multiples tentatives de définition de composants logiciels existent dans la littérature. Ces tentatives varient d'une vision de morceaux de code encapsulés via des interfaces avec un format standard (tel que le modèle Microsoft COM), jusqu'à une vision d'entités logicielles coopérantes indépendamment déployables au sein d'un environnement d'exécution commun (comme le modèle Corba Component Model, CCM).

La définition de composants logiciels pour la construction de CBS est dicté par un ensemble de concepts et mécanismes formant un *modèle de composant*. Les modèles de composant se rejoignent sur un ensemble de caractéristiques communes, mais également diffèrent sur d'autres.

Notre objectif est de dégager une méthode générique adéquate à l'analyse des performances d'une interconnexion de composants définissant un Système Basé Composant. A cet effet, il est indispensable de connaître les caractéristiques prédominantes des modèles à composant, afin d'en tenir compte lors de la modélisation et analyse de tels systèmes.

Pour ce faire, nous nous proposons dans ce chapitre de faire une analyse et classification des différents modèles de composant. Par conséquent, nous commençons par dresser un ensemble de critères d'analyse, nous guidant dans la classification. Puis, nous faisons une synthèse des différents modèles existant dans la littérature.

2.2 Critères d'analyse

Les modèles à composant peuvent être classifiés selon plusieurs critères, à savoir :

1. Appartenance au domaine industriel ou uniquement au domaine académique de recherche : Ce critère met en évidence l'importance du modèle de composant et l'étendue de son utilisation dans le milieu industriel.
2. Développement en tant que modèle de composant ou en tant que langage ADL : En effet, la définition d'un langage de description d'architecture n'a pas les mêmes objectifs que la définition d'un modèle de composant. Cette dernière définition s'est prévalue d'un haut niveau d'abstraction définissant les concepts de composant.
3. Composition (interconnexion) de composants effectuée d'une manière statique, dite *hors ligne*, ou bien d'une manière dynamique, dite *en ligne*, se basant sur un mécanisme de contrats apparaissant à l'exécution. La composition hors ligne adresse particulièrement des systèmes de contrôle critiques,

devant respecter des contraintes strictes déduites du cahier des charges. Elle permet de prédire le comportement temporel de l'application avant sa mise en service et donc, de garantir à priori les propriétés requises. Quant à la composition en ligne, elle adresse des systèmes où l'instanciation des composants est imprévisible hors-ligne. Généralement, ces systèmes ne sont pas soumis à des contraintes fonctionnelles et extra-fonctionnelles fortes.

4. Prise en compte des propriétés non fonctionnelles, en plus des propriétés fonctionnelles.
5. Composition plate où les composants ne peuvent pas être composites sur plusieurs niveaux, en opposition à la composition hiérarchique permettant de structurer une application suivant différents niveaux d'abstraction (et ainsi structurer de même ses composants composites). Dans un modèle de composants hiérarchique, les composants composites servent à avoir une vue uniforme d'une application suivant différents niveaux d'abstraction. Ainsi, un composant composite représente une structure plus ou moins complexe de composants interconnectés, décrite par une configuration stockée dans un descripteur d'architecture et peut donc être utilisée comme un simple composant avec des interfaces requises et fournies bien définies. La récursivité s'arrête avec les composants primitifs, correspondant à des unités de traitement.

Comme nous nous intéressons à l'analyse de performances des systèmes à composant, nous adoptons dans la suite une classification faite par Fabien Dagnat [63], basée sur les deux premiers critères énoncés ci-dessus. Cette classification met en relief les modèles de composant pour lesquels une évaluation des performances est critique, notamment ceux liés au milieu industriel, et les modèles académiques permettant de faire évoluer les concepts de systèmes à composant et pouvant faire l'objet d'utilisations futures en industrie. Pour chaque catégorie, nous présentons une synthèse des modèles les plus connus. Les autres critères d'analyse sont également considérés implicitement tout au long de cette synthèse.

2.3 Analyse et classification

2.3.1 Modèles du monde industriel

Les modèles à composant appartenant au monde industriel ont été développés principalement pour deux objectifs majeurs :

- Permettre la conception d'applications pour le contrôle de processus industriels pour certains modèles, et
- Fournir une approche générique de développement pouvant être multi-langages et pour des environnements spécifiques ou non spécifiques, pour d'autres modèles.

Le premier objectif a été ciblé par des modèles tels que Koala [211] et IEC 61499 [127]. Le modèle Koala (C[K]omponent Organizer And Linking Assistent) a été créé en 2000 par Philips pour aider ses architectes et développeurs dans la création d'une famille de produits de télévision. Une année après, en 2001, le modèle IEC61499 fit son apparition grâce au groupe de travail TC65/WG6 qui l'ont introduit dans le cadre d'un projet de collaboration entre l'Université Nationale de Singapour et Yamatake Corporation. Il représente une extension du modèle IEC 61131.3, utilisé comme standard de programmation pour les contrôleurs programmables logiques PLCs (Programmable Logic Controllers) [218].

Le deuxième objectif a été plutôt visé par des modèles plus ou moins généraux. Parmi ces modèles, on trouve le modèle EJB (Enterperise JavaBeans) [203], développé en langage Java en 1997 par la société Sun Microsystems, et dédié au développement et à la gestion de composants serveurs au sein d'applications distribuées 3-tiers. Le modèle .NET, développé en 2000 par Microsoft, en est un autre exemple. Il

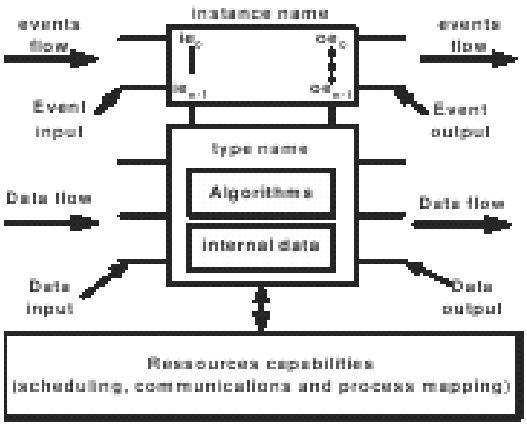


Figure 2.1: Bloc de Fonction IEC 61499

représente une plate-forme multi-langages pour composants, étroitement intégrée au Web, et dont l'infrastructure d'accueil des composants est fortement intégrée au système d'exploitation Windows. L'architecture .NET a fait suite aux architectures de composants COM (*Component object Model*), DCOM (*Distributed COM*) et COM+, publiées respectivement en 1992, 1996 et 1998 [56]. On discerne également le modèle CCM (*CORBA Component Model*) [169] qui est un modèle de composants indépendant des systèmes d'exploitation et des langages de programmation, défini en 2001 par l'OMG en vue d'ajouter des composants logiciels dans l'environnement CORBA existant (*Common Object Request Broker Architecture*) [164]. Principalement, CCM permet le déploiement de composants dans un environnement distribué (interconnexion de composants distribués sur différents serveurs).

2.3.1.1 Nature et constitution

En analysant les modèles introduits dans le domaine industriel, on discerne différentes vues d'un composant : selon l'objectif tracé, un composant peut être perçu comme une unité logicielle de conception tel que défini dans les modèles Koala et IEC61499, ou une unité d'exécution comme dans les modèles EJB, .NET ou encore CCM. La constitution du composant varie également entre des composants offrant uniquement des interfaces fonctionnelles, et d'autres composants combinant des interfaces fonctionnelles et/ou de contrôle.

Examinons la constitution d'un composant pour les modèles cités comme exemples.

Dans le modèle Koala, un composant est vu comme une unité hiérarchique de conception, développement et réutilisation. Il est doté de deux types d'interfaces : des interfaces fournies ou *provides* et des interfaces requises ou *requires*. Des services généraux tels que la gestion mémoire peuvent également être demandés à travers des interfaces *requires*. De plus, une interface spécifique étant l'*interface de diversité* (*diversity interface*) permet au composant de requérir des propriétés à remplir au moment de sa configuration, au lieu de les fournir comme paramètre à la création du composant. Il est également possible de définir des composants de base (primitifs) et des composants composites. Un exemple de composant Koala est donné ci-dessous.

Exemple 2.12 (Composant tuner d'une TV). *Un tuner est un dispositif matériel qui accepte un signal d'antenne sur une interface input, filtre une station particulière et l'envoie en output sur une fréquence*

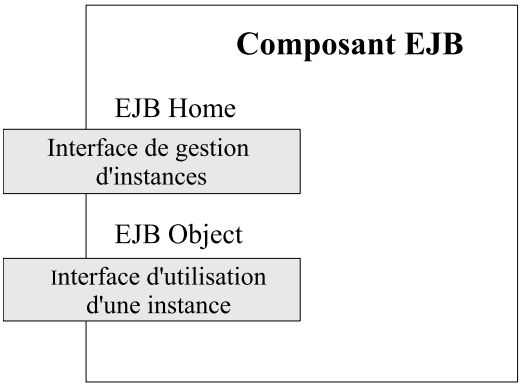


Figure 2.2: Composant EJB

intermédiaire. Ce signal alimente un processeur input (high-end input processor, HIP), qui produit une luminance décodée et des signaux de couleur, qui eux-mêmes alimentent un processeur output (high-end output processor, HOP) guidant l'écran TV. Chacun de ces dispositifs matériels est contrôlé par un pilote logiciel qui peut accéder au matériel via un bus série IC.

De même, dans le cas du modèle IEC61499, un composant est considéré comme une unité fonctionnelle réutilisable logicielle, appelée *Bloc de fonction* ou *Function Block, FB* (voir figure 2.1). Un bloc de fonction est composé d'une interface et d'une implantation [117]. L'interface est décrite par des ports d'événements input et output et des ports de données input et output. Les événements sont responsables de l'activation du bloc de fonction, alors que les données contiennent des informations valuées. Quant à l'implantation, elle consiste en un corps (données internes et algorithmes implantant les fonctionnalités du bloc) et une entête permettant la sélection des séquences d'algorithmes à exécuter lors de l'occurrence d'un événement input. Ce mécanisme de sélection est encodé sous-forme d'une machine à états dite le *diagramme de contrôle d'exécution* ou *Execution Control Chart, ECC*. Une application peut-être constituée de composants composites ou *sous-applications*, ce qui offre une hiérarchie de blocs de fonction. À l'encontre, le modèle EJB définit un composant (voir figure 2.2) comme un objet Java qui fournit un ensemble d'opérations métiers et un certain nombre d'opérations de contrôle. Ainsi, c'est une instance d'une classe java de type *EJBClass*, n'ayant qu'une interface serveur (aucune cliente), mais aussi implantant un ensemble d'interfaces spécifiques de contrôle, permettant notamment de gérer le cycle de vie du composant et de connaître ses services *métiers* et les méta-données associées. De plus, le modèle EJB est plat sans hiérarchie.

De manière similaire à un composant EJB, un composant .NET [140] est un objet constitué de plusieurs interfaces pouvant être génériques ou spécifiques au composant. Il reprend plusieurs aspects des composants COM+ [48].

Aussi, le modèle CCM est un modèle plat (sans hiérarchie), qui considère un composant comme une entité d'implantation, principalement décrite par des attributs et des interfaces typées dites *ports* : Les attributs sont des valeurs nommées permettant entre autres de configurer le composant à l'aide d'opérations d'accès (get) et de modification (set). Les interfaces (figure 1.3), décrites en langage CORBA IDL 3.0 (*Interface Definition Language*), se caractérisent par un mode de communication et sont de quatre sortes : les *facettes* et les *réceptacles* en mode de communication synchrone, et les *sources* et les *puits* en mode basé événement ou mode asynchrone. Une *facette* est une interface de type *provides*, déclarant un

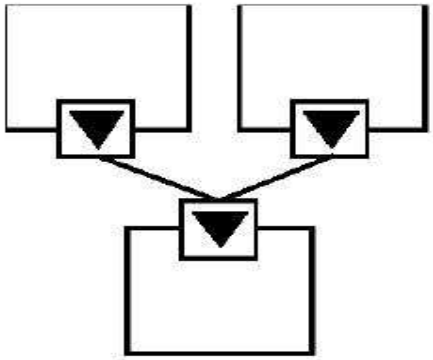


Figure 2.3: Un exemple de configuration Koala

ensemble de services (opérations) offerts et acceptant des invocations point à point d'autres composants sur ces opérations. Un *réceptacle* est une interface de type *uses*, déclarant les dépendances ou services requis par le composant. Une *source d'événements* est une interface de type *emit* ou *publishes*, traduisant un envoi de messages asynchrone 1-vers-1 ou 1-vers-n (cas diffusion d'un événement à plusieurs). Enfin, un *puits d'événements*, de type *consumes*, reçoit des notifications d'événements à partir d'une ou plusieurs sources. Il est à noter aussi que la spécification CCM ne fait aucune distinction entre les interfaces fonctionnelles et les interfaces de contrôle.

2.3.1.2 Interactions entre composants

Les interactions entre composants diffèrent et peuvent être visibles depuis l'architecture logicielle selon le modèle de composant.

Dans le cas du modèle Koala, les composants sont interconnectés d'une manière visible par des interactions de type appel de méthode et de type événements. Pour construire un produit, les interfaces de composants sont reliées par des *connecteurs*. Trois types de connecteurs sont utilisés : les liaisons ou *bindings*, les *glue codes* et les switchs. Un *binding* connecte une interface *requires* d'un composant à une interface *provides* du même type d'un autre composant, en contraste d'un *glue code* qui connecte ce type d'interface à une interface *provides* d'un type différent d'un autre composant. Quant au *switch*, c'est un glue code spécial permettant de passer d'une liaison à une autre. Il choisit dynamiquement entre des interfaces *provides* de différents composants. Les liaisons entre interfaces sont déployées de manière statique au moment de la configuration d'une hiérarchie de composants. On ne peut donc refaire une nouvelle composition d'instances de composants. L'interconnexion des composants et connecteurs donne lieu à une *configuration*. Un exemple de configuration est donné ci-après.

Exemple 2.13. *Le code suivant décrit une partie de la configuration Koala de la figure 2.3.*

```
interface Ituner
{   void SetFrequency(int f);
    int GetFrequency(void);
    }
component CtunerDriver
{   provides ITuner ptun;
             IInit pini;
```

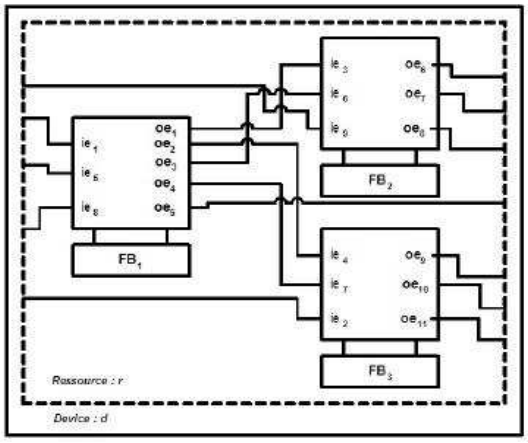


Figure 2.4: Application de contrôle IEC 61499

```
    requires II2c ri2c;
    }
component CtvPlatform
{   provides IProgram pprg;
    requires II2c slow, fast;
    contains
        component CFrontEnd cfre;
        component CTunerDriver ctun;
    connects
        pprg = cfre.pprg;
        cfre.rtun = ctun.ptun;
        ctun.ri2c = fast;
    }
```

Quant au modèle IEC 61499, les blocs de fonction sont assemblés en un *réseau de blocs de fonction* pour former une application de contrôle. Les noeuds sont des blocs de fonction ou des sous-applications et les branches sont les connexions. Les composants blocs de fonction sont interconnectés par deux types de connexions : des connexions événement et des connexions données. Chaque port input d'un bloc de fonction est connecté à un autre port output par un canal, qu'il s'agisse de ports d'événement ou de données. Le transfert de données entre blocs de fonction peut se faire en deux modes : Un mode de transfert unidirectionnel, donné par le modèle informatif publisher/subscriber, et un mode bidirectionnel, consistant en le modèle client/serveur.

Un exemple d'application IEC 61499 est donné ci-après.

Exemple 2.14. *La figure 2.4 montre une application de contrôle située dans une ressource d'un périphérique [116]. Elle est constituée de trois FBs. Chaque FB implante plusieurs fonctionnalités élémentaires (une pour chaque événement en entrée).*

Pour sa part, le modèle EJB (comme le modèle .NET) ne fournit pas de connexions visibles entre composants. Les interactions sont de type invocation de méthode (service). Entre un client et un com-

posant EJB, les communications se font à travers son conteneur. En effet, le client ne s'adresse jamais directement au composant, mais plutôt au conteneur qui intercepte les messages destinés au composant pour lui déléguer ensuite la tâche lorsque cela est pertinent. Pour cela, des interfaces entre un client et le conteneur d'un composant d'une part, et entre le composant et son conteneur d'autre part, sont définies précisément par des contrats (*EJB contracts*) .

De même, l'assemblage de composants CCM se fait d'une manière explicite sur la base de leurs interfaces, formant ainsi une application (*assembly*) CCM. Des interactions de type appel de méthode ou événements sont possibles entre les composants CCM. Les communications basées événements suivent le modèle *push Publish/Subscribe* [169], compatible avec le service de notification CORBA [168].

2.3.1.3 Implantation

Certains modèles de composant nécessitent un contexte d'exécution dit *conteneur* ou *container*. Les conteneurs implantent la gestion du cycle de vie (création, destruction, modification) des composants et sont en charge de leur fournir un environnement d'exécution comprenant des services non-fonctionnels de type persistance, gestion des transactions, gestion des exceptions et des communications synchrones (tel que RMI) et asynchrones (tel que JMS), service de nommage (comme JNDI pour le cas de composants EJB), service de sécurité, tolérance aux fautes, service de notification d'événement et service d'horloge. Ils sont également responsables de gérer les accès aux composants qu'ils accueillent, en interceptant les messages entrants et sortants de ces composants. L'exécution du conteneur est elle-même effectuée dans le cadre d'un processus serveur dit *le serveur de composants*.

Ainsi, les composants EJB sont hébergés dans un *conteneur*. Un conteneur EJB est une entité exécutable au-dessus d'une machine virtuelle Java sur un serveur d'application physique dit *EJB server*, qui définit la façon d'utiliser des services non-fonctionnels pour un contexte applicatif donné. Un serveur d'application est un intergiciel supportant l'exécution d'un ensemble de conteneurs. Le conteneur peut gérer plusieurs types d'EJB. Pour développer et exécuter des applications, plusieurs outils et environnements pour composants EJB existent, sous-forme de plateformes Java.

D'une manière similaire, les applications .NET sont exécutées dans une forme de conteneur dit *Common Language Runtime, CLR*. Le CLR joue le rôle d'un agent qui gère le code au moment de l'exécution. Il fournit des services essentiels comme la gestion de la mémoire et cycle de vie des composants, la gestion des threads, la gestion des transactions et l'accès distant. Il applique également une stricte sécurité des types et d'autres formes d'exactitude du code qui promeuvent un code sécurisé et robuste. Les applications .NET sont développées et exécutées dans un environnement dit le *.NET Framework*, comportant l'environnement d'exécution CLR et la bibliothèque de classes .NET Framework.

Quant au modèle CCM, un composant, comme dans la norme EJB, est localisé et exécuté dans un *conteneur* (voir figure 2.5) qui lui offre un environnement d'exécution (çàd un espace mémoire et un flot d'exécution). Il lui fournit également l'accès aux services non-fonctionnels qui lui sont nécessaires, l'initialisation et le contrôle du cycle de vie des instances de composants qu'il gère, et leur connexion à d'autres composants et aux services communs de middleware. Un conteneur CCM est spécifique à un type de composant. Pour implanter effectivement une application CCM, des outils de développement et d'exécution existent tels que *K2-CCM, Cadena, CCM container framework* et *CIF (Component Implementation Framework)*.

Le concept de conteneur se retrouve également dans le modèle IEC 61499 comme un support d'exécution, défini par un réseau de périphériques ou *devices*, où un périphérique est un conteneur de ressources, composé d'une unité de traitement, de capteurs, de déclencheurs et d'interfaces réseau. Pour

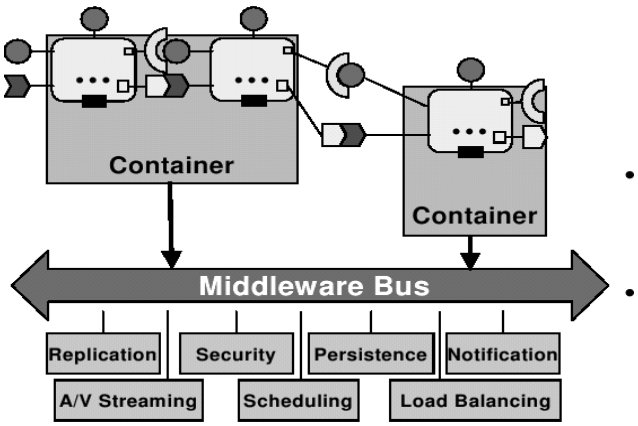


Figure 2.5: Conteneur dans le modèle CCM

s'exécuter, les instances de blocs de fonction sont allouées à des ressources contenues dans le périphérique associé. Un outil de développement appelé *FBDK, Function Block Development Kit* [13], puis un autre plus récent, l'outil ICT (*IEC-compliant CASE tool*) [208] ont été développés afin de permettre la construction d'applications de contrôle suivant le modèle IEC61499.

Par ailleurs, contrairement aux modèles décrits précédemment, la notion de conteneur n'apparaît pas dans le modèle Koala. Un outil de construction d'applications Koala a été développé. Il permet de lire toutes les descriptions des composants et interfaces, et instancie un composant au plus haut niveau de la hiérarchie. Puis, tous les sous-composants sont instanciés récursivement jusqu'à l'obtention d'un graphe direct de modules, d'interfaces et de liaisons.

2.3.2 Modèles du monde académique

Le domaine académique a également développé plusieurs modèles de composant. On retrouve par exemple le modèle PECOS développé en 2001 et les modèles FRACTAL et ArchJava en 2002.

PECOS [163; 219] a été développé suite au projet PECOS lancé par la commission européenne et le gouvernement suisse. L'objectif du projet était de permettre le développement de logiciels basés composant, spécifiques aux systèmes embarqués, en fournissant un environnement supportant la spécification, la composition, la vérification de configuration et le déploiement des systèmes embarqués construits à partir de composants logiciels. Il a été défini plus particulièrement pour exprimer l'architecture des dispositifs de terrain du type *field device*. Un *field device* est un système embarqué réactif, utilisant des capteurs pour rassembler continuellement des données, telles que la température, la pression, ...etc. Ces données sont analysées, puis le field device réagit en incitant des déclencheurs, des valves ou des moteurs.

FRACTAL [43] est un modèle de composant général, développé au sein du consortium ObjectWeb par France Telecom R&D et l'INRIA. Il vise à implanter, déployer, gérer et configurer dynamiquement des systèmes logiciels complexes, incluant des systèmes d'exploitation et des intergiciels ou middlewares. Il est utilisé dans divers projets de recherche allant de la mise en place de systèmes d'exploitation spécialisés pour des plate-formes embarquées [78; 4] jusqu'à la mise en place de systèmes autonomes pour des serveurs d'applications d'entreprises [35], en passant par des systèmes multimédia pour des applications mobiles [124].

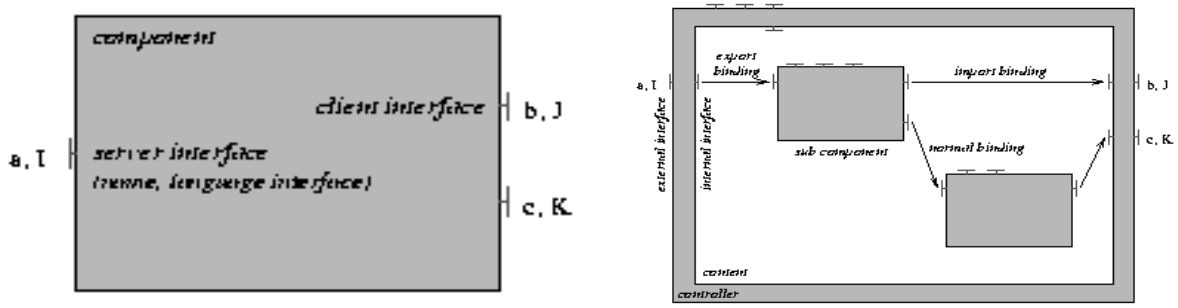


Figure 2.6: Vues externes (à gauche) et internes (à droite) d'un composant FRACTAL

Quant à ArchJava, c'est plutôt un ADL, créé principalement par J.Aldrich et C.Chambers [7] à l'université Carnegie Mellon de Washington, comme extension du langage Java, permettant aux programmeurs de décrire une architecture logicielle avec du code Java. Il constitue l'une des premières propositions qui visent à définir un langage à composants.

2.3.2.1 Nature et constitution

De même que pour les modèles de composant du domaine industriel, les modèles varient entre entité d'exécution et entité logicielle de conception ayant une certaine constitution.

Dans le modèle PECOS, un composant désigne un élément logiciel ayant un nom, un certain nombre de *ports*, des groupes ou paquets de propriétés (*property bundles*) et un comportement. Les ports représentent les données pouvant être partagées entre le composant et d'autres. Plus précisément, un port est une variable partagée, caractérisée par un nom, un type, une plage de valeurs et une direction (in, out ou inout). Les groupes de propriétés caractérisent le composant. Le comportement d'un composant consiste en une procédure qui lit et écrit des données disponibles sur les ports, et produit éventuellement des effets sur le monde extérieur. PECOS permet également de définir des composants composites, construits comme une hiérarchie de sous-composants.

Pour sa part, un composant FRACTAL est une entité *en exécution* interagissant avec son environnement (i.e. avec d'autres composants) à travers des *interfaces* bien définies pouvant être de type *client* ou *serveur* (figure 1.2). En plus de la nature client ou serveur, des propriétés de contingence (obligatoire ou optionnelle) et de cardinalité (acceptation de connexions simples ou multiples) sont attribuées aux interfaces. Un composant FRACTAL possède deux parties : une partie *contenu* consistant en un nombre fini de *sous-composants* permettant de construire des composites; et une partie de *contrôle*, appelée *membrane* ou *contrôleur* permettant de séparer l'implantation du métier du composant de sa partie de contrôle. La membrane expose des interfaces *de contrôle*, supportant l'introspection (monitoring), l'intercession et la reconfiguration de propriétés internes au composant, telles que la suspension et la reprise d'activités d'un sous-composant... Le modèle FRACTAL est indépendant du langage d'implantation. A l'heure actuelle, il existe plusieurs implantations de ce modèle dans divers langages, visant des contextes applicatifs différents. Plus de détails sont donnés dans le chapitre 8.

Á l'encontre, un composant ArchJava est une unité d'encapsulation et de traitement, instance d'une classe de composants. Il est décrit par un ensemble de *ports* définissant une combinaison de services (méthodes au sens Java classique) et pouvant être de trois types : les méthodes *provides fournies* par le composant, les méthodes *requires requises* par le composant pour fonctionner et les méthodes *broadcasts*

(*diffusion*) qui étendent les méthodes *requires*, en transmettant une invocation de méthode à plusieurs composants. Un composant ArchJava peut posséder des sous-composants le rendant ainsi composite.

Exemple 2.15. *Voici un exemple de déclaration de composant*

```
component class Serveur
{   private String message = new String("çà marche\n et Bonjour le
    monde !!");
    protected Serveur() { int c=1; }
    protected String getMessage() {return message; }
    public port s_conn
    {   provides String get_message() {return this.getMessage(); }
        }
    }
```

Ce composant offre un service get_message *au sein de l'interface* s_conn.

2.3.2.2 Interactions entre composants

Différents modes d'interactions sont utilisés dans les modèles de composant.

Les composants PECOS sont interconnectés de manière hiérarchique afin de construire le logiciel d'un *field device*. La connexion de composants se fait en connectant les ports de données qui représentent la même variable partagée. Elle est faite à travers des *connecteurs*. Un connecteur spécifie une relation de partage de données entre ports.

De leurs côté, les composants FRACTAL sont interconnectés par des *liaisons ou bindings*. La spécification FRACTAL définit des liaisons *primitives* et des liaisons *composites* (voir figure 2.6, à droite). Une liaison primitive est une connexion directe entre une interface cliente et une interface serveur, conforme à la sémantique définie par le langage d'implantation des composants. Une liaison composite désigne par contre un chemin de communication entre un nombre arbitraire d'interfaces de composants. Elle représente elle-même un composant FRACTAL construit en combinant des liaisons primitives et des composants ordinaires. Les interactions possibles entre composant sont de type invocation de service. Selon la plateforme d'implantation, ces interactions sont implantés par un mode de communication synchrone (comme dans l'implantation de référence *Julia* [41; 42]) ou asynchrone (comme dans l'implantation *Fractive* [22]). La spécification d'une architecture de composants FRACTAL se fait à l'aide du langage FRACTAL ADL (FRACTAL Architecture Descrition Language).

Quant au modèle ArchJava, la communication entre composants se fait par appel de méthodes. Elle est réalisée par la *connexion* des ports associés aux composants communiquants : Les méthodes fournies par un port de composant sont liées aux méthodes requises par un autre composant. D'autres types de connexions peuvent aussi être implantés en ArchJava : outre l'invocation classique de méthode, une abstraction a été proposée pour mettre en oeuvre des *connecteurs* supportant diverses sémantiques de communication, telle que la communication asynchrone [8]. Un connecteur sert à encapsuler le code de communication qui met en oeuvre les communications induites par la connexion. La connexion ou composition des composants se fait de manière hiérarchique donnant lieu à une *configuration* d'architecture logicielle. Celle-ci peut être modifiée dynamiquement (création ou destruction de composants, connexion de composants, ...).

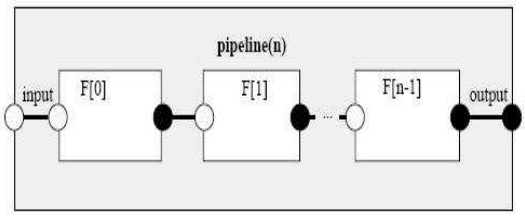


Figure 2.7: Composant composite Darwin

2.3.3 Langages de description d'architectures (ADLs)

Plusieurs modèles de composant ont été implicitement définis à travers l'introduction de langages de description d'architecture (ADLs). La plupart de ces langages servaient à décrire des architectures de systèmes, alors que la notion de modèle à composant n'avait pas encore été développée. Plus tard, les chercheurs ont rassemblé les caractéristiques et mécanismes nécessaires et ont défini des modèles de composant plus ou moins spécifiques.

Afin d'examiner les caractéristiques des ADLs, nous présentons une analyse des ADLs les plus répandus dans la littérature, à savoir Darwin (1994), MétaH (1996), Wright (1997), SOFA (1998) et AADL (2001). Darwin a été proposé par le groupe *Distributed Software Engineering Group* de l'Imperial College de Londres et développé par Magee et Kramer [135; 137]. Il est utilisé pour définir des compositions hiérarchiques de composants interconnectés, et est dédié à la construction d'applications réparties [136]. MétaH [214], quand à lui, a été proposé par l'institut de Technologie de Honeywell depuis 1996. Il est dédié à la description d'architectures logicielles, définies dans le cadre de systèmes avioniques et d'applications de guidage, de navigation et de contrôle [30]. Ce langage a été étendu en 2001 vers le langage AADL (Architecture Analysis and Design Language) [213; 188], langage normalisé par la société SAE (Society of Automative Engineers). AADL était d'abord dédié à la modélisation des systèmes avioniques, puis a couvert tous les systèmes embarqués temps-réel. Quant à Wright [9; 10], il a été développé par David Garlan à l'université Carnegie Mellon de Pittsburg. Il permet la description et vérification formelle d'architectures logicielles, en utilisant les notations de CSP (Client Server Protocol) [106]. Enfin, SOFA ou SOFtware Appliances [179; 180] est un modèle et une implantation de composants distribués, défini en 1998 dans le cadre d'un projet visant à fournir une plateforme pour les composants logiciels.

2.3.3.1 Nature et constitution

La notion de composant est perçue pour la plupart des ADLs comme une unité de traitement exécutable. Toutefois, elle peut être considérée comme une entité abstraite tels que perçu par le langage Wright.

En effet, un composant Darwin représente une unité de traitement de l'information, qui possède une interface composée d'une collection de services fournis ou requis. Il peut être composite ou primitif (voir l'exemple ci-après). Chaque service offert ou requis correspond à un port de connexion, associé à un mode de communication précis (synchrone ou asynchrone).

Exemple 2.16. *L'exemple de la figure 2.7 définit un pipeline de longueur variable, constitué d'instances de composant filtre (filter), telles que l'entrée de chaque instance F[k+1] est liée à la sortie de son prédecesseur F[k].*

```
component pipeline(int n) { provide output;
```

```
require input;

array F[n]: filter;
forall k:0..n-1
{ inst F[k] @ k+1;
  when k < n-1;
  bind F[k+1].input -- F[k].output;
  }
bind
  F[0].input -- input;
  output -- F[n-1].output;
}
```

Dans le langage MétaH, deux types de composants sont définis :

- Un composant *Process* qui est une unité exécutable de traitement d'information, ayant son propre flot d'exécution. Il est décrit à l'aide d'un type donné par une interface listant l'ensemble des propriétés visibles et d'un ensemble d'implantations. Les propriétés caractérisant un composant peuvent être des ports d'entrée (*in*) et de sortie (*out*), des signaux d'événements à émettre ou recevoir, des packages (collections de sous-programmes écrits en langage Ada) et des moniteurs de section critique (permettant la synchronisation et l'exclusion mutuelle). Une implantation sert uniquement à fixer des valeurs d'initialisation comme la fréquence du Process à l'exécution, ou la localisation du code source utilisé.
- Un composant matériel qui rend compte de l'infrastructure matérielle d'exécution d'une architecture logicielle. On trouve plusieurs entités : les processeurs, les mémoires (*Memory*), les *devices*, les canaux (*Channel*) permettant de connecter les ports des *processeurs* et des *devices*, et les *System* qui représentent l'infrastructure matérielle d'une application de contrôle.

Par contre, dans Wright, un composant est une entité élémentaire de traitement abstraite (qui n'a pas d'implantation associée), décrit par une partie *interface* et une partie *calcul*. L'interface consiste en un nombre fini de ports d'interaction, sans sémantique précise, mais doté d'une description formelle définissant le comportement. La sémantique est plutôt déduite du comportement du composant. La partie calcul, dite *computation*, décrit formellement le comportement global du composant, çàd ce que le composant fait réellement. Elle met en oeuvre les interactions décrites par les ports, et spécifie comment ils sont connectés ou *attachés* ensemble pour former un tout.

Par ailleurs, dans le langage SOFA, un composant est une entité hiérarchique constituée de deux parties : un *frame* et une architecture. Le frame est une "boîte noire" exposant des interfaces fournies et requises, ainsi que des propriétés éventuelles pour paramétrer le composant. Une architecture décrit la structure du composant correspondant au frame, en instanciant ses sous-composants directs, et en spécifiant les interconnexions entre sous-composants par liaisons d'interfaces. Si le composant ne contient aucun sous-composant, l'architecture est dite *primitive*, et sa structure/implantation est fournie dans un langage sous-jacent.

Exemple 2.17 (Exemple de composant SOFA). *L'exemple de la figure 2.8 illustre une application SOFA constituée de plusieurs composants dont le composant noté* DB. *Celui-ci est une instance du template* <Database, DatabaseV2>. *Il est lui-même composé de deux sous-composants :* Transm *et* Local. *Le composant* Local *offre les interfaces d et ds, alors qu'elle requiert les interfaces lg, da et tr. L'interface*

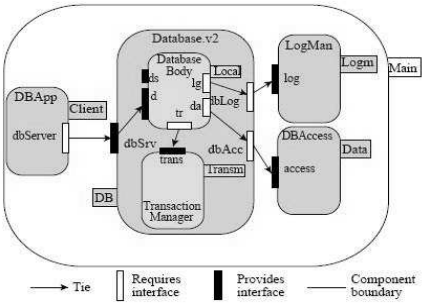


Figure 2.8: Un exemple de composant SOFA

dbSrv *de* DB *est déléguée à l'interface d de* Local*, tandis que l'interface lg de* Local *est substituée à dbLog de* Local*. L'interface requise* tr *est liée à l'interface offerte* trans *du composant* Transm*.*
Voici une portion de la spécification (écrite en langage CDL) de l'architecture :

```
interface ICfgDatabase
{   int GetTrModel();
    void SetTrModel(int model);
    };
frame DatabaseBody
{   provides:   IDBServer d;
                ICfgDatabase ds;
    requires:   IDatabaseAccess da;
                ILogging lg;
                ITransaction tr;
    };
frame Database
{   provides: IDBServer dbSrv;
    requires: IDatabaseAccess dbAcc;
    ILogging dbLog;
    };
architecture DatabaseV2 implements Database
{   inst TransactionManager Transm;
    inst DatabaseBody Local;
    bind Local:tr to Transm:trans using CSProcCall;
    exempt Local.ds;
    subsume Local:lg to dbLog using CSProcCall;
    subsume Local:da to dbAcc using CSProcCall;
    delegate dbSrv to Local:d using CSProcCall;
    };
```

Enfin, un composant AADL est un composant hiérarchique, constitué d'une partie interface décrivant les propriétés et les fonctionnalités du composant vu de l'extérieur, et une partie implantation donnant les éléments (sous-clauses) de la structure interne du composant. Les propriétés caractérisent le composant

telles que les propriétés temps-réel des tâches (période, durée d'exécution, ...). Quant aux fonctionnalités (*features*), elles comprennent des ports de communication pour la transmission d'informations entre composants, des sous-programmes d'interface décrivant des appels de méthode et des accès (fournis ou requis) aux sous-composants exprimant le partage d'un sous-composant entre plusieurs l'englobant. Les ports de communication peuvent être des ports d'événements (*event ports*) pour la transmission de signaux, des ports de données (*data ports*) pour l'échange de données et/ou des ports d'événement/donnée (*event data ports*) qui envoient des données tout en générant un événement à la réception. Par ailleurs, des caractéristiques non fonctionnelles telles que la performance et la fiabilité peuvent être représentées sous forme de valeurs de propriétés définies.

2.3.3.2 Interactions entre composants

Dans la plupart des langages ADLs, les composants interagissent en accédant à des services fournis entre eux ou par notification d'événements.

Ainsi, Darwin supporte les deux modes de communication synchrone et asynchrone. Chaque interconnexion est considérée comme un objet de communication. Les interconnexions peuvent être réalisées statiquement ou dynamiquement, donnant lieu à des architectures statiques et dynamiques. Une architecture statique est décrite par des instances de composants pouvant être interconnectés de manière hiérarchique et leurs interconnexions. Une architecture dynamique décrit une structure qui évolue.

Dans cette idée, les composants MétaH communiquent également par les deux modes de communication et même par d'autres modes, à savoir le partage. Les composants dits *process* sont connectés par ports de données ou événements et par partage de packages et moniteurs. Quant aux composants matériels, leur interconnexion se fait soit en reliant les *processeurs* aux *devices* par des *channels*, ou bien en reliant les *memories* aux *processeurs* pour décrire leur partage entre ces *processeurs*. L'interconnexion de composants permet de construire des composants composites pouvant être des *Macros* (description d'une collection de Process et de leurs connexions), des *Modes* qui sont des *Macros* spécifiques, ou bien des *Applications* qui sont des entités matérielles/logicielles agrégeant la description d'une application de contrôle.

Un moyen plus général pour connecter les composants est utilisé dans Wright, permettant ainsi d'exprimer tout mode de communication possible : cela consiste à utiliser la notion de *connecteur*. Un connecteur est un élément d'interconnexion qui lie les ports des composants. Comme un composant, le connecteur est décrit par une interface définissant un ensemble de rôles spécifiant le comportement d'un participant dans l'interaction, le comportement associé à chaque rôle et l'ensemble des synchronisations et échanges de messages entre rôles, La connexion de composants interagissant donne lieu à une *configuration*. Une configuration est une description statique d'architecture à différents niveaux d'agrégation, spécifiant les *instances* de composants présents et leurs interconnexions ou *attachements* par des instances de connecteurs. Wright permet également de décrire l'évolution dynamique des systèmes. Ceci se fait par l'intermédiaire de *configurateurs*. Un *configurateur* est une description de reconfiguration dynamique qui est déclenchée lorsque la présence d'une condition bien précise est observée.

Exemple 2.18. *Voici un exemple de spécification simple Wright.*

```
System simple_cs
    Component client =
        port send port send-request = [behavioral spec]
        spec = [behavioral spec]
    Component server =
```

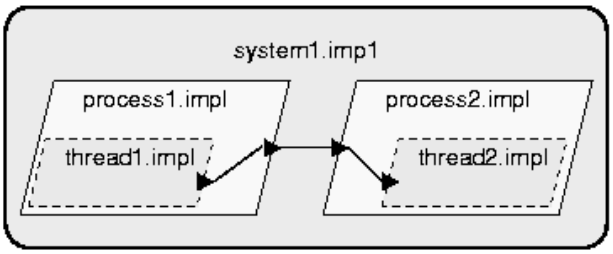


Figure 2.9: Connexion AADL entre deux threads

```
        port receive port receive-request= [behavioral spec]
        spec = [behavioral spec]
Connector rpc =
    role caller = (request!x -> result?x ->caller) ^ STOP
    role callee = (invoke?x  -> return!x -> callee) [] STOP
    glue = (caller.request?x -> callee.invoke!x -> callee.return?x
            -> callee.result!x -> glue) [] STOP
Instances
    s : server
    c : client
    r : rpc
Attachments :
    client.send-request as rpc.caller
    server.receive-request as rpc.callee
end simple_cs.
```

Dans le cas de SOFA, plusieurs styles de communication sont supportés, notamment l'invocation de méthode, l'échange de messages, le streaming (flux) et l'utilisation de mémoire partagée distribuée. Les composants sont interconnectés d'une manière hiérarchique par quatre types de liaisons [106]: le *binding* qui attache une interface requise d'un composant à une interface fournie d'un deuxième composant; le *delegating* qui délègue une interface fournie de composant à une autre fournie d'un sous-composant; le *subsuming* englobant une interface requise d'un sous-composant à une interface requise de composant, et l'*exempting* qui exempte une interface d'un sous-composant de n'importe quelle liaison. Une liaison de non exemption est réalisée par l'intermédiaire d'un connecteur. L'utilisation de connecteur également permet d'adopter tout type d'interaction.

Enfin, dans le cas du langage AADL, les connexions relient soit les composants périphériques aux composant logiciels, ou bien les composants logiciels et périphériques à la plateforme d'exécution. Les connexions sont orientées et de type appel de service ou basé événement.

Exemple 2.19. *Le schéma de la figure 2.9 montre un système contenant deux process, eux-même contenant chacun un thread. Une succession de ports, représentés par les triangles, et de connexions, représentées par les lignes, établit une liaison entre les deux threads.*

La notion de *connecteur* est également utilisée dans AADL mais au second plan. Elle n'est pas identifiée par une construction syntaxique unique, mais est principalement modélisée par des connexions

AADL ayant une sémantique restreinte. L'assemblage d'un ensemble de composants donne lieu à une *configuration* d'architecture, qui est essentiellement statique. Toutefois, un certain degré de dynamisme peut être exprimé par l'intermédiaire de la notion de *mode opérationnel AADL*. Un *mode* définit une configuration de fonctionnement pour les composants. Un mode opérationnel est un état observable opérationnel d'un composant AADL. Les modes AADL sont définis dans les implantations des composants. Les transitions entre modes modélisent le comportement dynamique opérationnel de l'architecture et sont déclenchées par des événements arrivant sur les ports événements.

2.4 Synthèse et Comparaison

L'analyse des différents modèles de composant nous a permis d'identifier un ensemble de concepts qui ont été utilisés dans le contexte de l'approche de programmation par composants, à savoir :

1. Les types d'interfaces supportées : Invocation de méthode, événement, autre, desquels découlent les types d'interaction synchrone et asynchrone.
2. Composition hiérarchique ou plate : possibilité de composer des composants d'une manière hiérarchique.
3. Utilisation de connecteurs, éléments qui assurent la liaison entre les composants.
4. Séparation des aspects fonctionnels et de contrôle via la notion de conteneur ou contrôleur : certains modèles font plus ou moins une séparation des aspects de contrôle à l'aide des conteneurs qui offrent un environnement d'exécution. D'autres proposent une distinction de ces aspects au niveau composant par des contrôleurs.
5. Multi-langages : support d'implantation dans des langages différents.

Dans la suite, pour des fins de comparaison, nous dressons des tableaux récapitulatifs des propriétés des modèles de composant, notamment les propriétés générales, de composant, d'interaction et de possibilité d'analyse.

Modèle	Année	Domaine d'application	Lang. ADL	Atelier de dévelop.	Langage	Utilis. indust.	Limites
Darwin	1994	Applications réparties	Oui	Regis, Conic	C++	Non	Absence de partage
MétaH	1996	Systèmes avioniques, de guidage, navigation, contrôle	Oui	Oui	-	Oui	Pouvoir d'expression limité. Non possibilité de définir de nouveaux attributs
Wright	1997	Analyse de protocoles de communication. Divers	Oui	Non	pas d'implant.	-	Aucun environnement n'intègre Wright. Structure statique
EJB	1997	Applications diverses, réparties	Non	Oui	Java	Oui	Pas de description d'assemblage. Pas d'hiérarchie. Ens. services non fonctionnels fixe
SOFA	1998	Applications de composants distribués	Oui	SOFA/ DCUP	Java	-	Pas de partage
.Net	2000	Divers	Non	Oui	Tout	Oui	Pas de description d'assemblage et d'interfaces requises
Koala	2000	Logiciels embarqués dans les produits TV	Oui	Oui	C	Oui	Pas de partage
CCM	2001	Applications réparties	Non	Oui	Tout langage	Oui	Ensemble de services fixe
AADL	2001	Systèmes embarqués temps réels	Oui	-	-	Oui	-
IEC61499	2001	Applications de contrôle distribuées	Non	ICT, FDBK	Java, Delphi	Oui	Pas de partage
PECOS	2001	Systèmes embarqués	Non	CoCo et outils	Java	Non	Types limités de communication entre composants
ArchJava	2002	Divers	Oui	Oui	Java	-	Pas de notations ADL formelles. Pas de support de propriétés non fonctionnelles.
Fractal	2002	Divers	Oui	Oui	Java, C++, SmallTalk	Non	Modèle académique non encore répandu en industrie

Table 2.1: Comparaison entre les modèles de composants : Propriétés générales

Modèle	Interfaces	Propriétés non fonctionnelles	Conteneur/ contrôleur	Partage	Distribution
Darwin	Services fournis et requis	Non	-	Non	Oui
MétaH	Ports in/out, ports événements	Sous-forme d'attributs	-	Oui	-
Wright	Ports avec sémantique précisée par le comportement	-	-	Non	-
EJB	Services métiers	Cycle de vie, méta-données, sécurité, persistance, transaction, notification, ...	Conteneur	Oui	Oui
SOFA	Interfaces fournies et requises	Oui	-	Non	Oui
.Net	Interfaces génériques et spécifiques	Gestion de cycle de vie, transaction,...	-	Oui	-
Koala	Interfaces requises et fournies	Sous forme de propriétés	-	Non	Non
CCM	Facette/réceptacle, puits/source d'événements	Oui	Conteneur	-	-
AADL	Ports de données, événements, événements/données	Sous forme de propriétés	-	Oui	-
IEC61499	Ports de données et d'événements	Services de communication, gestion d'application,...	Device	Non	Non
PECOS	Ports de données partagées	Certaines propriétés dont les propriétés temporelles	-	-	-
ArchJava	Ports fournis, requis et de diffusion	Non	Non	Non	Oui
Fractal	Interfaces serveur/client, interfaces de contrôle	Diverses	Contrôleur	Oui	Oui

Table 2.2: Comparaison entre les modèles de composants : Propriétés de composant

Modèle	Styles de communication	Utilisation Connecteur	Composition hiérarchique	Reconfiguration dynamique
Darwin	Synchrone, asynchrone	Non	Oui	Oui
MétaH	Appel méthode, événements	Oui	Oui	
Wright	Tout style	Oui	Oui	Quelque dynamicité
EJB	Appel de méthode, échange de messages	Oui	Oui	Oui
SOFA	Appel méthode, streaming, échange messages, partage de données	Oui	Oui	Oui
.Net	Appel de méthode	Non		
Koala	Appel de méthode, événements réalisés par port requis	Oui	Oui	Quelque dynamicité
CCM	Appel de méthode, notification d'événements	Non	Non	Non
AADL	Appel de méthode, notification d'événements	Oui	Oui	Oui
IEC61499	Appel de méthode, notification d'événements	Non	Oui	Non
PECOS	Partage de données	Oui	-	Non
ArchJava	Synchrone, diffusion	Oui	Oui	Oui
Fractal	Synchrone, asynchrone selon l'implantation	Non	Oui	Oui

Table 2.3: Comparaison entre les modèles de composants : Propriétés d'interaction

Modèle	Formalisme	Type d'analyse	Outils d'analyse
Darwin	LTS, FSP, $\pi-$calcul, MSC	Propriétés qualitatives (sûreté, vivacité)	LTSA, SAA
MétaH	Pas de formalisme. Notation MétaH	Ordonnancement, fiabilité	Oui
Wright	CSP	Propriétés qualitatives (absence interblocage, consistence, complétude)	Wright Toolset (modélisation et vérification)
EJB	Modèle quantitatif de performance basé sur les profils	Prédiction de performance	-
SOFA	Traces, behavior protocols	Détection d'erreurs, correction de la conception	Support SOFA pour les protocoles de comportement
.Net	Asml basé sur les systèmes de transition, contrats	Vérification dynamique (runtime verification)	Langage Asml et outils
Koala	MSC	Estimation de propriétés non fonctionnelles	-
CCM	Promela (automates finis)	Model checking (vérification de propriétés), séquencement d'événements, sûreté	Cadena, dSPin
AADL	GSPN, timed automata, error dependability model	Dépendabilité, fiabilité, disponibilité	OSATE
IEC61499	Traces, automates temporisés, Net condition/event systems (PN)	Analyse d'ordonnançabilité, vérification de délais, validation du comportement temporel	-
PECOS	PN, Time PN	Recherche de conflits, ordonnancement pour satisfaction de contraintes	-
ArchJava	-	-	-
Fractal	LTS, réseaux d'automates communicants	Propriétés qualitatives	Oui

Table 2.4: Comparaison entre les modèles de composants : Propriétés d'analyse

2.5 Conclusion

Nous avons présenté dans ce chapitre différentes propositions de modèles de composant, dans l'objectif de ressortir les propriétés communes à prendre en considération lors de l'analyse d'un CBS. Nous citons notamment les types d'interfaces exposées par les composants, les sémantiques d'interactions possibles, l'hiérarchie de composants, l'utilisation de connecteur ou non dans les communications entre composants et le support de propriétés ou services non fonctionnels à travers un contrôleur ou un conteneur.

En comparant ces différents modèles, nous notons que certains modèles supportent l'implantation de types d'interfaces et d'interactions multiples, alors que d'autres n'offrent que des interfaces de type invocation de méthode. C'est aussi vrai pour la composition hiérarchique qui est un concept puissant pour la mise en place de systèmes complexes. Nous remarquons que peu de modèles supportent intrinsèquement des modèles de communications flexibles, la plupart se basent sur la sémantique de communication fourni par le langage d'implantation sous-jacent. L'utilisation de connecteurs pour assurer les communications entre composants est également plus ou moins répandue, et permet d'adopter plusieurs sémantiques de communication. La séparation des aspects fonctionnels et de contrôle est un aspect émergeant dans les modèles EJB, CCM, COM+ et DCOM, mais ils ne sont pas supportés au niveau composant sauf dans les modèles OpenCOM et Fractal.

Cette variété dans les caractéristiques de modèles de composant nécessite de les considérer afin de proposer une méthode générique d'analyse d'un système basé composant.

Dans le prochain chapitre, nous présentons d'abord un état de l'art sur les méthodes formelles d'analyse qualitative et quantitative de systèmes, ainsi que les formalismes utilisés. Puis, nous étudions les travaux proposés pour l'analyse des CBS.

CHAPITRE 3

Analyse des CBS

3.1 Introduction

Les systèmes informatiques modernes se caractérisent par un comportement complexe incluant notamment la communication via des réseaux locaux et larges (Intranet et Internet), des services offerts par des machines serveurs, des mécanismes de contrôle de flots, ...etc. De surcroît, la charge de ces systèmes est souvent importante, consistant en un mélange de données batch ou de données interactives temps réel, en particulier pour la voix et la vidéo. Certaines applications sont mêmes critiques lorsqu'il s'agit, par exemple, d'applications liées à la vie humaine, telles que la construction des avions, la construction automobile ou encore des applications médicales. D'autres ont des effets économiques énormes si elles ne sont pas conçues d'une manière précise et sûre, telles les applications E-commerce.

Ainsi, la complexité et criticité de ces systèmes poussent les concepteurs à valider un système avant de le construire, afin de s'assurer de sa correction et de l'atteinte des objectifs tracés. On parle alors de validation ou analyse *à priori*. L'analyse des systèmes se fait en considérant deux aspects : un aspect qualitatif consistant à rechercher des propriétés qualitatives et un aspect quantitatif s'intéressant au calcul d'indices de performances.

Nous nous intéressons dans ce chapitre à présenter les méthodes d'analyse qualitative et quantitative des systèmes, tout en décrivant les formalismes de description utilisés dans la littérature. Nous nous focalisons ensuite sur l'analyse des systèmes basés composants et dressons un état de l'art des travaux proposés à cet effet.

3.2 Objectifs

L'analyse d'un système vise à fournir de l'information qualitative ou quantitative sur le comportement du système :

- L'objectif de l'analyse qualitative est de s'assurer que les propriétés qu'il possède correspondent bien aux propriétés attendues de lui, comme par exemple l'absence de blocage du système, la terminaison, atteindre un état spécifique, etc.
- L'analyse quantitative ou *évaluation des performances* permet de quantifier la performabilité du système étudié vis-à-vis des objectifs pour lesquels le système est construit. Les indices recherchés peuvent être le temps de réponse à une requête d'un utilisateur dans un système multiutilisateurs, la durée moyenne de transmission d'un message dans un réseau de communication, le débit des transactions dans les systèmes de bases de données, etc.

De nos jours, les techniques de vérification et d'évaluation de performance sont de plus en plus utilisées dans le milieu industriel pour l'analyse d'une grande variété de systèmes (systèmes matériels, logiciels, systèmes réactifs, systèmes temps réel, systèmes embarqués). Il est maintenant prouvé que ces

techniques sont efficaces et sont fréquemment utilisées pour détecter des bogues dans des cas industriels. De nombreuses études sont en cours pour élargir leur champs d'application et améliorer leur efficacité.

3.3 Vue sur les démarches d'analyse

Afin de vérifier les propriétés et rendre compte des performances d'un système, deux approches globales ont été suivies :

- La première approche, dite *à posteriori*, consiste à procéder à des bancs d'essai une fois le système réalisé, et mesurer les indices et propriétés souhaités sur le système en fonctionnement. Cette méthode est utile pour contrôler les systèmes et répondre à des objectifs précis tels que des réglages à faire, un équilibrage de charge, etc. Toutefois, la correction des erreurs risque d'être extrêmement coûteuse.
- Pour remédier aux inconvénients de la première approche, il est préférable de faire une analyse durant la phase de conception d'un système, c.-à-d. avant sa réalisation. Cette analyse est dite *à priori* : elle permet de révéler les fautes de conception qui ont pu être introduites au cours de n'importe quelle phase du cycle de développement. Dans ce cas, les éventuelles erreurs détectées sont corrigées durant l'étape d'analyse/validation du modèle, ce qui réduit le coût global de construction et maintenance d'un système. De cette façon, l'analyse à priori permet une meilleure réactivité face aux aléas pouvant survenir durant la vie d'un système, comme par exemple l'occurrence de pannes, les risques d'erreurs, les possibles modifications apportées à un système, etc.

L'analyse à priori peut être conduite en suivant deux catégories de méthodes:

1. Dans la première catégorie, les résultats d'analyse sont générés par un programme de simulation [156; 192; 187] qui imite le comportement du système. Par exemple, on peut simuler un système en générant aléatoirement des événements répartis dans le temps de manière discrète. Néanmoins, cette méthode ne peut englober tous les cas possibles pouvant apparaître pendant le fonctionnement réel d'un système. En général, les concepteurs ont recours à la simulation lorsque les autres méthodes (en l'occurrence les méthodes analytiques) ne peuvent être utilisées, souvent à cause de la taille du modèle à analyser. De ce fait, la simulation reste un outil important pour l'estimation des indices de performance.
2. La deuxième méthode consiste à utiliser des méthodes analytiques qui résolvent un modèle mathématique (formel), englobant les caractéristiques essentielles du système. Les propriétés et valeurs des indices de performance sont alors déduites à l'aide d'outils développés à cet effet.

Ces deux méthodes sont fondées sur des formalismes de modélisation qui sont utilisés afin de construire une représentation ou *modèle* du système à étudier. Ce modèle sert alors de base pour la recherche des propriétés souhaitées et d'indices de performance.

En particulier, les méthodes analytiques formelles permettent une analyse *à priori* automatique d'un système avant son implantation. Nous nous intéressons à cette catégorie de méthodes. Dans la suite, nous décrivons d'abord les deux types d'analyse qualitative et quantitative, puis, nous nous concentrons sur les formalismes et modèles de performance.

3.4 Le test

Le test est une technique dynamique qui consiste à exécuter un programme en lui fournissant des entrées valuées, dites *entrées de test*, et à vérifier la conformité des sorties par rapport au comportement

attendu pour déterminer leur correction s'il y a lieu [123], et mesurer des indices de performance souhaitées. L'environnement de test reproduit le plus fidèlement possible l'environnement opérationnel. En pratique, le programme est exécuté sur matériel cible en connexion avec un simulateur de l'environnement physique, offrant des possibilités d'injection de fautes, si la propriété ciblée doit être vérifiée en présence de telles fautes.

Comme le test exhaustif d'un programme sur toutes les entrées possibles est impraticable, le testeur est amené à choisir de manière pertinente un sous-ensemble du domaine d'entrée. Cette sélection s'effectue à l'aide de critères de test, qui peuvent être liés à un modèle de la structure du programme, ou à un modèle des fonctions que doit réaliser le programme.

Dans le contexte des systèmes basés composants, le test de ce type de système nécessite le test individuel de chacun des composants, ensuite le test du composant dans un environnement spécifique. On retrouve également deux types de tests utilisés pour valider un système : le test de *conformité* et le test d'*interopérabilité*. Le test de conformité s'intéresse à une implantation particulière et essaie de vérifier que son comportement correspond à ce qui est décrit dans la spécification d'origine. Le test d'interopérabilité essaie d'évaluer la capacité de deux ou plusieurs implantations à interagir correctement.

Comme exemple de test, on peut citer le test de *robustesse* envisagé dans [207]. Dans cette étude, le concepteur génère les séquences de test, puis y injecte des fautes dans le but de matérialiser l'hostilité de l'environnement. Ces séquences sont ensuite exécutées sur l'implantation du système. Les réponses reçues par les testeurs sont alors confrontées à la spécification dégradée du système pour décider de sa robustesse.

3.5 Analyse qualitative

L'analyse qualitative, ou vérification, consiste à rechercher les propriétés qualitatives du système à analyser.

On distingue trois grandes classes de techniques de vérification : la vérification de modèle (ou model-checking), la preuve et le test. Ces techniques sont en général utilisées d'une manière indépendante : une technique est choisie selon l'étape de développement pour un problème de vérification donné.

3.5.1 Propriétés à vérifier

Plusieurs types de propriétés peuvent être vérifiées. Nous en présentons ici les plus souvent rencontrées.

- *Propriété d'atteignabilité ou accessibilité (reachability)* : énonce que, sous certaines conditions, un état du système peut être atteint ou non, par exemple un état de défaillance peut survenir, l'état initial du système peut être repris, un état de disponibilité des ressources est atteint,... etc.
- *Propriété de vivacité (liveness)* : stipule que, sous certaines conditions, quelque chose finit par avoir lieu. Par exemple, toute requête finira par être satisfaite.
- *Propriété d'absence de blocage (no deadlock)* : traduit le fait que le système ne se trouve jamais dans un état de blocage, qu'il ne peut plus quitter. Par exemple, une panne peut toujours être réparée. Des définitions formelles de ces propriétés sont données dans [195].
- *Propriété de sûreté (safety)* : exprime que, sous certaines conditions, quelque chose n'arrive jamais. Par exemple, un état non souhaité ne se produit jamais.

3.5.2 Le model Checking

La vérification de modèle ou le model checking [195] consiste à construire un modèle fini d'un système et vérifier qu'une propriété cherchée est vraie dans ce modèle. Cette technique est complètement automatique et rapide.

En général, il existe deux façons de procéder à cette vérification :

- Vérifier qu'une propriété exprimée dans une logique temporelle est vraie dans le système, ou bien
- Comparer le système à une spécification (en utilisant une relation d'équivalence ou de préordre) pour vérifier si le système satisfait la spécification ou non.

Dans tous les cas et quel que soit le formalisme utilisé pour la modélisation du système, l'outil *model checker* procède par exploration des états. La plupart des model-checkers sont capables de générer une réponse positive si la propriété est garantie pour tous les comportements du modèle et une réponse négative complétée par un contre-exemple illustratif en cas de violation de la propriété.

Remarque 1. *Bien que cette approche ait l'avantage d'être exhaustive, elle est limitée par le problème classique de l'explosion combinatoire du nombre d'états.*

3.5.3 Le theorem proving

La preuve de théorèmes ou theorem proving est une technique consistant à exprimer le système et les propriétés recherchées sous-forme de formules dans une logique mathématique, puis procéder à une recherche de la preuve de ces propriétés. Le processus de recherche de la preuve d'une propriété est dit *theorem proving*. Il établit, par une suite finie d'étapes de raisonnement appelées *inférences*, qu'une certaine propriété est la conséquence logique d'axiomes, de règles de déduction et d'un ensemble d'hypothèses décrivant le système étudié [142]. Les inférences sont réalisées par mécanisme déductif ou en utilisant un système de réécriture.

La preuve peut être appliquée à toutes les phases de développement du système. Il est possible, par exemple, de chercher à prouver qu'une spécification satisfait certaines bonnes propriétés, qu'un programme est correct par rapport à une spécification détaillée (preuve de programme), ou qu'il termine toujours sous certaines conditions (preuve de terminaison).

Un cadre classique de preuve est le calcul des prédicats du premier ordre : il permet de raisonner sur des formules qui comportent des variables.

Remarque 2. *L'avantage du theorem proving par rapport au model checking est qu'il est possible de l'utiliser avec des espaces d'états infinis, à l'aide de techniques comme l'induction structurelle. Toutefois, au contraire du model-checking qui est complètement automatique et rapide, le processus de vérification est lent, difficile à automatiser, error-prone (sujet à l'erreur) et nécessite des utilisateurs spécialisés ayant beaucoup d'expertise.*

3.6 Analyse quantitative

L'analyse quantitative vise à calculer les paramètres quantitatifs de performance. Les méthodes d'évaluation des performances sont fondées sur des outils mathématiques, à savoir la théorie des probabilités et des processus stochastiques markoviens.

3.6.1 Principe

L'évaluation à priori des performances d'un système suit les étapes suivantes :

1. Décrire les aspects statiques et dynamiques du système étudié en utilisant un modèle formel.
2. Etudier le modèle en générant l'ensemble des états possibles. Cet ensemble d'états est calculé sous-forme d'un processus stochastique étant souvent du type *chaîne de Markov*.
3. Résoudre la chaîne obtenue afin de calculer les probabilités de se trouver dans chacun des états de la chaîne.
4. Calculer les indices de performance en utilisnt les probabilités obtenues.

On parle ici de méthodes *stochastiques markoviennes*.

3.6.2 Indices de performance

Les méthodes markoviennes ou apparentées [120; 54; 82] permettent d'étudier le système modélisé à un instant donné (*analyse transitoire*) ou à long terme (*résolution à l'équilibre* ou *régime stationnaire* ou *permanent*). Les indices de performance calculés informent sur l'efficacité et productivité du système étudié. L'analyse transitoire permet de calculer des indices du genre : combien de clients demandant un service X en moyenne ou en distribution vont être servis durant la prochaine heure. A l'encontre, l'analyse stationnaire répond aux questions telles que : durant une très longue période, quel est le pourcentage de temps moyen durant laquelle un serveur du système est occupé. Dans les deux types d'analyse, on retrouve un ensemble d'indices fréquemment calculés, notamment :

- *Le débit (throughput)* : c'est le nombre moyen de requêtes satisfaites par unité de temps.
- *Le temps de réponse moyen* : c'est le temps moyen pour exécuter une requête d'un client du système.
- *Le temps d'attente moyen* : c'est le temps moyen qu'attend un client avant le début de la prise en charge de sa requête.
- *L'utilisation moyenne* d'une machine/ressource ou serveur : c'est le pourcentage moyen de temps durant lequel la machine/ressource ou serveur est occupé(e) à servir les requêtes.
- La probabilité qu'une machine soit bloquée ou oisive.
- La fréquence moyenne de l'exécution d'une tâche.
- ...etc.

Ces paramètres sont des indicateurs sur la manière dont opère le système : le débit détermine si l'on peut répondre à la demande ou non, une faible utilisation moyenne d'une machine peut indiquer un problème au niveau de cette machine, une machine couramment bloquée ou oisive peut signifier que le taux de traitement d'une requête est plus rapide ou plus lent, ... En plus de ces valeurs moyennes, il peut être d'intérêt de connaître la variance de ces quantités ou plus généralement de leurs distributions. Par exemple, il peut être important de connaître la variance du temps d'attente d'un client, en plus de sa valeur moyenne.

Le calcul de la chaîne de Markov, ou plus généralement l'espace d'états, dépend du modèle ou formalisme utilisé pour la modélisation du système. Un panorama non exhaustif de ces modèles de performance est donné ci-après.

3.6.3 Modèles de performance

On distingue différents formalismes utilisés en vue de la modélisation et évaluation des performances d'un système. Comme les processus markoviens sont la base de l'évaluation des performances, nous en présentons avant tout une brève description. Puis, nous donnons les modèles de performance les plus utilisés.

3.6.3.1 Processus Markoviens

Les processus markoviens [120; 54] représentent une classe de processus stochastiques. Ceux-ci sont des modèles mathématiques utilisés pour la description de phénomènes de nature probabiliste en fonction d'un paramètre qui est généralement le temps.

Définition 3.1 (Processus stochastique)**.** Un processus stochastique $\{x(t), t \in T \subseteq |R\}$ est une famille de variables aléatoires définies sur le même espace de probabilité, à valeurs dans un même espace d'états E et indexées par le paramètre t, qui modélise habituellement le temps.

Un processus stochastique X peut être à temps *continu* ou *discret*. X est dit à temps *continu* ssi T est un intervalle de |R et à temps discret ssi $T \subseteq |N$. T est alors $|R^+$ ou l'ensemble inclus dans |N des instants où l'on observe le phénomène.

Le processus X peut être à espace d'états *discret* ssi E est dénombrable; dans ce cas, on l'identifie à un sous-ensemble de |N et on note i, j, k, ... ses éléments. Si E n'est pas dénombrable, il est considéré comme un sous-ensemble de |R.

La variable X_0 est appelée état initial du processus.

Définition 3.2 (Processus markovien)**.** Un processus stochastique à espace d'états discrets est dit *markovien* s'il satisfait la propriété *markovienne* suivante :

$$\forall n \in N, \forall j, i_1 < i_2 < ... < i_{n-1}, P\{x_n = j | x_1 = i_1, ..., x_{n-1} = i_{n-1}\} = P\{x_n = j | x_{n-1} = i_{n-1}\}.$$

La propriété markovienne traduit le fait que l'état futur d'un système (donnant son évolution) ne dépend que de son état présent : C'est la propriété de *perte de mémoire*. Les processus markoviens sont alors dits *sans mémoire*. Parmi les processus markoviens, on trouve les *chaînes de Markov*.

Définition 3.3 (Chaîne de Markov)**.** Une *chaîne de Markov* (CM) est un processus markovien dont l'espace des états est dénombrable. Elle est dite *homogène* si $P(X_t \leq x | X_{t_n} = x_n) = P(X_{t-t_n} \leq x | X_0 = x_n)$ (c.-à-d., cette probabilité ne dépend que de l'écart $t - t_n$).

En d'autres termes, un modèle de Markov consiste en un ensemble d'états discrets qui décrit tous les états possibles du système. Les transitions de l'état i à l'instant s vers l'état j à l'instant t se produit avec une probabilité (conditionnelle) $p_{ij}(s,t) = P\{X_t = j | X_s = i, i, j \in E\}$, ou bien $p_{ij}(t) = p_{ij}(s, s+t)$ lorsque la chaîne est homogène.

La chaîne est décrite par la *matrice de transition d'états* (dite *stochastique*) donnée par P(s,t) ou P(t), définie par:

$$\forall s,t \in T, \forall i, j \in E, p_{ij}(s,t) \geq 0.$$
$$\forall s,t \in T, \forall i \in E, \sum_{j \in E} p_{ij}(s,t) = 1.$$

La probabilité (inconditionnelle) que la chaîne soit dans l'état i à l'instant t est $\pi_i(t)$. On notera $\pi(t)$ le vecteur ligne correspondant ($\pi(0)$ est la distribution initiale de la chaîne). Le temps passé par une chaîne de Markov dans un état donné i, appelé *temps de séjour* ou de maintien dans l'état i, suit une loi exponentielle pour les chaînes de Markov à temps continu et géométrique pour les chaînes de Markov à temps discret. Ces lois sont les seules possédant la propriété d'être sans mémoire :

$$\forall x > 0, \forall y > 0, P(X \leq x+y | X \geq y) = P(X \leq x).$$

Les matrices P(s,t) et P(t) fournissent une description complète de la chaîne de Markov. Elles sont utilisées pour rendre compte des performances du système modélisé par la chaîne. On peut par exemple répondre à la question "quelle est la probabilité que le système revienne dans un état i après k transitions de l'état initial (t=0)" ?

D'une manière générale, l'étude du comportement (et donc des performances) d'une chaîne de Markov en fonction du temps peut être menée selon deux grandes directions :

1. L'étude en *régime transitoire*: consiste à déterminer $\pi(t)$ pour tout t>0. On utilise pour cela $\pi(0)$ et les matrices P(s,t) pour tout (s,t) et en particulier les P(0,t) et la relation $\pi_i(t) = \sum_{k \in E} \pi_k(0).p_{k,i}(0,t)$.
2. L'étude en *régime stationnaire ou permanent (à l'équilibre)* : consiste à vérifier si le système tend vers un équilibre (en termes probabilistes) lorsque le temps croît, c.-à-d. vérifier s'il existe une distribution de probabilités $\pi = (\pi_i)_{i \in E}$ telle que pour tout i : $lim_{t \to +\infty} \pi_i(t) = \pi_i$.

Nous nous intéressons dans cette thèse à l'étude des CBS à l'équilibre. Une chaîne de Markov qui possède une telle distribution limite, indépendamment de la distribution initiale, est dite *ergodique*. Sa résolution dépend du type de la chaîne :

- Une chaîne de Markov à temps discret (*Discrete Time Markov Chain, DTMC*) est entièrement déterminée par sa distribution initiale π_0 et la matrice stochastique P(1), notée simplement P, où $p_{ij} = P(X_{n+1} = j | X_n = i)$. Nous avons alors pour n>0 : $P(n) = P^n$. Pour toute DTMC ergodique, la distribution limite π est telle que $\pi = \pi P$. Dans le cas du régime transitoire, on utilise la relation $\pi(n) = \pi(0)P^n$.
- Quant aux chaînes de Markov à temps continu (Continuous Time Markov Chain, CTMC), l'équivalent de la matrice P des DTMCs est le *générateur* Q, appelé aussi *matrice des générateurs infinitésimaux*, qui vérifie les relations :
$$\forall i, j \in E, j \neq i, q_{ij} \geq 0.$$
$$\forall i \in E, q_{ii} = -\sum_{j \in E, j \neq i} q_{ij}$$
où q_{ij} représente la vitesse de transition de l'état i vers l'état j et le temps de séjour en i est une variable aléatoire de loi exponentielle de paramètre λ_i (notée $EXPO(\lambda_i)$) avec $\lambda_i = -\frac{1}{q_{ii}}$. Pour toute CTMC ergodique, la distribution limite π est telle que $\pi Q = 0$. Pour étudier le régime transitoire d'une CTMC, on utilise les relations $\frac{d\pi}{dt}(t) = \pi(t)Q$, donc : $\pi(t) = \pi(0)e^{Qt}$.

La résolution d'une chaîne de Markov munie d'un générateur $Q(n \times n)$ consiste alors à calculer le vecteur de probabilités π de $|R^n$ tel que :

$$\pi Q = 0.$$
$$\pi \geq 0, \sum_{i=1}^{n} \pi_i = 1.$$

Cette résolution peut être exacte, approchée ou limitée à la recherche de bornes. Elle utilise les techniques de l'analyse numérique linéaire lorsque l'ensemble des états est fini, à savoir les méthodes directes (telles que la méthode de Gauss ou de factorisation LU) ou itératives [178; 202] (méthodes de la puissance itérée, de Gauss-Seidel, de Jacobi, ... etc). Dans le cas infini, les formules analytiques ou leurs approximations peuvent être employées.

3.6.3.2 Réseaux de files d'attente

Les files ou systèmes d'attente (Queueing Networks, QN) [120; 121; 79] ont été largement utilisé(e)s pour l'évaluation des performances des systèmes à événements discrets tels que les systèmes informatiques, les réseaux de communication et les systèmes de production.

Un système d'attente simple consiste en une file d'attente, appelée aussi *buffer* ou *tampon*, et d'une station de service constituée d'une ou plusieurs ressources appelées *serveurs*. Des entités dites *clients* ou *travaux (jobs)*, générées par un *processus d'arrivée externe*, rejoignent la file pour recevoir un service offert par l'un des serveurs. Une politique de service (généralement premier servi de la file, FIFO) est adoptée pour servir les clients. A la fin du service, le client quitte le système. Un tel système est caractérisé par:

- Le nombre de serveurs,
- la discipline de service,
- la capacité du buffer,
- le processus d'arrivée, et
- le processus de service.

Un exemple d'un système d'attente est un guichet de dépôt et retrait dans une banque. Un nombre fixe d'employés répondent aux requêtes des clients arrivant dans une file d'attente.

D'une manière informelle, une file d'attente est définie en trois étapes : une étape de *définition* qui englobe la définition des serveurs, leur nombre, les clients et la topologie utilisée; la *paramétrisation* qui consiste à définir les paramètres du modèle, tels que les processus d'arrivée, le taux de service et le nombre de clients; et l'étape d'*évaluation* qui permet d'obtenir une étude quantitative du système modélisé en calculant des indices de performance. Cette dernière étape est menée, de manière analogue aux chaînes de Markov, en étudiant le comportement du système d'attente à court ou à long terme (*analyse transitoire* et *stationnaire*).

Une généralisation naturelle des files d'attente est de considérer des *réseaux de files d'attente*. Un réseau de file d'attente consiste en un ensemble de M files, dites également *stations*. Les réseaux de files d'attente sont souvent utilisés comme des outils de modélisation et d'analyse, du fait de leur efficacité d'analyse et de la haute précision des résultats de performance. Ils ont été utilisés pour l'étude des systèmes informatiques.

Inconvénient La puissance de modélisation de ce modèle pour la représentation des systèmes concurrents avec synchronisation est très limitée.

Pour pallier aux limites des QN, la communauté scientifique a développé de nouveaux modèles plus ou moins spécialisés qui répondent à la complexité de ces systèmes. On trouve les algèbres de processus stochastiques, les réseaux de Petri stochastiques, les Réseaux d'automates Stochastiques,... etc. Récemment, des extensions des modèles classiques de files d'attente ont été proposées afin de tenter de remédier à leurs inconvénients. On trouve :

- Les *files d'attentes étendues ou Extended Queuing Networks (EQN)* [125] qui permettent de représenter des caractéristiques intéressantes des systèmes réels, tels que les contraintes de synchronisation, de concurrence et la capacité finie des tampons, et
- les *files d'attentes en couches ou Layered Queuing Networks (LQN)* [220] qui modélisent des patterns de communication client/serveur dans les systèmes distribués. Elles permettent de capturer l'impact de plusieurs couches de serveurs logiciels. Un modèle LQN est exprimé sous forme d'un ensemble d'objets dits *tâches*. Une tâche fournit des services représentés par des entrées. Les

entrées d'une tâche invoque les entrées d'autres à des niveaux plus bas. Les tâches sont exécutées sur des processeurs. La modélisation d'exécutions séquentielles ou parallèles est donnée par l'*activité*. Les activités sont les plus petites unités de calcul. Elles peuvent envoyer des requêtes vers les entrées ou accepter des requêtes issues d'autres entrées.

On peut citer des exemples d'outils d'évaluation de performance pour les QN, EQN et LQN : *RESQ/IBM* [193], *QNAP2* [212], *LQN* et *LQNS tool* [220].

Dans ce qui suit, nous présentons les modèles algèbres de processus stochastiques et réseaux de Petri stochastiques.

3.6.3.3 Algèbres de processus stochastiques

Les *algèbres de processus* (*Process Algebra, PA*) sont des langages abstraits ayant été introduits comme cadre formel pour la spécification, la compréhension et l'analyse des systèmes complexes caractérisés par la communication et la concurrence. Ces systèmes sont décrits sous-forme de collections d'entités appelées *agents* ou *processus*, qui évoluent, dans la plupart des cas, indépendamment les unes des autres, et communiquent quelquefois pour atteindre un but commun. Les agents exécutent des *actions atomiques* constituant les blocs de construction du langage. Les systèmes complexes sont construits à partir de ces blocs de construction et en appliquant les constructeurs de l'algèbre.

Plusieurs opérateurs sont définis, à savoir le préfixe, le choix, la composition parallèle, la restriction, ... etc. En plus de ces opérateurs, des mécanismes d'abstraction, permettant de cacher les détails internes, sont fournis. Les termes de l'algèbre sont définis par une grammaire. Les actions peuvent être visibles ou non.

Une famille d'algèbres de processus a été développée, caractérisée d'une part par l'utilisation d'équations et d'inéquations entre les expressions de processus, et d'autre part par l'adoption de la communication synchronisée comme moyen primitif d'interaction entre les composants du système. Parmi les algèbres les plus connues, on trouve l'algèbre CCS (*Calculus of Communicating Systems*) de Milner [150; 151], l'algèbre CSP de Hoare [106; 37] et le π-calculus de Milner [152; 153], basé sur CCS mais offrant en plus le support de processus mobiles. Ces algèbres se distinguent principalement par les constructions par lesquelles les processus sont assemblés et la notion d'équivalence entre les expressions de processus. Toutefois, elles se partagent les points suivants :

- La modélisation compositionnelle : Les algèbres de processus fournissent un nombre de constructeurs pour la construction de systèmes larges à partir de petits systèmes.
- La sémantique opérationnelle : Les algèbres de processus sont dotées d'une sémantique opérationnelle structurelle (SOS) qui décrit les capacités des systèmes à l'exécution. En utilisant cette sémantique, les systèmes représentés par des termes dans l'algèbre peuvent être "compilés" en *systèmes de transitions étiquetées (Labelled Transition Systems, LTS)*.
- Le raisonnement comportemental à travers les équivalences et préordres : Les algèbres de processus utilisent des relations comportementales comme moyen pour relier différents systèmes décrits dans l'algèbre. Avec ce raisonnement, on peut vérifier si un système satisfait une certaine propriété.

Plusieurs extensions ont été proposées pour les algèbres de processus, entre autres, les *algèbres de processus stochastiques* ou *Stochastic Process Algebra, SPA*. Une SPA est une algèbre de processus associant des durées distribuées stochastiquement aux actions de l'algèbre. Les actions sont dites *activités*. Lorsqu'un composant peut faire une activité, une instance de la distribution de probabilité associée est échantillonnée. Le nombre résultant représente le temps que le composant prend pour faire l'action.

Aux algèbres stochastiques de processus correspondent des *diagrammes de transition* dont les arcs sont étiquetés par des paires (type action, taux). On obtient ainsi un système étiqueté (LTS). Le LTS

est associé à une chaîne de Markov à temps continu tel que chaque état du diagramme de transition correspond à un état de la chaîne et les transitions entre états de la chaîne sont définies par les arcs du diagramme. Les performances d'un système sont alors étudiées en résolvant la chaîne correspondante.

3.6.3.4 Réseaux de Petri stochastiques

Les réseaux de Petri stochastiques (*Stochastic Petri Nets, SPN*) [161; 157; 81] sont une extension des réseaux de Petri (PN) [174; 160]. Ceux-ci ont été largement utilisés pour la modélisation et l'analyse des systèmes informatiques. En effet, ils permettent la description des phénomènes de parallélisme, partage de ressources, synchronisation et communication, caractéristiques prédominantes des systèmes modernes. Nous commençons donc par donner un bref aperçu des PN, puis nous présentons les SPN.

Réseaux de Petri Un réseau de Petri est un graphe orienté biparti dont les noeuds sont de deux types : les places modélisant des ressources et les transitions modélisant des activités. L'état du système, dit *marquage*, est donné par une répartition de *jetons* dans les places.

Définition 3.4 (Réseau de Petri)**.** Un réseau de Petri est un tuple $\mathscr{N}$ = (P, T, Pre, Post, Inh, Pri) tel que :

- P est l'ensemble des places;
- T est l'ensemble des transitions
- Pre, Post et Inh : $P \times T \rightarrow Bag(P)$ sont les fonctions d'arcs[1] : Pre est la fonction d'incidence avant, Post la fonction d'incidence arrière et Inh la fonction d'inhibition.
- Pri : $T \rightarrow \mathbb{N}$ est la fonction de priorité des transitions.

C=Post-Pre est dite matrice d'incidence de $\mathscr{N}$.

Un marquage $M : P \rightarrow \mathbb{N}$ est une fonction associant à toute place p le nombre de jetons ou marques qu'elle contient, noté M(p). Le marquage initial, noté M_0, donne le nombre initial de jetons dans les places de P. Le couple $\mathscr{S}$ = ($\mathscr{N}$, M_0) est dit réseau de Petri *marqué*.

Le système modélisé évolue par le franchissement (tir) des transitions modélisant l'exécution des activités. Le franchissement modifie l'état du système comme suit :

Règle de franchissement : Une transition t est franchissable en M, noté $M[>t$, ssi :

- Les préconditions sont vérifiées indépendamment de la priorité de t :
  $\forall p \in P, M(p) > Pre(pt) \wedge M(p) < Inh(p,t)$, et
- Aucune autre transition de priorité supérieure n'a ses préconditions vérifiées.

Le franchissement de t en M conduit au marquage M' (noté M[t> M') tel que :
M'=M-Pre(.,t)+Post(.,t) [2].

Un marquage M est dit *accessible* depuis le marquage initial M_0 ssi il existe une séquence de franchissements $S = t_1 t_2 ... t_n$ après laquelle M est obtenu. On note $M_0[S > M$.
L'ensemble des marquages accessibles depuis M_0 ou *ensemble d'accessibilité* (*Reachability Set*) est noté RS(S) ou RS. Le *graphe d'accessibilité* d'un réseau marqué $\mathscr{S}$, noté RG(S) ou RG (*Reachability Graph*), est un graphe dont les noeuds sont les marquages et chaque arc reliant deux noeuds M et M' est étiqueté par un nom de transition t représentant ainsi le franchissement M[t>M'.

Le graphe d'accessibilité est utilisé pour mener des analyses qualitatives et quantitatives du système modélisé.

[1] Bag(P) est l'ensemble des multi-ensembles sur P : $Bag(P) = \{x = (x(p))_{p \in P} | \forall p \in P, x(p) \in \mathbb{N}\}$

[2] Pre(.,t), Post(.,t) et Inh(.,t) sont les fonctions induites par Pre, Post et Inh sur P

Propriétés
L'étude d'un système s'appuie sur un ensemble de propriétés dont les principales sont :

Définition 3.5 (Bornitude)**.** Une place p du réseau marqué $\mathscr{S}$ est k-bornée ($k \in \mathbb{N}$) ssi pour tout marquage accessible M de S, $M(p) \leq k$.
Un réseau marqué est borné ssi pour toute place p de ce réseau, il existe un entier k tel que p soit k-bornée.
Un réseau $\mathscr{N}$ est borné ssi pour tout marquage initial M_0, le réseau marqué $(\mathscr{N}, M_0)$ est borné.

L'ensemble d'accessibilité d'un réseau borné est fini.

Définition 3.6 (Vivacité)**.** Une transition t du réseau marqué $\in \triangle \triangledown \mathscr{S}$ est vivante ssi pour tout marquage accessible M de $\mathscr{S}$, il existe un marquage M' tel que $M' \in RS(\mathscr{N}, M)$ et M'[t>.
Un réseau marqué $\mathscr{S}$ est vivant ssi toutes ses transitions sont vivantes.
Un réseau $\mathscr{N}$ est vivant ssi il existe un marquage initial M_0 tel que le réseau marqué $(\mathscr{N}, M_0)$ soit vivant.

Définition 3.7 (État d'accueil)**.** Un réseau marqué S admet un état d'accueil M_a ssi pour tout marquage accessible M, $M_a \in RS(\mathscr{N}, M)$.

Flots et semi-flots [100]
Les flots sont des invariants structurels des réseaux de Petri. Il existe des invariants de places, dits *P-flots* et *P-semiflots*, et des invariants de transition, dits *T-flots* et *T-semiflots*. Un P-flot (resp.P-semi-flot) est une somme pondérée de places à coefficients entiers (resp. naturels). Un T-semiflot représente le vecteur d'occurrences d'une séquence de transitions, alors qu'un T-flot s'interprète comme la différence de deux vecteurs d'occurrences. Dans le cadre de cette thèse, nous nous intéressons uniquement aux P-flots.

A chaque P-flot correspond une somme constante des marquages pondérés des places (appelée *P-invariant*). Dans le cas des P-semiflots où les poids sont positifs, il y a conservation des jetons sur l'ensemble des places à poids non nul. Cette conservation est souvent interprétée lors de l'étude d'un système, comme étant par exemple "le nombre de tâches en cours est toujours n".

