REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université El Hadj Lakhdar – BATNA

Faculté des Sciences et des Sciences de l'ingénieur

Département d'informatique

N° d'ordre :……
Série :……

Mémoire
Présenté en vue de l'obtention du diplôme

Magister en Informatique

Option: **Informatique Industrielle**

SUJET DU MÉMOIRE :

Une approche de modélisation des logiciels à base de composants par les réseaux de Petri

Présenté le : 14 / 06 / 2009

Par : **ZERNADJI Tarek**

Composition du jury:

Mr. BELATTAR Brahim	Président	(Maître de Conférence à l'Université de Batna)
Mr. CHAOUI Allaoua	Rapporteur	(Maître de Conférence à l'Université de Constantine)
Mr. BILAMI Azzeddine	Examinateur	(Maître de Conférence à l'Université de Batna).
Mr. KAZAR Okba	Examinateur	(Maître de Conférence à l'Université de Biskra).

Remerciements

J'aimerais d'abord remercier mon encadreur qui m'a soutenu tout au long de la rédaction de ce mémoire, pour son écoute et ses conseils pertinents.

Je remercie également l'ensemble des membres du jury pour avoir consacrer leur temps à examiner ce travail malgré leurs nombreuses responsabilités, je leur suis reconnaissant pour l'attention qu'ils ont portée à mon travail.

Je tiens aussi à remercier ma famille et mes amis pour leur soutient inconditionnel et leur présence continue.

Enfin je remercie tous ceux qui m'ont aidé de près ou de loin pour la réalisation de ce travail.

SOMMAIRE

LISTE DES FIGURES

Introduction générale

Nous sommes témoin d'une énorme expansion dans l'utilisation du logiciel dans les entreprises, l'industrie, l'administration, la recherche et même dans la vie quotidienne. Le logiciel n'est plus utilisé marginalement dans les systèmes techniques ; au lieu de cela il est devenu un facteur central dans beaucoup de domaines. Les systèmes basés sur des fonctionnalités logicielles deviennent le facteur le plus important dans un marché compétitif. Cette tendance, a augmenté les exigences sur les produits logiciels tels qu'une rentabilité meilleure, la robustesse, la fiabilité, la flexibilité, l'adaptabilité, et une installation et un déploiement plus simple de ces derniers. Au moment ou ces demandes augmente fortement, la complexité des processus que le logiciel doit contrôler augmente avec la demande pour l'intégration des processus de différents secteurs. Par conséquent, les programmes logiciels deviennent de plus en plus grands et complexe. Le défi principal pour les développeurs de logiciels aujourd'hui est de faire face à la complexité et de s'adapter rapidement au changement.

Les approches classiques de développement de logiciel ont adressé les défis de la complexité croissantes et de la dépendance des logiciels externes en se focalisant sur un système à la fois, et sur la satisfaction des contraintes temporelles et financières de livraison du produit sans considérer les besoins évolutionnistes du système [1]. Se concentrer sur un système à la fois et négliger les changements qui peuvent se produire a mené à un certain nombre de problèmes: l'échec de la majorité des projets a satisfaire la contrainte de date limite (deadline), de budget, de qualité aussi bien que l'augmentation continue des coûts associée à la maintenance du logiciel.

Une solution clé à ces problèmes est le Développement Basé Composant DBC[1]. Cette approche émergente du domaine de l'ingénierie logicielle est basée sur la '*réutilisation'*, où les systèmes logiciels sont construits en assemblant des composants déjà développés et préparés pour l'intégration. Ainsi, elle possède l'avantage de fournir des systèmes logiciels de qualité meilleure en un temps plus court. Construire une application est donc considérée, non pas comme un développement intégral et complet (à partir de zéro), mais comme un assemblage de pièces réutilisables. Dans ce domaine, les technologies qui supporte la construction et l'assemblage de composants ont atteint un premier niveau de maturité, en particulier avec l'apparition des technologies de composants standards telles

[1] Nous utiliserons dans le reste de ce mémoire l'acronyme anglais CBD (Component-Based Development) à la place de l'équivalent français DBC.

que EJB (Enterprise JavaBeans), CCM (CORBA Component Model), ou (D) COM ((Distributed) Component Object Model).

Le cœur du CBD est le composant. Un composant logiciel est considéré comme une boite noire et est décrit par ces 'interfaces' à travers lesquelles il peut interagir avec les autres composants dans le système. Il offre des services et requiert d'autres pour accomplir ces fonctionnalités. Le but ultime du développement logiciel basé composant est l'assemblage effectué par les tierces parties. Pour ce faire, il est nécessaire d'être en mesure de spécifier les composants de façon à ce que nous pouvons raisonner sur leurs constructions et leurs compositions [46].

Les spécifications de composants utilisés dans la pratique du développement logiciels, sont aujourd'hui essentiellement limitées à ce que nous appelions les spécifications syntaxiques. Cette forme de spécification comprend les spécifications utilisées dans des technologies comme COM, CORBA, et JavaBeans [1]. Les deux premières utilisent les IDLs (Interface Description Langage), alors que le troisième utilise le langage de programmation Java. Or, les spécifications syntaxiques n'offre pas suffisamment d'informations sur la sémantique des opérations qui définissent les interfaces d'un composant, ce qui peut aider à mieux les inspectés, les évalués, et comprendre leurs comportements.

Les méthodes de spécification formelles sont de bons candidats pour la spécification des composants logiciels. On entend par méthodes formelles, les langages, techniques, et les outils basés sur la logique mathématique [47]. En effet la base des méthodes formelles est de nous permettre de construire des modèles mathématiques avec une sémantique bien définie du système à analyser. Les réseaux de Petri sont considérés comme des formalismes de spécification dotés d'une définition formelle permettant de construire des modèles exempts d'ambiguïté. Ils possèdent un grand pouvoir d'expression facilitant la description des systèmes complexes, concurrent et distribués.

Les ECATNets [48] (Extended Concurrent Algebraic Term Nets) constituent une catégorie des réseaux de pétri algébriques de hauts niveaux. Ils ont été proposés comme étant une manière pour la spécification, la modélisation, et la validation des applications du domaine des réseaux de communication, la conception des ordinateurs, et d'autres systèmes complexes. Ils sont construits autour d'une combinaison de trois formalismes. Les deux premiers formalismes constituent un modèle réseau/données, et sont utilisés pour la définition de la syntaxe du système, en d'autres termes pour capturer sa structure. Le modèle du réseau, qui est un type de réseaux de Petri avancé, est utilisé pour décrire l'architecture du processus du système, le modèle de données, qui est un formalisme algébrique est utilisé pour spécifier les structures de données du système. Le troisième formalisme, qui est une logique de réécriture est utilisé pour définir la sémantique du système, en d'autres termes, pour décrire son comportement.

L'objectif de l'ingénierie logicielle basée composant (CBSE) est d'accroître la productivité, la qualité, et le temps de mise sur le marché dans le développement logiciels. Les techniques d'ingénierie logicielles basée composant dans les phases, de modélisation, de vérification, ou de validation des systèmes logiciels à base de composants restent insuffisantes et demande plus d'efforts de recherche.

Notre travail s'inscrit dans le vaste domaine du génie logiciel à base de composants. Son objectif, est de proposer une approche formelle pour la modélisation des logiciels à base de composant en utilisant les ECATNets. Notre choix sur les ECATNets est motivé d'une part, par la puissance d'expression de ces derniers dus à leurs notations syntaxiques très riches basées sur un formalisme algébrique, permettant de fournir des modèles hautement compacts. D'autre part, ils possèdent une sémantique saine définit en terme de la logique de réécriture permettant d'exprimer remarquablement et intuitivement le comportement dynamique et le parallélisme qui caractérise les systèmes distribués. Elle donne aussi la possibilité d'appliquer la vérification formelle par le biais de spécification exécutable exprimée en langage Maude, le langage de la logique de réécriture.

Dans notre approche nous distinguons trois étapes ; la première consiste en la spécification des interfaces de chaque composant dans le système à base de composants ainsi que, les connexions entre eux en utilisant les notations proposées. Le résultat de cette étape est un modèle ECATNet représentant la spécification d'un assemblage de composants. Dans la deuxième étape, le comportement dynamique du système est exprimé par des règles de réécriture générées à partir du modèle spécifié dans l'étape précédente. La dernière étape représente la vérification du système via une spécification exécutable en langage Maude qui intègre les règles de réécriture engendrées dans l'étape précédente. A travers ce travail nous espérons atteindre les objectifs suivants :

- ✓ Premièrement, offrir une notation convenable pour décrire le comportement interne des composants logiciels concurrents et distribués.
- ✓ Deuxièmement, offrir une sémantique formelle non ambiguë pour les caractéristiques des composants logiciels.
- ✓ Troisièmement, offrir un raisonnement formel sur l'assemblage des composants logiciels.
- ✓ Quatrièmement, permettre la vérification formelle du système modélisé.

Ce mémoire est structuré suivant trois chapitres :

- Dans le premier chapitre nous allons introduire les approches classiques du développement logiciel, ensuite nous présenterons l'approche de développement logiciel à base de composants et les notions qui lui sont reliées tel la notion de réutilisation, composant logiciel,…etc. Nous parlerons éventuellement des racines de cette approche et de quelques modèles de cycles de vie qui ont été proposés dans son contexte.

- Le deuxième est destiné à la présentation des ECATNets et la logique de réécriture. Nous commençons par un bref survol sur les RdPs, puis nous introduisons la logique de réécriture et son langage Maude comme étant un Framework qui intègre les ECATNets. Ensuite nous détaillons ces derniers et nous montrerons comment peuvent-ils être utilisés pour la spécification des systèmes.
- Le dernier chapitre est consacré à la présentation de notre approche pour la modélisation des logiciels à base de composants.
- Le présent mémoire est achevé par une conclusion générale et des perspectives.

CHAPITRE I :

Développement logiciel à base de composants

1. Introduction

Les systèmes logiciels modernes sont de plus en plus grands, complexes et mal contrôlés, ce qui a entraîné des coûts de développement élevés, une faible productivité, et une qualité incontrôlable des produits logiciels [8]. Par conséquent, il y a une demande croissante pour la recherche d'un nouveau paradigme du développement logiciel, efficace et rentable.

Une solution clé à ces problèmes est le Développement Basé Composant CBD. Cette approche émergente du domaine de l'ingénierie logicielle est basée sur la '*réutilisation'*, où les systèmes logiciels sont construits en assemblant des composants déjà développés et préparés pour l'intégration. Cette approche va être le sujet de ce chapitre dans lequel nous allons présenté plus de détails, mais avant cela ; il est intéressant de ce positionner dans l'échelle de la chronologie des choses, et jeter un coup d'œil sur les approches classiques de développement logiciel par un historique non détaillé sur ces derniers qui montre leur évolution dans le temps.

2. Processus logiciel

Tout logiciel, particulièrement les grands logiciels produits par beaucoup de gens, devrait être produit en utilisant un certain type de méthodologie. Une méthodologie est une manière systématique de faire les choses. C'est un processus répétitif que nous pouvons suivre à partir des premiers pas du développement logiciel jusqu'à la maintenance d'un système opérationnel [2]. Un système logiciel peut être considéré d'un point de vue cycle de vie. Ceci signifie qu'on observe le système depuis la première notion de son existence jusqu'à son fonctionnement et sa gestion [1]. Plusieurs différentes approches ont été proposées pour le développement du cycle de vie d'un logiciel, ces dernières sont toutes basées sur les mêmes activités et se différent seulement par la manière dont elles sont réalisées.

2.1 Activités génériques du cycle de vie d'un logiciel

C'est sans doute Royce (1970) qui, était le premier, a proposé un modèle du cycle de vie. Depuis, de nombreuses améliorations et modifications y ont été apportées [49]. Il y a un certain nombre de phases communes à chaque développement, indépendamment de la méthodologie, en commençant par la capture des besoins et finir avec la maintenance. Nous allons essayer dans ce qui suit de présenter une brève description des différentes phases constituant le cycle de vie d'un système logiciel d'après [1].

a. Analyse de besoins et spécification du système

Dans cette phase les services, les contraintes, et les objectifs du système sont établis. Ils sont définis en détail et servent de spécification au système.

b. Conception du système et du logiciel

La conception du logiciel implique d'identifier et de décrire les abstractions fondamentales du système logiciel et les relations entre elles. Une architecture globale pour le système est définie suivie d'une conception détaillée.

c. Implémentation et tests unitaires

La formalisation de la conception sous une forme exécutable. Elle peut être composée de petites unités. Les unités sont vérifiées vis-à-vis de leurs spécifications.

d. Intégration, vérification du système et validation

Les unités du système sont intégrées; le système complet est vérifié, validé, et délivré au client.

e. Opération de support et maintenance

Le système est opérationnel. Il requiert un support continu et des opérations de maintenance.

f. Suppression du système

Cette activité est souvent oubliée dans le cycle de vie d'un système. Elle concerne la suppression propre du système et éventuellement son remplacement par un autre.

2.2 Modèles de cycle de vie d'un logiciel

Il faut un temps considérable pour développer un grand système logiciel qui, d'ailleurs, sera utilisé pendant très longtemps. On identifie donc un certain nombre d'étapes distinctes dans cette période de développement et d'utilisation ; ces étapes constituent le cycle de vie du logiciel. Plusieurs différentes approches pour le cycle de vie de développement d'un logiciel sont considérées ci-dessous ainsi qu'une brève description de leurs caractéristiques de base. Dans la littérature en trouve plusieurs classification de ces approches les séparant en deux classes : modèles linéaire et modèles non linéaire, ou bien modèles séquentiel et modèles évolutionnistes. Nous avons adopté la deuxième appellation qui est la plus utilisée.

2.2.1 Modèles séquentiels

Le modèle séquentiel (par exemple, le modèle en cascade de Royce 1970) suit une approche séquentielle systématique qui commence au niveau du système et progresse successivement de l'analyse jusqu'à la réalisation. Chaque activité doit être considérée comme approuvée avant que la prochaine activité commence. La sortie d'une activité est l'entrée de la prochaine. Cette approche pour le développement de logiciel (figure 1) est la plus vieille et la plus répandue.

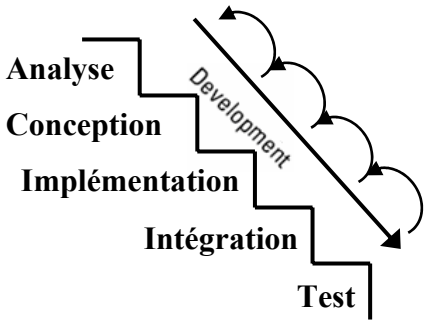


Fig. 1 Le modèle séquentiel [1]

Beaucoup de problèmes majeurs dans l'ingénierie logicielle résultent du fait qu'il est difficile à l'utilisateur d'exprimer explicitement tous les besoins. Le modèle séquentiel exige la spécification complète des besoins ce qui peut causer des difficultés d'adaptation face à l'incertitude naturelle qui existe au début de beaucoup de projets. Il est également difficile d'ajouter ou changer des besoins durant le développement car, une fois réalisées, les activités sont considérées comme accomplies. Un autre problème parfois rencontré quand un modèle séquentiel est appliqué est la réponse en retard au client. Cette approche rigide souffrira toujours du fait d'être concentré à résoudre un problème particulier, ce qui rend la tache de produire du code réutilisable plus difficile et par conséquent mal adaptée au monde composant [3].

2.2.2 Modèles évolutionnistes

Les modèles évolutionniste permettent de développer un système graduellement par itérations successives. A chaque étape, le résultat du système est montré à ses commanditaires et leurs remarques sont prises en compte pour faire évoluer le système [3]. L'intérêt de ce type de modèle est de réduire fortement le risque de découvrir un problème crucial tard dans le cycle de développement. Il permet également une meilleure gestion des changements de besoins apparaissant durant le processus de développement. Le problème de ce type de processus est qu'il peut engendrer un nombre coûteux d'itérations. Il peut, par exemple, être difficile de déterminer le nombre exact d'itérations à l'avance, et de nouvelles itérations peuvent être ajoutées si les besoins changent ou évoluent. Il existe plusieurs types de modèles évolutionnistes. L'approche itérative, le modèle incrémental, le modèle par prototypage, et enfin, le modèle à spirale défini par Boehm [4].

2.2.3 Processus unifié

Le processus de développement logiciel unifié (UP[2]) définit par Jacobson et al [5] est un processus de développement de système optimisé pour les applications objets et les systèmes à base de composants. Il s'agit d'un développement incrémental itératif composé de quatre phases :

1. Analyse des besoins, c'est la phase où le système est décrit d'une manière formalisée, fournissant ainsi une base pour le développement ultérieur;
2. Elaboration, lors de cette phase l'architecture du système est définie et créée;
3. Construction, le développement des produits complètement nouveaux, avec des possibilités de *réutilisation*;
4. Transition, concerne la livraison du système au client. Cette phase inclut l'installation du système et la formation de ses utilisateurs.

Pour chacune de ces étapes, il y a plusieurs itérations des activités classiques. Ces activités se produisent dans chacune des quatre phases, Analyse des besoins, élaboration, construction, et transition. C'est la partie itérative de l'UP. La partie incrémental est basée sur le fait que les itérations réussies auront comme conséquence le dégagement d'un système. Ce modèle est mieux adapté au monde composant. De plus il est souvent utilisé avec UML, dont la version 2.0 prend en compte la conception des applications à base de composant.

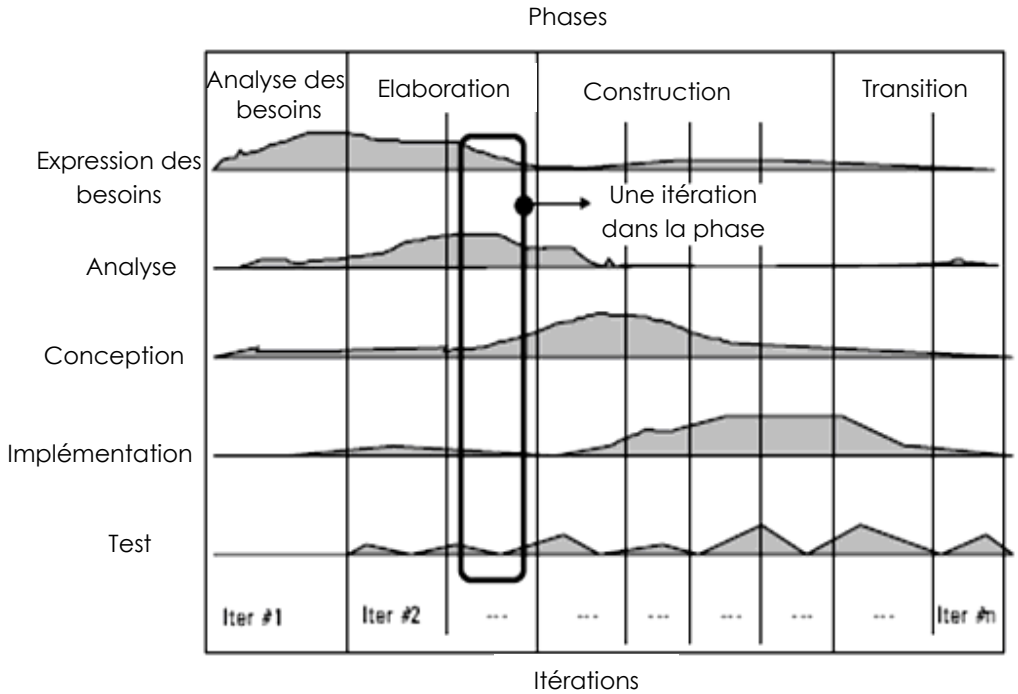


Fig. 2 Les quatre phases de l'UP et ces activités [1]

[2] Unified Process

3. Approche de réutilisation

Les systèmes logiciels modernes deviennent de plus en plus grands, complexe et difficile à contrôler, résultant en un coût de développement élevé, une productivité faible, une qualité incontrôlable du logiciel et le gros risque pour se déplacer vers une nouvelle technologie [8]. Par conséquence, il y a une demande croissante sur la recherche d'un nouveau paradigme efficace, et rentable pour le développement de logiciel.

Une des solutions les plus prometteuses est aujourd'hui l'approche de développement de logiciel basé composant (CBD). Cette approche est basée sur l'idée que les systèmes logiciels sont construits en assemblant des composants déjà développés et préparés pour l'intégration avec une architecture logiciel bien définit [1]. Cette nouvelle approche de développement de logiciel est différente des approches traditionnelles dans lesquelles les systèmes logiciels sont construits à partir de zéro.

3.1 Développement basé composant CBD

Les praticiens définissent le développement basé composant (CBD) comme approche dans laquelle tout Artefact (code exécutable, spécification d'interface, architectures, modèles, et partant du plus haut niveau des applications et des systèmes complets jusqu'au niveau des composants individuels) peut être construit en assemblant, adaptant, et en liant les composants existant ensemble dans une variété de configurations [12].

Cette définition met en relief un aspect important du développement basé composant (CBD), à savoir la capacité de construire des systèmes par assemblages de composants préexistants. D'autre part, afin de pouvoir assembler des composants, ces composants doivent d'abord exister. Plusieurs organismes prétendent qu'ils construisent des logiciels en utilisant une approche basée composant, et que l'industrie entière est apparemment dans une frénésie d'adoption du CBD. Une autre définition qui a été donnée au CBD est la suivante :

Le développement basé composant est une approche de développement logiciel où tous les aspects et phases du cycle de vie y compris l'analyse de besoins, la conception, la réalisation, test, déploiement, les infrastructures techniques supportés, et également la gestion de projet, sont basés sur les composants [12]. Cette définition montre explicitement que le développement basé composant consiste en la construction de logiciel en utilisant un raisonnement basé composant dans la mesure où tout le développement logiciel sera centré sur les composants. CBD est extrêmement bénéficial non seulement quand les composants sont disponibles pour l'assemblage, mais aussi si ces composants doivent être construits en étant des éléments du projet. En effet, penser à un système d'information en termes de composants, même si ces composants n'existent pas, est la meilleure manière de maîtriser les complexités de développement de grand système distribué aujourd'hui.

3.2 Racines du CBD

Construire des systèmes complexes à partir de composants plus simples n'est pas une invention de l'industrie logicielle. Il y a plus de deux siècles, Eli Whitney a proposé la fabrication de fusils avec des pièces remplaçables spécifiés clairement selon des modèles, au lieu de construire chaque fusil individuellement. Largement considéré comme le commencement de fabrication en série, cette stratégie des composants remplaçables a mené à une augmentation impressionnante de la capacité de production, en outre elle a fournit des avantages additionnels tel que, des opérations fiables et une maintenance facile dans divers secteurs industriels, et elle est devenu une approche standard dans par exemple la construction de divers dispositifs électroniques et dans l'ingénierie des systèmes avancés [10].

Dans le domaine de l'ingénierie des systèmes d'information, le CBD est également considéré comme une approche évolutionniste plutôt que révolutionnaire. Il a existé dans une forme ou une autre pendant un certain nombre d'années. L'idée de construire un système logiciel modulaire a été longtemps reconnue comme avantageuse au sein de la communauté software. Ceci remonte aux jours de la programmation structurée en utilisant des sous-programmes et des bibliothèques servant de 'composants'. Les programmes ont été construits en créant un programme principal pour contrôler (ou appeler) un certain nombre de sous-programmes. Pour réduire les efforts de programmation, les programmeurs dans une équipe de projet réutilisaient des sous-programmes durant la réalisation du projet [11]. Durant la conférence de l'OTAN de l'ingénierie logicielle, en 1968, M. D. Mcilory a présenté le concept de composants logiciels. Ces derniers ont été vus comme solution pour la crise logicielle [10].