De nombreuses méthodes de calcul des flots et semi-flots existent. Elles sont fondées sur l'algèbre linéaire appliquée aux éléments de définition du réseau (fonctions d'incidence).

Définition 3.8 (Flot et semiflot)**.** Soit $\mathscr{N}$ un réseau de Petri et C=Post-Pre sa matrice d'incidence. Un vecteur $f \neq 0$ de $\mathbb{Q}^{|P|}$ (resp. $(\mathbb{Q}^+)^{|P|}$) est un P-flot (resp. un P-semiflot) ssi $f^T.C = 0$.
Pour tout marquage accessible M d'un réseau marqué $(\mathscr{N}, M_0)$, on a alors la relation suivante, appelée *P-invariant* :

$$\sum_{p \in P} f_p.M(p) = \sum_{p \in P} f_p.M_0(p) = \alpha \text{ (une constante)}$$

Pour étudier les semiflots ou déterminer les conditions "minimales" de conservation, on calcule une famille appelée *famille génératrice minimale de semiflots* telle que tout semiflot est une combinaison linéaire à coefficients positifs de semiflots de cette famille. Un semiflot *minimal* est un élément d'une famille minimale.
Lorsque l'on veut modéliser des propriétés de conservation sur une partie seulement d'un système, on utilise la notion de semiflot *partiel*.

Définition 3.9 (Semiflot partiel)**.** f est un semi-flot partiel sur $\mathscr{N} = (P, T, ...)$ relativement à un ensemble $T' \subseteq T$ de transitions ssi f est un semiflot du réseau $\mathscr{N}' = (P, T', ...)$. L'ensemble T' permet d'isoler la partie du réseau concernée.

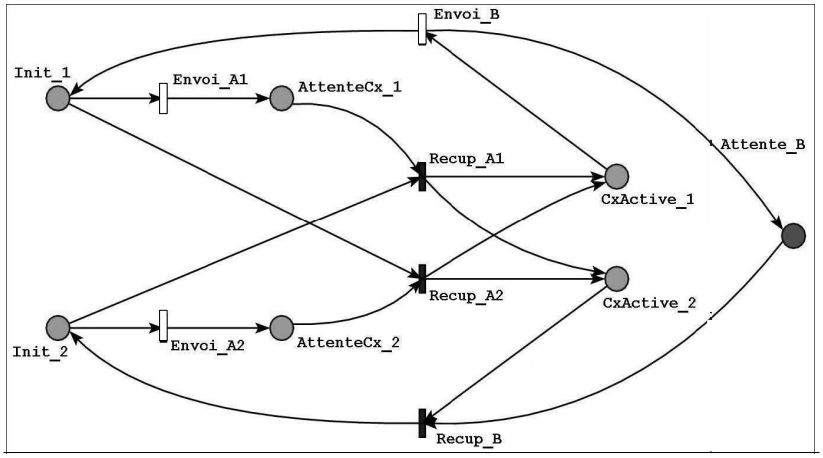


Figure 3.1: Modèle GSPN d'un protocole élémentaire

Réseaux de Petri Stochastiques Le modèle Réseau de Petri ordinaire permettait d'effectuer uniquement une analyse qualitative. Le souhait d'ajouter une dimension quantitative à ce modèle a été alors pressenti. De plus, dans les systèmes actuels, l'analyse de plusieurs phénomènes est fortement liée à la notion de temps, alors qu'il n'est pas modélisé dans ce modèle.

Suite à cela, plusieurs extensions aux réseaux de Petri ont été proposées, à savoir les Réseaux de Petri Stochastiques (SPN), les réseaux de Petri stochastiques généralisés (GSPN), les réseaux stochastiques déterministes et bien d'autres. Nous nous intéressons dans nos travaux aux réseaux de Petri stochastiques et plus généralement aux SPN généralisés.

Un réseau de Petri stochastique est un réseau de Petri qui associe à chaque transition une durée de tir aléatoire, donnée par une variable aléatoire qui suit une loi probabiliste. Pour franchir une transition à durée de tir non nulle, on échantillonne une durée, valeur initiale d'un compteur de temps. Lorsque ce compteur est à 0, la transition est franchie. Dans le cas où plusieurs transitions sont franchissables en même temps, on applique une politique de *course* où la transition à durée minimale obtenue à l'échantillonnage est choisie pour être franchie.

Lorsque certaines transitions peuvent avoir une durée de tir nulle, on parle de modèle SPN généralisé (Generalized SPN, GSPN). La durée de tir nulle modélise des cas où certaines activités sont urgentes et s'exécutent dès leur occurrence.

Enfin, pour obtenir un modèle stochastique analysable réductible à un modèle markovien, on se restreint aux lois exponentielles pour les durées non nulles.

Définition 3.10 (Réseau de Petri stochastique généralisé)**.** Un réseau de Petri stochastique généralisé est un couple $\mathscr{N} = (\mathscr{N}'; \theta)$ où :

- $\mathscr{N}' = (P, T, Pre, Post, Inh, Pri)$ est un réseau de Petri avec[3]: $T = T_E \uplus T_I$ où :
  - T_E est l'ensemble des transitions temporisées, à durée de tir de loi exponentielle et priorité nulle, et

[3] $A \uplus B$ désigne $A \bigcup B$ avec $A \bigcap B = \phi$

  - T_I est l'ensemble des transitions immédiates, à durée de tir nulle et priorité non nulle.
- $\theta : T \times Bag(P) \rightarrow \mathbb{R}^+$:
  - Si t est temporisée, $\theta(t,M)$ est la vitesse de tir de t en M, c'est à dire le paramètre de sa loi exponentielle;
  - Si t est immédiate, $\theta(t,M)$ est le poids de t en M.

Exemple 3.20. *Nous présentons sur la figure 3.1 le modèle GSPN d'un protocole élémentaire d'ouverture d'une connexion réseau entre deux processus P_1, P_2. Chaque processus peut ouvrir une connexion en envoyant un message de type A à son semblable. Néanmoins, seul le processus P_1 peut fermer la connexion en envoyant le message de type B. Les deux processus sont alors modélisés par un jeton dans chacune des places Init_1 et Init_2. Les transitions temporisées Envoi_A1 et Envoi_A2 modélisent l'envoi d'un message de type A respectivement par P_1, P_2. La récupération des messages envoyés se fait par les transitions Recup_A1 et Recup_A2 qui sont immédiates. A ce moment, il est plus cohérent d'associer la même priorité à ces deux transitions puisque les deux processus ont la même chance de récupérer leurs messages dans le système global. La fermeture de connexion est représentée par la transition temporisée Envoi_B. Lorsque le processus P_2 reçoit le message B, il doit immédiatement fermer sa connexion, ce qui est modélisé par la transition immédiate Recup_B.*

L'analyse des réseaux de Petri stochastiques généralisés permet de générer un graphe d'accessibilité constitué de deux types de marquages :

- Les marquages *tangibles* générés suite au franchissement des transitions temporisées.
- Les marquages *évanescents* générés lors du franchissement des transitions immédiates.

Définition 3.11 (Marquage tangible, évanescent)**.** Un marquage M est dit évanescent (vanishing) ssi il existe une transition immédiate franchissable à partir de M. Un marquage est dit tangible ssi il n'est pas évanescent.

L'ensemble d'accessibilité *tangible*, noté TRS, est la restriction de l'ensemble d'accessibilité aux marquages tangibles.

3.6.4 Évaluation de performances et réseaux de Petri Stochastiques

Les modèles SPN/GSPN sont largement utilisés dans divers domaines de l'évaluation de performances des systèmes informatiques Les modèles exploitables pour un calcul numérique sont ceux associant des lois exponentielles aux durées de franchissement des transitions temporisées. Ce type de loi est dit "sans mémoire" dans le sens où le temps de franchissement de la transition associée ne dépend que du franchissement lui-même et non des précédents franchissements.

Ce fait a été exploité par Molloy [157] pour démontrer que le graphe d'accessibilité engendré d'un SPN est isomorphe à celui d'une chaîne de Markov finie à temps continu (CTMC). Par conséquent, l'analyse d'un SPN s'inspire de la technique d'analyse des CTMC. Le générateur infinitésimal Q de la chaîne sous-jacente au graphe d'accessibilité est obtenu, à partir de ce graphe, comme suit :

$$\text{Pour } i \neq j,\ Q_{i,j} = \frac{\sum_{t,M_i[t>M_j} \theta(t)(M_i)}{\sum_{t,M_i[t>} \theta(t)(M_i)}$$
$$Q_{i,i} = -\sum_{j \neq i} Q_{i,j}$$

La chaîne admet un vecteur de probabilités à l'équilibre π si elle est ergodique, ce qui revient à vérifier si le graphe est fortement connexe. Cette condition est également satisfaite lorsque le SPN est vivant, borné et admet un état d'accueil [224].

Théorème 3.1. *Un SPN à temps continu, borné et vivant, tel que son marquage initial M_0 est un état d'accueil, est soit ergodique, soit réductible à une sous-chaîne ergodique unique.*

Le vecteur π est solution du système d'équations suivant:

$$\pi.Q = 0$$
$$||\pi|| = \sum_i \pi_i = 1$$

De nombreuses méthodes de résolution existent. Le livre de W. J. Stewart [202] constitue un des ouvrages de référence décrivant un panel de ces méthodes. Trois grandes classes de méthodes sont distinguées :

- Les méthodes directes comme la méthode de Gauss, de factorisation LU, ...
- Les méthodes itératives telles que les méthodes de la puissance itérée, Gauss Seidel, Jacobi, etc.
- Les méthodes de projection comme la méthode d'Arnoldi, la méthode GMRES (Generalized Minimal Residual method), etc.

Généralement, les méthodes itératives ou de projection sont souvent utilisées.

En utilisant le vecteur des probabilités stationnaires π, il est possible de calculer plusieurs indices de performances. La méthode générale est basée sur le calcul d'une fonction de récompense associant à chaque état une valeur numérique jugeant de l'état en adéquation avec la condition recherchée. Le vecteur de récompense r obtenu est alors multiplié avec le vecteur π pour obtenir l'indice R souhaité.

$$R = \sum_{M_i \in RS} r_i.\pi_i$$

Les principaux paramètres fréquemment recherchés sont :

1. La fréquence moyenne de franchissement d'une transition : $\theta^*(t_i) = \sum_{M_j \in RS, M_j[t_i>} \pi_j.\theta(t_i)(M_j)$
2. Le nombre moyen de marques dans une place : $M^*(p) = \sum_{M_i \in RS} \pi_i.M_i(p)$
3. Le temps moyen de séjour d'une marque dans une place : $A^*(p) = \dfrac{M^*(p)}{Post(p,.).\Theta^*}$
   où Θ^* est le vecteur des fréquences moyennes.

Réseaux stochastiques généralisés Les marquages évanescents dans un réseau stochastique généralisé correspondent à des états à temps de séjour nul. Ils ne peuvent donc être pris en compte lors du calcul du vecteur de probabilités stationnaires. Seuls les états tangibles intéressent l'analyse.

Les états évanescents sont alors éliminés. Plusieurs méthodes d'élimination existent :

1. Préservation des états évanescents : résolution de la chaîne de Markov incluse (déterministe), puis retour aux états tangibles.
2. Élimination des états évanescents directement à partir du graphe d'accessibilité.
3. Élimination des états évanescents par transformation du réseau en un réseau stochastique, sans transition immédiate.

L'élimination des marquages évanescents est motivé globalement par deux raisons :

- Les indices de performance recherchés sont ceux qui concernent les marquages tangibles.
- Souvent, le réseau génère un nombre d'états évanescents plus grand que celui des tangibles, ce qui pertube les méthodes de résolution.

Le modèle SPN a été lui-même étendu pour définir une sémantique temporelle dans le cadre d'un modèle de haut niveau. Ceci a donné naissance aux *Réseaux Stochastiques bien formés* ou SWNs qui seront détaillés plus tard, étant donné que nos travaux sont basés sur l'utilisation sur ce modèle.

Comme notre objectif est de définir une approche d'analyse des systèmes basés composant, il est important de choisir un modèle de performance le plus expressif possible, permettant d'éviter le problème d'explosion d'états et de mener une évaluation de performances la plus efficace possible. A cet effet, nous examinons les différents travaux proposés dans la littérature ayant trait à l'analyse des CBS, puis nous mettons en relief le modèle de performance sur lequel se base notre travail.

3.7 État de l'art : Tour d'horizon sur l'analyse des systèmes à composants

L'analyse des systèmes basés composant a fait l'objet de plusieurs travaux. Le souci initial des concepteurs était de vérifier des propriétés de compatibilité des composants à interconnecter, pouvant être de différents concepteurs. Cette vérification de compatibilité est principalement basée sur la comparaison de types de composants (interfaces/signatures de méthodes). Néanmoins, le besoin de pousser la vérification à des propriétés plus critiques a été ressenti. On en cite la propriété d'absence de blocage, la terminaison d'un service ou d'une application, le temps de réponse à une requête, etc. Des travaux ont alors été proposés pour permettre de mener une analyse qualitative (vérification de propriétés comportementales) et quantitative (calcul de performances). La plupart de ces travaux ont essentiellement adressé l'aspect qualitatif, alors que l'aspect quantitatif a été peu abordé.

Dans la suite, nous dressons l'essentiel des approches proposées pour chacun de ces deux aspects. Toutefois, nous commençons d'abord par présenter les travaux précurseurs inhérents à la modélisation et analyse de systèmes logiciels, vus comme une ensemble de processus ou composants interagissant entre eux.

3.7.1 Travaux précurseurs : Méthodologie PARSE

Les premiers travaux de modélisation d'un ensemble de modules processus communicants étaient ceux de [92]. L'objectif était de développer une méthodologie d'ingénierie du logiciel cohérente, permettant la conception de systèmes parallèles fiables et réutilisables en passant de la spécification au code. Cette méthodologie est appelée *PARSE* ou Parallel Software Engineering [91]. Elle se base sur une conception hiérarchique orientée objet, allant jusqu'à l'implantation. Pour spécifier le comportement dynamique du système étudié, une notation graphique appelée *graphe de processus* est employée. La méthodologie PARSE se déroule alors en plusieurs étapes (figure 3.2) :

1. *Conception structurelle* : Elle consiste à décomposer le système en un ensemble d'*objets processus*. Un objet processus est catégorisé par le concepteur comme un serveur de fonctions, un serveur de données ou un processus de contrôle. Il définit une interface de communication à laquelle d'autres objets processus peuvent être directement connectés. Les objets processus peuvent communiquer entre eux par échange de messages typés, à travers des chemins de communication pouvant être synchrones, asynchrones ou de diffusion, etc. Un objet processus peut également être constitué d'un ensemble d'objets processus de plus bas niveau auxquels il délègue la responsabilité de traiter les messages reçus. Les objets processus de plus bas niveau sont vus comme des unités séquentielles de calcul.

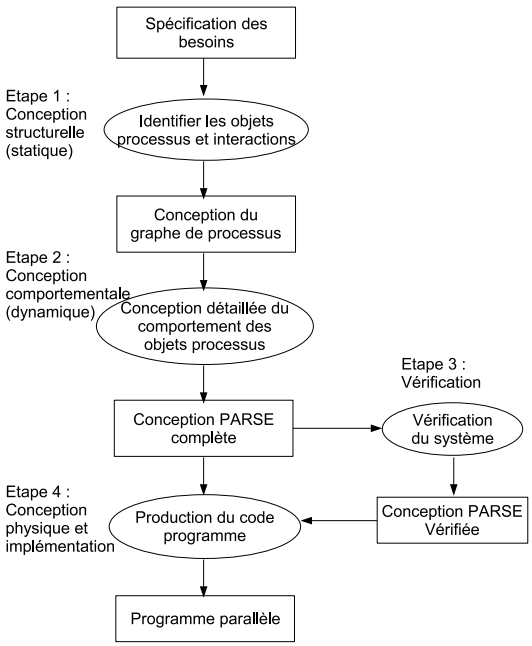


Figure 3.2: La méthodologie PARSE

2. *Conception comportementale* : Elle construit un graphe de processus représentant la structure logique du système conçu, une fois les objets processus et leurs interactions identifiés. Cette phase est dite *conception comportementale*. Un ensemble d'outils logiciels [95] a été développé pour faciliter la construction de graphes de processus et aider ainsi le concepteur à mener à bien cette phase. A chaque objet processus sont définis des chemins de communication en entrée et en sortie. Dans la plupart des cas, les interconnexions entre objets de bas niveau peuvent être transformées machinalement en des squelettes de réseaux de Petri par l'application de règles simples. Toutefois, il est nécessaire de compléter les réseaux par l'ajout de places et transitions internes et la définition du marquage initial.

3. *Vérification* : Elle consiste à analyser formellement le système global. A cet effet, les réseaux de Petri obtenus dans l'étape2 et correspondants aux objets processus sont combinés en un seul réseau. La combinaison des objets processus donne lieu au modèle comportemental du système complet, désigné par le processus objet de plus haut niveau. Ce réseau global peut être exécuté ou simulé à l'aide d'un outil d'exécution de réseaux de Petri. Il peut être également vérifié par des outils similaires, implantant les techniques d'analyse des réseaux de Petri, tel que DesignCPN [59; 52]. Ainsi, il est possible, grâce à cette vérification, de détecter les éventuelles erreurs de conception (analyse d'accessibilité, recherche de blocage, ...).

4. *Implantation* : Durant cette étape, une stratégie d'implantation est choisie, puis on procède à l'implantation. Le concepteur transforme la conception logique en une conception physique pouvant être implantée dans un langage choisi sur une machine parallèle donnée. Des règles de transformation automatique du graphe de processus vers le code peuvent être dérivées pour les langages parallèles tels que Parallel C, ADA, C/PVM, ...etc.

3.7.2 Analyse qualitative des CBS

La majorité des travaux proposés dans ce contexte repose sur une approche partant des comportements des composants d'un système et modélisant ces comportements en utilisant un formalisme donné. Le formalisme peut être de bas niveau comme le modèle des systèmes de transitions étiquetées (LTS), ou bien de haut niveau comme les algèbres de processus ou les réseaux de Petri. Les propriétés qualitatives recherchées sont ensuite vérifiées par des techniques de model checking et logique temporelle ou des techniques d'analyse structurelle.

Afin de mieux expliciter ces approches, examinons les travaux qui ont été proposés pour certains modèles de composants introduits en section 2, classifiés selon le formalisme utilisé.

3.7.2.1 Analyse par modèle de bas niveau (LTS)

Analyse d'applications .NET, 2002 Une méthode pour l'implantation de spécifications comportementales d'interface sur la plateforme .NET a été proposée dans [17], exprimées sous-forme de programmes de modèles exécutables. Ces programmes sont soit exécutés sous-forme de simulations stand-alone, ou bien utilisés comme des contrats afin de vérifier la conformité d'une classe d'implantation avec sa spécification. La vérification des contrats, dite *runtime verification*, est dynamique. Elle contrôle l'exécution d'un composant dans le cadre de l'exécution d'un programme ayant des données spécifiques en entrée. Pour exprimer des programmes modèles, décrire leurs contrats et spécifier des propriétés sur l'interaction entre composants, le langage de spécification AsmL (*Abstract State Machine Language*) est utilisé. AsmL emploie des machines d'états abstraites (ASMs) [96], qui se basent elles-mêmes sur des systèmes de transitions dont les états sont des algèbres de premier ordre. AsmL est également utilisé pour analyser et vérifier le comportement d'un composant écrit dans tout langage supporté par .NET, tel que VB, C# ou C++, et pour la génération semi-automatique de cas de tests.

Model-checking de systèmes CCM, 2003 L'analyse de systèmes CCM a d'abord été précédée par l'étude de systèmes publish/subscribe, dits *à invocation implicite* [85]. Dans ce travail, Garlan et Khersonsky ont décrit une approche de model checking pour ces systèmes à événements, mais ils ne fournissent aucune information de performance.

Par la suite, un environnement de développement et vérification, appelé *Cadena* [104], a été conçu pour la construction de systèmes distribués embarqués à temps réel, basés sur le modèle CCM. L'architecture typique de ces systèmes est un ensemble de composants faiblement couplés communiquant à travers des couches de middleware qui cachent les complexités associées à l'envoi/réception des données. L'idée de base est de partir des spécifications de composants, faire l'extraction de modèles de systèmes de transitions, puis vérifier des propriétés de correction sur le modèle global. Dans cet objectif, Cadena supporte des formes multiples de spécification de comportement des composants et de leurs dépendances, en plus de la spécification des composants à l'aide du langage CCM IDL. Il offre une variété de techniques d'analyse comportementale, variant entre des analyses statiques légères telles que la vérification de dépendances et la vérification de propriétés (model checking), focalisant sur les propriétés de sûreté et de séquencement d'événements.

Initialement, la vérification de propriétés dans Cadena s'est basée sur l'utilisation du model-checker *dSpin* [68]. Pour ce faire, les descriptions de systèmes CCM sont traduites dans le langage input de *dSpin*. Plus tard, comme dSpin ne permet pas la modélisation de certaines propriétés telles que le threading et les politiques de scheduling et n'utilise pas de techniques de réduction d'états, un nouveau model-checker appelé *Bogor* [185; 186] a été développé. *Bogor* autorise une modélisation directe des systèmes réels

basés sur les événements, appropriée au model checking. S'inspirant des travaux de [85], des descriptions à trois niveaux sont utilisées :

1. Des descriptions sémantiques des composants constituant le système à étudier. Les composants sont modélisés en utilisant un langage de modélisation simple basé transition, tel que *Promela* [107]. Pour exprimer les propriétés sémantiques des composants, spécifier les dépendances et décrire les transitions du système, Cadena définit une *spécification de propriétés de composants* (CPS).
2. Des modèles de l'infrastructure middleware implantant les mécanismes d'événements et de threading réutilisables dans les applications. Bogor offre des possibilités de représentation de files d'attente événements basées sur la priorité et de personalisation des stratégies d'ordonnancement (scheduling) utilisées dans le middleware RT CORBA. L'environnement (entités interagissant avec le système) est également modélisé.
3. Des actions de connexion qui spécifient la topologie d'interconnexion entre les composants.

La création des modèles de composants et de services middleware est faite à l'initialisation du système par une séquence de déclarations de créations d'objets correspondant à ces modèles. L'initialisation est suivie d'une séquence d'actions de connexions qui passent les références appropriées pour établir la connectivité. Pour cela, Cadena définit une *description d'un assemblage* (CAD), qui spécifie les allocations d'instances de composants et les informations de connexions d'une manière graphique, textuelle ou basée forme.

La spécification des propriétés de correction des conceptions Cadena se divise en deux parties :
(i) La définition des observations du comportement du système que l'utilisateur veut construire, et
(ii) La définition de patterns (modèles) d'observations constituant le comportement correct du système.
Ces propriétés sont exprimées sous-forme d'assertions, d'invariants et de propriétés de séquencement sur des états du système modélisé et sur des actions telles que les appels et retours de méthode et les publications et consommations d'événements, de manière identique à l'approche proposée dans [58], basé sur le langage de spécification *Bandera*.

Analyse comportementale de systèmes Fractal, 2005 Cette approche a été proposée dans le but de décrire et vérifier le comportement d'applications réelles, concurrentes, distribuées et asynchrones, construites à base de composants. En particulier, ce travail [18; 20; 19] a traité la vérification formelle des systèmes basée composants Fractal, construits avec l'implantation Fractive [22]. Deux formalismes ont été combinés à cet effet : les réseaux synchronisés et paramétrés d'automates communicants ((*Parameterized Synchronisation Network, pNet*)) [11; 12] et les graphes symboliques avec affectations [128].

La méthode procède alors comme suit [20] :

1. Modéliser le comportement de chaque composant Fractal sous forme d'un réseau hiérarchique d'automates communicants :
   (i) Le comportement fonctionnel d'un composant primitif est modélisé par un système de transitions étiquetées *paramétré* (Labelled Transition Systems, pLTSs). Ce formalisme est une extension des LTSs utilisant des paramètres pour l'encodage de données de messages et la manipulation de familles indexées de processus. Le pLTS est soit spécifié par le développeur, ou dérivé à partir de l'analyse de code. Il exprime les appels de méthode et réceptions dans les interfaces, ainsi que les actions internes.
   (ii) Le comportement d'un composant composite est modélisé par un réseau paramétré d'automates synchronisés (pNet), dont les arguments sont les modèles des sous-composants. Un pNet est une forme d'opérateur parallèle généralisé, où chacun de ses arguments est typé par l'ensemble de

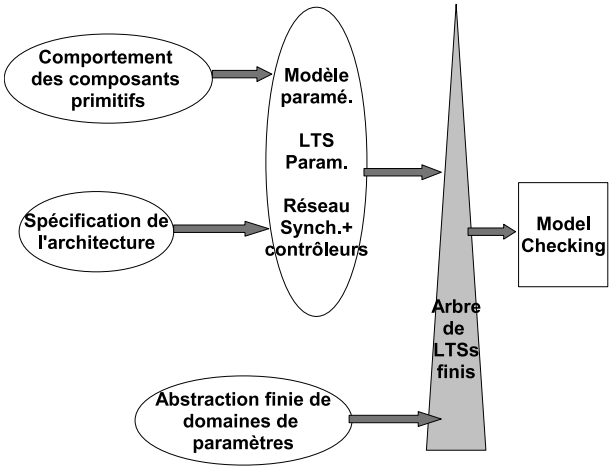


Figure 3.3: Principes de la plateforme Vercors

ses actions possibles observables. En plus du comportement fonctionnel, le modèle d'un composite encode certaines fonctionnalités de contrôle qui ne font pas changer la géométrie du composite, à savoir les opérations start/stop et bind/unbind liées au cycle de vie et aux liaisons. Ces fonctionnalités, extraites de la description architecturale du composant, sont modélisées comme le produit synchronisé de toutes ses parties et est nommé *l'automate contrôleur du composant*. La construction est faite depuis le bas vers le haut de la hiérarchie.

2. Avant de générer le produit synchronisé, des abstractions finies des modèles obtenus sont construites avec des abstractions finies des valeurs des paramètres. Quand l'outil model-checker le permet, l'instanciation est faite à la volée pendant la vérification. Cette abstraction des données est interprétée comme une partition des domaines de données et induit une interprétation abstraite du LTS paramétré. L'instanciation est aussi choisie à partir des valeurs qui sont présentes dans la propriété à vérifier.
3. Le comportement global de l'application Fractal est généré comme le produit synchronisé des LTSs de ses sous-composants avec le contrôleur associé, étant donné que l'application est le composite de plus haut niveau.
4. Le système résultant est vérifié contre des propriétés exprimées en logique temporelle ou avec des LTSs.

L'avantage de cette approche réside dans le fait qu'elle prend en compte, en plus des aspects fonctionnels, les aspects non-fonctionnels liés à l'arrêt de composants et la redéfinition des liens entre composants, ce qui peut affecter considérablement le comportement d'un système.

L'approche a été implantée sous forme d'une plateforme de vérification appelée *Vercors* 3.3. À partir de la spécification comportementale des composants primitifs et de la description architecturale des composants composites, des outils construisent des modèles exprimant les interactions entre composants. Une syntaxe concrète a été développée pour le modèle des réseaux paramétrés d'automates communicants, basés sur le format FC2 [36,118], appelé *FC2 Parameterized*, pour la description de systèmes de transitions étiquetées (LTSs). D'autres outils instancient les modèles paramétrisés en utilisant des abstractions finies et produisent une entrée pour les outils de vérification. La génération des modèles inclut les contrôleurs modélisant la gestion des reconfigurations dynamiques. La plateforme fournit également des outils pour analyser les propriétés comportementales de l'application modélisée.

3.7.2.2 Analyse par modèle de haut niveau (algèbre de processus, PA)

Plusieurs ADLs ont utilisé les algèbres de processus pour la spécification et vérification de comportements de composants. On en cite Darwin/FSP [135; 137; 138], Wright [9; 10] et PADL [26].

Analyse de systèmes Darwin, 1995 Une approche, dite *Tracta* [89], a été développée pour l'analyse d'architectures Darwin. Cette approche décrit les composants primitifs dans une notation d'algèbre de processus FSP (Finite State Processes) [138], vus comme des processus à états finis. Le comportement des composants composites (et de fait d'une architecture logicielle) est alors simplement une composition parallèle des comportements de leurs constituants (sous-composants). La communication est assurée par une synchronisation des actions partagées entre les processus.

Tracta est également équipé d'un ensemble de techniques pour la vérification de propriétés telles que la vivacité et la sûreté [90]. Pour ce faire, la notation FSP est traduite en une description des composants en termes de systèmes de transitions étiquetées (LTS). Pour un composant composite, la traduction de son comportement est une composition parallèle des LTSs associés à ses sous-composants. Lorsque le LTS de l'architecture est obtenu, les propriétés peuvent être vérifiées par model checking. Elles sont exprimées soit en logique linéaire temporelle (LTL) puis traduites en automates de Büchi, ou bien directement exprimées en automates de Büchi.

Cette approche a donné naissance à deux outils :
(i) *SAA, Software Architect's Assistant* [162] qui est un environnement permettant la conception et le développement de programmes distribués en utilisant des descriptions architecturales Darwin. SAA génère les expressions LTS d'un système, en se basant sur l'architecture.
(ii) *LTSA, Labelled Transition Systems Analyser* [138] qui fournit la composition automatique et l'analyse. Cet outil permet d'exécuter une recherche exhaustive sur l'espace d'états du modèle obtenu d'un système, en appliquant une analyse d'accessibilité compositionnelle, fondée sur les descriptions LTS des composants primitifs.

Analyse de systèmes Wright, 1997 Afin de vérifier un système Wright, Allen et al. [9; 10] ont proposé d'exprimer les descriptions comportementales des composants (interface et calcul) et de leurs interactions à l'aide d'une variante du formalisme CSP. Cette variante inclue les processus, les événements de communication et les opérateurs de préfixage, choix, décision et parallélisme. Des outils logiciels dédiés permettent d'effectuer des analyses sur une spécification CSP de processus (ports, rôles et glues). Un simulateur est alors utilisé pour exécuter les configurations décrites dans l'ADL Wright. Les principales vérifications pouvant être effectuées sont la consistence entre le comportement d'un composant et celui de ses ports, l'absence d'interblocages, la compatibilité d'attachement entre un port et un rôle, etc. De plus, les outils model-checking [55] peuvent être directement utilisés pour vérifier des propriétés temporelles de processus et des relations entre processus.

Analyse de systèmes PADL, 2002 Les travaux proposés pour Darwin et Wright n'étaient pas structurés autour d'un ensemble de directives explicitant les étapes de modélisation et d'analyse.

Pour remédier à cela, un langage *PADL* [26; 6] a été développé, inspiré des deux langages précédents. Ce langage a été construit suite à une révision de l'algèbre de processus classique, afin de permettre la conception hiérarchique de familles de systèmes paramétrisés. Une série de directives a été proposée pour cela. De plus, PADL est accompagné d'une technique permettant la vérification de propriétés arbitraires d'une manière compositionnelle. La vérification est basée sur la notion d'équivalence pour la détection d'erreurs architecturales et la provision d'informations de diagnostic orienté composant.

L'ensemble des directives introduites dans ce travail a été incorporé dans *Emilia*, l'ADL algébrique implanté dans l'outil *TwoTowers* [25]. La technique de vérification a été également implantée dans TwoTowers.

3.7.2.3 Analyse par modèle de haut niveau (RDP)

Les RDPs ont été utilisés par différents travaux pour la vérification formelle des CBS. He et al. [105] ont introduit un cadre de développement général d'architecture logicielle basé sur les RDPs et la logique temporelle. Les RDPs sont utilisés pour visualiser la structure et modéliser le comportement d'architectures logicielles, alors que la logique temporelle est utilisée pour spécifier les propriétés requises pour ces architectures. Dans [226], Ziaei et Agha ont défini les *SynchNets* dont l'idée principale est de séparer les aspects de coordination des aspects de calcul. D'autre part, une stratégie de conception, basée sur les *composants réseaux* ou *net components*, a été proposée par [170] pour des architectures dynamiques. Les net components sont eux-mêmes fondés sur les RDPs et sont utilisés pour modéliser les processus business ainsi que leur composition. Dans ces deux travaux, aucune vérification compositionnelle n'a été étudiée.

Dans la suite, nous passons en revue quelques travaux liés aux modèles de composant IEC61499 et PECOS. Puis, nous présentons un travail adressant une analyse compositionnelle en utilisant le modèle *Réseau de Petri Coloré hiérarchique, HCPN* [108; 109].

Analyse des systèmes PECOS, 2002 Le modèle PECOS a été complété par une modélisation par réseaux de Petri ordinaires et temporisés [163], permettant de détecter les compositions valides, de raisonner sur des contraintes temporelles et de générer des planifications temps réel.

Analyse d'architectures IEC61499, 2004 Le modèle IEC61499 a fait l'objet de plusieurs travaux visant à garantir un comportement correct d'un réseau de blocs fonctionnels IEC61499. Parmi ces travaux, une approche de validation des applications de contrôle IEC 61499 a été proposée dans [216]. Elle s'appuie sur la modélisation d'un bloc fonctionnel à l'aide du formalisme "Net Condition/Event Systems" [132], une extension du formalisme des réseaux de Petri. Par ailleurs, [115] a utilisé le modèle d'automate temporisé pour modéliser les blocs fonctionnels, les ports d'entrée d'événements et le contrôleur d'exécution (ECC) d'une application IEC61499. Une approche d'analyse de l'ordonnançabilité d'un sous ensemble de blocs fonctionnels a été également développée. Elle permet de construire un séquencement hors-ligne des tâches à l'intérieur d'une ressource [116; 118]. Des échéances d'exécution sur chacun des blocs fonctionnels du réseau considéré sont calculées et allouées à partir d'une contrainte globale sur le temps de réponse de bout en bout.

Vérification basée HCPN de systèmes embarqués, 2005 Perkusish et al. [71] ont défini une approche autorisant la spécification et la vérification formelle de systèmes embarqués concurrents basés composants. Cette méthode est fondée sur le modèle HCPN, qui est un modèle compact permettant de visualiser la structure et modéliser l'architecture logicielle d'un système, grâce à la notion d'hiérarchie et de types de données.

Elle se déroule comme suit :

1. L'architecture du système à étudier est formellement définie par un cadre décrit par un HCPN : Les composants sont modélisés à base de réseau de Petri coloré (CPN), puis mis ensemble dans une hiérarchie. Les modèles des composants sont soit nouvellement construits, soit récupérés d'un

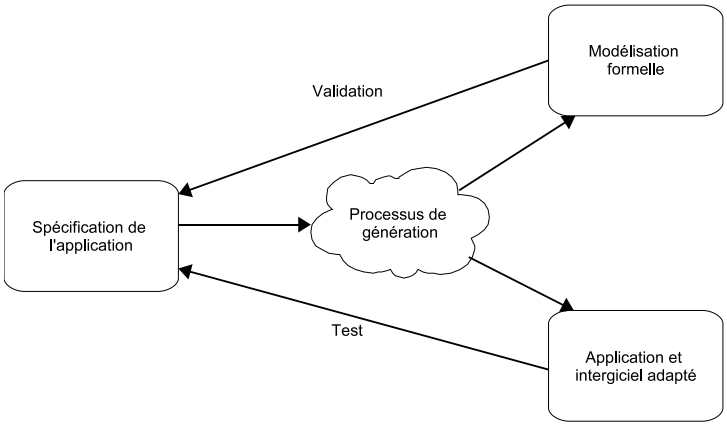


Figure 3.4: Processus de génération det configuration d'un intergiciel adapté

repository dans le cas où ils ont été développés au préalable. La communication entre composants est matérialisée par des places d'interface. Cette forme de modèle de composant est appelée *RDP basé composants ou Component based Petri Net, CBPN*. Pour éditer et analyser les modèles, l'ensemble d'outils Design/CPN [110] est utilisé. Les modèles de composants sont ensuite intégrés ensemble dans le cadre HCPN, résultant en un modèle plat sans modularité.

2. L'étape suivante se concentre sur la vérification. Elle est menée dans l'ordre suivant :
   (i) Identification de scénarios fonctionnels du système.
   (ii) Génération automatique du diagramme de séquences de message (MSC) décrivant chaque scénario. L'ensemble des diagrammes sont automatiquement générés durant la simulation du modèle HCPN.
   (iii) Définition des propriétés à prouver sur la base des scénarios. Comme celles-ci sont reliées à l'architecture et aux interfaces de composants, le modèle est utilisé sans prendre en compte les détails internes de composants.
   (iv) Définition des propositions atomiques et spécification de formules de logique temporelle.
   (v) Application du model checking. La bibliothèque ASK-CTL [52], qui est une logique de type CTL (logique d'arbre de calcul), est utilisée à cet effet. Les preuves sont effectuées au niveau interface des composants. Une spécification, dite *supposer-garantir*, est définie pour chaque composant comme une fonction allant d'une place d'entrée et son marquage initial vers une place output et son marquage. Les modèles de composants sont alors remplacés par leurs spécifications "supposer-garantir" et l'espace d'état est calculé sur la base de cette technique. Cet espace sera plus petit que celui du modèle construit à l'aide des modèles originaux. La preuve de propriétés est alors spécifiée et effectuée de manière modulaire.

Modélisation et vérification de systèmes embarqués basés modèle AADL, 2006 Les travaux de Vergnaud [213] ont permis de dégager une méthodologie pour la construction d'applications fiables et de l'intergiciel (middleware) de communication correspondants, pour le cadre particulier des systèmes temps réel embarqués. Elle consiste en un processus de génération qui produit automatiquement une application répartie vérifiée. Elle se décompose en plusieurs étapes, résumées par la figure 3.4 :

1. La première étape consiste à extraire les caractéristiques de l'application qui doivent être prises en compte pour spécifier l'intergiciel. Ces caractéristiques sont exprimées dans le langage AADL [213;

188]. Le choix de ce langage est motivé par sa capacité à rassembler au sein d'une même notation l'ensemble des informations concernant l'application et son déploiement, et donc concernant également son intergiciel.

2. La description de l'application est ensuite exploitée pour en déduire une configuration adéquate pour l'intergiciel. Il est alors possible d'interpréter cette description pour en extraire une description formelle de l'application et de l'intergiciel, pour mener une analyse et vérification comportementale. Le formalisme utilisé à cet effet est celui des réseaux de Petri. En fonction des résultats de l'analyse formelle, cette étape peut conduire à un raffinement de la description de l'application afin de préciser la structure de l'application.

Afin d'exploiter la description AADL de l'application pour l'analyse et la validation de l'architecture, des règles de traduction vers un modèle Réseau de Petri ont été définies. L'analyse concerne la recherche de dysfonctionnements et erreurs dans l'exécution de l'architecture et la validation de la structure architecturale de l'application vis-à-vis des flux d'exécution et de données. Un dysfonctionnement correspond à l'arrêt de la circulation des données et peut se manifester de plusieurs façons : les communications entre les threads ne sont pas coordonnées, l'absence d'une donnée dans l'exécution des séquences d'un thread, la famine d'un thread, le blocage de l'exécution d'un thread, l'utilisation de données dont la valeur n'est pas déterministe. L'apparition de telles situations dans l'architecture peut être traduite par des propriétés structurelles sur le réseau ou par des formules de logique temporelle. L'analyse est alors opérée par des techniques de model checking.

Plus particulièrement, les propriétés recherchées dans ce travail sont :
- Les données utilisées dans les sous-programmes sont définies de façon déterministe;
- L'assemblage des composants n'engendre pas d'interblocage;
- La structure de l'architecture n'engendre pas systématiquement de débordement de file d'attente;
- La validité des connexions;
- Le respect des contraintes exprimées dans les propriétés AADL telles que les temps d'exécution et les politiques d'ordonnancement.

Les détails de la traduction proposée en réseau de Petri sont donnés en annexe.

3.7.2.4 Autre méthodes d'analyse

Analyse des architectures SOFA, 1998 Le comportement de composants SOFA est modélisé à l'aide de traces ou séquences d'événements se produisant au niveau des connexions d'interfaces de composants. Ces séquences sont approximées et représentées par des expressions rationnelles (ou régulières) appelées *protocoles comportementaux*, qui servent à détecter des erreurs causées éventuellement par une mauvaise composition de composants. Un support pour les protocoles de comportement (behavior protocols) a été intégré dans l'environment SOFA/DCUP, sous forme d'outils automatisés.

Analyse des systèmes Koala, 2002 Le modèle Koala, quant à lui, n'a pas fait l'objet de travaux étudiant l'analyse et la vérification d'une configuration de composants. Toutefois, [80] a introduit une méthode pour l'évaluation des propriétés statiques d'une configuration sur la base des caractéristiques de ses composants. Un exemple de ces propriétés est la consommation de la mémoire. Pour permettre l'évaluation, des propriétés décrivant la demande de ressources sont fournies par les composants à l'aide d'une interface de *réflexion*. La formulation d'une propriété au niveau le plus élevé d'une architecture est alors construite et estimée sur la base des formules des constituants de l'architecture.

3.7.3 Analyse quantitative des CBS

L'analyse des performances des CBS est généralement menée sous-forme de mesures prises sur les systèmes existants. Toutefois, quelques travaux se distinguent dans la littérature, s'appuyant sur l'utilisation de différents formalismes de performance, à savoir les files d'attente (QN), les RDPs, ... La plupart de ces propositions ont été faites pour l'étude de certains modèles de composants. On retrouve également des travaux proposant un processus général d'analyse de performance, inspiré d'une approche globale d'analyse de performances de systèmes logiciels, dite (*Software Performance Engineering, SPE*). Cette approche a été redéfinie pour les systèmes basés composants et est connue sous le nom de *CB-SPE*. Dans la suite, nous présentons d'abord l'approche SPE, vu son utilisation dans le domaine du génie logiciel. Puis, nous survolons les différents travaux d'analyse quantitative des CBS, en les classifiant selon le modèle de performance utilisé.

3.7.3.1 L'approche SPE (Software Performance Engineering), 2000

C'est une approche systématique quantitative [200; 201], visant à construire des systèmes logiciels satisfaisant des objectifs spécifiés de performance. Elle permet de fait de valider *à priori* les choix de conception faits lors des phases initiales du cycle de vie d'un système logiciel et vérifier si les objectifs de performance peuvent être atteints pour réduire ainsi le risque de pannes.

Afin de procéder à l'évaluation d'un modèle de la conception, le processus SPE part de la spécification et des besoins vers l'implantation et la maintenance, en passant par plusieurs étapes :

1. Comprendre les fonctions du système et la fréquence de ses opérations et capturer les besoins en performance. Évaluer le niveau de risque et son impact sur les performances du système.
2. Identifier les cas d'utilisation critiques (les plus importants pour le passage à l'échelle et réactivité pour l'utilisateur).
3. Sélectionner des scénarios clés de performance (exécutés fréquemment ou perçus comme critiques pour les performances).
4. Établir des objectifs de performance : pour chaque scénario clé, spécifier les critères quantitatifs pour évaluer les caractéristiques de performance et les conditions (charge, intensité, ...) sous lesquelles l'objectif doît être atteint.
5. Construire les modèles de performance. Différentes formes de modèles peuvent être utilisées, à savoir les graphes d'exécution (graphes de tâches, diagrammes d'activité, ...), les machines à états communicantes, les réseaux de Petri, les algèbres de processus,...
6. Déterminer les besoins en ressources du logiciel (quantité de ressources logicielles et de traitement pour chaque étape du scénario: temps CPU, opérations disques, services réseau, ...).
7. Ajouter les besoins en ressources machines (charge imposée sur les périphériques utilisés dans les étapes de scénario) et les insérer comme paramètres des modèles construits.
8. Résoudre et évaluer les modèles de performance en utilisant la méthode d'analyse choisie, calculer les prédictions de performance.
9. Comparer les résultats par rapport aux besoins : c'est l'étape de validation.
10. Interpréter les prédictions et suggérer les changements nécessaires pour satisfaire les besoins en performance cités.

Comme des soucis de portabilité des modèles sont également visés, le processus SPE sépare le modèle logiciel (*Software Model, SM*) du modèle d'environnement (*Machinery Model, MM*). Celui-ci se réfère aux informations relatives aux modèles spécifiques de la plateforme. Cette distinction permet de définir des modèles logiciels et d'environnement séparément et de résoudre leur combinaison.

L'approche SPE a été exploitée par Merseguer et al. [24; 146; 147; 133]. Ceux-ci ont proposé de traduire des diagrammes UML d'une conception d'un système logiciel en des modèles analysables de type LGSPN (Labelled GSPN, une extension des GSPN) [5]. Un profil UML particulier a été utilisé à cet effet, documenté d'annotations en performance informant sur l'usage des ressources et les durées de tâches. Une procédure a été également fournie pour composer les LGSPNs correspondants aux diagrammes, obtenant ainsi un modèle global de performance analysable pour le système ou un scénario particulier. Afin d'automatiser ces traductions en GSPN et LGSPN, un outil prototype *CASE* a été développé. Les modèles de performance obtenus sont alors analysés ou simulés en utilisant les outils GSPN [173], pour le calcul d'indices de performance.

Ce travail a fait l'objet de l'étude de plusieurs types de diagrammes : les diagrammes d'activité (AD) [133], les diagrammes utilisation de cas (UC) [147] modélisant l'usage du système pour chaque acteur, les diagrammes de séquence (SD) et les diagrammes d'états (SC) [24; 146].

3.7.3.2 Analyse par modèle de bas niveau (CM)

Peu de travaux ont utilisé le modèle des chaînes de Markov, du fait de la complexité d'analyse induite par des chaînes de grande taille. Nous citons quand même :

1. [225] ont proposé une méthode permettant de calculer la fiabilité des systèmes basés composants. Cette méthode utilise des feuilles de données de composants "sur l'étagère" (COTS), en partitionnant chaque entrée de composant en sous-domaines. Une chaîne de Markov est ensuite créée, puis la fiabilité du système global est calculée sur la base des interactions entre composants.
2. [177] ont proposé une approche pour rendre possible l'analyse de systèmes larges par des modèles markoviens. L'approche consiste à calculer une solution analytique approximative (Mean Value Analysis, MVA) pour des modèles markoviens représentant une composition de composants. Cela consiste à créer des sous-modèles de chaînes de Markov agrégées approximées, et dériver les équations de valeur moyenne (MVA) à partir de ces sous-modèles. Un sous-modèle représente une vue de la chaîne de Markov du système global pour un ensemble de composants. Les solutions analytiques sont ensuite combinées en utilisant les relations identifiées pour chaque système.

3.7.3.3 Analyse par modèle de haut niveau (SPA)

Le modèle SPA a fait l'objet de plusieurs utilisations pour l'analyse de systèmes logiciels, notamment ceux décrits dans le langage UML. Les travaux de [119] en sont un exemple. L'approche décrite consiste à transformer d'une manière systématique des modèles UML annotés par une annotation appropriée vers des modèles d'algèbres de processus stochastiques. Puis, ces modèles SPA sont utilisés pour estimer des paramètres de performance.

Dans le contexte des systèmes à composants, une approche similaire a été développée pour l'analyse de performances des architectures logicielles [16]. Elle est fondée sur l'utilisation de l'ADL *Emilia*, qui offre un cadre formel pour la modélisation compositionnelle, graphique et hiérarchique des systèmes logiciels. Afin d'analyser une architecture, celle-ci est d'abord décrite par un ensemble de spécifications Emilia. La modélisation est basée sur le modèle SPA expressif (EMPA) [36]. Puis, les actions sont dotées d'une temporisation, exprimée principalement par des variables aléatoires exponentielles. La chaîne de

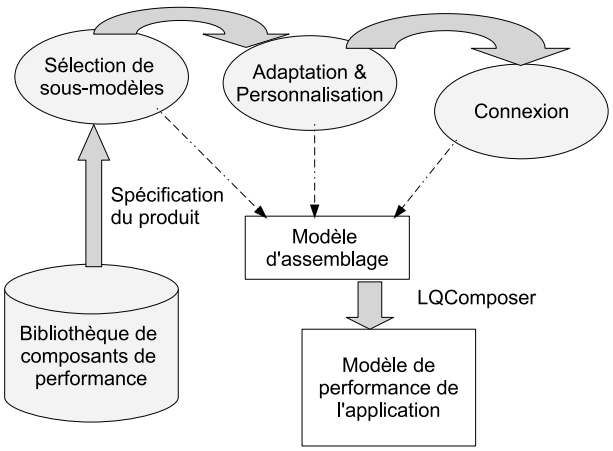


Figure 3.5: Modélisation basée composant de performance

Markov sous-jacente est alors construite puis analysée. Pour cela, Emilia est équipé d'outils de vérifications appropriés, permettant la détection d'éventuelles erreurs architecturales.

L'approche est intéressante. Toutefois, il est possible d'obtenir des chaînes assez larges, difficiles à analyser. Pour pallier à cet inconvénient, les auteurs proposent dans le même cadre de modélisation Emilia, de dériver des modèles de files d'attente, à partir des spécifications Emilia. Ainsi, les deux modèles SPA et QN sont combinés dans Emilia : le modèle SPA permet la vérification de propriétés fonctionnelles des systèmes, et les QNs autorisent une analyse de performance de ces systèmes.

3.7.3.4 Analyse par modèle de haut niveau (QN)

Les travaux basés sur un modèle de file d'attente ont été proposés pour l'analyse de certains modèles de composant tels que EJB. Les files d'attente ont été également utilisées dans l'approche d'ingénierie de performances SPE, adaptée aux CBS (*Component based SPE, CB-SPE*).

Analyse d'architectures EJB La modélisation et analyse des architectures EJB a été adressée dans [129] et [130]. Ces travaux ont défini une approche de prédiction des performances pour des applications quelconques basées sur une plateforme de middleware. L'approche a été restituée dans [130] et [131] pour des applications basées composant côté serveur. L'idée principale consiste à construire un modèle quantitatif de performance pour l'application étudiée, basé sur une description de l'application et sur des informations de profil de performance de la plateforme sous-jacente au modèle de composant. Ces informations sont données sous-forme d'un diagramme d'états arborant les aspects d'attente et d'utilisation des ressources de l'architecture du système étudié, calculés à l'aide d'une application simple de benchmarking. Le modèle quantitatif utilisé à cet effet est un modèle de file d'attente (QN). Les détails des composants de l'infrastructure et leur communication sont abstraits. A partir du modèle QN construit, le concepteur peut mesurer la performance et le passage à l'échelle d'une architecture de l'application et choisir entre plusieurs architectures. Cette méthode a été utilisée pour une application EJB afin de valider des prédictions de ses performances.

Modélisation de performance basée composants, 2002 L'idée de construire "à l'avance" des modèles de performance pour les composants d'un domaine particulier a été proposé par Wu et Woodside [222].

Les modèles seraient stockés dans une bibliothèque de modèles de composants propres à un certain domaine (tel que les services web par exemple). Les modèles prédéfinis de composants pourront alors être réutilisés pour construire le modèle de performance d'un nouveau CBS, en suivant le même principe que l'assemblage des composants. Ce processus est décrit par la figure 3.5 et est appelé *modélisation basée composants*.

La modélisation des composants se dérive à partir de spécifications de composants, décrits en l'occurrence par le langage UML [165]. Elle est faite à l'aide d'un langage dédié, dit (*Component Based Modeling Language, CBML*), basé sur XML [113]. Le formalisme utilisé est celui des files d'attente en couches (LQN) [220]. Un constructeur de modèle (*model builder*) et un outil *LQComposer* ont été développés à cet effet. *LQComposer* permet d'automatiser la génération du modèle LQN d'un système, à partir d'une définition du système, des modèles de composants récupérés à partir de la bibliothèque de composants de performance et des interactions entre composants.

La modélisation basée composants a été appliquée dans des outils tels que *Hyperformix* et *OpNet*. *Hyperformix* utilise des sous-modèles de performance préfabriqués pour le logiciel et le matériel, alors qu'*OpNet* récupère des sous-modèles de performance préfabriqués pour les protocoles réseau et les unités réseau telles que les routeurs, les switchs, etc.

Afin de permettre une construction systématique de modèles basée sur le modèle LQN, le langage de modélisation LQML (*Layered Queueing Modeling Language* [220; 221] a été défini. C'est un langage basé composant (CBML), modélisant un composant par une tâche, une interface offerte (serveur) par une entrée de tâche et une interface requise (cliente) par un appel d'entrée. Le modèle LQN obtenu pour un composant donné est paramétré par les attributs de performance de ce composant, en l'occurrence ses paramètres de charge. On cite notamment :

- Les paramètres d'appel : incluent la méthode ou mode d'appel entre un appelant et un composant invoqué et le nombre moyen d'appels. Le mode d'appel peut être synchrone, asynchrone ou autre.
- Besoins de l'hôte (CPU): exprimés par la quantité de temps moyen (en ms) consommé par l'hôte afin d'accomplir un service représenté par une entrée.
- La politique de scheduling comme FIFO, politique basée priorité, ...

Cette approche a été appliquée par Xu et al.[223] pour la modélisation et prédiction de performance d'une plateforme EJB [204], en proposant des sous-modèles LQN pour chaque type de composant EJB (session, entité, message). Pour modéliser un système EJB, les beans sont d'abord modélisés par des sous-modèles LQN, comme des tâches avec des paramètres estimés auparavant. Puis, les sous-modèles sont instanciés selon des besoins spécifiques d'application, pour envelopper chaque classe de bean dans un container et ajouter l'environnement d'exécution incluant la base de données. Les appels entre beans et les appels à la base de données font partie de l'assemblage final. Les sous-modèles peuvent être calibrés à partir de l'exécution des données, ou en combinant la connaissance sur les opérations de chaque bean et les paramètres de charge pré-calibrés pour les opérations de conteneur et base de données. Les sous-modèles sont enfin assemblés dans un modèle LQN global, désignant le modèle du système cible, utilisé dans l'analyse proprement dite.

L'approche CB-SPE, 2004 S'inspirant de l'approche SPE, un cadre général permettant la validation automatique *à priori* des CBS sur la base de spécifications architecturales a été défini. C'est une approche compositionnelle dont l'idée de base est d'analyser et déduire des propriétés de performance sur un assemblage donné, à partir de l'architecture complète et des modèles de composants constituant l'assemblage. Un langage compositionnel a été introduit [93] pour décrire un assemblage de composants, muni des informations adéquates pour l'analyse de performances. Pour tenir compte de ces informations, des attributs de performance sont définis tels que les temps d'exécution des composants, les temps de

réponse, l'utilisation des ressources, etc.

Notons que l'analyse proposée considère les aspects fonctionnels et non fonctionnels d'un système [28; 29].

L'approche *CB-SPE* opère au niveau de deux couches :

1. La couche *composants* dont l'objectif est d'obtenir des composants avec des propriétés de performance certifiées.
2. La couche *application* qui garantit la construction d'applications basées composants caractérisées par les performances requises.

La couche composants Le principe adopté dans cette approche est de spécifier les aspects de performance clés directement sur des modèles conceptuels du système à étudier, en l'occurrence des modèles UML. Pour ce faire, un profil UML a été proposé [3] pour modéliser les notions quantifiables de temps, d'usage des ressources et d'ordonnançabilité. Il s'agit du profil *RT-UML PA* (Performance analysis). Ce profil a été adopté comme un standard OMG. Il consiste à importer sous-forme d'annotations dans les modèles UML les caractéristiques relatives au domaine ciblé, de telle manière à permettre aux techniques d'analyse d'exploiter ces caractéristiques. Sa définition est telle qu'elle facilite l'utilisation de l'approche, en particulier pour les concepteurs non experts dans les modèles de performance spécialisés, mais familiers avec le language UML.

Comment déduire des prédictions de performance, partant de ce profil ? Un composant est défini comme un ensemble de formes reflétant chacune certains aspects du composant durant le développement de son cycle de vie : spécification du composant, interface, implantation, composant installé, composant objet ... Chaque forme est modélisée par un ensemble de diagrammes UML [49]. Le développeur de composants peut alors annoter ces diagrammes suivant le profil PA. Des exemples d'annotation sont le temps de réponse d'un service spécifique, la demande d'exécution hôte, la probabilité d'exécuter un service donné, le nombre de tâches dans un scénario, le débit des ressources,... Puis, pour chaque comportement de service offert, partant de la spécification RT-UML du composant, il est possible de suivre l'une des approches proposées dans la littérature [27; 60; 176] pour dériver automatiquement des modèles de performance (Software Models SM) paramétrés par les besoins en ressources spécifiés. Pour prendre en compte l'influence de l'environnement, une spécification abstraite et quantifiée de cet environnement est rajoutée dans le modèle de composant développé. Puis, selon le processus SPE, les objectifs de performance sont définis. Le modèle d'environnement est après validé en instanciant les valeurs obtenues dans l'approche indépendante de plateforme contre celles d'une plateforme concrète, afin de vérifier si cet environnement peut supporter le modèle.

La couche application Cette étape a pour but de construire des applications basées composants avec les propriétés de performance requises, en appliquant l'approche SPE. A cet effet, la procédure suivante est appliquée :

```
Procédure de construction d'une application
BEGIN
Input: Ensemble de composants candidats avec leurs annotations paramétriques de
performance sur les interfaces fournies (donnés par le développeur de composants)

Step 1. Déterminer le profil d'utilisation et les objectifs de performance
(assembleur système).
Step 2. Pré-sélection/recherche des composants disponibles qui satisfont les
besoins en performance (assembleur système).
Pour cela, il faut connaître les caractéristiques de la plateforme sur laquelle
les composants vont être déployés.
Step 3. Modélisation et annotation (assembleur système). Les propriétés de
```

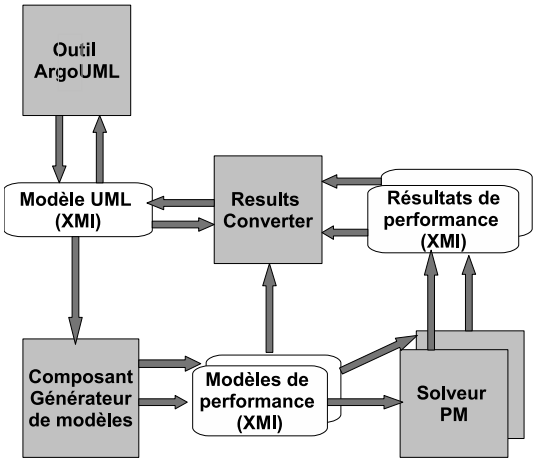


Figure 3.6: Architecture de l'outil CB-SPE

```
performance des composants sont instanciés aux environnements donnés.
Step 4. Analyse du meilleur/pire des cas (automatisée).
Pour chaque couple de candidats SM et MM, répéter
Step5. Génération du modèle CB-SPE (automatisée).
Step 6. Evaluation du modèle (automatisée).
Step 7. Analyse des résultats (assembleur système).

Output : Sélection des composants et de la modélisation finale de l'application
satisfaisant les besoins en performance, acquisition de ces composants et leur
assemblage. Autrement, déclaration de la non faisabilité des besoins en performances.
END
```

Parmi les intérêts de l'approche CB-SPE, la possibilité d'adopter une analyse compositionnelle des propriétés de performance : on peut progressivement intégrer et calculer les paramètres de performance visés au fur et à mesure que les composants sont ajoutés au système.

Un outil a été développé pour réaliser l'approche CB-SPE. L'architecture de l'outil est donné par la figure 3.6 :

1. L'outil *Argo-UML* a été choisi comme l'interface utilisateur pour l'édition UML. Les diagrammes UML standards sont augmentés avec des annotations selon le profil PA. Puis, Argo traite automatiquement les diagrammes annotés et génère un fichier XML/XMI constituant l'entrée d'un module *Générateur de modèle*.
2. Le générateur de modèle (réalisé en Java) fournit, selon le principe de l'approche SPE, deux différents modèles : un modèle global (EG) de performance stand-alone, et un modèle de performance basé sur la congestion (contention), à savoir un modèle file d'attente (QN). Le modèle EG est généré à partir des modèles de scénarios clés de performance représentés par les diagrammes de séquence SD (selon leur probabilité d'occurrence), et leurs vecteurs de requête sont instanciés selon les paramètres d'environnement donnés par le diagramme de déploiement DD. La sortie du générateur de modèle est un document XML représentant le EG.
3. Le module *solveur* est un composant qui applique des techniques standards d'analyse de graphe pour associer un coût total à chaque chemin dans le EG obtenu comme une fonction du coût de chaque noeud qui appartient à ce chemin. Ce résultat d'analyse stand-alone donne pour chaque

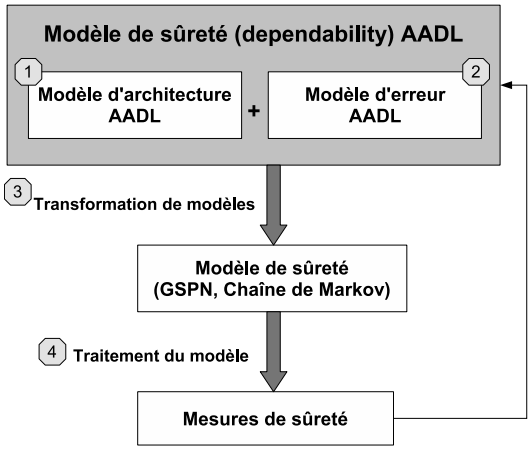


Figure 3.7: Cadre de modélisation AADL

ressource la demande moyenne totale qui, dans le cas optimal, correspond au temps de réponse moyen.

4. La dernière étape est exécutée par le module *Results converter*. Celui-ci reçoit comme input les objectifs de performance d'application en termes d'annotations PA pour les différents scénarios clés de performance et les résultats de performance fournis par le composant solveur. Il utilise également la deuxième sortie du modèle générateur de modèle, et qui est les modèles de files d'attente QN. Les file d'attente sont définies sur la base de la connaissance d'un ensemble de paramètres : (i) le nombre de centres de service et leurs caractéristiques (discipline de service, temps de service), (ii) la topologie du réseau et (iii) les jobs dans le réseau avec leurs demandes de service et routage. Les informations nécessaires pour (i) et (ii) peuvent être dérivées à partir des DDs et SDs annotés. Quant aux caractéristiques des jobs (iii), l'information essentielle est dérivée à partir des modèles détaillés des scénarios de performance représentés dans les EGs. Les différentes classes de jobs sur le réseau modélisent les scénarios clés de performance. La demande du service jobs adressée aux centres de service du réseau peut être calculée à partir des vecteurs de requêtes des ressources du EG. de même, le routage des jobs correspond à la dynamique d'application donnée par le modèle EG.

3.7.3.5 Analyse par modèle de haut niveau (RDP)

Modélisation et évaluation de la sûreté (dependability) d'architectures AADL, 2006 Dans ce travail, une approche itérative a été proposée pour la modélisation de la sûreté (dependability) de systèmes critiques à temps réel, basée sur le langage AADL [188]. L'approche fait partie d'un cadre complet permettant la génération de modèles d'évaluation de performance à partir de modèles AADL, afin de supporter l'analyse d'architectures logicielles. Le modèle de performance utilisé est celui des réseaux de Petri Stochastiques généralisés (GSPN). Ce cadre global, décrit par la figure 3.7, est constitué de quatre étapes principales:

1. La première étape est consacrée à la modélisation de l'architecture de l'application AADL (donnée en termes de composants et de modes opérationnels de ces composants). Pour ce faire, un modèle de sûreté AADL est d'abord construit et validé itérativement, en prenant en compte progressivement les dépendances entre composants. Puis, ce modèle de sûreté est transformé pour obtenir le

modèle GSPN global du système.

- La construction du modèle de sûreté se fait en construisant dans une première itération, les modèles des composants du système en les isolant de leur environnement. Ainsi, les modèles représentent le comportement des composants en présence de leurs propres fautes et des événements de recouvrement uniquement. Dans les itérations suivantes, les dépendances entre les modèles d'erreurs de base sont introduites progressivement pour obtenir le modèle de sûreté.
- Les modèles AADL sont transformés en modèles GSPN. Le modèle GSPN du système est alors structuré comme un ensemble de sous-réseaux interagissant, où un sous-réseau est soit associé à un composant ou à un bloc de dépendance identifié dans le diagramme de blocs de dépendances. Le sous-réseau associé à un composant décrit son comportement en présence de ses propres fautes et des événements de réparation. Le sous-réseau de dépendance modélise le comportement associé à la dépendance correspondante entre au moins deux modèles d'erreur dépendants.

2. La seconde étape concerne la spécification du comportement de l'application en présence de fautes, à travers les modèles d'erreurs associés aux composants du modèle d'architecture. Le modèle d'erreur de l'application est une composition de l'ensemble des modèles d'erreurs des composants. Le modèle d'architecture et le modèle d'erreur de l'application forment le modèle de sûreté AADL.
3. La troisième étape vise à construire un modèle d'évaluation de sûreté analytique à partir du modèle de sûreté AADL basé sur des règles de transformation du modèle.
4. La quatrième étape consiste au traitement du modèle d'évaluation de sûreté AADL. En d'autres termes, des mesures quantitatives caractérisant les attributs de sûreté sont évaluées. Cette étape est entièrement basée sur des outils existants.

3.7.3.6 Autres approches d'analyse

Estimation des ressources dans des systèmes Robocop Les auteurs de [112] ont proposé une méthode pour l'estimation de propriétés de ressources des CBS développés avec une variante du modèle Koala, RoboCop [184]. La méthode modélise le comportement des composants avec des *diagrammes de séquences de messages* (*Message Sequence Charts, MSC* [210], puis un modèle global de comportement de l'application est construit en composant les modèles de comportements des composants. Une prédiction basée scénario est alors menée : l'estimation des ressources est effectuée à l'aide d'un ensemble de scénarios représentant les exécutions plausibles/critiques du système à analyser.

Analyse d'ordonnancement et fiabilité des architectures MétaH L'analyse d'applications et de processus MétaH a principalement porté sur des analyses d'ordonnancement et de fiabilité. L'analyse de l'ordonnancement permet de vérifier et/ou observer la validité d'un ordonnancement des processus, le temps moyen du processeur utilisé par chaque processus et des informations sur leur période d'exécution (respect d'échéances, ..). L'analyse de la fiabilité permet de déterminer la probabilité de panne et la tolérance aux fautes d'une *application* sujette à l'arrivée d'erreurs aléatoires.

Les procédures d'analyse n'ont pas recours à un modèle de performance donné. Elles sont plutôt basées sur la définition de propriétés non fonctionnelles appelées *attributs*. Ces *attributs* font partie d'un vocabulaire spécifique à chaque procédure d'analyse, et sont établis d'une manière empirique. Ils sont valués avec des modèles d'erreur. Ces derniers spécifient les types de fautes qui peuvent survenir aléatoirement, les états d'erreur dans lesquels les composants logiciels et matériels peuvent se trouver lors d'une faute, ainsi que la réaction des composants logiciels à l'occurrence d'une faute (prise en charge ou

propagation). Un ensemble d'outils logiciels [126] utilisant ces spécifications a été développé à cet effet, proposant aux développeurs des procédures d'analyse/validation d'une architecture.

3.8 Conclusion : Modèle SWN

Ce chapitre résume les techniques d'analyse qualitative et quantitative des systèmes, ainsi que les formalismes de description utilisés dans ce cadre. En particulier, nous avons dressé un état de l'art concernant les approches proposées pour l'analyse des systèmes basés composants.

Comme nous nous intéressons à définir un cadre formel générique pour l'analyse de performances des CBS, il est impératif de bien choisir le modèle de performance à utiliser, tout en restant efficace lors de l'analyse proprement dite. Le modèle choisi doit être basé état afin de pouvoir évaluer des indices de performances inhérents à des configurations stables de systèmes. Il doit également être capable d'exprimer tout type de comportement.

Dans cet objectif, les travaux proposés dans la littérature ont utilisé différents formalismes. Toutefois, plusieurs insuffisances de ces travaux ont été soulevés. En effet, la plupart des travaux proposés sont soit définis pour certains cadres basés composants tels que le cadre AADL, ou bien sont basés sur des modèles non expressifs tels que les réseaux de files d'attente ou les algèbres de processus. Il est vrai que les modèles de files d'attente et chaînes de Markov sont relativement légers et représentent des modèles analytiques basiques rapidement résolus. Néanmoins, les formes de base des réseaux de files d'attente sont également limités aux systèmes utilisant une ressource à la fois. Quant aux modèles d'attente étendus qui décrivent l'utilisation de plusieurs ressources (tels que les files d'attente en couches, LQN), ils sont plus complexes à construire et plus longs à résoudre. Globalement, l'inconvénient majeur des modèles de files d'attente et des chaînes de Markov réside en leur manque d'expressivité : ils ne permettent pas d'exprimer les propriétés complexes des systèmes actuels, telles que la synchronisation, le parallélisme et les situations de conflits.

Il en est de même à un moindre degré des algèbres de processus. Ceux-ci souffrent plutôt de la difficulté d'analyse lorsque la taille des systèmes étudiés est importante.

De leur côté, les réseaux de Petri sont des modèles basés état, suffisamment connus pour leur pouvoir d'expression des propriétés des systèmes complexes. La version stochastique est très utilisée dans le domaine d'évaluation des performances. De plus, même si les RDPs ne constituent pas par eux-mêmes un modèle compositionnel, l'interaction entre réseaux pouvant modéliser des composants peut être aisément définie par la fusion de transitions ou de places.

Par ailleurs, si les composants du système à analyser sont complexes et caractérisés par des symétries de comportement, les RDPs de haut niveau constituent inévitablement le modèle de performance le plus approprié. Pour ces raisons, le modèle *réseau stochastique bien formé (Stochastic Wellformed Net, SWN)* est préconisé. Les SWNs sont une classe de RDPs stochastiques de haut niveau bénéficiant d'une large panoplie de résultats numériques, d'algorithmes et d'outils d'analyse. Afin de bien voir ces bénéfices, nous présentons les détails de ce modèle dans le prochain chapitre.

CHAPITRE 4

Modèle SWN

4.1 Introduction

L'identification des symétries de comportement dans un système permet d'une manière considérable de diminuer la complexité d'analyse de ce système, en recourant à l'agrégation des états. Parmi les modèles introduits à cet effet, les réseaux de Petri bien formés sont une forme structurée des réseaux de Petri colorés. L'intérêt de ce modèle réside dans sa puissance d'expression équivalente aux réseaux colorés généraux, mais aussi dans la possibilité d'obtenir de façon automatique les symétries intrinsèques au système modélisé et représenter efficacement son graphe d'accessibilité. Le graphe obtenu est plus réduit et condensé que le graphe ordinaire d'accessibilité.

Ainsi, le modèle SWN permet de réduire la complexité d'analyse pour des modèles générant un espace d'états important. Toutefois, même si un gain est obtenu grâce à ce modèle, il reste encore des systèmes difficiles à analyser, voire impossible à analyser. Afin de résoudre ce problème, des travaux ont proposé de recourir à des techniques de décomposition des RDPs en sous-réseaux plus réduits pouvant être analysés, puis les solutions des sous-réseaux sont combinées en une solution du réseau global. Une méthode de décomposition structurée a été également définie pour analyser un SWN global.

Nous présentons dans ce chapitre les principaux concepts liés aux réseaux de Petri bien formés, puis nous détaillons les réseaux de Petri stochastiques bien formés qui sont l'extension stochastique du premier modèle. Nous décrivons après la méthode de décomposition structurée proposée pour cette classe de réseaux, qui se base sur la communication synchrone ou asynchrone entre sous-réseaux [97; 98].

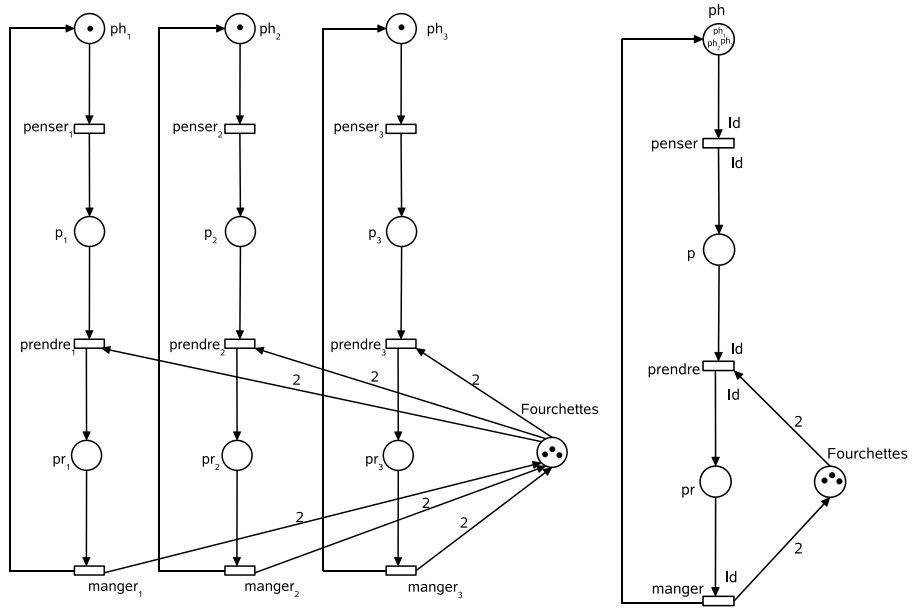
4.2 Le modèle bien formé WN

Le modèle Réseau de Petri bien formé est un réseau de haut niveau (coloré), ayant été la base pour l'introduction de méthodes d'analyse numérique efficaces. C'est un réseau coloré, muni d'un typage structuré des places et des transitions. Nous présentons d'abord informellement les réseaux colorés, puis nous enchaînons avec le modèle bien formé.

4.2.1 Les réseaux de Petri colorés

Les réseaux de Petri de haut niveau (Predicate/transition-nets [88], Coloured Petri Nets [111; 109]) n'étendent pas le pouvoir de modélisation des réseaux places/transitions, mais offrent une représentation plus compacte des systèmes larges constitués de plusieurs "composants" similaires en comportement. Ces 'parties" similaires sont condensées en une même partie de réseau de haut niveau tout en conservant ou en étendant les outils d'analyse existants (invariants, ..).

Essentiellement, les réseaux colorés (CPN) sont constitués de :



(a) Réseau ordinaire (b) Réseau coloré

Figure 4.1: Modèles coloré et ordinaire des philosophes

- jetons possédant une couleur d'un ensemble donné, qui permet de modéliser les caractéristiques de différentes entités (couleurs) d'un même type (ensemble de ces couleurs).
- places et transitions caractérisées par un domaine de couleurs : le domaine de couleurs d'une place représente le type de jetons qu'elle peut contenir, alors que celui d'une transition définit le type des valeurs employées pour son franchissement : le choix d'une couleur de ce type donne une instanciation de la transition pour laquelle on teste le franchissement (et que l'on franchit éventuellement).
- fonctions d'arcs du réseau, utilisées pour tester le franchissement d'une transition : étant donné une couleur c, on évalue ces fonctions dont le résultat est un multi-ensemble de jetons colorés des places d'entrée (et de sortie).

Exemple 4.21. *Le réseau de la figure 4.1(b) constitue un exemple élémentaire d'un réseau de Petri coloré modélisant un système à trois philosophes se partageant trois fourchettes. : Le domaine de couleur des places ph, p et pr et des transitions est l'ensemble* $\{ph_1, ph_2, ph_3\}$ *des philosophes. Le domaine de la place* Fourchettes *est neutre (jetons non différenciés). Pour pouvoir manger (passer à la place pr), un philosophe* ph_i *doit prendre deux jetons de la place* Fourchettes *en franchissant la transition* prendre*. Lorsqu'il termine de manger (franchissement de la transition* manger*, il remet les deux jetons dans* Fourchettes*. La sélection de l'un des philosophes pour prendre des fourchettes est réalisée par le choix de la couleur de tir de la transition* prendre *(et de la transition* manger *à la fin). Le choix est modélisé par les fonctions d'arcs données par la fonction* identité *(Id), telle que Id(*ph_i*)=*ph_i*.*

L'opération de *dépliage* d'un réseau de Petri coloré consiste à construire le réseau de Petri ordinaire équivalent. Le réseau ordinaire de l'exemple est donné par la figure 4.1(a).

Les réseaux de Petri bien formés (Well-formed nets) [76; 51] représentent une catégorie de réseaux

colorés particulièrement adaptée à la modélisation des systèmes fortement symétriques. L'idée de base derrière l'introduction de ce type de réseau est l'exploitation des symétries afin de représenter efficacement le graphe des marquages et définir et calculer les extensions des propriétés structurelles (invariants, ...). Ceci a nécessité de poser des *restrictions syntaxiques* aux constituants d'un réseau (domaines de couleurs, fonctions d'arcs), ce qui est expliqué dans la section suivante.

4.2.2 Couleurs, classes et domaines de couleurs

Dans le modèle réseau bien formé [76], une structure uniforme est imposée aux couleurs des jetons. Les objets d'un système ayant le même comportement sont regroupés en classes. Nous pouvons citer par exemple une classe de serveurs ou de processeurs. Les classes d'objets peuvent être combinées pour former des domaines de couleur. Un domaine de couleurs est associé aux places et aux transitions. Pour une place, il représente le *type d'objets* qu'elle peut contenir. Pour une transition, il définit le *type des paramètres* qui seront instanciés lors du franchissement de la transition.

Plus précisément, un domaine de couleurs correspond à un produit cartésien de *classes de couleurs de base* représentant des domaines élémentaires. En effet, une couleur (d'une place ou transition) est en général une association d'objets éventuellement de même type. Par exemple, une couleur peut définir une association (serveur, client, ressource). Les classes de base seront les classes des serveurs Serv, des clients Cl et des ressources Ress. Le domaine d'une place/transition sera alors Serv xCl x Ress. Un domaine peut aussi comporter plusieurs fois la même classe de base (modélisant des synchronisations internes à cette classe). Le marquage initial d'une place sera alors défini par un multi-ensemble (bag) de jetons colorés.

Formellement, notons l'ensemble C des classes de couleurs de base par :

$$C = \{C_1, \ldots, C_h, C_{h+1}, \ldots, C_n\}, 0 \leq h \leq n \text{ , où } i \neq j \Rightarrow C_i \cap C_j = \Phi,$$

En considérant $I = \{1, \ldots, n\}$ l'ensemble des indices des classes de couleurs de base, et soit $J = \{e_1, \ldots, e_n\}$ où e_i désigne le nombre d'occurrences de C_i dans le produit cartésien, un domaine de couleurs d'une place p ou d'une transition t est défini comme suit :

$$C(p) \text{ ou } C(t) = \Pi_{i=1}^{n} \Pi_{j=1}^{e_i} C_i = \Pi_{i=1}^{n} C_i^{e_i}$$

Une couleur $c = (c_1^1, \ldots, c_1^{e_1}, \ldots, c_i^1, \ldots, c_i^{e_i}, \ldots, c_n^1, \ldots, c_n^{e_n})$ de $\Pi_{i=1}^{n} C_i^{e_i}$ est notée $(c_i^j)_n^{e_i}$. M(p)(c) désignera le nombre de jetons de couleur c du marquage M dans une place p.

Souvent, les objets d'une classe ont le même comportement qualitatif et quantitatif. Cependant, quelquefois, ceci n'est pas vrai comme le cas de processus ayant des vitesses d'évolution différentes, ou bien des réseaux où une station joue un rôle particulier. Dans ce cas, la classe, contenant des objets de même type est divisée en sous-classes statiques, notées $D_{i,q}$, où chaque sous-classe regroupe des objets ayant exactement les mêmes caractéristiques, y compris en termes de performances. La partition d'une classe d'objets C_i en n_i sous-classes statiques est définie par : $C_i = \bigcup_{q=1}^{n_i} D_{i,q}$. On peut prendre l'exemple d'une classe de base de clients C_1, partitionnée en deux sous-classes $D_{1,1}$ de clients prioritaires et $D_{1,2}$ de clients ordinaires, et une autre classe de base de serveurs C_2.

Une classe peut contenir une seule couleur appelée couleur neutre (ε), modélisant des entités que l'on ne souhaite pas distinguer, ou bien représentant des états booléens du système tels que "bus libre/occupé". Par conséquent, un domaine peut-être neutre.

Une classe peut également être ordonnée (existence d'une relation d'ordre entre les objets qui la composent telle qu'un ordre temporel ou géographique), cet ordre étant exprimé par une fonction successeur.

4.2.3 Fonctions d'arc et prédicats standards

Une fonction de couleurs est définie de manière structurée pour chaque arc et spécifie les franchissements de la transition attachée à l'arc : pour une couleur donnée de la transition, la fonction définit le nombre de jetons colorés à ajouter à ou à retirer de la place correspondante. Elle peut être utilisée pour faire des choix (comme la sélection d'un processeur), des diffusions comme un broadcast ou envoi de message), etc.

Formellement, une fonction de couleur standard étiquetant un arc reliant une place p à une transition t, est définie par :

$$W^*(p,t) : Bag(C(t)) \rightarrow Bag(C(p))$$
$$W^*(p,t) = \sum_k \delta_k . g_k$$

où δ_k est un entier positif, g_k est une fonction gardée et Bag(E) est l'ensemble des multiensembles sur E. $Bag(E) = \{x = (x(e))_{e \in E} / \forall e \in E, x(e) \in \mathbb{N}\}$ (un multiensemble est un ensemble qui peut contenir plusieurs occurrences du même élément).

Une fonction d'arc (de C(t) dans Bag(C(p))) est définie à partir d'opérateurs standards (combinaison linéaire, composition, etc.) de fonctions de base. Il existe trois fonctions de base :

1. La projection (notée X ou X_i^j dans les figures) sélectionne un élément d'un tuple; elle est représentée par une variable typée ou par X si aucune confusion n'est possible.
2. La synchronisation/diffusion (notée Si ou Si,k) renvoie l'ensemble de toutes les couleurs d'une classe (Si) ou d'une sous-classe (Si,k).
3. La fonction successeur, notée $!X_i^j$, est définie pour les classes ordonnées et renvoie la couleur qui suit une couleur donnée.

Une transition (respectivement une fonction d'arc) peut être gardée par une *garde* (resp. un *prédicat*). Une garde ou un prédicat est une expression booléenne. Elle permet de limiter le franchissement de la transition gardée et est évaluée sur les couleurs instanciées dans le domaine de couleur de la transition. Les prédicats/gardes sont construits à partir de prédicats atomiques (élémentaires) sur les classes de base, à l'aide des opérateurs logiques usuels. Il existe cinq prédicats élémentaires :

1. true : la constante vrai.
2. $X_i^j = X_i^k$: le même objet est sélectionné pour la jième et la kième instanciation de la classe C_i : $X_i^j = X_i^k$ est vrai ssi $c_i^j = c_i^k$.
3. $X_i^j = !X_i^k$: l'objet sélectionné pour la jième instanciation de la classe ordonnée C_i est le successeur de celui choisi pour la kième instanciation : $X_i^j = !X_i^k$ est vrai ssi $c_i^j = !c_i^k$.
4. $d(X_i^j) = q$: l'objet choisi pour la jième instanciation de Ci est dans la sous-classe $D_{i,q}$: $d(X_i^j = q$ est vrai ssi $d(c_i^j) = q$.
5. $d(X_i^j) = d(X_i^k)$: les objets choisis pour les jième et kième instanciation de Ci sont dans la même sous-classe : $d(X_i^j) = d(X_i^k)$ est vrai ssi $d(c_i^j) = d(c_i^k)$.

4.2.4 Définition des réseaux de Petri bien formés

La définition formelle d'un réseau bien formé WN est donnée comme suit:

Définition 4.1 (Réseau de Petri bien formé (Well-formed Petri Net, WN))**.** Un réseau de Petri bien formé [51] S est un tuple $(P,T,C,cd,Pre,Post,Inh,Guard,Pri,M_0)$ avec:

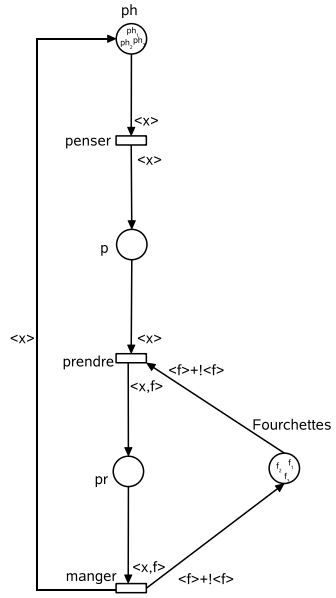


Figure 4.2: Modèle SWN des philosophes

- P, T : ensembles finis des places et transitions,
- $C = \{C_i / i \in I = \{1, \cdots, n\}\}$: ensemble des classes de couleurs de base ; C_i peut être partitionnée en n_i sous-classes statiques : $C_i = \bigcup_{j=1}^{n_i} C_{i,j}$,
- $cd : P \bigcup T \rightarrow Bag(I)$: $cd(r) = C_1^{e_1} \times C_2^{e_2} \times \ldots \times C_n^{e_n}$ est le domaine de couleurs du noeud r; $e_i \in \mathbb{N}$ est le nombre d'occurrences de C_i dans le domaine de couleurs de r;
- $Pre, Post, Inh$: fonctions d'incidence arrière, avant et d'inhibition de $C(t)$ dans $Bag(C(p))$;
- $Guard(t)$: $C(t) \rightarrow$\{true, false\} est le prédicat standard associé à la transition t . Par défaut, $Guard(t)$ est la fonction constante de valeur true;
- $Pri : T \rightarrow \mathbb{N}$ est la fonction de priorité. Par défaut, $\forall t \in T, Pri(t) = 0$;
- $M_0 : M_0(p) \in Bag(C(p))$ est le marquage initial de p.

Règle de franchissement
La règle de franchissement dans un réseau bien formé est identique à celle dans un réseau coloré avec priorités.

Définition 4.2 (Règle de franchissement d'un Réseau bien formé)**.** Une transition t est franchissable pour une couleur c dans le marquage M (noté M[t,c>) ssi :

1. Règle de franchissement sans priorité : $\forall p \in P, Pre(p,t)(c) \leq M(p) \wedge Inh(p,t)(c) > M(p) \wedge$ Guard(t)(c)
2. $\forall t'$ avec Pri(t')>Pri(t), $\forall c' \in$C(t'), t'(c') ne vérifie pas la condition 1 (pas de transition de priorité supérieure franchissable sans priorité).