Depuis la fin des années 70 jusqu'aux années 80, les ingénieurs utilisaient une méthodologie de développement logiciel structurée pour diviser un système en un certain nombre de modules en se basant sur ses besoins fonctionnelles. Après que les modules aient été développés, ils les intègrent pour former un système. Durant cette période de temps, beaucoup de bibliothèques de fonction ont été développées en langage de programmation Fortran comme étant des packages réutilisables pour supporter le développement d'applications scientifiques. Ces packages peuvent être considérés comme la première génération de composants logiciels [11].

Durant les années 90 les techniques orientées objet ont été considérées comme un moyen puissant pour résoudre le problème de la crise logicielle grâce au niveau élevé de réutilisation et de maintenance atteint par ces derniers. Quelques auteurs ont cité que le CBD est une extension et évolution naturelle du paradigme orienté objet, et qu'il contient beaucoup de caractéristiques et de concepts semblables. D'autres insistent sur l'échec de la technologie objet pour résoudre les problèmes de réutilisation des logiciels et sur la nécessité d'introduire un nouveau paradigme de développement de composant [10]. Le travail réalisé par Nierstrasz (1995) qui est en relation avec la composition logicielle pour les systèmes ouverts distribués, peut être considéré comme une importante contribution

aux futurs concepts et idées du CBD. Au cours des dernières années la recherche liée aux composants a également été menée sur divers sujets tels que les langages d'interconnexion de module (MILs[3]), spécification et analyse des interfaces de module, générateurs de logiciel, les Frameworks orientés objets et les patterns, langages de description d'architecture (ADLs[4]).

Les composants ont été initialement introduits au niveau implémentation pour la construction rapide d'interface utilisateur graphique en utilisant VBX (Visual Basic eXtensions) ou Java. Après, Microsoft ActiveX/DCOM/COM, objets et services de CORBA, et JavaBeans, suivis de Microsoft COM+/.NET, les composants CORBA et Entreprise JavaBeans (EJB) ont été proposés comme des technologies de composants standards.

L'objectif commun de toutes ces initiatives est de basculer depuis le développement de petits systèmes monolithiques centralisés vers le développement de systèmes complexes composés d'unités fonctionnelles déployées sur les noeuds du Web. Les deux concepts principaux qui ont émergé sont [10]:

1. Les composants en tant que entités d'un système de granularité forte.
2. Les architectures et les Frameworks comme modèles de système décrivant ses principaux modules et la manière de les composer en un tout cohérent.

3.3 Processus de développement d'applications basés composants

Un des objectifs du CBD est de généraliser la réutilisation de modules logiciels. L'intérêt majeur de la réutilisation est de réduire l'effort de développement en réutilisant des parties de logiciels déjà développées. Plus un composant est réutilisé, plus les éventuels bogues qu'il pouvait contenir peuvent avoir été détectés et corrigés.

Le processus de développement des applications basées composants repose globalement sur les mêmes activités que le développement d'une application classique. Cependant, il diffère des développements traditionnels notamment à travers sa focalisation sur l'identification d'entités réutilisables et les relations entre-elles et ce, notamment à partir de l'analyse des besoins. La réutilisation de composants implique un processus d'identification et de sélection du composant à réutiliser, voir un processus de spécification de celui-ci en vue de son développement dans le cas où il n'existerait pas. L'aspect modularité implique un effort important de la conception de l'architecture de l'application, ce qui n'était pas forcément le cas dans les développements classiques. Les modèles de développement classiques se prêtent plus ou moins à supporter le développement à base de composants.

[3] Module Interconnection Languages
[4] Architecture Description Languages

Comme on l'a cité plus haut dans ce chapitre, le modèle séquentiel correspond à une étude séquentielle du système, de la phase de recueillement des besoins jusqu'à la fin du cycle de vie. Chaque activité est terminée avant de passer à la suivante. Cette approche rigide est mal adaptée au monde composant. En effet, le développement à base de composants est typiquement un développement où des phases peuvent être menées en parallèle. Par exemple, si la phase d'analyse détermine le besoin spécifique d'un composant particulier, le modèle séquentiel implique de terminer la phase d'analyse avant de spécifier le composant à développer. Or, il aurait peut être était plus prudent de commander le développement de ce composant, à un éventuel fournisseur, le plus tôt possible, de manière à tenir compte de son temps de développement.

Outre le modèle séquentiel, les modèles évolutionnistes aussi sont mal adaptés au monde composant car elles n'ont pas étaient conçues autour de la problématique de la réutilisation. Le processus unifié est mieux adapter au monde composant car il est souvent utilisé avec UML (Unified Modelling Language) dont la version 2.0 prend en compte la conception des applications à base de composant comme déjà mentionné plus haut. Cependant [3], il ne traite réellement les aspects réutilisation que lors de la phase d'implémentation, ce qui est trop tard la plus part du temps, pour une réutilisation efficace. Les modèles de l'ingénierie logicielle classique pouvaient être utilisés comme base pour le développement d'application à base de composants. Ils sont toutefois mal adaptés et notamment sur deux aspects [13]. Premièrement, la réutilisation de composants est généralement envisagée trop tard dans le cycle de vie du logiciel, notamment après que la conception détaillée ait été finalisée, alors qu'une réutilisation optimale des composants requiert également la réutilisation des architectures logicielles. Deuxièmement, aucune méthode existante ne donne de consigne précise sur la manière dont un composant doit être développé pour être réutilisable. Les méthodes classiques devront donc être adaptées pour prendre en compte les particularités de ce type de développement, notamment en insistant sur les aspects développement de composants réutilisables, mais aussi sur les aspects développement de systèmes à partir de composants [14].

Nous allons essayer dans les prochaines sections d'expliquer le processus de développement des applications à base de composants et de présenter quelques modèles de cycle de vie pour le développement basé composant, ces derniers représentent les efforts des chercheurs durant ces dernières années dans l'ingénierie logicielle basée composant CBSE[5]. Mais avant cela nous nous penchons sur quelques aspects important qui caractérisent le processus de développement des applications à base de composants.

3.3.1 Notion de réutilisation

Le développement basé composant CBD, provient de la réutilisation. L'approche de réutilisation n'est pas nouvelle. La réutilisation est la philosophie de trouver des ressources appropriées, telle des modèles, des diagrammes, des fragments de code, ou de composants,

[5] Nous utiliserons dans le reste de ce mémoire l'acronyme anglais CBSE (Component-Based Software Engineering) à la place de l'équivalent français ILBC.

qui peuvent fournir une solution de départ. La réutilisation est une philosophie de développement de systèmes qui peut être adoptée dans une organisation si elle veut atteindre son potentiel en termes d'une réduction du temps de développement du projet et d'une minimisation des coûts [16].

La réutilisation fut évoquée en 1968 par M. McIlroy dans une conférence de l'OTAN sur l'ingénierie du logiciel, suite a un échec patent de la part de développeurs pour livrer des logiciels de qualité a temps et a prix compétitifs. Elle vise donc trois objectifs : diminuer les coûts de développement et de maintenance, réduire le temps de mise sur le marché (time-to-market), et améliorer la qualité du logiciel. La réutilisation se définit comme une nouvelle approche de développement de systèmes, qui propose notamment de construire un système nouveau à partir de composants logiciels sur étagère (COTS, Commercial Off The Shelf) qui se définissent comme étant un type particulier de composant. Elle s'oppose ainsi aux approches traditionnelles de développement, dans lesquelles la construction d'un nouveau système part de rien (from scratch) et nécessite de réinventer de grandes parties d'applications a chaque fois.

Depuis, cette idée s'est imposée et la réutilisation constitue un enjeu majeur de recherche en ingénierie des systèmes à base de composants. Les recherches sur la réutilisation ont progressivement évolué ces dernières années, passant de la réutilisation de code à la réutilisation de connaissances et de démarches de raisonnement de différents niveaux. Cette pratique pressente un enjeu supplémentaire pour garantir une meilleure qualité des développements [15].

En parlant de composants logiciels dont nous nous somme intéressé dans ce mémoire et qui constituent l'un des types de composants parmi ceux classifiés dans la littérature (nous avons pris le choix d'attarder leurs présentation dans ce chapitre pour des raisons purement pédagogiques), il est utile de distinguer entre la réutilisation 'boîte blanche' dans laquelle l'implémentation des composants réutilisés est exposée à un certain degré, et la réutilisation 'boîte noire', dans laquelle les composants ne peuvent être réutilisés que selon une interface de réutilisation spécialement fournit, ou contrat[6] [13].

- Une réutilisation de type 'boîte noire' consiste à réutiliser le composant tel quel sans aucune vision de la manière dont il est implémenté et sans aucune modification de code de la part du développeur de l'application.
- La réutilisation par approche boîte blanche, par opposition à l'approche précédente, consiste à réutiliser un composant en ayant accès à son implémentation et en permettant sa modification (adaptation). Ce type fait l'esprit des technologies open source : il est possible de réutiliser des technologies pour les adapter à ses propres besoins.

Des auteurs dans ce domaine ont montré une préférence pour la réutilisation boite noire tel est le cas pour Nierstrasz [13]. La facilité d'utilisation est obtenue en fournissant

[6] La notion de contrat sera expliquée plus loin dans ce chapitre

des interfaces 'boite noire' aux composants qui, d'une part limite la façon dont les composants peuvent être utilisés, et d'autre part, limite le besoin de comprendre les détails d'implémentation des composants. La vision historique de la réutilisation logiciels est la création de systèmes logiciels à partir d'éléments logiciels préexistant au lieu de les crées à partir de zéro. Le processus de la réutilisation logiciels se divise en deux parties correspondantes: l'ingénierie pour la réutilisation, et l'ingénierie avec la réutilisation [17]. La section suivante traite ces types de réutilisation.

3.3.2 Processus de réutilisation

Deux processus complémentaires sont considérés dans le développement basé composant : l'ingénierie de composants réutilisables (design for reuse) et l'ingénierie de systèmes par réutilisation de composants (design by reuse). Nous allons discuter ces processus séparément. Dans des situations réelles, ils seront combinés, et peuvent ne pas être distingués, comme étant des activités séparées. Toutefois, la séparation de ces processus est une caractéristique particulière de l'approche basée composant [1].

a. Ingénierie de systèmes par réutilisation (Design by Reuse)

Le développement de systèmes par assemblage de composants est similaire à celui des applications qui ne sont pas basées sur les composants. Il y a toutefois, une différence cruciale dans le processus de développement de systèmes à base de composant. L'intérêt n'est pas focalisé sur l'implémentation du système conçu, mais sur la réutilisation de composants préexistants. [18] définit le développement de systèmes logiciels par réutilisation comme le développement de systèmes en utilisant des composants réutilisables. L'ingénierie de systèmes par réutilisation couvre les taches de recherche, de sélection ou de création de composants propriétaire comme alternative, d'adaptation pour intégration, de composition et déploiement des composants, et remplacement de composants.

1) Recherche de composants

La recherche de composants consiste à rechercher un composant parmi une collection, permettant de répondre à un problème particulier. Elle tient donc compte du besoin du développeur. Pour accomplir avec succès cette procédure, un grand nombre d'éventuels candidats de composants doivent être disponibles ainsi que les outils pour les trouver.

2) Sélection

Sélectionnez les composants qui répondent aux exigences du système. Souvent, les exigences ne peuvent pas être satisfaites complètement et un effort supplémentaire d'analyse est nécessaire pour adapter l'architecture du système et de reformuler les besoins pour permettre l'utilisation des composants existants.

3) Création de composants

Alternativement, la création de composants propriétaires qui vont être utilisés dans le système peu se produire au lieu d'une sélection d'un entrepôt de composants. Dans une approche CBD, cette procédure est moins attrayante car elle exige plus d'efforts et de temps. Toutefois, les composantes de base incluant les fonctionnalités du produit sont susceptibles d'être développées en interne du fait qu'ils peuvent fournir un aspect compétitif du produit.

4) Adaptation

Une fois sélectionné, le composant doit être adapté au contexte du système en cours de développement. Il doit correspondre au modèle de composant existant ou à la spécification des besoins. Certains composants peuvent être directement intégrés au système, d'autre doivent être modifiés par un processus de paramétrisation, certains auraient besoin de code glu pour l'adaptation. Le code glu est le code ajouté aux composants afin de les rendre compatibles avec un environnement dans lequel ils n'ont pas été conçus pour fonctionner [3].