Le franchissement de l'instance de transition t(c) donne le marquage M' :

$$\forall p \in P, \text{ M'(p) = M(p)+ Post(p,t)(c)- Pre(p,t)(c).}$$

Exemple 4.22. *Reprenons l'exemple des philosophes. La figure 4.2 montre un réseau bien formé du modèle des philosophes. Deux classes de couleurs de base sont utilisées : la classe* $C_1 = \{ph_1, ph_2, ph_3\}$

représentant les philosophes et la classe ordonnée $C_2 = \{f_1, f_2, f_3\}$ des fourchettes. Les domaine de couleur des places et transitions sont respectivement C_1 pour les places ph et p et la transition penser, *C_2 pour la place Fourchettes, $C_1xC_2xC_2$ pour la place pr et les transitions* prendre *et* manger. *Les fonctions d'arcs modélisent le choix d'un philosophe (avec la fonction de projection <x>) et le choix d'une fourchette et de son successeur (avec la fonction <f>+!<f>).*

4.2.5 Flots et semi-flots dans les réseaux bien formés

Les notions de flots et semi-flots constituent des outils fondamentaux pour exprimer les conditions d'application imposées aux modèles que nous développons dans la suite. Pour cela, nous en présentons brièvement les principales définitions. Les flots et semi-flots ont été étendus aux réseaux de haut niveau.

Comme toute fonction de A dans Bag(B) peut être étendue de manière unique en une fonction linéaire de $Bag_Q(A)$ dans $Bag_Q(B)$ où $Bag_Q(E)$ est le Q-espace vectoriel canonique généré par Bag(E), nous pouvons utiliser des applications linéaires dans la définition qui suit.

Définition 4.3 (Flot et semi-flot symbolique dans les réseaux colorés)**.** Soit N un WN de places P et de matrice d'incidence C = Post-Pre (d'applications linéaires de $Bag_Q(C(t))$ dans $Bag_Q(C(p))$.
Un flot (symbolique) de N sur un ensemble A est un vecteur $f = (f_p)_{p\in P} \neq 0$ d'applications linéaires de $Bag_Q(C(p))$ dans $Bag_Q(A)$ telles que :

$$\forall t \in T, \sum_{p\in P} f_p \circ C(p,t) = 0$$

($f_p \circ C(p,t)$ est la composée des fonctions f_p et C(p,t)).

Pour tout marquage accessible M, on a alors :

$$\sum_{p\in P} f_p(M(p)) = \sum_{p\in P} f_p(M_0(p)) = \sum_{a\in A} \alpha(a).a \text{ (constante)}$$

Un semi-flot est un flot à fonctions f_p positives :

$$\forall a \in A, \forall c \in Bag(C(p)), f_p(c)(a) \geq 0$$

4.2.6 Graphe symbolique d'accessibilité

La définition structurée d'un WN permet d'exploiter automatiquement les symétries d'un système, en représentant de manière compacte son graphe d'accessibilité. Une version réduite de ce graphe peut être calculée automatiquement à partir de la définition même du réseau sans le calculer préalablement.

Donc, comment est construit ce graphe réduit d'accessibilité ?

L'idée de base est d'exploiter les symétries du système afin de construire des classes d'équivalences de franchissements, définir des classes équivalentes de marquages, puis choisir un représentant pour chaque classe de marquages et de franchissements, qui seul, figurera dans le graphe. Le graphe ainsi obtenu est plus réduit et condensé que le graphe ordinaire d'accessibilité. Ainsi, le calcul du graphe réduit est réalisé en suivant les étapes suivantes :

- Définition des classes d'états identiques à une symétrie de couleurs près : *les marquages symboliques*.
- Définition d'une représentation unique de ces marquages symboliques, appelée *représentation canonique*.

- Définition d'une règle de *franchissement symbolique*, fondée sur ces représentations : un franchissement symbolique d'un marquage symbolique vers un autre condense un ensemble de franchissements ordinaires.

Les symétries du système modélisé se traduisent dans la structure régulière des éléments qui constituent le modèle. La formalisation des symétries est réalisée à travers la notion de *permutation de couleurs*. Une permutation doit laisser stables les sous-classes statiques et être compatible avec l'opérateur successeur.

Définition 4.4 (Permutation admissible)**.** Soit $C = \{C_1, \ldots, C_h, C_{h+1}, \ldots, C_n\}$ l'ensemble des classes de couleurs de base d'un WN, ordonnées ssi i > h.
$s = (s_1, \ldots, s_h, s_{h+1}, \ldots, s_n) = (s_i)_{1 \leq i \leq n}$ est une permutation (de couleurs) admissible ssi

- $\forall 1 \leq i \leq h$, s_i est une permutation sur C_i telle que $\forall D_{i,q}$, $s_i(D_{i,q}) = D_{i,q}$,
- $\forall h < i \leq n$, s_i est une rotation[1] sur C_i telle que $\forall D_{i,q}$, $s_i(D_{i,q}) = D_{i,q}$ (si $n_i > 1$, cette condition implique que $s_i = Id$).

Soit r un noeud du WN, $c = (c_i^j)_n^{e_i} \in C(r)$ et s une permutation admissible. s(c) est alors défini par :

$$s((c_i^j)_n^{e_i}) = (s_i(c_i^j)_n^{e_i}$$

L'ensemble des permutations admissibles de C est un groupe pour la composition définie par :

$$s \circ s' = (s_i \circ s'_i)_{1 \leq i \leq n}$$

Un SRG se compose de *marquages symboliques*, qui représentent chacun un ensemble de marquages ordinaires ayant des comportements équivalents (voir [51] pour plus de détails). De nombreuses propriétés qualitatives peuvent être vérifiées sur le SRG.

Nous présentons dans la suite l'ensemble des concepts inhérents à la construction du graphe symbolique d'un réseau de Petri bien formé.

4.2.6.1 Marquage symbolique

Un *marquage symbolique* est un regroupement d'un ensemble de marquages ordinaires, basé sur les permutations de couleurs admissibles définies précédemment. La définition d'une permutation est alors étendue aux marquages comme suit, ξ étant un groupe opérant sur les marquages accessibles :

Définition 4.5 (Permutation d'un marquage)**.** Soit M un marquage du WN et $s \in \xi$. Alors $s.M$ est le marquage défini par :

$$\forall p \in P, \forall c \in C(p), (s.M)(p)(s(c)) = M(p)(c)$$

Comme les franchissements sont stables par permutation admissible :

$$\forall s \in \xi, \forall t \in T, \forall M, M' \in RS, \forall c \in C(t), M[t(c) > M' \Rightarrow s.M[t(s.c) > s.M'$$

Des marquages identiques à une permutation près génèrent des sous-graphes du graphe d'accessibilité du réseau également identiques à une permutation près. Le regroupement de ces marquages permettra donc aussi la fusion des sous-graphes dont ils sont la racine, ce qui donne lieu aux marquages symboliques.

Définition 4.6 (Marquage symbolique)**.** Soit $\mathscr{V}$ la relation d'équivalence définie par :

[1] i.e $\forall c \in C_i$, $s_i(!c) = !(s_i(c))$

$$M \mathscr{V} \Longleftrightarrow \exists s \in \xi, M' = s.M$$

Un marquage symbolique, noté $\mathscr{M}$, est une classe d'équivalence de $\mathscr{V}$.

Afin de construire le graphe des marquages symboliques, il est nécessaire de définir une représentation adéquate des marquages symboliques, qui va permettre de représenter de manière unique toute classe de marquage dans le graphe d'accessibilité : c'est la *représentation canonique*.

Une *représentation* décrit un marquage symbolique à l'aide d'un ensemble de sous-classes dynamiques (Z_i^j).

Définition 4.7 (Représentation d'un marquage symbolique)**.** Une représentation $\mathscr{R}$ de $\mathscr{M}$ est un quadruplet $(m, card, d, mark)$:

m **:** $I \longrightarrow \mathbb{N}^*$. $m(i)$ (ou m_i) est le nombre de sous-classes dynamiques de C_i dans $\mathscr{M}$. L'ensemble des sous-classes dynamiques de C_i est donc $\hat{C}_i = \{Z_i^j \mid 1 \leq j \leq m_i\}$[2]

$card$ **:** $\bigcup_{i \in I} \hat{C}_i \longrightarrow \mathbb{N}$;

d **:** $\bigcup_{i \in I} \hat{C}_i \longrightarrow \mathbb{N}$ et :

1. $d(Z_i^j)$ est l'indice q d'une sous-classe statique $D_{i,q}$;
2. $\sum_{d(Z_i^j)=q} card(Z_i^j) = |D_i^q|$;
3. $\forall 0 < i \leq n$, $\forall 0 < j < k \leq m_i$, $d(Z_i^j) \leq d(Z_i^k)$;

$mark$ **:** $\forall p \in P, mark(p) : \hat{C}(r) \longrightarrow \mathbb{N}$;

$\forall M \in \mathscr{M}$ **:** $\forall i \in I$, $\exists \eta_i : C_i \to \hat{C}_i$ telle que :

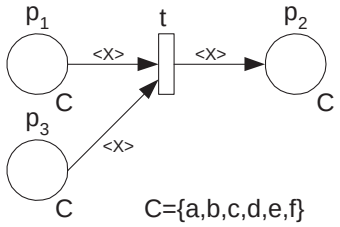
1. $\forall Z_i^j$, $|\eta_i^{-1}(Z_i^j)| = card(Z_i^j)$;
2. $\forall Z_i^j$, $\exists q$, $\eta_i^{-1}(Z_i^j) \subseteq D_{i,q}$ et $d(Z_i^j) = q$;
3. $\forall p \in P$; $\forall c \in C(p)$, $M(p)(c) = mark(p)((\eta_i(c_i^j))_n^{e_i(p)})$;
4. $\forall Z_i^j$, $i > h$, $\exists c \in \eta_i^{-1}(Z_i^j)$ tel que :
   $!c \in \eta_i^{-1}(Z_i^{(j+1) \bmod n}) \wedge \forall c' \in \eta_i^{-1}(Z_i^j)$, $c' \neq c$, $!c' \in \eta_i^{-1}(Z_i^j)$;

Plusieurs représentations d'un marquage peuvent exister. Pour faire le choix d'une représentation canonique unique, deux critères sont vérifiés :

- Minimalité : la représentation doit contenir le moins de classes Z_i^j possible ;
- Ordre : parmi les minimales, un ordre est défini et la plus petite est choisie.

Nous renvoyons le lecteur à [76] pour une présentation exhaustive du calcul de la représentation canonique d'un marquage symbolique.

[2] $\hat{C}(r) = \{(Z_i^j)_n^{e_i(r)} \mid \forall i \in I, \forall 1 \leq j \leq m_i, Z_i^j \in \hat{C}_i\}$ est l'ensemble de tous les tuples possibles de sous-classes dynamiques du nœud r



$$C = \{a,b,c,d,e,f,g,h\}$$
$$\mathscr{R}_1 = (Z_C^1 + Z_C^2).p_1 + (Z_C^3 + Z_C^4).p_2 + (Z_C^2 + Z_C^3).p_3$$

Figure 4.3: Exemple de marquage symbolique

Exemple 4.23. *La figure 4.3 présente un exemple de marquage symbolique et de franchissement symbolique sur un réseau très simple composé de trois places (p_1,p_2 et p_3), une transition (t) et une classe de couleurs de base C contenant huit couleurs $\{a,b,c,d,e,f,g,h\}$ réparties dans une seule sous-classe statique.*
Le domaine de couleurs des trois places est C ainsi que celui de la transition t. Le marquage symbolique est composé de quatre sous-classes dynamiques, Z_C^1 contient 2 éléments, Z_C^2 contient 3 éléments parmi ceux qui restent, Z_C^3 contient 1 élément et enfin Z_C^4 contient 2 éléments. On constate que les places p_2 et p_3 contiennent un élément identique représenté par Z_C^3, c'est pourquoi on ne peut fusionner Z_C^2 et Z_C^3 dans p_3 ni Z_C^2 et Z_C^4 dans p_2.

Définition 4.8 (Ensemble symbolique d'accessibilité)**.** L'ensemble symbolique d'accessibilité (*Symbolic Reachability Set, SRS*) d'un WN est l'ensemble des représentations canoniques de ses marquages symboliques.

Une fois les représentations canoniques des marquages symbolique sont définies, il est nécessaire de définir une règle de *franchissement symbolique* qui opère sur ces représentations.

4.2.6.2 Franchissement symbolique

De manière similaire au regroupement des marquages ordinaires en marquages symboliques, il est possible de regrouper plusieurs franchissements, identiques à une permutation admissible près. Les franchissements étant stables par permutation, on peut définir leur regroupement directement à partir des marquages symboliques, à travers la notion de *franchissement symbolique*.

Un franchissement symbolique correspond à une instanciation de transition : pour chacune des couleurs appartenant au domaine C_i de couleur de la transition, on choisit un élément *quelconque* d'une *sous-classe dynamique* Z_i^j d'une représentation donnée d'un marquage symbolique. Si C_i est présente plusieurs fois dans C(t), il faut préciser pour toutes les occurrences prises dans une même sous-classe dynamique Z_i^j les relations qui les lient. Cette instanciation est dite *instance symbolique*.

Définition 4.9 (Instance symbolique d'une transition)**.** Une instance symbolique de la transition t pour la représentation $\mathscr{R}$ est un couple (λ, μ) avec $\lambda = (\lambda_i)_{i\in I}$ et $\mu = (\mu_i)_{i\in I}$ où

- λ_i et $\mu_i : \{1,\ldots,e_i\} \longrightarrow \mathbb{N}^*$
- $\forall i \in I$, $\forall x \leq e_i$:
  1. $\lambda_i(x) \leq \mathscr{R}.m_i$;

2. $\mu_i(x) \leq \mathscr{R}.card(Z_i^{\lambda_i(x)})$;
3. si $i \leq h$, alors $\forall 0 < l < \mu_i(x),\ \exists x' < x$ tel que $\lambda_i(x') = \lambda_i(x) \wedge \mu_i(x') = l$.

On note $\mu_i^j = \max\{\mu_i(x) \mid \lambda_i(x) = j\}$ le nombre d'instances distinctes dans Z_i^j (0 par définition si Z_i^j n'est pas instanciée).
Si C_i n'est pas dans $C(t)$ ($e_i = 0$), λ_i et μ_i ne sont pas définies.
On note $\langle \lambda, \mu \rangle$ les instances symboliques associées à la représentation canonique du marquage.

$\lambda_i(k)$ sélectionne la sous-classe dynamique de C_i dans laquelle on instancie sa $k$ième occurrence dans $C(t)$: $\lambda_i(k) = j$ ssi on choisit Z_i^j. μ_i précise le choix en cas de multiples occurrences et on impose des choix consécutifs à partir de 1.

Reprenons l'exemple de la figure 4.3, pour lequel on souhaite franchir la transition t de domaine de couleur C, le seul choix de sous-classe dynamique pour le franchissement de t est : x dans Z_C^2 car on doit choisir un élément commun à p_1 et p_3. On obtient l'instance symbolique $\lambda_C(1) = 2, \mu_C(1) = 1$.

Une fois le choix des sous-classes dynamiques effectué, le franchissement s'opère en quatre étapes données par la figure 4.4.

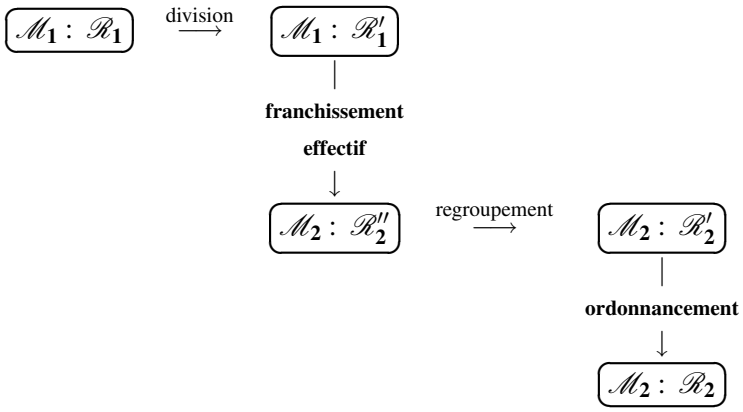


Figure 4.4: Les quatre étapes d'un franchissement symbolique

La première étape consiste à diviser la sous-classe Z_C^2 en deux sous-classes $Z_C^{2,0}$ et $Z_C^{2,1}$ afin d'isoler l'élément selectionné. La première sous-classe contient les deux éléments non sélectionnés pour le franchissement et la deuxième contient cet élément. La représentation de ce marquage est :

$$\mathscr{R}'_1 = (Z_C^1 + Z_C^{2,0} + Z_C^{2,1}).p_1 + (Z_C^3 + Z_C^4).p_2 + (Z_C^{2,0} + Z_C^{2,1} + Z_C^3).p_3$$

Les jetons sont replacés dans les places de sortie de la transition en suivant les fonctions des arcs sortant pendant le franchissement effectif. Dans notre exemple, la fonction du seul arc sortant de t est l'identité sur la variable x, consistant à replacer l'élément instancié $Z_C^{2,1}$ dans p_2. La représentation du marquage obtenu est :

$$\mathscr{R}''_2 = (Z_C^1 + Z_C^{2,0}).p_1 + (Z_C^{2,1} + Z_C^3 + Z_C^4).p_2 + (Z_C^{2,0} + Z_C^3).p_3$$

L'étape suivante regroupe les sous-classes dynamiques d'une même place si cela est possible. Les classes $Z_C^{2,1}$ et Z_C^4 sont regroupables en une seule classe car les éléments qu'elles contiennent ne sont présents dans aucune autre place. Le marquage obtenu après regroupement est le suivant :

$$\mathscr{R}'_2 = (Z_C^1 + Z_C^{2,0}).p_1 + (Z_C^4 + Z_C^3).p_2 + (Z_C^{2,0} + Z_C^3).p_3$$

Il suffit alors de renommer correctement les sous-classes dynamiques pour obtenir la représentation canonique finale.

$$\mathscr{R}_2 = (Z_C^1 + Z_C^2).p_1 + (Z_C^4 + Z_C^3).p_2 + (Z_C^2 + Z_C^3).p_3$$

La cardinalité de la classe Z_C^2 est maintenant égale à 1 et celle de Z_C^4 à 3.

Définition 4.10 (Graphe symbolique d'accessibilité)**.** Le graphe symbolique d'accessibilité d'un réseau bien formé est le graphe dont les nœuds sont les éleḿents de l'ensemble d'accessibilité et les arcs les franchissements symboliques.

4.3 Le modèle stochastique bien formé SWN

À partir des WNs a été dérivé le modèle Réseau de Petri stochastique bien formé (*Stochastic Well-formed Net, SWN*). C'est un WN associant à chaque transition une durée de tir sous forme de variable aléatoire de loi exponentielle $EXPO(\theta(t))$. Comme pour les GSPNs, une transition peut être immédiate à durée de tir nulle (Pri(t)>0), ou temporisée (Pri(t)=0).

Toutefois, le comportement stochastique de ce modèle doit préserver les symétries de couleurs des WNs. En général, les vitesses de tir des transitions peuvent dépendre aussi bien du marquage que de l'instance de couleur choisie pour le franchissement. Pour respecter la symétrie du modèle, ces vitesses sont données en fonction des sous-classes statiques d'où proviennent les couleurs instanciés. On se base pour cela sur la notion de *domaine de sous-classe statique*.

Définition 4.11 (Domaine de sous-classe statique)**.** Soit $C(r) = \prod_{i=1}^{n} C_i^{e_i}$ le domaine de couleurs d'un noeud r. Soit $\widetilde{C}_i = \{D_{i,q} | 1 \leq q \leq n_i\}$ l'ensemble des sous-classes de C_i.
Le domaine de sous-classes statiques (Static Subclass Domain) de r est :

$$C(r) = \prod_{i=1}^{n} \prod_{j=1}^{e_i} \widetilde{C}_i = \prod_{i=1}^{n} \widetilde{C}_i^{e_i}.$$

Pour $c \in C(r), \tilde{c} \in \tilde{C}(r)$, est le tuple de sous-classes statiques auquel chaque c_i^j appartient :

$$\tilde{c} = (D_{i,q_{i,j}})_n^{e_i} \text{ avec } \forall i,j, c_i^j \in D_{i,q_{i,j}}.$$

Comme la vitesse de tir $\theta(t)$ peut être fonction du marquage, il faut restreindre la dépendance du marquage de telle manière à ce que $\theta(t)$ soit constante pour les marquages d'un marquage symbolique. Pour cela, a été définie la partition statique d'un marquage.

Définition 4.12 (Partition statique d'un marquage)**.** La partition statique d'un marquage M est $\tilde{M} \in \prod_{p \in P} Bag(\tilde{C}(p))$ avec :

$$\forall p \in P, \forall \tilde{c} \in \tilde{C}(p), \tilde{M}(p)(\tilde{c}) = \sum_{c', \tilde{c'} = \tilde{c}} M(p)(c')$$

$\tilde{M}(p)(\tilde{c})$ est le nombre de jetons de M(p) dont les composantes sont dans les mêmes sous-classes statiques que c.

A l'aide de ces concepts, nous rappelons ci-après la définition d'un SWN.

Définition 4.13 (Réseau de Petri stochastique bien formé (SWN))**.** Un réseau de Petri stochastique bien formé (SWN) [51] est un couple (S, θ) tel que :

- S est un réseau de Petri bien formé;
- θ est une fonction définie sur T telle que :

$$\theta(t) : \tilde{cd}(t) \times \prod_{p \in P} Bag(\tilde{C}(p)) \longrightarrow R^+.$$

où $\theta(t)(\tilde{c}, \tilde{M})$ représente :

- Le poids de t pour la couleur c dans le marquage M, si $\pi(t) > 0$ (t est immédiate). La probabilité de tir de $t(c)$ en M est alors:

$$\frac{\theta(t)(\tilde{c}, \tilde{M})}{\sum_{(t',c'),M[t'(c')>} \theta(t')(\tilde{c'}, \tilde{M})}.$$

- La vitesse de franchissement de t pour la couleur c en M, si $\pi(t) = 0$ (t est temporisée): la durée de sensibilisation avant franchissement de $t(c, M)$ suit une loi exponentielle de moyenne $\theta(t)(\tilde{c}, \tilde{M})$.

Dans cette définition, $\tilde{c}$ est la représentation de la couleur c en termes de sous-classes statiques, et $\tilde{M}(p)$ est la représentation du marquage symbolique de p en termes de tuples de sous-classes statiques. $\theta(t)$ ne dépend donc que des sous-classes statiques des couleurs en jeu.

Le graphe symbolique SRG d'un SWN, augmenté des informations stochastiques des franchissements, fournit une chaîne de Markov agrégée de la chaîne dérivée du réseau coloré [76]. On peut ainsi étudier les performances d'un système directement sur cette chaîne agrégée, permettant ainsi un gain en temps et en espace mémoire lors de l'analyse du SWN.

Théorème 4.1. *Le processus stochastique sous-jacent au SRG d'un SWN est une chaîne de Markov agrégée de celle du réseau original. De plus, tous les marquages d'un marquage symbolique donné ont la même probabilité à l'équilibre.*

Les réseaux bien formés ont permis un gain en temps et en espace mémoire lors de l'analyse d'un système présentant de fortes symétries de comportement, à travers l'agrégation des états fournie de manière naturelle par le modèle. Toutefois, beaucoup de systèmes restent difficiles à analyser.

Afin d'élargir les possibilités d'analyse des performances des systèmes, des techniques de décomposition et de composition ont été utilisées, ce qui fait l'objet de la section suivante.

4.4 Décomposition des SWNs

Lors de l'analyse d'un réseau de Petri, on s'aperçoit que la taille du graphe des marquages associé augmente très rapidement, donnant ainsi lieu à des ensembles d'états très grands. Dans ce cas, l'analyse devient très difficile, voire impossible à mener. Une approche commune pour résoudre un problème est de partitionner ce problème en parties plus réduites, résoudre ces parties, puis combiner les solutions de ces parties en une seule solution pour le problème initial.

Dans cette même optique, des travaux ont été proposés dans la littérature, basés sur la méthode classique du génie logiciel [75; 155], qui décompose un système global en plus petits composants, étudie en isolation chacun des composants, puis utilise des descriptions tensorielles (de Kronecker) du graphe d'accessibilité global pour calculer des indices des performances du système initial.

Ce type de méthode à été introduit par B. Plateau [181] pour la décomposition des automates stochastiques, puis par S. Donatelli [182; 75] pour la décomposition des réseaux de PETRI stochastiques

généralisés. Plus tard, Moreaux & al. [97; 98; 158] ont étendu la méthode de décomposition aux réseaux de Petri Stochastiques bien formés (SWN). Dans cette méthode, deux types de décomposition ont été étudiées :

- Décomposition en un ensemble de sous-réseaux communiquant par "rendez-vous" en se synchronisant via une ou plusieurs transitions : on parle alors de *décomposition synchrone*.
- Décomposition en un ensemble de sous-réseaux communiquant par échange de jetons à travers des places du réseau de Petri (échange de messages) : on parle alors de *décomposition asynchrone*.

L'analyse de ce méthodes de décomposition est basée sur une algèbre dite *l'algèbre de Kronecker* ou *l'algèbre tensorielle*.

Pour cela, nous explicitons dans un premier temps l'algèbre tensorielle, puis nous présentons les deux types de décomposition synchrone et asynchrone.

4.4.1 Méthode tensorielle

L'algèbre tensorielle [64] permet d'établir une expression tensorielle des matrices des probabilités de transition (respectivement générateur) de processus stochastiques à temps discret (resp. continu) à K composantes. Cette expression rend compte des interactions entre les composantes d'un processus stochastique.

Précisément, prenons K processus stochastiques $(X_{1,t}),(X_{2,t}),\ldots,(X_{K,t})$ à valeurs dans un espace fini S_k , de cardinal r_k . Pour rendre compte de leurs interactions, on considère le processus X défini par $X_t = (X_{1,t},X_{2,t},\ldots,X_{K,t})$, prenant ses valeurs dans $S = \prod_{k=1}^{K} S_k$. On ordonne S selon l'ordre *lexicographique* des composantes : un état $e = (e_1,\ldots,e_K)$ de S dont la composante e_k a pour indice i_k est identifié par l'entier $i = (i_1,\ldots,i_K)$ dans la multibase $(r_1,\ldots,r_K)$.

Deux types de transitions opèrent dans l'évolution du processus global X :

1. Les transitions *locales* à un processus X_k qui ne modifient que l'état de X_k, et
2. Les transitions de synchronisation entre au moins deux processus qui modifient l'état des processus qui se synchronisent.

Définition 4.14.
- Une transition τ est locale à X_k ssi : $i \xrightarrow{\tau} j \Rightarrow \forall k' \neq k, j_{k'} = i_{k'}$
- Une transition τ est de synchronisation ssi :

$$i \xrightarrow{\tau} j \Rightarrow \exists k', k'', k' \neq k'', j_{k'} \neq i_{k'} et\, j_{k''} \neq i_{k''}$$

Les matrices considérées sont à valeurs réelles.

4.4.1.1 Produit tensoriel

Le principe est de multiplier chaque élément d'une matrice par une seconde.

Définition 4.15. Soient deux matrices $\mathbf{A} \in \mathscr{M}_{n_1,p_1}$ et $\mathbf{B} \in \mathscr{M}_{n_2,p_2}$. Le produit tensoriel ($\bigotimes$) de **A** et **B** est la matrice $\mathbf{C} \in \mathscr{M}_{n_1.n_2,p_1.p_2}$:

$$\mathbf{C} = \mathbf{A} \bigotimes \mathbf{B} \text{ avec } c_{i,j} = a_{i_1 j_1} b_{i_2 j_2}$$

où $i = (i_1,i_2)$ dans la multibase (n_1,n_2) et $j = (j_1,j_2)$ dans (p_1,p_2).

Exemple 4.24. *Soient deux matrices* $\mathbf{A} \in \mathscr{M}_{m.n}$ *et* $\mathbf{B} \in \mathscr{M}_{p.q}$,

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{pmatrix} \quad \textit{and} \quad \mathbf{B} = \begin{pmatrix} b_{0,0} & \cdots & b_{0,q-1} \\ \vdots & \ddots & \vdots \\ b_{p-1,0} & \cdots & a_{p-1,q-1} \end{pmatrix}$$

Le produit tensoriel $\mathbf{A} \bigotimes \mathbf{B} \in \mathscr{M}_{mn.pq}$ *est donné par*

$$\mathbf{A} \bigotimes \mathbf{B} = \begin{pmatrix} a_{0,0}\mathbf{B} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1}\mathbf{B} \end{pmatrix} =$$

$$\begin{pmatrix} a_{0,0}b_{0,0} & \cdots & a_{0,0}b_{0,q-1} & \cdots & a_{0,n-1}b_{0,0} & \cdots & a_{0,n-1}b_{0,q-1} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{0,0}b_{p-1,0} & \cdots & a_{0,0}b_{p-1,q-1} & \cdots & a_{0,n-1}b_{p-1,0} & \cdots & a_{0,n-1}b_{p-1,q-1} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{m-1,0}b_{0,0} & \cdots & a_{m-1,0}b_{0,q-1} & \cdots & a_{m-1,n-1}b_{0,0} & \cdots & a_{m-1,n-1}b_{0,q-1} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{m-1,0}b_{p-1,0} & \cdots & a_{m-1,0}b_{p-1,q-1} & \cdots & a_{m-1,n-1}b_{p-1,0} & \cdots & a_{m-1,n-1}b_{p-1,q-1} \end{pmatrix}$$

L'intérêt du produit tensoriel est de pouvoir le gérer en utilisant les sous-matrices sans jamais stocker la matrice complète. Les éléments de $\mathbf{A} \bigotimes \mathbf{B}$ sont donnés par la formule :

$$(\mathbf{A} \bigotimes \mathbf{B}[i,j] = \mathbf{A} \left[\frac{i}{p}, \frac{j}{q} \right] \mathbf{B}[i \bmod p, j \bmod q]$$

Le produit tensoriel s'étend à K matrices :

$$\bigotimes_{k=1}^{K} = M_1 \bigotimes \ldots \bigotimes M_K = M$$

Notons que le produit tensoriel n'est pas commutatif.

4.4.1.2 Somme tensorielle

Afin de décrire de façon tensorielle le générateur d'une chaîne de MARKOV à temps continu de K processus stochastiques synchronisés, on utilise la *somme tensorielle*.

Définition 4.16 (Somme tensorielle)**.** Soient $A \in \mathscr{M}_n$ et $B \in \mathscr{M}_p$ deux matrices carrés. La somme tensorielle ($\bigoplus$) de A par B est la matrice $C \in \mathscr{M}_{n.p}$:

$$C = A \bigoplus B = A \bigotimes I_p + I_n \bigotimes B$$

Exemple 4.25. *Si*

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad et \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

$$A \bigoplus B = \left(\begin{array}{ccc|ccc} a_{11}+b_{11} & b_{12} & b_{13} & a_{12} & & \\ b_{21} & a_{11}+b_{22} & b_{23} & & a_{12} & \\ b_{31} & b_{32} & a_{11}+b_{33} & & & a_{12} \\ \hline a_{21} & & & a_{22}+b_{11} & b_{12} & b_{13} \\ & a_{21} & & b_{21} & a_{22}+b_{22} & b_{23} \\ & & a_{21} & b_{31} & b_{32} & a_{22}+b_{33} \end{array}\right)$$

La somme tensorielle s'étend à K matrices :

$$\bigoplus_{k=1}^{K} = M_1 \bigoplus \ldots \bigoplus M_K = M$$

L'algèbre tensorielle peut être exploitée pour définir une forme tensorielle d'un générateur d'un processus constitués d'un ensemble de processus synchronisés. Le produit tensoriel permet de calculer le générateur du processus global concernant uniquement les événements de synchronisation (mettant en jeu deux ou plusieurs sous-processus). La somme permet le calcul du générateur correspondant aux événements locaux (mettant en jeu un seul processus). Le théorème suivant donne l'expression du générateur d'une chaîne de MARKOV à temps continu correspondant à K processus synchronisés.

Théorème 4.2 (Générateur d'une CTMC à K composantes)**.** *Soit $X = (X_1, \ldots, X_K)$ une chaîne de Markov à temps continu et $\mathscr{T}_s$ l'ensemble de ses transitions de synchronisation. Le générateur de X est*

$$Q = \bigoplus_{k=1}^{K} Q'_k + \sum_{\tau \in \mathscr{T}_s} \lambda(\tau) \left[\bigotimes_{k=1}^{K} C_k - \bigotimes_{k=1}^{K} A_k \right] \tag{4.1}$$

où Q'_k est la restriction de Q aux transitions locales de S_k et, si $\tau \in \mathscr{T}_s$ est EXPO$(\lambda(\tau))$ et $i \xrightarrow{\tau} j$:

$$\begin{array}{rcccl} C_k & = A_k & = I_{r_k} & si & \tau(i_k) = i_k \\ C_k = 1_{r_k}(i_k, j_k) & et & A_k = 1_{r_k}(i_k, i_k) & si & \tau(i_k) = j_k \neq i_k \end{array}$$

$1_n(j, j')$ étant la matrice de $\mathscr{M}_n$ dont tous les termes sont nuls sauf celui d'indice (j, j') qui vaut 1.

L'approche de l'algèbre tensorielle a été appliquée sur les SPN, GSPN et SWNs. Elle est séduisante dans le sens où elle permet d'exprimer le générateur infinitésimal d'un SPN en termes de Q_i matrices issues de composants ou sous-réseaux de plus petits espaces d'états. On peut ainsi implanter la résolution du système d'équations linéaire $\pi.Q = 0$ sans calculer et stocker Q [44; 74; 75; 46; 114]. Tous les travaux effectués dans ce sens et cités ci-dessus sont inspirés du travail pionnier de Plateau [181] sur les *réseaux d'automates stochastiques*. Comme la méthode tensorielle est basée sur les composants du réseau global, et donc sur la structure du réseau, la méthode est dite *méthode structurée*.

Cet avantage est mis en évidence lors de l'analyse d'une décomposition d'un réseau global, ce qui est explicité dans ce qui suit.

4.4.2 Décomposition synchrone des SWN

La décomposition synchrone [97] a été d'abord étudiée par [75] sur des réseaux de Petri stochastiques généralisés. Puis, [97] ont étendu la méthode au cas des réseaux stochastiques bien formés, en considérant les couleurs des entités en synchronisation.

La décomposition ou composition synchrone est définie sur un système composé d'un ensemble de sous-réseaux ayant des transitions communes (appelées *transitions de synchronisation*) et des ensembles disjoints de places. Les transitions de synchronisation doivent être définies de la même manière dans chaque sous réseau (elles doivent travailler sur les mêmes variables). Les fonctions d'incidence restent les mêmes et les classes de couleur peuvent être différentes suivant le sous réseau (mais on peut toujours associer à chaque sous réseau l'union de celles-ci).

4.4.2.1 Définition de la décomposition synchrone

Une décomposition synchrone fait communiquer exactement deux sous-réseaux. Elle est formellement définie comme suit :

Définition 4.17 (Composition synchrone de SWN)**.** La composition synchrone des SWN $(N_k = (P_k, T_k, \mathscr{C}, J_k, Pre_k, Post_k, guard_k, pri_k, M_{0,k}, \theta_k))_{k\in K}$ est le SWN $N = (P, T, \mathscr{C}, J, Pre, Post, guard, pri, M_0, \theta)$ avec :

1. $\mathscr{C} = \{C_i, i \in I\}$ (l'ensemble (identique) des classes de couleurs de base de chaque N_k);
2. $P = \biguplus_{k\in K} P_k$ ($(P_k)_{k\in K}$ est une partition de l'ensemble des places).
   $M_0(p) = M_{0,k}(p)$ si $p \in P_k$ (le marquage initial),
3. $T = \bigcup_{k\in K} T_k$ avec, $\forall t \in T_k \bigcap T_{k'}$, $J_k(t) = J_{k'}(t)$, $pri_k(t) = pri_{k'}(t)$, $guard_k(t) = guard_{k'}(t)$ et $\theta_k(t) = \theta_{k'}(t)$ (l'ensemble des transitions);
   (a) $J(t) = J_k(t)$ si $t \in T_k$ et $J(p) = J_k(p)$ si $p \in P_k$ (J définit le domaine de couleurs $C(x)$ du nœud x: si $J(x) = (e_i)_{i\in I} \in \mathrm{Bag}(I)$, alors $C(x) = \prod_{i\in I} C_i^{e_i}$);
   (b) $guard(t) = guard_k(t)$ si $t \in T_k$ (la fonction de garde);
   (c) $pri(t) = pri_k(t)$ si $t \in N_k$ (la fonction de priorité);
   (d) $\theta(t) = \theta_k(t)$ si $t \in T_k$ (la vitesse ou la fonction de poids des transitions);
4. $Pre(p,t) = Pre_k(p,t)$ et $Post(p,t) = Post_k(p,t)$ si $p \in P_k$ (les fonctions d'incidence).

D'une manière générale, l'approche d'analyse d'une décomposition synchrone suit le principe donné ci-après :

1. Décomposer le modèle SWN global en un ensemble de sous-réseaux communicants.
2. Étendre chaque sous-réseau N_k obtenu en l'augmentant de parties agrégeant la synchronisation avec l'autre sous-réseau. Les nouveaux réseaux obtenus sont dits *étendus* et sont notés $\overline{N_k}$.
3. Générer le graphe symbolique d'accessibilité $\overline{SRG_k}$ pour chaque sous-réseau étendu $\overline{N_k}$.
4. Déduire une expression tensorielle du générateur de la CTMC sous-jacente au SWN initial en fonction des matrices génératrices associées aux $\overline{SRG_k}$.

Notons que la méthode a été définie pour des SWNs dont les transitions sont exponentielles de sorte que le processus stochastique généré soit une CTMC. Aussi, il n'y a pas d'arc inhibiteur et $\theta(t)$ est indépendant du marquage pour toute transition.

Nous explicitons ces étapes dans ce qui suit. Durant ces explications, nous nous appuyons sur un exemple d'un système de client/serveur tiré de [65], donné ci-après.

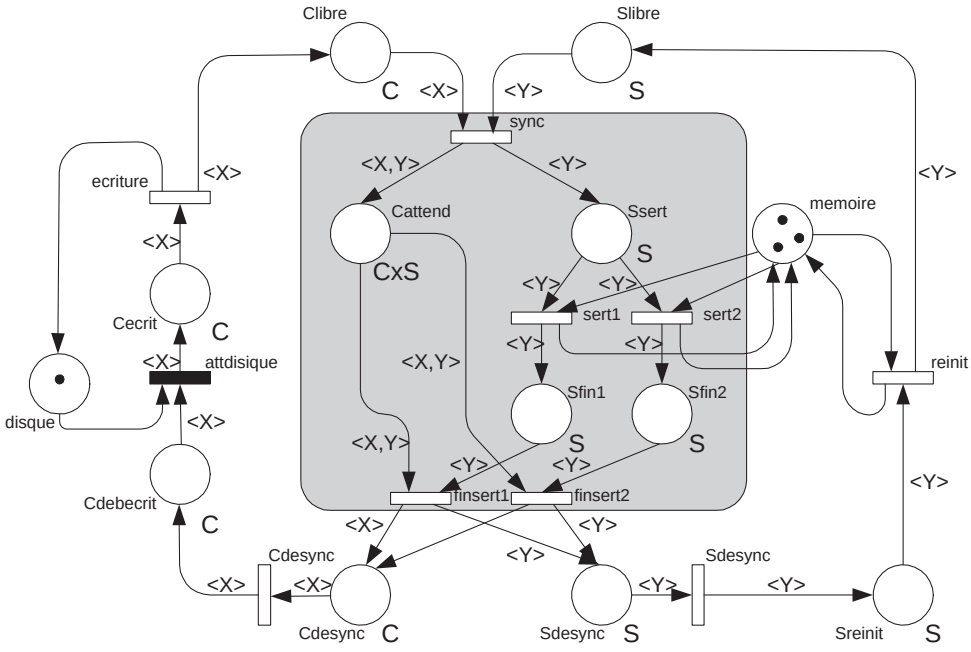


Figure 4.5: Un système client serveur

Exemple 4.26. *Le modèle SWN de la figure 4.5 représente un système client/serveur où un client invoquant une requête se synchronise avec un serveur et reste bloqué jusqu'à la fin du service. Le SWN utilise deux classes de couleurs de base : la première C contient un ensemble de processus clients, la deuxième S un ensemble de processus serveurs. Les processus clients inoccupés sont contenus dans la place Clibre, les serveurs dans Slibre. Le début de synchronisation entre un client x et un serveur y est modélisé par le franchissement de la transition sync plaçant un couple client-serveur* (x,y) *dans la place* $Cattend$.
Le travail rendu par le serveur est représenté via les transitions $sert1$ *et* $sert2$ *qui sélectionnent une ressource mémoire libre des processus serveurs provenant de la place memoire. Deux choix s'offrent aux processus serveurs : utiliser une ou deux ressources. Dans le second cas, le temps moyen de franchissement de la transition est inférieur. Les deux entités se désynchronisent sur les transitions* $finsert1$ *ou* $finsert2$ *selon que le serveur a utilisé une ou deux ressources. Le client est placé dans Cdesync, le serveur dans Sdesync. Pour être à nouveau opérationnel, un serveur doit être réinitialisé. Cette étape utilise une ressource mémoire. Un client doit sauvegarder les résultats obtenus par le serveur sur un disque partagé par tous les clients.*

4.4.2.2 Décomposition

La décomposition d'un SWN en sous-réseaux nécessite d'identifier les transitions déclenchant la synchronisation des sous-réseaux. Deux types de synchronisations ont été distinguées selon qu'il y ait échange de couleurs ou non entre les sous-réseaux :

- La décomposition synchrone *anonyme* correspond à une communication de type "rendez-vous" entre des sous-réseaux n'ayant pas de comportement commun. Dans ce cas, il n'y a pas de synchronisation entre les tirs successifs des transitions de synchronisation, signifiant qu'il n' y a pas

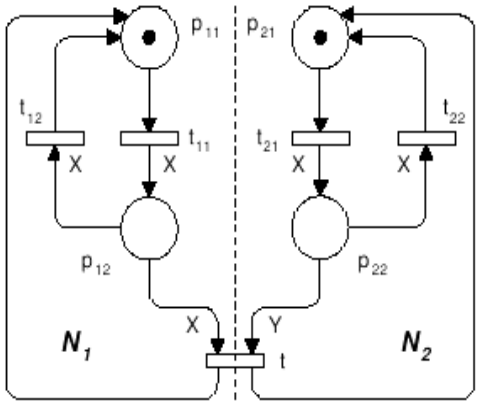


Figure 4.6: Synchronisation anonyme

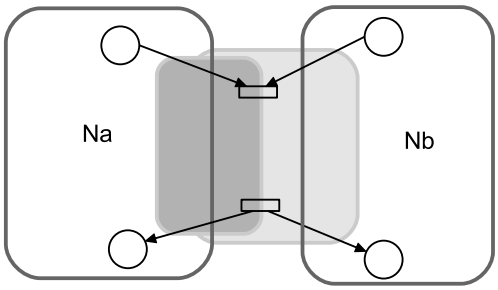


Figure 4.7: Décomposition synchrone à deux phases (N_a: client, N_b: server)

d'échange de couleurs entre les sous réseaux. Un exemple d'une telle synchronisation est donnée par la figure 4.6. Cette simple composition entraîne une analyse isolée des deux sous-réseaux, suivie d'un produit synchronisé des chaînes de Markov sous-jacentes, calculé à travers des franchissements symboliques des transitions de synchronisation. Pour calculer des indices de performances, une expression tensorielle du générateur de la CTMC sous-jacente au SWN global est utilisé.

- La décomposition synchrone à *deux phases* est plus complexe. Elle se présente lorsqu'au moins un des deux sous-réseaux est caractérisé d'un comportement basé sur l'utilisation de classes de couleurs de base, tirées des deux sous-réseaux (voir figure 4.7). Elle consiste à identifier une *zone de synchronisation* entre les deux sous-réseaux. Cette zone doit satisfaire quatre contraintes :
  – Mettre en jeux une paire de sous-systémes liés par une relation asymétrique de type client/serveur
  – Un objet (couleur) déjà synchronisé ne doit pas être resynchronisé.
  – Suivre un schéma de synchronisation en deux étapes (d'où le nom à deux phases) : synchronisation de deux partenaires appelés *client* et *serveur*, engagée par un ensemble de *transitions d'entrée*, travail en commun puis désynchronisation via des *transitions de continuation et sortie*.

  La *zone de synchronisation* est délimitée par les transitions d'entrée et de sortie. Le principe de la décomposition consiste à laisser le pilotage de la zone de synchronisation au sous-réseau serveur et de prendre en compte les problèmes de mémoire de couleurs synchronisées.

Exemple 4.27 (Décomposition du modèle SWN client/serveur). *On peut observer dans le modèle SWN de la figure 4.5 une relation asymétrique entre les clients et les serveurs. Un client ne peut quitter la synchronisation sans son serveur associé et réciproquement, ce qui satisfait la condition de non synchronisation d'une couleur déjà synchronisée. La zone de synchronisation est bien formée de deux phases : une synchronisation client/serveur sur sync, puis une continuation formée par le travail des serveurs et l'attente des clients terminée par la désynchronisation ($finser1$ et $finser2$).*
En conséquence, la décomposition s'effectue sur la zone de synchronisation délimitée par la transition de synchronisation sync et les transitions de désynchronisation $finser1$ et $finser2$. On obtient deux sous-réseaux N_1 et N_2, respectivement le client et le serveur. Chaque sous-réseau est constitué des places et transitions ne mettant en jeux que des comportements locaux (en dehors de toute synchronisation).

Dans la suite, nous nous intéresserons uniquement à ce deuxième type de décomposition, étant donné qu'il est plus général et que le premier type est plus simple à résoudre. Nous le définissons formellement, après avoir donné quelques notations.

Définition 4.18. Soit t une transition de $\mathscr{N}$:

- $Var(t) = \{X \mid X$ apparaît dans une $Pre(p,t)$ ou une $Post(p,t)\}$ (variables d'entrée ou sortie de t);
- $Range(t,X) = \{k \in K \mid \exists p \in P_k$ telle que X apparaît dans $Pre(p,t)$ ou dans $Post(p,t)\}$ (indices des sous-réseaux qui ont des places d'entrée ou sortie pour X et t).

Notations

- $TS = \bigcup_{k \neq k'} T_k \bigcap T_{k'}$ est l'ensemble des transitions de synchronisation;
- $M = (M_k)_{k \in K}$ est un marquage de N avec M_k sa restriction à N_k;
- RS est l'ensemble d'accessibilité de N;
- S est le groupe des permutations admissibles de N;
- EAS_k (resp. GSA_k) est l'ensemble symbolique d'accessibilité (resp. le graphe symbolique d'accessibilité) de N_k.

Nous pouvons maintenant définir la (dé)composition synchrone à deux phases. Dans toutes les définitions citées ci-après, k désigne l'indice d'un sous-réseau client et k' celui d'un serveur.

Définition 4.19 (Composition synchrone à deux phases de SWNs)**.** N est une composition synchrone à deux phases des SWNs $(N_k)_{k \in K}$ ssi :

1. N est la composition synchrone des $(N_k)_{k \in K}$;
2. $\forall k \in K$, il existe $(P_{k,k'})_{k' \in K}$ tels que $P_k = \biguplus_{k' \in \{1 \ldots K\}} P_{k,k'}$;
3. $\forall k \in \{1 \ldots K\}$, il existe $(T_{k,k'})_{k' \in \{1 \ldots K\}}$, $T^-_{k,k'}$ et $T^+_{k,k'}$ tels que $T_k = \biguplus_{k' \in \{1 \ldots K\}} T_{k,k'}$ et $T_{k,k'} = T^-_{k,k'} \biguplus T^+_{k,k'}$, avec les propriétés suivantes :
4. $\forall k,k',l,l' \in \{1 \ldots K\}$ si $T_{k,k'} \bigcap T_{l,l'} \neq \emptyset$, alors $(k,k') = (l,l')$, ou $k \neq k'$ et $k' = l = l'$, ou $l \neq l'$ et $l' = k = k'$;
5. $\forall k,k' \in K \; k \neq k' \left(T^\bullet_{k,k'} \bigcup {}^\bullet T_{k,k'}\right) \bigcap P_k \subseteq P_{k,k'} \biguplus P_{k,k}$;
6. $\forall k \in \{1 \ldots K\}$, $\left(T^\bullet_{k,k} \bigcup {}^\bullet T_{k,k}\right) \bigcap P_k \subseteq P_{k,k}$.

Les ensembles introduits dans cette définition sont explicités comme suit :

- $P_{k,k'}$ est l'ensemble des places d'entrée et de sortie de $\mathscr{N}$ qui sont impliquées uniquement dans la synchronisation client $\mathscr{N}_k$ - serveur $\mathscr{N}_{k'}$.

- Nous avons différents ensembles de transitions :
  - $T^{-}_{k,k'}$ est l'ensemble des transitions de début de synchronisation;
  - $T^{+}_{k,k'}$ est l'ensemble des transitions de synchronisation de continuation et fin;
  - $T_{k,k} \setminus \biguplus_{k' \neq k} T_{k',k}$ est l'ensemble des transitions de $\mathscr{N}_k$ qui modèlisent des activités locales dans $\mathscr{N}_k$, non impliquées dans la synchronisation avec d'autres sous-réseaux.

Toute transition présente dans l'ensemble $T^{-}_{k,k'}$ est une transition *d'entrée de synchronisation*, modélisant la synchronisation d'un client avec un serveur. Les transitions présentes dans $T^{+}_{k,k'}$ sont des transitions de *continuation* ou de *sortie de synchronisation*. Toute autre transition $t \notin T^{-}_{k,k'} \bigcup T^{+}_{k,k'}$ est dite locale et n'est présente que dans un seul sous-réseau $\mathscr{N}_k$.

Exemple 4.28. *Soient les deux sous-réseaux de l'exemple précédent. Le sous-réseau client $\mathscr{N}_1$ est composé de l'ensemble des places $P_1 = \{Clibre, Cecrit, disque, Cdebecrit, Cdesync\}$ et de l'ensemble de transitions $T_1 = \{ecriture, sync, finsert1, finser2, Cdesync, attdisque\}$, ainsi que l'ensemble des arcs reliant ces places et transitions. Le sous-réseau serveur $\mathscr{N}_2$ est composé de l'ensemble des places $P_2 = \{Slibre, memoire, Sdesync, Sreinit\}$ et de l'ensemble de transitions $T_2 = \{$ sync , sert1 , sert2 , finsert1 , finsert2 , Sdesync , reinit $\}$. D'après la définition des ensembles de transitions caractérisant les deux sous-réseaux on en déduit que $TS = \{sync, finsert1, finsert2\}$. La seule transition d'entrée de synchronisation est sync donc $T^{-}_{1,2} = \{sync\}$ et $T^{+}_{1,2} = \{finsert1, finsert2\}$. L'ensemble des places contenues dans la synchronisation $P_{1,2}$ est $\{Cattend\}$; elle est la seule place de la synchronisation à mettre en jeu des jetons client-serveur.*

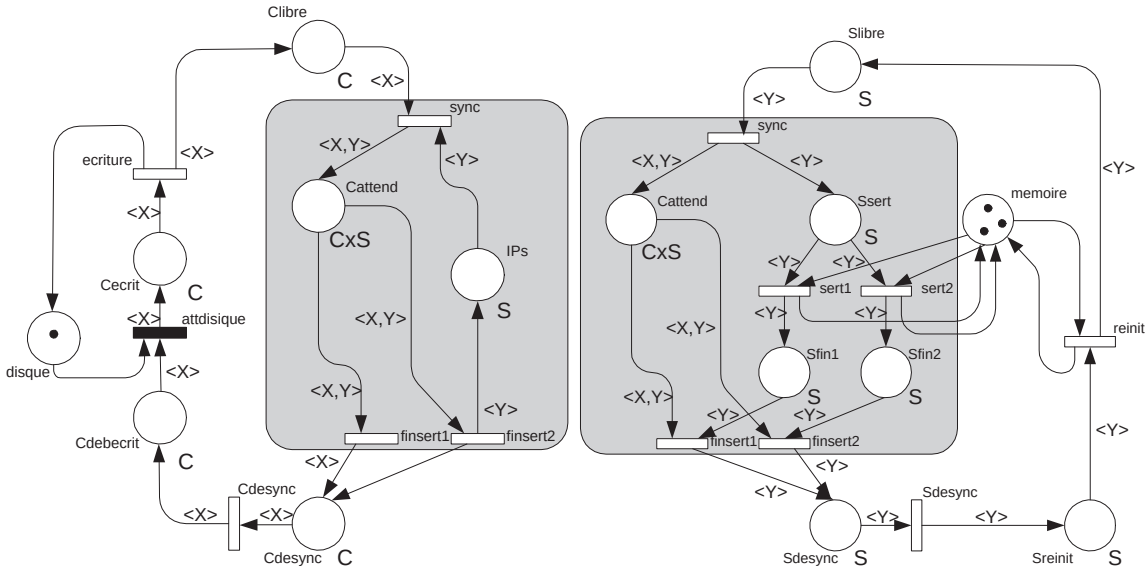
4.4.2.3 Extension des sous-réseaux

Le mécanisme d'extension des sous-réseaux permet de mémoriser les couleurs engagées dans la synchronisation, afin d'établir une correspondance entre les tuples de marquages des sous-réseaux et les marquages associés au réseau SWN original. Il s'opère différemment pour les sous-réseaux client et serveur.

- Concernant le sous-réseau client, il est simplement étendu avec des places implicites contenant les classes de couleur de base serveur impliquées dans la synchronisation. Ceci permet de limiter le franchissement de la transition d'entrée aux serveurs non synchronisés.
- Le sous-réseau serveur est étendu avec la zone de synchronisation client (en boite gris foncé dans la figure 4.7).

Définition 4.20 (Extension d'un sous-réseau)**.** Soit $\mathscr{N}$ composition synchrone à deux phases des $(\mathscr{N}_k)_{k \in \{1...K\}}$. l'extension de $\mathscr{N}_k$ est le réseau bien formé $\overline{\mathscr{N}}_k = \mathscr{N}_k \bigcup (\bigcup_{k' \neq k} \mathscr{N}_{k',k})$, où $\mathscr{N}_{k',k}$ est le sous-réseau de $\mathscr{N}_{k'}$ restreint aux places de $P_{k',k}$, aux transitions de $T_{k',k}$ et aux arcs entre ces nœuds. Nous notons $\overline{P}_k$ l'ensemble des places de $\overline{\mathscr{N}}_k$.

Exemple 4.29. *Reprenons l'exemple de la figure 4.5. Une fois la décomposition réalisée, on procède à l'extension des sous-réseaux. La figure 4.8 donne les deux sous-réseaux étendus $\overline{N_1}$ et $\overline{N_2}$. Le sous-réseau client est formé par son comportement en isolation, auquel on ajoute une restriction de la zone de synchronisation à la transition d'entrée sync, les transitions de sortie finser1 et finser2 et les places mettant en jeu des couples client/serveur, ici la seule place Cattend. On ajoute une place implicite PIs dont le marquage initial contient toutes les couleurs serveurs pouvant se synchroniser. Cette place permet de limiter le franchissement de la transition d'entrée aux serveurs non synchronisés. Pour le sous-réseau serveur, on ajoute à son comportement en isolation la zone de synchronisation complète avec le client.*



(a) sous-réseau client $\overline{N_1}$ (b) sous-réseau serveur $\overline{N_2}$

Figure 4.8: Sous-réseaux étendus du modèle de la figure 4.5

4.4.2.4 *Conditions syntaxiques d'agrégation*

Des conditions structurelles ont été énoncées sur les sous-réseaux en composition synchrone à deux phases, afin de rendre possible la génération d'une chaîne de Markov agrégée à partir de la structure des SWNs.

D'une manière informelle, ces conditions peuvent se résumer aux points suivants :

- En dehors du sous-réseau délimité par ces transitions (en boîte grise dans la figure 4.7), il ne faut pas avoir de synchronisation entre les classes de couleurs de base des deux sous-réseaux.
- Les rôles des sous-réseaux sont non symétriques : le sous-réseau *client* n'interagit pas (dans une "façon colorée") avec le reste de son réseau durant la synchronisation. Le sous-réseau *serveur* peut éventuellement interagir avec le reste de son réseau durant la synchronisation.

Définition 4.21 (Conditions d'agrégation pour la composition synchrone)**.** N remplit les conditions syntaxiques d'agrégation ssi : $\forall k,k' \in K,\ k \neq k'$, nous avons les propriétés suivantes :

1. Si une classe de couleurs C_i est dans le domaine de couleurs de $N_{k,k'}$ et $\overline{N}_k \backslash N_{k,k'}$:
   pour chaque $p \in \overline{P}_k \backslash P_{k,k'}$ avec $C(p) = \prod_{j\in I} C_j^{e_j}$, et chaque $p' \in P_{k,k'}$ avec $C(p') = \prod_{j\in I} C_j^{e'_j}$:
   $\forall 1 \leq j \leq e_i,\ 1 \leq j' \leq e'_i$, il existe un semi-flot dans $\overline{N}_k$, $f = (f_q)_{q\in\overline{P}_k}$ sur C_i tel que $\forall M,\ f(M) = S_i$, avec :
   $$f_q = \begin{cases} 0 \text{ ou une projection} & \text{si} \quad q \neq p, q \neq p' \\ j\text{i ème projection} & \text{si} \quad q = p \\ j'\text{i ème projection} & \text{si} \quad q = p' \end{cases}$$
2. Si une classe de couleurs C_i est dans le domaine de couleurs de $N_{k,k'}$ mais pas dans celui de $\overline{N}_k \backslash N_{k,k'}$, alors C_i est dans le domaine de couleurs d'au moins deux places et l'une d'elles, notée $p_{i,k,k'}$, vérifie $C(p_{i,k,k'}) = C_i$; de plus, pour chaque $p \in P_{k,k'}$, $p \neq p_{i,k,k'}$, avec $C(p) = \prod_{j\in I} C_j^{e_j}$:
   $\forall 1 \leq j \leq e_i$ il existe un semi-flot (f_q) dans $N_{k,k'}$ sur C_i tel que $\forall M\ f(M) = S_i$ avec :

$$f_q = \begin{cases} 0 \text{ ou une projection} & \text{si} \quad q \neq p, q \neq p_{i,k,k'} \\ j\text{i ème projection} & \text{si} \quad q = p \\ \text{Identité (sur } C_i) & \text{si} \quad q = p_{i,k,k'} \end{cases}$$

3. $\forall t \in T^+_{k,k'}$, $\forall X \in Var(t)$
   (a) $\exists p \in P_{k,k'}$, $p \neq p_{i,k,k'}$, telle que $Pre(p,t)$ contient un terme positif $\langle \ldots, X, \ldots \rangle$;
   (b) $\forall p \in P_k \backslash P_{k,k'}$, X n'apparaît pas dans $Pre(p,t)$.
4. $\forall t \in T^-_{k,k'}$ $Var(t) = Var_k(t) \uplus Var_{k'}(t)$ (donc $C(t) = C^k(t) \times C^{k'}(t)$) avec :
   (a) $\forall X \in Var_k(t)$, $\forall p \in P_{k'}$, X n'est ni dans $Pre(p,t)$ ni dans $Post(p,t)$;
   (b) $\forall X \in Var_{k'}(t)$ de domaine C_i et avec une $p_{i,k,k'}$, $Pre(p_{i,k,k'},t) = X$ et, $\forall p \in P_k,\ p \neq p_{i,k,k'}$, X n'est pas dans $Pre(p,t)$;
   (c) $\forall X \in Var_{k'}(t)$, de domaine C_i sans $p_{i,k,k'}$, $\forall p \in P_k$, X n'apparaît pas dans $Pre(p,t)$ ni dans $Post(p,t)$;
   (d) Soit $C^{k'}(t) = \prod_{i \in I} C_i^{e_i}$. Si $e_i > 0$, alors C_i n'est ni dans le domaine de couleurs d'une place de $\overline{P}_k \backslash P_{k,k'}$, ni dans $C^k(t)$.

La première condition statue sur les couleurs clientes synchronisées. Cette condition permet de reconnaître les couleurs clientes sychronisées de celles présentes dans le comportement local, cela signifie qu'une entité cliente ne peut être trouvée à la fois en local et en synchronisation. Dans notre exemple, le flot sur la couleur synchronisée cliente C est $f_c = X_c.Clibre + X_c.Cattend + X_c.Cdesync + X_c.Cdebecrit + X_c.Cecrit$. La deuxième condition est identique pour les couleurs serveurs synchronisées obligeant le modélisateur à ajouter au sous-réseau client et par couleur serveur synchronisée dans ce sous-réseau une place implicite entre les transitions de sortie et d'entrée. Ce jeux de places permet de limiter les tirs de transitions d'entrée aux serveurs non synchronisés. Dans notre exemple, la seule couleur serveur synchronisée est S, on ajoute alors la place implicite IPs dans la zone de synchronisation du sous-réseau client. Le flot sur la couleur serveur synchronisée S est $f_s = X_s.Cattend + X_s.IPs$. La condition 3 limite les interactions entre activités locales et en synchronisation qui ne peuvent exister que pour des activités locales non colorées. La dernière condition donne le principe d'engagement d'une synchronisation client-serveur. Les clients se synchronisant ne sont pas connus par le serveur. Les serveurs qui s'engagent sont présents dans la place d'abstraction. Des couleurs provenant du sous-réseau serveur peuvent conditionner l'engagement sans participer à la synchronisation.

4.4.2.5 Écriture tensorielle du générateur de la chaîne de Markov

Le générateur de la chaîne de MARKOV à temps continu agrégée est donné par sa forme tensorielle.

$$Q' = \bigoplus_{k=1}^{K} Q'_k + \sum_{t \in TS} \sum_{d} \theta(t,d) \left[\bigotimes_{k=1}^{K} C_k(t,d) - \bigotimes_{k=1}^{K} A_k(t,d) \right] \tag{4.2}$$

Avec les notations suivantes :

- $d = (d_i^j)_n^{e_i(t)}$ (avec $1 \leq d_i^j \leq n_i$ et n_i le nombre de sous-classes statiques de C_i) est un choix de sous-classes statiques relatif à $C(t)$,
- $\forall\ \langle \lambda, \mu \rangle$ (fonctions d'instanciation pour la restriction t_k de t à $\overline{\mathscr{N}}_k$), $\overline{\mathscr{M}}_k$ et $\overline{\mathscr{M}'}_k$ dans $\overline{SRS}_k$:

– $(d(Z_i^{\lambda_i(j)}))_n^{e_i(t)}$ sont les sous-classes statiques des sous-classes dynamiques (de la représentation canonique de $\overline{\mathscr{M}}_k$) de $\langle \lambda, \mu \rangle$, $d^k = (d_i^j)_n^{e_i(t_k)}$ est la restriction de d relative à $C(t_k)$ et

$$1_{(t,d,\langle \lambda,\mu\rangle,\overline{\mathscr{M}}_k,\overline{\mathscr{M}}'_k)} = \begin{cases} 1 & \text{si} \quad d^k = (d(Z_i^{\lambda_i(j)}))_n^{e_i(t_k)} \\ & \text{et} \quad \overline{\mathscr{M}}_k[t_k(\langle \lambda,\mu\rangle)\rangle\overline{\mathscr{M}}'_k \\ 0 & \text{sinon} \end{cases}$$

$$1_{(t,d,\overline{\mathscr{M}}_k,\overline{\mathscr{M}}'_k)} = \bigvee_{\langle \lambda,\mu\rangle} 1_{(t,d,\langle \lambda,\mu\rangle,\overline{\mathscr{M}}_k,\overline{\mathscr{M}}'_k)}$$

avec $\bigvee$ l'addition booléenne (ou logique),

– pour $t \in T^-_{k,k'}$ (nous supposons pour alléger l'écriture, et sans perte de généralité que $C^k(t) = \prod_{i=1}^{n(t)} C_i^{e_i(t)}$ et $C^{k'}(t) = \prod_{i=n(t)+1}^{n} C_i^{e_i(t)}$), nous notons $\langle \lambda, \mu \rangle^k$ la restriction de $\langle \lambda, \mu \rangle$ à $C^k(t)$, $\langle \gamma^k, \delta^k \rangle$ des fonctions d'instanciation pour $C^k(t)$ et

$$1_{(t,d,\langle \gamma^k,\delta^k\rangle,\overline{\mathscr{M}}_k,\overline{\mathscr{M}}'_k)} = \bigvee_{\substack{\langle \lambda,\mu\rangle \\ \langle \lambda,\mu\rangle^k = \langle \gamma^k,\delta^k\rangle}} 1_{(t,d,\langle \lambda,\mu\rangle,\overline{\mathscr{M}}_k,\overline{\mathscr{M}}'_k)}$$

$$F^-_{(\langle \gamma^k,\delta^k\rangle,\overline{\mathscr{M}}_k,\overline{\mathscr{M}}'_k)} = \prod_{i=1}^{h(t)} \prod_{j=1}^{m_i} \frac{\mathrm{card}(Z_i^j)!}{(\mathrm{card}(Z_i^j) - \delta_i^{k,j})!}$$

$$F'^-_{(\langle \gamma^{k'},\delta^{k'}\rangle,\overline{\mathscr{M}}_{k'},\overline{\mathscr{M}}'_{k'})} = \prod_{i=n(t)+1}^{h'(t)} \prod_{j=1}^{m_i} \frac{\mathrm{card}(Z_i^j)!}{(\mathrm{card}(Z_i^j) - \delta_i^{k',j})!}$$

avec $h(t)$ (resp. $h'(t)$) le plus grand indice de classe non ordonnée de $C^k(t)$ (resp. $C^{k'}(t)$), m_i le nombre de sous-classes dynamiques de C_i dans la représentation canonique de $\overline{\mathscr{M}}_k$ ou $\overline{\mathscr{M}}'_k$, et $\delta_i^{k,j} = \sup\{\delta_i^k(x) \mid \gamma_i^k(x) = j\}$ le nombre d'instanciations différentes dans la sous-classe dynamique Z_i^j pour $\langle \gamma^k, \delta^k \rangle$ ($\delta_i^{k,j} = 0$ si Z_i^j n'est pas instanciée).

– pour $t \in T^+_{k,k'}$

$$F'^+_{(\langle \lambda,\mu\rangle,\overline{\mathscr{M}}_{k'},\overline{\mathscr{M}}'_{k'})} = \prod_{i=1}^{h} \prod_{j=1}^{m_i} \frac{\mathrm{card}(Z_i^j)!}{(\mathrm{card}(Z_i^j) - \mu_i^j)!}$$

avec h le plus grand indice de classe non ordonnée de $C(t)$ et m_i , μ_i^j comme au-dessus.

La somme tensorielle des générateurs Q'_k comptabilise les franchissements des transitions locales des K sous-réseaux. Une matrice $C_k(t,d)$ prend en compte les tirs d'une transition t de synchronisation du sous-réseau k pour une instance statique d donnée. Cette matrice est l'identité si $t \notin TS_k$. Une matrice A_k est associée à chaque C_k permettant de calculer la diagonale du générateur.

4.4.3 Décomposition asynchrone des SWN

La décomposition asynchrone [98] des SWNs modélise des sous-systèmes communicant par envoi de messages ou entités (tâches, ressources,...). Initialement, elle a été étudiée par P.Buchholz pour le cas de Réseaux de Petri Stochastiques Généralisés et d'autres types de RDP [44; 45]. Dans ces travaux, il a proposé plusieurs modèles orientés vers une description hiérarchique de la composition.

Plus tard, Moreaux et al. [98] ont proposé une méthode de décomposition asynchrone des SWNs, fondée sur le même principe que celui des réseaux stochastiques généralisés [46; 57].

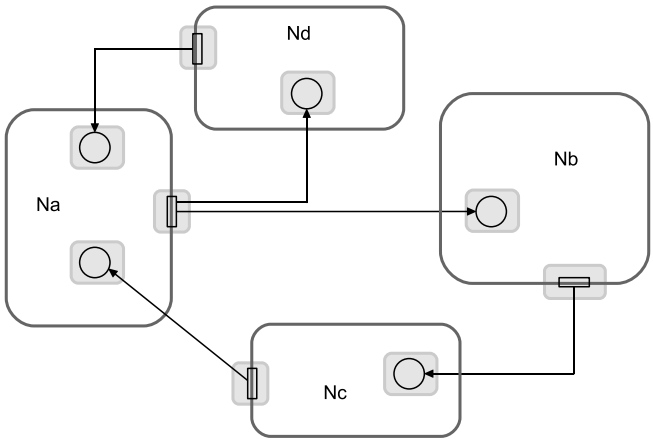


Figure 4.9: (Dé)composition asynchrone: structure générale

4.4.3.1 Définition de la décomposition asynchrone

La décomposition asynchrone lie un ensemble de sous-réseaux par un ensemble de transitions dites *transitions de sortie*. La communication est initiée par un sous-réseau, puis est propagée par le transfert d'entités (couleurs) à travers une chaîne d'autres sous-réseaux séquentiellement, se terminant dans le sous-réseau initial (voir figure 4.9).

Formellement, la (dé)composition asynchrone est définie comme suit.

Définition 4.22 (Composition asynchrone de SWN)**.** Soit $((P_k,T_k))_{k\in K}$ des familles de places et transitions du réseau bien formé $N=(P,T,\ldots)$. N est la composition asynchrone de *ses sous-réseaux*[3] $(N_k=(P_k,T_k,\ldots))_{k\in K}$ ssi :

1. $T=\biguplus_{k\in K}T_k$ (partition des transitions);
2. $P=\biguplus_{k\in K}P_k$ (partition des places);
3. $\forall k\in K$,
$$\forall t\in T_k,\ {}^\bullet t\bigcup t^\bullet\subseteq P_k$$

Notations

- $TO_k=\{t\in T_k\mid t^\bullet\bigcap(P\backslash P_k)\neq\emptyset\}$ est l'ensemble des transitions de sortie de $\mathscr{N}_k$;
- $TO=\bigcup_{k\in K}TO_k$ est l'ensemble de toutes les transitions de sortie;
- $T\backslash TO$ est l'ensemble des transitions locales (aux $\mathscr{N}_k$);
- $M=(M_k)_{k\in K}$ est un marquage de $\mathscr{N}$ avec $\mathbf{m}_k=\mathbf{m}(P_k)$;
- RS est l'ensemble symbolique d'accessibilité de $\mathscr{N}$;
- S est le groupe des permutations admissibles de $\mathscr{N}$;

Les classes de couleurs de base correspondant aux entités traversant un sous-réseau vers un autre sont dites classes *globales*, alors que les autres classes sont appelées *locales*. L'ensemble des sous-réseaux participant à cette décomposition est appelé *chaîne asynchrone*.

De manière similaire à l'analyse de la décomposition synchrone de SWNs, la décomposition asynchrone requiert plusieurs étapes :

[3] restrictions de N aux places de P_k, transitions de T_k et aux arcs qui les joignent

1. Décomposer le modèle SWN global en un ensemble de K sous-réseaux communicant de la manière décrite.
2. Étendre chaque sous-réseau N_k obtenu en l'augmentant de parties agrégeant la synchronisation avec les autres.
3. Générer le graphe symbolique d'accessibilité $\overline{SRG_k}$ pour chaque sous-réseau étendu $\overline{N_k}$.
4. Déduire une expression tensorielle du générateur de la CTMC sous-jacente au SWN initial en fonction des matrices génératrices associées aux $\overline{SRG_k}$.

Nous détaillons ces étapes dans ce qui suit. Ici également, nous nous basons sur l'exemple suivant tiré de [65] pour expliciter la méthode.

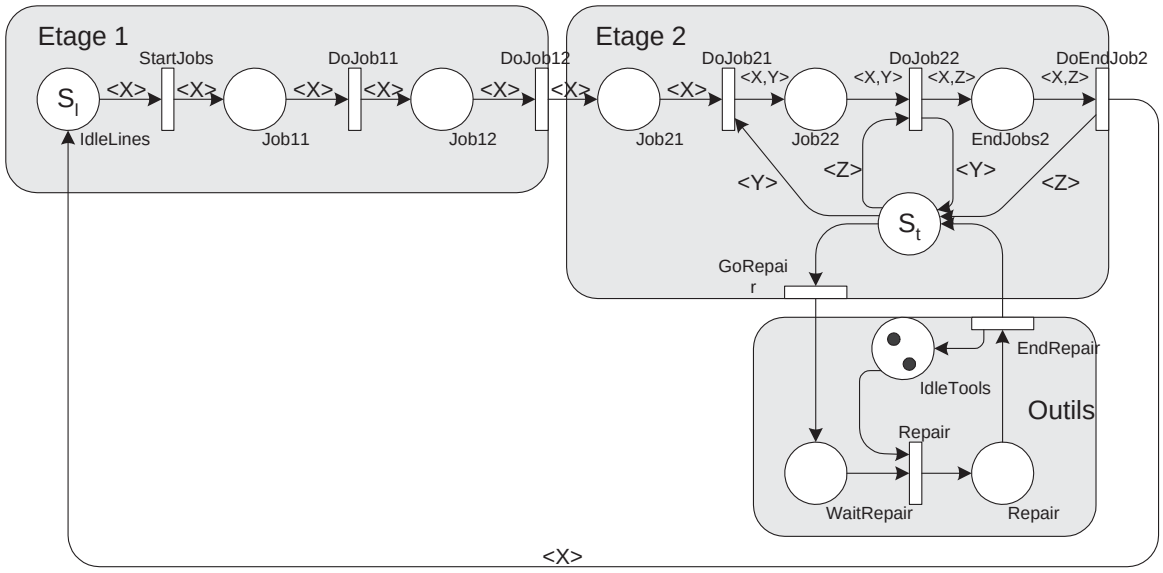


Figure 4.10: Modèle SWN du système manufacturier

Exemple 4.30. *Soit le réseau bien formé de la figure 4.10 correspondant à une chaîne de montages identiques fonctionnant en parallèle. Chaque chaîne assemble le même type de produit et possède deux étages. Le premier étage exécute une seule tâche alors que le deuxième en réalise deux. La deuxième de ces tâches nécessite l'utilisation d'outils partagés par l'ensemble des lignes. Les outils sont régulièrement entretenus par un service de maintenance. Le modèle contient deux classes de couleurs de base, les lignes L et les outils T.*

4.4.3.2 Décomposition

Cela consiste à identifier un ensemble de transitions permettant à certaines entités (couleurs) de migrer de sous-réseau en sous-réseau. Ces transitions délimitent les sous-réseaux communicants. Notons qu'une décomposition d'un SWN n'est pas unique.

Exemple 4.31. *Reprenons l'exemple de la figure 4.10. Nous identifions quatre transitions de sortie :*

- *La transition* DoJob12 *qui envoie des entités à la partie (sous-réseau) du SWN désignant le second étage.*
- *La transition* DoEndJob2 *qui renvoie les entités reçues vers le premier étage.*
- *La transition* GoRepair *qui envoie des entités pour réparation.*
- *La transition* EndRepair *qui renvoie les entités réparées au second étage.*

Nous décomposons donc notre modèle en trois sous-réseaux, le premier étage, le second et la gestion des outils. Les deux couleurs sont globales car l'état d'une ligne est donné par la position de son jeton sur l'ensemble des deux étages et un outil peut quitter le deuxième étage pour être réparé. Les trois sous-modèles étendus sont donnés par la figure 4.11. L'ensemble des transitions de sortie est :

$$TO = \{DoJob12, DoEndJob2, GoRepair, EndRepair\}.$$

4.4.3.3 Extension des sous-réseaux SWN

Les sous-réseaux d'une décomposition asynchrone d'un SWN ne sont pas totalement autonomes : chacun interagit avec son environnement (autres sous-réseaux). De ce fait, pour analyser un sous-réseau en isolation, il est nécessaire de le compléter par une partie modélisant et agrégeant cet environnement. Ceci est réalisé par la définition de *vues abstraites* pour chaque réseau.

Une vue abstraite d'un sous-réseau consiste en une agrégation de son comportement et une abstraction de ses détails par rapport aux couleurs globales vu de l'extérieur. Elle permet la mémorisation des entités globales en synchronisation (en dehors du sous-réseau considéré). Les vues abstraites sont automatiquement dérivées à partir de la structure des sous-réseaux, dès la satisfaction d'un ensemble de conditions détaillées. La construction d'une vue abstraite d'un réseau est basée sur l'étude des semiflots partiels projetés sur les couleurs globales, et plus précisément sur des *semiflots d'abstraction*.

Définition 4.23 (Semi-flot d'abstraction)**.** f est un semi-flot d'abstraction relativement à une classe globale C_i de $\mathcal{N}_k$ ssi f est semi-flot partiel de $\mathcal{N}$ par rapport à $T_k \backslash TO_k$ tel que :

- $C(f) = C_i$;
- $\forall p \in P_k,\ f_p$ est 0 ou b fois une projection (avec b une constante positive).

On peut définir la notion de vue abstraite d'un sous-réseau.

Définition 4.24 (Vue abstraite d'un sous-réseau)**.** Un sous-réseau $\mathcal{N}_k$ admet une vue abstraite ssi il existe un semi-flot d'abstraction relativement à chaque classe globale de $\mathcal{N}_k$.

Soit $F_k = \{f_{C_i} \mid C_i$ classe globale de $\mathcal{N}_k\}$ l'ensemble des semi-flots d'abstraction d'un sous-réseau admettant une vue abstraite.

- La *vue abstraite* de $\mathcal{N}_k$, notée $\mathcal{A}(\mathcal{N}_k)$, est l'ensemble des places $PA_k = \{p_f \mid f \in F_k\}$ avec :
  - $C(p_f) = C(f)$ (le domaine de couleurs de p_f);
  - $M_0(p_f) = \sum_{p \in P_k} f(M_0(p))$ (le marquage initial de p_f).
- Pour tout marquage $M = (M_k)_{k \in K}$ de $\mathcal{N}$, le *marquage abstrait* de M_k est

$$\mathrm{am}(M_k) = \left(\sum_{p \in P_k} f(M(p)) \right)_{f \in F_k}$$

également noté $M(PA_k)$ par extension de M (les valeurs des semi-flots de F_k dans le marquage M_k).

L'extension d'un sous-réseau $\mathcal{N}_k$ est alors construite à partir des vues abstraites des autres sous-réseaux $\mathcal{N}_{k'}$ avec $k' \neq k$.

Définition 4.25 (Extension d'un sous-réseau)**.** Soit N un réseau bien formé composition asynchrone des $(\mathscr{N}_k)_{k\in K}$. L'extension $\overline{\mathscr{N}}_k$ de $\mathscr{N}_k$ est le réseau bien formé $(\overline{P}_k, \overline{T}_k, \mathscr{C}, J, \overline{Pre}_k, \overline{Post}_k, \overline{pri}, \overline{M}_{0,k}, \theta_k)$ avec :

- $\overline{P}_k = P_k \bigcup_{k'\neq k} PA_{k'}$: pour chaque marquage M de $\mathscr{N}$, le marquage correspondant de $\overline{\mathscr{N}}_k$ est $\overline{M}_k = (M_k, (M(PA_{k'}))_{k'\neq k})$
- $\overline{T}_k = T_k \bigcup_{k'\neq k} TO_{k'}$
- $\forall p \in P_k,\ \forall t \in \overline{T}_k,\ \ \overline{Pre}_k(p,t) = Pre_k(p,t)$ et $\overline{Post}_k(p,t) = Post_k(p,t)$
- $\forall p_f \in PA_{k'},\ \forall t \in \overline{T}_k, \overline{Pre}(p_f,t) = \sum_{p'\in {}^\bullet t} f_{p'} \circ Pre(p',t)$ et $\overline{Post}(p_f,t) = \sum_{p'\in t^\bullet} f_{p'} \circ Post(p',t)$

$\overline{M}_k(PA_{k'})$ est encore appelé le *marquage abstrait* de $\mathscr{N}_{k'}$ (dans $\overline{M}_k$).

Après avoir été augmenté des vues abstraites des $\mathscr{N}_{k'}, k \neq k'$, $\overline{\mathscr{N}_k}$ peut être étudié en isolation.

Exemple 4.32. *Étudions l'extension du sous-réseau concernant le deuxième étage de l'exemple précédent. Elle est constituée du sous-réseau $\mathscr{N}_2$ composé des places Job*21*, Job*22*, EndJobs*2 *et IdleTools et des transitions DoJob*21*, DoJob*22*, GoRepair et DoEndJob, et des vues abstraites de $\mathscr{N}_1$ et $\mathscr{N}_3$.*
Pour calculer la vue abstraite de $\mathscr{N}_1$, on calcule le semiflot donné sur la couleur globale L. On obtient le semi-flot suivant :

$$f_L = IdleLines.X_L + Job11.X_l + Job12.X_L$$

*Pour obtenir une vue abstraite, chaque semi-flot est remplacé par une place implicite de domaine de couleur égal au domaine de la couleur globale du semiflot et de marquage initial égal à l'ensemble des entités de la couleur initialement présentes dans le sous-réseau. La vue abstraite de $\mathscr{N}_1$ est donc formée par son unique transition de sortie DoJob*12 *et une place implicite PA*1*L ayant pour domaine de couleur L et pour marquage initial toutes les couleurs de L initialement présentes dans $\mathscr{N}_1$, soit la classe L complète. Cette place relie la transition de sortie DoEndJonb*2 *de $\mathscr{N}_2$ à DoJob*12*.*
Pour obtenir la vue abstraite du sous-réseau $\mathscr{N}_3$, on calcule d'abord le semi-flot correspondant à la couleur globale T. Celui obtenu est donné par

$$f_T = WaitRepair.X_T + Repair.X_T$$

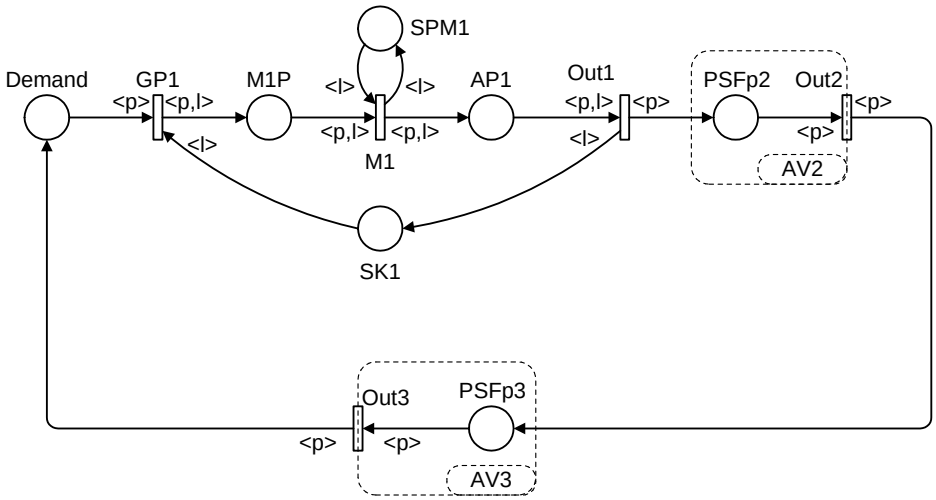
*La vue calculée est alors formée de la transition de sortie EndRepair et d'une place implicite PA*3*T de domaine de couleur T ne contenant aucun jeton car tous les outils sont initialement présents dans $\mathscr{N}_2$. Cette place relie Gorepair et EnRepair.*
Le réseau étendu obtenu $\overline{\mathscr{N}_2}$ est donné par la figure 4.11, ainsi que les réseaux étendus $\overline{\mathscr{N}_1}$ et $\overline{\mathscr{N}_3}$ des deux autres sous-réseaux.

4.4.3.4 Conditions syntaxiques d'agrégation

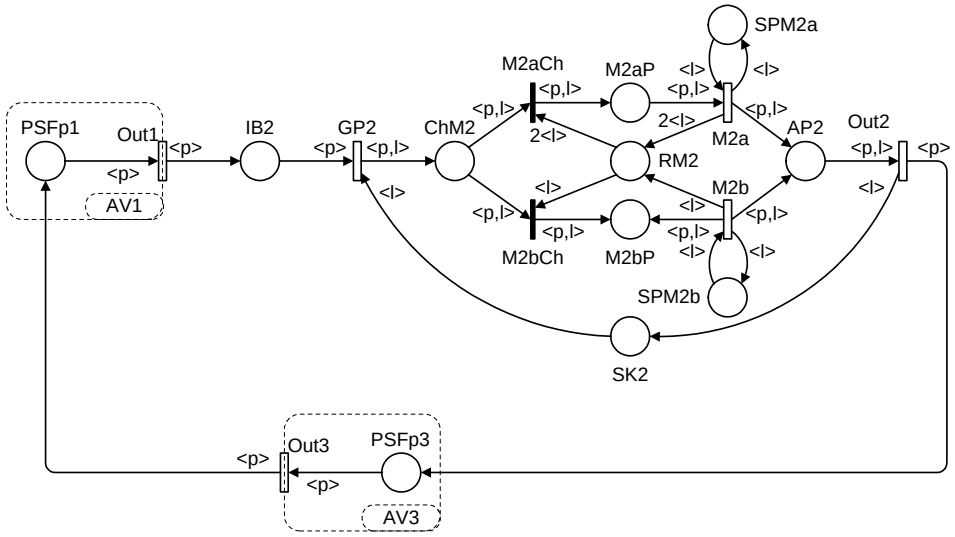
Des conditions ont été imposées afin d'assurer que la chaîne de MARKOV obtenue par expression tensorielle est une agrégation exacte et qu'elle est une sur-matrice du générateur du système global.