5) Composition et déploiement (intégration)

Dite aussi phase d'intégration. La phase d'intégration du système est la phase durant laquelle les composants sélectionnés et développés sont composés (c'est-à-dire assemblés entre eux) pour créer le système et cela en utilisant un Framework pour les composants. Typiquement, les modèles de composants fournissent ces fonctionnalités.

6) Remplacement

Le remplacement des versions de composants anciennes par les nouvelles. Cela correspond à la phase de maintenance du système. Les bugs sont éliminés et de nouvelles fonctionnalités peuvent être ajoutées au système.

b. Ingénierie de composants réutilisables (Design for Reuse)

Elle vise à développer des méthodes et des outils pour supporter le processus de production de composants mis en œuvre par le concepteur de systèmes de réutilisation [19]. L'ingénierie de composants réutilisables recouvre les taches d'identification, de spécification, de qualification, d'organisation et d'implémentation de composants. L'ensemble de ces taches est essentiel, puisqu'il a pour objectif de produire les ressources nécessaires à la mise en œuvre d'une approche d'ingénierie par réutilisation.

1) Identification

L'identification des ressources candidates à la réutilisation, soit par l'étude de systèmes existants, soit par l'analyse de domaine. Les connaissances susceptibles de

contenir des éléments réutilisables et approuvés peuvent être des produits de développement existants (programmes, modèles conceptuels, etc.), ou des expressions de connaissances (documents techniques, etc.).

2) Spécification

La spécification doit favoriser la réutilisation des composants tout en fournissant l'ensemble des informations utiles et nécessaires à leurs bonnes utilisations (telles que les indications et les contraintes d'utilisation, les forces de la solution proposée,…). Elle doit permettre de décider, par exemple, de la pertinence d'utiliser un composant ou un autre au moment de la recherche de ceux-ci. La réutilisation des composants est plus efficace s'ils sont associes a une spécification pertinente. Deux techniques sont utilisées dans tout modèle de spécification pour la spécification des composants : l'abstraction et la généricité. La première consiste à distinguer deux types de connaissances dans la spécification d'un composant : les connaissances effectivement réutilisables (un modèle s'il s'agit d'un composant conceptuel par exemple, ou un fragment de programme s'il s'agit d'un composant logiciel) et celles requises pour une utilisation correcte de celui-ci [15]. La deuxième introduit une forme de variabilité dans la spécification des composants. En effet, le plus souvent, la solution proposée par un composant n'offre qu'une réalisation possible de celui-ci. Pour augmenter sa réutilisation, un composant doit offrir plusieurs réalisations possibles : sa solution doit ainsi être flexible afin de pouvoir l'adapter aux spécificités du système en cours de développement.

3) Qualification

La qualification des composants s'apparente à une forme de validation de ceux-ci. Elle consiste à vérifier la qualité de composants identifiés du point de vue de leur réutilisation.

4) Organisation

L'organisation des composants traite différents types de liens entre ceux-ci, afin de pouvoir utiliser conjointement plusieurs composants pour obtenir des architectures plus importantes. Elle a donc une influence directe sur la manière d'exploiter les composants dans le processus d'ingénierie de systèmes à base de composant. Elle facilite la sélection de composants et améliore leur réutilisation.

5) Implémentation

L'implémentation des composants consiste a outiller la réutilisation (gestion automatique de leurs collections, recherche, sélection, composition, adaptation, …) de ceux-ci.

Ces différentes techniques contribuent à la mise en place de bases de composants pour capitaliser des savoirs et des savoir-faire réutilisables dans le cadre d'une ingénierie par réutilisation.

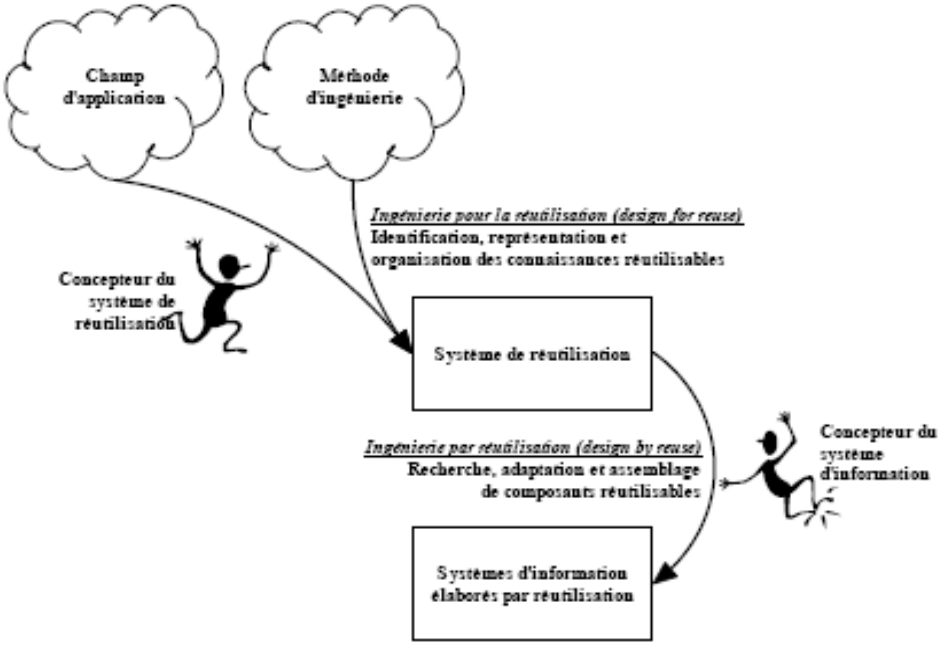


Fig. 3 Ingénierie pour la réutilisation et ingénierie par réutilisation [19]

L'objectif du développement pour la réutilisation est de construire des composants réutilisables qui répondent à un ensemble de contraintes de qualité et de réutilisation. Le développement de composants réutilisables se focalise sur les critères de qualité au détriment des coûts de production du composant [18]. Les activités de développement par réutilisation se focalisent sur l'ensemble du processus de développement, de l'identification, et la recherche de composants réutilisables dans la base de composants réutilisables, jusqu'à la construction de nouveaux composants (pas nécessairement ceux réutilisables) par l'adaptation ou la composition de composants réutilisables.

3.3.3 Cycle de vie pour le CBD

L'ingénierie logicielle basée composant CBSE aborde des défis et des problèmes similaires à ceux rencontrés en ingénierie logicielle. Beaucoup de méthodes, d'outils et principes en ingénierie logicielle utilisés dans d'autres types de système sont utilisés d'une manière identique ou similaire dans la CBSE. Il y a toutefois, une différence: la CBSE se concentre sur les questions relatives aux composants et, dans ce sens, elle distingue le processus de développement des composants de celui du développement de système avec des composants [14].

Il est important de remarquer que les différentes activités constituant les deux processus peuvent être réalisées indépendamment les unes des autres. En réalité [1], les deux processus sont réellement séparés parce que de nombreux composants sont développés par des tierces parties, indépendamment du développement du système. Cependant, dans des situations réelles ils sont combinés et peuvent ne pas être distingués comme activités séparées.

Comme les logiciels sont de plus en plus complexes, il en va de même pour l'effort nécessaire pour leur production, et dans ce sens de nombreux modèles de cycle de vie de logiciel ont été proposés. Leur principale utilité est d'identifier et d'organiser les phases et les étapes impliquées dans le développement et l'évolution des logiciels [20]. Il y a une différence significative entre l'approche de développement logiciel basé composant et les approches de développement logiciel traditionnelles. Une standardisation technique est nécessaire, et une méthode adaptée au CDB doit être suivi [21]. Les processus de cycle de vie incluent toutes les activités d'un produit ou un système durant toute sa vie, en commençant depuis l'idée de l'entreprise pour son développement, jusqu'à son utilisation et sa mise hors service. Différents modèles ont été proposés et exploités dans l'ingénierie logicielle. Nous avons mentionné dans la section 2.2 que deux principaux groupes de modèles peuvent être distingués: séquentiel et évolutionniste. Les modèles séquentiels définissent une séquence d'activités dans laquelle le passage vers une autre activité ne se fait qu'après achèvement de celle en cours. Les modèles évolutionnistes permettent l'exécution de plusieurs activités en parallèle sans l'exigence de compléter une activité pour pouvoir commencer une autre. Ces modèles peuvent être appliqués dans le développement basé composant, mais doivent être adaptés aux principes de l'approche basée composant [22]. Dans ce cadre, divers modèles de cycle de vie pour le développement basé composant ont été proposés. Nous allons présenter brièvement dans la section suivante la philosophie générale de quelques un. Indépendamment du type de modèle, nous pouvons identifier les activités de base présentes dans tout modèle de cycle de vie qu'on a présenté dans la section 2.1.

a. Modèle en V

Dans le modèle en V (figure 4) adopté pour l'approche de développement basée composant proposé par [22], durant le cycle de vie d'un produit logiciel (franchissant les phases de développement et de maintenance), un entrepôt de composants est mis à disposition depuis lequel les composants peuvent être sélectionnés dans les phases initiales, et dans lequel les composants peuvent être ajoutés dans les phases ultérieures (les composants nouvellement développés pour une application) [21].

Dans ce modèle le processus commence d'une manière habituelle par l'ingénierie des besoins et la spécification des besoins, où les analystes essayent de trouver, à partir d'un entrepôt de composants (chercher puis sélectionner) ceux pouvant satisfaire les besoins spécifiés. Dans une approche classique le processus continuerait avec la conception, l'implémentation et les tests. Si de tels composants ne sont pas disponibles, alors les besoins sont probablement négociés et modifiés afin de pouvoir utiliser les composants logiciels disponibles.

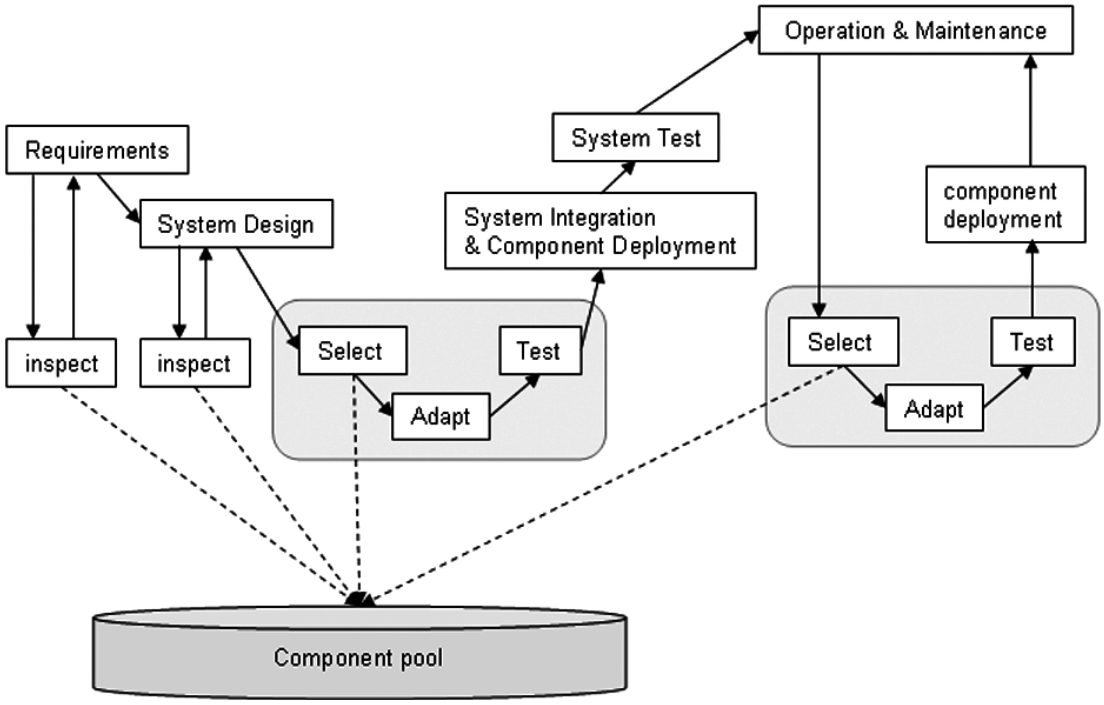


Fig. 4 Modèle en V adopté pour le développement basé composant [22]

b. Modèle en Y

Le modèle en Y (figure 5) a été proposé comme alternative viable pour adresser la réutilisation de logiciels durant la production de systèmes logiciels basé composant [20]. La création de logiciels se caractérise par le changement et l'instabilité, par conséquent la représentation schématique du modèle en Y permet le recouvrement de phases et des itérations quand c'est nécessaire. Bien que les phases principales peuvent se recouvrir et l'itération est permise, les phases prévues sont: ingénierie de domaine, établissement de Framework, assemblage, archivage, analyse du système, conception, implémentation, test, déploiement et maintenance.

La caractéristique principale de ce modèle de cycle de vie d'un logiciel est sa focalisation sur la réutilisation pendant la création et l'évolution du logiciel et la production de composants potentiellement réutilisables qui peuvent être utiles dans de futurs projets logiciels. La réutilisation dans ce cycle de vie est plus efficace que dans les modèles traditionnels parce qu'il intègre dans son noyau l'intérêt pour la réutilisation et les mécanismes pour la réaliser [20].

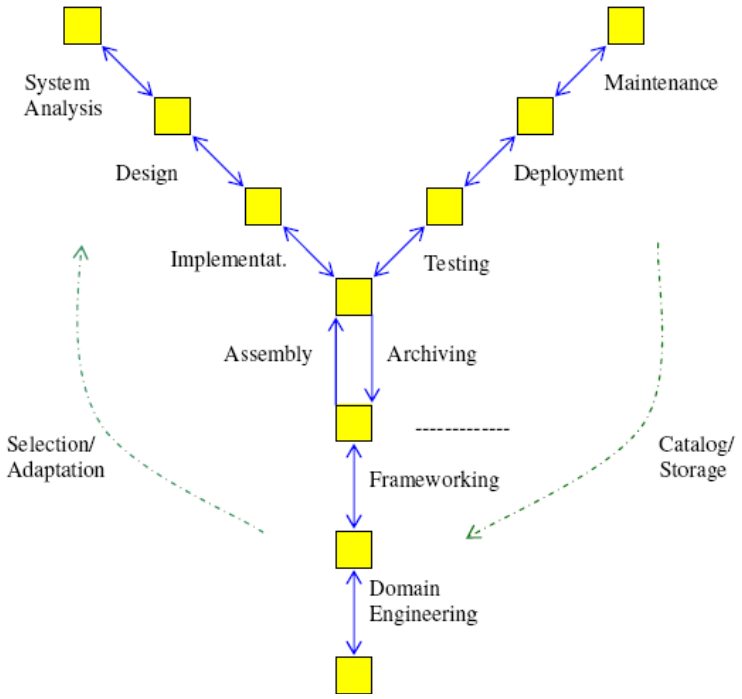


Fig. 5 Modèle en Y pour le développement logiciel basé composant [20]

c. L'approche EPIC (Evolutionary Process for Integrating COTS)

C'est une forme modifiée du processus rationnel unifié (RUP[7]). Pour s'adapter aux changements continus induits par le marché des COTS (Commercial Off The Shelf), EPIC utilise le modèle en spirale basé sur la gestion des risques. Les utilisateurs de l'approche EPIC gèrent la collecte d'informations du marché et raffinent ces informations par l'analyse et la négociation dans une solution cohérente qui est incorporée dans une série de représentations exécutables durant la vie du projet [21]. Les quatre phases sont, comme dans RUP, Analyse des besoins, élaboration, construction, et transition.

1) Le but de la phase d'analyse des besoins est d'atteindre la concurrence entre les utilisateurs affectés sur les objectifs de cycle de vie pour le projet. La phase d'analyse des besoins établit la faisabilité à travers le cas d'affaires qui montre qu'une ou plusieurs solutions candidates existent.
2) Le but de la phase d'élaboration est d'atteindre une stabilité suffisante au niveau de l'architecture et des besoins; pour sélectionner et acquérir des composants; et pour minimiser les risques de sorte qu'une solution simple de haute fiabilité puisse être identifiée avec le coût prévu.
3) Le but de la phase de construction est de dégager un produit de qualité prêt pour la communauté d'utilisateur concerné. La solution choisie est préparée pour sa mise sur terrain.
4) Le but de la phase de transition est à la livraison de la solution à ses utilisateurs. La solution choisie est mise sur terrain à la communauté d'utilisateur.

[7] Rational Unified Process

Beaucoup de gens dans l'industrie logiciel commencent à voir le CBD comme nouvelle approche passionnante au développement d'application qui offre la promesse de réduire la durée du cycle de développement de logiciel, et améliore la qualité des applications fournies [9]. Le CBD a beaucoup d'avantages. Ceux-ci incluent une gestion plus efficace de la complexité, réduire le temps de mise sur le marché, une meilleure productivité, une qualité améliorée, et un large spectre d'utilisation. Cependant, il y a plusieurs inconvénients et risques dans l'utilisation du CBD qui peut compromettre son succès [14] :

- Le temps et l'effort nécessaire pour le développement des composants. Parmi les facteurs qui peuvent décourager le développement des composants réutilisables est le temps et l'effort croissant exigé, la construction d'une unité réutilisable a nécessite de trois à cinq fois l'effort exigé pour développer une unité pour un usage spécifique.
- Les besoins peu clairs et ambigus. En général, la gestion des besoins est une partie importante du processus de développement, son objectif principal est de définir les besoins complets et consistent des composants. Par définition, les composants réutilisables doivent être utilisés dans différentes applications, dont certains peuvent encore ne pas être connues et dont leurs besoins ne peuvent pas être prévus. Ceci s'applique aux besoins fonctionnels et non fonctionnels.
- Conflit entre la réutilisation et la réutilisabilité. Pour être largement réutilisable, un composant doit être suffisamment général, extensible et adaptable et donc plus complexe à utiliser, et demande plus de ressources informatiques, donc il est plus chère à utiliser. Le besoin de la réutilisabilité, qui consiste à dégager, pour le composant, un potentiel à être aisément réutilisable [3], peut mener à une autre approche de développement, par exemple établir un nouveau niveau, plus abstrait qui donne moins de flexibilité, mais atteint une meilleure simplicité.
- Les coûts de maintenance des composants. Tandis que les coûts de maintenance d'application peuvent diminuer, celles des composants peuvent être très élevées puisque le composant doit répondre à différents besoins de différentes applications fonctionnant dans des environnements divers, avec des exigences en termes de fiabilité différentes.
- Fiabilité et sensibilité aux changements. Comme les composants et les applications ont des cycles de vie séparés et différents types de besoins, il y a un certain risque qu'un composant ne répondra pas complètement aux exigences d'une application ou qu'il peut inclure des caractéristiques cachées non connues pour les développeurs d'applications. On introduisons des changements au niveau d'application tel que la mise à jour du système d'exploitation, la mise à jour d'autres composants, etc.…, il y a un risque que le changement effectué causera l'échec du système.

Cependant, l'apparition de nouvelles technologies a augmenté de façon remarquable les possibilités de construire des systèmes et des applications à partir de composants réutilisables. Les clients et les fournisseurs s'attendaient beaucoup du CBD, mais leurs

espoirs n'ont pas été toujours satisfaits. L'expérience a montré que le développement basé composant exige une approche systématique pour se focaliser sur les aspects de composants dans un développement logiciel [14].

Les disciplines traditionnelles de l'ingénierie logicielle doivent être ajustées sur la nouvelle approche, et de nouvelles procédures doivent être développées. L'ingénierie logicielle basée composant CBSE est devenue une nouvelle sous discipline de l'ingénierie logicielle.

3.4 Ingénierie logicielle basée composant (CBSE)

Il est évident que le CBD et le CBSE sont dans les premières phases de leur expansion. CBD est connue comme étant une nouvelle approche puissante qui n'a amélioré de manière significative si ce n'est pas révolutionner le développement et l'utilisation des logiciels en général.

Les concepts de CBD et de CBSE sont parfois utilisés comme des synonymes. Pourtant leurs signification n'est pas la même. Le CBD traduit l'idée de création d'une application par utilisation/réutilisation de composants. La CBSE désigne les méthodes, techniques et outils permettant la conception et la mise en œuvre d'applications basées composants. Elle a ainsi permis au CBD de passer d'une approche artisanale à une approche industrielle [3].

Les principales missions de la CBSE sont les suivantes [1] :

- fournir un support pour le développement des composants ;
- supporter le développement des composants en tant qu'entités réutilisables, c'est-à-dire anticiper la réutilisation et concevoir la réutilisabilité ;
- faciliter la maintenance et l'évolution des systèmes en permettant la modification et le remplacement de leurs composants.

Les définitions des concepts inhérents à la CBSE sont encore mal formalisées et portent parfois à confusion. Le flou qui entoure certaines définitions vient de la manière dont la CBSE a été créée et a évolué, aux confins de plusieurs domaines. Nous citons les deux principaux : l'approche orientée objet et les travaux portant sur les architectures logicielles. L'expérience tirée de l'approche objet a été fondatrice pour la prise en considération de l'aspect réutilisation dans la CBSE.

3.4.1 Concepts de base

La CBSE est basée sur le concept de composant. D'autres termes, tels que l'interface, le contrat, le Framework, et le patron, sont de ce fait étroitement liées au développement logiciel basé composant.

Un composant est une unité réutilisable de déploiement et de composition. Une vue commune est qu'un composant est étroitement lié à un objet et que le CBD est donc une extension du développement orienté objet. Cependant, beaucoup de facteurs, tels que la granularité, les concepts de composition et de déploiement, et même les processus de développement, distinguent clairement les composants des objets [1]. Nous allons essayer de présenter une vue d'ensemble sur ces termes, et leurs définitions.

3.4.1.1 Interface

Un composant doit être clairement différencié des autres composants et de leur contexte d'exécution. Le mécanisme permettant cela est l'interface. Les interfaces sont un mécanisme de contrôle des dépendances existantes entre les différents modules d'un programme ou d'un système [3]. Une interface d'un composant peut être définie comme une spécification de ses points d'accès [23]. C'est-à-dire qu'elle offre la liste des services fournis ou requis par le composant. Les clients accèdent aux services fournis par le composant en utilisant ces points. Il est important de noter qu'une interface n'offre aucune implémentation d'aucune de ses opérations. Au lieu de cela, elle se présente sous la forme d'une collection de noms d'opérations. Elle fournit une description pour chaque opération. Cette séparation entre l'implémentation du composant et son interface permet de: remplacer la partie implémentation sans changer l'interface, et de cette façon améliorer les performances du système sans reconstruire le système; et, d'ajouter de nouvelles interfaces (et des implémentations) sans changer l'implémentation existante, et de cette façon améliorer l'adaptabilité du composant.

Les interfaces définies dans les technologies de composants standard (par exemple, le langage de définition d'interface IDL[8] de CORBA ou COM) peuvent seulement exprimer les propriétés fonctionnelles, et cela on se focalisant seulement, partiellement sur la partie syntaxique [23]. En général, les propriétés fonctionnelles incluent une partie de signature dans laquelle les opérations fournies par un composant sont décrites, et une partie de comportement, dans laquelle le comportement du composant est spécifié. Nous pouvons distinguer deux types d'interfaces. Les composants peuvent exporter et importer des interfaces depuis et vers l'environnement qui peut inclure d'autres composants.

Une interface exportée décrit les services fournis par un composant à l'environnement, alors qu'une interface importée spécifie les services requis par un composant de l'environnement [1]. La mise en œuvre de la notion d'interface est maintenant aisée car de nombreux langages de programmation récents la supportent, comme par exemple Java. Dans le cas contraire, des langages de spécification d'interfaces ont été également développés, indépendamment de tout type de langage, comme le langage IDL définit par le DCE (Distributed Computing Environment). Un exemple de spécification d'interface en IDL est donné dans la figure 6. Elle représente une interface de service fournie (ProvIneterPAM) d'un composant simple (un pilote automatique de bateau). Cette interface décrit deux opérations : setCap et setTarget.