Définition 4.26 (Conditions d'agrégation pour la composition asynchrone)**.** $\mathscr{N}$ remplit les conditions syntaxiques d'agrégation ssi $\forall k \in K$ nous avons les propriétés suivantes :

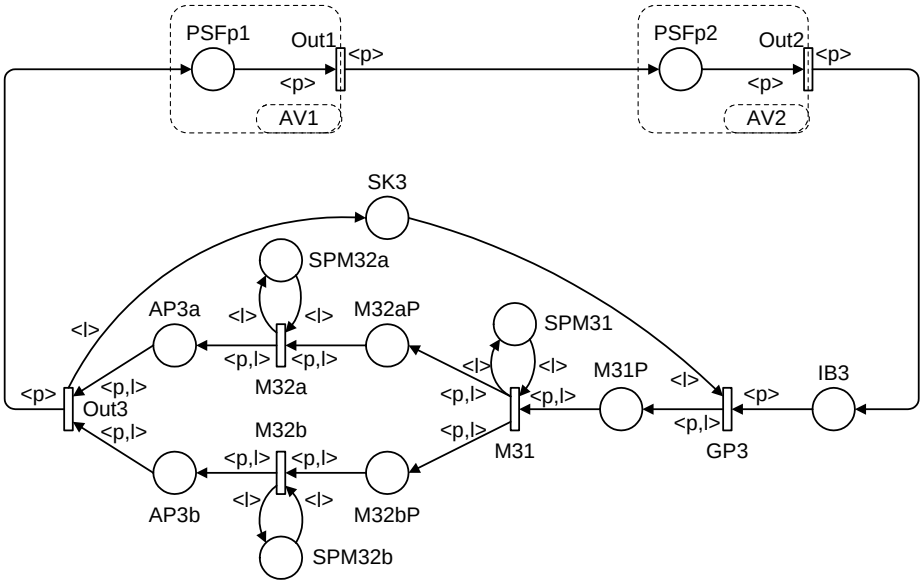
1. $\forall p \in {}^\bullet TO_k$ avec $C(p) = \prod_{i\in I} C_i^{e_i}$, $\forall i \in I$ tel que C_i est une classe globale de $\mathscr{N}_k$ et $e_i > 0$, nous avons : $\forall 1 \leq j \leq e_i$, X_i^j (la $j$ième projection sur C_i) est dans le semi-flot d'abstraction $f_{C_i} \in F_k$.
2. $\forall t \in TO_k,\ \ \forall X \in Var(t)$ correspondant à une classe de couleurs globales de $\mathscr{N}_k$, X est dans un terme positif d'une $Pre(p,t)$.



(a) sous-réseau étendu étage 1 $\overline{\mathcal{N}_1}$



(b) sous-réseau étendu étage 2 $\overline{\mathcal{N}_2}$



(c) sous-réseau étendu outils $\overline{\mathcal{N}_3}$

Figure 4.11: Sous-réseaux étendus du modèle de la figure 4.10

3. $\forall p \in PA_k$ avec $C(p) = C_l$, $\forall k' \neq k$, $\forall p' \in P_{k'}$ avec $C(p') = \prod_{i \in I} C_i^{e'_i}$:
   $\forall\ 1 \leq j' \leq e'_l$, il existe un semi-flot $g = (g_q)_{q \in \overline{P}_{k'}}$ sur C_l dans $\overline{\mathcal{N}}_{k'}$, tel que $\forall M,\ g(\overline{M}_{k'}) = S_l$ avec :

$$g_q = \begin{cases} 0 \text{ ou une projection} & \text{si} \quad q \neq p, q \neq p' \\ \text{Identité} & \text{si} \quad q = p \\ j\text{ième projection} & \text{si} \quad q = p' \end{cases}$$

La première condition spécifie que les entités globales doivent garder leur identité lors d'un transfert vers un autre sous-réseau. La deuxième annonce le fait qu'il ne peut y avoir ni création ni destruction d'entité, mais uniquement des transferts d'entités entre sous-réseaux. La dernière énonce que les activités d'une couleur globale sont concentrées dans un seul sous-réseau à un instant donné.

4.4.3.5 Écriture tensorielle du générateur de la CM

Le générateur de la chaîne de MARKOV à temps continu agrégée est donné par son expression tensorielle.

$$Q' = \bigoplus_{k=1}^{K} Q'_k + \sum_{t \in TO} \sum_{d} \theta(t,d) \left[\bigotimes_{k=1}^{K} C_k(t,d) - \bigotimes_{k=1}^{K} A_k(t,d) \right] \tag{4.3}$$

où $d = (d_i^j)_n^{e_i(t)}$ (avec n_i le nombre de sous-classes statiques de C_i et $1 \leq d_i^j \leq n_i$) est un choix de sous-classes statiques pour les franchissements symboliques de t.

Avec les notations suivantes.

- $\forall\ \langle \lambda, \mu \rangle$ (fonctions d'instanciation dans $\overline{N}_k$ pour t), $\overline{\mathscr{M}}_k$ et $\overline{\mathscr{M}}'_k$ dans $\overline{SRS}_k$:

$$1_{(t,d,\langle \lambda,\mu \rangle, \overline{\mathscr{M}}_k, \overline{\mathscr{M}}'_k)} = \begin{cases} 1 & \text{si} \quad d = (d(Z_i^{\lambda_i(j)}))_n^{e_i(t)} \\ & \text{et} \quad \overline{\mathscr{M}}_k[t(\langle \lambda, \mu \rangle)\rangle \overline{\mathscr{M}}'_k \\ 0 & \text{sinon} \end{cases}$$

$$1_{(t,d,\overline{\mathscr{M}}_k, \overline{\mathscr{M}}'_k)} = \bigvee_{\langle \lambda,\mu \rangle} 1_{(t,d,\langle \lambda,\mu \rangle, \overline{\mathscr{M}}_k, \overline{\mathscr{M}}'_k)}$$

  $\bigvee$ étant l'addition booléenne (ou logique).
- pour $t \in TO_k$

$$F_{(\langle \lambda,\mu \rangle, \overline{\mathscr{M}}_k, \overline{\mathscr{M}}'_k)} = \prod_{i=1}^{h} \prod_{j=1}^{m_i} \frac{\text{card}(Z_i^j)!}{(\text{card}(Z_i^j) - \mu_i^j)!}$$

  avec h le plus grand indice de classe non ordonnée de $C(t)$.

4.5 Algorithme d'analyse structurée d'une (dé)composition synchrone/asynchrone de SWNs

La démarche d'analyse d'une composition synchrone ou asynchrone est résumée dans l'algorithme suivant :

Les travaux d'analyse structurée ont été exploités dans [66; 67; 65] afin de fournir une implémentation du noyau principal de la méthode, à savoir la génération des SRGs des sous-réseaux étendus, le

ALGORITHME Analyse d'une décomposition de SWN
BEGIN
Soit N un SWN.

1. Décomposer le SWN global N en K SWNs communicant soit par composition synchrone, soit par composition asynchrone. On obtient ainsi un ensemble de SWNs $((N_k)_{k\in K}$.
2. Vérifier les conditions syntaxiques d'applicabilité de la décomposition du SWN N.
3. Si les conditions sont vérifiées :
   (a) Etendre chaque N_k pour tenir compte des interactions avec les autres sous-réseaux. Les SWN étendus sont notés N_k.
   (b) Générer les SRGs de ces réseaux SWNs étendus.
   (c) Calculer le produit synchronisé de ces SRGs et la représentation tensorielle du générateur de la chaîne de Markov agrégée sous-jacente.
   (d) Calculer la distribution stationnaire ou transitoire du modèle agrégé et les indices de performance recherchés.

END

calcul de la forme tensorielle du générateur de la chaîne de Markov agrégée, ainsi que le calcul de la distribution stationnaire et de quelques indices de performance. Des études de cas de décomposition synchrone et décomposition asynchrone ont également été réalisées. Un gain intéressant en temps de calcul et en occupation mémoire a été obtenu par l'agrégation implicite issue du formalisme des SWNs d'une part, et d'autre part par la méthode d'analyse structurée de la décomposition synchrone et asynchrone.

4.6 Conclusion

Ce chapitre a fait l'objet de la description du modèle SWN sur lequel nous nous basons dans notre étude. Nous avons également présenté la méthode structurée de décomposition synchrone et asynchrone développée pour le modèle SWN. L'intérêt de cette méthode est qu'elle permet une réduction dramatique du temps de calcul, ainsi qu'une capacité à manipuler des espaces d'états très larges.

Il serait donc intéressant de tirer profit des avantages de cette méthode pour l'analyse des performances des systèmes basés composants. En effet, la méthode est compositionnelle et peut être adaptée à l'analyse d'un assemblage de composants. Elle peut être exploitée pour permettre une rapidité de calcul d'indices de performances lors de l'analyse d'un CBS, mais aussi permettre d'analyser des CBS caractérisés par un espace d'états important, jusqu'ici difficiles ou impossible à analyser avec les outils actuels.

Cependant, la méthode de décomposition a été proposée pour l'analyse d'une décomposition uniquement synchrone de SWNs ou d'une décomposition uniquement asynchrone de SWNs. Or, un CBS peut contenir des composants interagissant de différentes manières. Le mélang e des deux types de décomposition au sein d'un même modèle global risque d'altérer l'applicabilité de la méthode structurée.

Ces problèmes d'adaptation à l'analyse des CBS ainsi que le développement d'une méthode générique d'analyse des CBS incluant les deux étapes de modélisation et d'analyse sont étudiés dans la deuxième partie de ce manuscrit.

PARTIE II

Analyse structurée des CBS

CHAPITRE 5

Méthode d'analyse structurée des CBS

5.1 Introduction

L'étude des systèmes basés composants a permis de dégager les caractéristiques essentielles à considérer pour effectuer une analyse d'un CBS. À partir de ces caractéristiques, nous avons développé une méthode générique pour la modélisation et l'analyse comportementale et quantitative (calcul de performances) des systèmes basés composants. Nous nous concentrons sur l'aspect performances et nous tirons profit de la compositionnalité de ces systèmes afin de réduire la complexité d'analyse en termes de temps de calcul et d'occupation mémoire et d'élargir le champs d'application à l'analyse de systèmes larges.

Ce chapitre décrit les différentes étapes de notre approche, situe les problèmes étudiés pour développer notre méthode et détaille la première étape consistant en la construction d'un modèle global pour le CBS à analyser.

5.2 Vue d'ensemble

5.2.1 Objectifs

Dans cette étude, nous nous focalisons sur l'analyse des performances d'un système basé composant. Notre objectif est de fournir à un concepteur une méthode générique qui lui permet, à partir de la description architecturale de son système et des codes sources de ses composants, de dériver les performances du système. Ainsi, il peut analyser *à priori* un système et connaître ses performances, avant son introduction dans le processus de développement : D'une part, il peut détecter et éventuellement corriger des erreurs de conception et des dysfonctionnements. D'autre part, il peut quantifier la performabilité du système vis-à-vis des objectifs pour lesquels il est construit.

Par ailleurs, en plus de rechercher des performances sur le système global, notre méthode va permettre également l'évaluation d'un composant en isolation.

À cet effet, il est nécessaire de pouvoir :

1. Construire un modèle stochastique à partir de la description de l'architecture à composants et des comportements de ses composants.
2. Analyser le plus efficacement possible le modèle obtenu tout en tirant profit de la compositionnalité du système.

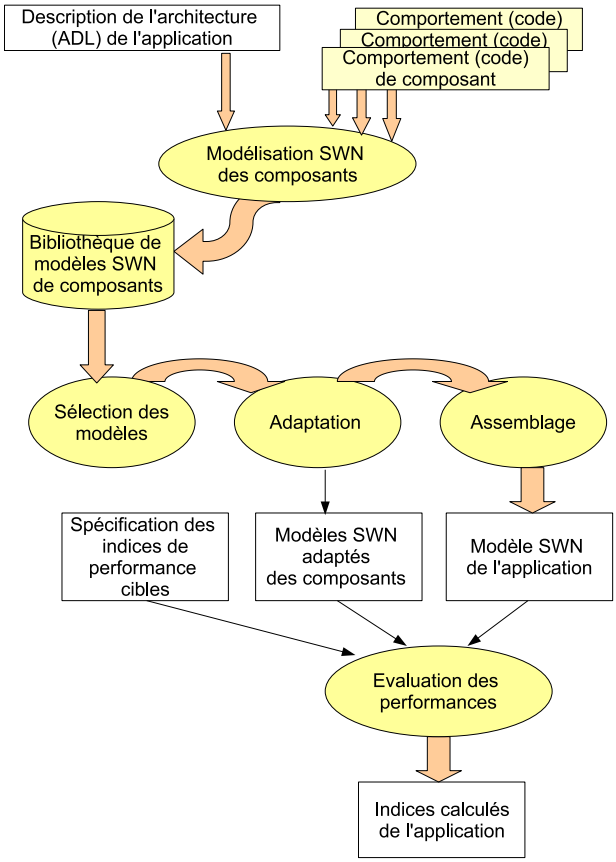


Figure 5.1: Principe général de la méthode d'analyse d'un CBS

5.2.2 Principe

L'idée de base sur laquelle repose notre méthode d'analyse consiste à passer de la description d'une architecture à composants vers un modèle stochastique analysable le plus efficacement possible. Nous utilisons à cet effet le formalisme Réseau de Petri stochastique bien formé (SWN) pour les raisons citées dans la conclusion du chapitre 3.

Les grandes lignes de notre méthode figurant sur le schéma de la figure 5.1 sont données ci-après :

1. Déterminer à partir de la description d'un CBS les éléments de l'architecture. Trois éléments peuvent être extraits : les composants, éventuellement les connecteurs de composants et les interactions entre composants.
2. Modéliser les composants et éventuellement les interactions complexes et connecteurs par des SWNs. Le modèle SWN d'un composant doit préciser quelles parties représentent ses interfaces. Bien entendu, des bibliothèques peuvent contenir des modèles SWNs de composants construits au préalable, qu'il est alors possible de réutiliser.
3. Adapter les modèles SWN obtenus de manière à être composables entre eux d'une part, et d'autre part permettre de mener une analyse structurée des performances.
4. Assembler les modèles SWN adaptés pour construire le SWN global du CBS, en interconnectant les interfaces concernées selon la description donnée de l'architecture. Le modèle global est vu

comme une composition des modèles SWNs des composants et des interconnexions. En fait, les modèles SWN des composants sont des SWNs paramétrés, dans le sens où un ensemble de paramètres doit être spécifié à l'assemblage des composants, comme les couleurs synchronisées, les vitesses des activités, etc.

5. Effectuer l'analyse du SWN global obtenu en s'appuyant sur sa structure compositionnelle et connaissant les indices de performance recherchés (voir section 6.5).

Un exemple d'étude suivant ce principe a été présenté dans [190].

5.2.3 Types de modèle à composants

Plusieurs modèles à composants ont été présentés dans le chapitre 3. La plupart de ces modèles s'accordent sur un ensemble de caractéristiques communes. Toutefois, ils diffèrent sur certaines propriétés telles que la nature d'un composant (entité de conception ou entité d'exécution), des types d'interaction variés, des propriétés fonctionnelles et éventuellement non fonctionnelles.

Globalement, notre méthode s'applique à tout type de composant. Les points de discordance entre les modèles de composant sont pris en considération comme suit :

5.2.3.1 Nature d'un composant

Un composant peut être défini comme une entité de conception ou une entité d'exécution. Cette nature est à considérer lors de l'étape de modélisation. En effet, une entité d'exécution s'exécute dans un environnement d'exécution. Cet environnement influe sur les performances globales du système considéré. Il doit donc être pris en compte lors de la modélisation et l'évaluation des performances. En général, il peut être explicitement modélisé comme une partie du modèle global obtenu du système, comme il peut l'être implicitement en associant des vitesses de franchissement dépendantes de l'environnement aux activités modélisées des composants.

Lorsque le composant est considéré comme une unité de conception, l'évaluation d'un assemblage de telles unités peut se faire sans tenir compte d'un environnement précis, afin de quantifier les performances du système indépendamment de la platefeforme d'exécution. Néanmoins, il est plus judicieux de considérer un environnement donné lors de la modélisation afin de rendre compte de l'impact de l'environnement sur les performances. L'assemblage à analyser peut alors être évalué pour plusieurs environnements pour des fins de comparaison.

5.2.3.2 Types d'interactions

Les composants offrent des types d'interfaces (et donc d'interactions) diverses. Principalement, on trouve deux types fréquents d'interactions :

- L'interaction de type invocation de service, qui correspond à un appel synchrone de méthode, comme par exemple un appel RPC ou RMI, ou l'envoi d'un message (avec attente de réponse) et sa réception (avec traitement puis réponse) entre un client et un service. Durant le traitement de la requête, le client n'effectue rien, et est en attente de la réponse.
- Une interaction asynchrone de notification/réception d'événements. Contrairement au premier type d'interaction, un composant émettant un événement à un autre peut continuer son travail pendant que le deuxième composant traite l'événement reçu.

Selon le type d'interaction, une modélisation appropriée est nécessaire (voir section 5.5).

On peut également trouver d'autres types d'interaction dans les modèles à composant, comme par exemple le partage de données. La modélisation de ces interactions peut être aisément déduite : le partage de données peut être vue par exemple comme un cas particulier de l'invocation de méthode dans le cas de variables partagées. Quant aux interactions complexes constituées en plusieurs étapes d'un protocole de communication, elles peuvent elles-mêmes être modélisées par un modèle SWN propre à elle.

Dans notre étude, nous nous sommes restreints aux deux types d'interactions (invocation de service et communication par événements) puisqu'ils reviennent souvent dans les modèles à composant.

5.2.3.3 Propriétés fonctionnelles et non fonctionnelles

Un composant peut être caractérisé par des propriétés fonctionnelles et non fonctionnelles. Les propriétés fonctionnelles sont exposées à travers des interfaces. Les propriétés non fonctionnelles peuvent être données sous-forme d'interfaces de composants ou sous-forme de services offerts par l'environnement d'exécution des composants (tel qu'un conteneur). La modélisation de ces propriétés est à priori nécessaire. Cependant, il est possible de se passer de la modélisation de certaines propriétés non fonctionnelles liées aux reconfigurations dynamiques d'une application basée composant. En effet, la reconfiguration dynamique d'une architecture implique sa modification en cours d'exécution, ce qui fait passer le système par une situation instable (transitoire) pendant un laps de temps.

Comme nous nous intéressons au calcul d'indices de performances généralement effectué pendant des périodes longues stables (stationnaires), nous nous sommes limités à modéliser les interfaces fonctionnelles et les non fonctionnelles n'ayant pas trait aux reconfigurations dynamiques (voir plus de détails en section 5.5.2).

Avant de détailler notre approche, nous introduisons un ensemble de termes liés aux modèles SWNs que nous utilisons tout au long de notre étude.

5.2.4 Terminologie

5.2.4.1 Modèles SWNs des composants C-SWNs et CC-SWNs

Un élément d'un CBS (composant ou connecteur) est modélisé par un modèle SWN, construit à partir du comportement du composant et des sous-modèles de ses interfaces.

- Lorsque le composant est primitif, son contenu consiste en un simple composant, Il est alors modélisé par un expert SWN : les places représentent des ressources de stockage de données, l'état d'entités actives (threads, processus), etc; les transitions correspondent aux activités consommant des données et produisant d'autres.
- Un composant composite contient un ensemble de sous-composants interconnectés. En conséquence le modèle associé à son comportement consiste en l'interconnexion des modèles de comportements de ses sous-composants, et éventuellement des modèles associés aux connecteurs adaptant une interaction donnée.

Nous appelons *Composants SWN* ou *C-SWNs* les modèles des composants primitifs ou composites. Les C-SWNs peuvent être groupés dans des bibliothèques pour des fins de réutilisation, afin de permettre à un utilisateur non familier avec le formalisme SWN d'évaluer son application sans effort de modélisation au préalable.

La traduction d'une interface d'un composant dans le C-SWN correspondant est fortement dépendante de la sémantique du modèle de composant et du type d'interaction engagée par l'interface. En

général, le sous-modèle d'une interface consiste en un sous-réseau du C-SWN pouvant être formé uniquement d'un ensemble de places, d'un ensemble de transitions, ou même un sous-réseau plus complexe constitué d'une combinaison de places et de transitions. Nous donnons la traduction des différents types d'interfaces dans la section 5.5.

L'étape qui suit la modélisation des composants consiste à adapter les modèles C-SWNs obtenus de telle manière à être composable entre eux d'une part et pour permettre de mener une analyse structurée des performances. Cette adaptation consiste en une modification des C-SWNs sans altérer la sémantique de modélisation. Les modèles transformés sont dits *Composants SWNs Composables* ou *CC-SWNs*.

5.2.4.2 Modèles SWNs d'interaction I-SWNs et IC-SWNs

Les composants sont reliés par des interactions décrites dans la description d'architecture du système à modéliser. Les C-SWNs (et CC-SWNs) doivent également être interconnectés par ces interactions. Celles-ci peuvent être basiques suivant un schéma de communication simple du type requête/réponse, tel que la communication par invocation de service. Elle peut être également complexe consistant en un protocole de communication complexe. Le premier schéma d'interaction (simple) ne nécessite pas de modélisation propre à lui dans notre approche. Il est plutôt implicitement exprimé à travers la modélisation des interfaces associées. À l'encontre, lorsque l'interaction est complexe, elle est traduite par un modèle SWN appelé *Interaction SWN* ou *I-SWN*, de la même manière qu'un composant.

En adaptant le modèle I-SWN à notre méthode d'analyse, nous obtenons des *Interactions SWN composables* ou *CI-SWNs*.

5.2.4.3 Le modèle SWN global du CBS, G-SWN

Le modèle global d'un CBS, dit *Global SWN* ou *G-SWN* est généré à partir de l'ensemble des CC-SWNs et leurs CI-SWNs correspondants. L'ensemble de ces modèles est composé selon la description d'architecture du système modélisé, à travers la fusion des éléments d'interfaces interconnectées.

5.3 Étapes de notre approche

Afin d'analyser un CBS, notre méthode comprend deux phases principales :

1. Génération d'un SWN global (le G-SWN) modélisant le CBS à partir de sa description d'architecture et des comportements de ses composants. Ce modèle est vu comme une composition des modèles SWNs des composants et des interactions.
2. Application de la méthode d'analyse structurée sur la composition obtenue (le G-SWN) pour le calcul d'indices de performances.

Ces deux phases sont détaillées en cinq étapes (voir figure 5.2). Les trois premières étapes sont consacrées à la génération du G-SWN. Les deux autres étapes concernent l'analyse proprement dite des performances du système. Ce sétapes sont :

1. Traduction de la description ADL du CBS et de la description des comportements (codes source) des composants et de leurs interactions dans le formalisme SWN, donnant lieu à un ensemble de C-SWNs et un ensemble de I-SWNs.
2. Modification des C-SWNs et I-SWNs pour qu'ils soient composables avec d'autres au sens de la composition des réseaux de Petri (fusion des places ou transitions), ainsi que pour permettre une analyse structurée. L'ensemble des modèles obtenus est constitué des CC-SWNs et des CI-SWNs.

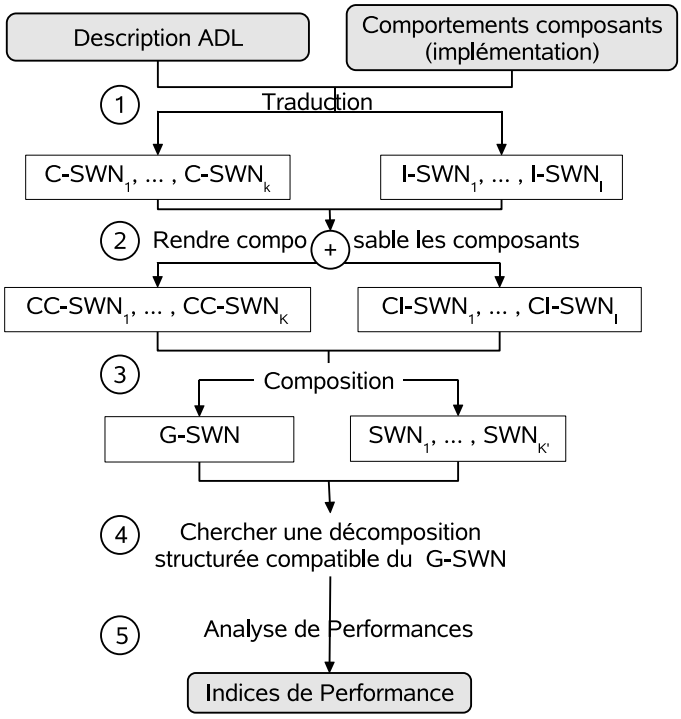


Figure 5.2: Méthode d'analyse d'un CBS

3. Composition des CC-SWNs et CI-SWNs à travers la fusion des éléments des interfaces, fournissant ainsi le modèle global G-SWN correspondant au système. Les CC-SWNs et CI-SWNs sont maintenant vus comme un ensemble unique de sous-réseaux $\text{SWN}_1,\ldots,\text{SWN}_{K'}$ (voir figure 5.2).
4. À partir de l'ensemble des $\text{SWN}_{k\ k\in\{1,\ldots,K'\}}$, recherche d'une configuration de SWNs $(\mathscr{N}_k)_{1\leq k\leq K''}$ représentant une décomposition possible du G-SWN respectant les conditions syntaxiques énoncées dans la méthode de décomposition [97; 98] (section 4.4) pour une représentation structurée du SRG et de son générateur agrégé. Chacun de ces SWNs peut être un des $(\text{SWN}_k)_{1\leq k\leq K'}$ ou bien un groupement d'un sous-ensemble d'entre eux. En effet, lorsque les conditions ne sont pas satisfaites entre un sous-ensemble de SWN_k, ces SWNs sont groupés dans un nouveau SWN qui va les remplacer dans la configuration recherchée.
5. Une fois les conditions syntaxiques sont satisfaites, application de la méthode d'analyse structurée pour le calcul des indices de performance (voir le chapitre 6).

Le développement de ces étapes pour une application de la méthode structurée a suscité plusieurs problèmes, décrits dans ce qui suit.

5.4 Problèmes à résoudre

Afin de réaliser une analyse de performances efficace en temps de calcul et en espace mémoire, nous nous proposons d'étendre la méthode de décomposition structurée de SWNs [97; 98] présentée dans le chapitre précédent et de l'adapter à l'analyse des CBS. Ceci requiert de traiter trois problèmes majeurs :

1. Le premier problème consiste à composer les modèles SWNs des composants au lieu de décomposer un modèle global tel que réalisé dans la méthode de décomposition structurée.
2. Le second problème tente de faire correspondre une composition synchrone ou asynchrone de SWNs à une interconnexion de composants SWNs du CBS.
3. Le troisième problème consiste à étudier l'impact de la présence simultanée de compositions synchrones et asynchrones au sein du même modèle global, étant donné que les résultats de la méthode structurée ont été énoncés pour une (dé)composition uniquement synchrone de SWNs ou bien une (dé)composition uniquement asynchrone de SWNs.

5.4.1 De la décomposition à la composition

La méthode de (dé)composition synchrone et asynchrone de SWNs de [97; 98] est basée sur la décomposition manuelle d'un réseau global en sous-réseaux, puis une analyse de la composition de ces sous-réseaux.

Dans le contexte des CBS, nous adoptons une approche ascendante : connaissant les composants, nous partons de la définition des composants interconnectés entre eux pour aboutir aux performances. Ceci facilite l'analyse compositionnelle d'une part puisque les composants sont connus. D'autre part, la composition est importante pour la vérification des conditions syntaxiques de la décomposition structurée des SWNs correspondant aux composants, qui doit se faire au niveau du modèle global de l'assemblage.

Toutefois, l'application de la méthode structurée nécessite d'identifier et de modéliser les interactions d'un composant avec les autres, afin de procéder à l'extension des modèles SWNs obtenus. Pour répondre à cette préoccupation, nous proposons de modéliser les interfaces des composants d'une manière adéquate au type d'interaction engendrée, adaptée aux conditions de l'analyse structurée. Les détails sont donnés en section 5.5.

5.4.2 Mise en correspondance d'un assemblage de composants à une composition de SWNs

L'interconnexion d'un ensemble de composants SWNs d'un CBS donne lieu à des compositions quelconques de modèles SWNs. Afin d'exploiter la méthode de (dé)composition synchrone et asynchrone de SWNs pour réduire la complexité d'analyse des CBS, il est nécessaire de structurer ces compositions dans la forme d'une composition synchrone ou asynchrone telle que définie dans la section 4.4.

En fait, ce problème est intrinsèquement dépendant du point précédent : la structuration d'un assemblage de composants en une composition synchrone ou asynchrone revient à adopter une modélisation appropriée des interfaces. Comme des types d'interfaces (et donc d'interactions) se distinguent, la mise en correspondance d'un assemblage avec un type de composition donné dépend étroitement du modèle adopté pour l'interaction (interfaces) sous-jacente à l'assemblage. Ce point est abordé en section 5.5 et montré plus loin, en section 6.3.

5.4.3 Composition mixte synchrone et asynchrone

Les résultats énoncés dans la méthode structurée sont applicables pour une configuration uniquement synchrone de SWNs, ou uniquement asynchrone. Le mixage de compositions synchrones et asynchrones de SWNs au sein du même modèle risque d'altérer la satisfaction des conditions d'application de la méthode structurée. Ceci est dû au fait que plus des interactions interviennent entre les entités colorées,

plus la structure précédemment établie (sous-entendu le comportement structuré des couleurs) risque d'être brisée. Nous appelons ce type de composition *composition mixte* de SWNs.

Ce troisième problème est plus critique et nécessite de revoir les conditions syntaxiques énoncées auparavant, pour en spécifier de nouvelles conditions nécessaires pour mener une analyse structurée. L'investigation des conséquences de ce problème et la spécification de nouvelles conditions sont présentées dans le chapitre 6.

Dans ce qui suit, nous commençons par développer la première phase de notre méthode.

5.5 Passage d'un CBS à son modèle SWN : construction du G-SWN

Dans cette section, nous détaillons les trois premières étapes de notre approche, l'objectif étant de générer *automatiquement* un modèle stochastique pour le CBS à analyser, afin de l'exploiter dans l'analyse compositionnelle structurée. Pour effectuer le traduction automatique de la description du CBS vers le formalisme SWN, un ensemble de règles est proposé [189; 191].

Tout d'abord, nous présentons un exemple d'application basée composant que nous utilisons tout au long de ce chapitre pour illustrer notre méthode. Nous discutons nos choix de modélisation. Ensuite, nous détaillons la modélisation des interfaces de composants, des composants primitifs et composites, de la notion de conteneur, puis la construction du G-SWN.

5.5.1 Un système de gestion des stocks de bourse, exemple de CBS

Tout au long du reste de ce chapitre, nous allons illustrer les démarches adoptées dans notre approche par un exemple d'une application basée composant, à savoir un *système de gestion des informations des stocks de bourse*. Cette application, illustrée par la figure 5.3, est basée sur le modèle CCM introduit précédemment dans l'exemple 1.4 et détaillé en section 9.2. Elle s'inspire d'un exemple d'application CCM présentée dans [194] que nous avons étendue pour englober plusieurs concepts à modéliser.

Le système gère une base de donnée d'informations en temps réel sur la bourse. Il est constitué de deux composants *StockDistributor* gérant la base de données, deux composants *StockBroker* guettant les changements des informations de la bourse et un composant *Executor* traitant un sous-ensemble de ces informations de stocks. Lorsque la valeur d'un stock particulier change, un composant StockDistributor publie un événement contenant le nom du stock, via une source d'événement, en notifiant les puits d'événements intéressés par ce changement de stock; ces puits étant implantés par les composants StockBroker. Chaque composant StockDistributor est responsable d'un ensemble de stocks.

À la réception d'une notification d'événement, l'un des composants StockBrokers invoque un service du composant Executor à travers une interface cliente, alors que le second composant StockBroker traite localement l'événement. Le composant Executor traite la requête du premier StockBroker, génère des données et invoque lui-même une requête au service de persistance offert par le conteneur pour le stockage des données.

Lorsqu'aucun événement n'est généré, les composants StockBroker exécutent continuellement une tâche locale.

Une portion de la définition des composants StockBroker et StockDistributor et des interfaces associées est donné ci-après en langage CIDL (*Component Implementation Definition Language*) et IDL 3.x, ainsi qu'une portion du comportement (code source) du modèle (template) du composant StockBroker.

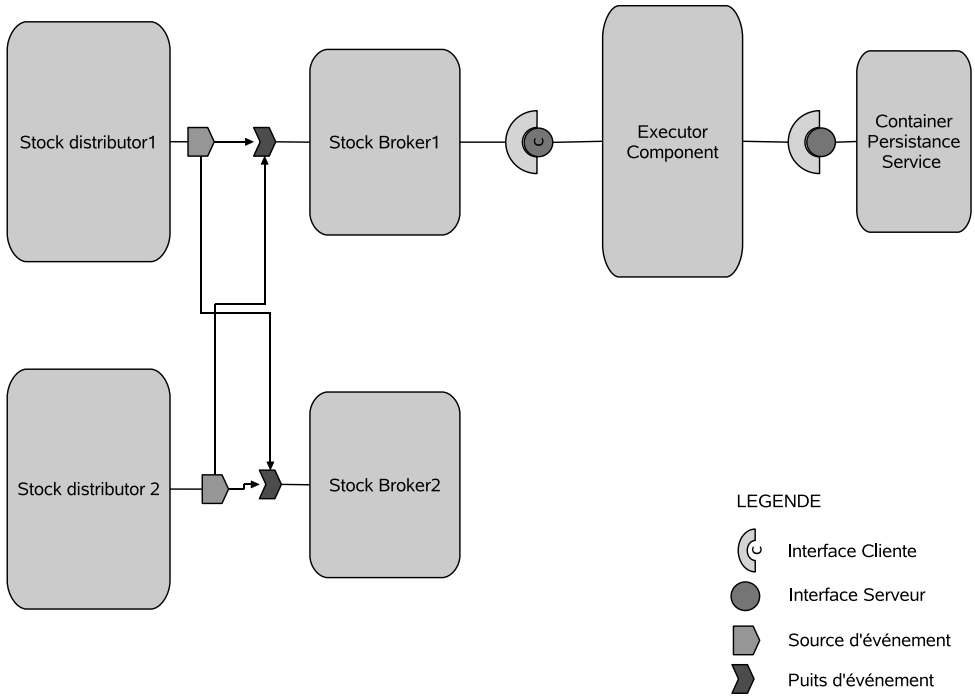


Figure 5.3: Une application basée composant

```
interface StockQuoter
{ StockInfo get_stock_info (in string stock_name);
  };
interface Trigger
{ void start ();
  void stop ();
  }
struct StockInfo
{ string name;
  long high;
  long low;
  long last;
  // ...
  };
eventtype StockName
{ public string name;
  };

component StockBroker
{ consumes StockName notifier_in;
  uses StockQuoter quoter_info_in;
  };
```

```
component StockDistributor supports Trigger
{ publishes StockName notifier_out;
  provides StockQuoter quoter_info_out;
  attribute long notification_rate;
  };

class StockBroker_Servant : public virtual POA_StockBroker,
      public virtual PortableServer::RefCountServantBase
{
public:
    // Operation for the 'consumes" port.
    StockNameConsumer_ptr get_consumer_notifier_in ();
    // Operation for the 'uses" port.
    void connect_quoter_info_in (StockQuoter_ptr c);
    StockQuoter_ptr disconnect_quoter_info_in ();
    StockQuoter_ptr get_connection_quoter_info_in ();
    // Operations for Navigation interface.
    CORBA::Object_ptr   provide_facet (const char *name);
    Components::FacetDescriptions get_all_facets ();
    Components::FacetDescriptions
       get_named_facets (const ::Components::NameList &);
    // Operations for Receptacles interface.
    Components::Cookie *connect (const char *name,
                                 CORBA::Object_ptr connection);
    CORBA::Object_ptr  disconnect (const char *name,
                                   ::Components::Cookie *ck);
    Components::ReceptacleDescriptions *get_all_receptacles ();
    // Operations for Events interface.
    Components::EventConsumerBase_ptr
       get_consumer (const char *sink_name);
    Components::Cookie *subscribe (const char *publisher_name,
                ::Components::EventConsumerBase_ptr subscriber);
    // more operations..
};
```

5.5.2 Considérations générales

5.5.2.1 Modélisation de configurations stables

Avant d'être disponible aux utilisateurs, une application basée composant doit être déployée et configurée. Ces étapes incluent notamment la définition des contextes d'initialisation et d'exécution, la souscription des composants munis d'une interface puits d'événements à certaines classes d'événements, ... Lorsque ces étapes sont achevées, l'application est activée. Dans ce cas, on dit que l'application est dans une *configuration stable* ou *stationnaire, à l'équilibre*. Pendant son exécution, l'*architecture* de l'application peut évoluer à travers des reconfigurations dynamiques, telles que par exemple la création ou suppression d'instances de composants ou de liaisons, ou encore la souscription/désouscription des

composants à une classe spécifique d'événement, ...

Bien que la capacité d'évolution d'une architecture soit une des principales propriétés de plusieurs middlewares offertes aux CBS, il apparaît toutefois que l'analyse comportementale d'un CBS ne peut être efficacement effectuée avec un unique modèle formel du système. En fait, toute modification de l'architecture exige une modélisation spécifique, principalement consacrée à vérifier que, à partir d'une configuration A, l'application atteindra par la suite une configuration donnée B. En revanche, il est possible d'analyser une configuration donnée (dite A ou B), en abordant les deux aspects qualitatif (accessibilité, exclusion mutuelle, etc.), et quantitatif (calcul des indices de performance) de cette configuration. Il est encore possible de comparer les résultats d'analyse de deux configurations A et B.

Dans le contexte de l'évaluation des performances, la commutation d'une configuration A à une autre B correspond probablement à une période de temps assez "courte" (phase transitoire, analyse d'horizon fini), tandis que nous sommes intéressés par des indices de performance de systèmes logiciels, calculés principalement sur de longues périodes (analyse stable).

Dans notre travail, nous n'abordons pas la vérification des comportements de reconfiguration des CBSs et nous nous focalisons sur des architectures "stables" (i.e. fixes). Par conséquent, nous ne modélisons pas les services (propriétés) non fonctionnels (les) qui concernent les étapes d'initialisation et de reconfiguration. Cependant, nous pouvons évidemment *comparer* les résultats d'analyse des deux configurations A et B, chacune étudiée avec notre méthode.

5.5.2.2 Dépendances d'implantation

Comme notre objectif est de dériver des indices de performance d'un CBS, nous mettons l'accent sur le fait que la description de l'architecture d'un CBS n'est pas suffisante pour la modélisation des performances: *elle doit être complétée par des informations sur l'implantation* du modèle de composant. En effet, une implantation d'un modèle de composant peut différer d'une autre. C'est le cas, par exemple, pour le modèle FRACTAL où l'implantation *Julia* utilise une méthode d'invocation synchrone de service, alors que l'implantation *Fractive* emploie une opération d'invocation asynchrone (en retard).

5.5.2.3 Couleurs

Les entités de données et les entités actives des composants sont modélisées par ses classes de couleur de base. Les entités de données consistent en un flux de données tel que les requêtes, les paramètres de requêtes, les données des requêtes, les données des événements ou encore les états des ressources. Les entités actives correspondent aux flux d'exécution processus, threads, etc.

Dans notre contexte, nous nous focalisons sur la modélisation des entités impliquées dans les interfaces et les interactions, puisque les entités utilisées à l'intérieur d'un composant dépendent uniquement de ce composant. Plus précisément, nous adoptons les classes de base de couleur pour modéliser les méthodes d'interfaces invoquées ainsi que leurs paramètres, les types d'événements, les éventuelles ressources et les flux d'exécution d'un composant. Concernant ce dernier point, il est nécessaire de connaître si le composant supporte plusieurs threads (multithreaded) ou non. Il est également utile de colorer les données liées aux événements lorsque les messages associés ont un impact d'un point de vue performance.

Exemple 5.33. *Nous illustrons les classes de couleur de base par celles utilisées dans notre application exemple du système de gestion des informations de stocks de la bourse. Les classes utilisées sont les classes d'événements de changement de stocks, les threads des composants StockDistributor et Stock-*

Broker, les requêtes d'informations de stock, les threads serveurs du composant Executor, ses méthodes et paramètres, ...

5.5.2.4 Modélisation des entités d'événements

Lorsqu'une communication par événement est utilisée dans un modèle de composant, il est possible que les événements soient acheminés à travers un canal d'événements (voir chapitre 1, section 1.3.1.2). Ce mécanisme est généralement précisé dans la spécification du service de gestion d'événements supporté par le middleware de la plateforme d'exécution (Un exemple de spécification est celle du service de notification de CORBA [168]). Dans ce cas, est-il nécessaire d'adopter une modélisation explicite du canal d'événement ?

En fait, lorsque le canal d'événements reçoit une notification d'événement d'une source, il accuse généralement réception de cet événement à la source. Puis, il achemine la notification vers tous les puits intéressés par cet événement. L'accusé de réception peut être envoyé à la source de l'événement dès la réception de la notification par le canal. Il peut également être retardé jusqu'à ce que les puits eux-mêmes reçoivent la notification d'événement et en accusent réception. Ceci peut engendrer une influence sur les temps de réponse et plus globalement les performances du système global.

De ce fait, afin de prendre en compte l'influence de l'envoi/réception des événements sur les performances d'un système, il est plus intéressant de modéliser explicitement le canal d'événements.

5.5.2.5 Services de conteneur

Un conteneur agit comme un gestionnaire de composants. Il offre trois types de services:

1. Des opérations liées au cycle de vie des composants, telles que créer, supprimer, modifier un composant, etc.
2. Des opérations liées aux liaisons entre composants et invocations de services de composants, et
3. Des appels aux services du système d'exploitation ou du middleware, comme par exemple le service de gestion des transactions, gestion de la sécurité, gestion des événements et gestion de la persistance.

Le support de gestion du cycle de vie et des liaisons est lié aux configurations transitoires qui ne sont pas couvertes par la présente étude pour les raisons citées en section 5.5.2.1. Cependant, nous modélisons les autres services de conteneur, non liés à la modification de l'architecture. Ces services sont considérés comme des composants abstraits munis de leurs propres interfaces et sont modélisés par des SWNs de la même manière que les composants (voir section 5.5.7).

Après avoir décrit les choix de modélisation que nous avons adopté, nous abordons maintenant le détail de la phase de modélisation.

5.5.3 Modélisation des interfaces d'invocation de service

L'invocation de service met en jeu deux composants où l'un est muni d'une interface client requérant un service et l'autre expose une interface serveur offrant le service demandé par le client. Elle est initiée lorsque le client spécifie la méthode (service) requise et ses paramètres. Ainsi, nous traduisons les correspondances entre les interfaces client/serveur d'un composant et leurs modèles SWN en appliquant les règles de traduction suivantes.

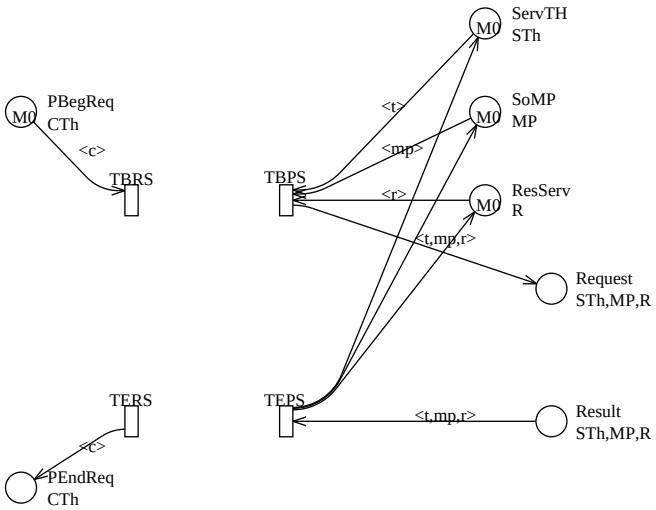


Figure 5.4: Modèles SWNs des interfaces: Client (à gauche), serveur (à droite)

Règle de traduction 1: Interface serveur
Une interface serveur, identifiée par un ensemble de couleurs STh modélisant les threads serveurs et offrant un ensemble MP d'opérations ou de méthodes munies de leurs paramètres, est modélisées par deux transitions, notées t_{BPS} et t_{EPS} représentant respectivement le début et la fin du service fourni (figure 5.4 (à droite)). La transition t_{BPS} est contrôlée par deux places *ServTH* et *SoMP* modélisant respectivement les threads serveurs et les méthodes avec leurs paramètres. Éventuellement, une troisième place *ResServ*, colorée avec une classe de base R, est utilisée pour la modélisation des ressources nécessaires pendant l'exécution d'un service. La transition t_{EPS} est contrôlée par une place *Result*, colorée par des tuples appartenant à $STh \times MP \times R$ modélisant le résultat du traitement de la requête.

Nous formalisons la définition d'une interface serveur d'un C-SWN comme suit :

Définition 5.1 (Interface serveur)**.** Une interface serveur I d'un C-SWN est un tuple $< P_I, T_I >$, tel que:
• $P_I = \{ServTH, SoMP, ResServ\}$;
• $T_I = \{t_{BPS}, t_{EPS}\}$ avec $\mathscr{D}(t_{BPS}) = \mathscr{D}(t_{EPS}) = ST \times MP \times R$;
• $Pre(ServTH, t_{BPS}) = Post(ServTH, t_{EPS}) = Id$;
• $Pre(SoMP, t_{BPS}) = Post(SoMP, t_{EPS}) = Id$.

Règle de traduction 2: Interface client
Une interface client, identifiée par un ensemble de couleurs CTh modélisant des threads de requêtes du composant client est modélisée avec deux transitions t_{BRS} et t_{ERS} représentant respectivement le début et la fin de l'invocation de la requête (figure 5.4 (à gauche)). La transition t_{BRS} (resp.t_{ERS}) est contrôlée par une place *PBegReq* (resp. *PEndReq*) colorée avec la classe CTh, modélisant respectivement les threads requêteurs et ces mêmes threads libérés à la fin du traitement de la requête.

Nous donnons ci-après la définition formelle de l'interface client d'un C-SWN.

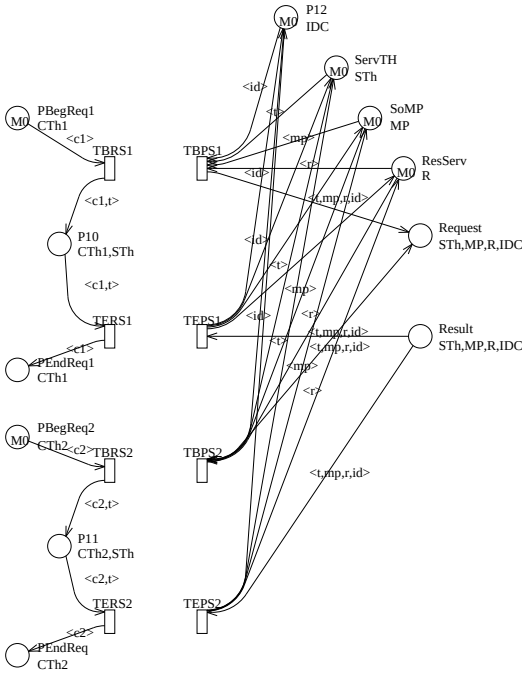


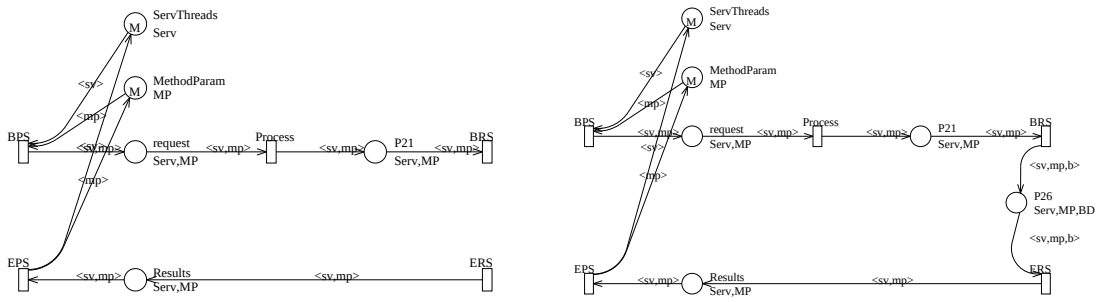
Figure 5.5: Modèles CC-SWNs de plusieurs interfaces clientes connectées à la même interface serveur

Définition 5.2 (Interface client)**.** Une interface client I d'un C-SWN est un couple de transition (t_{BRS}, t_{ERS}), tel que:
• $\mathscr{D}(t_{BRS}) = \mathscr{D}(t_{ERS}) = CT \times MP$ ($\mathscr{D}(x)$ représente le domaine de couleur du noeud x);
• $t_{BRS}^{\bullet} =^{\bullet} t_{ERS} = \emptyset$;
• $\exists! p_{BegReq} \in P$, $\exists! p_{EndReq} \in P$, $p_{BegReq} \neq p_{EndReq}$,
$Pre(p_{BegReq}, t_{BRS}) = Post(p_{EndReq}, t_{ERS}) = Id$.

Les modèles des interfaces client/serveur dépendent à priori de la méthode appelée et de ses paramètres. Dans notre règle de traduction, nous ne distinguons pas entre les méthodes et les paramètres. Si, par contre, un tel niveau de détail est requis pour l'analyse des performances, une classe de couleur spécifique peut être définie constituée de sous-classes statiques triant l'ensemble des paires possibles (méthode, paramètres) dans des sous-ensembles disjoints. Ainsi, lorsque la paire (méthode, paramètres) est non pertinente pour un niveau de détail indiqué, nous omettons simplement cette classe de couleur.

La règle de traduction 3 définit le CC-SWN, qui étend la partie interface client d'un C-SWN afin de permettre la composition de SWNs et une analyse structurée ultérieure, sans modification de la sémantique du composant.

Règle de traduction 3 : CC-SWN pour une interface client
L'interface client d'un C-SWN est modifiée, donnant lieu à un CC-SWN, en ajoutant une place (et les arcs associés) comme post-condition de la transition de début de requête t_{BRS} et comme pré-condition de la transition de fin t_{ERS} (voir la partie à gauche de la figure 5.5). Le domaine de couleur de cette place est



(a) Modèle C-SWN (b) Modèle CC-SWN

Figure 5.6: Modèles C-SWN et CC-SWN du composant Executor

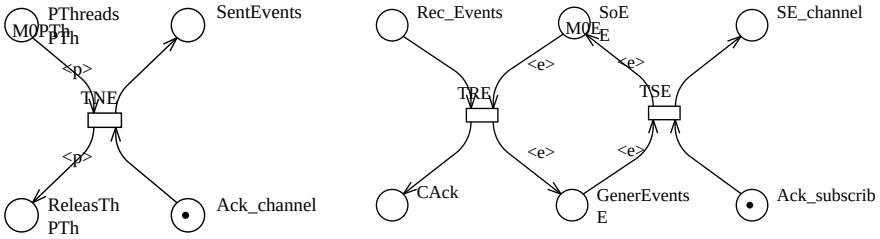
soit $CTh \times STh$ ou alors $CTh \times STh \times IDC$ lorsque plusieurs composants client, identifiés par la classe de couleur IDC, requièrent le même service (voir ci-dessous).

Exemple 5.34. *Nous illustrons la modélisation des interfaces client et serveur par la modélisation du composant Executor de l'exemple du système de gestion des informations de la bourse. Ce composant reçoit des requêtes du composant StockBroker 1 sur son interface serveur. Celle-ci est modélisée suivant la règle de traduction 1 par la partie à gauche de la figure 5.6(a). Deux classes de couleurs de base sont utilisées pour modéliser cette interface : la classe* Serv *représentant les threads serveurs du composant et* MP *modélisant les méthodes invoquées avec leurs paramètres.*
D'autre part, le composant Executor est muni d'une interface cliente à travers laquelle il invoque le service de persistance. Cette interface est modélisée suivant la règle de traduction 2 par la partie à droite de la figure 5.6(a). Nous obtenons ainsi le modèle C-SWN du composant Executor.
En appliquant la règle de traduction 3, nous modifions le C-SWN et nous obtenons le modèle CC-SWN de la figure 5.6(b).

Traitement de plusieurs clients Lorsqu'une interface serveur d'un composant est connecté à plusieurs interfaces client d'autres composants, le C-SWN du premier composant doit être modifié afin d'être composable avec plusieurs interfaces clientes, au sens de la composition de réseaux de Petri (fusion de places ou transitions). Ceci est réalisé en appliquant la règle de traduction 4. Le CC-SWN résultant garde la même sémantique que le C-SWN correspondant.

Règle de traduction 4 : Cas de plusieurs clients d'une interface serveur
L'interface serveur d'un C-SWN, ayant plusieurs clients connectés à l'interface, est modifiée (en regard au cas d'un seul client connecté) comme suit:
(i) Les transitions de début et de fin de service t_{BPS} et t_{EPS} sont dupliquées autant de fois que le nombre d'interfaces clientes;
(ii) Une classe de couleur IDC est introduite afin de distinguer entre plusieurs composants exposant une interface client.
L'interface résultante est illustrée sur la figure 5.5 (à droite). On obtient ainsi le modèle CC-SWN du composant serveur.

Remarque 3. *Nous notons que les règles de traduction proposées construisent des modèles d'interfaces*



(a) SWN d'une interface publisher (b) SWN d'un canal d'événements

Figure 5.7: Modèles SWN d'une interface publisher et d'un canal d'événements

de client/serveur dont l'interconnexion résulte en une composition synchrone à deux phases au sens de la méthode structurée, telle que donnée dans la définition 4.19.

5.5.4 Modélisation des interfaces basées événement

La communication basée événement est généralement réalisée suivant un pattern donné. Parmi les patterns les plus répandus dans les middlewares, le modèle *publish/subscribe*, utilisé dans la spécification CORBA [168], introduit en section 1.5.2.2.

Nous nous intéressons, dans le présent document, à la modélisation du modèle *publish/subscribe* fondé sur l'utilisation d'un canal d'événement, tel que décrit dans la spécification CORBA [168]. Dans cette spécification, le canal d'événement contrôle un type spécifique d'événements. Un publisher émet un événement d'un type spécifique au canal d'événement, puis continue son travail. Le canal reçoit la notification et accuse réception au publisher. Après cela, il notifie les subscribers intéressés. Ceux-ci reçoivent l'événement, accusent réception à leur tour au canal d'événements, puis déclenchent le gestionnaire approprié de l'événement (event handler).

Afin de modéliser ce schéma d'interaction, nous traduisons dans le contexte SWN l'interface publisher, le canal d'événements et le subscriber.

5.5.4.1 Traduction SWN de l'interface publisher

Nous traduisons une interface publisher par la règle de traduction 5.

Règle de traduction 5: Interface publisher
Une interface publisher d'un composant, identifié par un ensemble de couleurs PTh modélisant les threads éventuels du publisher, est modélisée par une transition *TNE* traduisant la notification des événements (voir figure 5.7(a)) ayant comme pré-conditions les places *Pthreads* et *Ack_channel*, et comme post-conditions les places *ReleasTh* et *SentEvents*.

Dans le modèle du publisher, les deux pré-conditions *Pthreads* et *Ack_channel* de la transition *TNE* traduisent respectivement les threads du publisher et l'état "prêt" du composant. Tandis que les post-conditions *ReleasTh* et *SentEvents* modélisent respectivement les threads du publisher reprenant leurs activités ainsi que les notifications d'événement. Les places *Ack_channel* et *SentEvents* sont non colorées puisque nous modélisons les publishers notifiant des événements de type spécifique.

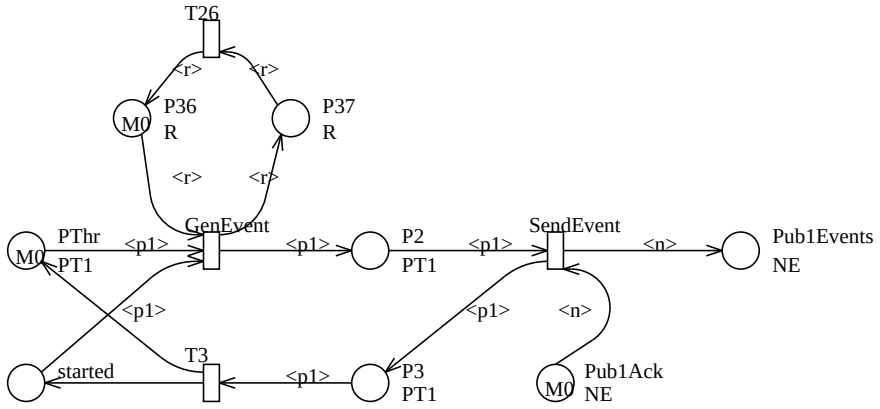


Figure 5.8: Modèle C-SWN du composant StockDistributor

Formellement, nous définissons le modèle d'une interface publisher comme suit:

Définition 5.3 (Interface publisher)**.** Une interface publisher I d'un C-SWN N est un tuple $< P_I, T_I >$ tel que:
• $P_I = \{PThreads, SentEvents, ReleasTh, Ack\_channel\}$, avec $\mathscr{D}(PThreads) = \mathscr{D}(ReleasTh) = PTh$, $\mathscr{D}(Ack\_channel)$ = $\mathscr{D}(SentEvents)$ =ε [1];
• $M_0(Ack\_channel) = 1$;
• $T_I = \{$TNE$\}$ with $\mathscr{D}$(TNE)=PTh;
• Pre(PThreads,TNE)=Post(ReleasTh,TNE) = Id ;
• Post(SentEvents,TNE)=Pre(*Ack_channel*,TNE) =1.

Exemple 5.35. *Nous illustrons la modélisation d'une interface source d'événement ou publisher avec le composant StockDistributor 1 du système de gestion des informations de stocks de bourse. Le StockDistributor est muni d'une interface unique étant une source d'événements. C'est un composant publisher de l'événement "changement d'une information de stock". Nous modélisons sa source d'événements suivant la règle de traduction 5. Le publisher est identifié par une classe de couleurs* PT1 *désignant les threads associés générant les événements. Pour modéliser les événements et leurs accusés de réception, une classe* NE *est utilisée. En modélisant tout le composant, nous obtenons le C-SWN de la figure 5.8.*

5.5.4.2 Traduction SWN du canal d'événements

Nous traduisons un canal d'événements par la règle de traduction 6.

Règle de traduction 6 : Canal d'événement
Un canal d'événement gérant un ensemble E d'événements d'un type spécifique est modélisé par deux transitions *TRE* et *TSE*, exprimant respectivement la réception des événements provenant des publishers et l'envoi de ces événements aux subscribers (figure 5.7(b)). La transition *TRE* est contrôlée par les places

[1] ε est la couleur neutre.

Rec_Events et *SoE* et possède comme post-conditions les places *CAck* et *GenerEvents*. La transition *TSE*, quant à elle, est contrôlée par les places *GenerEvents* et *Ack_subscrib* et a comme post-conditions les places *SoE* et *SE_channel*.

Dans le modèle du canal d'événement, les places *Rec_Events* et *SoE* modélisent respectivement les notifications reçues et l'ensemble des événements possible. La place *Rec_Events* est non-colorée. Quant à la place *SoE*, elle peut être colorée lorsque le canal d'événement gère plusieurs types d'événements. Dans notre cas, nous étudions des canaux d'événements supportant un type spécifique d'événements (selon la spécification CORBA), ce qui ne nécessite pas de colorer cette place.

D'autre part, les post-conditions de la transition *TE* (*CAck* et *GenerEvents*) modélisent respectivement les accusés de réception et les événements générés devant être retransmis aux subscribers. La place *CAck* est également non colorée, et *GenerEvents* a le même domaine de couleur que *SoE*. La place *Ack_subscrib* modélise l'état "prêt à diffuser les notifications" du canal. Elle est également non colorée comme la place *Ack_channel* du modèle du publisher. La place *SE_channel* modélise les événements envoyés aux subscribers via le canal.

Formellement, le modèle d'un canal d'événement est défini comme suit :

Définition 5.4 (Canal d'événement)**.** Un modèle de canal d'événement est un tuple $< P_I, T_I >$ tel que:
• $P_I = \{SoE, Rec\_Events, SE\_channel, GenerEvents\}$, avec $\mathscr{D}(SoE) = \mathscr{D}(GenerEvents) = E$,
$\mathscr{D}(Rec\_Events)$ = $\mathscr{D}(SE\_channel)$=$\mathscr{D}(CAck)$= $\mathscr{D}(Ack\_subscrib)$=ε ;
• $M_0(Ack\_subscrib) = 1$;
• $T_I = \{TRE, TSE\}$ avec $\mathscr{D}(TRE) = \mathscr{D}(TSE) = E$;
• Pre(SoE,TRE)= Post(GenerEvents,TSE)=
Pre(GenerEvents,TRE)= Post(SoE,TSE)= Id;
• Pre(*Rec_Events*,TRE)= Post(CAck,TRE)=
Pre(*Ack_subscrib*,TSE)= Post(*SE_channel*, TSE)= 1.

Exemple 5.36. *Reprenons l'exemple de l'application de gestion des informations de la bourse. Les composants StockDistributor, qui jouent le rôle de publishers, sont reliés aux deux StockBroker (subscribers). Un canal d'événements les relie. Ce canal gère un ensemble E d'événements. Il est donc modélisé en appliquant la règle de traduction 11. Nous obtenons le modèle de la figure 5.9.*

5.5.4.3 Traduction SWN de l'interface subscriber

La modélisation d'une interface subscriber requiert la connaissance de l'implantation du gestionnaire d'événement et le point de reprise après le traitement de l'événement.

Traitement des événements Différents schémas de traitement sont possibles pour implanter le gestionnaire (handler) d'un événement :

- Le subscriber déclenche une opération ou fonction interne à lui, implantant le handler associé à l'événement,
- ou bien il déclenche une requête de service à un autre composant qui peut être le publisher lui-même. Cette stratégie, connue sous le nom de mode *control-push data-pull*, est communément rencontrée lorsqu'un producteur de données, étant le publisher, publie un événement indiquant que certaines données ont été mises à jour et sont prêtes à être consommées. Pour récupérer les données, le subscriber invoque alors un service (méthode) offert par une interface serveur exposée par le composant publisher stockant les données, ou

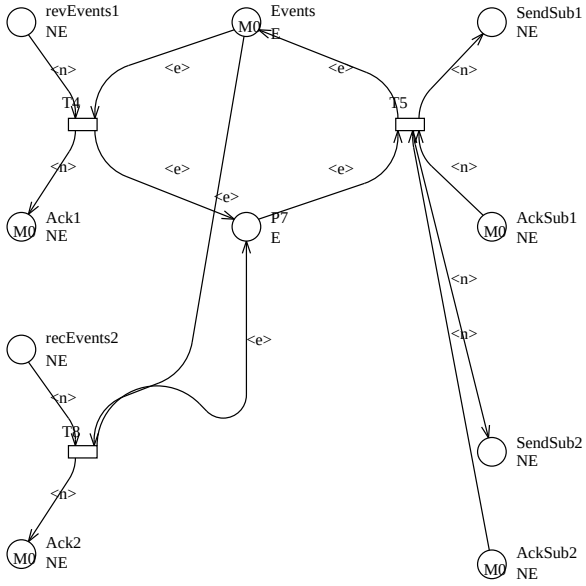


Figure 5.9: Modèle CC-SWN du canal d'événement du système de gestion des informations de la bourse

- le subscriber invoque un service offert par un troisième composant différent du publisher et du subscriber, suite à la réception de l'événement.

Mode de reprise après réception d'un événement Lorsqu'un événement est reçu, le subscriber suspend son exécution et déclenche le handler d'événement soit immédiatement après la notification, ou bien le handler peut être retardé après une période donnée. Par ailleurs, après le traitement d'un événement, le subscriber peut reprendre l'activité suspendue, ou sauter à un autre point de reprise diffèrent du point d'interruption. En conséquence, nous nous restreignons au cas du déclenchement immédiat du handler d'événement dès sa réception, mais nous modélisons deux modes de reprise : la reprise de l'activité interrompue du composant et le saut à un nouveau mode de reprise.

Étant donnés les deux points précédents, nous considérons pour notre modélisation les trois comportements possibles du subscriber adoptés lors de la réception d'un événement, avec chacun des deux modes de reprise après traitement.

Pour cela, nous présentons d'abord dans cette section la modélisation d'un subscriber qui traite localement les événements reçus, tout en tenant compte des deux possibilités de reprise. Les deux autres cas concernant le mode control-push data pull et traitement par un composant tierce sont exposés dans la section suivante.

Cas 1 : Traitement local de l'événement avec reprise de l'activité interrompue Une interface subscriber est traduite ici par la règle de traduction 7.

Règle de traduction 7 : Interface subscriber (1)
L'interface subscriber d'un composant, identifié par un ensemble de couleurs STh modélisant les threads éventuels du subscriber (voir figure 5.10 comme exemple), est modélisé par un SWN constitué de deux

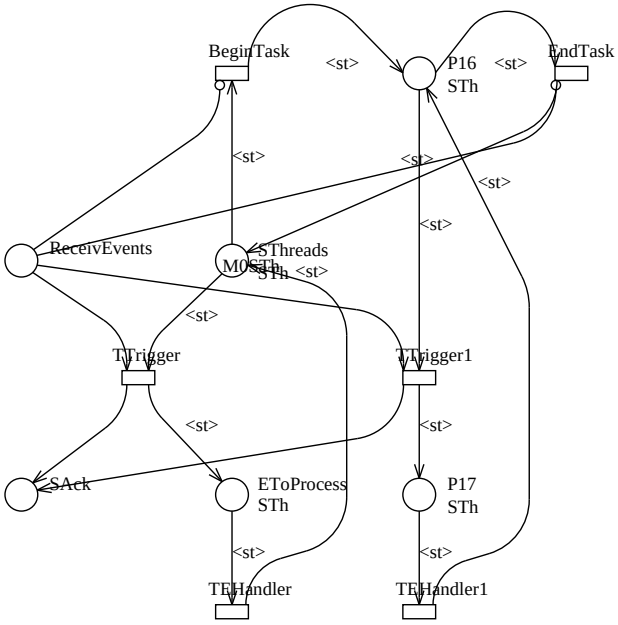


Figure 5.10: Modèle SWN d'une interface subscriber (cas 1)

parties:
(i) un traitement local modélisé par les transitions *ReceivEvents* et *SThreads*, contrôlées par les deux places *ReceivEvents* et *SThreads*, et
(ii) une partie de traitement des événements déclenchée par la transition *TTrigger* qui modélise la réception d'un événement. La transition *TTrigger* est aussi contrôlée par *ReceivEvents* et *SThreads*, et possède comme post-conditions les places *SAck* et *EToProcess*. Le gestionnaire d'événement (handler) est modélisé avec la transition *TEHandler*.

Dans le modèle du subscriber, le traitement local devrait être interrompu à la réception des événements. Pour cela, nous modélisons cette interruption par des arcs inhibiteurs qui interdisent le tir des transitions associées au traitement local (*BeginTask*, *EndTask* et éventuellement d'autres exprimant plus d'activités locales), aussitôt qu'un événement est reçu dans la place *ReceivEvents*. La place *SThreads* est colorée avec la classe de base STh, tandis que *ReceivEvents* a le même domaine que *SoE* (neutre ou ensemble de couleurs d'événement lors du traitement de plusieurs types d'événement).

La réception d'un événement induit, d'une part l'envoi d'un accusé de réception dans la place non colorée *SAck* par le tir de la transition *TTrigger* (ou *TTrigger1*), et d'autre part l'exécution d'un gestionnaire d'événement. Dans ce modèle, le traitement d'un événement est modélisé par la transition d'abstraction *TEHandler* (ou *TEHandler1*). Il est possible de raffiner cette transition par un sous-réseau détaillant le gestionnaire d'événements lorsque l'intérêt est porté sur l'influence des détails du traitement sur les performances du système.

Notons que, habituellement, le gestionnaire doit se déclencher après une "courte" période de temps, comparé aux activités du composant. Afin d'atteindre cet objectif, les vitesses de franchissements des transitions correspondantes doivent être choisies adéquatement à la situation. On peut spécifier par exemple des taux de franchissement de 1/0.001 des transitions *TTrigger, TTrigger1* par rapport aux tran-

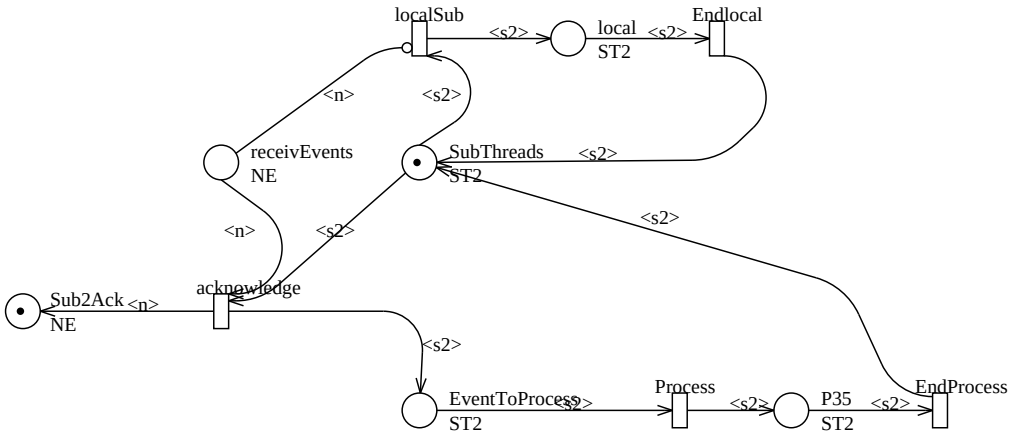


Figure 5.11: Modèle C-SWN du composant StockBroker 2

sitions *BeginTask, EndTask* et *TEHandler, TEHandler1*.

Exemple 5.37. *Nous illustrons la modélisation d'une interface puits d'événement ou subscriber avec le composant StockBroker 2. Celui-ci est muni d'une interface unique étant un puits d'événements. Les threads associés sont distingués par une classe de couleur* ST2*. Étant subscriber, nous modélisons son puits d'événements suivant la règle de traduction 7. En modélisant tout le composant, nous obtenons le C-SWN de la figure 5.11.*

Cas 2 : Traitement local de l'événement avec saut à un nouveau point de reprise Un raisonnement similaire est suivi pour dériver le modèle associé à une interface subscriber dans le cas de la réception d'événements causant un saut vers un nouveau point de reprise. Toutefois, une étape préliminaire est nécessaire avant la modélisation, afin de connaître les points de reprise du subscriber : À partir du comportement du composant subscriber, on récupère l'ensemble des couples (p_i, p_r) définissant les points interruptibles p_i avec leur points de reprise p_r.

Dans ce cas, nous établissons la règle de traduction 8 pour traduire une interface subscriber se comportant de la sorte.

Règle de traduction 8 : Interface subscriber (2)
En considérant l'ensemble J de couples des points interruptibles et des points de reprises $J = \{(pi_k, pr_k), k = 1..N\}$, l'interface subscriber d'un composant, identifié par un ensemble de couleurs STh modélisant les threads éventuels du subscriber est modélisé par le SWN de la figure 5.12. Ce SWN est constitué identiquement au cas 1 de deux parties : traitement local et traitement des événements. Cependant, la structure du SWN suit la contrainte suivante :
Pour chaque couple de transitions (*TTrigger*,*TEHandler*) tel que la place pi_k est pré-condition de *TTrigger*, alors la place pr_k correspondante est une post-condition de *TEHandler*.

Par exemple, dans la figure 5.12, trois couples (p_i,p_r) sont identifiés : $(STheads, p_2), (p_1, p_1), (p_2, STheads)$.

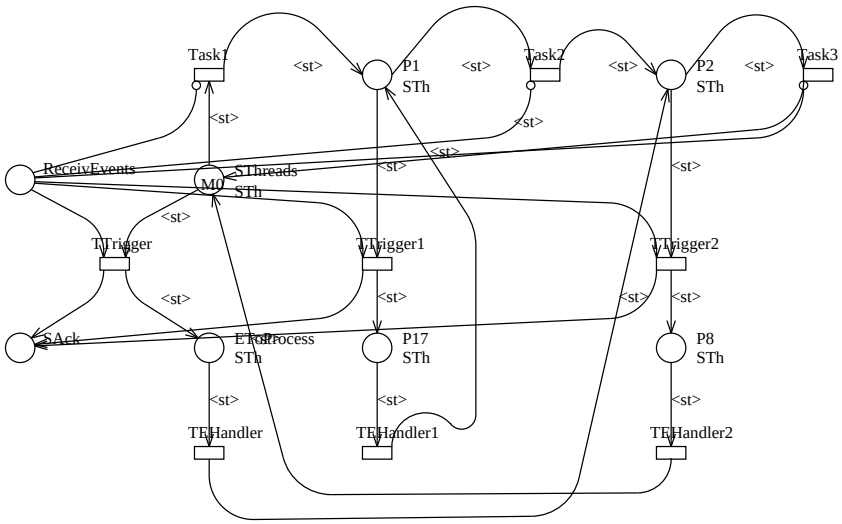


Figure 5.12: Modèle SWN d'une interface subscriber (cas 2)

Définition formelle de l'interface subscriber Formellement, le modèle SWN d'une interface subscriber construit pour les deux cas précédents sont définis comme suit :

Définition 5.5 (Interface subscriber)**.** Une interface subscriber I d'un C-SWN N est un tuple $< P_I, T_I >$ tel que:
• $P_I = \{SThreads, ReceivEvents, EToProcess, SAck\}$, avec $\mathscr{D}(SThreads) = \mathscr{D}(EToProcess) = STh$, $\mathscr{D}(SAck)$ =$\mathscr{D}(ReceivEvents)$ =ε;
• $T_I = \{TTrigger, TEHandler\}$ with $\mathscr{D}(TTrigger)$=$\mathscr{D}(TEHandler) = STh$;
• Pre(SThreads,TTrigger)=Post(EToProcess,TTrigger)=Id;
• Pre(ReceivEvents,TTrigger)= Post(SAck,TTrigger) =1.

5.5.4.4 Mode control-push data-pull

Dans ce cas, chacun des composants publisher et subscriber sont munis de deux interfaces: le publisher expose une source d'événement et une interface serveur, et le subscriber un puits d'événement et une interface client. La règle de traduction 9 explicite la modélisation correspondante. Dans un souci de clarté, nous faisons abstraction dans ce modèle des détails du traitement local du composant subscriber, en le représentant par une unique transition.

Règle de traduction 9 : Mode control-push data-pull Le SWN de la figure 5.13 modélise les composants publisher (à gauche), canal d'événement (au milieu) et subscriber (à droite) dans un mode control-push data-pull. Le publisher génère un ensemble E d'événements. Les données liées à un événement sont fournies au subscriber via son interface cliente, en utilisant un ensemble MP de méthodes munies de leurs paramètres.

Les modèles proposés dans cette règle de traduction restent identiques à ceux définis par les règles 5, 6 et 7 (ou 8), à l'exception de l'ajout d'une interface serveur au publisher et une interface cliente au

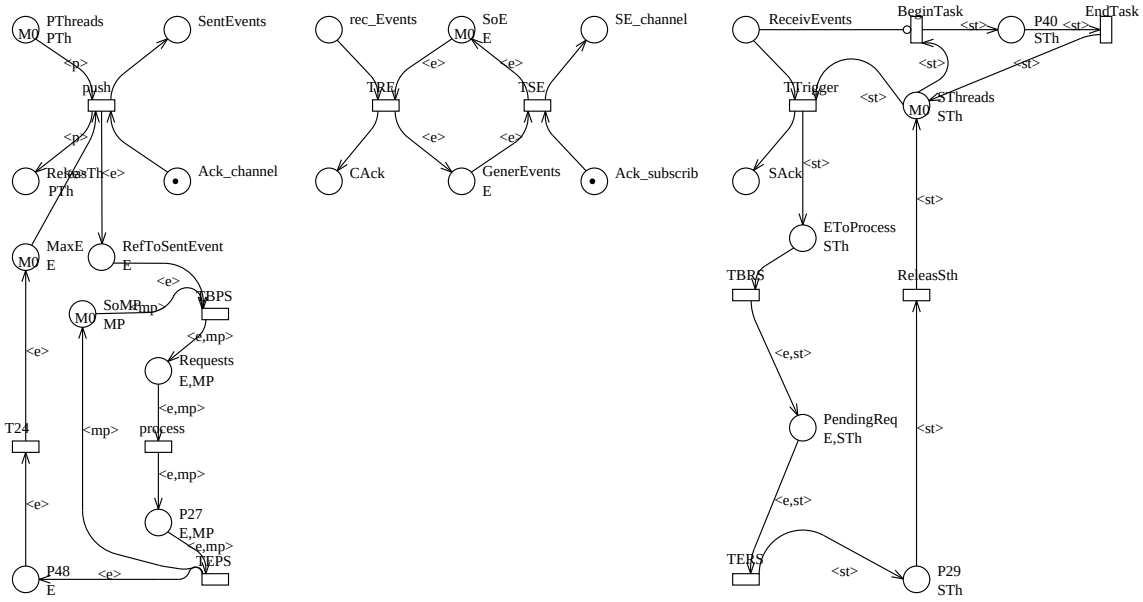


Figure 5.13: Modèles SWN dans le mode control-push data-pull

subscriber. Celui-ci réalise un traitement local jusqu'à réception d'une notification d'événement (reçue dans la place *ReceivEvents*). Dans ce cas, il envoie un accusé de réception pour cet événement à travers la transition *TTrigger*, puis invoque un service métier (transition *TBRP*) au composant publisher.

5.5.4.5 Traitement des événements par un composant tierce

La modélisation de ce cas est déduite de la précédente section, en distinguant la source d'événement (publisher) du composant offrant le service de récupération des données, ce qui implique des composants séparés.

Règle de traduction 10 : Traitement par composant tierce Le SWN de la figure 5.14 modélise les composants publisher, subscriber et le composant tierce. Le publisher génère un ensemble E d'événements. Le composant fournissant les données liées à un événement est identifié par une classe Serv modélisant les threads serveur. Il expose une interface serveur spécifiant un ensemble MP de méthodes métier avec leurs paramètres correspondants.

Ici également, les modèles proposés dans cette règle de traduction restent identiques à ceux définis par les règles 5, 6 et 7 (ou 8), à l'exception de l'ajout d'une interface cliente au subscriber et du modèle du composant tierce Le subscriber réalise un traitement local jusqu'à réception d'une notification d'événement (reçue dans la place *ReceivEvents*). Il envoie alors un accusé de réception pour cet événement à travers la transition *TTrigger*, puis invoque un service métier (transition *TBRP*) au composant tierce.

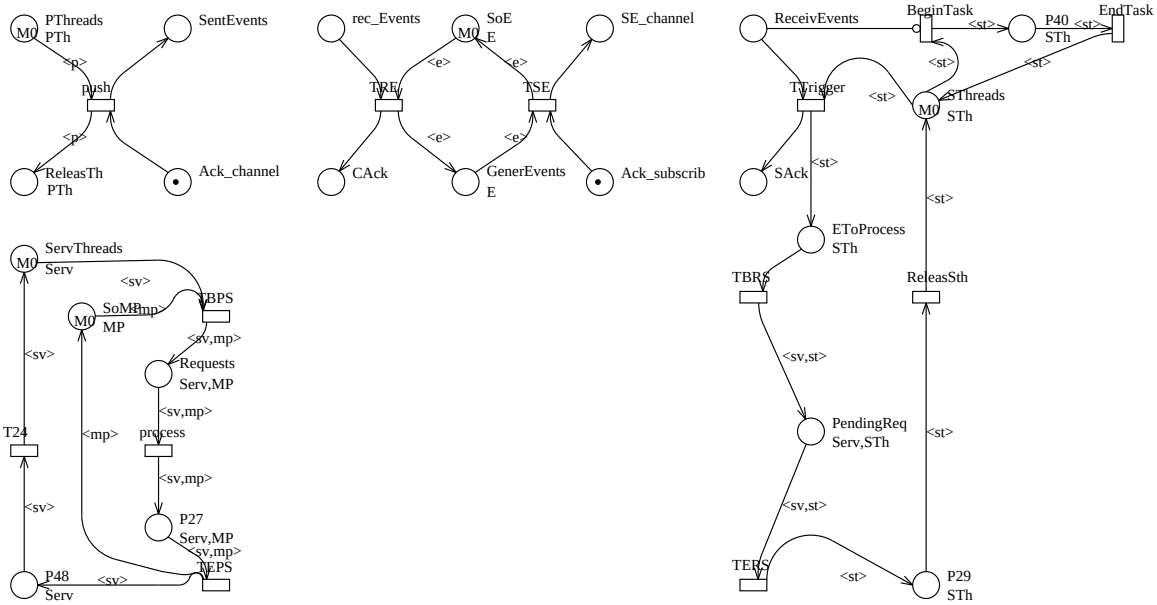


Figure 5.14: Modèles SWN dans le mode traitement par un composant tierce

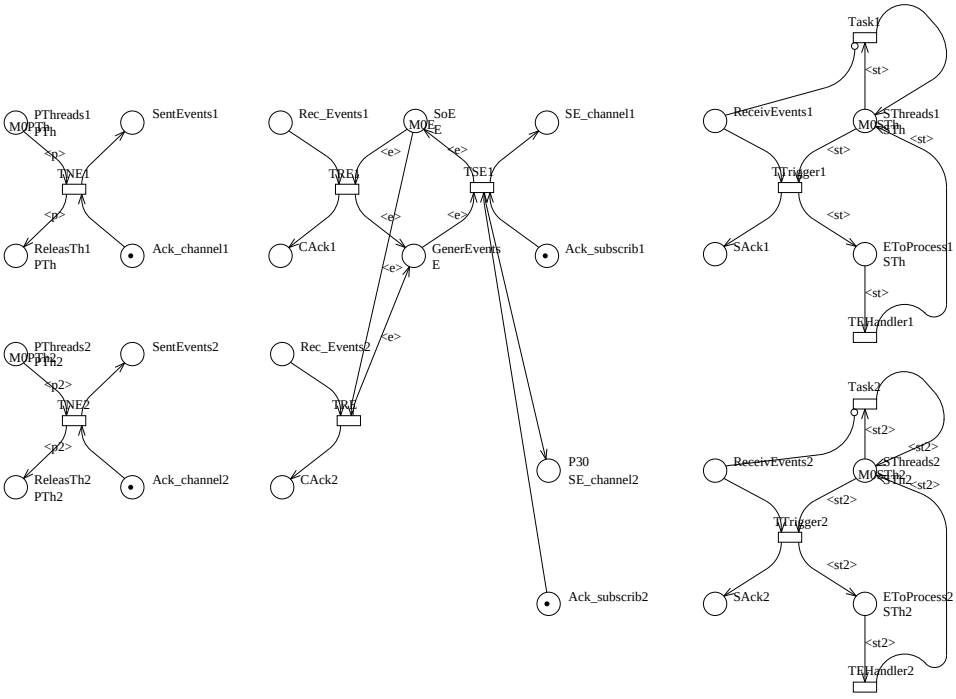


Figure 5.15: Modèles SWNs des interfaces d'événement avec plusieurs publishers et subscribers

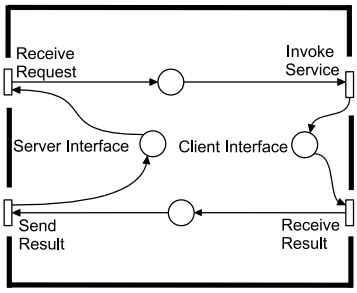


Figure 5.16: A very abstract CC-SWN of a component

5.5.4.6 Traitement de plusieurs publishers et subscribers

Une interface publisher d'un composant peut publier des événements vers plusieurs subscribers. De la même manière, une interface subscriber peut recevoir des événements à partir de différents publishers générant les mêmes type d'événements. C'est le rôle du canal d'événement d'assurer le médium de communication entre les publishers et les subscribers.

Dans ces cas, le modèle SWN (C-SWN) du canal d'événement doit être modifié afin d'être composable en même temps avec plusieurs modèles de publishers et de subscribers (au sens de la fusion de places et/ou de transitions). Cette modification est automatisée à l'aide de la règle de traduction 11. Notons que ceci n'influe aucunement sur la sémantique du système global.

Règle de traduction 11: Interfaces multiples d'événement
Le C-SWN d'un canal d'événement gérant plusieurs publishers (figure 5.15) est modifié par duplication de la transition *TRE* avec ses arcs et places associés (*Rec_Events, CAck*), autant que fois que le nombre de publishers.
En présence de multiples subscribers, les places *SE_channel, Ack_subscribe* sont dupliquées (avec leurs arcs associés), autant de fois qu'il y a de subscribers.

Remarque 4. *Nous notons que les règles de traduction proposées ici construisent des modèles d'interfaces dont l'interconnexion résulte en une composition asynchrone au sens de la méthode structurée, telle que donnée dans la définition 4.22.*

Jusqu'ici, nous avons présenté les différentes modélisations adoptées pour les interfaces d'un composant. Examinons maintenant comment exploiter ces modélisations pour construire les modèles associés aux composants primitifs, composites, puis une application basée composant.

5.5.5 Modélisation des composants primitifs

Dans cette section, nous générons un modèle SWN, en l'occurrence le C-SWN, pour chaque composant primitif.

Manifestement, il est possible d'abstraire la modélisation d'un composant primitif à cette étape-ci en choisissant un niveau approprié de détails souhaités :

- Au plus haut niveau d'abstraction, le contenu d'un composant primitif peut être modélisé par un SWN très simple tel celui de la figure 5.16. Ceci est possible lorsque l'intérêt du modélisateur ne se focalise pas sur les détails du composant, mais plutôt vise à calculer les performances à un niveau architectural de son application.

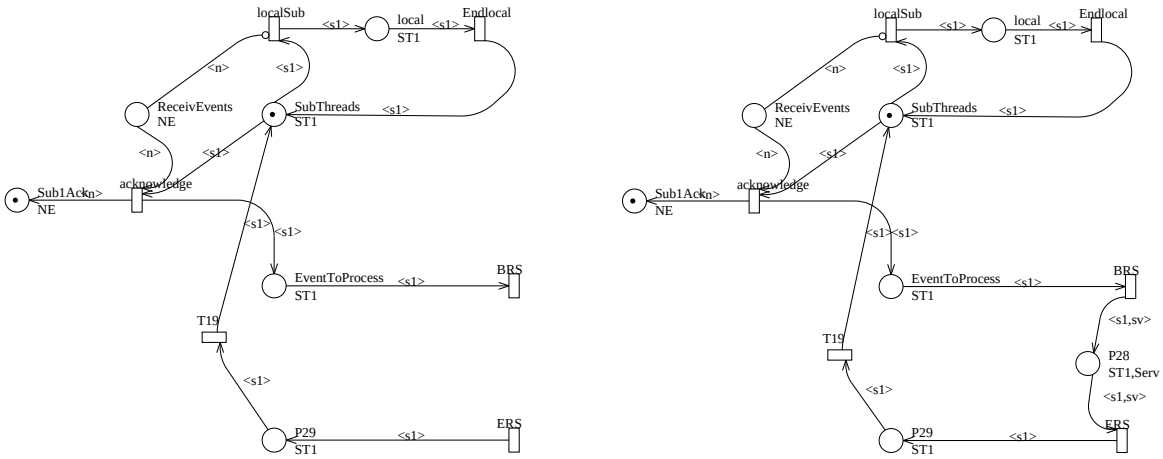
• À un niveau très fin de détails, le modélisateur représente toutes les activités internes au composant.

D'une manière générale, le C-SWN d'un composant primitif est construit comme suit :

```
ALGORITHME Construction du CC-SWN d'un composant primitif
BEGIN
 1. Analyser le code source du composant, fixer un niveau de détail
    pour la modélisation.
 2. Pour chaque ensemble de méthodes liées à une interface serveur :
    a. Modéliser l'interface serveur en utilisant la règle de
       traduction 1.
    b. Modéliser les activités internes à l'interface serveur,
       selon le niveau de détails internes requis .
 3. Pour chaque invocation de service, modéliser l'interface
    cliente en utilisant la règle de traduction 2.
 4. Compléter les modèles d'interfaces serveur et client
    éventuellement en utilisant les règles de traduction 3 et 4.
 5. Pour chaque source d'événements, traduire en SWN en utilisant
    la règle de traduction 5 (ou 9 dans le cas d'un mode control-push
    data-pull).
 6. Pour chaque puits d'événements :
    a. Traduire en SWN en utilisant les règles de traduction 7, 8, 9
       ou 10 selon le cas qui se présente.
    b. Modéliser les activités locales du subscriber, selon le niveau
       de détails internes requis.
    c. Modéliser les activités internes au gestionnaire d'événement
       selon le niveau de détails internes requis, dans le cas d'un
       traitement local au subscriber.
 7. Modéliser chaque canal d'événements (si plusieurs) suivant la
    règle de traduction 6.
 8. Compléter le modèle obtenu pour chaque canal d'événement en
    utilisant la règle de traduction 11, si plusieurs publishers
    et/ou subscribers interviennent.
 10. Si toute autre fonction (traitement) est appelée d'une manière
    interne à une interface (lors de l'appel d'une méthode de
    l'interface), modéliser les activités associées à la fonction.
END
```

Suite à cette étape, le CC-SWN obtenu doit être complété avec une temporisation stochastique. Pour cela, nous associons des taux de franchissement appropriés aux transitions du CC-SWN de la même façon que dans les modèles stochastiques. Ces taux peuvent être calculés à travers *une phase préalable d'estimation des paramètres du modèle*, où une application "test" (voir par exemple l'outil *Grinder*) est exécutée afin de mesurer les paramètres requis pour prédire les performances.

Remarque 5. *Notons que la modélisation d'interactions complexes peut se faire identiquement à la modélisation des composants primitifs pour la construction des modèles I-SWNs et CI-SWNs, qui n'est pas détaillée dans ce manuscrit.*



(a) Modèle C-SWN

(b) Modèle CC-SWN

Figure 5.17: Modèles C-SWN et CC-SWN du composant StockBroker 1

Exemple 5.38. *Nous illustrons la construction d'un modèle CC-SWN à travers la modélisation du composant StockBroker 1. Nous commençons par modéliser son interface puits d'événements en appliquant la règle de traduction 7. Les threads de ce composant sont modélisés par la couleur* ST1 *et les événements reçus par la classe* NE*. Nous construisons après le sous-modèle de l'interface cliente en utilisant la règle de traduction 2. Nous obtenons le C-SWN de la figure 5.17(a). Nous complétons après l'interface cliente an appliquant la règle de traduction 3. Le CC-SWN final est alors donné dans la figure 5.17(b).*

5.5.6 Modélisation des composants composites

Après la modélisation des composants primitifs d'une description d'architecture CBS, nous modélisons les composants (composites) de plus haut niveau. Un composant composite d'un niveau *i* est constitué d'un ensemble sous-composants interconnectés de niveau $i-1$, qui sont primitifs ou eux-mêmes composites. La construction du modèle SWN du composite nécessite la connexion des sous-composants CC-SWNs obtenuset la modélisation de ses interfaces externes. Cette procédure est itérée pour chaque niveau jusqu'à atteindre le plus haut niveau du composite.

À cette fin, nous appliquons l'algorithme suivant:

```
ALGORITHME Construction du CC-SWN d'un composant composite
BEGIN
Soit N le nombre de niveaux du composant composite.
1. Modéliser les composants primitifs du niveau 0.
2. Pour (i=1; i<N; i++)
   a. Pour chaque composite C du niveau i:
      (i) Assembler les composants du niveau i-1:
          - Pour chaque couple de sous-composants connectés par une
          invocation de service, fusionner les transitions
```

```
          correspondantes (TBRS,TBPS) et (TERS,TEPS).
          - Pour chaque couple de composants connectés par une
          interaction d'événement, fusionner les places du publisher
          et du canal d'événement (SentEvents,Rec_Events) et
          (Ack_channel,CAck), ainsi que les places du canal
          d'événement et du subscriber (SE_channel,ReceivEvents)
          et (Ack_subscriber,SAck).
      (ii) Chaque interface non connectée d'un sous-composant est
          considérée comme une interface externe de C.
   b. Modéliser les composants primitifs du niveau i.
END
```

La fusion de deux transitions (respectivement places) consiste à définir une transition (resp. place) unique et à associer les arcs des transitions (resp. places) fusionnées à cette transition (resp. place). Les classes de couleur des deux transitions sont mises en correspondence une à une pour des paramètres communs de l'interface (nom d'une méthode par exemple) et des classes de couleur spécifiques pour chaque transition (resp.place) sont gardées. Ainsi, le domaine de couleur de la transition résultant de la fusion est le produit cartésien des classes de couleur de la transition sans répétition, avec les classes de couleurs spécifiques de chaque transition. Tandis que les classes de couleur de deux places fusionnées sont mises en correspondance une à une, donnant ainsi le domaine de couleur de la place résultant de la fusion. Cette définition de la fusion est différente de l'approche proposée par [23] où plusieurs transitions peuvent être fusionnées, mais certains arcs peuvent être dupliqués pour plusieurs transitions fusionnées.

En assemblant les modèles SWN des sous-composants, des conflits de noms (d'une place, transition ou classe de couleur) peuvent apparaître entre les réseaux SWN. Ils sont résolus en les renommant. Ce renommage, comme il est de l'instanciation des classes de couleur, est nécessaire une fois l'analyse achevée, pour fournir les résultats (propriétés obtenues et indices de performance calculés) dans le contexte initial du CBS.

5.5.7 Modélisation des conteneurs

Un conteneur est composé d'un ensemble de composants interconnectés et un ensemble de services offerts aux composants. En plus des services qu'il offre, le conteneur joue le rôle de médiateur entre les composants qu'il contient et les composants d'autres conteneurs. Il permet d'acheminer les invocations des composants de et vers les composants externes appartenant à d'autres conteneurs.

Les invocations sont acheminées à travers deux formes d'objets associés avec chaque composant contenu :

- Soit une interface externe dite *interface de rappel (callback)* qui agit comme un intercepteur pour tous les appels entrants vers le composant, visible à partir de l'extérieur.
- Ou bien un intercepteur pour les appels sortants, interne au conteneur.

Le conteneur peut également fournir une interface locale afin de réduire le coût de communication entre les composants contenus.

Par conséquent, la construction du modèle SWN du conteneur requiert :
(i) la connexion des CC-SWNs des composants,
(ii) la modélisation des services, et
(iii) la modélisation du rôle de médiateur.

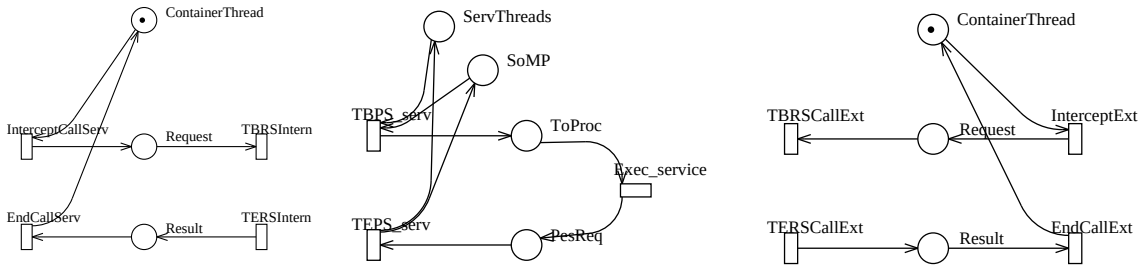


Figure 5.18: Modèles SWNs du routage d'une requête de service de conteneur (à gauche), d'un service de conteneur (au milieu) et d'une interface de rappel (à droite)

5.5.7.1 Connexion des CC-SWNs des composants

L'interconnexion des CC-SWNs se traduit par la fusion des places/transitions correspondant aux interfaces des composants communicants. Précisément, nous procédons comme suit, de manière identique à la fusion des sous-composants d'un composite :

- Fusion deux à deux des couples de transitions (TBRS,TBPS) et (TERS,TEPS) associées aux interfaces client/serveur de chaque couple de composants interagissant entre eux.
- Fusion deux à deux des couples de places (SentEvents, Rec_Events) et (Ack_channel, CAck) associées aux interfaces de type événement de chaque source publisher et de son canal d'événement, et des couples de places (SE_channel,ReceivEvents) et (Ack_subscriber, SAck) associées aux interfaces de type événement de chaque puits subscriber et du canal d'événement lui correspondant.

La même définition de la fusion que celle donnée dans la section 5.5.6 est appliquée ici.

5.5.7.2 Modélisation des services de conteneur

Cette modélisation requiert d'associer d'une part un modèle CC-SWN pour chaque service de conteneur, et d'autre part, un second modèle SWN exprimant le routage d'une requête d'un composant au service voulu. On suppose que le conteneur est monothreadé. La règle de traduction 12 décrit cette modélisation.

Règle de traduction 12 : Services de conteneur
• Un service de conteneur est modélisé par un composant abstrait (figure 5.18, au milieu) possédant une unique interface de service, identifiée par une classe de couleurs modélisant les threads serveurs (place *ServThreads*) et offrant un ensemble MP de méthodes (place *SoMP*). Le début et la fin du service sont modélisés par deux transitions *TBPS_serv* et *TEPS_serv*. Une transition unique *Exec_service* abstrait le service.
• Le routage d'une requête vers un service invoqué de conteneur est modélisé par le modèle de la figure 5.18, à gauche. Une place unique *ContainerThread* modélise l'unique thread du conteneur. La transition *InterceptCallServ* exprime l'interception d'une requête. Elle est contrôlée par la place *ContainerThread*. La transition *TBRSIntern* modélise la requête invoquée par le conteneur à son service. Le résultat est obtenu avec la transition *TERSIntern* et est envoyé vers le requêteur utilisant la transition *EndCallServ*.

Nous avons choisi d'abstraire l'activité induite par un service de conteneur, puisque notre objectif est

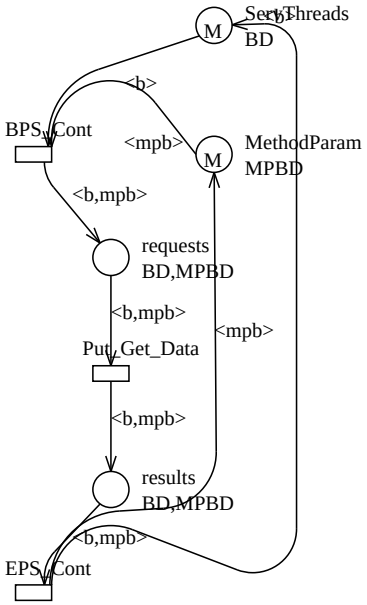


Figure 5.19: Modèle CC-SWN du service de persistance du conteneur

de considérer l'impact d'un service sur l'exécution d'un CBS.

Par ailleurs, le conteneur peut gérer ses instances de composants, les threads ou les ressources en utilisant la technique du pooling pour réduire la surcharge. Si le concepteur est intéressé par connaître l'impact de ce pooling sur les performances de son application, nous pouvons considérer cela soit en représentant dans le modèle plus de détails internes au conteneur, ou bien en associant un taux de franchissement conséquent à la transition *TBRSIntern* liée à l'opération invoquée.

Notons que, pour plus de clarté, les SWNs donnés en figure 5.18 ne sont pas colorés, mais ils doivent l'être.

Exemple 5.39. *Reprenons l'exemple du système de gestion des informations de la bourse. Le service de persistance est offert par le conteneur. Nous modélisons donc ce service suivant la règle de traduction 12, en faisant abstraction de ses détails. Nous représentons uniquement son interface serveur en suivant la règle de traduction 1. Cette interface utilise deux classes de couleurs de base :* BD *et* MPBD*, modélisant respectivement les threads du serveur de la base de données et les méthodes exposées. Nous obtenons ainsi le modèle C-SWN qui est lui-même le CC-SWN du service de conteneur de la figure 5.19, puisque aucune modification n'est nécessaire.*

5.5.7.3 Modélisation du rôle de médiation

Les appels sortants de composants sont acheminés par le conteneur. Ceci est modélisé, comme dans le cas d'une invocation de service de conteneur, en utilisant la règle de traduction 12. Les invocations externes aux services de composants situés dans un conteneur sont aussi interceptés par ce conteneur sur une interface externe et routée au composant concerné. Ceci est modélisé suivant la règle de traduction

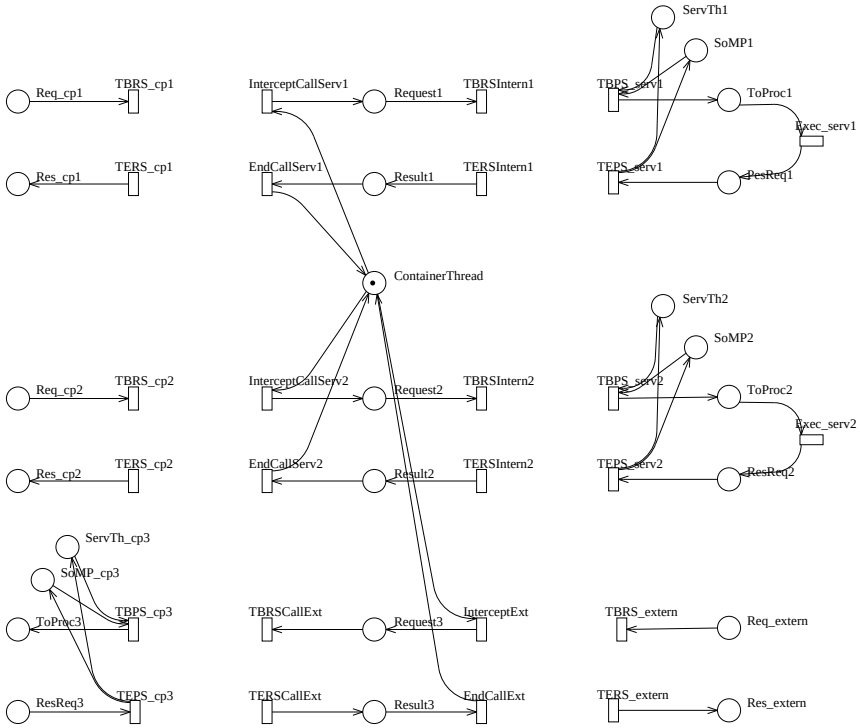


Figure 5.20: Exemple du CC-SWN d'un conteneur

13.

Règle de traduction 13 : Interfaces de rappel
Le routage d'une invocation externe à un service offert par un composant d'un conteneur est modélisé par le modèle de la figure 5.18, à droite. La transition *InterceptExt* modélise l'interception d'une requête. Elle est contrôlée par la place *ContainerThread*. La transition *TBRSCallExt* représente la soumission d'une requête au composant concerné. Le résultat est obtenu via la transition *TERSCallExt* et envoyé au requêteur en utilisant la transition *EndCallExt*.

5.5.7.4 CC-SWN d'un conteneur

La modélisation d'un conteneur aboutit à un CC-SWN dont les interfaces sont définies comme les interfaces callback et les interfaces non connectées associées à l'invocation interne des services externes.

```
ALGORITHME Construction du CC-SWN d'un conteneur
BEGIN
1. Modéliser chaque service de conteneur en utilisant la règle de
   traduction 12.
2. Connecter les composants communicants.
3. Pour chaque invocation d'un service de conteneur, construire une
   partie médiateur en utilisant la règle de traduction 12, la connecter
```

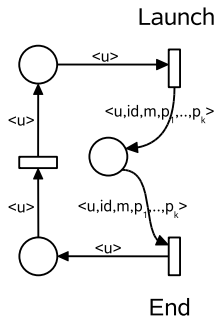


Figure 5.21: Réseau de Petri fermant une application

```
   au composant requêté sur le côté gauche et au modèle de service dans
   la partie droite .
4. Modéliser chaque interface callback en utilisant la règle de
   traduction 13 et la connecter au service de composant requêteur.
5. Pour chaque invocation interne d'un service offert par un composant
   externe, construire une partie médiateur suivant la règle de
   traduction 12 et la connecter au composant requêteur.
END
```

Exemple 5.40. *Nous pouvons illustrer la modélisation d'un conteneur par le SWN donné au milieu de la figure 5.20. Dans cette figure, deux composants cp*1 *and cp*2 *invoquent respectivement une requête au service1 et au service2 du conteneur. Un client externe demande également un service d'un composant cp3 interne au conteneur.*

5.5.8 Construction du G-SWN d'un CBS

Le G-SWN d'un système basé composant est construit en interconnectant les modèles CC-SWNs obtenus des "éléments" qu'il contient, ces éléments étant des composants, des canaux d'événements et éventuellement des conteneurs :

- Lorsque le CBS ne contient pas de conteneur, le modèle associé au composant composite de plus haut niveau représente lui-même le modèle G-SWN de notre application.
- En revanche, si le CBS fait appel à des conteneurs, sa modélisation requiert l'interconnexion des CC-SWNs obtenus des conteneurs. Ceci est réalisé en fusionnant les éléments d'interfaces des conteneurs, d'une manière similaire à l'interconnexion des modèles de composants.

Enfin, le modèle SWN global est complété en "fermant" les interfaces de l'application à l'aide d'un réseau de Petri "simple" muni d'un marquage initial approprié, afin de fournir un modèle G-SWN ayant un espace d'états *fini*. Ceci est une méthode classique, permettant de limiter le nombre d'entités dans le modèle. Un exemple d'un tel réseau de Petri est donné en figure 5.21.

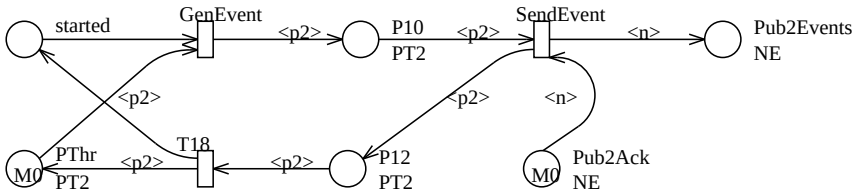


Figure 5.22: Modèle CC-SWN du composant StockDistributor 2

5.6 Application à l'exemple du système de gestion des informations de stocks de bourse

Pour obtenir le G-SWN de notre application présentée par la figure 5.3, nous construisons d'abord les modèles CC-SWNs des composants StockDistributor 1 et 2, StockBroker 1 et 2 et du composant Executor.

Nous avons déjà présenté le C-SWN du composant StockDistributor 1 et StockBroker 2 (figures 5.8 et 5.11). Comme aucune modification n'est nécessaire à ces modèles, chaque C-SWN est lui-même le CC-SWN Le second composant StockDistributor 2 est de manière similaire modélisé par le CC-SWN de la figure 5.22. Il utilise une classe de couleurs *PT2* représentant les threads associés. Nous avons également construit les CC-SWNs du composant StockBroker 1 (figure 5.17(b)), du composant Executor (figure 5.6(b)), du canal d'événement (figure 5.9) et du service de persistance (figure 5.19).

Nous modélisons ensuite le conteneur qui va router les requêtes du composant Executor au service de persistance. Nous connectons après tous les CC-SWNs des composants, du conteneur et du service de persistance, puis fermons le modèle global par un réseau de Petri fermant. Ainsi, nous obtenons le G-SWN de l'application complète (voir figure 5.23). Remarquons que certaines places et transitions ont été renommées, vu la similitude de noms entre les deux composants StockDistributor, entre les deux composants StockBroker et entre le composant Executor et le service de persistance, qui offrent chacun un service donné.

Remarque 6. *Nous notons que le modèle G-SWN est construit uniquement dans un souci de vérifier les conditions d'application de la méthode structurée de décomposition des SWNs. L'analyse des performances se fait sur les modèles SWN des composants et non sur le G-SWN.*

5.7 Conclusion

Dans ce chapitre, nous avons présenté les grandes lignes de notre méthode d'analyse de performances des CBS, après avoir posé les objectifs et principaux problèmes à étudier. Nous avons également détaillé la première phase de notre approche, consistant en la construction d'un modèle global SWN du système à analyser, à partir de la description d'architecture et des comportements des composants.

Le modèle SWN obtenu, ainsi que les modèles des composants seront exploités pour mener une analyse compositionnelle efficace des performances du système global. Cette analyse va reposer sur les travaux de décomposition synchrone et asynchrone de SWNs présentés dans le chapitre précédent.

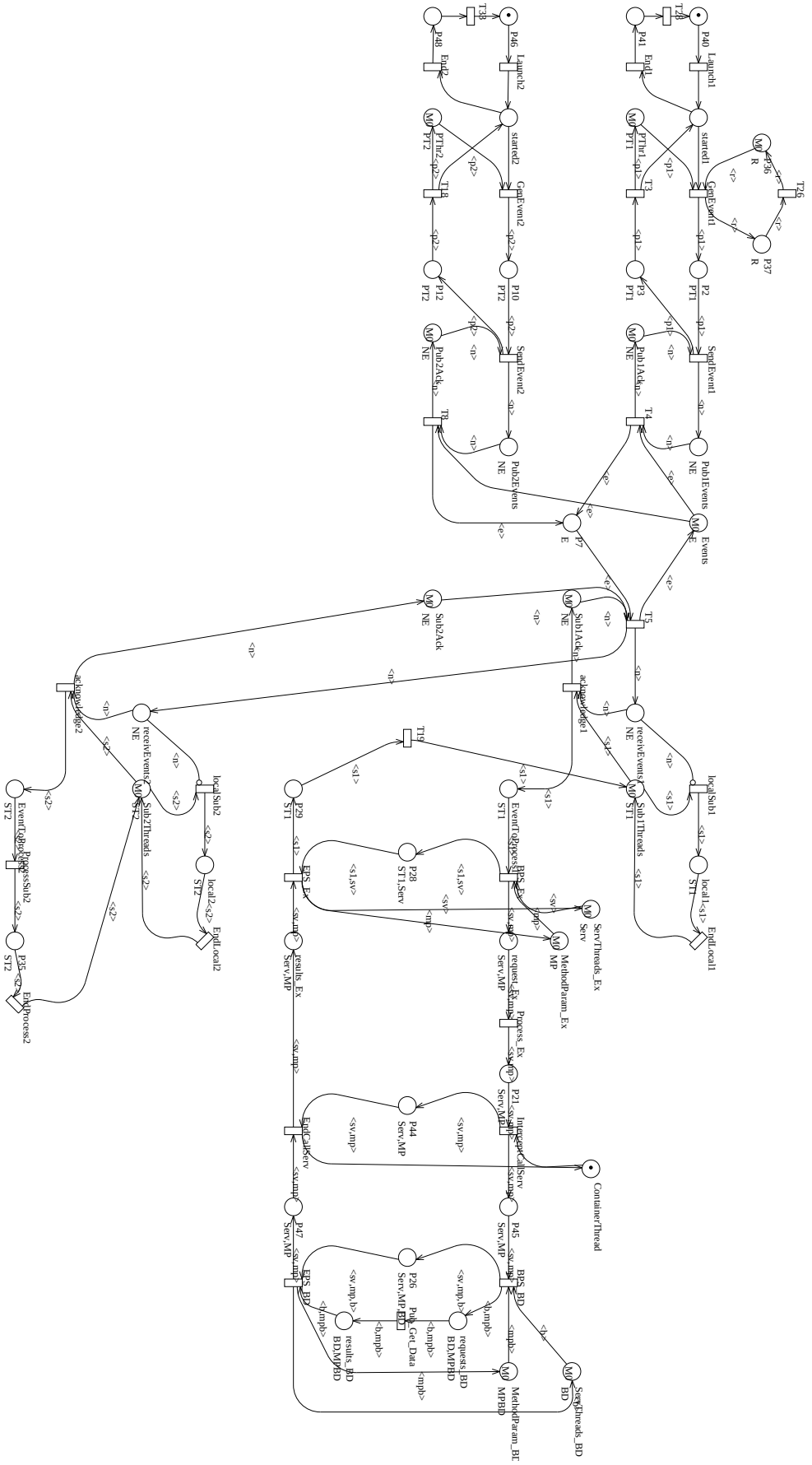


Figure 5.23: Modèle G-SWN du système de gestion des informations de la bourse

Néanmoins, dans le contexte d'un CBS, il est possible d'obtenir des compositions mixtes synchrone et asynchrone des modèles SWNs des composants, issus de la première phase de modélisation. Par conséquent, il est nécessaire d'étudier l'applicabilité des résultats de décomposition synchrone et asynchrone dans le cadre d'un CBS. Cette étude fait l'objet du chapitre suivant.

CHAPITRE 6

Analyse de performances des CBS

6.1 Introduction

L'analyse d'un CBS peut s'appuyer sur le SWN global résultant de l'assemblage des composants. C'est l'approche suivie dans [23] et implantée dans l'outil Algebra de l'ensemble logiciel GreatSPN [173]. Dans notre approche, nous proposons plutôt de tirer profit de la définition compositionnelle d'un CBS, pour mener une analyse efficace en temps de calcul et en occupation mémoire.

Dans ce but, nous adaptons au contexte d'un CBS la méthode d'analyse structurée [97; 98; 158] décrite en 4.4. Cette méthode a été décrite pour analyser soit une composition synchrone de SWNs, soit une composition asynchrone de SWNs. Cependant, un CBS peut comporter des composants interagissant en même temps par invocation synchrone de service et par notification asynchrone d'événements.

Dans le chapitre précédent, nous avons fait correspondre une communication par invocation de méthode entre deux composants à une composition synchrone des CC-SWNs correspondants d'une part, et d'autre part une communication par événements entre deux composants à une composition asynchrone de leurs CC-SWNs. Ainsi, il est fort possible d'obtenir un G-SWN du CBS à analyser, constitué de compositions mixtes synchrones et asynchrones de CC-SWNs. La question se pose alors de savoir si les résultats de la méthode structurée restent applicables pour des compositions mixtes de SWNs. En effet, les couleurs traversant les compositions mixtes risquent d'interférer, violant ainsi les conditions d'analyse précédemment énoncées.

En conséquence, nous étudions dans ce chapitre l'applicabilité des résultats d'analyse structurée pour l'évaluation des performances des CBS. Nous en dégageons les éléments nécessaires pour mener efficacement cette analyse.

6.2 Étude de la composition mixte synchrone et asynchrone de SWNs

6.2.1 Spécification du problème

Rappelons d'abord les conditions syntaxiques énoncées pour chacune des deux décompositions synchrone et asynchrone.

L'analyse d'une décomposition synchrone entre deux SWNs impose globalement trois conditions, énoncées informellement comme suit:

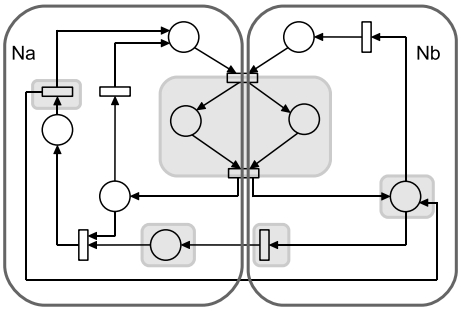


Figure 6.1: Interférence entre une composition synchrone et une composition asynchrone

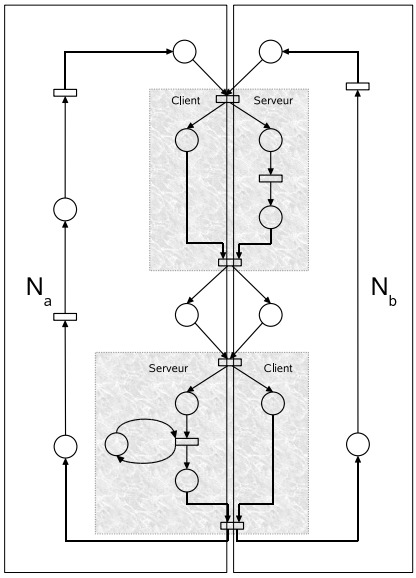


Figure 6.2: Interférence entre deux compositions synchrones de SWNs

- Durant la synchronisation, le réseau *client* ne doit pas interagir (d'une "façon colorée") avec le reste de son réseau, tandis que le réseau *serveur* peut éventuellement interagir avec le reste de son réseau.
- En dehors du sous-réseau délimité par la zone de synchronisation des deux SWNs, il ne devrait pas y avoir de synchronisation entre les classes de couleurs de base des deux sous-réseaux, en l'occurrence, les classes de couleur serveur et client.
- Les couleurs clientes se synchronisant ne sont pas connues par le serveur.

Par ailleurs, l'analyse d'une décomposition asynchrone entre deux ou plusieurs SWNs est possible si les trois conditions informelles suivantes sont satisfaites:

- Les couleurs globales doivent garder leur identité lors d'un transfert vers un autre réseau.
- Il ne peut y avoir ni création ni destruction d'entité mais uniquement des transferts d'entités entre réseaux.
- Les activités d'une couleur globale sont concentrées dans un seul réseau à un instant donné.

En analysant l'interférence de ces conditions, on remarque à priori que les conditions de la décom-

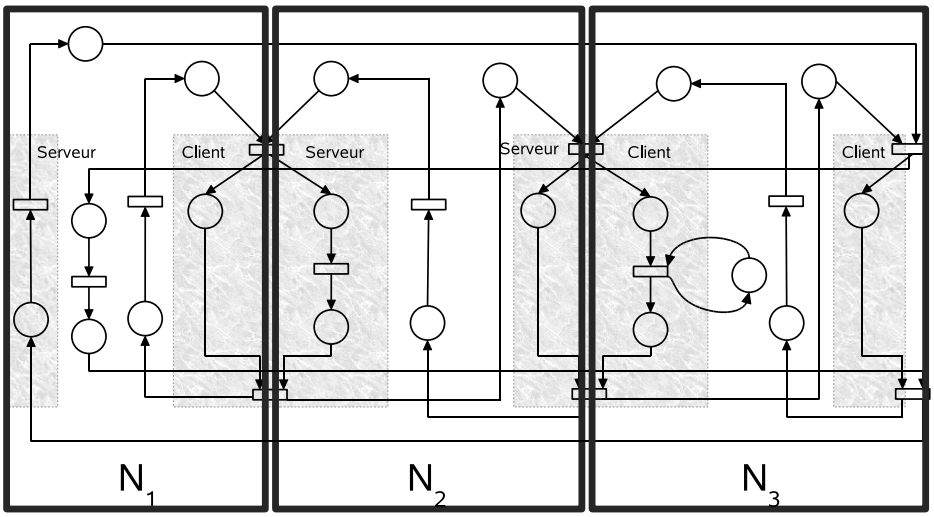


Figure 6.3: Chaine fermée de compositions synchrones de SWNs

position asynchrone ne risquent pas d'être altérées si dès le départ (avant sa composition avec la composition synchrone), elles sont remplies. Par contre, les deux dernières conditions de la décomposition synchrone sont susceptibles d'être violées dès qu'une composition asynchrone ou une autre synchrone est mise en jeu. Intuitivement, trois problèmes majeurs risquent de se poser, tels que le montrent les deux exemples suivants:

1. Le premier problème intervient lorsqu'une composition synchrone de SWNs se présente simultanément avec une composition asynchrone liant *directement* ou *indirectement* ces SWNs. Considérons, par exemple, un SWN simple constitué de deux réseaux N_a and N_b (figure 6.1, sans inscriptions techniques). Ces deux réseaux partagent une composition synchrone (rectangle gris) et une composition asynchrone (places et transitions grisées). Si une couleur synchronisée de N_a est également une couleur globale de la composition asynchrone, il est clair que « l'indépendance » requise entre les parties externes à la composition synchrone dans chaque réseau n'est plus satisfaite.
2. Un problème analogue risque de se poser lorsque deux SWNs se partagent plusieurs compositions synchrones indépendantes. En effet, l'analyse structurée exige que les couleurs clientes ne soient pas présentes dans le composant serveur. Prenons par exemple deux SWNs partageant deux compositions synchrones (rectangles gris sur la figure 6.2, sans inscriptions techniques) tels que les couleurs clientes de N_a sont synchronisés avec des couleurs serveurs de N_b, puis ces mêmes couleurs clientes deviennent serveurs dans une nouvelle composition synchrone avec N_b. Ceci risque de compromettre cette condition à cause de l'interférence des couleurs clientes et serveurs.
3. Un troisième problème peut survenir lorsqu'un ensemble de SWNs sont reliés deux à deux en composition synchrone dans une suite chronologique, et le dernier est également relié au premier en composition synchrone. Évidemment, la condition de « l'indépendance » souhaitée entre les couleurs synchronisées (en zone de synchronisation) et ceux se trouvant dans les parties externes à la composition synchrone dans chaque réseau n'est plus assurée. Un exemple de cette situation est donnée par la figure 6.3, où trois réseaux N_1, N_2 et N_3 sont en composition synchrone deux à deux: (N_1, N_2), (N_2, N_3) et (N_3, N_1). Remarquons que les couleurs serveur du réseau N_2 peuvent se propager à travers les compositions (N_2, N_3) et (N_3, N_1) pour aboutir à la partie locale de N_1, ce

qui peut causer problème.

Le principe de l'extension de l'analyse structurée est d'une part d'étendre les mécanismes de constitution de sous réseaux étendus aux cas où les deux compositions sont présentes à la fois. D'autre part, il est nécessaire de dégager les conditions pour lesquelles les tuples des marquages accessibles symboliques correspondant au produit synchronisé des graphes d'accessibilité symboliques (SRG) des sous-réseaux étendus définis sont une partition de l'ensemble des marquages accessibles du réseau SWN global.

Comme indiqué ci-dessus, l'analyse structurée d'un SWN comportant des compositions *mixtes* synchrones et asynchrones n'est possible qu'en imposant des conditions supplémentaires à ces compositions. Ces conditions s'appuient sur la notion de *multisynchronisation* que nous introduisons ci-dessous. Précisons d'abord la notion de composition mixte. Nous aurons besoin des notations suivantes:

Notations

- On note N_k Sync N'_k une composition synchrone de deux SWNs N_k et N'_k.
- On note Async $(N_{k_1},\ldots,N_{k_m})$ une composition asynchrone sur un ensemble de SWNs $(N_{k_i})_{i=1\ldots m}$. L'ensemble de ces SWNs composés d'une manière asynchrone est appelé *chaîne asynchrone*. Par abus de langage, on note Async (N_k) la chaîne asynchrone à laquelle appartient N_k.

Définition 6.1 (Composition mixte)**.** $\mathscr{N}$ est une composition mixte d'un ensemble de SWNs $(N_k)_{k\in K}$ ssi:

- $\exists k,k' \in K$, N_k Sync N'_k.
- $\exists J = \{k_1,\ldots,k_p\} \subset K$ tel que Async $(N_{k_1},\ldots,N_{k_p})$, k,k' pouvant appartenir ou non à J.

Nous pouvons maintenant définir la notion de sous-réseau étendu pour les compositions mixtes.

Définition 6.2 (Réseaux étendus dans une composition mixte)**.** Soit $\mathscr{N} = (N_k)_{k\in K}$ une composition mixte de SWNs satisfaisant les conditions du théorème 6.1.

Pour chaque SWN $N_k, k = 1,\cdots,K$, on définit un réseau étendu $\overline{N_k}$ construit comme suit:

- Initialement, $\overline{N_k} = N_k$.
- $\forall k' = 1,\cdots,K, k' \neq k$ tel que N_k Sync $N_{k'}$, ajouter à $\overline{N_k}$ soit la zone de synchronisation $N_{k,k'}$ restreinte à $N_{k'}$ si N_k joue le rôle de serveur dans la composition synchrone, soit une place implicite marquée par les couleurs de base serveur si N_k joue le rôle de client.
- Si N_k appartient à une chaîne asynchrone Async (N_k), alors pour tout $N_{k'}, k \neq k'$ appartenant à Async (N_k), ajouter à $\overline{N_k}$ la vue abstraite $\mathscr{A}(N_{k'})$.

Nous énonçons alors, après avoir donné quelques notations, la proposition suivante qui s'appuie sur un théorème que nous prouvons plus loin.

Notations

- RS est l'ensemble des marquages accessibles d'un SWN $\mathscr{N}$.
- $\overline{SRS}$ (respectivement $\overline{SRG}$) est l'ensemble des marquages accessibles symboliques (respectivement le graphe d'accessibilité symbolique) du réseau étendu de $\mathscr{N}$.
- $\overline{XSRS} = \prod_{i=1,..,k} \overline{SRS}$.
- $\overline{\mathscr{M}} = (\overline{\mathscr{M}_i})_{i=1,..,k}$ est un élèment de $\overline{XSRS}$.
- $\mathscr{D}(\overline{XSRS}) = \{M | \exists \overline{\mathscr{M}} \in \overline{XSRS}$ such that $\forall i = 1,..,k, \overline{M_i} \in \overline{\mathscr{M}_i}\}$.

Proposition 6.1. *Soit $\mathscr{N} = (N_k)_{k\in K}$ une composition mixte de SWNs satisfaisant les conditions du théorème 6.1. Alors:*

1. *Les tuples des marquages accessibles symboliques correspondant au produit synchronisé des graphes d'accessibilité symboliques (SRG) des réseaux étendus sont une partition de l'ensemble des marquages accessibles du réseau SWN global $\mathcal{N}$ (RS):*
   $RS \subseteq \mathcal{D}(\overline{XSRS})$.
2. *De plus, la chaîne de Markov correspondante est une agrégation exacte de la chaîne de Markov sous-jacente au réseau global $\mathcal{N}$.*

La preuve de cette proposition se ramène aux deux cas synchrone et asynchrone.

6.2.2 Composition multisynchronisée et non multisynchronisée

Le point clé pour prouver l'extension de l'analyse structurée aux compositions mixtes réside dans deux points essentiels:

- la relation entre les compositions asynchrones et les compositions synchrones (problème cité en 1).
- la relation entre plusieurs compositions synchrones (problème cité en 2 et 3).

Étant donné que seules les conditions de la composition synchrone risquent d'être altérées dans une composition mixte, formalisons-les d'abord afin de pouvoir démontrer certains résultats que nous énonçons ci-après.

Soit N_a le réseau client, N_b le réseau serveur et $N_{a,b}$ le sous-réseau commun de synchronisation, appelé *zone de synchronisation*.

- *Condition 1* : la première condition à risque impose que les synchronisations de couleurs du réseau client et du réseau serveur soient limités à la *zone de synchronisation*. Alors :
  Si une classe de couleurs C_i est dans le domaine de couleurs de $N_{a,b}$ et C_i est dans le domaine de couleurs de N_a, alors C_i n'est pas dans le domaine de couleurs de $N_b \backslash N_{a,b}$.
- *Condition 2* : la deuxième condition pouvant être compromise stipule que les couleurs clientes se synchronisant ne doivent pas être connues par le serveur; en d'autres termes : Si une classe de couleurs C_i est dans le domaine de couleurs de N_a (réseau client) et C_i est dans le domaine de couleurs de $N_{a,b}$, alors C_i n'est pas dans le domaine de couleurs de la partie de $N_{a,b}$ restreinte à $\mathcal{N}_b$.

Premièrement, nous étendons la relation Sync telle que définie dans la définition 4.19 aux chaînes asynchrones:

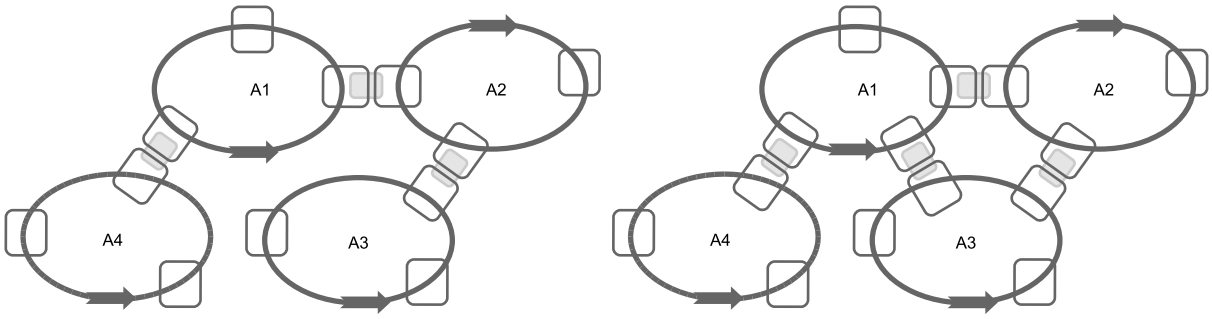
Définition 6.3 (Synchronisation de chaînes asynchrones)**.** Deux chaînes asynchrones U et V sont *synchronisées* (U Sync V) si et seulement s'il existe $N_u \in U$ et $N_v \in V$ tels que N_u Sync N_v.

Si la fermeture transitive de Sync sur les chaînes asynchrones a au moins deux classes d'équivalence, nous pouvons décomposer le système en au moins deux sous-systèmes totalement indépendants, et chaque sous-système peut être analysé séparément. Par conséquent, nous excluons ce cas dans la suite.

Dans le cas contraire, nous remarquons que lorsqu'une composition synchrone colorée existe entre deux SWNs, et que ces réseaux appartiennent à la même chaîne asynchrone directement ou indirectement, l'indépendance requise par la méthode d'analyse structurée n'est plus garantie, ce qui nous amène à définir la notion de *multisynchronisation*.

Définition 6.4 (SWNs multisynchronisés)**.** Deux SWNs N_a et N_b sont *multisynchronisés* (noté N_a MSync N_b) ssi:

1. N_a et N_b partagent une ou plusieurs compositions synchrones: N_a Sync N_b, et



(a) SWNs non multisynchronisés (b) SWNs multisynchronisés

Figure 6.4: Composition non multisynchronisée et multisynchronisée de SWNs

2. Async $(N_a) =$ Async (N_b), ou il existe une séquence $A_1,\ldots,A_n$ $(n > 1)$ de chaînes asynchrones telle que: Async $(N_a) = A_1$, Async $(N_b) = A_n$ et pour $1 \leq i < n$, A_i Sync A_{i+1}.

La figure 6.4 illustre cette définition. À gauche, les quatre chaînes asynchrones $A1, A2, A3$ et $A4$ sont représentées par des ellipses grises et les SWNs sont les petits rectangles. Une composition synchrone entre deux réseaux est représentée par un carré en gris clair. Aucune paire de SWNs n'est multisynchronisée. Au contraire, le schéma de droite montre plusieurs paires de SWNs multisynchronisés.

Par ailleurs, nous définissons également la notion de *chaîne synchrone fermée*.

Définition 6.5 (Chaîne synchrone fermée)**.** Soit $(N_k)_{k\in K}$ un ensemble de SWNs.
$(N_k)_{k\in K}$ est dit *chaîne de compositions synchrones* ssi: Pour chaque k, $1 \leq k < K$, N_k Sync N_{k+1}.

Elle est dite *chaîne fermée de compositions synchrones* si de plus et N_K Sync N_1.

6.2.2.1 Conditions d'application de l'analyse structurée d'une composition mixte de SWNs

À partir des définitions précédentes, on peut démontrer le résultat suivant.

Théorème 6.1. *Soit $\mathcal{N} = (N_k)_{k\in K}$ une composition mixte de SWNs. L'analyse structurée de N est possible lorsque:*

1. *$\forall\, a,b \in K$, il existe au plus une composition* Sync *, telle que N_a* Sync *N_b, et*
2. *Il n'existe pas de paire de SWNs multisynchronisés dans le système: $\forall\, a,b \in K, \neg(N_a$ MSync $N_b)$*
3. *Il n'existe pas de chaîne fermée de compositions synchrones.*

Preuve
Partons des conditions énoncées dans le théorème 6.1 pour vérifier si une interférence quelconque peut survenir entre les compositions synchrones et asynchrones.

On part de l'hypothèse que les conditions de composition synchrone et asynchrone sont déjà satisfaites entres les SWNs $(N_k)_{k\in K}$ deux à deux. Plusieurs cas peuvent se présenter. Considérons deux réseaux N_a et N_b, $a,b \in K$.

1. Soit $\neg(N_a$ Sync $N_b)$: Analyse possible évidente puisqu'aucune composition synchrone risquant d'être altérée n'existe entre N_a et N_b.

2. Il existe une composition Sync , telle que N_a Sync N_b.
   On sait que, $\forall\, k,k' \in K, \neg(N_k \text{ MSync } N'_k)$. Ceci implique que $\neg(N_a \text{ MSync } N_b)$, c.-à-d.:
   (i) Async $(N_a) \neq$ Async (N_b), et
   (ii) Il n'existe pas de séquence $A_1,\ldots,A_n$ $(n > 1)$ de chaînes asynchrones telle que: Async $(N_a) = A_1$, Async $(N_b) = A_n$ et pour $1 \leq i < n$,A_i Sync A_{i+1}.
   Vérifions donc si les conditions de la composition N_a Sync N_b sont satisfaites dans ce cas.
   - Si une classe de couleurs C_i est dans le domaine de couleurs de $N_{a,b}$, et C_i est dans le domaine de couleurs de N_a, alors vérifions si C_i peut être dans le domaine de couleurs de $N_b \backslash N_{a,b}$.
     La seule possibilité que C_i se retrouve dans le réseau N_b est qu'elle soit une couleur globale qui se propage à travers les réseaux. Si C_i est globale, elle peut se propager dans la chaîne Async (N_a) et dans toute séquence $A_1,\ldots,A_n$ $(n > 1)$ de chaînes asynchrones liées à Async (N_a) par une composition synchrone.
     Comme Async $(N_a) \neq$ Async (N_b) et il n'existe pas une séquence de chaînes asynchrones liées en même temps à Async (N_a) et à Async (N_b), C_i ne sera jamais dans le domaine de couleur de $N_b \backslash N_{a,b}$.
   - La condition 2 est vérifiée intrinsèquement à la composition N_a Sync N_b.
3. De même, on suppose qu'il existe une composition Sync , telle que N_a Sync N_b.
   On fait l'hypothèse qu'il n'existe pas une chaîne fermée de compositions synchrones, c.-à-d. que si l'ensemble $(N_k)_{k\in K}$ constitue une chaîne fermée de compositions synchrones, ceci signifie que: $\exists j$, $1 \leq j < K$ tel que $\neg(N_j \text{ Sync } N_{j+1})$. Vérifions donc si les conditions de la composition N_a Sync N_b sont satisfaites dans ce cas.
   - Si une classe de couleurs C_i est dans le domaine de couleurs de $N_{a,b}$ et C_i est dans le domaine de couleurs de N_a, alors vérifions si C_i peut être dans le domaine de couleurs de $N_b \backslash N_{a,b}$.
     La seule possibilité que C_i se retrouve dans le réseau N_b est qu'elle soit acheminée de réseau en réseau jusqu'à atteindre le premier réseau. Or, la chaîne synchrone n'est pas fermée, c.-à-d. $\exists j$, $1 \leq j < K$ tel que $\neg(N_j \text{ Sync } N_{j+1})$. D'où, C_i ne peut être dans le domaine de couleurs de $N_b \backslash N_{a,b}$.
   - La condition 2 est vérifiée intrinsèquement à la composition N_a Sync N_b.

Ceci peut se généraliser à tous les réseaux N_a et N_b, $a,b \in K$, c.q.f.d.

Si les conditions d'analyse d'une composition mixte sont satisfaites, on procède à l'extension des SWNs pour appliquer les résultats de l'analyse structurée (voir ci-dessus).

6.2.3 Analyse de compositions mixtes multisynchronisées

Lorsqu'une composition mixte de SWNs est multisynchronisée, on ne peut appliquer les résultats précédents. Dans ce cas, on peut suivre deux approches pour analyser le système global.

1. La première approche est de fusionner les SWNs dont la composition ne répond pas aux conditions énoncées en un seul SWN de taille plus importante. Ceci peut se répéter jusqu'à obtenir une famille de SWNs satisfaisant les conditions. Manifestement, plus la fusion des SWNs est large, moins on bénéficiera de l'analyse structurée proposée. Dans le pire des cas, on obtient le réseau global sans aucune analyse compositionnelle. Ce cas reflète des systèmes fortement couplés.
2. La deuxième approche consiste à modifier ou compléter les SWNs avec des places et des transitions de telle manière à satisfaire les conditions de l'analyse structurée. En effet, compléter les réseaux correspond à l'introduction de *réseaux de connexion* comme dans l'approche OsMoSys [215]

pour la *spécification basée composant* de systèmes. Les applications possibles de cette approche sont très dépendantes du système modélisé. Néanmoins, ceci implique un accroissement de la complexité du modèle résultant et un risque de modification de la sémantique d'interaction entre composants. Enfin, il semble difficile d'établir des critères de décision d'application d'une telle modification pour des ensembles de configurations suffisamment généraux.

Nous étudions maintenant comment les résultats d'analyse de composition mixte de SWNs peuvent être appliqués à l'analyse des performances des CBS.

6.3 Conditions suffisantes d'analyse structurée d'un CBS

Afin d'analyser un système basé composants, nous nous proposons de partir de la configuration initiale de SWNs consistant en l'ensemble des modèles CC-SWNs et CI-SWNs obtenus durant la première phase de notre méthode. Nous recherchons une configuration de SWNs qui satisfait les conditions d'analyse structurée, puis nous analysons les performances d'une telle configuration.

Dans cette section, nous étudions l'applicabilité des conditions de l'analyse structurée pour l'analyse d'un CBS. Nous montrons d'abord que les modélisations proposées des deux types d'interface (invocation de service et communication par événements) aboutissent à des compositions de SWNs satisfaisant les conditions de composition synchrone et asynchrone. Puis, nous reportons les conditions que nous avons énoncées pour l'analyse qu'une composition mixte (théorème 6.1) dans le contexte d'interactions de composants.

6.3.1 Satisfaction des conditions dans le cas d'une invocation de service

Comme souligné dans le chapitre précédent, la modélisation proposée pour un schéma d'invocation de service entre deux composants a été construite en respectant la structure d'une composition synchrone de SWNs. De plus, le modèle CC-SWN du composant doté d'une interface client joue le rôle de « client » au sens de la composition synchrone à deux phases (voi définition 4.19), et le modèle CC-SWN du composant doté d'une interface serveur joue le rôle de « serveur ».

Ceci nous incite à démontrer le résultat suivant:

Théorème 6.2. *Soit N_1 et N_2 deux CC-SWNs associés respectivement à deux composants CB_1 et CB_2 liés par une invocation de service, tel que CB_1 expose une interface client et CB_2 une interface serveur.*

Alors, l'interconnexion des CC-SWNs N_1 et N_2 est une composition synchrone satisfaisant les conditions syntaxiques données dans la définition 4.21.

Preuve

Déterminons d'abord, à partir de la figure 6.5, les ensembles des classes de couleur participant dans le réseau client N_1 et le réseau serveur N_2, notés respectivement $\mathscr{D}(N_1)$ et $\mathscr{D}(N_2)$:

- $\mathscr{D}(N_1) = \{CTh, C_{11}, \ldots, C_{1p}\}$ où CTh représente la classe des couleurs threads du composant client, et $C_{11}, \ldots, C_{1p}$ sont les classes de couleur éventuelles internes au composant client, mais n'apparaissant pas lors de la synchronisation.
- $\mathscr{D}(N_2) = \{STh, MP, R, C_{21}, \ldots, C_{2q}\}$ où STh représente la classe des couleurs threads du composant serveur, MP la classe des méthodes avec leurs paramètres, R la classe des ressources utilisées éventuelles pendant le traitement associé au service, et $C_{21}, \ldots, C_{2q}$ sont les classes de couleur éventuelles internes au composant serveur, mais n'apparaissant pas lors de la synchronisation.

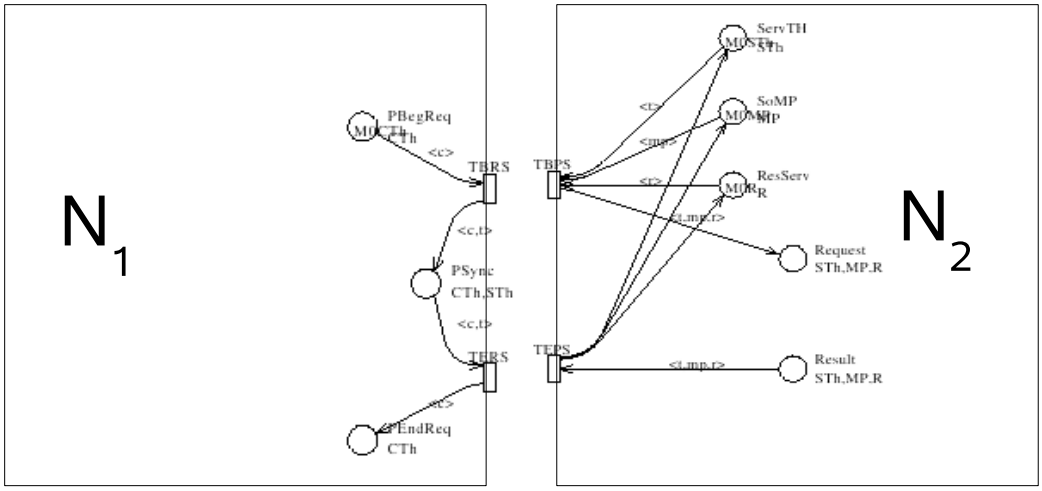


Figure 6.5: Composition associée à une invocation de service

Vérifions les conditions énoncées dans la définition 4.21.

1. La première condition (voir item 1 de la définition 4.21) impose que pour toute classe de couleurs C_i étant dans le domaine de couleurs de $N_{1,2}$ et dans celui de $\overline{N}_1 \backslash N_{1,2}$, il existe un semi-flot contenant toutes les places colorées p se trouvant en dehors de la zone de synchronisation (c.à.d. $p \in \overline{P}_1 \backslash P_{1,2}$) et toutes les places p' de la zone de synchronisation ($p' \in P_{k,k'}$). Comme seule la couleur CTh est dans la zone de synchronisation du client, alors: il suffit au modélisateur de construire le CC-SWN (partie interne) de CB_1 tel qu'il existe un semi-flot f dans N_1 sur la classe CTh, où $f_c = X_c.PBegReq + X_c.PSync + X_c.PEndReq + X_c.P_{11} + \ldots + X_c.P_{1_{m_1}}$, $P_{11}, \ldots, P_{1_{m_1}}$ étant des places internes à N_1.
2. La deuxième condition (voir item 2 de la définition 4.21) est identique pour les couleurs serveurs synchronisées, qui sont regroupées dans la classe STh dans notre cas. Elle consiste à trouver un semi-flot dans la zone de synchronisation $N_{1,2}$ sur la classe STh. En rajoutant une place implicite IP_s modélisant les couleurs STh lors de la construction du réseau étendu $\overline{N}_1$, nous avons effectivement le semi-flot $f_s = X_s.PSync + X_s.IP_s$.
3. La troisième condition (voir item 3 de la définition 4.21) limite les interactions entre activités locales et en synchronisation, en imposant uniquement des interactions avec des activités locales non colorées. Dans notre cas, seule la transition résultat de la fusion des transitions *TERS* et *TEPS* appartient à l'ensemble des transitions de continuation/sortie de synchronisation. Cette transition n'a pas de place pré-condition appartenant à la zone locale de N_1. Donc, cette condition ne se pose pas.
4. La dernière condition (voir item 4 de la définition 4.21) spécifie le principe d'engagement d'une synchronisation client-serveur:
   (a) Les clients se synchronisant ne sont pas connus dans le serveur, ce qui est vérifié puisqu'en regardant l'unique transition en entrée de la composition (transition résultat de la fusion des transitions *TBRS* et *TBPS* est en entrée), elle n'a aucune place précondition ou postcondition dans le réseau serveur N_2 contenant les couleurs clientes CTh.
   (b) Les serveurs qui s'engagent sont présents dans la place implicite rajoutée au réseau étendu de N_1, par construction de la place implicite.

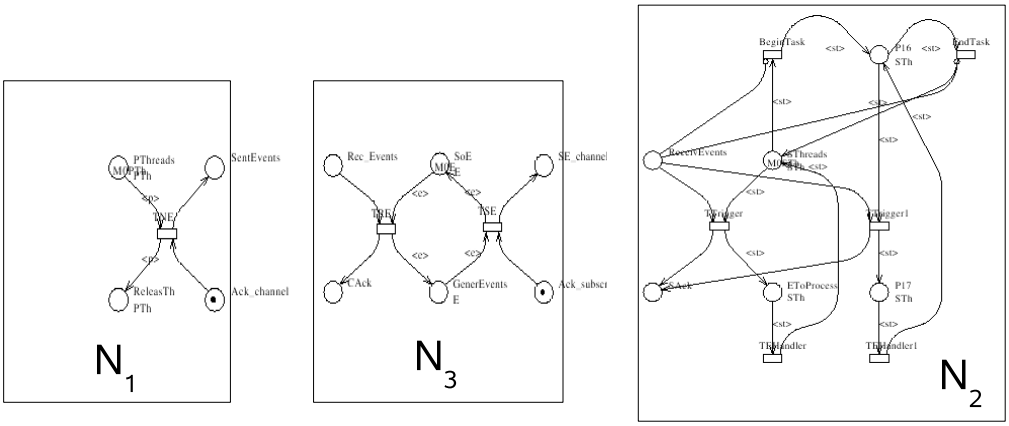


Figure 6.6: Composition associée à une composition par événement

(c) Les couleurs provenant du réseau serveur N_2 peuvent conditionner l'engagement à la synchronisation. Ici, on vérifie si les couleurs appartenant au domaine de N_2, participant à la synchronisation et qui ne sont pas dans la classe serveur (STh) présente dans la place implicite, ne se retrouvent pas dans une place de N_1 précondition ou postcondition de la transition d'entrée. Dans notre cas, les classes *MP* et *R* participent à la synchronisation, mais ne se retrouvent pas dans la partie de synchronisation client, ce qui est demandé.

Ce qui montre que les conditions de composition synchrone structurée sont remplies pour une composition issue d'une invocation de service.

6.3.2 Satisfaction des conditions dans le cas d'une interaction par événements

Nous avons vu dans le chapitre précédent, la modélisation proposée pour un schéma de communication par événements entre deux composants a été construite en respectant la structure d'une composition asynchrone de SWNs. Ceci nous incite à démontrer le résultat suivant:

Théorème 6.3. *Soit N_1 et N_2 deux CC-SWNs associés respectivement à deux composants CB_1 et CB_2 liés par une communication par événements, tel que CB_1 expose une interface source d'événements et CB_2 une interface puits d'événements.*

Alors, l'interconnexion des CC-SWNs N_1 et N_2 est une composition asynchrone satisfaisant les conditions syntaxiques données dans la définition 4.26.

Preuve

Déterminons d'abord, à partir de la figure 6.6, les caractéristiques de la composition des CC-SWNs qui met en jeu un troisième CC-SWN, N_3 correspondant au canal d'événement.

La composition obtenue est constituée d'une chaîne asynchrone de 3 réseaux N_1, N_2 et N_3. Notons d'abord que, comme le réseau associé au canal d'événement est très petit, il n'est pas intéressant de le considérer en isolation pour l'analyse. En effet, lorsque les réseaux d'une composition sont très petits, le temps de calcul peut augmenter par rapport au temps nécessaire pour l'analyse du réseau global. Ce problème de granularité est discuté dans le chapitre 7.

Pour cette raison, nous préférons analyser le réseau du composant source d'événement (publisher) augmenté du modèle associé au canal d'événement. Soit N'_1 ce nouveau réseau.

Dans la composition de N'_1 et N_2, une seule couleur globale est utilisée. Cette couleur est soit neutre, comme dans le schéma donné en figure 6.6, soit colorée par une classe de couleurs désignant l'ensemble des événements.

Dans le cas où la couleur globale est neutre, il n y a pas de conditions à vérifier; la composition asynchrone est analysable par la méthode structurée.

Dans le cas où la classe de couleur associée aux événements est utilisée comme couleur globale, il est nécessaire de vérifier les conditions énoncées par la définition 4.26. Soit C cette classe globale.

1. La première condition (voir item 1 dans la définition 4.26) impose que les entités globales gardent leur identité lors d'un transfert vers un autre réseau. Pour vérifier cela, le semi-flot d'abstraction f_C sur la couleur globale C défini pour chaque réseau N_k doit contenir toutes les places en amont des transitions de sortie du réseau N_k ($p \in {}^\bullet TO_k$):
   - N'_1 a comme transition de sortie *TSE*, et le semi-flot d'abstraction associé à C est $f_{C_1} = X_c.Ack\_subscr + X_c.PA_2$, PA_2 étant la place d'abstraction de N_2 dans $\overline{\mathcal{N}'}_1$.
   - N_2 a comme transition de sortie *TTrigger*, et le semi-flot d'abstraction associé à C est $f_{C_2} = X_c.ReceivEvents + X_c.PA_1$, PA_1 étant la place d'abstraction de N'_1 dans $\overline{\mathcal{N}}_2$

   La condition est donc vérifiée : f_{C_1} et f_{C_2} contiennent tous les deux les places en amont de la transition de sortie associée.
2. La seconde condition (voir item 2 dans la définition 4.26) souligne le fait qu'il ne peut y avoir ni création ni destruction d'entité mais uniquement des transferts d'entités entre sous-réseaux. On la vérifie en regardant si les fonctions d'arcs en entrée des transitions de sortie contiennent les variables correspondant à la classe globale C. D'après la modélisation adoptée, les fonctions d'arcs $Pre(Ack\_subscr, TSE)$ et $Pre(ReceivEvents, TTrigger)$ contiennent bien X_c.
3. La dernière condition (voir item 3 dans la définition 4.26) énonce que les activités d'une couleur globale sont concentrées dans un seul réseau à un instant donné. Pour cela, on cherche les semi-flots associés à la couleur globale C dans chaque réseau. Ces semi-flots ont été donnés dans 1.

Ce qui démontre que la composition de SWNs issue d'une communication par événements vérifie les conditions de composition asynchrone structurée.

6.3.3 Énoncé des conditions d'analyse d'un CBS

Les interactions par invocation de service et par événements entre deux composants génèrent respectivement une composition synchrone et une composition asynchrone des modèles SWNs associés aux composants. Dans le contexte d'un CBS, plusieurs connexions par invocation de service et par événements interagissent au sein du même modèle global. Ce qui mène à une composition mixte de SWNs. Ceci implique une interférence éventuelle des compositions synchrones et asynchrones. De ce fait, il est indispensable de dégager l'ensemble des conditions que doit remplir un CBS afin d'autoriser une analyse structurée compositionnelle. Ces conditions sont une conséquence des conditions énoncées dans le théorème 6.1 pour une composition mixte, reportées dans la sémantique des interactions entre composants. On énonce alors le théorème suivant, déduit du théorème 6.1: On notera que nous proposons uniquement un ensemble de *conditions suffisantes* pour développer une analyse structurée. Il nous semble en effet impossible de dégager des conditions nécessaires et suffisantes pour une telle analyse, sauf à se restreindre à des CBS très spécifiques.

Théorème 6.4. *Soit $\mathscr{N} = (\mathscr{N}_k)_{1\leq k\leq K}$ l'ensemble des modèles SWNs associés aux composants d'un CBS et $\mathscr{N}$ le modèle G-SWN correspondant. Alors:*

$\mathscr{N}$ est analysable par la méthode d'analyse structurée si:

1. *Si $(\mathscr{N}_1,\mathscr{N}_2)$ et $(\mathscr{N}_1,\mathscr{N}_3)$ (resp. $(\mathscr{N}_2,\mathscr{N}_3)$) sont en interaction client/serveur (par invocation de service), alors $(\mathscr{N}_2,\mathscr{N}_3)$ (resp. $(\mathscr{N}_1,\mathscr{N}_3)$) ne sont pas en relation client/serveur.*
2. *Si $(\mathscr{N}_1,\mathscr{N}_2)$ sont en communication par événement publish/subscribe, alors s'ils sont en interaction client/server, alors la classe de couleurs des événements ne participe pas dans l'interaction client/serveur.*
3. *Si $(\mathscr{N}_1,\mathscr{N}_2)$ et $(\mathscr{N}_1,\mathscr{N}_3)$ sont en communication par événement publish/subscribe, alors si $(\mathscr{N}_2,\mathscr{N}_3)$ sont en interaction client/serveur, alors la classe de couleurs des événements ne participe pas dans l'interaction $(\mathscr{N}_2,\mathscr{N}_3)$.*
4. *$\forall k' \in K$ tel que N_k Sync $N_{k'}$, si une classe de couleurs C_i est dans le domaine de couleur d'un réseau N_k, et dans le domaine de couleur de $N_{k,k'}$, alors il existe au plus une classe $C_{i'}$ appartenant au domaine de couleur de $N_{k'}$ telle que C_i est synchronisée avec $C_{i'}$, c.-à-d.:*
   $\forall p \in P_{k,k'}$, si C_i est dans le domaine de p, alors:
   - *soit aucune classe du domaine de couleur de $N_{k'}$ n'est dans le domaine de p,*
   - *soit uniquement $C_{i'}$ est dans le domaine de couleur p avec C_i.*

La première condition est déduite du fait que l'existence d'une chaîne fermée de compositions synchrones générées par un sous-ensemble de SWNs $N_{a1},\ldots,N_{al}$ synchronisés deux à deux risque de violer les conditions d'analyse structurée tel qu'énoncé par le théorème 6.1.

La seconde condition stipule que si deux composants (réseaux) sont en même temps en interaction client/serveur et en interaction par événement, alors les couleurs synchronisées dans l'interaction client/serveur ne doivent pas être globales dans l'autre interaction. Comme la couleur globale dans l'interaction par événements est elle-même la classe des couleurs modélisant les événements, celle-ci ne doit pas être synchronisée dans la composition synchrone. Ceci risque d'enfreindre la condition 1 ci-dessus de la composition synchrone qui limite la présence des couleurs synchronisées au seul sous-réseau de synchronisation.

De même que la deuxième condition, la troisième assure que les couleurs synchronisées dans la composition synchrone ne se propagent pas en dehors de la zone de synchronisation. Pour cela, la classe des couleurs événements ne doit pas participer à la composition synchrone.

La dernière condition impose qu'une entité d'un réseau donné, pouvant être synchronisée, doit être soit non synchronisée, soit synchronisée avec uniquement une seule entité d'un autre sous réseau.

Afin de mener une analyse structurée compositionnelle, il est nécessaire de trouver une décomposition du CBS en SWNs satisfaisant les conditions énoncées, cette décomposition du CBS *n'étant pas nécessairement la configuration correspondant à l'architecture initiale* de l'application basée composants. Ceci est développé dans la section suivante.

6.4 Recherche d'une décomposition du CBS compatible avec l'analyse structurée

Les conditions d'analyse structurée sont vérifiées sur l'ensemble des modèles SWNs (CC-SWNs, CI-SWNs) obtenus durant la phase de modélisation du CBS à analyser. Nénamoins, il est possible que

ces conditions ne soient pas satisfaites sur certains sous-ensembles de SWNs.

Nous proposons donc de rechercher une configuration de SWNs qui satisfasse ces conditions.

6.4.1 Principe

L'idée clé que nous adoptons pour retrouver une décomposition de SWNs « compatible » à l'analyse structurée est de partir de la configuration initiale des CC-SWNs et CI-SWNs, vue comme un ensemble unique de sous-réseaux $SWN_1, \ldots, SWN_K$ et vérifier les conditions énoncées pour chaque SWN ses interactions avec les SWN adjacents. Si une des conditions n'est pas satisfaite, nous regroupons le sous-ensemble J des SWN_k dont la composition compromet cette condition et on forme un nouveau SWN plus large qui remplacera le sous-ensemble J. On itère ainsi ces vérifications jusqu'à épuiser toutes les compositions faisant partie du CBS. L'algorithme suivant résume ces étapes.

6.4.2 Algorithme

ALGORITHME Recherche d'une décomposition structurée de SWNs
BEGIN

1. *Configuration initiale*: ensemble des CC-SWNs et CI-SWNs, notés (N_k), $k = 1 \ldots, K$.
   CurrentConf = (N_k), $k = 1 \ldots, K$.
2. FinalConfig = $\emptyset$
3. Pour chaque SWN $N_k \in CurrentConf$
   (a) Pour chaque SWN $N_{k'}$ adjacent à N_k
      (i) Vérifier l'ensemble C des conditions énoncées dans le théorème 6.4 sur la composition de $(N_k, N_{k'})$ dans le G-SWN.
      (ii) Si l'ensemble des conditions de C sont satisfaites, passer au SWN adjacent suivant.
      aller à (i)
      Sinon NewN= Fusion(N_k,$N_{k'}$)
         CurrentConf = CurrentConf - $\{N_{k'}, N_{k'}\}$ + NewN
         aller à 3)
   (b) Si l'ensemble des conditions de C sont satisfaites pour tous les SWNs $N_{k'}$:
      FinalConf = FinalConf + $\{N_k\}$
      CurrentConf= CurrentConf – $\{N_k\}$, aller à 3)
4. Condition d'arrêt: CurrentConf = $\emptyset$

END

Exemple 6.41.
Soit le SWN de la figure 6.7 modélisant un système embarqué constitué de 9 SWNs composés comme suit :

- *Les SWNs N_1, N_2, N_3, N_4, N_5, N_8 et N_9 forment une chaîne asynchrone avec comme couleur globale, la classe de base Sg.*
- *Les SWNs N_6 et N_7 sont en composition asynchrone avec Rq comme couleur globale.*

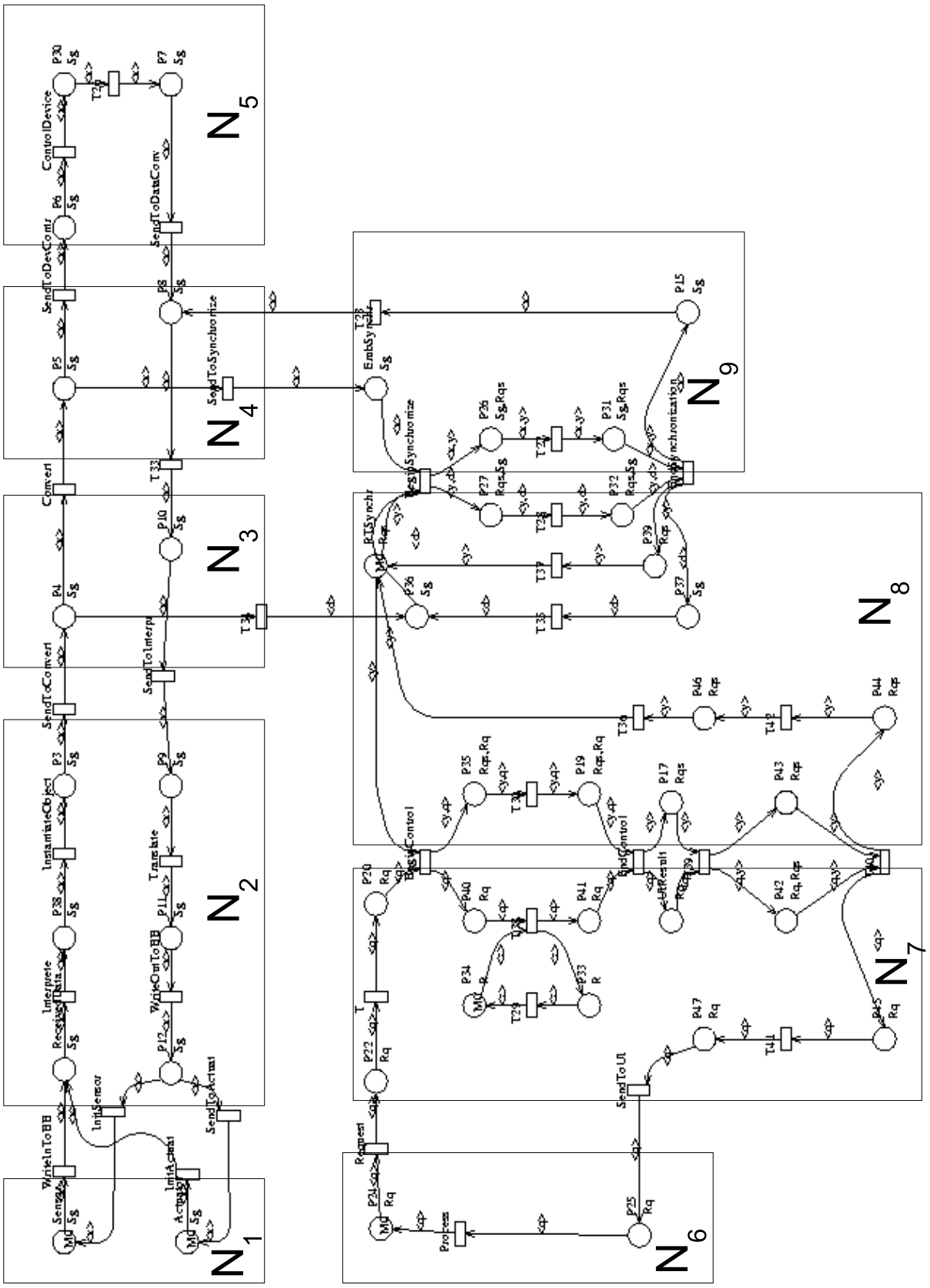


Figure 6.7: Exemple de composition multisynchronisée de SWNs

- *Les SWNs N_7 et N_8 sont reliés par deux compositions synchrones, tels que N_7 joue le rôle de serveur et N_8 de client dans la première composition et N_7 joue le rôle de client et N_8 de serveur dans la deuxième composition.*
- *Enfin, les SWNs N_8 et N_9 sont en composition synchrone également avec N_8 jouant le rôle de serveur et N_9 de client.*

Pour rechercher une décomposition compatible, nous vérifions les conditions entre chaque deux SWNs adjacents (voisins), en commençant par N_1. La composition asynchrone deux à deux entre N_1, N_2 et N_3 satisfait les conditions. Toutefois, en vérifiant la composition entre N_3, N_4, N_8 et N_9, nous remarquons qu'il s'agit bien d'une composition multisynchronisée. En conséquence, nous regroupons les 4 réseaux en un seul SWN noté N_{3489}. Nous vérifions ce nouveau SWN avec ses voisins.
Comme deux compositions synchrones existaient entre N_7 et N_8, ces deux compositions sont translatées entre le nouveau SWN N_{3489} et N_7 (puisque N_{3489} contient N_8). Ces deux compositions violent les conditions d'application de la méthode structurée. Nous les regroupons donc en un nouveau SWN plus important en taille, noté N_{34789}.
Par contre, les conditions de composition asynchrone entre N_6 et N_7 sont satisfaites, ainsi que pour la composition asynchrone entre N_4 et N_5, qui est maintenant translatée entre N_{34789} et N_5.
En dernier, nous obtenons une décomposition compatible à l'analyse structurée constituée de la configuration de SWNs N_1, N_2, N_5, N_6 et N_{34789}, (voir la figure 6.8).

Nous avons présenté tous les éléments clés nécessaires à l'analyse des performances d'un CBS. Nous pouvons maintenant résumer les étapes d'analyse de notre méthode dans l'algorithme donné ci-après.

6.5 Algorithme général d'analyse structurée d'un CBS

La méthode proposée est résumée dans l'algorithme suivant.

```
ALGORITHME Analyse structurée d'un CBS
BEGIN
Soit S un CBS à analyser.
1. Déterminer à partir de la description d'architecture du CBS S
   les composants et leurs interfaces.
2. Modéliser ces composants en suivant les étapes décrites dans
   la section Modélisation des composants primitifs.
3. Déterminer les interactions entre paires de composants, en
   déduire le type de chaque interaction.
4. Modéliser éventuellement les connecteurs SWN si besoin est.
5. Vérifier s'il y a des conflits de noms entre composants et les
   renommer lorsque c'est nécessaire.
6. Construire le G-SWN en connectant les composants.
7. Rechercher une décomposition structurée de SWNs satisfaisant
   les conditions énoncées de l'analyse structurée, en appliquant
   l'algorithme de recherche d'une décomposition compatible.
8. Construire l'extension de chaque composant SWN Nk suivant la
   définition donnée ci-avant.
9. Construire les SRGs des réseaux étendus.
10.Calculer le produit "synchronisé" des SRGs, obtenant ainsi
```

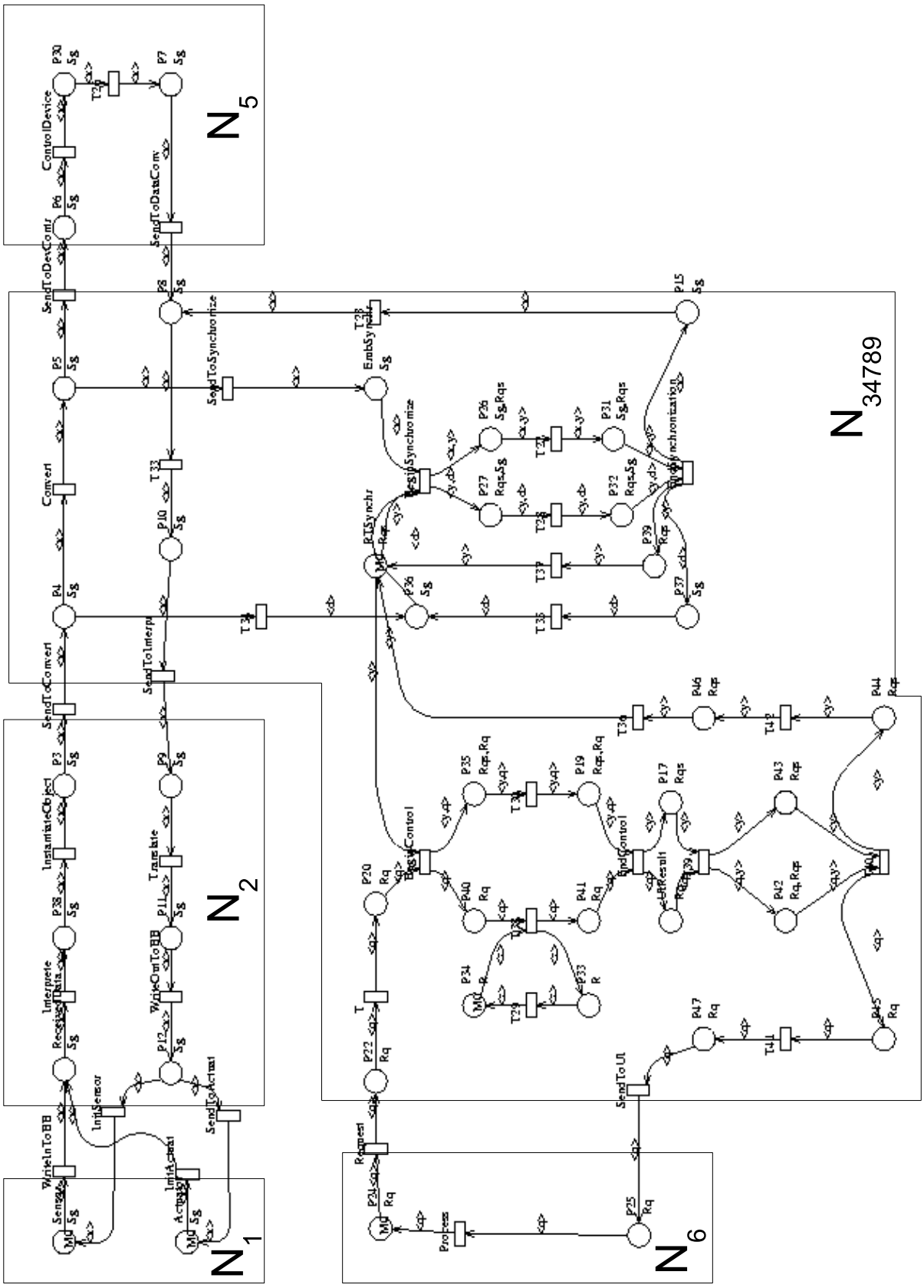


Figure 6.8: Décomposition de SWNs de l'exemple 6.7 compatible à l'analyse structurée

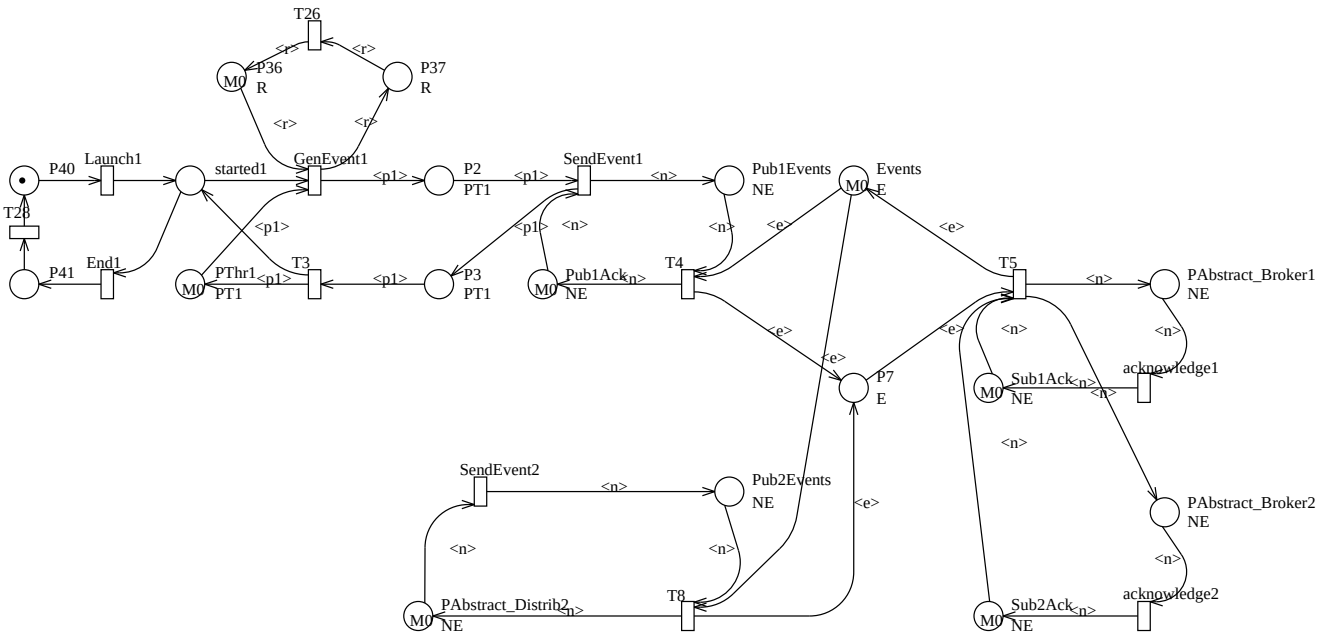


Figure 6.9: Réseau étendu du modèle SWN du composant StockDistributor 1 groupé avec son réseau fermant et le canal d'événements

```
   un espace d'états des tuples de marquages symboliques et une
   représentation tensorielle du générateur de la chaîne de Markov
   agrégée.
11.Calculer les indices de performance en utilisant cette expression
   tensorielle.
12.Traduire les résultats dans le contexte initial des composants.
END
```

6.6 Application à l'exemple du système de gestion des stocks de la bourse

Reprenons l'exemple du système de gestion des stocks de bourse, présenté en section 5.5.1.

Pour appliquer notre méthode structurée étendue aux CBS, après la modélisation que nous avons présenté dans le chapitre 5, nous cherchons d'abord une décomposition de SWNs compatible avec les conditions énoncées dans le théorème 6.4. La configuration de modèles SWNs à vérifier est celle incluant les modèles CC-SWNs des composants, le modèle SWN du canal d'événements à travers lequel sont interconnectés les composants *StockDistributors* et *StockBrokers*, le modèle du conteneur et les deux réseaux de Petri fermants qui ont été rajoutés. Nous notons que cet ensemble de (SWN_k) satisfait nos conditions d'analyse structurée.