[8] Interface Definition language

```
1   interface ProvIneterPAM {
2           void setCap (out int iCap) ;
3           void setTarget (in float iLattitude , in float iLongitude) ;
4   } ;
```

Fig. 6 Exemple de définition d'interface en IDL [3]

3.4.1.2 Contrat

La plupart des techniques pour décrire les interfaces telles que IDL sont seulement concernées par la partie de signature, dans laquelle les opérations fournies par un composant sont décrites. Ils ne décrivent pas le comportement global du composant mais juste la liste de ces services [1]. La notion de contrat permet notamment de pallier cette lacune. Il est admis que les contrats permettent de spécifier de manière plus rigoureuse le comportement des composants.

Un contrat liste les contraintes globales que le composant doit maintenir (invariants) [30]. Pour chaque opération d'un composant, un contrat liste également les contraintes qui doivent être assurées par le client (pré-conditions) et celles que le composant promet d'établir en retour (post-conditions). Les pré-, post-conditions et invariants constituent une spécification du comportement d'un composant [23]. Au delà des spécifications du comportement d'un composant simple, les contrats peuvent également être utilisés pour spécifier les interactions dans une collection de composants. Un contrat spécifie les interactions entre composants, en termes de [1]:

- L'ensemble de composants participant;
- Le rôle de chaque composant à travers ses engagements contractuels, tels que les engagements de type, qui exigent du composant de supporter certaines variables et une interface, et les engagements causals, qui exigent du composant de réaliser une séquence ordonnée d'actions, y compris l'envoi de messages aux autres composants.
- L'invariant à maintenir par les composants;
- La spécification des méthodes qui font l'instanciation du contrat.

L'utilisation des contrats pour spécifier les interactions entre les composants nécessite l'utilisation de langage. Eiffel est un exemple de langage qui permet la description des interfaces par contrats. Les contrats permettent aux développeurs de logiciel d'isoler et de spécifier explicitement, à un haut niveau d'abstraction, les rôles de différents composants dans un contexte particulier. En second lieu, les différents contrats permettent la modification et l'extension des rôles de chaque participant indépendamment. Finalement les nouveaux contrats peuvent être définis en associant différents participants avec différents rôles.

3.4.1.3 Composition

La composition de composants permet de créer soit une application basée composants, soit de construire un composant de granularité plus importante. Il existe de nombreuses définitions de la notion de composition. Parmi celles-ci, la définition suivante : « la composition logiciel est la construction d'applications logicielles à partir de composants qui implémentent des abstractions concernant un problème de domaine particulier[9]» [3]. La composition logicielle y est décrite comme un mécanisme, sans préciser lequel, permettant la construction d'une application à partir de composants. Selon [3], il existe deux niveaux de composition :

- La composition haut niveau, ou composition conceptuelle, concerne la phase d'analyse et la conception globale du système. On traite l'activité d'identification et de description des abstractions fondamentales du système logiciel (composants) et les relations entre elles (compositions). Il s'agit de considérer la composition au niveau modèle et donc de préparer et guider l'assemblage au niveau programmation. Les langages de modélisation et de spécification permettent la réalisation de cette phase.
- La composition bas niveau, ou composition logicielle, concerne :
  - La description de l'architecture globale du système et la spécification de la conception détaillée. Cette phase peut être réalisée conjointement avec des ADL (Architecture Description Language) et des langages de modélisation et de spécification.
  - L'implémentation et le déploiement des compositions. Cette phase peut être réalisée à l'aide des modèles de composants, des langages de script ou de glu, ou encore de simples langages de programmation.

3.4.1.4 Patron

Un architecte nommé Christopher Alexander a introduit pour la première fois le nouveau concept de patron dans la fin des années 70. Dans ce contexte, 'un patron définit une solution récurrente pour un problème récurrent'. Les patrons de conception forment une méthode de réutilisation des informations de conception. Ils ont été introduits dans le monde OO (Orienté Objet). Gamma et al [24] ont raffiné après cette définition en spécifiant les caractéristiques d'un patron et ses objectifs.

On peut établir quatre catégories de patrons. Chaque catégorie est basée sur le niveau d'abstraction à partir duquel ils sont utilisés [23] :

1) Les patrons d'architecture travaillent sur les propriétés globales et architecturales du système. Ils décrivent la structure globale du système, et fournissent des moyens de décrire l'ensemble des sous-systèmes, de spécifier leur rôle et de décrire les relations existant entre eux ;

[9] En anglais « Software composition is the construction of software applications from components that implement abstractions pertaining to a particular problem domain »

2) Les patrons de conception raffinent la structure et les comportements des sous-systèmes et des composants. Par leur aspect générique, les patrons sont considérés comme des microarchitectures visant à réduire la complexité, à promouvoir la réutilisation et à fournir un vocabulaire commun aux concepteurs. Ils décrivent la structure et le comportement du système et des composants et les communications entre eux ;
3) Les patrons de programmation sont dépendants des paradigmes et des langages de programmation utilisés.

La figure 7 montre un exemple d'un type de parton de conception de Gamma nommé le pattern Observer.

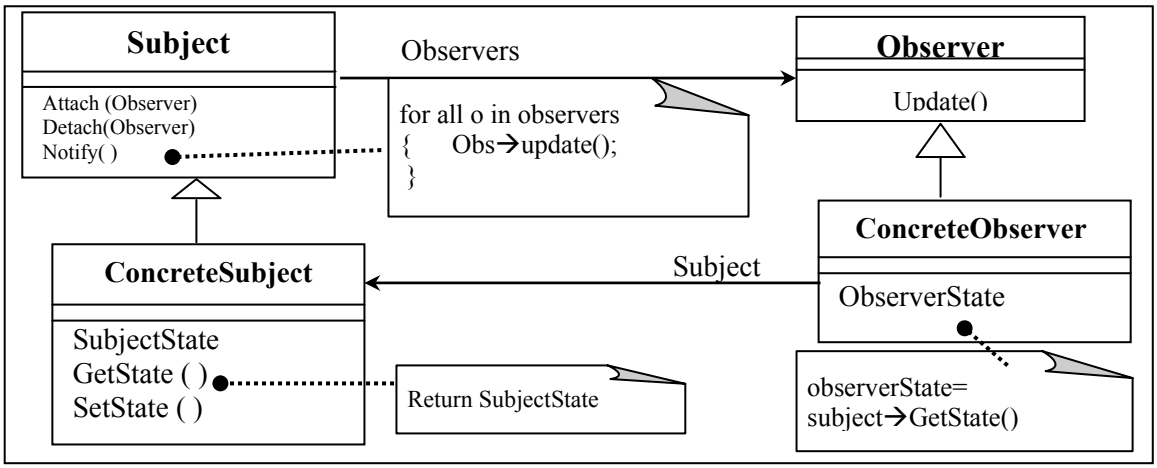


Fig. 7 La structure du Pattern Observer [24]

Les patrons ont été appliqués dans la conception de beaucoup de systèmes orientés objet, et sont considérés comme des microarchitectures réutilisables qui contribuent dans l'architecture globale d'un système. Cependant, l'utilisation des patrons de conception n'est pas sans problèmes. Le premier problème est que la connaissance codée dans le patron de conception est une connaissance non structurée porteuse de nombreuses ambiguïtés en raison de la description informelle des solutions. La relation existante entre patrons de conception et composants peut être vue comme suit : les patrons de conception, dans le processus de conception des applications basées composants, aident à déterminer les unités réutilisables et, le cas échéant, à les identifier sous la forme de composants existants, ou à les développer sous la forme d'unités réutilisables. Enfin, les patrons de conception peuvent être utilisés pour décrire les compositions de composants [1]. Il est envisageable de décrire dans des patrons, différents cas de composition type que l'on pourrait utiliser.

3.4.1.5 Framework

Tout comme la définition de composant, la notion de Framework, est un autre moyen aidant à réaliser une application basée composants. CBSE signifie que nous construisons des logiciels assemblant plusieurs pièces ensemble. Dans un tel environnement il est essentiel qu'un contexte existe dans lequel les pièces peuvent être utilisées. Les Framework constituent un moyen pour fournir de tels contextes. Un Framework peut être

vu comme une conception réutilisable d'une partie d'un système, fournissant un ensemble de classes abstraites ainsi que les moyens de les faire interagir. Il peut être utilisé pour décrire quelque chose qui est utilisable, non seulement tel quel dans une situation particulière, mais également, après modification, dans une situation similaire à la situation conçue initialement. Cette modification peut avoir lieu durant la phase de conception ou d'exécution, et est appelée instanciation du Framework [1]. La contribution majeure des Frameworks est qu'ils forcent les composants à effectuer leurs tâches à l'aide de mécanismes contrôlés par le canevas lui-même, ce qui renforce le respect des principes architecturaux [3]. Un exemple de canevas d'applications le JAF (pour JavaBeansTM Activation Framework).

Les Frameworks décrivent une situation réutilisable typique au niveau modèle, tandis que les Frameworks de composants constituent une carte circuit avec des slots vides en attente pour être rempli de composants. Quand ceci est fait le Framework est instancié [23]. On distingue trois types ou niveaux de Frameworks [23] :

1) Les Frameworks techniques qui sont semblables à ce qui se nomme autrefois des Frameworks d'application;
2) Les Frameworks industriels ;
3) Les Frameworks d'application.

3.4.2 Relation entre concepts

Dans la communauté du CBSE, il y a souvent des confusions sur les concepts de base qu'on a présentés. Par exemple, le concept des modèles de composants est souvent sujet de confusion avec le concept des Frameworks de composants. La figure 8 définit le modèle de composant comme un ensemble de types de composant, leurs interfaces, et, en plus, la spécification de leurs partenaires d'interaction parmi les types de composants. Un Framework de composant fournit une variété de services de déploiement et d'exécution pour supporter le modèle de composant. La figure 8 tirée de [25] schématise la relation entre les concepts abordés plus haut.

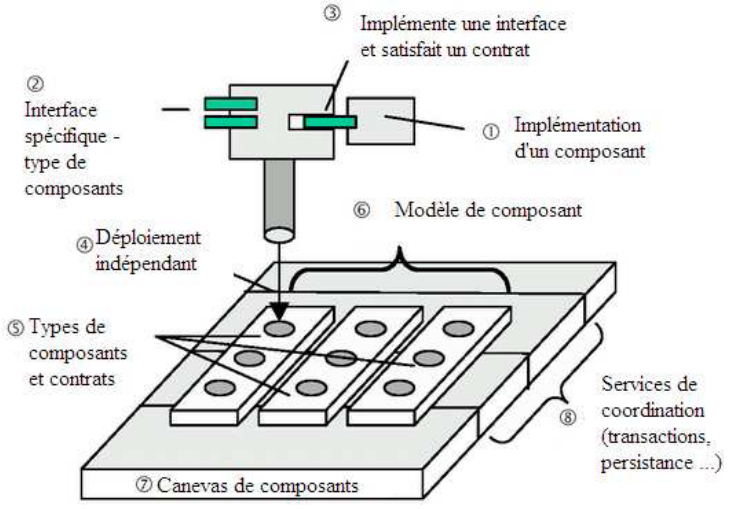


Fig. 8 Relation entre les concepts pour une application basée composants [25]

Un composant (1) est une implémentation logicielle mettant en œuvre un ensemble de services décrits par une ou plusieurs interfaces (2). Les composants respectent certaines obligations qui sont décrites par des contrats (3). Ceux-ci assurent que des composants développés indépendamment obéissent à des règles d'interaction, et peuvent être déployés dans un environnement de compilation et d'exécution standard (4). Une application basée composants est construite à partir d'instances de composants. Chaque composant est basé sur un type de composants (5). Un modèle de composants (6) est un ensemble de types de composants, de leurs interfaces, et éventuellement de patrons de conception décrivant les interactions entre les différents types de composants. Un canevas d'applications (7) fournit un ensemble de critères de déploiement et d'exécution (8) comme support au modèle de composants.