Par ailleurs, pour éviter de faire les calculs pour des SWNs de taille très petite, nous avons choi-

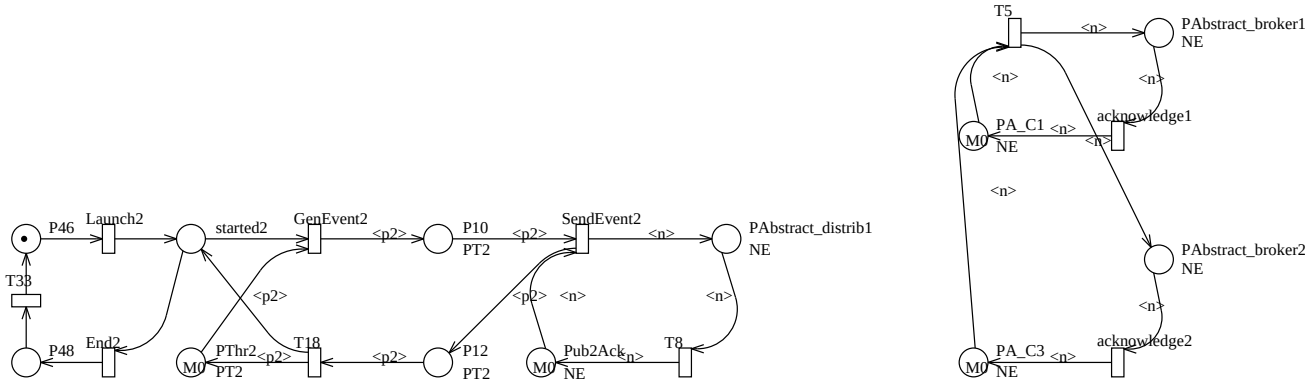


Figure 6.10: Réseau étendu du modèle SWN du composant StockDistributor 2 groupé avec son réseau fermant

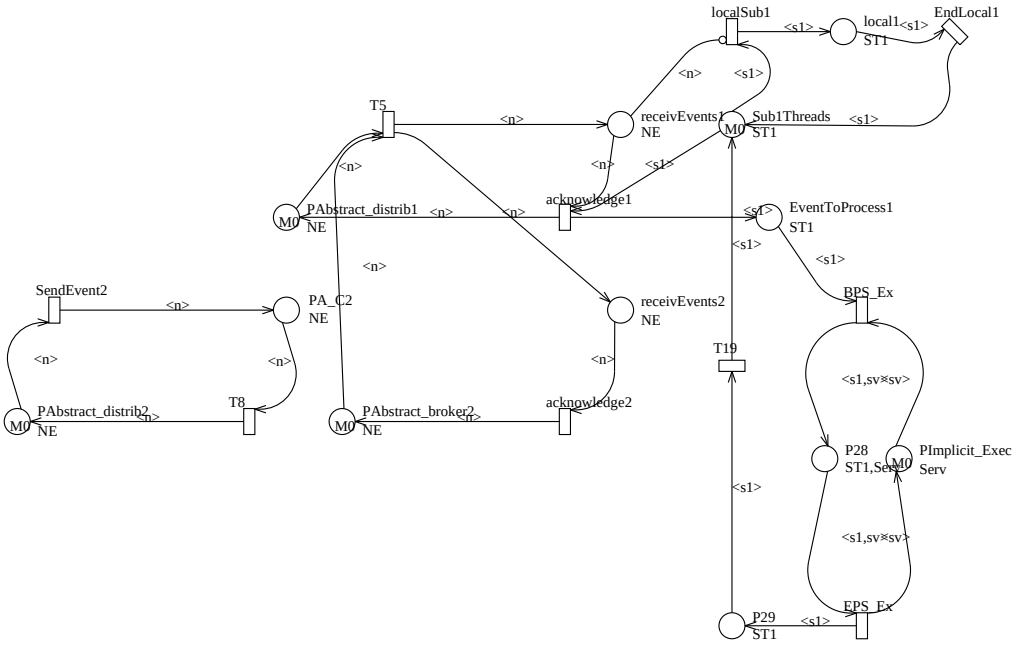


Figure 6.11: Réseau étendu du modèle SWN du composant StockBroker 1

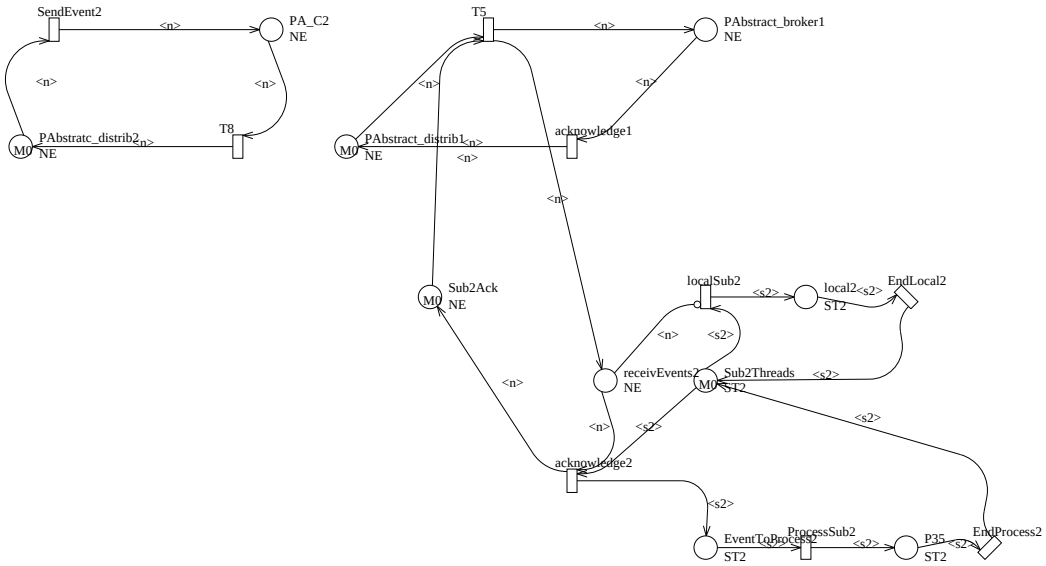


Figure 6.12: Réseau étendu du modèle SWN du composant StockBroker 2

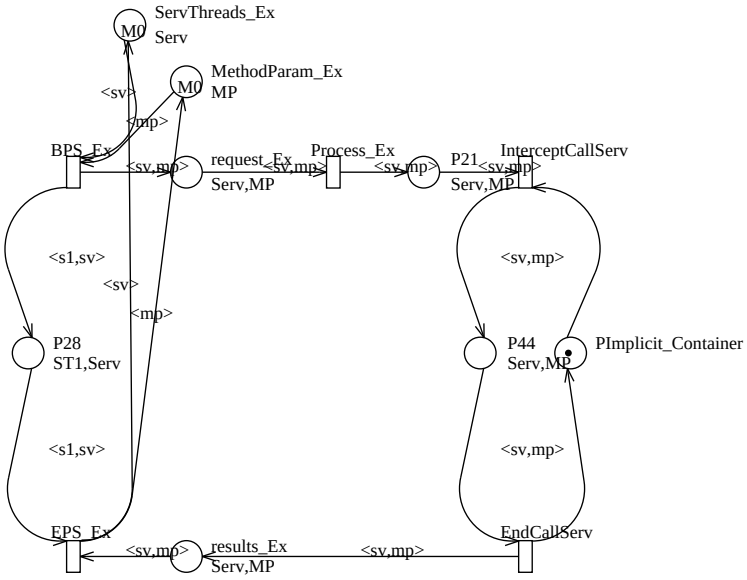


Figure 6.13: Réseau étendu du modèle SWN du composant Executor

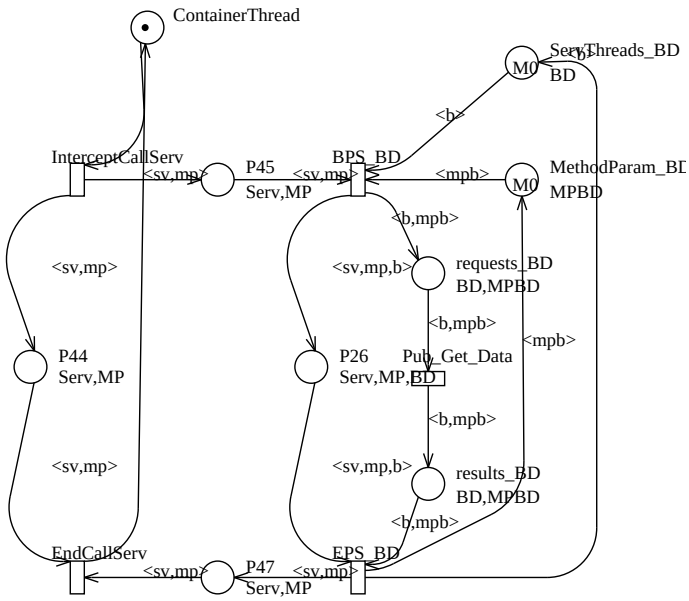


Figure 6.14: Réseau étendu du modèle SWN du conteneur groupé avec son service de persistence

Cf	|E|	|R|	|Serv|	|MP|	|BD|	|MPBD|
1	1	4	3	5	4	3
2	4	4	3	5	4	3
3	8	4	3	5	4	3
4	12	4	3	5	4	3

Table 6.1: Configurations de calcul de l'application

sis de grouper le composant *StockDistributor 1* avec son réseau fermant et le canal d'événement, le composant *StockDistributor 2* avec son réseau fermant, et le modèle du conteneur avec celui du service de persistance. Nous étendons donc ces SWNs en réseaux étendus et obtenons les figures 6.9, 6.10, 6.11, 6.12, 6.13 et 6.14.

Nous utilisons notre outil d'analyse compositionnelle *compSWN* (voir chapitre 10) sur cet ensemble de réseaux SWNs pour calculer les SRGs des réseaux étendus et les probabilités à l'équilibre.

6.6.0.1 Gains de la méthode structurée

Nous montrons d'abord les gains en temps de calcul et en occupation mémoire obtenus avec la méthode d'analyse structurée. Á cette fin, nous utilisons l'outil GreatSPN [173] (voir chapitre 10) sur le réseau global G-SWN obtenu pour comparer les résultats des deux méthodes d'analyse plate (GreatSPN) et compositionnelle (*compSWN*).

Nous faisons varier les cardinalités de nos classes de couleur de base, et nous étudions le compor-

Cf	NbS	NbO	TGreat	TComp	MGreat	MComp
1	51840	7538688	147	25	13.46	0.19
2	129600	60309504	537	116	37.13	0.45
3	233280	964952064	1130	382	68.65	0.78
4	336960	15439233024	1650	855	100.17	1.13

Table 6.2: Taille de l'espace d'états, temps de calcul et occupation mémoire pour diverses configurations de l'application

tement des deux solveurs pour des configurations diverses données dans la table 6.1. Les solveurs sont installés sur une station de travail Suse linux 9.2 avec 512 MO. Notons que nous supposons une cardinalité de valeur 1 pour les classes de couleur PT1, PT2, ST1 and ST2 représentant respectivement les publishers (StockDistributors) et subscribers (StockBrokers), étant donné qu'on a supposé des composants monothreadés.

Nous reportons dans la table 6.2 les résultats obtenus à partir des deux solveurs (GreatSPN and *compSWN*) en termes d'usage de la mémoire (en bytes) et temps de calcul (en secondes) incluant le calcul des probabilités à l'équilibre. Les probabilités obtenues par les deux solveurs sont identiques, avec une certaine précision d'erreur. Nous indiquons également la taille de l'espace d'états généré par le réseau global (trouvée identique également dans les deux solveurs) pour chacune des configurations de test. Les notations utilisées dans les tables 6.1 et 6.2 sont comme suit :

- |Couleur| est la cardinalité de la sous-classe statique de couleurs dénotée par *Couleur*,
- NbS est le nombre de marquages symboliques obtenus,
- NbO est le nombre de marquages ordinaires obtenus,
- TGreat est le temps de calcul global obtenu par GreatSPN,
- TComp est le temps de calcul global obtenu par *compSWN*,
- MGreat est la taille mémoire (en Megabytes) utilisée par GreatSPN, et
- MCcomp est la taille mémoire utilisée par *compSWN*.

On voit que les temps de calcul et taille mémoire obtenus par *compSWN*sont nettement meilleurs par rapport à ceux de GreatSPN.

6.6.0.2 Calcul des temps de réponse

Nous nous sommes intéressés à étudier la variation de trois indices de performance, à savoir:

1. Le temps de réponse du traitement d'un événement reçu dans le premier composant StockBroker, par rapport au taux de réception des événements, puisqu'il invoque un service d'un autre composant l'Executor, qui lui aussi invoque le service de persistance du conteneur.
2. Le temps de réponse du traitement local du premier composant StockBroker, par rapport au taux de réception des événements.
3. Le temps de réponse du composant Executor pour traiter une requête du composant StockBroker, par rapport au taux d'envoi des événements.

L'objectif de l'étude de ces variations est d'évaluer:

- Pour le premier indice, l'impact du service de conteneur sur le traitement d'une requête du composant StockBroker,

Transition	Taux	Transition	Taux
GenEvents1	0.75	BPS	0.9
GenEvents2	0.65	localSub1	8
SendEvents1	0.8	localSub2	8
SendEvents2	0.7	process	0.9

Table 6.3: Taux de franchissement des transitions de la configuration étudiée

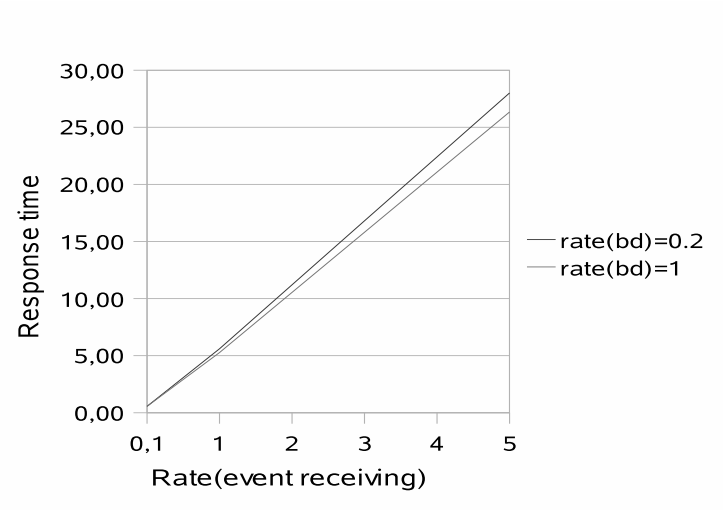


Figure 6.15: Temps de réponse du StockBroker pour le traitement d'un événement versus taux de réception des événements pour deux taux de traitement du service de conteneur

- Pour le second et troisième indice, l'impact de l'envoi et réception d'événements sur l'activité du composant StockBroker (traitement local et traitement d'un événement).

À cet effet, nous avons choisi une configuration de la table 6.2, à savoir la configuration 4, puisque le nombre de marquages symboliques est important par rapport aux autres configurations.

Notons toutefois que, même si nous avons utilisé des cardinalités petites de nos classes de couleur, une couleur peut modéliser un groupe d'entités élémentaires; par exemple une couleur de requête peut représenter 10, 100 ou 1000 requêtes. Évidemment, les taux de franchissement des transitions instanciant cette couleur doivent être adaptés à la sémantique d'un jeton coloré (100 requêtes impliquent éventuellement un taux de traitement de requête 100 fois (ou plus) plus lent par exemple).

Nous avons fixé les valeurs des taux de franchissement pour un ensemble critique de transitions (voir la table 6.3) et nous avons fait varier les taux de tir des transitions en relation avec l'indice calculé. Les transitions n'apparaissant pas sur la table ont par défaut un taux égal à 1, i.e. plus rapides par rapport aux autres transitions, les taux étant donné dans la même unité.

- Pour la première variation, nous avons fixé le taux de la transition *BPersistServ* du modèle SWN du service de persistence à une valeur de 0.2 puis 1. Nous avons fait varier le taux de la transition *receiveEvent1* du premier composant StockBroker, puis nous avons calculé le temps de réponse pour le traitement d'un événement. Le diagramme obtenu est présenté par la figure 6.15.

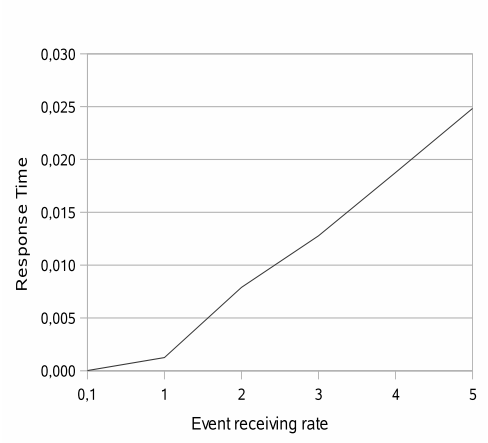


Figure 6.16: Temps de réponse du traitement local du StockBroker versus taux d'envoi d'un événement

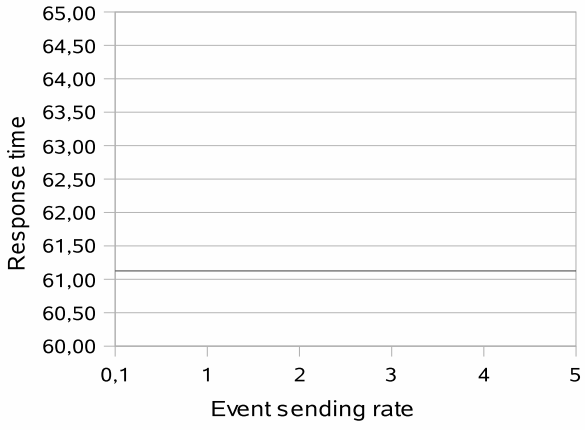


Figure 6.17: Subscriber request response time versus event sending rate

- Pour la seconde et la troisième variations, nous avons pris plusieurs valeurs des taux des transitions *SendEvent1*, respectivement *receiveEvent1*. Nous avons calculé le temps de réponse pour une requête du StockBroker traitée par le composant Executor, ainsi que pour une activité locale du StockBroker. Les figures 6.16 et 6.17 montrent ces variations.

Le premier diagramme montre un temps de réponse qui augmente avec l'augmentation du taux de réception des événements pour les deux courbes. Cependant, nous pouvons distinguer un temps de réponse amélioré (réduit) dans la seconde courbe, correspondant à un taux de traitement de l'Executor plus important. Ce phénomène est une preuve de l'effet du service de persistance du conteneur sur le temps de réponse du traitement d'une requête, et donc un impact sur les performances globales du système. Il est donc préconisé d'utiliser judicieusement les services d'un conteneur lorsqu'on développe une application basée composant.

Dans le second diagramme, le temps de réponse s'accroît très lentement dans la première période de temps, puis rapidement augmente avec l'augmentation du taux de réception des événements. Ceci est

attendu, d'une manière similaire à l'augmentation du temps de traitement d'une activité locale, qui est ralenti par le traitement des événements.

Le dernier diagramme montre également un impact négligeable du taux d'envoi des événements sur le traitement d'une requête du StockBroker: le temps de réponse reste pratiquement similaire (61.1 pour les taux d'envoi d'événements allant de 0.1 a 5). Ceci est également attendu puisque les événements n'interrompent pas le travail du composant Executor.

6.7 Conclusion

Nous avons détaillé dans ce chapitre l'étape d'analyse des performances proprement dite d'un CBS, en donnant les éléments nécessaires et les conditions à vérifier pour satisfaire une forme structurée des modèles SWNs, permettant une analyse compositionnelle.

Nous avons montré que la modélisation adoptée dans le chapitre précédent répond déjà aux conditions syntaxiques de décomposition synchrone et asynchrone de SWNs, ce qui élargit le champs d'application de la méthode que nous proposons. Toutefois, de nouvelles conditions sont nécessaires pour éviter les interférences entre les compositions synchrones et asynchrones. Si ces conditions ne sont pas vérifiées pour certains sous-ensembles de SWNs, nous avons proposé de les regrouper pour tenter de faire une analyse compositionnelle à un niveau plus grand de granularité des SWNs. Évidemment, plus la fusion des SWNs est large, moins on bénéficiera de l'analyse structurée proposée. Dans le pire des cas, on obtient le réseau global sans aucune analyse compositionnelle.

Par ailleurs, la granularité des SWNs à analyser présente un choix essentiel pour réduire ou augmenter la complexité d'analyse. Ce point est discuté dans le chapitre suivant.

CHAPITRE 7

Étude de la granularité d'analyse d'une composition de SWNs

7.1 Introduction

La méthode d'analyse structurée donne lieu à des gains significatifs en temps de calcul et en occupation mémoire, par rapport à une analyse du réseau global. Ceci permet d'analyser des CBS non analysables avec une approche globale.

Toutefois, à travers les exemples de compositions mixtes de CBS que nous avons étudiées, nous nous sommes rendus compte que la présence de nombreux « petits » SWNs augmentait notablement les temps de calcul. Ceci peut s'expliquer par le fait qu'un grand nombre de SWNs à analyser nécessite beaucoup de synchronisations de couleurs de chaque réseau étendu avec les autres, ce qui augmente le temps de calcul du SRG de chaque réseau étendu, et donc augmente le temps de calcul global.

En conséquence, il est important d'étudier le choix de la granularité de composition à adopter avant d'effectuer une analyse sur une configuration de SWNs. Partant de la configuration initiale des CC-SWNs et CI-SWNs d'un CBS à analyser, la question qui se pose est de savoir si l'on va garder cette configuration pour l'analyse (lorsque les conditions d'analyse structurée sont satisfaites évidemment), ou si l'on doit regrouper des composants (leurs CC-SWNs) afin de former des SWNs plus grands. Nous étudions ce problème dans ce chapitre.

7.2 Étude du problème

La méthode d'analyse d'un CBS modélisé par une composition mixte de SWNs s'opère en cinq étapes:

1. Extension des réseaux CC-SWNs $N_k, k = 1, \ldots, K$. L'ensemble des réseaux étendus est noté $\left\{\overline{N_k}, \ldots, \overline{N_K}\right\}$.
2. Génération des $\overline{SRG_k}$ des SWN étendus $\overline{N_K}$. Cette étape est effectuée avec une fonction de l'outil GreatSPN qui permet le calcul du graphe symbolique d'accessibilité d'un SWN. Pour chaque réseau étendu $\overline{N_K}$, un ensemble d'accessibilité est obtenu, complété par une collection de matrices contenant les franchissements des transitions. Cet ensemble de matrices est composé d'une matrice Q' regroupant les transitions dites locales (ne mettant en jeu que les réseaux N_K eux-mêmes) et une série de matrices contenant les franchissements des transitions globales (mettant en jeu plusieurs réseaux étendus).

3. Calcul et stockage de l'ensemble effectif d'accessibilité du G-SWN. Cette étape est menée suivant une technique d'encodage de l'ensemble par une structure *Multi Decision Diagram, MDD* [154], proposée par Ciardo et Miner. Le principe des MDDs est de stocker un ensemble de K-tuples d'états sous-forme d'un arbre à $K+1$ niveaux, en tenant compte du fait que les états sont générés par la survenue d'événements locaux ou globaux. Un MDD est une fonction qui, à un tuple associe 1 s'il est accessible et 0 sinon. Le $K+1$ nième niveau contient les deux noeuds spéciaux 0 et 1 donnant la valeur de fonction. L'approche MDD est basée sur la manipulation des événements du système global. Deux types d'événements sont distingués : les événements locaux qui ne mettent en jeu qu'un seul réseau étendu, et les événements globaux faisant intervenir plusieurs réseaux. La construction de ces ensembles d'événements se fait sur les $\overline{SRG_k}$. À partir de l'ensemble des événements, une méthode dite de *saturation* [139] est utilisée pour générer de façon optimale le MDD représentant l'ensemble des K-tuples effectivement accessibles. La méthode de saturation implantée dans CompSWN est identique à celle de Ciardo du point de vue algorithmique, par contre elle n'intègre pas encore les mécanismes d'optimisation de type caches et tampons.
4. Calcul de la forme tensorielle du générateur. Ce générateur était initialement une sur-matrice du générateur obtenu par la méthode classique sur le modèle global. Cette sur-matrice est définie sur l'ensemble potentiel, produit cartésien des sous-ensembles d'accessibilité. Les travaux de Ciardo et Miner ont permis de restreindre l'expression tensorielle aux tuples d'états effectivement accessibles en se servant du MDD préalablement calculé. Ils ont défini une structure de données, appelée Matrix decision Diagram (MxD) [53], permettant une représentation optimale d'un produit tensoriel de n matrices, restreint à un ensemble de tuples d'états représenté sous forme de MDD. Un MxD est fortement ressemblant à un MDD, à la différence que les noeuds contiennent des matrices. Dans cette représentation, la forme tensorielle est limitée aux tuples d'états effectivement accessibles codés par le MDD, ce qui fait son intérêt. Le principe est de calculer récursivement la restriction d'une matrice d'un noeud aux états présents dans le noeud adjacent du MDD.

   L'algorithme de calcul du MxD est basé sur l'ensemble des événements et évidemment sur le MDD associé calculé précédemment. Comme pour le calcul du MDD, les algorithmes implantés sont identiques à ceux définis par Ciardo, mais sans optimisation de cache et tampons.
5. Résolution à l'équilibre du système linéaire $\pi.Q = 0$. Cette résolution se base sur l'expression tensorielle du générateur sous forme de MxD et d'une méthode itérative de type Gauss-Seidel. Cette méthode peut être encodée relativement simplement pour utiliser la représentation en MxD.

On conçoit que cette méthode soit sensible à la taille des composants constituant le modèle du CBS à analyser. Sans précision particulière, on peut penser intuitivement qu'un nombre « important » de « petits » composants SWNs induit un temps de calcul plus grand par rapport à celui d'un modèle fusionnant « certains » de ces SWNs. Ceci serait particulièrement dû au temps requis pour l'étude de chaque réseau étendu exposant plus de comportements que la partie correspondante dans le système global. Afin d'aller plus avant, étudions deux exemples de CBS et différents regroupements de leurs composants.

7.3 Étude du système EJB/CORBA

Le premier exemple est celui d'une infrastructure basée EJB (Enterprise JavaBeans)/CORBA [31], interconnectant un serveur J2EE offrant des composants EJB [204] à un système CORBA [166](voir figure 7.1). Ce système a été mis en place par plusieurs entreprises ayant d'une part des applications développées avec différents langages de programmation orientés objet connectées à travers des bus logiciels dits ORBs (Object Request Broker), et d'autre part des applications basées sur des systèmes J2EE.

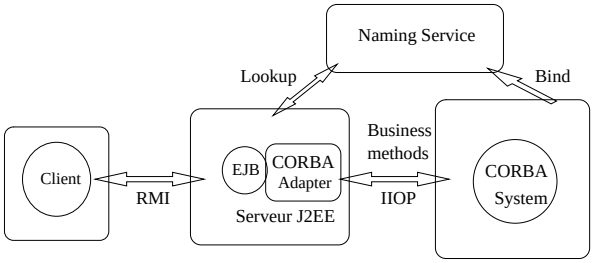


Figure 7.1: L'architecture du système EJB/CORBA

Un besoin de connecter des systèmes CORBA à des systèmes J2EE a alors émergé. Dans ce type d'infrastructure, des clients communiquent avec les composants EJB hébergés dans un serveur J2EE, à travers le protocole RMI.

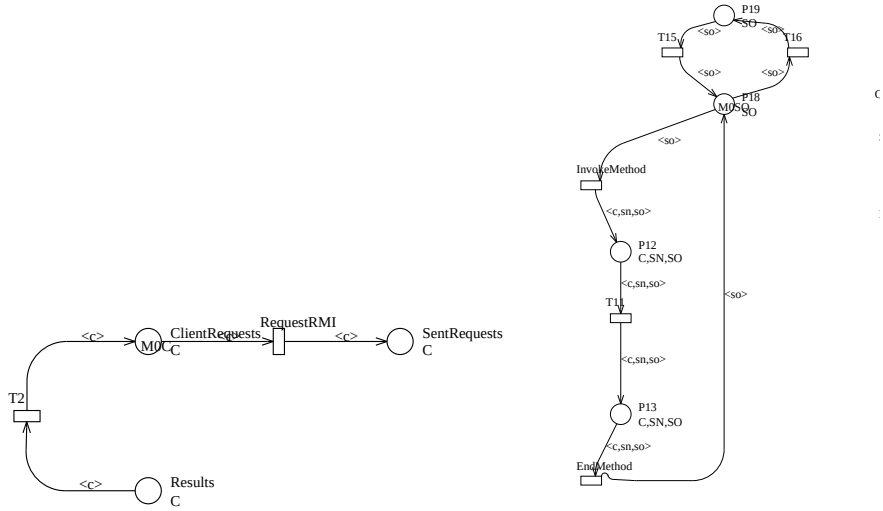
Le serveur J2EE communique avec le système CORBA en utilisant le protocole IIOP (Internet Inter-Orb Protocol) et devient un client CORBA. Afin de permettre la communication, l'architecture est dotée d'un service d'annuaire implanté par un service de noms. Un scénario de requête est donné ci-après: une application cliente envoie une requête RMI au conteneur EJB. Pour traiter la requête, soit la méthode invoquée est exécutée localement, dans le cas où l'objet requis est local au conteneur EJB, soit le serveur CORBA est sollicité dans le cas d'un objet distant. L'architecture EJB/CORBA est donc constituée essentiellement de trois composants (en omettant le composant serveur de noms): le client, le composant EJB et le système CORBA. La modélisation donne lieu aux 3 composants SWNs de la figure 7.2(a).

7.3.1 Composants du CBS

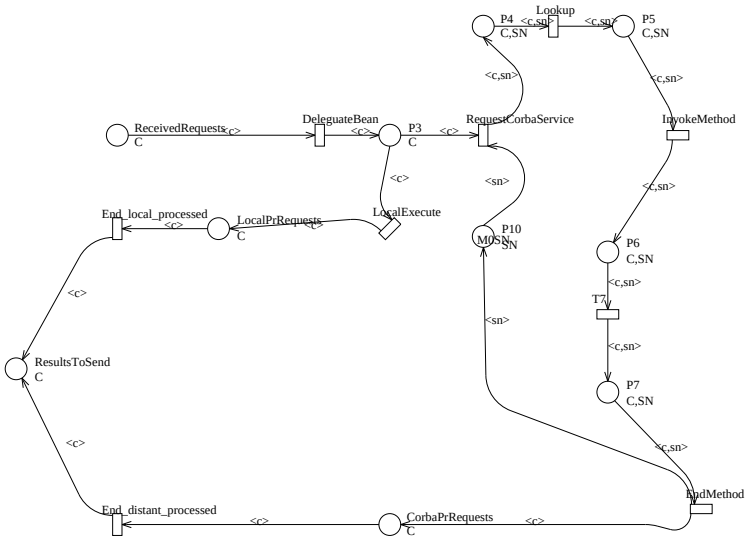
Le composant client Le composant client (figure 7.2(a)) décrit une requête demandant à exécuter un service. La requête est identifiée en utilisant une classe de couleurs de base notée C modélisant l'identification des clients. Le composant client interagit avec le serveur d'objets CORBA à travers le serveur EJB J2EE. Il envoit une requête RMI (ou IIOP) via la transition *RequestRMI* vers la place *SentRequests*. Lorsque le serveur EJB aura traité la requête, les résultats de requête sont envoyés au client et reçus dans la place *Results*.

Le composant EJB Le composant serveur EJB (figure 7.2(c)) est modélisé par une exécution locale de la requête client (voir la transition *LocalExecute*), ou une requête à un service CORBA. Le composant EJB a besoin d'identifier d'abord quel est le nom du service qui est invoqué. Ceci nécessite l'utilisation de la classe de couleurs SN représentant les noms de service CORBA. L'identification du service CORBA est accompli en faisant une demande de recherche dans le serveur de noms qui est abstrait ici (voir transition *lookup*). L'appel à un service CORBA se fait à travers la transition *InvokeMethod* et se termine par la transition *End_method*. Une fois la requête traitée, le composant termine sa communication avec le client en lui envoyant les résultats via la transition *End_local_processed* si le traitement de la requête est local au composant EJB, ou bien par la transition *End_distant_processed* dans le cas où le traitement est effectué par le système CORBA.

Le système CORBA Lorsque le système CORBA (figure 7.2(b)) est sollicité pour exécuter une méthode métier, il se synchronise avec le serveur EJB demandeur via la transition *InvokeMethod*, en utilisant



(a) Client (b) Composant CORBA



(c) Composant EJB

Figure 7.2: SWNs des composants de l'architecture EJB/CORBA

Config	|C|	|SN|	|SO|
1	5	2	2
2	10	2	2

Table 7.1: Différentes configurations de test du système EJB/CORBA

Étape		Nb Symb.	Nb Ord.	Durée pour 2 composants	Durée pour 3 composants
Calcul SRG	Composant Client	21	243	-	0
	Composant EJB	1631	315625	-	4
	Système Corba	10	244	0	0
	Composant client+EJB	3158	711296	5	-
Calcul MDD		-	-	0	1
Calcul MxD		-	-	1	1
Résolution $\pi.Q = 0$		-	-	3	4
Temps global				9	10

Table 7.2: Taille des SRSs et temps de calcul du système EJB/CORBA pour la configuration 1

une nouvelle classe de couleurs SO modélisant les objets CORBA. Il fait alors une recherche des données requises, puis termine la requête en envoyant les résultats à travers la transition *End_method*.

7.3.2 Analyse et comparaisons

L'analyse du système EJB/CORBA a été faite de deux manières sur deux configurations différentes des paramètres du modèle G-SWN obtenu (voir tableau 7.1):

- La première manière consiste à garder l'architecture telle quelle, c.à.d. composée de trois composants, puis appliquer la méthode structurée d'analyse en notant les temps de calcul des SRGs des composants SWNs étendus, le temps de fusion de ces SWNs (temps de calcul du MDD), le temps de calcul de la forme tensorielle du générateur de la composition (temps de calcul du MxD) et le temps de résolution du système linéaire associé à l'équilibre.
- La seconde manière consiste à fusionner deux composants, résultant ainsi en une configuration de deux composants uniquement, puis appliquer la méthode structurée d'analyse et comparer les temps de calcul engendrés avec ceux obtenus pour la configuration de 3 composants. Les deux composants fusionnés sont le client et le serveur EJB, étant donné que le client est vu comme le plus petit composant d'une part, et d'autre part il ne peut être fusionné qu'avec son voisin qui est le composant EJB.

À l'issue de cette étude, les tableaux 7.2 et 7.3 ont été obtenus. Nous remarquons qu'en fusionnant les composants EJB et client, le temps global d'analyse évolue différemment selon les configurations par rapport à celui de l'analyse de la composition des trois composants séparés. Dans la première configuration, la réduction de temps est faible en regroupant Client et EJB. Dans la deuxième configuration, qui accroît le cardinal de l'ensemble d'accessibilté symbolique (SRS) du composant EJB, le temps de calcul

Étape		Nb Symb.	Nb Ord.	Durée pour 2 composants	Durée pour 3 composants
Calcul SRG	Composant Client	66	59049	-	0
	Composant EJB	23949	5063636075	-	131
	Système Corba	10	884	0	0
	Composant client+EJB	54912	17192549376	679	-
Calcul MDD		-	-	4527	231
Calcul MxD		-	-	194	49
Résolution $\pi.Q = 0$		-	-	3521	727
Temps global				8921	1138

Table 7.3: Taille des SRSs et temps de calcul du système EJB/CORBA pour la configuration 2

augmente fortement en regroupant le client et l'EJB! Ce comportement est contre-intuitif: regrouper des composants conduit à augmenter les temps de calcul! On peut émettre l'hypothèse que le regroupement de SWN à tailles de SRGs très différentes ne conduit pas à un gain de temps.

7.4 Étude d'un système d'achat en ligne

Le second exemple est celui d'un système gérant un processus métier d'achat en ligne (Online Purchase Business Process [72]). La figure 7.3 montre cette application e-commerce. Dans ce processus business, l'utilisateur peut se connecter en utilisant un compte existant, ou sélectionner un ou plusieurs articles à acheter, ou bien créer un nouveau compte. Une fois l'utilisateur connecté et ayant choisi les articles à acheter, le paiement est activé. La réception des articles a lieu une fois le paiement effectué.

Le paiement se fait comme suit: initialement, l'utilisateur est autorisé à sélectionner une méthode de paiement. Selon la méthode choisie, une « requête d'envoi de facture » est soumise au composant de facturation, ou bien une vérification de l'avoir du compte bancaire de l'utilisateur est réalisée par le composant banque. Si l'utilisateur ne dispose pas de suffisamment d'argent nécessaire à sa transaction, il va recevoir une facture. Ainsi, une « requête d'établissement de facture » peut être activée soit à la « vérification du compte », soit directement à la « sélection de la méthode de paiement ». Si l'utilisateur dispose de l'argent nécessaire à son achat en ligne, le compte est débité. L'interaction de « paiement abstrait » est achevée soit lors du débit direct du compte utilisateur soit à la réception d'une notification du composant de facturation indiquant que le paiement a été effectué

L'application est donc constituée de quatre composants: l'interface utilisateur, le composant processus métier, le composant banque et le composant de facturation. Le composant banque est divisé en deux sous-composants: l'un s'occupe de la vérification des comptes bancaires, et l'autre gère les opérations de débit. Le composant de facturation est constitué également de deux sous-composants: l'un pour l'établissement des factures et l'autre pour la réception du paiement.

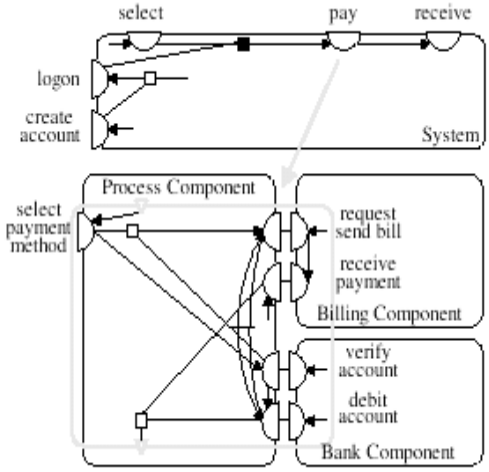


Figure 7.3: Architecture du système d'achat en ligne

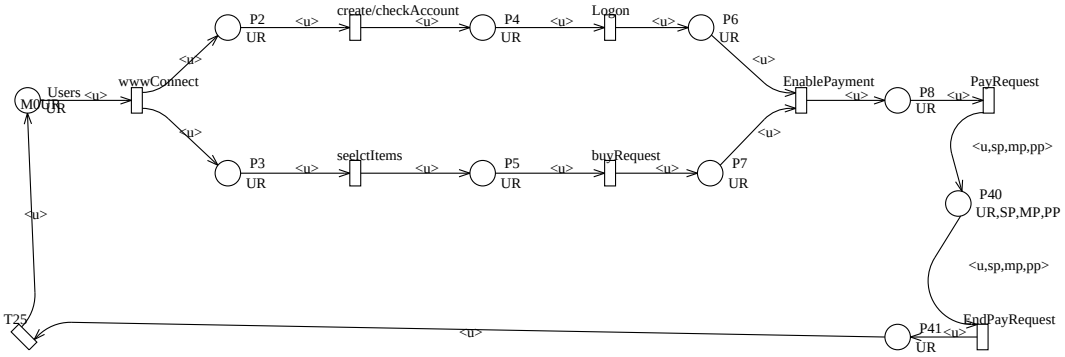
7.4.1 Composants du CBS

Notons que les modèles associés aux composants du CBS donnés dans les figures 7.4, 7.5 et 7.6 sont les CC-SWNs obtenus après modification des interfaces.

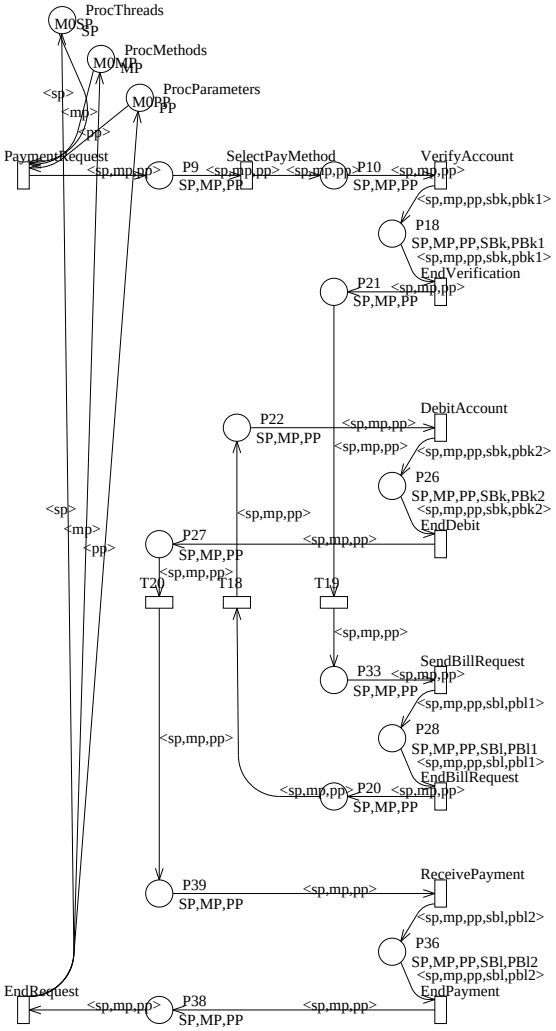
Le composant interface utilisateur Ce composant (voir figure 7.4(a)) décrit la connexion d'un utilisateur, la sélection de produits à acheter et le lancement de la requête du paiement. Les requêtes utilisateur sont identifiées grâce à une classe de couleurs de base notée UR. Elles se terminent par une requête de paiement adressée au composant processus métier à travers la paire de transitions *PayRequest* et *EndPayRequest*. D'autres classes de couleur (SP, MP et PP) interviennent lors de la synchronisation du composant interface avec le composant processus métier (voir l'explication ci_dessous).

Le composant processus métier Celui-ci reçoit les requêtes utilisateur et les fait passer par une suite d'étapes, tout en faisant appel aux composants banque et de facturation (figure 7.4(b), en bas): sélection de la méthode de paiement, vérification du compte bancaire, débit du compte, demande de facture, ... Les classes de couleur utilisées sont SP, MP et PP représentant respectivement les entités threads du composant, les méthodes appelées et leurs paramètres. L'appel de méthodes associées aux composants banque et de facturation se fait via une synchronization du composant processus métier avec ces composants, ce qui fait intervenir de nouvelles classes de couleur correspondant à ces composants, à savoir SBk, PBk1, PBk2, SBl, PBl1, PBl2 (voir ci_après pour l'explication des classes de couleurs).

Le composant banque Le composant banque (figure 7.5) consiste à faire une vérification du compte bancaire d'un client et réaliser un débit de compte. Chacune de ces opérations sont opérées par un sous-composant distinct. La vérification de compte nécessite l'utilisation de deux classes de couleur SBk et PBk1 désignant les méthodes associées au sous-composant et leurs paramètres. Le débit de compte requiert également l'utilisation d'une autre classe de couleur PBk2 désignant les paramètres correspondant aux méthodes invoquées.

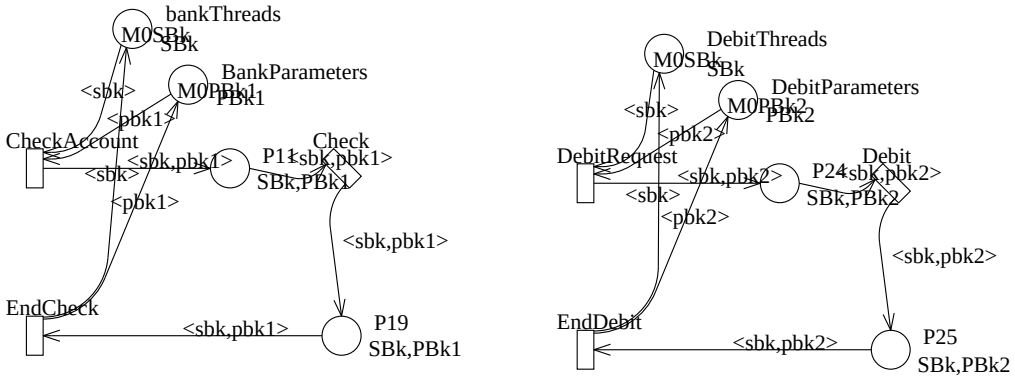


(a) Client



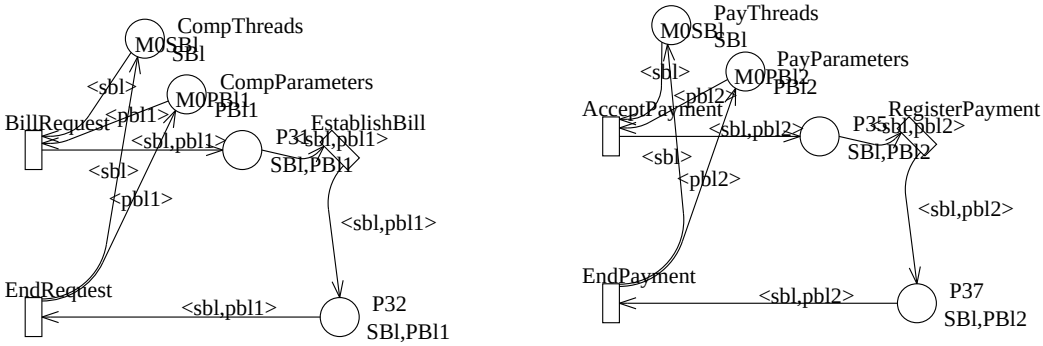
(b) Composant CORBA

Figure 7.4: SWNs de l'interface utilisateur et du composant processus métier



(a) Banque 1 (b) Banque 1

Figure 7.5: SWNs des composants banque



(a) Facturation 1 (b) Facturation 1

Figure 7.6: SWNs des composants de facturation

Config	|UR|	|SP|	|MP|	|PP|	|SBk|	|PBk1|	|PBk2|	|SBl|	|PBl1|	|PBl2|
1	2	2	2	2	2	2	2	2	2	2
2	5	2	2	2	2	2	2	2	2	2

Table 7.4: Différentes configurations de test du système d'achat en ligne

Le composant de facturation De même que le composant banque, le composant de facturation (figure 7.6) peut établir des factures à l'aide d'un premier sous-composant et répondre aux requêtes de réception de paiement à travers un deuxième sous-composant. Le premier sous-composant utilise deux classes de couleur SBl et PBl1 identifiant les méthodes du composant et les paramètres associés. Le second sous-composant utilise en plus une autre classe de couleur PBl2 donnant les paramètres propres aux méthodes invoquées au sous-composant.

7.4.2 Analyse et comparaisons

Comme pour le système EJB/CORBA, l'analyse du système d'achat en ligne a été faite pour deux configurations différentes des paramètres du modèle G-SWN obtenu (voir tableau 7.4). Cette analyse a été conduite sur une composition de 6 composants du CBS et sur une autre composition de 2 composants:

- La première organisation de composants contient l'interface utilisateur, le composant processus métier, les deux sous-composants de banque et les deux sous-composants de facturation.
- La seconde organisation est obtenue en gardant le composant interface utilisateur tel quel et en fusionnant le reste des composants en un seul macro-composant. Ce choix de fusion a été fait sur la base des tailles des SRS des composants de base (voir les tableaux ci-après), en regroupant les composants à SRS les plus petits, conformément à l'hypothèse émise lors de l'étude de l'exemple précédent : en effet, le composant interface utilisateur comporte un espace d'états très important par rapport aux autres composants, en particuliers par rapport aux composants banque et facturation.

Pour ces deux organisations, la méthode structurée d'analyse a été appliquée et les temps de calcul cités dans l'exemple du système EJB/CORBA ont été notés. Les tableaux 7.5 et 7.6 ont été obtenus. Nous constatons que les temps de calcul augmentent dans la configuration 2, faiblement pour l'organisation 1 et très fortement pour l'organisation 2. Ceci confirme l'intuition selon laquelle la fusion de composants réduit les temps de calcul.

7.5 Synthèse et conclusion

La première conclusion que nous déduisons de cette étude est que la structure d'une conception orientée composant *peut être différente* de celle utilisée pour une analyse efficace orientée composant.

La deuxième remarque qui découle des deux exemples ci-dessus et de l'ensemble des exemples que nous avons traité pendant notre travail de thèse, est qu'il semble très difficile d'énoncer des règles générales permettant de guider un regroupement de modèles SWN de composants pour améliorer l'analyse structurée.

Toutefois, nous donnons ci-après quelques observations qui peuvent conduire à une analyse plus efficace par rapport à une analyse fondée sur la traduction directe d'un CBS.

Étape		Nb Symb.	Nb Ord.	Durée pour 2 composants	Durée pour 3 composants
Calcul SRG	Interface utilisateur	91	344	0	0
	processus métier	200	209633	-	1
	S/composant banque1	27	577	-	0
	S/composant banque2	27	577	-	0
	S/composant factur.1	27	577	-	0
	S/composant factur.2	27	577	-	0
	Composant process+banque+factur.	340	24481	3	-
Calcul MDD		-	-	0	0
Calcul MxD		-	-	0	0
Résolution $\pi.Q = 0$		-	-	0	2
Temps global				3	3

Table 7.5: Taille des SRS et temps de calcul du système d'achat en ligne pour la configuration 1

Etape		Nb Symb.	Nb Ord.	Durée pour 2 composants	Durée pour 3 composants
Calcul SRG	Interface utilisateur	6501	1301252	23	23
	processus métier	200	58701	-	7
	S/composant banque1	27	577	-	0
	S/composant banque2	27	577	-	0
	S/composant factur.1	27	577	-	0
	S/composant factur.2	27	577	-	0
	Composant process+banque+factur.	340	24481	16	-
Calcul MDD		-	-	104	341
Calcul MxD		-	-	4	6
Résolution $\pi.Q = 0$		-	-	111	1877
Temps global				258	2254

Table 7.6: Taille des SRS et temps de calcul du système d'achat en ligne pour la configuration 2

Tout d'abord, il faut rappeler que le choix de granularité des SWN dépend des *comportements* de ces SWNs, c'est à dire de leurs SRGs, et pas seulement de leur structure syntaxique. C'est donc l'influence des différents SWNs et leurs interactions sur les SRG étendus resultants qui conditionne l'efficacité de tel ou tel regroupement.

Sur le plan syntaxique, c'est à dire en se fondant sur la définition des réseaux et non sur le calcul des SRGs, les éléments suivants doivent guider le modélisateur :

- la taille du marquage initial (rappelons que nous considérons des marquages initiaux symétriques) conditionne les tailles des SRGs. Cette information peut renseigner sur la taille des SRGs, mais pas nécessairement.
- En particulier, les cardinaux des classes de couleur et spécialement ceux des couleurs de synchronisation ou globales ont une influence importante. Il est pertinent de regrouper des SWNs qui ont de telles classes de couleur de cardinal significatif.
- De même, il est pertinent de regrouper des SWNs qui ont un nombre plus élevé de couleurs de synchronisation ou de couleurs globales que d'autres SWNs.
- Les compositions asynchrones peuvent conduire à des SWNs étendus nettement moins contraints que les parties du SWN global correspondantes. Il peut être pertinent de regrouper des SWNs en composition asynchrone plutôt que synchrone.

Au delà de ces observations, les exemples étudiés semblent indiquer que le regroupement de SWNs dont les SRSs ont les tailles les moins différentes possibles est plus efficace que le regroupement de SWNs dont les tailles de SRSs sont « très » différentes. Nous n'avons pas d'explication satisfaisante de ce phénomène. À la lecture des résultats expérimentaux, l'accroissement des temps de calcul est différent selon que l'on considère le temps de construction du MDD et des MxD ou celui de la résolution de $\pi.Q = 0$. Il est probable que la méthode numérique de résolution de cette équation influe sur le temps de calcul. Par ailleurs, l'importance des interactions entre SWNs est également un paramètre primordial dans l'efficacité du regroupement; or elle ne dépend pas nécessairement de la taille des SRSs des SWNs en jeu : un SWN peut comporter un nombre élevé d'états liés par des franchissements locaux.

En résumé, partant de la composition issue de l'architecture du CBS, nous conseillons de déterminer si possible les SWNs dont les SRS sont les plus petits et de regrouper ces SWNs lorsqu'ils sont en interaction. Le nombre de regroupements est pratiquement impossible à préciser *a priori*. Si l'analyse s'avère impossible ou particulièrement lente, il est nécessaire de réduire le nombre de regroupements.

Nous présentons dans la suite des applications de notre méthode d'analyse à certains modèles de composant, à savoir le modèle *Fractal* et le modèle *CCM*.

CHAPITRE 8
Analyse des CBS FRACTALJulia

8.1 Introduction - Objectifs

Le modèle FRACTAL est l'un des modèles de composant académiques les plus répandus dans la littérature. Divers projets de recherche l'ont adopté et utilisé pour le développement d'applications, allant de la mise en place de systèmes d'exploitation spécialisés pour des plate-formes embarquées [78; 4] jusqu'à la mise en place de systèmes autonomes pour des serveurs d'applications d'entreprises [35], en passant par des systèmes multimédia pour des applications mobiles [124].

Comme ce modèle est générique, très riche en concepts et en pouvoir d'expression des architectures à composant, il est intéressant d'étudier l'application de notre méthode d'analyse sur de tels systèmes basées composant.

Nous nous intéressons donc dans ce chapitre à instancier notre méthode d'analyse au CBS FRACTAL. Nous avons auparavant précisé dans le chapitre 5 l'influence de l'implantation d'un modèle sur l'analyse. Pour cela, nous avons opté pour étudier des CBS FRACTAL développés pour l'implémentation de référence Java *Julia*. Pour ce faire, nous présentons d'abord le modèle FRACTAL, puis l'application de l'analyse structurée.

8.2 Le modèle FRACTAL

FRACTAL [43] est un modèle de composant général, développé au sein du consortium ObjectWeb par France Telecom R&D et l'INRIA. Il vise à implanter, déployer, gérer et configurer dynamiquement des systèmes logiciels complexes, incluant des systèmes d'exploitation et des intergiciels ou middlewares. Il est utilisé dans divers projets de recherche.

8.2.1 Caractéristiques

Constitution du modèle Un composant FRACTAL est une entité *en exécution* interagissant avec son environnement (i.e. avec d'autres composants) à travers des *interfaces* bien définies. (figure 1.2). Deux types d'interfaces ont été définis dans le modèle FRACTAL : une interface *serveur* correspondant à une offre de services, et une interface *cliente* permettant d'émettre des requêtes de services vers d'autres composants. En plus de la nature client ou serveur, des propriétés de contingence (obligatoire ou optionnelle) et de cardinalité (acceptation de connexions simples ou multiples) sont attribuées aux interfaces. Ces propriétés améliorent le niveau de description de l'architecture logicielle et enrichissent entre autres les vérifications architecturales que l'on peut implanter.

Un composant FRACTAL possède deux parties : une partie *contenu* et une partie de *contrôle* :

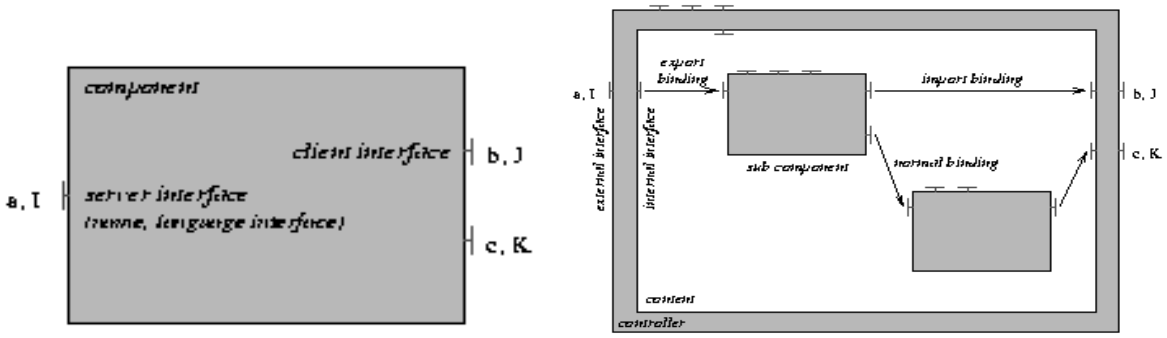


Figure 8.1: Vues externes (à gauche) et internes (à droite) d'un composant FRACTAL

- La partie contenu consiste en un nombre fini de *sous-composants*, rendant le modèle récursif et permettant d'emboîter des composants à un niveau arbitraire. Un composant ayant un tel contenu est dit *composite*, à l'encontre d'un composant *primitif* qui dispose d'un contenu sous-forme de boîte noire, dont la fonction est programmée dans un langage de programmation.
- La partie contrôleur, appelée *la membrane* permet de séparer l'implantation du métier du composant de sa partie de contrôle. Elle implante des fonctions de contrôle telles que des fonctions d'interception pour tracer ou filtrer les interactions d'un composant avec son environnement. Elle expose aussi un ensemble d'interfaces dites *de contrôle*, supportant l'introspection (monitoring), l'intercession et la reconfiguration de propriétés internes au composant, telles que la suspension et la reprise d'activités d'un sous-composant. Plusieurs interfaces de contrôle ont été définies dans la spécification FRACTAL à savoir :
  - Le contrôleur de cycle de vie (Life cycle Controller, LC) qui gère le cycle de vie d'un composant, en support pour la reconfiguration dynamique. Les méthodes de base sont le démarrage et l'arrêt de l'exécution du composant.
  - Le contrôleur de liaisons (Binding Controller, BC) qui gère les connexions (dites *liaisons*) avec d'autres composants, permettant de lier et délier des interfaces de composants communicants.
  - Le contrôleur de contenu (Content Controller, CC) qui fournit des opérations de contenu telles que lister, ajouter et supprimer des sous-composants.
  - Le contrôleur d'attributs(Attribute Controller, AC) qui expose des opérations de lecture (get) et de modification (set) d'attributs.

  Ces interfaces ne sont pas fixes. D'autres interfaces de contrôle peuvent être spécifiées et rajoutées en fonction des besoins spécifiques liés au contexte applicatif.

La *membrane* d'un composant peut également avoir des interfaces *externes* et *internes* (voir figure 8.1, à gauche). Les interfaces externes sont accessibles à partir de l'extérieur du composant, alors que les interfaces internes le sont uniquement à partir des sous-composants et sont non visibles de l'extérieur. Les interfaces externes d'un sous-composant sont *exportées* par des *intercepteurs* comme une interface externe du composant parent. Les intercepteurs peuvent introduire des opérations spécifiques entre les invocations entrantes et sortantes d'opérations.

D'autre part, un composant (composite ou primitif) peut être partagé entre plusieurs composants l'englobant. Dans ce cas, il est sujet au contrôle de leurs contrôleurs respectifs. La sémantique exacte de la configuration résultante est déterminée par un composant englobant tous les composants de cette configuration.

Interactions entre composants Afin de définir l'architecture d'une application FRACTAL les composants FRACTAL sont interconnectés pr des *liaisons ou bindings*. La spécification FRACTAL définit des liaisons *primitives* et des liaisons *composites* (voir figure 8.1, à droite). Une liaison primitive est une connexion directe entre une interface cliente et une interface serveur, conforme à la sémantique définie par le langage d'implantation des composants. Elle peut être *normale* lorsque les interfaces cliente et serveur sont externes, et les composants communicants correspondants ont un composant parent direct commun. Elle peut également être *une liaison export*, respectivement *import*, lorsque l'interface cliente est interne, l'interface serveur externe et le composant exposant le service (serveur) est un sous-composant de l'autre (respectivement, l'interface cliente est externe, l'interface serveur est interne et le composant requérant le service (client) est un sous-composant de l'autre). Une liaison composite désigne par contre un chemin de communication entre un nombre arbitraire d'interfaces de composants. Elle représente elle-même un composant FRACTAL construit en combinant des liaisons primitives et des composants ordinaires.

La spécification FRACTAL définit un ensemble de contraintes sur l'occurrence d'opérations fonctionnelles et non fonctionnelles, à savoir:

- Les opérations de contrôle de contenu et de liaisons sont possibles uniquement lorsque le composant est arrêté.
- Un composant peut émettre ou accepter des invocations d'opérations lorsqu'il est démarré.
- Un composant ne peut émettre des invocations d'opérations lorsqu'il est arrêté et doit accepter des invocations à travers les interfaces de contrôle.

La spécification d'une architecture de composants FRACTAL se fait à l'aide du langage FRACTAL ADL (FRACTAL Architecture Descrition Language).

Implantation Le modèle FRACTAL est indépendant du langage d'implantation. A l'heure actuelle, il existe plusieurs implantations de ce modèle dans des langages divers et variés, visant des contextes applicatifs différents. Parmi celles-ci, nous citons Julia[41; 42] et Fractive [22] en Java, Aokell [196] en AspectJ, Think [78] en C et C++ et FracNet [47] en .Net.

8.2.2 FRACTAL ADL

Afin de définir des architectures composant pour les modèles FRACTAL, un langage libre et extensible (open source) a été développé. Il s'agit du langage de description d'architecture FRACTAL ADL [39].

Comme son nom l'indique, une définition d'ADL décrit uniquement l'architecture d'un composant, autrement dit, il ne décrit pas le système résultant à l'exécution. Il est composé d'un ensemble de modules ADL libres et extensibles, où chaque module définit une syntaxe abstraite pour un "aspect" architectural donné, tel que les interfaces, les liaisons, les attributs ou les définitions de contenus.

En fait, FRACTAL ADL est un ADL basé sur le langage XML, utilisé pour décrire les configurations de composants FRACTAL. C'est également le nom d'un environnement proposant une chaîne d'outillages, permettant l'introduction de nouveau outils pendant la génération des processus d'une application à base de composants FRACTAL. Un ADL simplifié, non basé sur XML, peut être aussi utilisé, et de nouveaux ADLs peuvent être créés si nécessaire (en fait, ces ADLs ne font pas parties du modèle de composants FRACTAL lui-même : ce sont uniquement des outils basés dur ce modèle). Dans notre cas, nous travaillons sur un ADL FRACTAL basé XML.

L'ADL FRACTAL est fortement typé. La première étape dans la définition d'une architecture à composants consiste à définir les types des composants. Chaque type de composant doit spécifier que fournit un composant de ce type auquel et que requiert-il des autres composants. La définition d'un composant

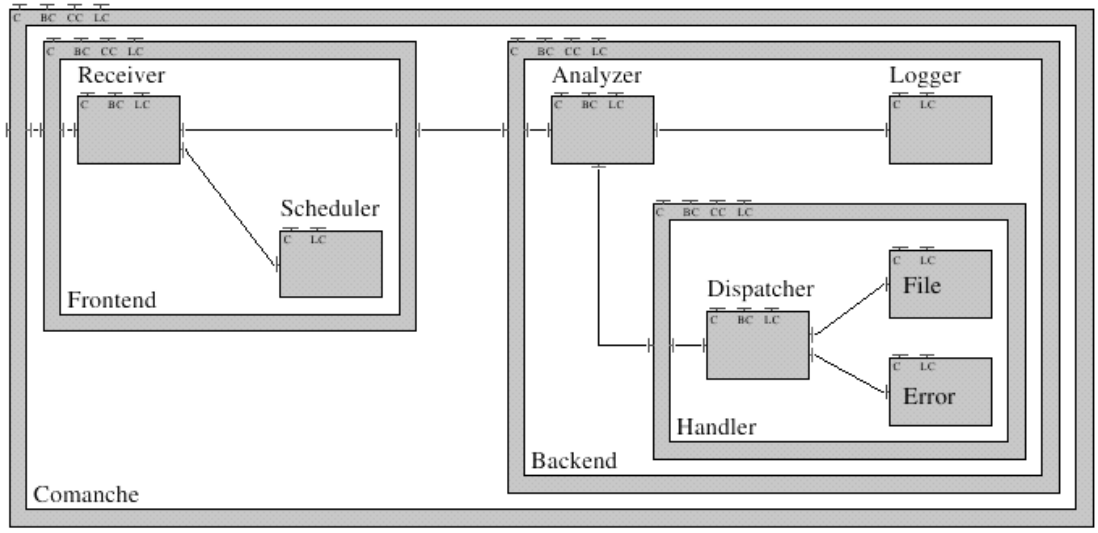


Figure 8.2: Une application FRACTAL: le Serveur Comanche

primitif est faite en spécifiant les interfaces qu'il fournit, les interfaces qu'il requiert et les classes qui implantent le composant.

D'une manière similaire, un composant composite est défini en spécifiant ses interfaces, les sous-composants qu'il contient et les liaisons entre ces sous-composants et avec le composant composite lui-même. Les définitions de composants sont alors établies en étendant les définitions existantes. Le mécanisme d'extension est similaire à l'héritage de classes, i.e., une sous-définition peut ajouter et annuler des éléments dans sa super définition. Ce mécanisme peut être employé pour définir des composants concrets comme des sous-définitions de définitions abstraites de composants.

Une fois l'architecture de l'application définie, elle peut être soit compilée, ce qui génère un code source comprenant une classe Java (pour l'implantation Julia), ou bien elle peut être directement interprétée. Dans les deux cas, le parseur FRACTAL ADL réalise des traitements préliminaires afin de vérifier l'architecture et, en particulier, vérifier qu'aucune liaison n'est absente ou invalide.

Nous présentons dans ce qui suit un exemple d'application FRACTAL et une portion de sa description d'architecture.

8.2.3 Une application FRACTAL : le Serveur Comanche

Nous considérons, tout au long de ce chapitre, un exemple effectif afin d'illustrer notre approche. Cet exemple consiste en un serveur minimal HTTP *Comanche*, utilisé dans [40] pour exemplifier l'implantation et le déploiement d'applications à base de composants FRACTAL. Ce serveur accepte des connexions sur une socket serveur. Pour gérer chaque connexion, il lance un nouveau thread. Chaque connexion est gérée en deux étapes : la requête est analysée et journalisée sur la sortie standard. Puis, le fichier correspondant à la requête est renvoyé au client (ou, au contraire, une erreur est renvoyée si le fichier requis n'a pas été trouvé).

Au niveau le plus haut de l'architecture Comanche, deux services principaux sont identifiés, à savoir : un service récepteur de requêtes (*request receiver*) et un service de traitement de requêtes (*request processor*). Au niveau le plus bas, le service de réception de requêtes utilise un service d'ordonnancement

(*scheduler service*) responsable de la création d'un nouveau thread pour chaque requête. Le service d'ordonnancement peut être implanté de différentes manières : séquentielle, multithreadée, multithreadée avec un pool de threads, etc. Nous supposons dans notre cas qu'il est multithread.

Afin de traiter la requête, un service analyseur de requêtes (*request analyzer*) et un service gestionnaire de journal (*logger service*) sont invoqués avant de répondre effectivement à une requête. Cette réponse est elle-même construite par un service *dispatcher* de requêtes (*request dispatcher*), qui utilise un service de fichiers (*file server*) ou un service gestionnaire d'erreurs (*error manager*). Le service dispatcher expédie les requêtes vers plusieurs gestionnaires de requêtes (*request handlers*) séquentiellement, jusqu'à ce qu'un gestionnaire arrive à traiter la requête (nous pouvons imaginer des gestionnaires de fichiers, des gestionnaires de servlets, etc).

En termes de composants, l'application définit un composant pour chaque service, conduisant ainsi à sept composants primitifs: le récepteur de requêtes , l'analyseur de requêtes, le dispatcher de requêtes, le gestionnaire de fichiers, le gestionnaire d'erreurs, l'ordonnanceur et le gestionnaire de journal (voir figure 8.2).

Certains de ces composants sont encapsulés dans des composants composites : les composants récepteur et ordonnanceur sont compris dans le composant *Frontend*, les composants dispatcher et gestionnaires de fichiers et d'erreur constituent le composant gestionnaire de requêtes. Ce dernier accompagné de l'analyseur et du gestionnaire de Journal constituent le composant *Backend*. The Frontend and Backend composite components are themselves contained in the Comanche server application, which is the highest level component. Les composants composites Frontend et Backend sont eux-mêmes contenus dans l'application du serveur Comanche, qui constitue le composant de plus haut niveau qui est l'application.

Une partie de la description ADL de l'application est donnée ci-après, ainsi qu'une légère partie du code source. La totalité de la définition d'architecture et du code source est donné en annexe.

```
<!- ============== Définition d'architecture ============== ->

<!- ============== Component types ============== ->
<definition name="comanche.RunnableType">
  <interface name="r" signature="java.lang.Runnable" role="server"/>
  </provides>
</definition>
<definition name="comanche.FrontendType" extends="comanche.RunnableType">
  <interface name="rh" signature="comanche.RequestHandler" role="client"/>
</definition>
<definition name="comanche.ReceiverType" extends="comanche.FrontendType">
  <interface name="s" signature="comanche.Scheduler" role="client"/>
</definition>
<definition name="comanche.SchedulerType">
  <interface name="s" signature="comanche.Scheduler" role="server"/>
</definition>
<definition name="comanche.AnalyzerType">
  <interface name="a" signature="comanche.RequestHandler" role="server"/>
  <interface name="rh" signature="comanche.RequestHandler" role="client"/>
  <interface name="l" signature="comanche.Logger" role="client"/>
</definition>
```

```
... etc

<!- ============== Primitive components ============== ->

<definition name="comanche.Receiver" extends="comanche.ReceiverType">
  <content class="comanche.RequestReceiver"/>
</definition>
<definition name="comanche.SequentialScheduler"
            extends="comanche.SchedulerType">
  <content class="comanche.SequentialScheduler"/>
</definition>
<definition name="comanche.Analyzer" extends="comanche.AnalyzerType">
  <content class="comanche.RequestAnalyzer"/>
</definition>
<definition name="comanche.Logger" extends="comanche.LoggerType">
  <content class="comanche.BasicLogger"/>
</definition>

...  etc

<!- ============== Composite components ============== ->

<definition name="comanche.Handler" extends="comanche.HandlerType">
  <component name="rd" definition="comanche.Dispatcher"/>
  <component name="frh" definition="comanche.FileHandler"/>
  <component name="erh" definition="comanche.ErrorHandler"/>
  <binding client="this.rh" server="rd.rh"/>
  <binding client="rd.h0" server="frh.rh"/>
  <binding client="rd.h1" server="erh.rh"/>
</definition>

<definition name="comanche.Backend" extends="comanche.HandlerType">
  <component name="ra" definition="comanche.Analyzer"/>
  <component name="rh" definition="comanche.Handler"/>
  <component name="l" definition="comanche.Logger"/>
  <binding client="this.rh" server="ra.a"/>
  <binding client="ra.rh" server="rh.rh"/>
  <binding client="ra.l" server="l.l"/>
</definition>

--- etc


<!- ============== Code source ============== ->
```

```
/* ============== Component interfaces ============== */

public interface RequestHandler {
  void handleRequest (Request r) throws IOException;
}

public interface Scheduler {
  void schedule (Runnable task);
}

... etc

/* ============== Component implementations ============== */

public class BasicLogger implements Logger {
  public void log (String msg) { System.out.println(msg); }
}

public class SequentialScheduler implements Scheduler {
  public synchronized void schedule (Runnable task)
         { task.run(); }
}

public class FileRequestHandler implements RequestHandler {
  public void handleRequest (Request r) throws IOException {
    File f = new File(r.url);
    if (f.exists() && !f.isDirectory()) {
      InputStream is = new FileInputStream(f);
      byte[] data = new byte[is.available()];
      is.read(data);
      is.close();
      r.out.print("HTTP/1.0 200 OK\n\n");
      r.out.write(data);
    } else { throw new IOException("File not found"); }
  }
}

... etc
```

8.3 Considérations de modélisation

Afin de procéder à la modélisation de notre système, et comme précisé auparavant dans le chapitre 5, nous avons considéré certains points critiques à prendre en compte (voir section 5.5.2 du chapitre 5). Nous reprenons ici les mêmes considérations en précisant les particularités du modèle FRACTAL.

- **Modélisation de configurations stables :** Dans le cadre de cette thèse, nous nous intéressons uniquement aux configurations stables de CBS. Dans le contexte FRACTAL, les configurations transitoires peuvent être provoquées par des opérations de reconfiguration dynamiques, réalisées par l'appel de méthodes définies par les interfaces de contrôle d'un composant.
  De ce fait, comme nous ne traitons pas ce type de configurations, nous avons choisi de ne pas modéliser les interfaces de contrôle des composants FRACTAL.
- **Dépendances avec l'implantation** La modélisation d'un CBS FRACTAL ne peut se faire sans connaître les détails d'implantation. Plusieurs implantations du modèle FRACTAL existent dans la littérature : Julia (l'implantation de référence Java), Fractive [22] , AOKell [196] en Java, et Think [78] l'implantation en C, etc.
  En étudiant ces implantations, nous avons constaté qu'elles diffèrent significativement. Par exemple, Fractive utilise une opération d'invocation asynchrone (tardive), qui permet au client de poursuivre son activité locale jusqu'à ce qu'il ait besoin du résultat retourné par le service qu'il a précédemment invoqué. Inversement, Julia et Think utilisent un appel classique synchrone de méthode (appel bloquant). Dans notre cadre d'étude, nous modélisons l'implantation FRACTAL Julia.
- **Couleurs** La modélisation des entités d'une application FRACTAL englobe la modélisation des entités requêtes, méthodes et paramètres, données, threads composants, etc. Nous pouvons illustrer ces entités par les couleurs de base utilisées dans notre exemple de serveur Comanche. Nous considérons les sockets, les requêtes HTTP, les fichiers et les threads comme les classes de couleurs de base. Précisément, nous utilisons les classes suivantes: *UC* modélise les requêtes utilisateur HTTP, *IDS* modélise les threads ordonnancés, *IDL* est la classe de base des requêtes de journal, *IDF* est celle des requêtes de fichiers et *IDE* modélise les erreurs identifiables.

À l'optique de ces considérations, nous traduisons une application basée FRACTAL dans le contexte SWN en appliquant notre méthode d'analyse.

8.4 Traduction vers le modèle SWN et analyse de performances

8.4.1 Particularités liées au modèle FRACTAL

Le modèle FRACTALse caractérise par quelques particularités qui peuvent se retrouver dans d'autres modèles à composant :

- Il est hiérarchique, emboîtant les composants dans d'autres composites.
- Il offre des opérations de contrôle à travers ses interfaces de contrôle constituant la membrane.
- Il permet un style unique de communication qu'est l'invocation de méthode, ce qui facilite l'analyse structurée et élargit son champs d'application (voir plus loin).

Ces particularités doivent être prises en compte lors de l'analyse.

8.4.2 Directives générales

L'instanciation de notre méthode d'analyse structurée aux CBS FRACTAL se fait en réalisant la modélisation du CBS et génération de son G-SWN, puis en appliquant l'analyse structurée de performances.

8.4.2.1 Génération du G-SWN

La construction du G-SWN se fait à partir de la description FRACTAL ADL et des codes sources des composants, telle que nous l'avons présentée dans le chapitre 5.

Comme le modèle FRACTAL est hiérarchique, on suit la procédure suivante, à l'image de l'algorithme présenté en section 5.5.6 pour la modélisation d'un composant composite.

1. On part du plus bas niveau de l'architecture et on modélise les composants primitifs, obtenant un ensemble de CC-SWNs. Les interfaces fonctionnelles sont modélisées suivant les règles systématiques que nous avons introduit. Notons que le modèle FRACTAL définit uniquement la communication par invocation de service (méthode).

   De plus, comme l'implantation Julia se base sur une communication synchrone entre les composants, les modèles d'interface proposés par les règles de traduction de 1 à 4 correspondent exactement, ce que nous avons utilisé.

2. Une fois les composants primitifs modélisés, on remonte dans les niveaux : pour chaque composite, on construit son CC-SWN en fusionnant les éléments d'interfaces des CC-SWNs correspondant aux sous-composants interconnectés, ainsi de suite jusqu'à atteindre le plus haut niveau, qui est celui de l'application.

8.4.2.2 Analyse du CBS

L'approche d'analyse structurée est appliquée aux CBS FRACTAL. Comme le modèle est hiérarchique et comme l'analyse se fait sur une configuration plate de SWNs, il est nécessaire d'abord d'"aplatir" la configuration de modèles CC-SWNs et CI-SWNs obtenus durant la première étape. Ceci consiste à garder uniquement les modèles de composants primitifs, tout en translatant les connexions entre composants composites au niveau de leurs composants primitifs, jusqu'à monter au plus haut niveau de l'application.

Dans le contexte de Julia, les communications entre composants sont traduites en des compositions synchrones de SWNs. Comme il n'y a pas d'interactions basées événements, il ne peut y avoir des compositions multisynchronisées de SWNs. Il est donc possible d'analyser diverses configurations d'applications FRACTAL. Seule la dernière condition énoncée par le théorème 6.4 doit être satisfaite. Cette condition se traduit, par exemple au niveau de la programmation, par le fait que les entités à synchroniser avec d'autres composants ne doivent pas être dupliquées, par exemple avec des processus ou threads concurrents dans un même composant.

Dans la suite, nous illustrons l'application de la méthode sur l'exemple du serveur Comanche.

8.4.3 Modéliser l'application FRACTAL

La construction du G-SWN commence par la modélisation de composants primitifs. Illustrons la construction du CC-SWN d'un composant primitif avec le composant Analyseur de l'application Comanche. Le code d'implantation de ce composant est donné ci-dessous.

```
public class RequestAnalyzer implements RequestHandler
{
  private Logger l;
  private RequestHandler rh;
  // functional concern
  public void handleRequest(Request r) throws IOException
  { r.in = new InputStreamReader(r.s.getInputStream());
    r.out = new PrintStream(r.s.getOutputStream());
    String rq = new LineNumberReader(r.in).readLine();
    l.log(rq);
```

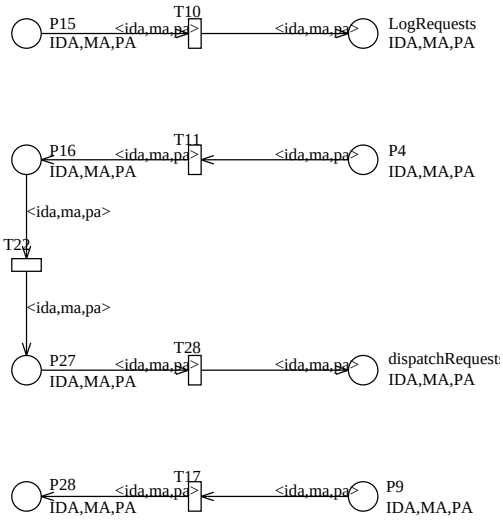


Figure 8.3: Modèle SWN du composant Analyseur

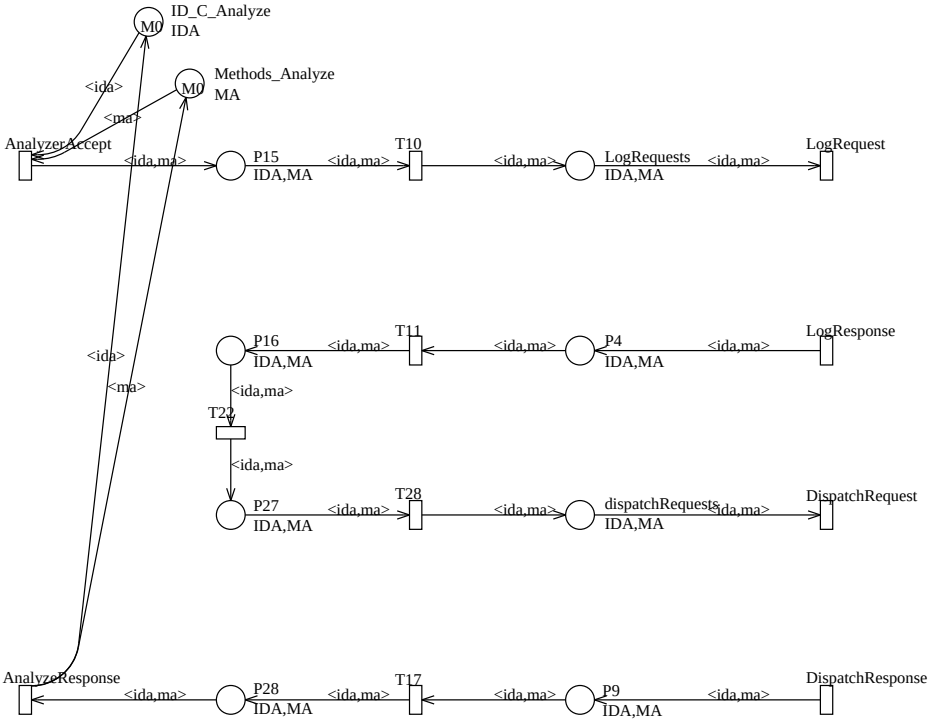


Figure 8.4: Modèle C-SWN du composant Analyseur

```
    if (rq.startsWith("GET "))
    { r.url = rq.substring(5, rq.indexOf(' ', 4));
      rh.handleRequest(r);
      }
    r.out.close();
    r.s.close();
    }
}
```

L'analyseur reçoit des requêtes sur son interface serveur utilisant deux classes de couleurs de base *IDA* et *MA*. Ces deux classes modélisent respectivement les requêtes analysées identifiées, ainsi que les méthodes associées invoquées avec leurs paramètres. En plus de l'interface serveur, l'analyseur invoque deux opérations à travers deux interfaces clientes :

1. une opération d'enregistrement (voir l'opération *l.log(rq)* dans le code), et
2. une opération de traitement (voir l'opération *rh.handleRequest(r)* dans le code).

Tout d'abord, nous modélisons le noyau du composant. Ensuite, nous ajoutons les modèles d'interfaces client et serveur, introduisant ainsi les transitions de demande et fin de requête pour chaque interface. Nous obtenons le C-SWN de la figure 8.4. Enfin, celui-ci est complété avec des places et ses arcs pour obtenir le CC-SWN de la figure 8.5 (en bas).

En suivant la même démarche, on obtient les figures 8.5, 8.6, 8.7 et 8.8 qui présentent les CC-SWNs des septs composants:

1. Le récepteur expose une interface serveur qui utilise deux classes de couleurs de base *IDC* et *M*. Ces deux classes modélisent respectivement les clients identifiés et les méthodes avec leurs paramètres. Ce composant crée une tâche pour chaque requête reçue, et l'envoie à l'ordonnanceur en créant un thread associé. Une fois le thread lancé, il invoque une requête d'analyse. Ainsi, le récepteur a deux interfaces clientes : une pour demander à ordonnancer la requête et l'autre pour invoquer l'analyse de requête.
2. L'ordonnanceur a uniquement une interface de type serveur. Il utilise deux classes de couleurs de base *IDS* et *MS*, modélisant respectivement les threads ordonnancés associés aux requêtes et les méthodes invoquées avec leurs paramètres.
3. L'analyseur: déjà expliqué ()voir ci-dessus).
4. Le dispatcher traite les opérations reçues sur son interface serveur, en les envoyant soit vers le gestionnaire de fichiers ou le gestionnaire d'erreurs. Deux classes de couleurs de base *IDD* et *MD* sont utilisées lorsqu'une opération est reçue, modélisant respectivement les requêtes envoyées et les méthodes correspondantes invoquées avec leurs paramètres. Cet envoi de requêtes est réalisé en invoquant des opérations de gestion via ses deux interfaces clientes.
5. Les gestionnaires de journal, de fichiers et d'erreur exposent chacun une interface serveur. Les classes de couleurs de base impliquées dans ces interfaces sont :
   - *IDL* et *ML*, représentant respectivement les requêtes de journalisation et les méthodes du journal avec leurs paramètres, dans le gestionnaire de journal;
   - *IDF* et *MF* modélisant les requêtes d'accès aux fichiers et les méthodes d'accès aux fichiers avec leurs paramètres, dans le gestionnaire de fichiers; et
   - *IDE* et *ME* identifiant les types d'erreurs et les méthodes de traitement d'erreurs avec leurs paramètres.

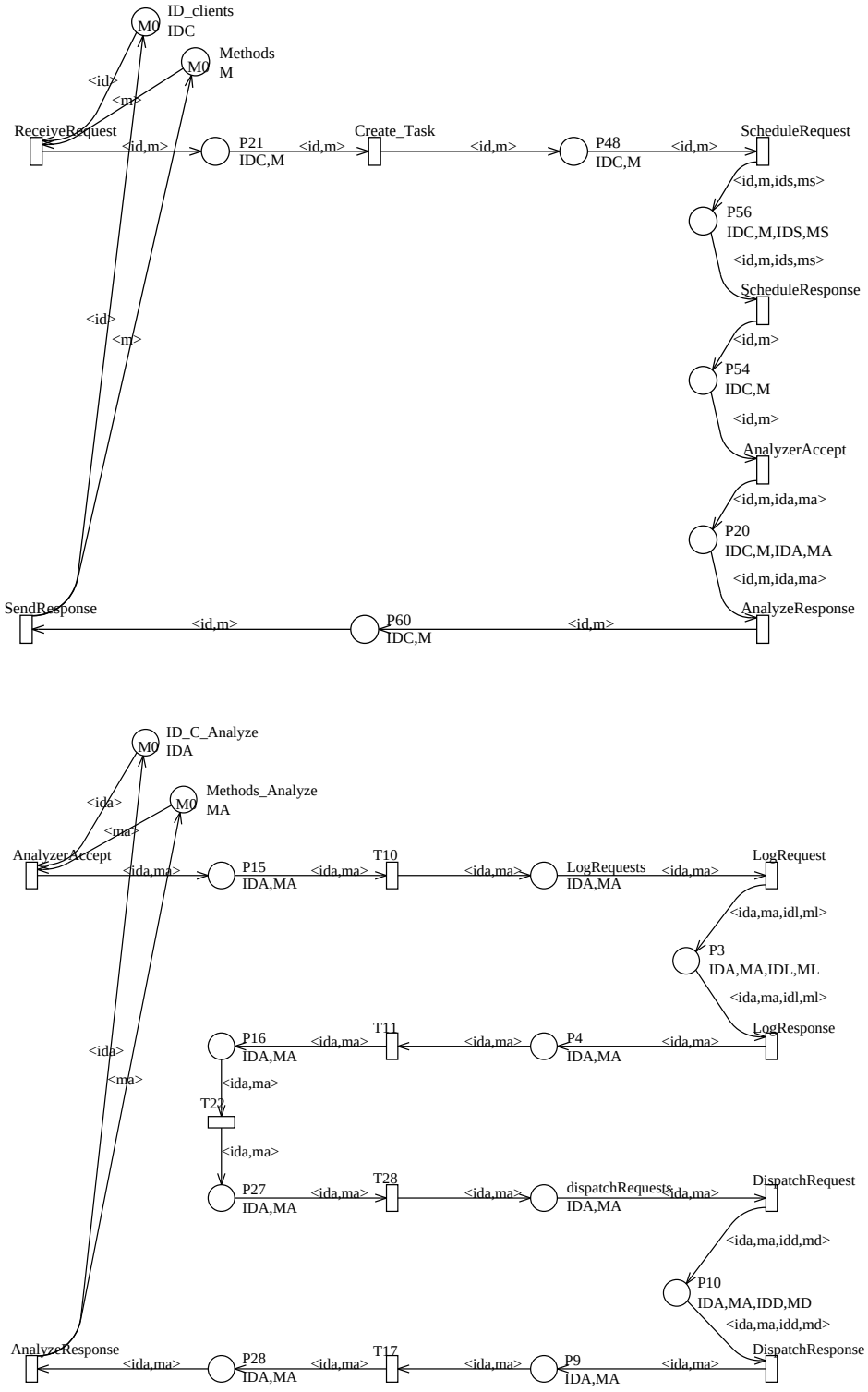


Figure 8.5: CC-SWNs des composants récepteur (haut) et analyseur (bas)

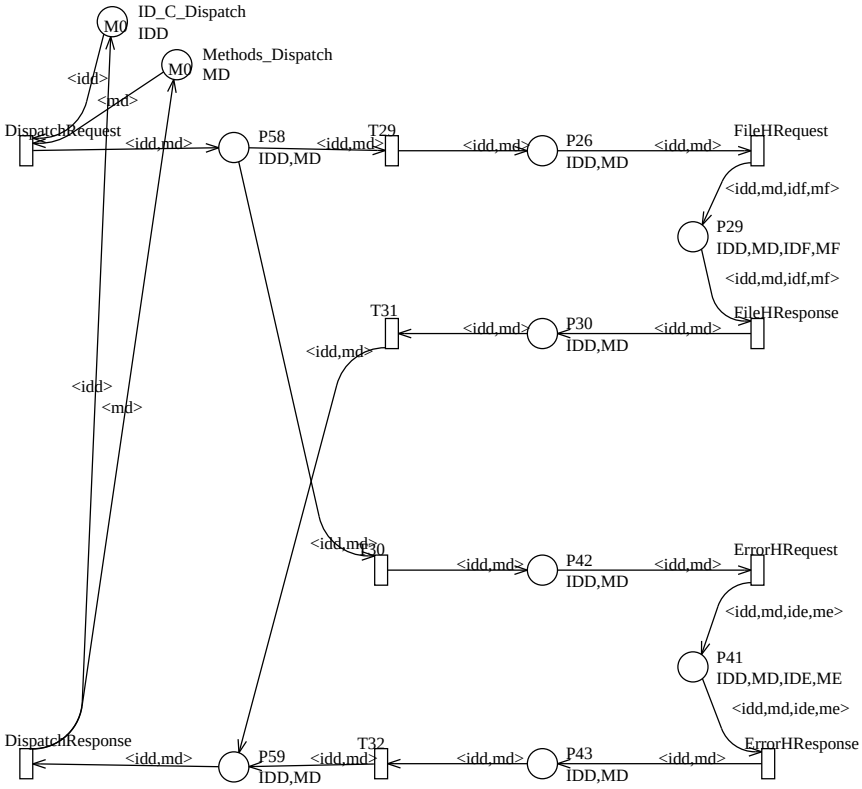


Figure 8.6: CC-SWN du composant dispatcher

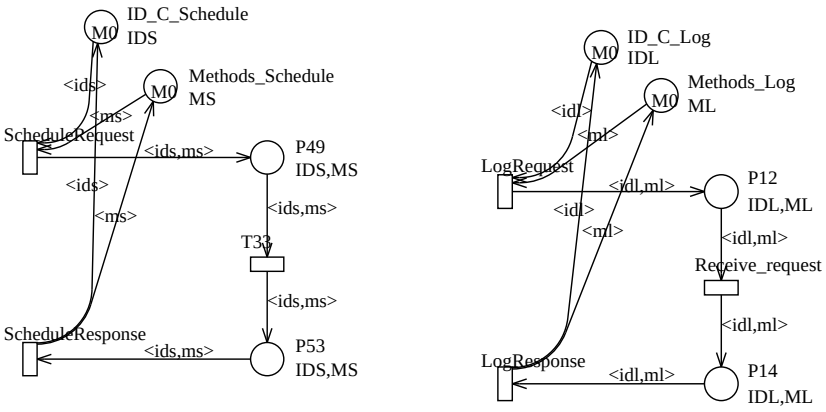


Figure 8.7: CC-SWNs des composants ordonnanceur (haut) et du gestionnaire de journal (bas)

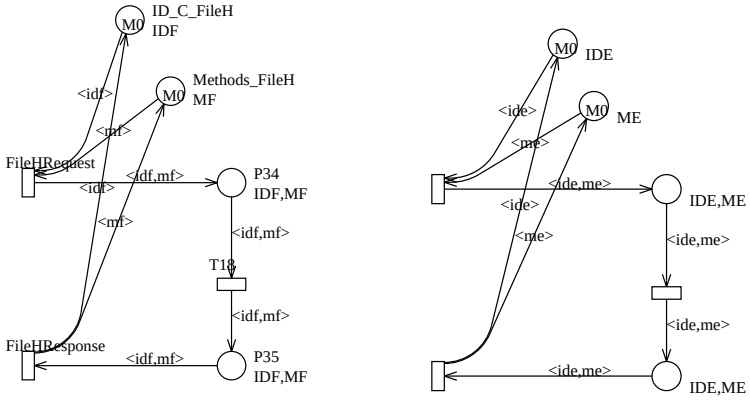


Figure 8.8: CC-SWNs des composants gestionnaire de fichier (haut) et gestionnaire d'erreur (bas)

Notons qu'il n'y a pas, dans cette application, d'interface serveur à laquelle sont connectés plusieurs clients, de sorte que l'identité des composants clients n'est pas nécessaire à modéliser dans le CC-SWN.

Une fois les composants primitifs modélisés, nous pouvons construire les CC-SWNs des composants composites : le CC-SWN du composant gestionnaire de requêtes, ensuite celui du composite Frontend et Backend. Enfin, en connectant les modèles CC-SWNs obtenus des composites et en fermant le modèle final, nous obtenons le G-SWN donné dans la figure 8.9.

8.5 Conclusion

Nous avons présenté l'instanciation de notre méthode d'analyse de performances aux systèmes basés composants FRACTAL, en particulier la phase de modélisation.

Globalement, nous n'avons pas eu besoin de modifier ou changer certains points dans la méthode présentée, puisque les particularités du modèle FRACTAL pouvant être également des caractéristiques d'autres modèles de composant, ont été considérées. Ce qui montre la généricité de notre méthode.

L'étape d'évaluation des performances est présentée dans la suite, dans le chapitre 11.

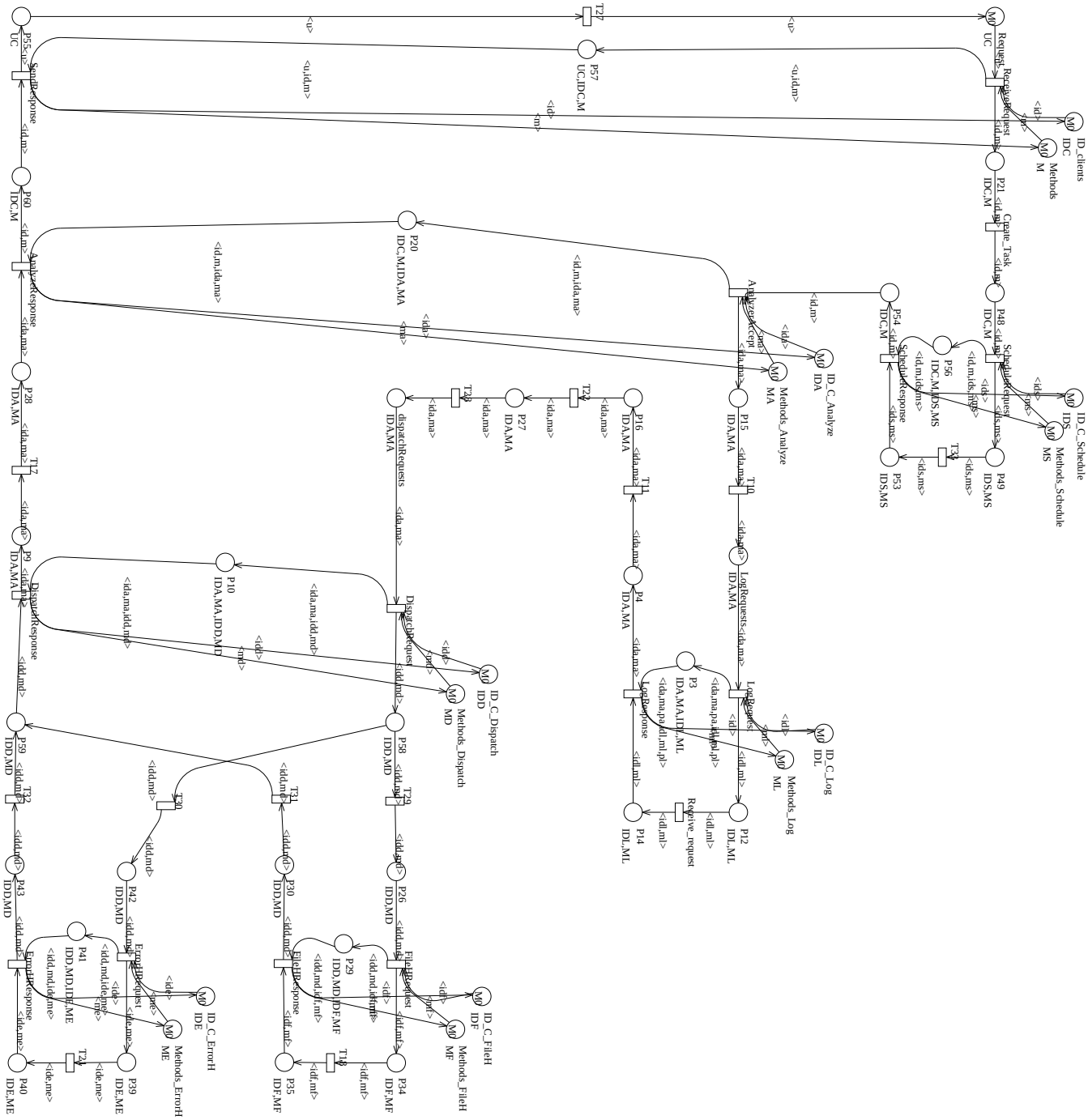


Figure 8.9: The G-SWN of the Comanche application

CHAPITRE 9
Analyse des CBS CCM

9.1 Introduction - Objectifs

Parmi les modèles de composant leaders en industrie, le modèle CCM (*CORBA Component Model*) constitue un modèle ouvert, non restreint à un langage de programmation et permettant de construire des applications à base de composants hétérogènes. Il spécifie essentiellement une structure de composant et un environnement d'exécution pour les composants.

Nous nous sommes principalement intéressés à ce modèle à la vue des deux types d'interaction offerts par les composants : l'invocation de service et la communication par événements. De plus, un second point fait la différence avec d'autres modèles tels que le modèle FRACTAL : l'utilisation d'un conteneur comme environnement d'exécution d'un composant, exposant un ensemble de services non fonctionnels.

Pour ces raisons, nous présentons ici une application de notre approche d'analyse aux systèmes basés sur le modèle CCM. Nous verrons que l'ensemble des caractéristiques du modèle CCM est déjà pris en compte dans notre méthode.

9.2 Le modèle CCM

Le modèle CCM [169] est un modèle de composants indépendant des systèmes d'exploitation et des langages de programmation, défini en 2001 par l'OMG en vue d'ajouter des composants logiciels dans l'environnement CORBA existant (*Common Object Request Broker Architecture*) [164]. C'est une partie clé du standard CORBA 3.0, Principalement, CCM permet le déploiement de composants dans un environnement distribué (interconnexion de composants distribués sur différents serveurs).

9.2.1 Caractéristiques

Constitution d'un composant Le modèle CCM est un modèle plat (sans hiérarchie). Dans ce modèle, un composant est considéré comme une entité d'implémentation, principalement décrite par des attributs et des interfaces typées dites *ports* : Les attributs sont des valeurs nommées permettant entre autres de configurer le composant à l'aide d'opérations d'accès (get) et de modification (set). Les interfaces (figure 1.3), décrites en langage CORBA IDL 3.0 (*Interface Definition Language*) se caractérisent par un mode de communication et sont de quatre sortes : les *facettes* et les *réceptacles* en mode de communication synchrone, et les *sources* et les *puits* en mode basé événement ou mode asynchrone. Une *facette* est une interface identifiée par le mot clé *provides*, déclarant un ensemble de services (opérations) offerts et acceptant des invocations point à point d'autres composants sur ces opérations. Un *réceptacle* est une interface identifiée par le mot *uses*, déclarant les dépendances ou services requis par le composant. Une *source d'événements* est une interface distonguée par le mot-clé *emit* ou *publishes*, traduisant un envoi de messages asynchrone 1-vers-1 ou 1-vers-n (cas diffusion d'un événement à plusieurs). Enfin, un *puits*

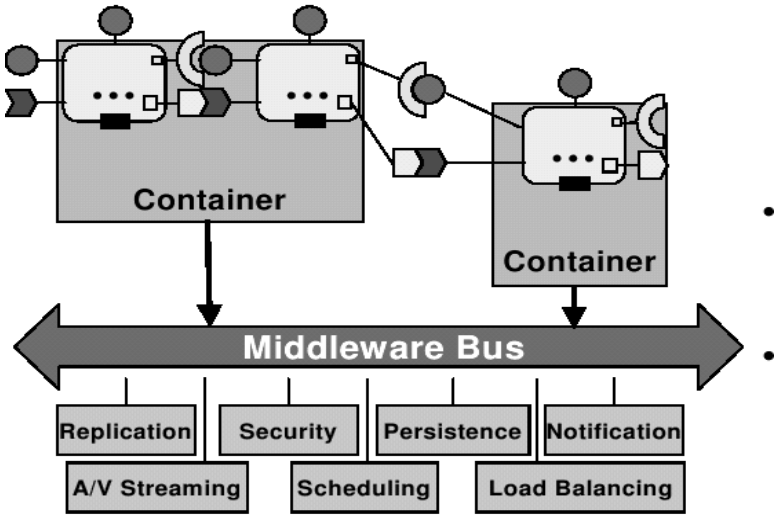


Figure 9.1: Conteneur dans le modèle CCM

d'événements, reconnu par le mot-clé *consumes*, reçoit des notifications d'événements à partir d'une ou plusieurs sources. D'autre part, la spécification CCM ne fait aucune distinction entre les interfaces fonctionnelles et les interfaces de contrôle.

Un composant CCM peut être implanté dans n'importe quel langage de programmation, à partir du moment où des règles de projection depuis l'OMG IDL ont été définies. Il est localisé et exécuté dans un *conteneur* (voir figure 9.1) qui lui offre un environnement d'exécution (çàd un espace mémoire et un flot d'exécution). Il lui fournit également l'accès à un ensemble de services tels que le service de persistence, la niotification d'événements, le service de gestion des transactions, le service de sécurité, etc. chaque conteneur est responsable de l'initialisation des instances de types de composant qu'il gère, contrôlant ainsi leur cycle de vie, leur connexion à d'autres composants et aux services communs de middleware, incluant les services d'évenement et des canaux d'événements. Un conteneur CCM est spécifique à un type de composant. L'exécution d'un conteneur est elle-même faite dans le cadre d'un serveur de processus dit *serveur de composants (component server).*

Interactions entre composants L'assemblage de composants CCM se fait d'une manière explicite sur la base de leurs interfaces, formant ainsi une application (*assembly*) CCM. Des interactions de type appel de méthode ou événements sont possibles entre les composants CCM. Les communications par méthode sont réalisées par les interfaces facettes et réceptacles. Les communications basées événements suivent le modèle *push publisher/subscriber* [169], compatible avec le service de notification CORBA [168]. Ce service de notification introduit des *canaux d'événements* qui acheminent les messages d'événement entre les sources et les puits.

Pour gérer la notification d'événements, CCM définit deux catégories de sources : ceux qui communiquent avec un unique subscriber, dit *émetteurs*, et ceux qui peuvent communiquer avec plusieurs subscribers, connus sous le nom de *publishers*. Tous les deux sont implémentés en utilisant les canaux d'événement et la communication typée (événements typés). Un canal supporte soit une seule interconnexion entre un publisher et plusieurs subscribers, ou bien plusieurs interconnexions émetteur-subscriber.

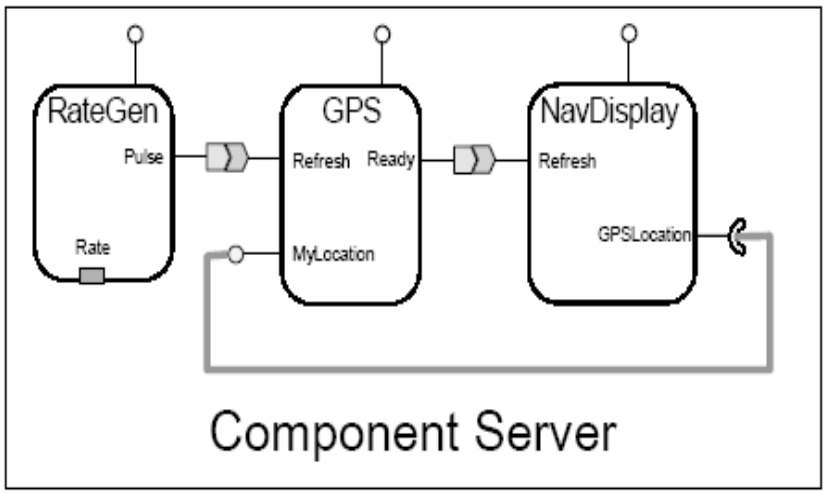


Figure 9.2: Système de contrôle avionique

L'interconnexion des composants construit une application CCM, en définissant une entité *assembly* qui groupe les composants et définit des métadonnées décrivant les composants. Les métadonnées d'un composant décrivent ses caractéristiques et les propriétés qu'elles requièrent (telles que leurs dépendances). Les assemblages CCM sont définis en utilisant des modèles de schéma XML (XML Schema templates), qui fournissent un mécanisme indépendant de l'implémentation pour la description des propriétés de composant et la génération des configurations par défaut des composants CCM.

Pour implanter effectivement une application CCM, des outils de développement et d'exécution existent tels que *K2-CCM, Cadena, CCM container framework* et *CIF (Component Implementation Framework)*.

Nous présentons dans ce qui suit un exemple d'application CCM et une partie de la description de son architecture.

9.2.2 Exemple d'application CCM : Application de contrôle avionique

Nous illustrons l'application de notre approche sur une application industrielle typique : un système de contrôle avionique (figure 9.2) présentée dans [217]. L'application consiste en trois composants : un générateur d'mpulsions (*Rate Generator*) qui envoie des impulsions d'événements périodiques vers un subscriber, un capteur de position ou en d'autres termes un *GPS*, et une unité d'affichage *Displaying device*. Le *GPS* raffraîchit les coordonnées stockées en mémoire cache, disponibles à travers une interface facette appelée *MyLocation*. Ensuite, le *GPS* notifie ses propres subscribers avec des événements de type "*Ready*", informant que les coordonnées sont prêtes à être consommées dans le cache. Un seul subscriber est notifié pour ces événements "Ready" : Il consiste en l'unité d'affichage (*Displaying device*) qui lit les coordonnées courantes à travers son interface réceptacle *GPSLocation*, puis met à jour l'affichage.

Un partie de la description d'architecture de l'application est donnée ci-après. Le code d'implémentation de quelques composants est décrit en annexe.

```
<! Assembly descriptors associate components with implementations-->
<! in software packages defined by softpkg descriptors (*.csd) files-->
```

```
<componentfiles>
<componentfile id="com-RateGen">
<fileinarchive name="RateGen.csd"/>
</componentfile>
<componentfile id="com-GPS">
<fileinarchive name="GPS.csd"/>
</componentfile>
<componentfile id="com-Display">
<fileinarchive name="NavDisplay.csd"/>
</componentfile>
</componentfiles>
<! Instantiating component homes/instances -->
    ...
<connections>
 <connectinterface>
<usesport>
<usesidentifier>GPSPosition</usesidentifier>
<componentinstantiationref idref="a_NavDisplay"/>
</usesport>
<providesport>
<providesidentifier>MyLocation</providesidentifier>
<componentinstantiationref idref="a_GPS"/>
</providesport>
</connectinterface>
<connectevent>
<consumesport>
<consumesidentifier>Refresh</consumesidentifier>
<componentinstantiationref idref="a_GPS"/>
</consumesport>
<publishesport>
<publishesidentifier>Pulse</publishesidentifier>
<componentinstantiationrefidref="a_RateGen"/>
</publishesport>
</connectevent>
 ...
</connections>
```

9.3 Considérations de modélisation

Les propriétés du modèle CCM sont prises en compte lors de la modélisation en considérant les points suivants :

- **Modélisation de configurations stables :** Les services non fonctionnels offerts par un conteneur et relatifs aux phases d'initialisation et de reconfiguration résultent en une configuration transitoire. Comme nous nous concentrons sur des architectures fixes “stables”, nous ne modélisons pas ces services.

- **Modélisation des événements et des conteneurs** En ce qui concerne la modélisation des événements et des conteneurs, nous adoptons les mêmes choix expliqués dans le chapitre 5, sans aucune modification .
- **Couleurs** La modélisation des entités d'une application CCM inclut notamment la modélisation des entités requêtes, méthodes et paramètres, données, threads composants, événements, données d'événements, etc. Nous illustrons cela par les classes de base utilisées dans notre application exemple :
  - *PT1* modélise les threads du composant *RateGen*.
  - *ST1* modélise la classe de couleur de base des threads du composant *GPS*.
  - *ST2* moélise la classe de base des threads du composant *NavDisplay*.
  - *E* et *NE* modélise les classes des couleurs d'événements.

Nous passons maintenant à la construction des modèles SWN d'une application basée CCM, en appliquant notre méthode d'analyse.

9.4 Traduction vers le modèle SWN et analyse de performances

9.4.1 Particularités liées au modèle CCM

Le modèle CCM se caractérise principalement par les spécificités suivantes :

- Il est plat, sans aucun emboîtement des composants.
- Des services non fonctionnels sont offerts à travers le conteneur.
- Il offre les deux types de communication : l'invocation de service à travers les interfaces facettes et réceptacles, et la communication basée événements.

9.4.2 Directives générales

L'instanciation de notre méthode d'analyse aux CBS CCM s'opère sans modification particulière.

9.4.2.1 Génération du G-SWN

La modélisation d'un CBS CCM, se fait en modélisant les composants, les conteneurs, puis en connectant les modèles obtenus pour générer le G-SWN.

En identifiant les mots-clés du langage IDL3 permettant di'dentifier les interfaces, nous personnalisons l'algorithme de construction du CC-SWN d'un composant CCM par l'algorithme qui suit.

Le reste des algorithmes reste identique, sauf pour celui de la construction du G-SWN qui se fait pour un CBS à un unique niveau d'hiérarchie.

9.4.2.2 Analyse du CBS

La méthode d'analyse structurée est appliquée aux CBS CCM. Le G-SWN construit est une composition mixte synchrone et asynchrone de SWNs, étant donné que les deux types interactions (par invocation de méthode et par événements) sont supportées. Par conséquent, toutes les conditions énoncées par le théorème 6.4 doivent être satisfaites.

Nous illustrons dans ce quit l'application de la méthode sur l'exemple du système de contrôle avionique.

ALGORITHME Construction du CC-SWN d'un composant CCM
BEGIN

1. Analyser le code source du composant, fixer un niveau de détail pour la modélisation.
2. Pour chaque ensemble de méthodes liées à une facette définie par le mot-clé *provides*:
   - Modéliser la facette en utilisant la règle de traduction 1.
   - Modéliser les activités internes à la facette, selon le niveau de détails internes requis .
3. Pour chaque réceptacle identifié par le mot clé *uses*, modéliser l'interface en utilisant la règle de traduction 2.
4. Compléter les modèles d'interfaces facettes et réceptacles. éventuellement en utilisant les règles de traduction 3 et 4.
5. Pour chaque source d'événements identifié par le mot-clé *"publishes* ou *emits*, traduire en SWN en utilisant la règle de traduction 5 (ou 9 dans le cas d'un mode control-push data-pull).
6. Pour chaque puits d'événements identifié par le mot-clé *consumes* :
   - Traduire en SWN en utilisant les règles de traduction 7, 8, 9 ou 10 selon le cas qui se présente.
   - Modéliser les activités locales du subscriber, selon le niveau de détails internes requis.
   - Modéliser les activités internes au gestionnaire d'événement selon le niveau de détails internes requis, dans le cas d'un traitement local au subscriber.
7. Modéliser chaque canal d'événements (si plusieurs) suivant la règle de traduction 6.
8. Compléter le modèle obtenu pour chaque canal d'événement en utilisant la règle de traduction 11, si plusieurs publishers et/ou subscribers interviennent.
9. Si toute autre fonction (traitement) est appelée d'une manière interne à une interface (lors de l'appel d'une méthode de l'interface), modéliser les activités associées à la fonction.

END

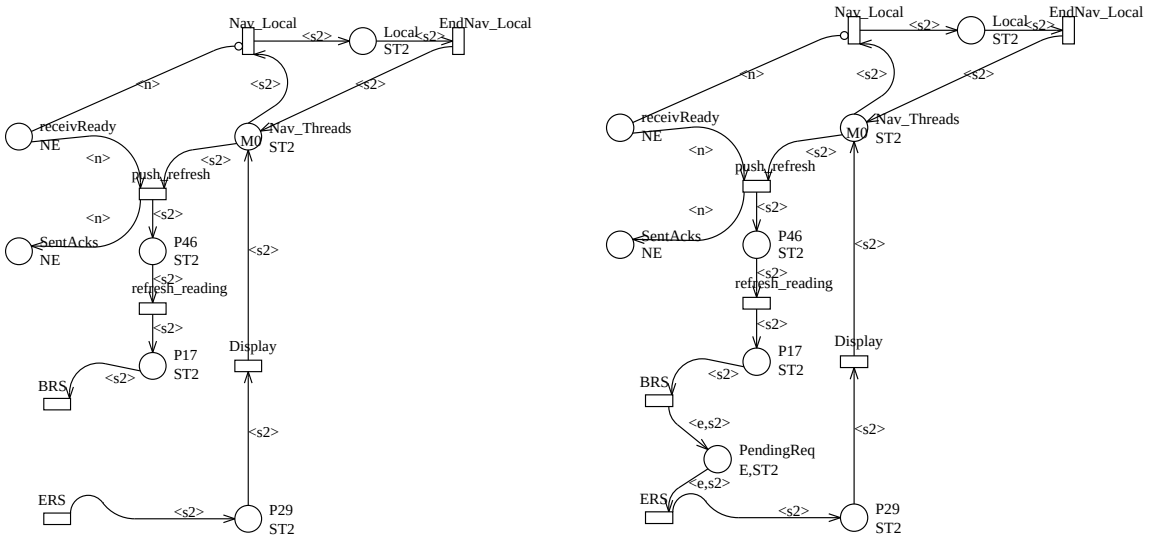
9.4.3 Modéliser l'application CCM

Afin d'illustrer la construction du CC-SWN d'un composant CCM, nous considérons le composant unité d'affichage (*NavDisplay*) de l'exemple du système de contrôle avionique. Nous partons de son code d'implémentation donné ci-dessous :

```
valuetype tick :Components:: EventBase
 { public rateHz rate; };
eventtype tick { public rateHz rate; };
interface position { long get_pos(); };
interface tickConsumer:Components:: EventConsumerBase
 { void push_tick(in tick the_tick);
   };
interface NavDisplay : Components:: CCMObject
 { void connect_GPSLocation(in position c);
   position disconnect_GPSLocation();
   position get_connection_GPSLocation ();
   tickConsumer get_consumer_Refresh();
    };
component NavDisplay
 { uses position GPSLocation;
   consumes tick Refresh;
   };
class NavDisplay_Executor_Impl: public virtual CCM_NavDisplay,
public virtual CORBA:: LocalObject
{ public:
    virtual void push_Refresh(tick *ev)
    { this->refresh_reading();
      }
    virtual void refresh_reading(void)
    { position_var cur = this->context_->get_connection_GPSLocation();
      long coord = cur->get_pos();
      };
   };
```

A partir de la définition des interfaces du composant *NavDisplay*, nous dérivons d'abord une modélisation pour l'interface réceptacle *GPSLocation*. *ST2* est la classe de couleur de base modélisant les threads du composant *NavDisplay*. L'interface facette *MyLocation* du composant *GPS* expose une méthode *get_pos()* (voir les détails d'implémentation du composant *GPS* dans l'annexe A). Le modèle associé au réceptacle est donc donné par les transitions *BRS* et *ERS*.

Nous modélisons ensuite l'interface puits d'événement pour laquelle un gestionnaire d'événement *refresh_reading* est associé. Ce gestionnaire est déclenché lorsque le composant *GPS* émet des événements vers le composant *NavDisplay* (la notification se fait en appelant l'opération *push_refresh*). Il obtient d'abord une référence vers l'interface facette à laquelle le réceptacle *GPSLocation* est connecté (en utilisant la méthode $\check{}context\_->get\_connection\_GPSLocation()\acute{}$). Ensuite, il invoque la méthode *get_pos()*. Ainsi, nous modélisons le gestionnaire avec plusieurs transitions: une transition de départ *refresh_reading*, un couple de transitions du réceptacle *BRS, ERS* modélisant la requête invoquée à la



(a) Modèle C-SWN

(b) Modèle CC-SWN

Figure 9.3: Les modèles C-SWN et CC-SWN du composant NavDisplay

facette du composant *GPS* et enfin une transition *EndHandler* qui termine le gestionnaire.

Comme il n'y a pas d'autres fonctions internes à l'implémentation du composant *NavDisplay*, nous obtenons le C-SWN de la figure 9.3(a). Ce C-SWN est complété en utilisant la règle de traduction 3, conduisant ainsi au CC-SWN de la figure 9.3(b).

Notons que nous ne modélisons pas les fonctions associées aux connexions et déconnexions aux interfaces facettes, ainsi que les fonctions de souscription et désouscription aux événements du fait qu'elles donnent lieu à des configurations non stables.

Les autres composants sont modélisés de façon similaire. Les figures 9.4 et 9.5 présentent les CC-SWNs obtenus. Dans ces modèles, *PT1* et *ST1* sont les classes de couleurs de base modélisant respectivement les threads des composants *RateGen* et *GPS*. Nous remarquons que la communication entre les composants *GPS* et *NavDisplay* constitue un sécnario de type control-push data-pull. Les canaux d'événement entre les composants *RateGen* et *GPS* et entre les composants *GPS* et *NavDisplay* sont également modélisés en respectant la règle de traduction 6.

Une fois les CC-SWNs construits, nous passons à la modélisation des conteneurs et la construction du G-SWN. Les trois composants sont contenus dans un seul conteneur et n'invoquent pas de services conteneur. Nous ne rajoutons donc pas de modèle du conteneur, pour ne pas ajouter de complexité à notre modèle final. Les composants CC-SWNs sont ensuite interconnectés par fusion des éléments d'interface. Le modèle global G-SWN est alors obtenu en fermant l'interface externe *rate_control* du composant *RateGen*. Cette interface fournit deux méthodes *start* et *stop*, qui lance et arrête effectivement l'application globale. Le modèle G-SWN de l'application est donné par la figure 9.6.

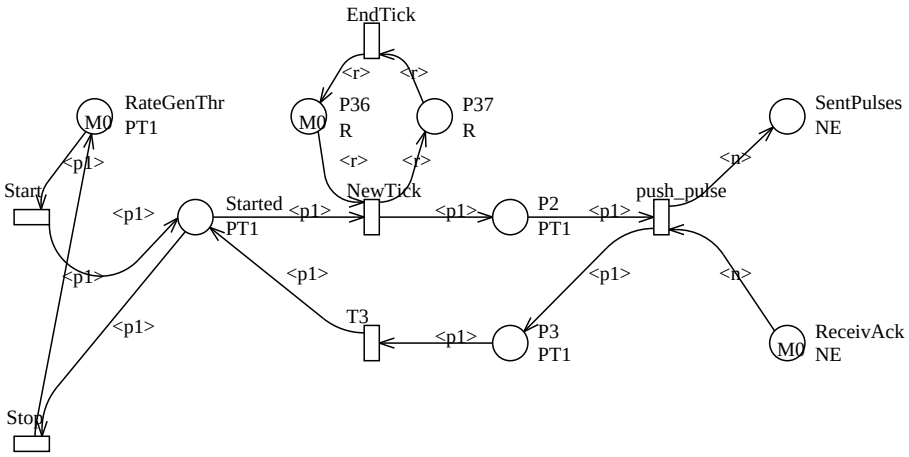


Figure 9.4: Modèle CC-SWN du composant RateGen

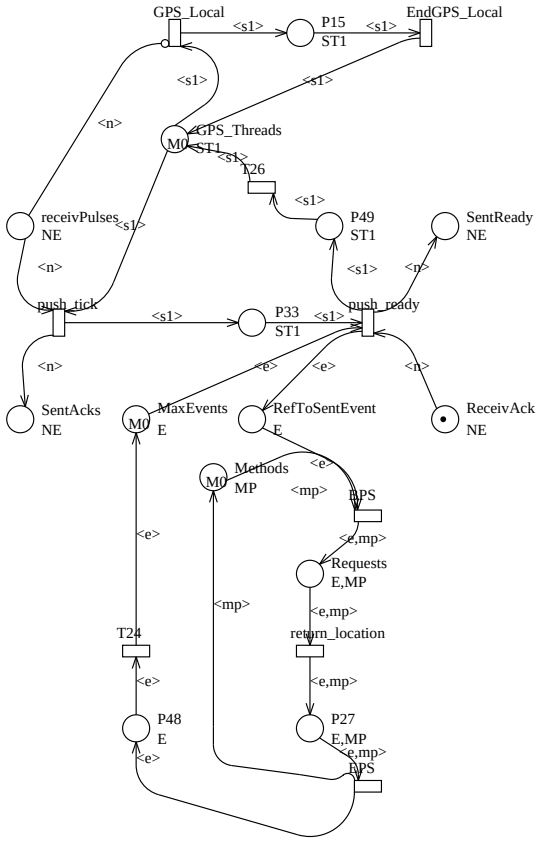


Figure 9.5: Modèle CC-SWN du composant GPS

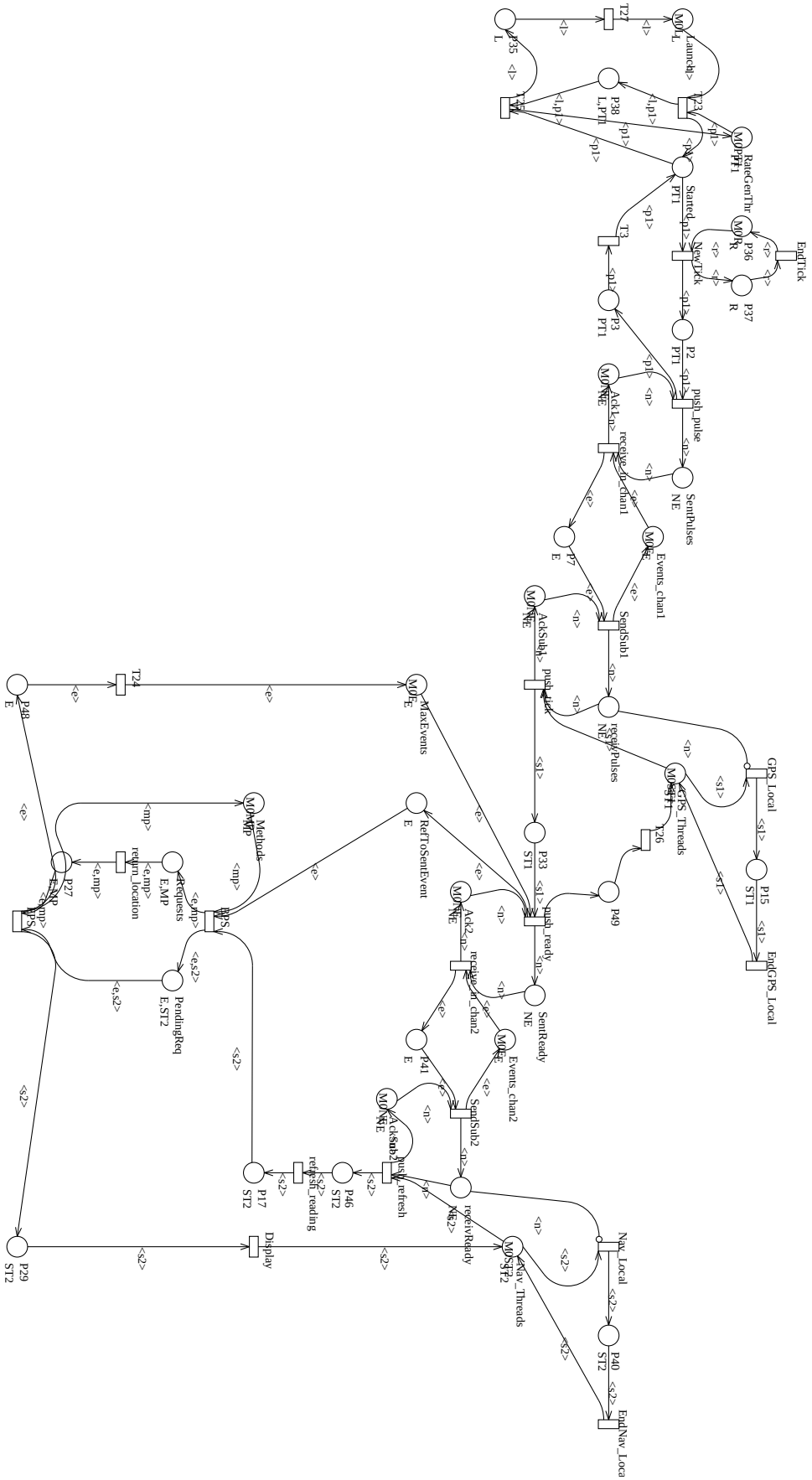


Figure 9.6: Le modèle G-SWN du système de contrôle avionique

9.5 Conclusion

Ce chapitre a fait l'objet de l'instanciation de notre méthode d'analyse de performances aux systèmes basés composants CCM, en particulier la phase de modélisation.

Globalement, nous n'avons pas eu besoin de modifier ou changer certains points dans la méthode présentée. L'étape d'évaluation des performances est présentée dans la suite, dans le chapitre 12.

PARTIE III

Outils et étude de cas

CHAPITRE 10
Outils

10.1 Introduction

Lors de l'étude des performances d'un système conçu dans le cadre d'un projet donné, il est souhaitable de pouvoir calculer effectivement les paramètres de performance à l'aide d'outils spécialisés. Ceci permet de vérifier si les objectifs ciblés par les utilisateurs finaux du système considéré vont être atteints et mettre en évidence les propriétés non soupçonnées générant des phénomènes imprévisibles.

Un outil d'analyse d'un modèle de système stochastique à événements discrets (SSED) doit assurer au minimum les fonctions de calcul des paramètres de performances à partir de la description du système dans le cadre de modélisation choisi. En ce qui concerne les modèles à base de réseaux de Petri stochastiques, l'outil doit pouvoir générer la chaîne de Markov sous-jacente et résoudre le système linéaire $\pi.Q = 0$. Comme nous utilisons dans notre étude le formalisme des SWNs et comme nous travaillons au niveau symbolique, le seul outil qui supporte ce modèle, à notre connaissance, est l'outil GreatSPN [173; 50] de l'Université de Turin (Italie).

Par ailleurs, les travaux de Moreaux & al. [97; 98; 158], qui ont abouti à la définition des compositions synchrone et asynchrone, ont été implantées par Delamare [65; 66; 67] sous-forme d'un outil, *TenSWN*. Cet outil se base sur le code de GreatSPN pour la génération du SRG d'un modèle SWN. Il a été amélioré par un lancement en parallèle du calcul des SRGs des réseaux étendus. Ceci a donné lieu à une nouvelle version *compSWN*.

Dans notre travail, nous avons utilisé les deux outils *compSWN* et GreatSPN, l'un pour le calcul compositionnel, l'autre pour le calcul sur le réseau global en vue de la comparaison des résultats de temps de calcul et taille de l'espace d'états d'un modèle avec les résultats obtenus par *compSWN*.

Pour calculer de manière *exacte* les performances de nos modèles qui sont de haut niveau, les outils doivent pouvoir les fournir au niveau symbolique. Toutefois, ni GreatSPN ni *compSWN* ne peuvent calculer ces indices au niveau symbolique. De plus, la gestion des résultats calculés (graphe symbolique, probabilités à l'équilibre, etc.) et la construction de graphiques, d'expressions complexes, après calcul des indices, ne sont pas intégrées également dans ces outils. Ceci est dû, pour ce qui est de GreatSPN, au fait que le souci des chercheurs de Turin était concentré, à juste titre, sur les fonctionnalités les plus importantes qui demandent le plus d'expertise scientifique. Quant à l'objectif de *TenSWN* (et plus tard de *compSWN*), il consistait à permettre un calcul compositionnel d'une composition de SWNs.

Les outils présentés jusqu'à présent concernent le calcul de l'espace d'états d'un modèle et de ses probabilités à l'équilibre. Qu'en est-il du calcul des indices de performance ?

Dans le domaine du calcul scientifique, plusieurs outils informatiques sont disponibles, étendant le paradigme de la calculatrice au calcul matriciel numérique (MatLab, Scilab), voire symbolique (Mathematica, Maple, Mupad). Le mode interactif, la large gamme de fonctions et les capacités de programmation (scripts interprétés) de ces outils sont très appréciés.

Afin de profiter des fonctionnalités de ces outils pour la dérivation d'indices de performance de modèles SWNs traités par GreatSPN, Sene et al. [99] ont développé une solution sous la forme d'un

ensemble d'outils appelé *PERFSWN*, dont l'objectif était de coupler l'outil de gestion des SWN avec l'outil interactif extensible Scilab. Ces outils logiciels ont pour fonction:

- d'assurer une exportation des résultats fournis par l'outil GreatSPN, et d'en calculer de nouveaux;
- de rendre ces nouveaux résultats disponibles pour leur manipulation au sein de l'environnement de calcul scientifique Scilab;
- de fournir un jeu de fonctions sous Scilab pour gérer les résultats importés et modifiés à partir de GreatSPN: calculs complémentaires, graphes, etc.

Les outils externes à Scilab sont écrits en langage Perl. Les outils sous Scilab sont regroupés dans des bibliothèques de fonctions Scilab.

Nous nous sommes également basés, pour le calcul d'indices de performance de nos modèles SWNs sur les outils *PERFSWN*.

Enfin, pour permettre d'automatiser toutes les étapes d'analyse d'un système basé composant, il est nécessaire de développer un ensemble d'outils, permettant la modélisation automatique des composants (si possible), la recherche d'une décomposition compatible à l'analyse structurée, l'extension des réseaux SWNs choisis issus de la décomposition et l'exécution de *compSWN* puis des outils *PERFSWN*. Ceux-ci doivent être réécrits pour l'exploitation des résultats issus du calcul compositionnel sous l'outil Scilab.

Dans la suite, nous présentons l'ensemble de ces outils.

10.2 GreatSPN

GreatSPN [173; 50] est un puissant outil graphique, dédié aux GSPNs et aux SWNs, initialement développé pour la spécification et l'évaluation des performances des architectures informatiques. Son développement a commencé en 1984 au sein du groupe de recherche en évaluation de performance de l'université de Turin (Italie). GreatSPN possède de nombreuses fonctionnalités, à savoir :

- une interface graphique permettant à l'utilisateur de définir son réseau et tous ses paramètres;
- un ensemble de fonctions d'analyse qualitative:
  - calcul des T-invariants, P-invariants;
  - détection d'interblocage, marquages « morts »), situation de conflits;
  - vérification de la bornitude du réseau;
  - calcul du graphe d'accessibilité.
- des outils d'analyse quantitative, notamment les étapes d'une analyse de performance :
  - définition des indices de performance;
  - calcul du vecteur des probabilités à l'état stationnaire;
  - calcul d'indices de performance définis par l'utilisateur;
  - calcul de la distribution des jetons dans les places et des débits moyens des transitions dans les modèles *GSPN*.
- un simulateur stochastique pour les modèles de grande taille.

La figure 10.1 montre une session sur GreatSPN. Ce logiciel est diffusé par son équipe de développement et fonctionne sur les systèmes Solaris et Linux (de type Unix) en environnement X11-Motif pour la partie graphique.

Une caractéristique importante de GreatSPN est la possibilité de sauvegarder le SRG sous forme de fichier texte (codé en ASCII, en lignes). Ce fichier contient tous les marquages symboliques (tangibles et évanescents), leurs représentations canoniques avec leurs descriptions et l'ensemble des franchissements symboliques. Notons cependant que la taille de ce fichier peut être considérable (de 1 à plusieurs Go) lorsque la taille du modèle analysé est importante.

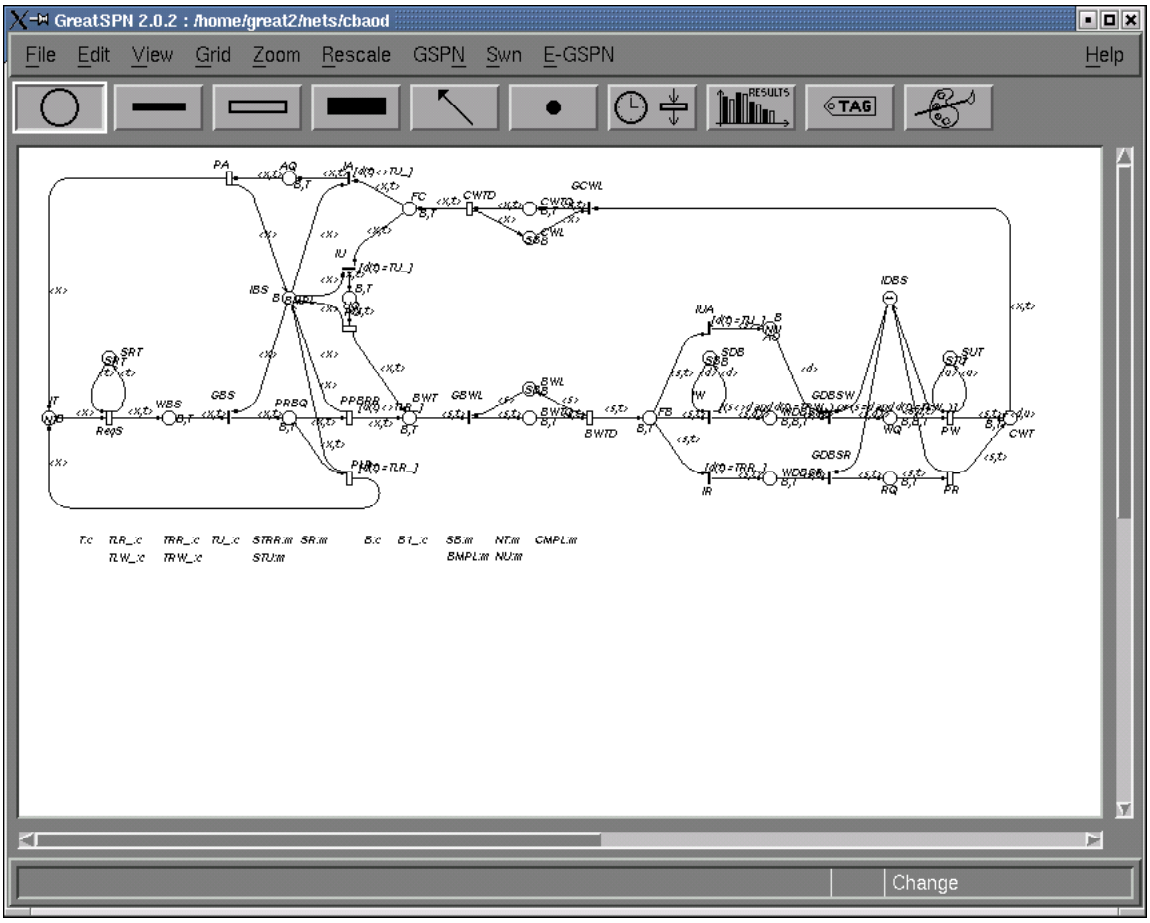


Figure 10.1: Session sous GreatSPN

En ce qui concerne les SWNs, l'utilisateur peut :

- définir tous les éléments du modèle SWN à partir de l'environnement graphique,
- calculer le SRG (marquages ordinaires et symboliques), et
- calculer les probabilités à l'état stationnaire de la chaîne de Markov agrégée.

Le simulateur WNSIM prend en compte les indices de performances symboliques des SWN. En raison de la complexité de ce modèle, le développement de la partie de l'outil relative aux SWNs est une tâche difficile et longue. De ce fait, tous les résultats théoriques concernant le modèle SWN ne sont pas encore intégrés dans l'outil. Par exemple, le calcul d'invariants symboliques pour certaines sous-classes statiques de SWN n'est pas implanté; les indices de performance au niveau symbolique (par exemple débit d'une transition suivant un tuple de sous-classes statiques, distribution d'un tuple de sous-classes statiques dans une place) ne sont pas encore disponibles dans le cadre de la résolution exacte (mais ils sont disponibles avec le simulateur WNSIM).

Afin de pallier ces insuffisances, un ensemble d'outils, *PERFSWN*, ont été développés. Ces outils se basent sur l'environnement scientifique Scilab. Nous présentons ces deux outils dans ce qui suit.

10.3 Scilab

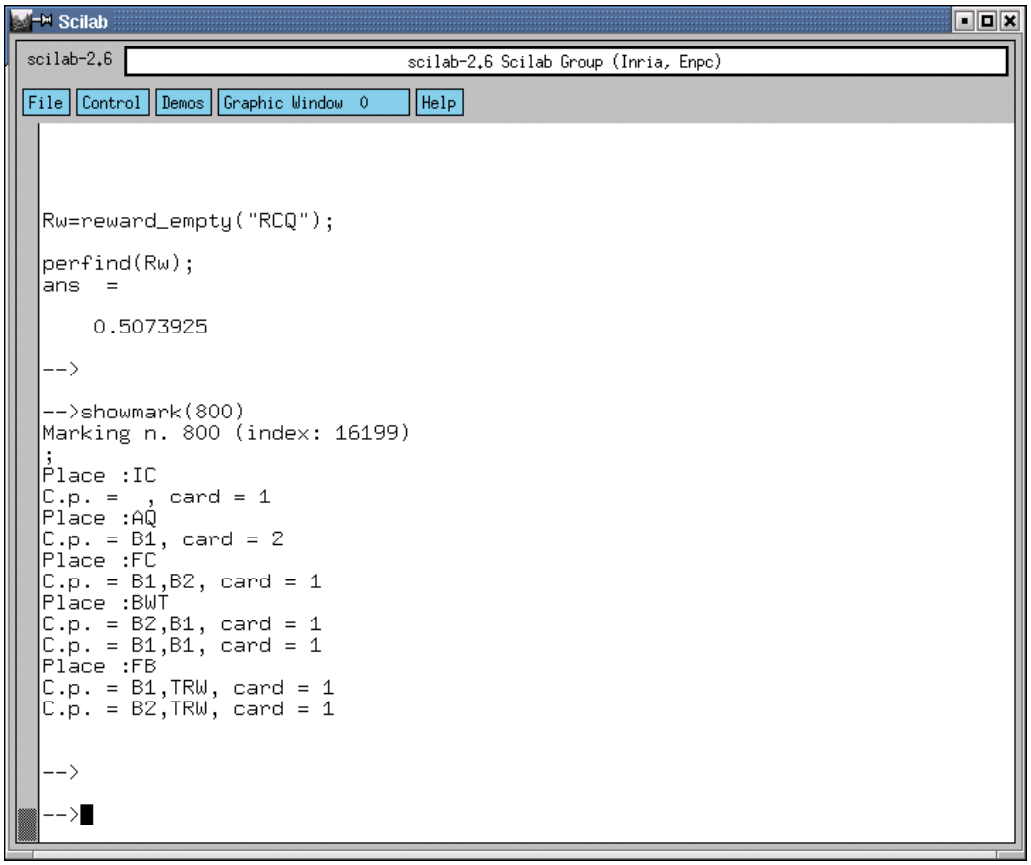


Figure 10.2: Session sous Scilab

Scilab (disponible à partir de http://www-rocq.inria.fr/scilab) est un environnement de calcul numérique interactif développé par l'Institut National de Recherche en Informatique et Automatique (INRIA) Elle fonctionne sur la plupart des systèmes d'exploitation actuels. La figure 10.2 montre une session Scilab dans laquelle nous pouvons calculer des indices de performance à partir des bibliothèques de *PERFSWN*.

Comme le logiciel commercial Matlab, Scilab utilise un modèle de calcul orienté matriciel. Ceci s'adapte parfaitement à nos types de problèmes puisque nous utilisons fréquemment des tableaux. Par exemple, la récompense moyenne $\overline{r}$ associée à une fonction de récompense r définie sur les marquages symboliques, se calcule comme le produit scalaire $r.\pi$ du vecteur r sur l'ensemble des marquages, par le vecteur π des probabilités stationnaires.

La plupart des méthodes et algorithmes de base en calcul numérique sont disponibles sous Scilab, notamment (liste non exhaustive):

- les outils de l'algèbre linéaire numérique (inversion de matrice, calcul de vecteurs et valeurs propres, résolution de systèmes linéaires, etc.),
- les outils de l'analyse numérique (transformation de Fourier rapide, solveurs d'équations différentielles, etc.), et

- les outils de gestion graphique (tracés de fonctions, figures élémentaires, etc.).

Scilab permet également de développer une interface utilisateur moderne, en mode graphique, en proposant un jeu de fonctions issues de l'interface graphique Tk (disponible aussi sur toutes les plate-formes). Un point intéressant est la capacité de sauvegarder une session en cours sur disque : on peut ainsi interrompre un travail et le reprendre plus tard.

Au delà du mode interactif, Scilab offre un langage de programmation procédural classique. Les programmes (appelés scripts) sont de simples fichiers textes interprétés et exécutés par Scilab à leur chargement. Par ailleurs, l'environnement de base peut être étendu avec de nouvelles fonctions, chargées en cours de session et placées dans des fichiers textes ou binaires appelés bibliothèques. L'utilisateur peut ainsi aisément étendre le domaine d'application de l'outil. Les outils *PERFSWN* utilisent les deux approches.

Enfin, un autre mode d'utilisation est la connexion de Scilab avec les programmes écrits en langage C. Cette liaison peut se faire ´ dans les deux sens ˇ. On peut appeler le moteur de calcul de Scilab à partir d'un programme en C; on bénéficie ainsi de toute la panoplie d'algorithmes déjà codés (et bien codés!). On peut à l'inverse, appeler du code C à partir de Scilab, par exemple pour des questions d'efficacité.

10.4 *PERFSWN*

PERFSWN est un ensemble d'outils, offrant un environnement interactif pour définir, calculer et présenter à l'utilisateur des indices de performances de modèles SWN calculés à l'équilibre, liés uniquement à des sous-classes statiques du SWN analysé. Ces outils complètent l'outil GreatSPN 2.0.2 pour exploiter le SRG et le vecteur de probabilités stationnaires d'un SWN. En plus de GreatSPN2.0.2, *PERFSWN* s'appuie sur plusieurs scripts Perl développés par [99], ainsi que sur l'environnement interactif numérique Scilab.

Afin de calculer des indices de performances, l'utilisateur met en œuvre les deux programmes GreatSPN et Scilab et exécuter certains logiciels de *PERFSWN* sous shell.

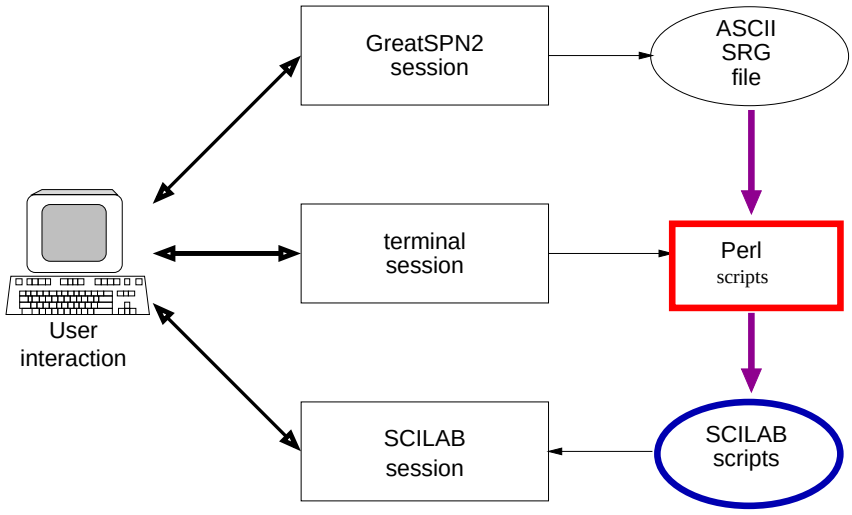


Figure 10.3: Relation entre les sessions

L'environnement utilisateur comporte trois sessions (Figure 10.3):

- Une session GreatSPN est dédiée à la définition du modèle SWN et au calcul des indices de performances pouvant être disponibles avec l'outil sous interface graphique;
- Une session Scilab permet de charger les résultats calculés par GreatSPN, de calculer différents indices de performance, de visualiser (sous forme graphique si nécessaire) ou de stocker les résultats;
- Une session terminal (shell) permettant à l'utilisateur de:
  - lancer les logiciels d'interface de *PERFSWN* (scripts Perl),
  - lancer en mode shell le calcul du SRG et de π à l'état stationnaire (logiciels de GreatSPN).

Concrètement, un utilisateur commence par créer son SWN sous GreatSPN. Puis, sous shell, il lance les calculs du SRG et de π. Les données nécessaires sont ensuite préparées, élaborées à partir des résultats fournis par GreatSPN (SRG et probabilités à l'équilibre). Ces données consistent en la définition des classes de base, sous-classes statiques, places du SWN, ... Cette préparation est suivie par l'exécution de scripts Perl traitant le fichier du SRG pour le calcul des scripts Scilab. Enfin, sous Scilab, l'utilisateur charge les informations calculées auparavant, puis lance des commandes Scilab pour le calcul des indices de performance souhaités.

10.5 *TenSWN* et sa version améliorée *compSWN*

Les travaux de décomposition structurée synchrone et asynchrone de SWNs [97; 98; 158] ont été implantés sous forme d'une première version d'outil, *TenSWN* [66; 67]. Cet outil permet de calculer les probabilités à l'équilibre de marquages symboliques et ordinaires d'un modèle SWN (soit N), pouvant être décomposé en plusieurs sous-réseaux SWN (soit N_1, N_2, N_m), liés d'une manière soit asynchrone ou synchrone. Il utilise le format GreatSPN pour les réseaux en entrée, et le noyau de GreatSPN pour le calcul de données intermédiaires, notamment les SRGs des sous-réseaux étendus.

L'analyse d'un tel SWN est réalisée en plusieurs étapes:

1. Partant du SWN global N, les sous-réseaux $N_1, \ldots N_m$ sont définis de telle manière à satisfaire les conditions syntaxiques données dans les définitions 4.21 et 4.26. Cette définition est sous la responsabilité du modélisateur (manuelle). Une autre approche possible est de combiner ou composer plusieurs sous-réseaux précédemment définis. Dans ce cas aussi, l'utilisateur doit vérifier les conditions syntaxiques énoncées sur le réseau SWN global, composition des sous-réseaux.
2. Les réseaux étendus sont ensuite définis de la manière expliquée dans la définition 6.2. Ces réseaux étendus sont également définis par l'utilisateur.
3. Les réseaux étendus doivent être stockés au format `.net` et `.def` de GreatSPN. Ceci est fait à l'aide de l'interface graphique de GreatSPN.
4. Un fichier de *synchronisation* est après défini : c'est un fichier texte contenant toutes les informations nécessaires à *TenSWN* concernant les compositions synchrones ou asynchrones du SWN N. Il s'agit des noms des réseaux étendus, le type des compositions (synchrone ou asynchrone), les transitions de synchronisations, les vues abstraites des sous-réseaux, la liaison entre les sous-réseaux (réseau client, serveur, etc), ...
5. Enfin, l'outil est exécuté procédant comme suit (tel qu'expliqué en chapitre 7) :
   - Génération des $\overline{SRG_k}$ des SWN étendus $\overline{N_k}$, en utilisant la fonction de calcul du SRG d'un SWN de l'outil GreatSPN. Pour chaque réseau étendu $\overline{N_K}$, un ensemble d'accessibilité est obtenu, complété par une collection de matrices contenant les franchissements des transitions. Cet ensemble de matrices est composé d'une matrice Q' regroupant les transitions dites locales de

chaque réseau (ne mettant en jeu que le réseau $\overline{N_k}$ lui-même) et une série de matrices contenant les franchissements des transitions globales (mettant en jeu plusieurs réseaux étendus).
- Calcul et stockage de l'ensemble effectif d'accessibilité du SWN global dans une structure de MDD (*Multi Decision Diagram*), en suivant une technique d'encodage proposée par Ciardo et Miner [154].
- Calcul de la forme tensorielle du générateur global qui est une sur-matrice de celui obtenu par la méthode classique sur le modèle global. Pour cela, *TenSWN* se base sur l'utilisation d'une structure de MxD (*Matrix decision Diagram*) [53], qui permet une représentation optimale d'un produit tensoriel de n matrices, restreint à un ensemble de tuples d'états représenté sous forme de MDD.
- Résolution à l'équilibre du système linéaire $\pi.Q = 0$. Cette résolution se base sur l'expression tensorielle du générateur sous forme de MxD et d'une méthode itérative de type Gauss-Seidel. On note que la représentation en MxD s'adapte facilement aux méthodes Gauss-Seidel.

La technique d'encodage de l'ensemble d'accessibilité du SWN N de Ciardo et Miner consiste à le considérer comme un ensemble de K-tuples d'états et le stocker dans un MDD sous-forme d'un arbre à K+1 niveaux, en tenant compte du fait que les états sont générés par la survenue d'événements locaux ou globaux. La construction de ces ensembles d'événements se fait sur les $\overline{SRG_k}$. A partir de l'ensemble des événements, la méthode de *saturation* [139] est utilisée pour générer de façon optimale le MDD représentant l'ensemble des K-tuples effectivement accessibles. L'espace d'états du SWN est ainsi efficacement stocké.

La forme tensorielle du générateur global de N est définie sur l'ensemble potentiel, produit cartésien des sous-ensembles d'accessibilité. Les travaux de Ciardo et Miner avaient pour objectif de restreindre l'expression tensorielle aux tuples d'états effectivement accessibles en se servant du MDD préalablement calculé. Ils ont défini une structure de données, appelée Matrix decision Diagram (MxD) [53], permettant une représentation optimale d'un produit tensoriel de n matrices, restreint à un ensemble de tuples d'états représenté sous forme de MDD. Dans cette représentation, la forme tensorielle est limitée aux tuples d'états effectivement accessibles codés par le MDD, ce qui fait son intérêt. Le principe est de calculer récursivement la restriction d'une matrice d'un noeud aux états présents dans le noeud adjacent du MDD. L'algorithme de calcul du MxD est basé sur l'ensemble des événements et évidemment sur le MDD associé calculé précédemment.

Comme pour le calcul du MDD, les algorithmes implantés sont identiques à ceux définis par Ciardo, mais sans optimisation de cache et tampons.

En résumé, l'algorithme global implanté par *TenSWN* pour l'analyse d'un SWN consiste dans les étapes suivantes :

1. Calcul des SRG_k en isolation;
2. Construction de l'ensemble des événements locaux et synchronisés;
3. Calcul du MDD de l'ensemble effectif d'accessibilité;
4. Calcul du MxD de l'expression tensorielle du générateur;
5. Résolution à l'équilibre de la CTMC agrégée avec la méthode itérative de Gauss-Seidel.

Plus tard, l'outil *TenSWN* a été amélioré en autorisant une exécution en parallèle du calcul des SRG_k des réseaux étendus. Ceci a donné lieu à une nouvelle version appelée *compSWN*.

10.6 Outils nécessaires pour l'analyse de modèles CBS

Un ensemble d'outils est nécessaire à développer pour pouvoir automatiser l'analyse des performances d'un système basé composant. Ces outils doivent permettre :

1. La modélisation la plus automatique possible du CBS, en générant systématiquement le G-SWN associé.
2. La recherche d'une décomposition du G-SWN compatible à l'analyse structurée.
3. L'extension des réseaux SWNs issus de la décomposition.
4. L'exécution de *compSWN* puis utilisation des outils *PERFSWN* pour le calcul d'indices de performances.

10.6.1 Automatisation de la construction du G-SWN d'un CBS

Cette étape requiert une étape préalable de modélisation du noyau des composants primitifs de l'application. Cette étape peut être réalisée pour chaque domaine d'ingénierie : si une bibliothèque donnée de composants a été développée, le modélisateur prépare également une bibliothèque des modèles SWNs de ces composants, construits auparavant en suivant les règles de traduction proposées dans ce document pour la modélisation des interfaces.

Un outil devra alors récupérer les modèles SWNs des composants primitifs, faire les renommages nécessaires pour éviter les conflits de noms entre les SWNs, puis construire le G-SWN associé suivant l'algorithme donné en chapitre 5, en se basant sur la description d'architecture ADL du CBS.

Actuellement, cette étape n'a pas été automatisée mais se fait par le modélisateur.

10.6.2 Recherche d'une décomposition compatible du G-SWN

Cette étape est implantée par l'algorithme 6.4.2, donné en chapitre 6. Elle n'a pas non plus fait l'objet d'une implantation pour l'instant.

10.6.3 Extension des réseaux SWNs de la décomposition

Un premier travail d'implémentation a été fait dans ce sens. Ce travail consiste à :

1. récupérer des informations sur un ensemble de SWNs à composer d'une manière mixte. Ces informations, contenues dans des fichiers XML, consistent en les noms des réseaux SWNs à composer, les types des compositions (synchrone, asynchrone), les transitions de synchronisation dans les compositions synchrones, les transitions de sortie dans les compositions asynchrones, etc. Notons que les SWNs sont définis au format GreatSPN. Une difficulté se pose toutefois pour la construction des vues abstraites des SWNs : les semi-flots d'abstraction et les fonctions d'arcs associées sont difficiles à calculer, étant donné que GreatSPN ne permet pas le calcul d'invariants pour le modèle SWN. Pour cela, nous nous basons sur une connaissance au préalable des semi-flots, donnés dans le fichier d'informations XML.

   Un exemple de fichiers XML est donné en annexe.
2. construire les réseaux étendus correspondant aux SWNs.
3. construire le fichier de synchronisation nécessaire au calcul de *compSWN*.
4. lancer l'outil *compSWN*.

10.6.4 Calcul des performances

L'outil *compSWN* calcule quelques indices de performance tels que les débits des transitions, en décoloré.

Pour calculer des indices spécifiques, nous utilisons les outils *PERFSWN*. Nous définissons les indices qu'on souhaite calculer, et écrivons les scripts Scilab nécessaires à cela, puis nous faisons les calculs et génération des graphiques associés, suivant *PERFSWN*.

Toutefois, ces outils ont été écrits pour l'analyse d'un SWN N plat sans aucune composition. Ils doivent donc être réécrits pour fournir des indices de performance concernant une composition de SWNs, et partant des résultats fournis par *compSWN* (SRG_k, probabilités à l'équilibre du réseau global N, ...). Ces scripts sont en cours de réécriture.

10.7 Conclusion

Ce chapitre a présenté l'ensemble des outils impliqués pour le calcul de performances d'un CBS. Plusieurs étapes sont en cours de réalisation.

Par ailleurs, il serait intéressant pour les modèles à composant que nous avons étudié (FRACTAL et CCM) de fournir une automatisation de la construction d'un modèle SWN d'un composant, à partir de l'analyse de son code.

CHAPITRE 11

Application Fractal Julia - Analyse

11.1 Introduction

Nous avons présenté dans le chapitre 8 l'application de la méthode d'analyse structurée sur des systèmes basés sur le modèle FRACTAL. Un exemple d'application a été exhibée. Nous reprenons cet exemple pour étudier les performances de l'application.

11.2 Analyse de l'application exemple décrite auparavant

À partir de la modélisation présentée dans le chapitre 8, nous avons obtenu un ensemble de CC-SWNs associés aux composants, ainsi que le G-SWN de l'application. En exploitant les réseaux obtenus, nous cherchons d'abord une décomposition de SWNs compatible aux conditions énoncées dans le théorème 6.4. On note que l'ensemble des (CC-SWN_k) obtenus satisfait nos conditions, d'où la meilleure décomposition compatible est celle de la configuration initiale des CC-SWNs. Nous étendons donc ces SWNs en réseaux étendus. Nous obtenons les réseaux étendus des figures 11.1, 11.2, 11.3 et 11.4.

Nous utilisons notre outil d'analyse *compSWN* sur cet ensemble de réseaux SWNs afin de calculer les SRGs des réseaux étendus et les probabilités à l'équilibre. Nous avons également utilisé l'outil GreatSPN pour comparer les résultats (temps de calcul et espace mémoire nécessaire) des deux méthodes d'analyse (analyse sur le modèle global et analyse compositionnelle). Les outils sont installés sur une station Suse linux 9.2 avec 512 MO.

11.2.1 Paramètres du modèle

Afin de montrer les gains de la méthode structurée, nous avons fait varier les cardinalités des classes de couleurs de base, puis étudié ces diverses configurations (données par la table 11.1) avec les deux solveurs. La notation |Couleur| désigne la cardinalité de la sous-classe statique de couleurs dénotée par *Couleur*.

Par ailleurs, nous avons calculé des indices de performance pour une configuration donnée (Cf4) du système. Nous nous sommes intéressés précisément à voir l'évolution de certains temps de réponse. Pour cela, nous avons fait varier certains taux de franchissement de transition, puis nous avons étudié les temps de réponse à partir des probabilités stationnaires obtenues. Les taux des transitions principales sont données dans la table 11.2. Les transitions n'apparaissant pas dans cette table ont un taux égal à 1 (i.e. plus rapide par rapport aux autres transitions qui ont même unité).

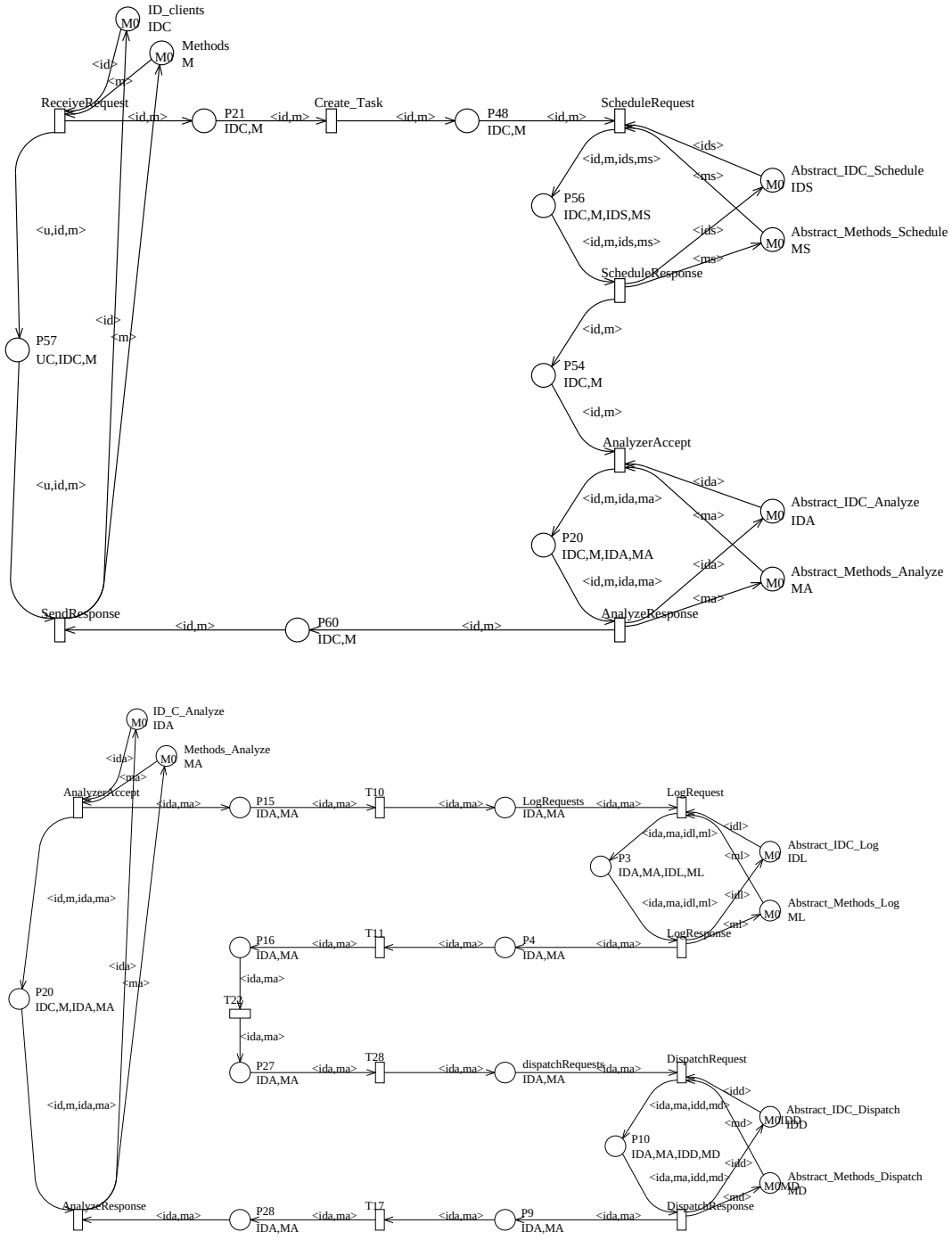


Figure 11.1: Réseaux étendus des composants récepteur (haut) et analyseur (bas)

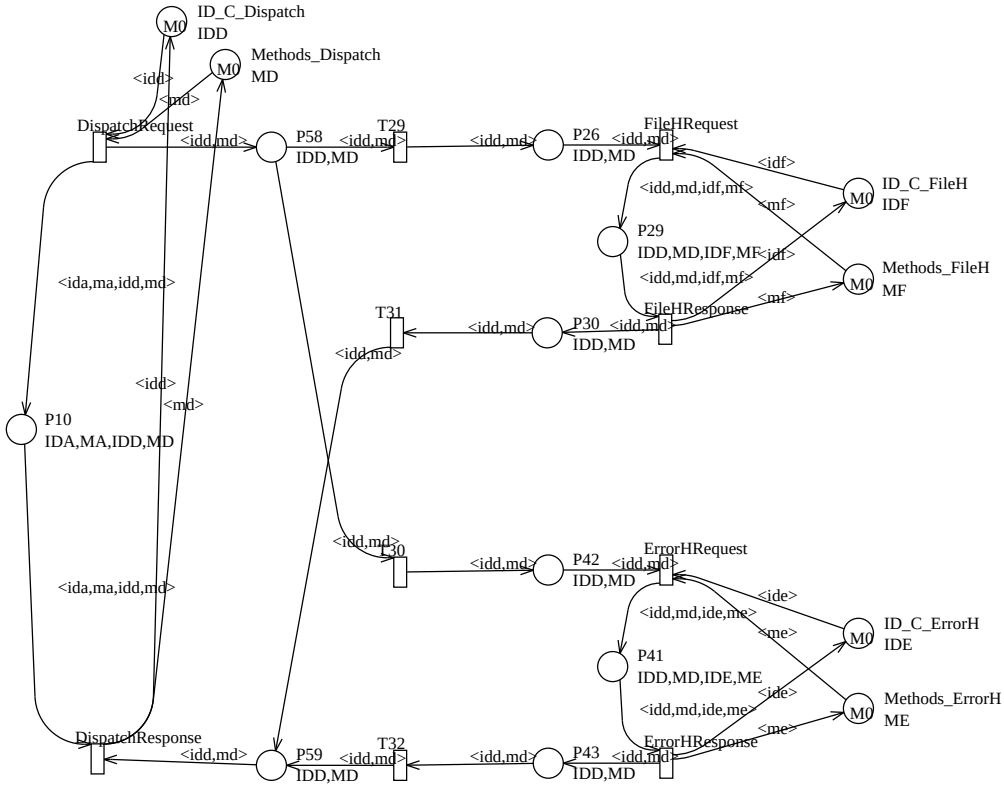


Figure 11.2: Réseau étendu du composant dispatcher

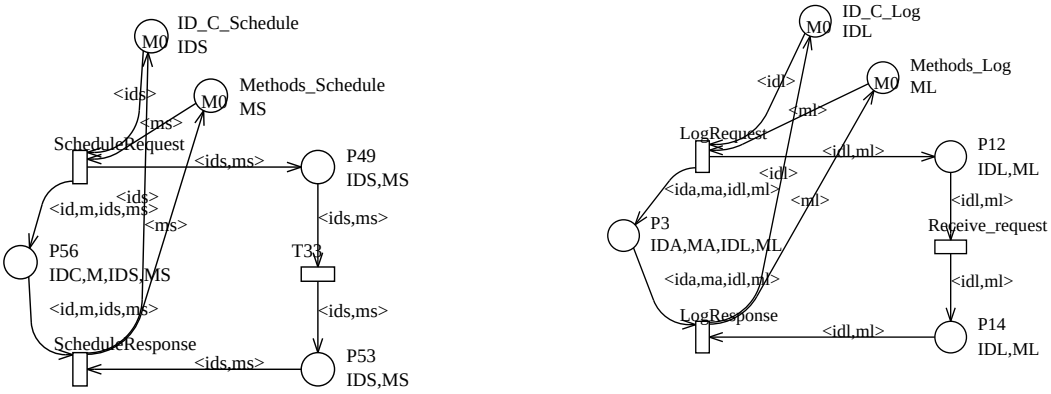


Figure 11.3: Réseaux étendus des composants ordonnanceur (haut) et du gestionnaire de journal (bas)

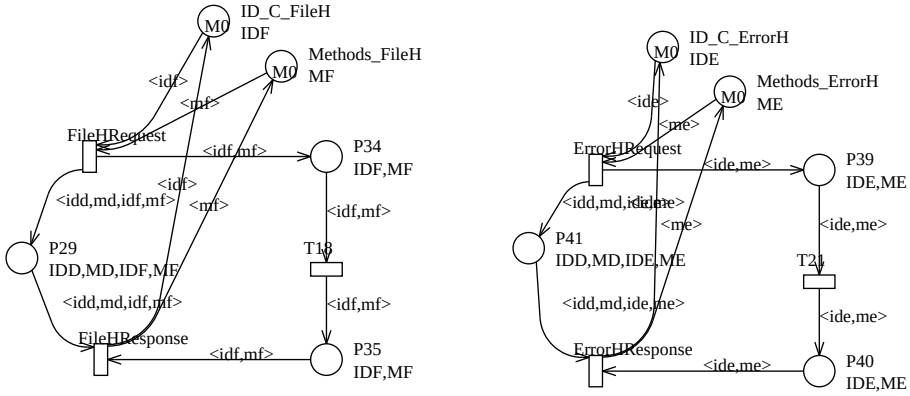


Figure 11.4: Réseaux étendus des composants gestionnaire de fichier (haut) et gestionnaire d'erreur (bas)

$\|Couleur\|$	Cf1	Cf2	Cf3	Cf4	Cf5	Cf6	Cf7	Cf8
$\|UC\|$	3	5	10	2	3	5	10	20
$\|IDC\|$	1	1	1	2	2	2	2	2
$\|M\|$	2	2	2	2	2	3	3	3
$\|IDS\|$	3	5	10	2	3	5	10	20
$\|MS\|$	2	2	2	2	2	2	2	2
$\|IDA\|$	1	1	1	2	2	3	3	3
$\|MA\|$	2	2	2	2	2	2	2	2
$\|IDL\|$	2	2	2	2	2	5	5	5
$\|ML\|$	2	2	2	2	2	3	3	3
$\|IDD\|$	1	1	1	2	2	3	3	3
$\|MD\|$	2	2	2	2	2	3	3	3
$\|IDF\|$	2	2	2	2	3	5	5	5
$\|MF\|$	2	2	2	2	2	5	5	5
$\|IDE\|$	2	2	2	2	2	5	5	5
$\|ME\|$	2	2	2	2	2	4	4	4

Table 11.1: Configurations diverses de l'exemple du serveur Comanche

Composant	Transition	Taux
Récepteur	ReceiveRequest	0.6
Ordonnanceur	ScheduleRequest	0.9
Analyseur	AnalysisRequest	0.6
Analyseur	T10	0.75
Logger	LogRequest	0.9
Dispatcher	DispatchRequest	0.9
Dispatcher	T29	0.9
Dispatcher	T30	0.1
Gestionnaire de fichiers	FileHRequest	0.9
Gestionnaire d'erreurs	ErrorHRequest	0.1

Table 11.2: Taux de franchissement des transitions de la configuration étudiée

Config	NbS	NbO	TGreat(s)	TComp(s)	MGreat (B)	MComp (B)
Cf1	82	35144	4	0	402	1484
Cf2	136	239392	5	0	510	1652
Cf3	271	16139264	12	0	780	2072
Cf4	406	2392068	485	1	6305	5336
Cf5	784	24279944	4919	1	7095	6008
Cf6	1540	3656635680	-	2	-	7368
Cf7	3430	3113239552	-	3	-	10728
Cf8	7210	999926785	-	22	-	17448

Table 11.3: Taille de l'espace d'états, temps de calcul et occupation mémoire pour diverses configurations de l'application Comanche

11.2.2 Taille et complexité du modèle obtenu - Gains de la méthode structurée

Avant de présenter quelques indices de performance, nous montrons ici les gains obtenus de la méthode structurée, en termes de temps de calcul et d'occupation mémoire. Le comportement des deux solveurs (GreatSPN and *compSWN*) pour le calcul nécessaire aux configurations (Cfi) résumées dans la table 11.1 est donné par la table 11.3. L'espace mémoire est donné en bytes, alors que les temps de calcul sont donnés en secondes. Les notations utilisées dans le table 11.3 sont comme suit :

- NbS est le nombre de marquages symboliques obtenus,
- NbO est le nombre de marquages ordinaires obtenus,
- TGreat est le temps de calcul global obtenu par GreatSPN,
- TComp est le temps de calcul global obtenu par *compSWN*,
- MGreat est la taille mémoire (en Megabytes) utilisée par GreatSPN, et
- MCcomp est la taille mémoire utilisée par *compSWN*.

À partir de cette table, nous remarquons que l'outil *compSWN* a été capable de calculer le SRG du G-SWN et ses probabilités à l'équilibre pour toutes les configurations données, alors que l'outil GreatSPN n'a pas pu calculer certaines configurations (en l'occurrence Cf6, Cf7, Cf8) à cause de l'im-

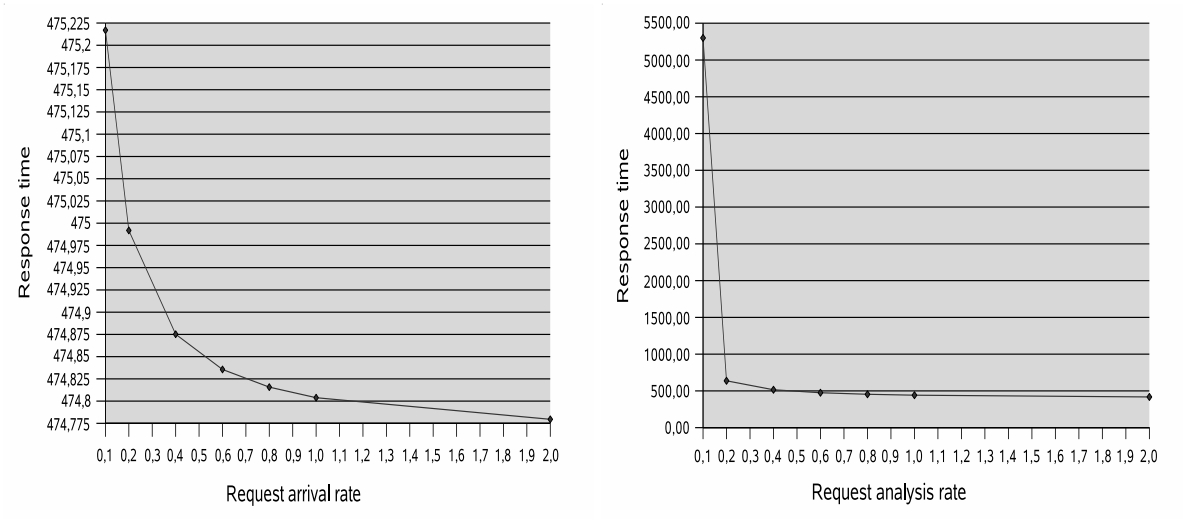


Figure 11.5: Temps de réponse versus taux d'arrivée des requêtes (gauche) et versus taux d'analyse des requête (droite)

portance de l'espace d'états du réseau global. Quant aux configurations que les deux solveurs ont pu étudié, les temps de calcul et occupation mémoire de *compSWN* sont bien meilleures par rapport à ceux de GreatSPN.

11.2.3 Indices de performances - Variations de temps de réponse

Nous nous sommes intéressés à étudier la variation du temps de réponse par rapport à plusieurs paramètres, à savoir :

- la charge induite par les requêtes des clients;
- le taux d'analyse (traitement) d'une requête;
- le taux de récupération des données d'une requête; et
- le taux d'erreurs.

La figure 11.5 montre la variation du temps de réponse par rapport aux deux premiers paramètres : la charge du système et le taux d'analyse d'une requête.

À partir du premier diagramme (situé à gauche), nous notons que le CBS présente un temps de réponse légèrement meilleur au fur et à mesure que le taux d'arrivée des requêtes augmente. À priori, c'est un comportement quelque peu contradictoire, indiquant que le système n'est pas saturé jusqu'à atteindre un taux d'arrivée égal à 2. En fait, la courbe devient plus plate lorsque le taux d'arrivée augmente.

Le second diagramme montre un temps de réponse réduit avec l'augmentation du taux d'analyse des requêtes. Ceci est attendu puisque le système devient plus puissant avec une vitesse élevée de traitement. Toutefois, nous observons que le temps de réponse chute au départ d'une manière significative, puis devient globalement plus stable (à 0.4 et plus). Ce phénomène est la preuve que l'analyse ne constitue pas un goulet d'étranglement pour le système à partir de taux supérieurs à 0.4.

Par ailleurs, la figure 11.6 montre la variation du temps de réponse par rapport au taux de récupération des données (fichiers) et par rapport au taux d'erreurs. Il semble, à partir de ces deux diagrammes que le temps de réponse sont très instables, montrant quelquefois la rapidité du CBS à répondre à une requête, et d'autres fois sa lenteur pour le traitement d'une requête, probablement due à la survenue d'erreurs.

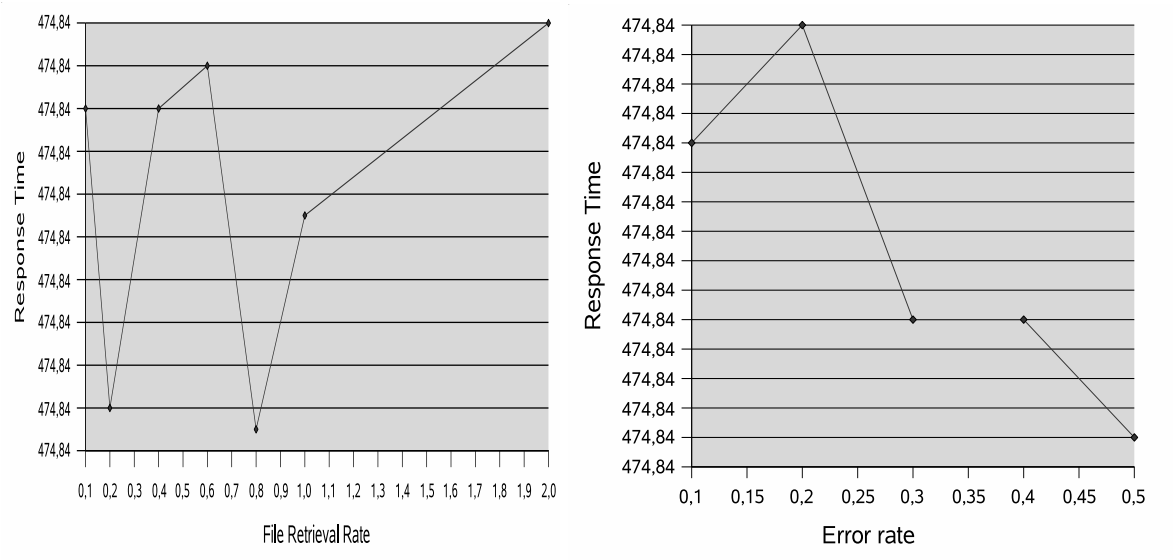


Figure 11.6: Temps de réponse versus taux de récupération des données (gauche) et taux d'erreurs (droite)

11.3 Conclusion

À la vue des résultats obtenus par l'application de notre méthode d'analyse sur les systèmes basés FRACTAL, nous pouvons dire que l'instanciation de la méthode structurée peut donner de bons résultats effectifs d'analyse, en autorisant l'analyse de systèmes larges en taille d'espace d'états.