3.4.3 Notion de composant

Les composants représentent le cœur de la CBSE, et on a besoin d'une définition précise de ce que c'est un composant afin de comprendre les bases du CBSE. Plusieurs définitions du concept de composant peuvent être trouvées dans la littérature. Ces définitions diffèrent naturellement en fonction du domaine d'application et en fonction de la compréhension qui peut être attachée aux composants [27].

3.4.3.1 Définition d'un composant

Dans le domaine de la réutilisation du logiciel par exemple, un composant est vu comme n'importe quelle entité qui peut être réutilisée. Les concepteurs voient les composants comme des éléments de conception. Les déployeurs les considèrent comme des unités de déploiement. Dans d'autres considérations, un composant peut être également vu comme un élément d'un projet, comme une configuration ou même comme une documentation. A l'heure actuelle, il n'existe pas de normalisation de la notion de composant. Généralement un composant réutilisable est défini comme « Une unité de conception (de n'importe quel niveau d'abstraction) identifiée par un nom, avec une structure définie et des directives de conception sous la forme de documentation pour supporter sa réutilisation » [28]. La documentation d'un composant illustre le contexte dans lequel il peut être utilisé en spécifiant les contraintes et les autres composants dont il a besoin pour offrir sa solution.

Cette définition du concept de composant est générale car elle admet la possibilité d'utiliser le concept de composant dans des niveaux d'abstraction autres que le niveau d'implantation. D'une manière générale, un composant est vu comme une solution testée et acceptée pour résoudre un problème fréquemment rencontre lors du développement de systèmes [15].

3.4.3.2 Type de composant

Pour intégrer la réutilisation dans tout le processus de développement des systèmes à base de composant, une grande variété de composants réutilisables a été proposée. Il existe

trois types de composant qu'on peut trouver dans la littérature: les composants conceptuels, les composants logiciels et les composants métier.

a. Composant conceptuel

« Un composant conceptuel est une solution à un problème conceptuel sous la forme d'un modèle (ou une partie d'un modèle) destinée à être réutilisée ». Suivant l'approche de modélisation utilisée, la solution peut être spécifiée avec le langage UML (Unified Modeling Language) ou tout autre langage de modélisation. Les composants conceptuels peuvent être de deux types :

- Les composants produit : un composant produit est une partie cohérente d'un modèle qui peut être réutilisée avec d'autres composants produit pour assembler un modèle complet. Un produit correspond au but à atteindre lorsqu'on utilise la solution offerte par le composant produit. Par exemple, le patron « composite » de Gamma [24] peut être considéré comme un composant conceptuel produit.
- Les composants processus : un composant processus est une partie cohérente d'un processus qui peut être réutilisée avec d'autres composants processus pour assembler un processus complet. Un processus correspond au chemin à parcourir pour atteindre la solution offerte par le composant processus. Par exemple, le patron « revue technique » d'Ambler est un composant conceptuel processus [29].

b. Composant métier

A l'heure actuelle, il n'existe pas de normalisation de la notion de composant métier. Le concept de composant métier résulte de celui d'objet métier. Ainsi, l'OMG (Object Management Group) définit la notion d'objet métier comme suit: « Business Objects are representations of the nature and behaviour of real world things or concepts in terms that are meaningful to the enterprise. Customers, products, orders, employees, trades, financial instruments, shipping containers and vehicles are all examples of real-world concepts or things that could be represented by Business Objects » [33].

D'autres spécialistes définissent un composant métier comme une unité de réutilisation de connaissances de domaines, d'un point de vue conceptuel uniquement, par exemple, Casanave dans la définition suivante [26]:

> « Un composant métier est vu comme une représentation de la nature et du comportement d'entités du monde réel dans des termes issus du vocabulaire d'une entreprise » [34].

c. Composant logiciel

Les composants logiciels sont définis de diverses manières et ils n'ont pas de définition universellement acceptée. Parmi celles proposées, nous avons choisis de présenter quelques définitions les plus connues données par les experts. L'une des premières définitions est donnée par Gready Booch :

« Un composant logiciel réutilisable est un module logiquement cohésif et faiblement couplé qui dénote une simple abstraction [10]».

Cette définition reflète l'idée qu'un composant réutilisable est un module logiciel encapsulé comprenant des éléments étroitement liés [11]. Plus tard, Clément Szyperski a présenté sa célèbre définition d'un composant logiciel dans la Conférence européenne sur la programmation orientée objet en 1996.

« Un composant logiciel est une unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites au contexte. Un composant logiciel peut être déployé indépendamment et est sujet à une tierce composition ».

Cette définition est la plus couramment admise parmi celle proposées. Elle montre trois propriétés d'un composant :

1) Un composant encapsule totalement son implémentation et communique avec son environnement au travers de ses interfaces. Les interfaces spécifient les fonctionnalités fournies par le composant, et pour assurer ces fonctionnalités, certaines dépendances vers d'autres composants ou même vers l'environnement doivent être résolues
2) Un composant est une unité de composition et est sujet à composition. il doit donc être conçu pour la composition, c'est-à-dire pour être combiné avec d'autres composants afin de construire des entités logicielles de taille plus importante.
3) Un composant peut être déployé d'une manière indépendante. Il n'est pas nécessaire de l'intégrer dans une application pour pouvoir le déployer.

En 2000, une notion plus générale d'un composant logiciel a été définit par Alan W. Brown [9]: « une partie fonctionnelle indépendamment délivrée qui permet l'accès à ses services à travers des interfaces.[11] »

Récemment, Bill Councill et George T. Heineman ont donné une nouvelle définition qui souligne l'importance, d'un modèle de composants et de son standard de composition dans la construction de composant et de logiciel à base de composant [11].

10 En anglais « A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction ».
11 En anglais « An independently deliverable piece of functionality providing access to its services through interfaces».

« Un composant logiciel est un élément logiciel conforme à un modèle de composants et peut être indépendamment déployé et composé sans modification selon un standard de composition.»

3.4.3.3 Modèle de composants

Un modèle de composants consiste en un ensemble de conventions à respecter dans la construction et l'utilisation des composants [26]. L'objectif de ces conventions est de permettre de définir et de gérer d'une manière uniforme les composants. Elles couvrent toutes les phases du cycle de vie d'un logiciel à base de composants : la conception, l'implantation, l'assemblage, le déploiement et l'exécution. Concrètement, un modèle de composants décrit certains aspects des composants comme la définition de composants à partir d'objets (classes, modules, etc.), les relations entre les composants, les propriétés fonctionnelles et non fonctionnelles de chaque composant, les techniques d'assemblage des composants, le déploiement et l'exécution d'un logiciel à base de composants, etc.

Dans la pratique, un modèle de composants donné est spécifique à une phase du cycle de vie du composant et du logiciel. On trouve ainsi des modèles de composants pour la phase de conception (patrons de conception), et d'autres pour les phases d'implantation et de déploiement (EJB, CCM, etc.) [26]. Dans la section suivante des exemples sur des standards de composants logiciels sera présenté.

3.4.3.4 Technologies Standards pour le CBD

Nous présentons dans cette section quelques exemples de modèles de composants :

a. Component Object Model

COM (Component Object Model) [31] est un modèle de composants développé par Microsoft en 1995. Son objectif est de permettre le développement d'applications en assemblant des composants pouvant être écrits en différents langages, et communiquant en COM. Ces composants, pour qu'ils puissent interagir, doivent adhérer à une structure binaire commune, spécifiée par Microsoft. Ceci signifie que les compilateurs des différents langages de programmation, génèrent les composants suivant le même format binaire. Un composant COM, écrit dans un langage donné, peut ainsi interagir avec un autre composant, même s'il est écrit dans un langage différent [27].

b. Java Beans

Le modèle des Java Beans a été proposé par Sun Microsystems en 1996. Son premier et principal domaine d'application est la construction d'interfaces graphiques, mais son utilisation peut être étendue à d'autres domaines.

Suivant la définition de Sun Microsystems, un Java Bean est un composant logiciel réutilisable qui peut être manipulé visuellement à travers un outil logiciel [7]. Un Java

Bean possède une interface qui décrit ses méthodes et ses attributs. La composition des instances repose sur la notification et l'écoute d'événements. Le modèle des Java Beans a contribué à l'évolution de la notion de composant sur étagère. Le modèle de communication par événements a révolutionné la façon d'assembler des composants logiciels.

Cette forme de composition permet à des Java Beans n'étant pas initialement conçus pour fonctionner ensemble, d'être composés a posteriori : il suffit que l'un soit producteur et l'autre consommateur d'un même type d'événement. Pour faciliter un assemblage a posteriori, il faut évidemment posséder des types d'événements standards à partir desquels les Java Beans sont définis [7].

c. Enterprise Java Bean

Le modèle des composants EJB (Enterprise Java Bean) est le fruit d'un travail de normalisation, publié dans sa version initiale en juin 1998 (spécification 1.0). Il est dédié au développement, à la gestion et à l'évolution de composants serveurs au sein d'applications distribuées 3-tiers (clients/serveurs/bases de données) [7].

Plus particulièrement, un EJB est un composant logiciel écrit en Java, et s'exécutant du côté serveur. La norme EJB est essentiellement orientée vers le développement d'applications offrant des services complémentaires au dessus d'un système d'information d'entreprise [27]. La couche client comporte les applications (applets, applications Java, pages HTML…) qui permettent aux utilisateurs d'interagir avec les données de l'entreprise. La couche donnée concerne les bases de données de l'entreprise. Enfin, la couche serveur joue le rôle d'intermédiaire entre les deux autres couches. Elle englobe un serveur Web incluant les JSP (JAVA server page), les servlets JAVA, et un serveur de composants métiers lié à des conteneurs EJB [27].

d. CORBA Component Model

Le modèle à composants de CORBA (CCM) ajoute une couche au dessus de l'intergiciel CORBA permettant de définir l'architecture d'une application distribuée sous forme de composition d'instances de composant [32]. La description d'une composition, appelée 'Assembly', est réalisée de façon déclarative et permet de guider le déploiement, sur plusieurs emplacements, des classes de composant ainsi que la création, configuration, connexion et activation des instances de ces derniers. Une autre caractéristique de ce modèle est qu'il supporte l'implémentation de composants dans différents langages ; ceci est facilité par l'utilisation d'un langage abstrait pour la description d'interfaces (IDL).

Le modèle CCM repose sur deux points essentiels : le premier est qu'un composant CCM est une boite noire dont on ne sait rien de l'implantation; le second que la structure du composant doit être entièrement décrite en IDL [7]. La structure est définit en terme d'interfaces fournies (i.e. que le composant implémente) et d'interfaces requises (i.e. dont le composant requiert l'utilisation). A chaque interface, fournie ou requise, correspond un

port qui définit le mode de communication associé à cette interface. Il y a cinq différents types de ports [10]:

- Facettes, qui sont les interfaces fournies par les composants pour l'interaction du client;
- Réceptacles, qui sont les points de connexion qui décrivent les interfaces utilisées par le composant;
- Sources d'événement, qui sont les points de connexion qui émettent des événements d'un type spécifique aux consommateurs d'événement intéressés;
- Puits d'événement, qui sont les points de connexion dans lesquels des événements d'un type spécifique sont annoncés;
- Les attributs, qui sont des valeurs principalement utilisées pour la configuration du composant.