Il reste quand même à étudier l'analyse des reconfigurations possibles, étant donné que les interfaces de contrôle joue un rôle important pour le modèle FRACTAL mais également pour les applications actuelles.

CHAPITRE 12
Application CCM - Analyse

12.1 Introduction

Nous avons présenté dans le chapitre 9 l'instanciation de la méthode d'analyse structurée sur des systèmes basés sur le modèle CCM. Un exemple d'application a été présenté. Nous reprenons cet exemple pour étudier les performances de l'application.

12.2 Analyse de l'application du système de contrôle avionique

Durant la phase de modélisation de notre exemple du système de contrôle avionique, présenté dans le chapitre 9, nous avons obtenu un ensemble de CC-SWNs associés aux composants, ainsi que le G-SWN de l'application. En exploitant les réseaux obtenus, nous cherchons d'abord une décomposition de SWNs compatible aux conditions énoncées dans le théorème 6.4. L'ensemble des (CC-SWN_k) obtenus satisfait nos conditions, d'où la meilleure décomposition compatible est celle de la configuration initiale des CC-SWNs. Notons toutefois que les modèles CC-SWNs des composants sont interconnectés à travers les modèles SWNs des canaux d'événement. En conséquence, pour éviter à faire les calculs pour des SWNs de taille très petite, nous avons choisi de grouper le composant *RateGen* avec le réseau fermant de l'application d'une part, et d'autre part avec le canal d'événement qui le relie au composant *GPS*. Le composant *GPS* a été également groupé avec le canal d'événement qui le relie au composant *NavDisplay*. Nous étendons donc ces SWNs en réseaux étendus. Nous obtenons les réseaux étendus des figures 12.1 et 12.2 :

- Le SWN groupant le composant *RateGen*, le réseau fermant et le premier canal d'événement a été augmenté de la vue abstraite du composant *GPS* (place *PAbstract_GPS* et transitions *push_tick* et *SendSub2*) et de la vue abstraite du composant *NavDisplay* (place *PAbstract_Nav* et transition *push_refresh*).
- Le SWN groupant le composant *GPS* et le second canal d'événement a été augmenté de la vue abstraite du composant *RateGen* (place *PAbstract_Rate* et transition *SendSub1*) et de la vue abstraite du composant *NavDisplay* (place *PAbstract_Nav* et transition *push_refresh*).
- Le SWN du composant *NavDisplay* a été augmenté de la vue abstraite du composant *GPS* (place *PAbstract_GPS* et transitions *SendSub2* et *push_tick*) et de la vue abstraite du composant *RateGen* (place *PAbstract_Rate* et transition *SendSub1*).

Nous utilisons notre outil d'analyse *compSWN* sur cet ensemble de réseaux SWNs afin de calculer les SRGs des réseaux étendus et les probabilités à l'équilibre. Nous utilisons également l'outil Scilab 3.0

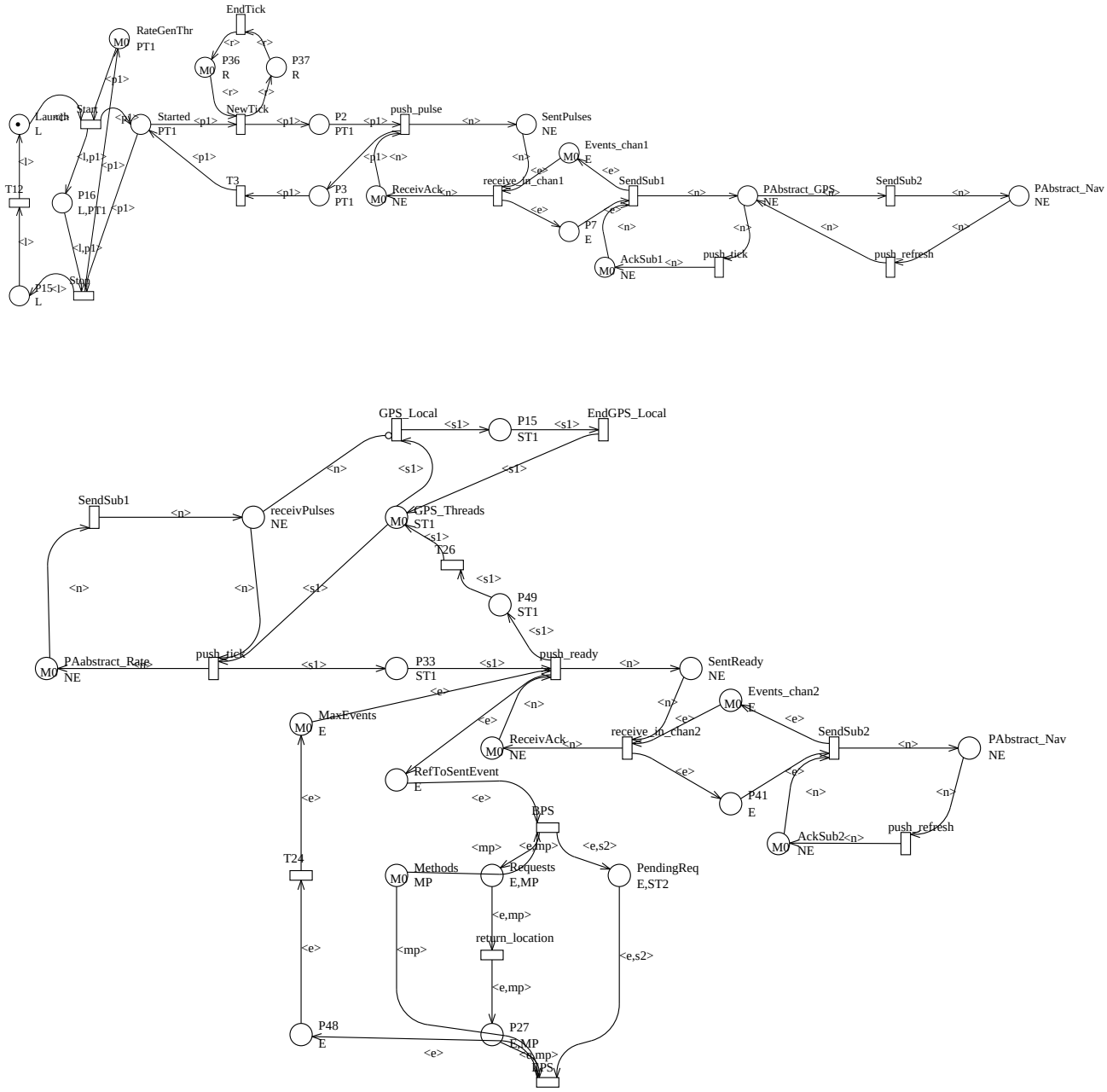


Figure 12.1: Réseaux étendus des composants RateGen (haut) et GPS (bas)

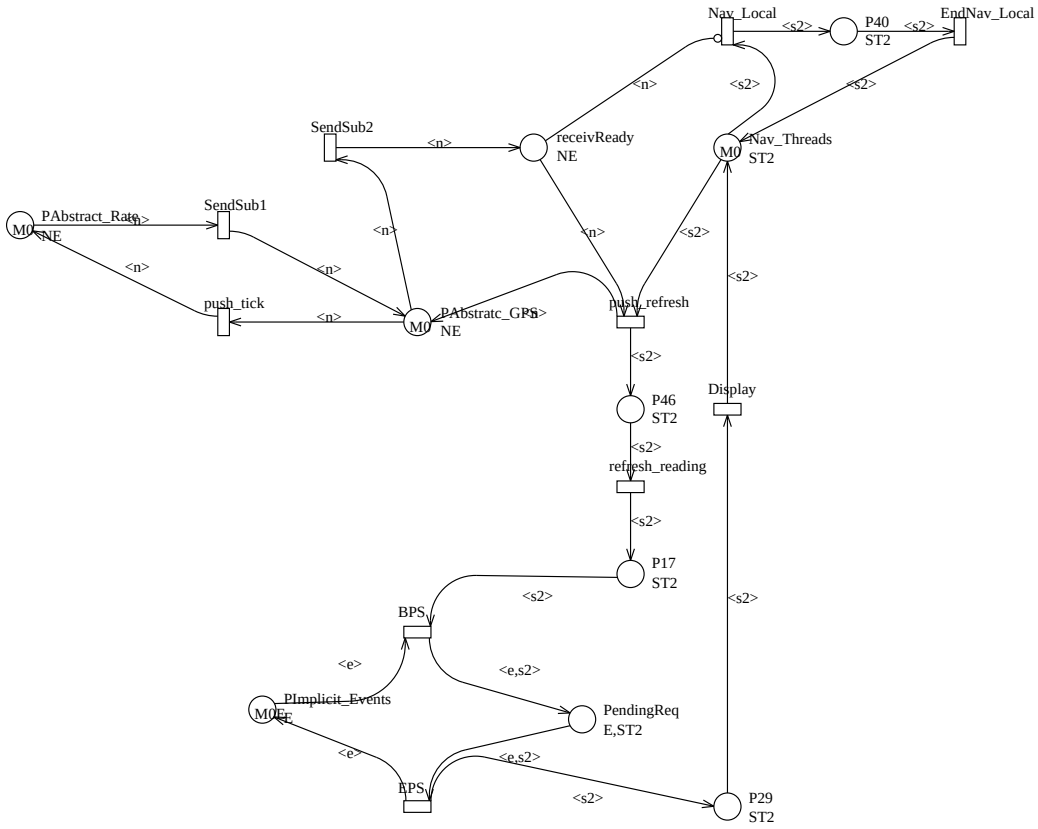


Figure 12.2: Réseau étendu du composant NavDisplay

Transition	Taux	Transition	Taux	Transition	Taux
NewTick	0.5	push_ready	0.75	return_location	0.9
push_pulse	0.8	push_refresh	0.8	BPS	8
push_tick	0.8	GPS_Local	0.4	Nav_Local	0.4

Table 12.1: Taux de franchissement des transitions de la configuration étudiée

ainsi qu'un ensemble de scripts Perl *PerfSWN* pour le calcul d'indices de performances. Les outils sont installés sur une station Suse linux 9.2 avec 512 MO.

Notre intérêt est porté sur le calcul de certains indices de performance, à savoir des temps de réponse pour une configuration donnée du système. Nous avons choisi une configuration qui génère un espace d'états symbolique (SRG) de taille égale à 281760 marquages symboliques (correspondant à 3937280 marquages ordinaires). Nous appliquons la méthode d'analyse structurée.

12.2.1 Paramètres du modèle

Pour calculer la variation des temps de réponse qui nous intéressent, nous considérons des valeurs fixes des taux de franchissement d'un ensemble critique de transitions (voir le tableau 12.1). Ensuite, nous faisons varier les taux correspondant aux transitions en relation avec les temps de réponse cibles. Ceux-ci sont étudiés partir des probabilités stationnaires obtenues. Les transitions n'apparaissant pas dans le tableau ont un taux égal à 1, i.e. plus rapide que toutes les autres transitions, les taux étant donnés avec la même unité).

12.2.2 Indices de performances - Évolution des temps de réponse

Nous nous intéressons à l'étude de la variation de deux indices de performance :

- Le temps de réponse induit pour le traitement d'une requête dans le composant *GPS* par rapport au taux de réception des notifications (taux de la transition *push_tick*), et
- Le temps de réponse du gestionnaire d'événements du composant *NavDisplay* par rapport au taux de réception des notifications (taux de la transition*push_refresh*).

Les diagrammes obtenus sont donnés par la figure 12.3.

Le diagramme de gauche montre une amélioration du temps de réponse à mesure que le taux de réception de la transition *push_tick* augmente. Ceci peut paraître quelque peu surprenant à première vue, car le temps de réponse diminue lorsque le nombre d'événements reçus par le composant GPS augmente. Ce comportement à priori contradictoire indique que le système n'est pas stable lors de la phase d'initialisation, mais adapte son activité aux événements arrivés et améliore son temps de réponse correspondant. Il n'atteint pas un état surchargé.

Le diagramme de droite montre la variation du temps de réponse pour deux taux différents de traitement d'une requête au sein du composant *GPS* (taux de franchissement de la transition *return_location*). Le diagramme présente une très légère augmentation du temps de réponse lorsque le taux de réception des événements augmente dans le composant *NavDisplay*. L'augmentation du temps de réponse est prévisible puisque la charge des deux composants (*NavDisplay* et *GPS*) devient plus importante. Toutefois, nous notons la très lente augmentation du temps de réponse, ce qui montre une bonne configuration qui reste non saturée pour une longue période de temps.

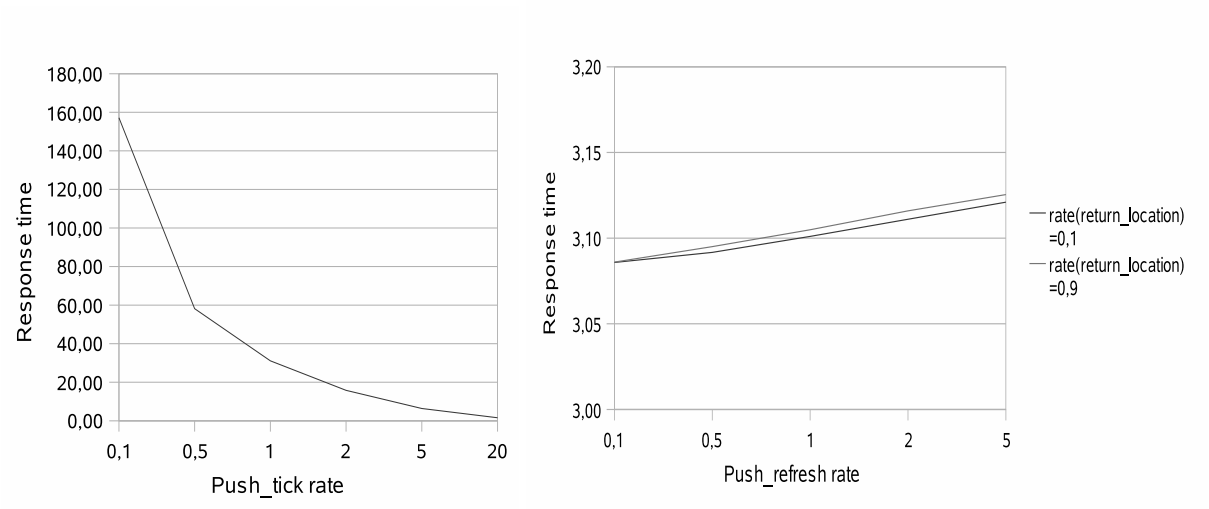


Figure 12.3: Temps de réponse d'une requête au GPS (gauche) et temps de réponse du gestionnaire d'événements du NavDisplay (droite) versus taux de réception des événements

L'analyse des performances peut être complétée par la recherche d'autres indices et le calcul de probabilités de certains marquages.

12.3 Conclusion

Dans ce chapitre, nous avons présenté un exemple d'analyse des performances d'un système basé composants CCM.

Cet exemple diffère de l'exemple FRACTAL du serveur Comanche dans le fait que le G-SWN obtenu est une composition mixte synchrone et asynchrone de SWNs. L'analyse compositionnelle de cet exemple a été possible, même en présence de compositions mixtes. En fait, ceci dépend du degré de dépendance entre les composants munis de facettes/réceptacles et d'interfaces basées événements.

Il serait également intéressant d'étudier l'analyse des configurations faisant intervenir des invocations de services non fonctionnels des conteneurs.

Conclusion générale

La conception et développement d'un système basé composant doit être accompagné d'une étape de vérification comportementale et analyse des performances du système résultant, afin de s'assurer de sa correction et qu'il atteint les objectifs pour lesquels il est conçu.

Dans cette optique, cette thèse a focalisé sur le développement d'une méthode générique pour l'évaluation des performances des systèmes basés composant, qui serait applicable pour des systèmes importants en taille.

L'idée fondamentale de notre approche comporte deux étapes essentielles : la première revient à générer un modèle SWN global (le G-SWN) modélisant le CBS à partir de sa description d'architecture et des comportements de ses composants. Ce modèle est vu comme une composition des modèles SWNs des composants et des interactions. La seconde étape consiste en l'application d'une méthode d'analyse structurée sur la composition obtenue (le G-SWN) pour le calcul d'indices de performances. Cette étape exploite la compositionnalité du modèle afin de réduire la complexité de l'analyse en termes de temps de calcul et d'occupation mémoire.

Pour cela, nous avons étudié, dans un premier temps, les caractéristiques communes aux différents systèmes à composants conçus suivant divers modèles de composants. À partir de là, nous avons dégagé les types d'interactions fréquents qui peuvent exister entre les composants, ainsi que les interfaces associées. Nous avons également discuté les raisons pour lesquelles le formalisme réseau de Petri bien formé (SWN) a été adopté, en passant en revue les modèles de performance existants, ainsi que les travaux proposés pour l'analyse des CBS.

Ensuite, nous avons exploité les caractéristiques dégagées des CBS, afin de définir une modélisation appropriée à une analyse structurée de performance. L'objectif tracé de cette étape de modélisation est d'obtenir un modèle global de CBS analysable d'une manière compositionnelle structurée, ce qui permettra de réduire la complexité d'analyse. À cet effet, nous modélisons les composants d'une application, en prenant en considération deux types d'interaction entre composants : l'interaction synchrone par invocation de service (méthode) et la communication asynchrone basée événement. Pour chaque interaction, nous avons proposé un ensemble de règles permettant de traduire les interfaces correspondantes des composants en des sous-modèles SWNs. Les modèles SWNs des composants sont complétés et interconnectés pour former le modèle global de l'application basée composant.

Ceci a constitué la première contribution de notre travail.

Dans un deuxième temps, nous nous sommes appuyés sur des travaux précédents ayant proposé une méthode structurée pour analyser une décomposition synchrone ou asynchrone d'un modèle SWN. Nous avons étendu cette méthode pour l'analyse d'une composition mixte synchrone et asynchrone d'un ensemble de SWNs et l'avons adapté au contexte des CBS. Il en découle que les conditions d'applicabilité de la méthode structurée d'une composition synchrone ou asynchrone de SWNs peuvent être violées en présence de compositions mixtes. Pour remédier à cela, nous avons défini un ensemble de conditions pour lesquelles l'analyse structurée d'une composition mixte est possible.

Dans le contexte des CBS, nous soulignons que la modélisation que nous avons proposée dans la première partie de notre travail, a été construite de telle manière à satisfaire dès le départ les conditions d'une composition synchrone ou asynchrone de SWNs, ce qui constitue l'intérêt de notre modélisation. Toutefois, de nouvelles conditions s'imposent devant les compositions mixtes de modèles SWNs des

composants. Ces conditions ont été également redéfinies dans le contexte des CBS, en termes d'interactions entre composants.

Cette deuxième partie constitue notre seconde contribution.

Pour montrer l'intérêt de notre méthode d'analyse et les gains apportés, nous avons choisi deux modèles de composant à étudier : le modèle FRACTAL et le modèle CCM.

Le modèle FRACTAL est caractérisé par l'unique type d'interaction, qu'est l'invocation de service. Un modèle SWN de CBS sera ainsi formé uniquement de compositions synchrones entre des modèles SWNs. Ceci est quelque part au profit de notre méthode, puisque les conditions d'application de l'analyse structurée se résument en une unique condition, permettant ainsi d'élargir le champs d'application à un ensemble large de CBS FRACTAL.

De son côté, le modèle CCM dote ses composants de possibilité d'interactions par invocation de méthode et par événements. Un modèle SWN de CBS sera alors vu comme une composition mixte de modèles SWNs des composants, nécessitant ainsi de vérifier toutes les conditions d'analyse structurée.

Par ailleurs, nous remarquons, à travers l'étude des modèles FRACTAL et CCM, que l'instanciation de notre approche d'analyse structurée à des CBS suivant un modèle donné de composant peut s'opérer sans modification importante de la méthode. Cependant, il est tout à fait clair que, si le modèle de composant étudié présente certaines spécificités non prises en compte lors de notre étude, notre méthode doit être adaptée à ces spécificités.

En résumé, notre méthode d'analyse structurée des CBS permet :

- Un gain intéressant en temps de calcul et en occupation mémoire lors de l'analyse d'un CBS.
- L'analyse de configurations larges de CBS, générant un espace d'états de taille importante.
- L'analyse isolée du comportement d'un composant. En effet, lorsque le concepteur s'intéresse à étudier les performances d'un composant uniquement d'une application basée composant, il est possible d'abstraire les autres composants de l'application et détailler le composant en question, puis appliquer notre méthode et analyser cette configuration.

Néanmoins, notre modélisation exige une connaissance du formalisme SWN. Pour éviter des efforts de modélisation, il serait intéressant de développer des bibliothèques ou catalogues (*repositories*) de modèles SWN des composants d'un domaine donné, de la même façon que sont développées les bibliothèques de composants à réutiliser. La méthode d'analyse structurée se résumera alors à choisir les modèles SWNs associés aux composants d'une configuration CBS à analyser, à partir d'un catalogue donné, puis appliquer notre méthode pour la dérivation des indices de performance cibles.

Une deuxième difficulté pour l'application de notre méthode s'impose également : En raison des conditions d'application de la méthode structurée, il est possible d'avoir des configurations de modèles SWNs de composants qui ne satisfont pas ces conditions. Dans ce cas, nous avons proposé de grouper les modèles SWNs jusqu'à trouver une configuration pour laquelle les conditions sont totalement remplies. Toutefois, la granularité des modèles SWNs tend à grandir avec cette solution, ce qui ne permet pas d'exploiter pleinement la compositionnalité du CBS. Quelquefois même, il est possible d'être dans une situation extrême où tous les modèles SWNs sont groupés en un seul modèle qui est celui de l'application. Dans ce cas, l'analyse au pire se fait sur le modèle global, sans aucun profit de la méthode structurée.

En termes de perspectives, notre travail peut trouver plusieurs prolongements. Le premier prolongement naturel est la poursuite du développement des outils nécessaires visant à automatiser l'ensemble des étapes allant de la génération du modèle SWN du CBS à partir des modèles SWNs des composants jusqu'au calcul d'indices de performance requis par l'utilisateur. Ceci requiert l'écriture du code nécessaire pour la traduction d'une description d'architecture vers le modèle SWN associé, puis l'extension

de l'outil d'analyse *compSWN* pour englober les étapes de construction des réseaux étendus et le calcul d'indices de performance particuliers en se basant sur les outils *PERFSWN*.

À long terme, nous pouvons également penser à automatiser la modélisation d'un composant qui suit un modèle donné de composant, à partir de l'analyse de son code source.

Il serait intéressant aussi de développer une méthode d'abstraction des composants d'une configuration de CBS dans l'objectif d'étudier un seul composant de la configuration.

Un autre prolongement de notre travail consiste à intégrer la possibilité d'étudier l'analyse des reconfigurations dynamiques de CBS, vu l'intérêt qui est porté sur ce type d'opérations dans les systèmes actuels.

Enfin, une dernière perspective de notre travail peut être de chercher une solution permettant de choisir un niveau de granularité de composition des modèles SWNs d'un CBS, autorisant un gain optimal de la méthode structurée en temes de temps de calcul et espace mémoire requis. Cette dernière perspective est l'un des points qui nécessitera sans doute une étude approfondie.

Bibliographie

[1] *Component-Based Software Engineering: Putting the pieces together, G.Heineman, W.Councill*, chapter Overview of the CORBA Component Model. Addison-Wesley, 2001.

[2] *Handbook of Software Engineering and Knowlede Engineering*, volume vol.1. World Scientific Publishing Company, Singapore, 2001.

[3] UML profile for schedulability, performance and time specification. on line at `http://www.cgi.omg.org/docs/ptc/02-03-02.pdf`, Mar 2002.

[4] S. Jean A.E. ÖZCAN et J.B. STEFANI. Bringing ease and adaptability to mpsoc software design : A component-based approach. In *CASSIS*, pages 118–137, 2005.

[5] M. AJMONE MARSAN, G. BALBO, G. CONTE, S. DONATELLI et G. FRANCESCHINIS. *Modelling with Generalized Stochastic Petri Nets*. Wiley series in parallel computing. John Wiley & Sons, England, 1995.

[6] A. ALDINI et M. BERNARDO. On the usability of process algebra: An architectural view. *Theoretical Computer Science*, 335(2–3):281–329, May 2005.

[7] J. ALDRICH, C.C HAMBERS et D. NOTKIN. Archjava: Connecting software architecture to implementation. In *Proceeding of ICSE 2002*, May 2002.

[8] J. ALDRICH, V. SAZAWAL, C. CHAMBERS et D. NOTKIN. Language support for connector abstractions. In *Proceeding of the 2003 European Conference on Object-Oriented Programming (ECOOP 2003)*, Darmstadt, Germany, July 21-25 2003.

[9] R.J. ALLEN. *A formal approach to software architecture*. Thèse de Doctorat, School of Computer Science, Carnegie Mellon University, May 1997.

[10] R.J. ALLEN, R. DOUENCE et D. GARLAN. Specifying and analyzing dynamic software architectures. In *Proceeding of the Conference on Fundamental approaches to Software Engineering*, Lisbon, Portugal, March 1998.

[11] A. ARNOLD. Nivats processes and their synchronization. *Theor. Comput. Sci.*, (281(1-2)):31–36, 2002.

[12] A. ARNOLD. Finite transition systems: semantics of communicating systems. In *Pren-tice Hall International (UK) Ltd.* Hertfordshire, UK, 94.

[13] Holobloc Incorporation. Rockwell AUTOMATION. Function block development kit. `http://www.holobloc.com`.Last updated : December 2007.

[14] F. BACCELLI, G. BALBO, R. BOUCHERIE, J. CAMPOS et G. CHIOLA. Annotated bibliography on stochastic petri nets. In *In Proc. of the third QMIPS workshop*, volume Part 1, pages 25–44, Torino, Italy, September 1993. CWI, Amsterdam, The Netherlands.

[15] F. BACCELLI, G. COHEN, G. J. OLSDER et J.P. QUADRAT. *Synchronization and linearity*. J. Wiley & sons, 1992.

[16] S. BALSAMO, M. BERNARDO et M. SIMEONI. Combining stochastic process algebras and queueing networks for software architecture analysis. In *Proc. of WOSP2002*, pages 190–202, Rome, Italy, July 24–26 2002.

[17] M. BARNETT et W. SCHULTE. Contracts, components, and their runtime verification on the .net platform. Rapport technique, Microsoft Research-One Microsoft Way, April 2002.

[18] T. BARROS. *Formal specification and verification of distributed component systems*. Thèse de Doctorat, UNIVERSITE DE NICE-SOPHIA ANTIPOLIS - UFR Sciences, Nov 2005.

[19] T. BARROS, A. CANSADO, E. MADELAINE et M. RIVERA. Model checking distributed components: The Vercors platform. In *3rd workshop on FACS*. ENTCS, Sep 2006.

[20] T. BARROS, L. HENRIO et E. MADELAINE. Behavioural models for hierarchical components. In *Model Checking Software, 12th International SPIN Workshop*, volume LNCS 3639, pages 154–168, San Francisco, CA, USA, August 2005. Springer.

[21] L. BASS, Ch. BUHMAN, S. CORMELLA-DORDA, F. LONG, J. ROBERT, R. SEACORD et K. WALLNAU. Volume 1: Market assessment of component-based software engineering. Rapport technique Technical Note CMU/CEI-2001-TN-007, 2001.

[22] F. BAUDE, D. CAROMEL et M. MOREL. From distributed objects to hierarchical grid components. In *On The Move to Meaningful Internet Systems 2003: Coopis, DOA and ODBASE*, volume 2888 of *LNCS*, pages 1226–1242. Springer Verlag, 2003.

[23] S. BERNARDI, S. DONATELLI et A. HORVÁTH. Implementing compositionality for stochastic Petri nets. *Int. J. STTT*, (3):417–430, 2001.

[24] S. BERNARDI, S. DONATELLI et J. MERSEGUER. From UML sequence diagrams and statecharts to analysable petri net models. In *Proc. of 3rd WOSP2002*, pages 35–45, Rome, Italy, July 2002.

[25] M. BERNARDO. TwoTowers 5.0 user manual. `http://www.sti.uniurb.it/bernardo/twotowers`, 2004.

[26] M. BERNARDO, P. CIANCARINI et L. DONATIELLO. Architecting families of software systems with process algebras. *ACM Trans. on Software Engineering and Methodology*, pages 386–426, 2002.

[27] A. BERTOLINO, E. MARCHETTI et R. MIRANDOLA. Real-time UML based performance engineering to aid manager's decisions in multi-project planning. In *In Proc. of the WOSP 2002, third Int. Workshop on Software and performance*, 2002.

[28] A. BERTOLINO et R. MIRANDOLA. Modeling and analysis of non-functional properties in component-based systems. *ENTCS*, 82(6), 2003.

[29] A. BERTOLINO et R. MIRANDOLA. Software performance engineering of component-based systems. In *Proc. of the WOSP 04*, Redwood City, C.A., janvier 14–16 2004.

[30] P. BINNS, M. ENGELHART, M. JACKSON et S. VESTAL. Domain specific software for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, Vol.6(No 2), 1996.

[31] J. BINU, V. TRANG, M. VERNIK, S. WOLNY et S. YU. Performance evaluation of enterprise javabeans (EJB) CORBA adapter to CORBA server interoperability. `http://java.sun.com/developer/technicalArticles/ebeans/corba/`, janvier 2002.

[32] D. BLEVINS. *Component-Based Software Engineering: Putting the pieces together, G.Heineman, W.Councill*, chapter Overview of the Enterprise JavaBeans Component model. Addison-Wesley, 2001.

[33] G. BOOCH. *Analyse et conception orientées objets.* Editions Addison-Wesley France, SA, Octobre 1990.

[34] G. BOOCH, J.RUMBAUGH et I.JACOBSON. *The Unified Modeling Language User guide*. Editions Addison-Wesley, 1999.

[35] S. BOUCHENAK, F. BOYER, D. HAGUMONT, S. KRAKOWIAK, A. MOS, N. de PALMA, V. QUEMA et J.B. STEFANI. Architecture-based autonomous repair management : An application to J2EE clusters. In *The 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*, Orlando, FL, USA, October 2005.

[36] M. BRAVETTI et M. BERNARDO. Compositional asymmetric cooperations for process algebras with probabilities, priorities, and time. In *in Proc. of MTCS 2000*, volume 3, State College (PA), 2000. Electronic Notes in Theoretical Computer Science.

[37] S.D. BROOKES, C.A.R. HOARE et A.W. ROSCOE. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[38] E. BRUNETON. Fractal tutorial. `http://fractal.objectweb.org/tutorials/fractal/index.html` (September 12 2003), September 2003.

[39] E. BRUNETON. Fractal adl tutorial. `http://fractal.objectweb.org/tutorials/adl/index.html`, March 2004.

[40] E. BRUNETON. Tutorial : Developping with Fractal. `http://fractal.objectweb.org/tutorial/index.html` (Mars 10 2004), March 2004.

[41] E. BRUNETON, T. COUPAYE, M. LECLERCQ, V. QUÉMA et J.B. STEFANI. An open component model and its support in java. In *Proceeding of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland.

[42] E. BRUNETON, T. COUPAYE, M. LECLERCQ, V. QUÉMA et J.B. STEFANI. The fractal component model and its support in java. *Software practice and experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(n. 11-12):1257–1284, 2006.

[43] E. BRUNETON, T. COUPAYE et J.B. STEFANI. The fractal component model, version 2.0-3. Rapport technique, `http://fractal.objectweb.org/specification/` (Oct. 2006), Feb 2004.

[44] P. BUCHHOLZ. A hierarchical view of GCSPNs and its impact on qualitative and quantitative analysis. *Journal of Parallel and Distributed Computing*, 15:207–224, 1992.

[45] P. BUCHHOLZ. Aggregation and reduction techniques for hierarchical GCSPN. In *Proc.of the 5th International Workshop on Petri Nets and Performance Models*, pages 216–225. IEEE, 1993.

[46] P. BUCHHOLZ et P. KEMPER. Numerical analysis of stochastic marked graph nets. In *In Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 32–41, Durham, NC, USA, October 1995.

[47] C.ESCOFFIER et D.DONSEZ. Fractnet web site. `http://www-adele.imag.fr/fractnet`. Last accessed: March 2008.

[48] J.M. CHAUVET. *Composants et transactions, comprendre l'architecture des serveurs d'applications*. Eyrolles, 1999.

[49] J. CHEESMAN et J. DANIELS. UML components, a simple process for specifying component-based software. 2001.

[50] G. CHIOLA. Greatspn 1.5 software architecture. Rapport technique, Università di Torino, April 1991.

[51] G. CHIOLA, C. DUTHEILLET, G. FRANCESCHINIS et S. HADDAD. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Trans. on Comp.*, 42(11):1343–1360, Nov 1993.

[52] S. CHRISTENSEN et K.H. MORTENSEN. Design/CPN ASK-CTL manual. University of Aarhus, 0th edition, 1996.

[53] G. CIARDO. Distributed and structured analysis approaches to study large and complex systems. *Lecture Notes in Computer Science*, 2090, 2001.

[54] E. CINLAR. *Introduction to stochastic processes*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1975.

[55] Jr.E.M. CLARKE, O. GRUMBERG et D.A. PELED. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[56] COLLECTIF, SOUS LA DIRECTION DE M. OUSSALAH. *Ingénierie des composants. Concepts, techniques et outils*. Editions Vuibert Informatique, 20, rue Berbier-du-Mets, 75013 Paris, 2005.

[57] J. COMPOS, M. SILVA et S. DONATELLI. Structured solution of stochastic DSSP systems. In *In Proc. 7th Int Work. On PNPM'1997*, pages 91–100, 1997.

[58] J. C. CORBETT, M. B. DWYER, J. HATCLIFF et ROBBY. A language framework for expressing checkable properties of dynamic software. In *In Proc. of the SPIN Software model checking workshop*, volume 1885, August 2000.

[59] Meta Software CORP.. Design/CPN reference manual, 1992.

[60] V. CORTELLESSA et R. MIRANDOLA. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming. Elsevier Science*, 2002.

[61] B.J. COX. *Object-Oriented programming - An evolutionary approach.* Addison-Wesley, 1986.

[62] C.U.SMITH et L.G.WILLIAMS. *Performance Solutions*. Addison-Wesley, 2002.

[63] F. DAGNAT. L'approche composant. un état de l'art et de la pratique. `http://info.enstb.org/enseignement/fc/composants/cours.pdf`. Last accessed: March 2008, Octobre 2005. Formation continue. Enst Bretagne,

[64] M. DAVIO. Kronecker products and shuffle algebra. *IEEE Transactions on Computers*, 30(2):116–125, 1981.

[65] C. DELAMARE. *Étude de grands systèmes stochastiques symétriques par réseaux décomposables*. Thèse de Doctorat, Université de Reims Champagne Ardenne, Reims, France, Octobre 2003.

[66] C. DELAMARE, Y. GARDAN et P. MOREAUX. Efficient implementation for performance evaluation of synchronous decomposition of high level stochastic Petri nets. In *Proc. of the ICALP2003*, pages 164–183, Eindhoven, Holland, juin 21-22 2003. University of Dortmund, Germany.

[67] C. DELAMARE, Y. GARDAN et P. MOREAUX. Performance evaluation with asynchronously decomposable SWN: implementation and case study. In *Proc. of the 10th Int. Workshop on PNPM03*, pages 20–29, Urbana-Champaign, IL, USA, septembre 2–5 2003. IEEE Comp. Soc. Press.

[68] C. DEMARTINI, R. IOSIF et R. SISTO. DSpin : A dynamic extension of SPIN. *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, September 1999.

[69] P.J. DENNING et J.P. BUZEN. Operationnal analysis of stochastic closed queueing networks. 10(3):225–262, 1978.

[70] F. DEREMER et H. KRON. Programming-in-the large versus programming-in-the small. In *Proceeding of the Int.Conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM Press.

[71] L. DIAS DA SILVA et A. PERKUSICH. Composition of software artifacts modelled using colored Petri nets. *Science of Computer Programming*, 56(1-2):171–189, Apr 2005.

[72] R.M. DIJKMAN, J.P. Andrade ALMEIDA et D.A.C. QUARTEL. Verifying the correctness of component-based applications that support business processes. In *Sixth ICSE Workshop on Component-Based Software Engineering (CBSE 2003)*, pages 43–48, Portland, OR, United States (2003-05-03/2003-05-04), May 2003. Carnegie Mellon University and Monash University.

[73] E.W. DIJKSTRA. The structure of the multiprogramming system. *Communications of the ACM 11*, n.5:341–346, 1968.

[74] S. DONATELLI. Superposed stochastic automata: A class of stochastic petri nets with parallel solutionand distributed state space. *Performance evaluation*, 18:21–36, 1993.

[75] S. DONATELLI. Superposed generalized stochastic Petri nets : definition and efficient solution. In *Proc.of the 15th International Conference on Application and Theory of Petri Nets*, volume 815, pages 258–277. LNCS, Springer Verlag, 1994.

[76] C. DUTHEILLET. Symétries dans les réseaux colorés. définition, analyse et application à l'évaluation de performance. Master's thesis, 1991.

[77] T. EWALD. *Component-Based Software Engineering: Putting the pieces together*, chapter Overview of COM+. Addison-Wesley, 2001.

[78] J. FASSINO, J. STEFANI, J. LAWALL et G. MULLER. Think: A software framework for component-based operating system kernels. In *Usenix Annual Technical Conference*, Monterey (USA), June 2002.

[79] S. FDIDA et G. PUJOLLE. *Modèles de Systèmes et de Réseaux*. Eyrolles, Paris, 1989. Tome 1: Performance,

[80] A.V. FIOUKOV, E.M. ESKENAZI, D. HAMMER et M. CHAUDRON. Evaluation of static properties for component-based architectures. In *Proceedings of 28th EUROMICRO conference, Component-based Software Engineering track*, Sept 2002.

[81] G. FLORIN. *Réseaux de Petri stochastiques. Théorie, techniques de calcul, applications*. Thèse de doctorat, Institut de programmation, Université Pierre et Marie Curie Paris VI, Paris, France, Juin 1985.

[82] D. FREEMAN. *Markov chains*. Springer-Verlag, 1983.

[83] E. GAMMA, R. HELM, R. JOHNSON et J. VLISSIDES. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[84] D. GARLAN. Software architecture and object-oriented systems. In *Proceeding of the IPSJ Object-Oriented Symposium*, Rokyo, Japan, August 2000.

[85] D. GARLAN et S. KHERSONSKY. Model checking implicit invocation systems. In *In Proc. of the 10th Workshop on Software Specification and Design*, Nov 2000.

[86] D. GARLAN, R. MONROE et D. WILE. Acme : An architecture description interchange language. In *Proceeding of CASCON'97*, Toronto, Canada, November 1997.

[87] D. GARLAN, R. MONROE et D. WILE. Acme : Architectural description of component based systems. *Leavens Gary and Sitaraman Murali, Foundations of Component-Based systems*, pages pp 47–68, 2000. Cambridge University Press,

[88] H.J. GENRICH et K. LAUTENBACH. System modelling with high-level petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[89] D. GIANNAKOPOULOU, J. KRAMER et S. Chi CHEUNG. Behaviour analysis of distributed systems using the Tracta approach. *Automated Software Engineering*, 6(1):7–35, 1999.

[90] D. GIANNAKOPOULOU, J. KRAMER et J. MAGEE. Practical behaviour analysis for distributed software architectures. In *UK Programmable Networks and Telecommunications Workshop*, Hewlett-Packard Laboratories, Bristol, September 1998. `http://www.informatics.sussex.ac.uk/research/projects/safetynet/prognet/statements/JeffreyKramer.html`,

[91] I. GORTON, I. JELLY et J. GRAY. Parallel software engineering with PARSE. In *In proceedings of the 17th International Computer Software and Applications Conference*, pages 124–130, Phoenix, USA, Nov 1993.

[92] I. GORTON, I. JELLY, J. GRAY et T. S. CHAN. Reliable parallel software construction using parse. *Concurrency practice and experience*, 8(2):125–146, March 1996.

[93] V. GRASSI et R. MIRANDOLA. Towards automatic compositional performance analysis using queueing network models. In *In Proc. Wosp 2004*, 2004.

[94] V. GRASSI, R. MIRANDOLA et A. SABETTA. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *J. Syst. Softw.*, 80(4):528–558, 2007.

[95] J. GRAY, I. GORTON et I. JELLY. CASE tools for PARSE. In *In Proceedings of Parallel Computing and Transputers PCAT-93*, pages 343–349, Brisbane, November 1992. IOS Press.

[96] Y. GUREVICH. Evolving algebras 1993: Lipari guide. *Specification and Validation Methods. E.Börger editor*, 1995. Oxford University Press,

[97] S. HADDAD et P. MOREAUX. Aggregation and decomposition for performance evaluation of synchronous product of high level Petri nets. Document du Lamsade 96, LAMSADE, Université Paris Dauphine, Paris, France, septembre 1996.

[98] S. HADDAD et P. MOREAUX. Asynchronous composition of high level Petri nets : a quantitative approach. In *Proc. of the 17th ICATPN*, volume 1091, pages 193–211. LNCS, 1996.

[99] S. HADDAD, P. MOREAUX et M. SENE. Performance evaluation with SWN: a technical contribution. *Réseaux et systèmes répartis - Calculateurs parallèles*, 2001. To appear,

[100] S. HADDAD et F. VERNADAT. *Les réseaux de Petri*, chapter Méthodes d'analyse des réseaux de Petri. 2001.

[101] R. HAMOUCHE. *Modélisation des systèmes embarqusés à base de composants et d'aspects*. Thèse de Doctorat, Université d'Evry, Mai 2004.

[102] D. HAREL. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.

[103] J. HARRIS et A. HENDERSON. A better mythology for system design. In *Proceedings of CHI99 Computer Science-Human computer Interaction conference*, Pittsburgh, USA, May 1999.

[104] J. HATCLIFF, W. DENG, M. DWYER, G. JUNG et V. PRASAD. Cadena: an integrated development, analysis and verification environment for component-based systems. In *In Proceedings of the 25th International Conference on Software Engineering*, 2003.

[105] X. HE, H. YU, T. SHI, J. DING et Y. DENG. Formally analyzing software architectural specifications using SAM. *Journal of Systems and Software*, 71(1–2):11–29, 2004.

[106] C.A.R. HOARE. *Communicating sequential processes.* Inc. Upper Saddle River, NJ, USA, 1985.

[107] G. J. HOLZMANN. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[108] K. JENSEN. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag, 1992.

[109] K. JENSEN. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume vol.2. Springer-Verlag, 1997.

[110] K. JENSEN et AL.. Design/CPN 4.0. on-line version: `http://www.daimi.au.dk/design/CPN/`, 1999.

[111] K. JENSEN et G. ROZENBERG. *High-Level Petri Nets*. Theory and Application, 1991.

[112] M. De JONGE, J. MUSKENS et M. CHAUDRON. Scenario-based prediction of run-time resource consumption in component-based software systems. In *6th International Conference on Software Engineering (ICSE) Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, Portland, Oregon, USA, May 3-4 2003.

[113] M. KEMPA et V. LINNEMANN. XML-based applications using XML schema. In *XML-based Data Management and Multimedia Engineering-EDBT Workshops*, pages 67–90. Lecture Notes in Computer Science, 2002.

[114] P. KEMPER. Numerical analysis of superposed GSPN. *IEEE Transactions on Software Engineering*, 22(4):615–628, September 1996.

[115] M. KHALGUI, X. REBEUF et F. SIMONOT-LION. A behavior model for IEC 61499 function blocks. In *MOCA04*, Denmark, 2004.

[116] M. KHALGUI, X. REBEUF et F. SIMONOT-LION. A schedulability condition for an IEC 61499 control application with limited buffers. In *FET 05*, Mexico, 2005.

[117] M. KHALGUI, X. REBEUF et F. SIMONOT-LION. Component-based deployment of industrial control systems : an hybrid scheduling approach. In *ETFA06*, Czech, 2006.

[118] M. KHALGUI, X. REBEUF et F. SIMONOT-LION. A tolerant temporal validation of component based applications. In *In 12th IFAC Symposium on Information Control Problems in Manufacturing INCOM06*, St-Etienne, France, May 2006.

[119] P. KING et R. POOLEY. Using UML to derive stochastic petri net models. `http://citeseer.ist.psu.edu/245149.html`,

[120] L. KLEINROCK. *Queueing systems. Volume I : Theory*. Wiley-Interscience, New-York, 1975.

[121] L. KLEINROCK. *Queueing systems. Volume II:*. Wiley-Interscience, New-York, 1976.

[122] S. KRAKOWIAK. *Middleware Architecture with Patterns and Frameworks*. Creative Commons license, June 2007.

[123] J.C. LAPRIE, J. ARLAT, J.P. BLANQUART, A. COSTES, Y. CROUZET, Y. DESTWARTE, J.C. FABRE, H. GUILLERMAIN, M. KAANICHE, K. KANOUN, C. MAZET, D. POWELL, C. RABJAC et P. THVENOD. *Guide de la Sûreté de Fonctionnement*. Cépaduès Editions, 1995.

[124] O. LAYAIDA et D. HAGIMONT. Designing self-adaptive multimedia applications through hirerachical reconfiguration. *Dais, LNCS*, vol.3543, 2005. Springer,

[125] E. LAZOWSKA, J. KAHORJAN, G.S. GRAHAM et K.C. SEVCIK. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, 1984.

[126] B. LEWIS, E. COLBERT et S. VESTAL. Developing evolvable, embedded, time critical systems with metah. In *Proceeding of the TOOLS 2000*, 2000.

[127] R. LEWIS. *Modelling Control systems using IEC61499*. The Institution of Electrical Engineers, 2001.

[128] H. LIN. Symbolic transition graph with assignment. In *In Proc. of CONCUR 96*, volume 1119, Pisa, Italy, August 1996. LNCS.

[129] Y. LIU, A. FEKETE et I. GORTON. Predicting the performance of middleware-based applications at the design level. In *WOSP'04: Proceedings of the 4th international workshop on Software and performance*, pages 166–170, New York, NY, USA, 2004. ACM.

[130] Y. LIU, A. FEKETE et I. GORTON. Design-level performance prediction of component-based applications. *IEEE Transactions on Software Engineering*, 31(11):928–941, 2005. Member-Yan Liu and Member-Alan Fekete and Member-Ian Gorton,

[131] Y. LIU et I. GORTON. Accuracy of performance prediction for ejb applications: A statistical analysis. In *Software Engineering and Middleware*, volume 3437 of *LNCS*, pages 185–195. Springer, 2005.

[132] A. LOBOV, J.L. Martinez LASTRA, R. TUOKKO et V. VYATKIN. Methodology for modeling visual flowchart control programs using net condition/event systems formalism in distributed environments. In *Proceedings off the 9th IEEE Conference on Emerging Technologies in Factory Automation (ETFA'03)*, Portugal, 2003.

[133] J.P. LÓPEZ-GRAO, J.MERSEGUER et J.CAMPOS. From UML activity diagrams to stochastic Petri nets: Application to software performance engineering. In *In Proc. of WOPS'04*, Jan 2004.

[134] D.C. LUCKHAM, J. VERA, L. AUGUSTIN et F. BELZ. Partial ordering of event sets and their application to prototyping concurrent timed systems. *Journal of systems and software*, vol.21(no.3):253–265, June 1993.

[135] J. MAGEE, N. DULAY, S. EISENBACH et J. KRAMER. Specifying distributed software architectures. In *On Proceeding of the fifth European Software Engineering Conference*, Barcelona, Spain, September 1995.

[136] J. MAGEE, N. DULAY et J. KRAMER. A constructive development environment for parallel and distributed programs. In *In International Workshop on Configurable Distributed Systems*, March 1994.

[137] J. MAGEE et J. KRAMER. Dynamic structure in software architecture. In *On Proceeding of ACM/SIGSOFT'96: Fourth Symposium on Foundations of Software Engineering (FSE4)*, pages 3–14, Barcelona, Spain, September 1996.

[138] J. MAGEE et J. KRAMER. *Concurrency : State models and JAVA programs*. Wiley, 1999.

[139] R. MARMORSTEIN, G. CIARDO et R. SIMINICEANU. Saturation unbound. In *In Proc.of Tools and Algorithms for construction and analysis of systems (TACAS)*, number number 2612 in LNCS, pages 379–393, College of Willian and Mary, Williamsburg, Virginia 23187, Avril 7-7 2003. Springer-Verlag.

[140] D. MARSHALL. *.NET Security Programming*, chapter .NET ARCHITECTURE. Wiley Publishers, April 2003. `http://www.windowsitlibrary.com/Content/905/01/toc.html`,

[141] D. MCILROY. Mass-produced software components. In *Proceedings of the 1st International Conference on Sotware Engineering*, Garmisch Pattenkirschen, Germany, October 1968.

[142] M. MEDJDOUD. *Contribution à l'analyse des systèmes pilotés par calculateurs : Extraction de scénarios redoutés et vérification de contraintes temporelles*. Thèse de Doctorat, Université Paul Sabatier de Toulouse, Toulouse, Mars 2006.

[143] N. MEDVIDOVIĆ, D. ROSENBLUM et R.N. TAYLOR. A language and environment for architecture-based software development and evolution. In *Proceeding of the 21st International Conference on Software Engineering (ICSE'99)*, pages 44–53, 1999.

[144] N. MEDVIDOVIĆ et R. N. TAYLOR. A classification and comparison framework for software architecture description languages. In *IEEE Trans. On Soft. Eng.*, volume 26, pages 70–93, 2000.

[145] N.R. MEHTA et N. MEDVIDOVIĆ. Understanding software connector compatibilities using a connector taxonomy. In *Proceedings of First Workshop on Software Design and Architecture*, Bangalore, India, 2002.

[146] J. MERSEGUER, S. BERNARDI, J. CAMPOS et S. DONATELLI. A compositional semantics for UML state machines aimed at performance evaluation. In *In Proc. of the 6th Int. Workshop on Discrete Event Systems (WODES2002)*, pages 295–302, Zaragoza, Spain, October 2002. IEEE Computer society Press.

[147] J. MERSEGUER et J. CAMPOS. Exploring roles for the uml diagrams in software performance engineering. In *In Proc. of the 3th Int. Conference on Software Engineering research and Practise (SERP'03)*, pages 43–47, Las Vegas, USA, June 2003. CSREA Press.

[148] B. MEYER. On to components. Rapport technique, IEEE Computer, January 1999.

[149] MICROSOFT. .Net 3.0 framework. `http://msdn.microsoft.com/netframework` (July 2007), 2007.

[150] R. MILNER. *Communication and Concurrency*. Prentice Hall, 1989.

[151] R. MILNER. Operational and algebraic semantics of concurrent processes. pages 1201–1242, 1990.

[152] R. MILNER. *Communications and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[153] R. MILNER, J. PARROW et D. WALKER. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–77, 1992.

[154] A.S. MINER et G. CIARDO. A data structure for the efficient kronecker solution of GSPNs. In *In Proc. of the 8th Int. Workshop on Petri nets and performance models (PNPM99)*, pages 22–31, Zaragoza, Spain, September 8-10 1999. IEEE Computer Society Press.

[155] A.S. MINER et G. CIARDO. Efficient reachability set generation and storage using decision diagrams. In *Proc. of the 20th International Conference on Application and Theory of Petri Nets ICATPN'99*, number 1639 in LNCS, pages 6–25, Williamsburg, VA, USA, juin 21–25 1999. Springer–Verlag.

[156] I. MITRANI. *Simutation Techniques for Discrete Events Systems*. Cambridge University Press, Cambridge, 1982.

[157] M.K. MOLLOY. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, C-31(9):913–917, September 1982.

[158] P. MOREAUX. Structuration des chaînes de markov des réseaux de petri stochastiques. décomposition tensorielle et agrégation. Master's thesis, 1996.

[159] P. MÜLLER, C. STICH et C. ZEIDLER. Components@work: Component technology for embedded systems. In *Proc. of the Component-based Software Engineering Track at the 27th IEEE Euromicro Conference (Euromicro CBSE'01)*, Sep 2001.

[160] T. MURATA. Petri nets : properties, analysis and applications. In *Proc. of IEEE*, volume 77, pages 541–580, April 1989.

[161] S. NATKIN. *Les réseaux de Petri stochastiques et leur application l'évaluation des systèmes informatiques*. Thèse de docteur ingénieur, CNAM, Paris, France, juin 1980.

[162] K. NG, J. KRAMER, J. MAGEE et N. DULAY. A visual approach to distributed programming. In *In Tools and Environments for Parallel and Distributed Systems*, pages 7–31. Kluwer Academic Publishers, February 1996.

[163] O. NIERSTRASZ, G. AREVALO, S. DUCASSE, R. WUYTS, A. BLACK, P. MÜLLER, C. ZEIDLER, T. GENSSLER et R. Van der BORN. A component model for field devices. In *Proceedings First International IFIP/ACM Working Conference on Component Deployment*, Berlin, Germany, June 2002.

[164] OBJECT MANAGEMENT GROUP. The common object request broker: Architecture and specification, 3.0.2 edition. Rapport technique, OMG, Dec 2002.

[165] OBJECT MANAGEMENT GROUP. UML2, ptc/03-08-02. August 2 2003. accessible from `http://www.omg.org/docs/ptc/03-08-02.pdf`,

[166] OBJECT MANAGEMENT GROUP. Common object request broker architecture (CORBA) - specification, version 3.1, part 1: CORBA interoperability. `http://www.omg.org/cgi-bin/doc?pas/04-08-01.pdf` (July 2007), 2004.

[167] OBJECT MANAGEMENT GROUP. Common object request broker architecture (CORBA) - specification, version 3.1, part 2: CORBA interfaces. `http://www.omg.org/cgi-bin/doc?pas/04-08-02.pdf` (July 2007), 2004.

[168] OBJECT MANAGEMENT GROUP. Notification service. version 1.1. http://www.omg.org/technology/documents/formal/notification_service.htm (April 2007), octobre 2004.

[169] OBJECT MANAGEMENT GROUP. CORBA component model specification. version 4.0. http://www.omg.org/cgi-bin/apps/doc?formal/06-04-01.pdf (April 2007), avril 2006.

[170] J. PADBERG, H. WEBER et A. SUNBUL. Petri net based components for evolvable architectures. In *In Proc. of Integrated Design and Process Technology*, 2000.

[171] D. PARNAS. On the criteria for decomposing systems into module. *Communications of the ACM 15*, n.12:1053–1058, 1972.

[172] R. PASSAMA. *Conception et déveoppement de contrôleurs de robots. Une méthodologie basée sur les composants logiciels*. Thèse de Doctorat, Université Montpellier II, 30 Juin 2006.

[173] PERF. EVAL. GROUP. GreatSPN home page: http://www.di.unito.it/~greatspn, 2002.

[174] C.A. PETRI. *Kommukationen MIT Automaten*. Thèse de Doctorat, University of Bonn, 1962.

[175] D. PETRIU, C. SHOUSHA et A. JALNAPURKAR. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[176] D.C. PETRIU et H. SHEN. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specification. In *Proc. of Performance Tools 2002*, London, England, April 14–17 2002. Springer-Verlag.

[177] D.C. PETRIU et C.M. WOODSIDE. Approximate mean value analysis based on markov chain aggregation by composition. *Elsevier Science journal*, 386C:335–358, 2004. Linear Algebra and its Applications,

[178] B. PHILIPPE, Y. SAAD et W.J. STEWART. Numerical methods in markov chain modeling. Rapport technique INRIA Report 1115, INRIA, Rocquencourt, Rocquencourt, France, November 1989.

[179] F. PLASIL, D. BALEK et R. JANECEK. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, May 1998. IEEE CS Press.

[180] F. PLASIL, M. BESTA et S. VISNOVSKY. Bounding component behavior via protocols. In *Proceeding of Technology of Object-oriented languages and Systems (TOOLS'99)*, Santa Barbara, USA, 1999.

[181] B. PLATEAU. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *In Proc. of SIGMETRICS Conference*, pages 147–154, Austin, TX, USA, August 1985. ACM.

[182] B. PLATEAU et J.M. FOURNEAU. A methodology for solving markov models of parallel systems. *Journal of Parallel and Distributed Computing*, 12:370–387, 1991.

[183] R. PRIETO-DIAZ et J.M. NEIGHBORS. Module interconnection languages. *The journal of systems and software*, Vol.6, 1986.

[184] Philips RESEARCH. *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, chapter A Robust Component Model for Consumer Electronic Products, pages 167–192. Springer Netherlands, March 30 2006. ISBN:978-1-4020-3453-4 (Print) 978-1-4020-3454-1 (Online),

[185] ROBBY, M. B. DWYER et J. HATCLIFF. Bogor: an extensible and highly-modular model checking framework. Rapport technique SANTOS-TR2003-3, Kansas City University, 2003.

[186] ROBBY, M. B. DWYER et J. HATCLIFF. Bogor website. `http://www.cis.ksu.edu/bandera/bogor`, 2003.

[187] R.Y. RUBINSTEIN. *Monte Carlo Optimization, Simulation and Sensitivity of Queueing Networks*. J. Wiley & sons, 1986.

[188] A.E. RUGINA, K. KANOUN et M. KAANICHE. A system dependability modeling framework using AADL and GSPNs. Rapport technique 05666, LAAS, Nov 2006.

[189] N. SALMI, P. MOREAUX et M. IOUALALEN. Formal models of fractal component based systems for performance analysis. In *Proc. of ISoLA 2007 Workshop On Leveraging Applications of Formal Methods, Verification and Validation*, pages 49–60, Poitiers, France, 2007.

[190] N. SALMI, P. MOREAUX et M. IOUALALEN. Structured analysis of component based systems: an EJB/CORBA middleware application. In *Proc. of the 1st Int. Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'2007)*, Algiers, Algeria, 2007.

[191] N. SALMI, P. MOREAUX et M. IOUALALEN. From architectural design to swn models for compositional performance analysis of component based systems: application to ccm based systems. In *Proc. of the 24th UKPEW 2008 Performance Engineering Workshop*, pages 123–136, Imperial College. London, UK, 2008.

[192] C.H. SAUER et E.A. MACNAIR. *Simulation of Computer Communication Systems*. Prentice Hall, Englewood Cliffs, NJ, USA, 1983.

[193] C.H. SAUER, M. REISER et E.A.MACNAIR. RESQ - a package for solution of generalized queueing networks. In *Proceedings of the National Computer Conference*.

[194] D.C. SCHMIDT et S. VINOSKI. Object interconnections: The CORBA component model: Part 2, defining components with the IDL 3.x types. *C/C++ Users Journal*, April 2004.

[195] P. SCHNOEBELEN. *Vérification de Logiciels: Techniques et outils du model-checking*. Vuibert, Paris, 1999. ouvrage collectif, coordination P. Schnoebelen,

[196] L. SEINTURIER, N. PESSEMIER, L. DUCHIEN et T. COUPAYE. A component model engineered with components and aspects. In *Proc. of CBSE'06*, volume 4063 of *LNCS*, pages 139–153, Mälardalen University, Västerås, Sweden, jun 2006. Springer.

[197] M. SHAW, R. DELINE, D. KLEIN, T. ROSS, D. TOUNG et G. ZELESNIK. Abstraction for software architecture and tools to support them. *IEEE Transactions.Software Engineering*, (SE-21(4)):315–335, Avril 1995.

[198] M. SHAW, R. DELINE, D. KLIEN, T.L. ROSS, D.M. YOUNG et G. ZELESNIK. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, April.

[199] M. SHAW, R. DELINE et G. ZELESNIK. Abstraction and implementations for architectural connections. In *Proceeding of the third International Conference on Configurable Distributed Systems*, May 1996.

[200] C.U. SMITH. *Performance Engineering of Software Systems*. Addison-Wesley, Reading, Mass., 1990.

[201] C.U. SMITH et M. WOODSIDE. Performance validation at early stages of software development. pages 383–396, Boca Raton, FL, USA, 2000. CRC Press, Inc.

[202] W. J. STEWART. *Introduction to the numerical solution of Markov chains*. Princeton University Press, USA, 1994.

[203] SUN MICROSYSTEMS. J2EE connector architecture. Rapport technique, Sun Microsystems, Inc., `http://java.sun.com/j2ee/connector`(jan. 2006), 2006.

[204] SUN MICROSYSTEMS. EJB 3.0 specification. `http://java.sun.com/products/ejb/docs.html`, jul 2007.

[205] C. SZYPERSKI. *Component software*, volume 2nd Edition. 2002.

[206] C. SZYPERSKI. Component technology - what, where, and how? In *Proc. 25th Int. Conf. on Software Engineering*, pages 684–693. IEEE, mai 3–10 2003.

[207] A. TARHINI, A. ROLLET et H. FOUCHAL. A pragmatic approach for robustness testing on real time component based systems. In *In The 3rd ACS/IEEE International Conference on Computer Systems and Application (AICSSA05)*, Cairo, Egypt, January 2-5 2005.

[208] K.C. THRAMBOULIDIS et C.S. TRANORIS. Developing a CASE tool for distributed control applications. *The Int. Journal of Advanced Manufacturing Technology*, 24(1-2), July 2004. Springer-Verlag,

[209] V. TRAAS et J.Van HILLEGERSBERG. The software component market on the internet current sytus and conditions for grows. *Software Engineering Notes*, Vol.25(N.1), January 2000.

[210] International Telecommunications UNION. ITU-TS recommendationz.120: Message Sequence Charts (MSC), 1996.

[211] R. van OMMERING, F. van der LINDEN, J. KRAMER et J. MAGEE. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, mars 2000.

[212] M. VERAN et D. POTIER. Qnap2 : A portable environment for queueing systems modelling. Rapport technique 314, INRIA-Rocquencourt, France, June 1984. `http://www.inria.fr/rrrt/rr-0314.html`,

[213] T. VERGNAUD. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées*. Thèse de Doctorat, ENST, Paris, 1er Décembre 2006.

[214] S. VESTAL. Métah users manual, version 1.2. Rapport technique, Honeywell Technology Center, 1998.

[215] V. VITTORINI, M. IACONO, M. MAZZOCCA et G. FRANCESCHINIS. OsMoSys: a new approach to multi-formalism modeling of systems. *Journal of software and system modeling*, 3(1):68–81, mar 2004.

[216] V. VYATKIN et H. HANISCH. Formal-modelling and verification in the software engineering framework of iec 61499 : a way to self-verifying systems. In *ETFA01*, Nice, 2001.

[217] N. WANG et C. RODRIGUES. Tutorial on corba component model (CCM). BBN Technologies and Washington University, July 6th 2003.

[218] Y. WEI. *Implementation of IEC61499 Distributed Function Block Architecture for Industrial Measurement and Control Systems (IPMCS)*. Thèse de Doctorat, National University of Singapore, 2002.

[219] M. WINTER, T. GENSSLER, A. CHRISTOPH, O. NIERSTRASZ, S. DUCASSE, R. WUYTS, G. AREVALO, P. MÜLLER, C. STICH et B. SCHÖNHAGE. Components for embedded software. the PECOS approach. In *Second International Workshop on Composition Languages, In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP)*, Málaga, Spain, June 11 2002.

[220] C. WOODSIDE, J. NEILSON, S. PETRIU et S. MJUMDAR. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transaction on Computer*, 1995.

[221] M. WOODSIDE. Tutorial introduction to layered modeling of softwage performance. Accessible from `http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialg.pdf`, May 2002. Edition 3.0,

[222] X. WU et M. WOODSIDE. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.

[223] J. XU, A. OUFIMTSEV, M. WOODSIDE et L. MURPHY. Performance modeling and prediction of enterprise JavaBeans with layered queueing network templates. *ACM*, 2005.

[224] A. ZENIE. *Les réseaux de Petri stochastiques colorés : Application à l'analyse des systémes répartis en temps réel*. Thèse de Doctorat, Université Pierre et Marie CURIE, PARIS 6, Octobre 1987.

[225] Y. ZHU et J. GAO. A method to calculate the reliability of component-based software. *ats*, 0:488, 2003.

[226] R. ZIAEI et G. AGHA. Synchnet: a Petri net based coordiantion language for distributed objects. In *In Proc. of the Second International Conference on Generative Programming and Component Engineering*, pages 324–343, New York Inc., 2003.

Annexes

ANNEXE A
Annexe

A.1 Traduction d'une description AADL vers un réseau de Petri, travaux [213]

La traduction d'une architecture AADL en réseau de Petri s'effectue sur l'architecture instanciée globale (les instances de composants sont considérées et non leurs déclarations). L'intérêt a été principalement porté sur les composants applicatifs actifs (threads et sous-programmes) et les instances de composants de donnée. Les composants AADL décrivant la plate-forme d'exécution sont ignorés. Les autres composants logiciels (groupes de threads et processus) et les systèmes sont considérés comme de simples conteneurs.

Etant donné qu'une description AADL est constituée de composants acceptant des données en entrée et produisant d'autres en sortie, les flux de données et d'exécution circulant à travers l'architecture sont modélisés par des jetons. Les jetons sont colorés : les classes de couleurs sont respectivement la classe de flux de données, et la classe des flux de contrôle dans les threads. Les transitions modélisent les composants actifs qui consomment et produisent des données, et les connexions AADL. Les places symbolisent les entités permettant de stocker des données. Une instance composant de donnée est traduite par une place. Les composants actifs de haut niveau (systèmes, processus, threads) sont traduits par une transition, représentant le composant lui-même, entourée de places modélisant ses éléments d'interface. La représentation de l'implantation d'un composant consiste à remplacer cette transition par le réseau de Petri décrivant les séquences d'appel décrits dans l'implantation du composant.

Un port d'entrée événement/donnée ou un accès à un sous-composant de donnée est traduit par une place connectée à la transition du composant. Tous les ports de sortie sont trduits par une place de sortie reliée à la transition du composant. Un processus ou un système contenant des sous-composants n'est pas traduit en réseau de Petri. Seuls ses sous-composants sont transcris dans le réseau de Petri. Les différents composants englobants ne sont pas traduits en réseau de Petri. Ainsi, un composant possédant des ous-composants est systèmatiquement remplacé par ces derniers. Seuls les sous-composants sont représentés donc dans un réseau de Petri.

Chaque sous-programme est modélisé dans le réseau autant de fois qu'il est appelé. Chaque appel de sous-forme est donc assimilié à une instance, çàa à un appel normal au sous-programme correspondant. Un appel de sous-programme est traduit par une transition et des places correspondant à ses différents paramètres. Le flux d'exécution contrôlant l'appel du sous-programme est matérialisé par deux places transmettant un jeton de contrôle à la transition du sous-programme. Le jeton de contrôle est celui du thread dans lequel l'appel est effectué.

Ainsi, le réseau de Petri correspondant à une architecture AADL de haut niveau est la mise à plat de toutes les instances de threads qui représentent les entités réellement impliquées dans les flux d'exécution et de données.

Les connexions AADL représentant une transmission de données par des transitions. Contrairement aux transitions des composants, ces transitions ne correspondent pas à une transformation des données;

elles permettent simplement d'exprimer la coordination de la transmission des données. Les connexions aux composants de donnée reviennent à fusionner la place modélisant le composant de donnée considéré avec la place correspondant à l'élément d'interface correspondant. Seules les connexions directes sont considérées. La connexion de deux ports AADL se traduit par une transition. Les multiples connexions d'un port de sortie à plusieurs ports d'entrée sont traduites par une seule transition qui modélise la duplication des données transférées. Une connexion de port d'événement/donnée se traduit par une simple transition. Une connexion de port de donnée se traduit par une transition consommant à la fois le jeton transmis et le jeton placé dans la place de destination. Le jeton précédemment stocké dans la place de destination est ainsi remplacé par le nouveau jeton.

A.2 Code d'implémentation du Système de gestion des stocks des Bourses (suite)

```
home StockBrokerHome manages StockBroker {};
home StockDistributorHome manages StockDistributor{};

component StockDistributor supports Trigger {};
component StockBroker {};

local interface CCMContext {
        Principal get_caller_principal ();
        CCMHome get_CCM_home ();
        boolean get_rollback_only () raises (IllegalState);
        Transaction::UserTransaction get_user_transaction () raises
(IllegalState);
        boolean is_caller_in_role (in string role);
        void set_rollback_only () raises (IllegalState);
};

local interface SessionContext : CCMContext {
    Object get_CCM_object () raises (IllegalState);
};

eventtype StockName {
public string name;
};

local interface EnterpriseComponent { };

local interface SessionComponent : EnterpriseComponent {
    void set_session_context ( in SessionContext ctx) raises (CCMException);
    void ccm_activate () raises (CCMException);
    void ccm_passivate () raises (CCMException);
    void ccm_remove () raises (CCMException);
```

```
};

local interface CCM_StockBroker : ::Components::EnterpriseComponent {
       void push_notifier_in (in ::StockName e);
};
local interface CCM_StockBrokerHomeImplicit  {
    Components::EnterpriseComponent create ();
};
local interface CCM_StockBrokerHomeExplicit : Components::HomeExecutorBase {};
local interface CCM_StockBrokerHome : CCM_StockBrokerHomeExplicit,
                                      CCM_StockBrokerHomeImplicit {};
local interface StockBroker_Exec : CCM_StockBroker,
                                   Components::SessionComponent {};
local interface StockBrokerHome_Exec : ::CCM_StockBrokerHome {};


local interface CCM_StockBroker_Context : ::Components::SessionContext {
     StockQuoter get_connection_quoter_info_in ();
};
```

A.3 Code d'implémentation de l'exemple du Serveur Comanche

Voici le code source complet de l'exemple du serveur Comanche.

```
/* =============== Component interfaces =============== */

public interface RequestHandler {
  void handleRequest (Request r) throws IOException;
}

public interface Scheduler {
  void schedule (Runnable task);
}

public interface Logger {
  void log (String msg);
}

public class Request {
  public Socket s;
  public Reader in;
  public PrintStream out;
  public String url;
  public Request (Socket s) { this.s = s; }
}
```

```
/* =============== Component implementations =============== */

public class BasicLogger implements Logger {
  public void log (String msg) { System.out.println(msg); }
}

public class SequentialScheduler implements Scheduler {
  public synchronized void schedule (Runnable task) { task.run(); }
}

public class MultiThreadScheduler implements Scheduler {
  public void schedule (Runnable task) { new Thread(task).start(); }
}

public class FileRequestHandler implements RequestHandler {
  public void handleRequest (Request r) throws IOException {
    File f = new File(r.url);
    if (f.exists() && !f.isDirectory()) {
      InputStream is = new FileInputStream(f);
      byte[] data = new byte[is.available()];
      is.read(data);
      is.close();
      r.out.print("HTTP/1.0 200 OK\n\n");
      r.out.write(data);
    } else { throw new IOException("File not found"); }
  }
}

public class ErrorRequestHandler implements RequestHandler {
  public void handleRequest (Request r) throws IOException {
    r.out.print("HTTP/1.0 404 Not Found\n\n");
    r.out.print("<html>Document not found.</html>");
  }
}

public class RequestDispatcher implements RequestHandler, BindingController {
  private Map handlers = new TreeMap();
  // configuration concern
  public String[] listFc () {
    return (String[])handlers.keySet().toArray(new String[handlers.size()]);
  }
  public Object lookupFc (String itfName) {
    if (itfName.startsWith("h")) { return handlers.get(itfName); }
    else return null;
  }
```

```
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.startsWith("h")) { handlers.put(itfName, itfValue); }
  }
  public void unbindFc (String itfName) {
    if (itfName.startsWith("h")) { handlers.remove(itfName); }
  }
  // functional concern
  public void handleRequest (Request r) throws IOException {
    Iterator i = handlers.values().iterator();
    while (i.hasNext()) {
      try {
        ((RequestHandler)i.next()).handleRequest(r);
        return;
      } catch (IOException _) { }
    }
  }
}

public class RequestAnalyzer implements RequestHandler, BindingController {
  private Logger l;
  private RequestHandler rh;
  // configuration concern
  public String[] listFc () { return new String[] { "l", "rh" }; }
  public Object lookupFc (String itfName) {
    if (itfName.equals("l")) { return l; }
    else if (itfName.equals("rh")) { return rh; }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.equals("l")) { l = (Logger)itfValue; }
    else if (itfName.equals("rh")) { rh = (RequestHandler)itfValue; }
  }
  public void unbindFc (String itfName) {
    if (itfName.equals("l")) { l = null; }
    else if (itfName.equals("rh")) { rh = null; }
  }
  // functional concern
  public void handleRequest (Request r) throws IOException {
    r.in = new InputStreamReader(r.s.getInputStream());
    r.out = new PrintStream(r.s.getOutputStream());
    String rq = new LineNumberReader(r.in).readLine();
    l.log(rq);
    if (rq.startsWith("GET ")) {
      r.url = rq.substring(5, rq.indexOf(' ', 4));
      rh.handleRequest(r);
    }
```

```
    r.out.close();
    r.s.close();
  }
}

public class RequestReceiver implements Runnable, BindingController {
  private Scheduler s;
  private RequestHandler rh;
  // configuration concern
  public String[] listFc () { return new String[] { "s", "rh" }; }
  public Object lookupFc (String itfName) {
    if (itfName.equals("s")) { return s; }
    else if (itfName.equals("rh")) { return rh; }
    else return null;
  }
  public void bindFc (String itfName, Object itfValue) {
    if (itfName.equals("s")) { s = (Scheduler)itfValue; }
    else if (itfName.equals("rh")) { rh = (RequestHandler)itfValue; }
  }
  public void unbindFc (String itfName) {
    if (itfName.equals("s")) { s = null; }
    else if (itfName.equals("rh")) { rh = null; }
  }
  // functional concern
  public void run () {
    try {
      ServerSocket ss = new ServerSocket(8080);
      while (true) {
        final Socket socket = ss.accept();
        s.schedule(new Runnable () {
          public void run () {
            try { rh.handleRequest(new Request(socket)); }
            catch (IOException _) { }
          }
        });
      }
    } catch (IOException e) { e.printStackTrace(); }
  }
}
```

Voici la définition complète d'architecture de l'application Comanche

```
<!- ============== Component types ============== ->

<definition name="comanche.RunnableType">
  <interface name="r" signature="java.lang.Runnable" role="server"/></provides>
```

```
</definition>

<definition name="comanche.FrontendType" extends="comanche.RunnableType">
  <interface name="rh" signature="comanche.RequestHandler" role="client"/>
</definition>

<definition name="comanche.ReceiverType" extends="comanche.FrontendType">
  <interface name="s" signature="comanche.Scheduler" role="client"/>
</definition>

<definition name="comanche.SchedulerType">
  <interface name="s" signature="comanche.Scheduler" role="server"/>
</definition>

<definition name="comanche.HandlerType">
  <interface name="rh" signature="comanche.RequestHandler" role="server"/>
</definition>

<definition name="comanche.AnalyzerType">
  <interface name="a" signature="comanche.RequestHandler" role="server"/>
  <interface name="rh" signature="comanche.RequestHandler" role="client"/>
  <interface name="l" signature="comanche.Logger" role="client"/>
</definition>

<definition name="comanche.LoggerType">
  <interface name="l" signature="comanche.Logger" role="server"/>
</definition>

<definition name="comanche.DispatcherType" extends="comanche.HandlerType">
  <interface name="h" signature="comanche.RequestHandler" role="client"
                cardinality="collection"/>
</definition>

<!- ============== Primitive components ============== ->

<definition name="comanche.Receiver" extends="comanche.ReceiverType">
  <content class="comanche.RequestReceiver"/>
</definition>

<definition name="comanche.SequentialScheduler"
              extends="comanche.SchedulerType">
  <content class="comanche.SequentialScheduler"/>
</definition>

<definition name="comanche.MultiThreadScheduler"
              extends="comanche.SchedulerType">
```

```
  <content class="comanche.MultiThreadScheduler"/>
</definition>

<definition name="comanche.Analyzer" extends="comanche.AnalyzerType">
  <content class="comanche.RequestAnalyzer"/>
</definition>

<definition name="comanche.Logger" extends="comanche.LoggerType">
  <content class="comanche.BasicLogger"/>
</definition>

<definition name="comanche.Dispatcher" extends="comanche.DispatcherType">
  <content class="comanche.RequestDispatcher"/>
</definition>

<definition name="comanche.FileHandler" extends="comanche.HandlerType">
  <content class="comanche.FileRequestHandler"/>
</definition>

<definition name="comanche.ErrorHandler" extends="comanche.HandlerType">
  <content class="comanche.ErrorRequestHandler"/>
</definition>

<!- ============== Composite components ============== ->

<definition name="comanche.Handler" extends="comanche.HandlerType">
  <component name="rd" definition="comanche.Dispatcher"/>
  <component name="frh" definition="comanche.FileHandler"/>
  <component name="erh" definition="comanche.ErrorHandler"/>
  <binding client="this.rh" server="rd.rh"/>
  <binding client="rd.h0" server="frh.rh"/>
  <binding client="rd.h1" server="erh.rh"/>
</definition>

<definition name="comanche.Backend" extends="comanche.HandlerType">
  <component name="ra" definition="comanche.Analyzer"/>
  <component name="rh" definition="comanche.Handler"/>
  <component name="l" definition="comanche.Logger"/>
  <binding client="this.rh" server="ra.a"/>
  <binding client="ra.rh" server="rh.rh"/>
  <binding client="ra.l" server="l.l"/>
</definition>

<definition name="comanche.Frontend" extends="comanche.FrontendType">
  <component name="rr" definition="comanche.Receiver"/>
  <component name="s" definition="comanche.MultiThreadScheduler"/>
```

```
  <binding client="this.r" server="rr.r"/>
  <binding client="rr.s" server="s.s"/>
  <binding client="rr.rh" server="this.rh"/>
</definition>

<definition name="comanche.Comanche" extends="comanche.RunnableType">
  <component name="fe" definition="comanche.Frontend"/>
  <component name="be" definition="comanche.Backend"/>
  <binding client="this.r" server="fe.r"/>
  <binding client="fe.rh" server="be.rh"/>
</definition>
```

A.4 Code d'implémentation de l'application Système de contrôle avionique

Voici une partie du code source de l'exemple de ce système, à savoir le code d'implémentation pour les composants RateGen and GPS.

```
typedef unsigned long rateHz;
interface rate_control
{ void start();
  void stop();   };
component RateGen supports rate_ control
{ publishes tick Pulse;
  emits tick trigger ;
  attribute rateHz Rate;   };
interface RateGen :Components:: CCMObject,rate_ control
{ Components:: Cookie subscribe_Pulse(in tickConsumer c);
  tickConsumer unsubscribe_Pulse(in Components:: Cookie ck);
  attribute rateHz Rate;   };
class RateGen_ Executor_Impl : public virtual CCM_ RateGen,
public virtual CORBA:: LocalObject
{ public:
   virtual void send_pulse(void)
    { tick_var ev = new tick();
      this->context_->push_Pulse(); }
  };

component GPS
{ provides position MyLocation;
  consumes tick Refresh;
  publishes tick Ready;   };
interface GPS :Components:: CCMObject
{ position provide_MyLocation();
  tick_consumer get_consumer_refresh();
```

```
  components::Cookie subscribe_Ready(in tickconsumer c);
  tickconsumer unsubscribe_Ready(in Components::Cookie ck);   };
class GPS_ Executor_Impl : public virtual GPS_Executor,
public virtual CORBA:: LocalObject
{ private:
   CCM_ GPS_Context_var context_;
  public:
   virtual CCM_position_ptr get_MyLocation() { return this; }
   virtual CORBA:: Long get_pos() { return cached_current_location; }
   void set_session_context(Components:: SessionContext_ptr c)
    { this->context_ =CCM_GPS_Context:: narrow(c); };
   };
```

A.5 Exemple de description XML pour l'automatisation de l'analyse d'une composition mixte de SWNs

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Composition SYSTEM "composition.dtd">

<!-- @version: 1.0 -->
<Composition>
  <File>interface_n1.xml</File>
  <File>interface_n2.xml</File>
  <File>interface_n3.xml</File>
  <Interconnection_File>interconnections.xml</Interconnection_File>
</Composition>
**********************************************************************
// Fichier interface_n1.xml
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Interfaces SYSTEM "interface.dtd">

<!-- @version: -->
<Interfaces>
  <Module>n0</Module>
  <Synchronous_interface id="1">
      <Input_transitions>
<transition>TSI</transition>
      </Input_transitions>
      <Output_transitions>
<transition>TSO</transition>
      </Output_transitions>
  </Synchronous_interface>
</Interfaces>
**********************************************************************
```

```
// Fichier interface_n2.xml
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Interfaces SYSTEM "interface.dtd">

<!-- @version: 1.0 -->
<Interfaces>
  <Module>n1</Module>
  <Synchronous_interface id="1">
      <Input_transitions>
<transition>TSI</transition>
      </Input_transitions>
      <Internal_transitions>
        <transition>TSi1</transition>
      </Internal_transitions>
      <Output_transitions>
<transition>TSO</transition>
      </Output_transitions>
  </Synchronous_interface>
  <Asynchronous_interface>
      <Input_places>
<place>P12</place>
      </Input_places>
      <Output_interface>
<output transition="TAO">
<arc id="1" expression="x"></arc>
</output>
      </Output_interface>
  </Asynchronous_interface>
</Interfaces>
**********************************************************************
// Fichier interface_n3.xml
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Interfaces SYSTEM "interface.dtd">

<!-- @version: 1.0 -->
<Interfaces>
  <Module>n2</Module>
  <Asynchronous_interface>
      <Input_places>
<place>P8</place>
      </Input_places>
      <Output_interface>
<output transition="TAI">
<arc id="1" expression="x"></arc>
</output>
      </Output_interface>
```

```
  </Asynchronous_interface>
</Interfaces>
**********************************************************************
// Fichier interconnections.xml
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Interconnections SYSTEM "interconnections.dtd">

<!-- @version: 1.0 -->

<Interconnections>
  <Synchronous_connection>
   <Module1 name="n0" synchronous_interface_id="1"></Module1>
<Module2 name="n1" synchronous_interface_id="1"></Module2>
   <Input_associations>
<associate transition_module1="TSI" transition_module2="TSI"></associate>
</Input_associations>
<Output_associations>
<associate transition_module1="TSO" transition_module2="TSO"></associate>
</Output_associations>
  </Synchronous_connection>
  <Asynchronous_connection>
   <List_modules>
<module name="n1"></module>
<module name="n2"></module>
</List_modules>
<Connections>
<connect name_net1="n1" name_net2="n2">
<item place_module1="P12" transition_module2="TAI" id_arc="1"></item>
</connect>
<connect name_net1="n2" name_net2="n1">
<item place_module1="P8" transition_module2="TAO" id_arc="1"></item>
</connect>
</Connections>
  </Asynchronous_connection>
</Interconnections>
```

Analyse de performances des systèmes basés composants

Nabila SALMI

Résumé

L'industrie du logiciel et du matériel s'oriente de plus en plus vers la conception de systèmes sous la forme d'assemblage de composants. L'objectif de ce type de conception est de réduire le coût et le temps de développement par réutilisation des composants, et d'atteindre un haut degré de maintenabilité, d'extensibilité et de dynamicité. La vérification de la correction d'un tel système reste importante, tant du point de vue qualitatif que quantitatif. Dans cette optique, nous développons, dans cette thèse, une méthode d'analyse qualitative et quantitative (performances) d'un système construit par assemblage de composants (CBS), concentrée sur les performances. L'intérêt de la méthode réside dans le fait de tirer parti de l'architecture compositionnelle de ces systèmes pour réduire la complexité d'analyse en termes de temps de calcul et d'occupation mémoire, et de permettre ainsi d'analyser des systèmes à espace d'états important. Nous partons de l'architecture à composants et nous modélisons systèmatiquement et adéquatement un CBS pour appliquer après une méthode structurée pour l'analyse des performances du système global. Les composants sont modélisés en utilisant un modéle de haut niveau, les Réseaux de Petri Stochastiques bien formés (Stochastic Well-formed Net), largement utilisés pour l'évaluation de performances des systèmes complexes partiellement ou totalement symétriques. Deux types majeurs d'interaction entre composants sont considérés : la communication par invocation de service et la communication basée événements. Pour réduire la complexité d'analyse, l'analyse structuée d'un CBS est fondée sur une description tensorielle du générateur de la chaîne de Markov agrégée sous-jacente. Des études de cas illustrent notre approche.

Mots-clés : Performances, Systèmes basés composants, SWN, Composant, Interconnexion, invocation de service, communication par événement, méthode tensorielle

Abstract

Software and hardware industry is being more and more oriented to component based design of systems. The aim of such design is to reduce development cost and time by reuse of components, and to achieve high degree of maintainability, extensibility and dynamics. Ensuring correctness and analysis of these component based systems (CBS) seems to be an important matter, even qualitative or quantitative analysis. In this optic, we develop, in this thesis, a new method allowing to perform a qualitative and quantitative analysis of a CBS. The main benefit of our method is to exploit the compositional architecture of such systems in order to reduce the complexity of analysis (computation time and memory savings), and to allow thus the analysis of important state spaces. This approach starts from the definition of the component architecture of a CBS, and models systematically and adequately this CBS in order to apply a structured method for performance analysis of the global system. The components are modeled with a high level model, the Stochastic Well-formed Net, widely used for performance evaluation of complex systems partially or totaly symmetrical. Two main kinds of interaction between components are considered : communication by service invocation and event based communication. In order to reduce analysis complexity, the structured analysis of a CBS is based on a tensorial descrition of the generato of the underlying Markov chain. Case studies illustrate our approach.

Keywords: Performances, Component-Based Systems, SWN, Component, Interconnection, service invocation, event based communication, tensorial method