4. Conclusion

Nous avons présenté dans ce chapitre une vue d'ensemble sur le génie logiciel à base de composant. Pour ce faire, on a commencé tout d'abord, par une brève description des différentes approches classiques qui on précédées les approches de l'ingénierie logicielle à base de composants et qui ont été proposées en ingénierie logicielle. Ces approches ont montré des insuffisances et des lacunes en étant concentré sur un système à la fois sans considérer les besoins évolutionnistes du système. Le développement basé composant (CBD) a été une solution clé pour pallier aux problèmes des approches classiques. Cette approche favorise la réutilisation de briques logicielles préfabriquées pour construire des systèmes logiciels entièrement nouveaux et cela par assemblage. Cela a induit bien évidement à réduire le temps et le coût de développement et à des logiciels de qualité meilleure. Dans le processus de développement des applications à base de composant, la sélection des composants logiciels représentant les entités réutilisables constitue une phase très importante. Les composants sélectionnés doivent être évalué. Le processus d'évaluation inclut des aspects techniques qui sont [1] : l'intégration, la validation et la vérification des composants. Dans de nombreux cas, il est peut être plus adéquat d'évaluer un ensemble de composants composés comme un assemblage que d'évaluer un composant [1]. L'intégration des composants sélectionnés en un assemblage résulte en une unité fonctionnelle. Dans de tels cas, il est nécessaire d'évaluer cette composition. Pour cette fin une spécification d'une telle composition ainsi que celles des entités élémentaire s'avère indispensable. C'est dont nous nous sommes proposé à faire dans ce mémoire en utilisant le formalisme des ECATNets (Extended Concurrent Algebraic Term Nets), qui sont un type de réseau de pétri algébrique de haut niveau intégrés dans la logique de réécriture. Ceux-ci seront le sujet du prochain chapitre.

CHAPITRE II :

ECATNets et logique de réécriture

1. Introduction

On entend par méthodes formelles, les langages, techniques, et les outils basés sur la logique mathématique. En effet la base des méthodes formelles est de nous permettre de construire des modèles mathématiques bien définis du système à analyser. Une fois le modèle en main, des techniques mathématiques de haut niveau peuvent être utilisées pour raisonner sur le modèle. En outre puisque la sémantique du modèle est bien définit grâce aux fondements mathématiques, on peut utiliser des outils qui peuvent analysés le modèle de diverses manières [47].

En tant que formalisme de spécification, les réseaux de Petri présentent un certain nombre de qualités très importantes. Une définition formelle permettant de construire des modèles exempts d'ambiguïté, un grand pouvoir d'expression facilitant la description des systèmes complexes, et leurs capacités d'être exécutable et interpréter par un programme offrant la possibilité de simuler le fonctionnement du système en cours de spécification.

Les ECATNets sont des types de réseaux de Petri algébrique de haut niveau. Ils ont été proposés comme une manière pour la spécification, la modélisation, et la validation des applications du domaine des réseaux de communication, la conception des ordinateurs, et d'autres systèmes complexes [51]. Ce qui les distinguent des autres types de réseaux est leur sémantique interpréter en termes de la logique de réécriture. La logique de réécriture introduite par 'Meseguer' est une logique pour raisonner correctement sur les systèmes concurrents ayant des états, et qui évoluent au moyen de transitions. Elle constitue un Framework ou tout modèle peut être naturellement représenté au sein de cette logique. A travers ce chapitre nous nous sommes penché sur la présentation de ce couplage entre les ECATNets et la logique de réécriture et cela de la manière suivante. D'abord un bref survol sur les RdPs, puis la présentation des RdPs de haut niveau et algébriques. Nous montrons ensuite les ingrédients qui nous permettrons de se mettre dans le bain de la logique de réécriture, cela se fera par le biais de quelques définitions de base. Ensuite, en parlons de cette logique et dans la logique des choses nous présenterons le langage Maude,

le langage de la logique de réécriture. Enfin, la dernière section de ce chapitre sera consacrée aux ECATNets dont nous allons détailler leurs formalismes en se basant sur des exemples illustratifs.

2. Réseaux de Petri

Les réseaux de Petri sont des outils de modélisation graphique et mathématique qui peuvent être appliqués à plusieurs systèmes. Ils sont considéré comme des outils prometteur pour décrire et étudier les systèmes qui se caractérisent comme étant concurrent, asynchrone, distribué, parallèle, non déterministe, et / ou stochastiques.

2.1 Définitions de base

Un réseau de Petri (RdP) est un graphe biparti constitué de places, de transitions et d'arcs orientés qui relient les transitions aux places et les places aux transitions. Il est représenté formellement par un quintuple PN = (P, T, F, W, M_0) où [35]:

$P = \{p_1, p_2, \ldots, p_m\}$ est un ensemble finie de places,

$T = \{t_1, t_2, \ldots, t_n\}$ est un ensemble finie de transitions,

$F \ (P \times T) \cup (T \times P)$ est un ensemble d'arcs,

$W : F \rightarrow \{1, 2, 3, \ldots.\}$ est la fonction définissant le poids porté par les arcs,

$M_0 : P \rightarrow \{0,1, 2, 3, \ldots.\}$ est le marquage initial,

$P \cap T = \emptyset$ et $P \cup T \neq \emptyset$

Une structure N = (P, T, F, W) d'un RdP sans aucun marquage initial est désignée par N.

Un réseau de Petri avec un marquage initial donné est dénoté par (N, M_0).

2.2 Règle de franchissement

Le comportement de plusieurs systèmes peut être décrit en termes des états du système et de leurs changements. Pour simuler le comportement dynamique d'un système, un état de marquage dans un RdP est changé suivant les règles de franchissement des transitions [35] :

1) On dit qu'une transition **t** est franchissable à partir d'un marquage M, si et seulement si chaque place **p** d'entrée de la transition contient au moins un nombre de jetons qui est supérieur ou égale au poids de l'arc reliant cette place d'entrée **p** avec la transition **t,** tel que : M (**p**) ≥ W (**p**, **t**) où M(p) représente le marquage de la place p et W (**p**, **t**) est le poids de l'arc reliant la place p avec la transition t.
2) Le franchissement d'une transition franchissable t enlève W (**p**, **t**) jetons à partir de chaque place d'entrée à la transition **t** et ajoute W (**t**, **p**) jetons dans chaque place de sortie de la transition **t**, où W (**t**, **p**) est le poids de l'arc reliant la transition t avec la place p.

2.3 Représentation graphique d'un réseau de Petri

L'un des aspects les plus agréables des réseaux de Petri est qu'il dispose d'une représentation graphique attrayante, qui accroît la lisibilité et facilite la compréhension des modèles. Sa structure peut être représentée à travers un graphe bipartie fait de deux types de sommet : les places et les transitions reliées alternativement par des arcs orientés qui portent des poids entier positifs, si un poids n'est pas porté alors il est égal à 1 [39]. Le marquage d'un RdP est représenté par la distribution de jetons dans l'ensemble de places telle que chaque place peut contenir un ou plusieurs jetons. L'exemple suivant (figure 9) montre une représentation graphique d'un RdP modélisant une réaction chimique $2H_2 + O_2 \rightarrow 2H_2O$. Les deux jetons dans chaque place d'entrée de la transition t dans la figure (a) montre que deux unités de H_2 et O_2 sont disponible et que la transition t est franchissable. Après franchissement le nouveau marquage est illustré dans la figure (b).

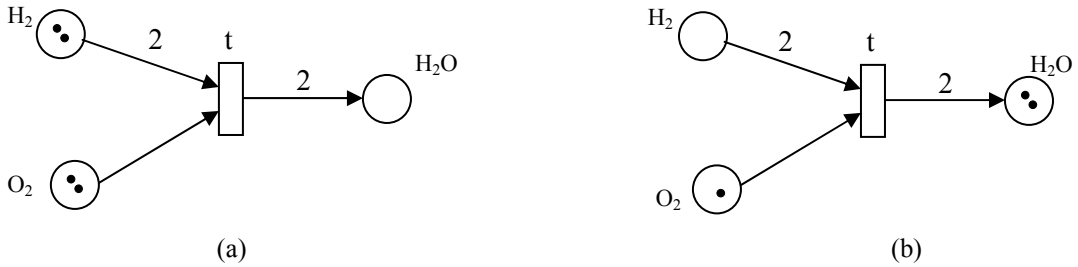


Fig. 9 Représentation graphique d'un RdP modélisant une réaction chimique :
(a) marquage avant de franchir la transition t, (b) marquage après le franchissement.

2.4 Propriétés comportementales d'un RdP

Un des points forts des réseaux de Petri est leur support pour l'analyse de nombreuses propriétés et les problèmes associés aux systèmes concurrents. Deux types de propriétés peuvent être étudiées avec un RdP: ceux qui dépendent du marquage initial et ceux qui sont indépendants du marquage initial. Le premier type représente les propriétés comportementales, alors que le deuxième représente les propriétés structurelles [36]. Nous allons se limiter à présenter les propriétés comportementales, étant donné que, à travers ces dernières on peut étudier l'aspect dynamique des systèmes modélisés.

a) Atteignabilité

Etant donné un RdP PN= (P, T, F, W, M_0) est un marquage M. le problème d'atteignabilité consiste à vérifié s'il existe une séquence de transitions qui transforme M_0 en M [36].

b) Bornitude

Une place Pi est bornée pour un marquage initial M_0 si pour tout marquage atteignable à partir de M_0, le nombre de jetons dans Pi reste borné. Elle est dite k-bornée si le nombre

de jetons dans Pi est toujours inférieur ou égal à k. Un RdP marqué est (k) borné si toutes ses places sont (k) bornées [37].

c) Vivacité et blocage

L'évolution du marquage d'un RdP se fait par franchissement de transitions. Lorsqu'au cours de son évolution, certaines transitions ne sont jamais franchies, cela indique que l'événement associé à la transition ne se produit pas et que le marquage d'une partie du RdP n'évolue pas. Cela indique que le sous système modélisé par cette partie-là ne fonctionnera pas [37].

- Une transition T_j est vivante pour un marquage initial M_0 si pour tout marquage atteignable M_k, il existe une séquence de franchissements à partir de M_0 contenant Tj.
- Un RdP marqué est vivant pour un marquage initial M_0 si toutes ses transitions sont vivantes pour ce marquage initial.
- Un marquage atteignable M d'un RdP est blocage si aucune transition n'est franchissable à partir de M.
- Un RdP marqué est dit sans blocage pour un marquage initial M_0 si aucun marquage atteignable n'est un blocage.

3. Réseaux de Petri de haut niveau

Les réseaux de Petri ont été utilisés pour décrire une large gamme de systèmes depuis leur invention en 1962. Leur problème est l'explosion du nombre d'éléments de leurs formes graphiques quand ils sont utilisés pour décrire des systèmes complexes. Les réseaux de Petri de haut niveau (HLPN[12]) ont été développés pour surmonter ce problème en introduisant des concepts de haut niveau, tels que l'utilisation de structures de données complexe comme des jetons, et en utilisant des expressions algébriques pour annoter les éléments du réseau [38]. Les HLPN peuvent être considéré comme l'intégration de la description de processus et de type de données [40]. L'avantage principal de cette intégration est qu'elle permet une présentation plus compacte des systèmes complexes [41]. Une des premières formes des HLPN sont les réseaux de Petri colorés introduit pour la première fois en 1979 et développé au cours des années 1980 et les réseaux de Petri algébrique de haut niveau (AHL nets[13]) [40].

3.1 Définition d'un HLPN

Une structure algébrique comprenant [38]:

- un ensemble de places, un ensemble de transitions,
- un ensemble de types,
- une fonction associant un type à chaque place,
- et un ensemble de modes (type) à chaque transition,

[12] High-Level Petri Nets
[13] Algebraic High-Level nets

- *Pre* est une fonction qui impose les exigences de jeton (multi-ensembles[14] de jetons) sur les places pour chaque mode de transition;
- *Post* une fonction qui détermine les jetons de sortie (multisets de jetons) pour les places pour chaque mode de transition ;
- et un marquage initial.

Un **HLPN** est une structure HLPN = (P; T; D; Type; Pre; Post; M_0) où [38]:

- P est un ensemble fini d'éléments appelés Places.
- T est un ensemble fini d'éléments appelés Transitions $P \cap T = \emptyset$.
- D est un ensemble non vide de domaines non vide où chaque élément de D est appelé un type

- Type: $P \cup T \rightarrow D$ est une fonction utilisée pour attribuer les types aux places et de déterminer les modes de transition.
- Pre; Post : TRANS $\rightarrow \mu$PLACE sont les mappages Pre et Post avec

  TRANS = {(t, m) | t $\in$ T, m $\in$Type (t)}

  PLACE = {(p, g) | p $\in$ P, g $\in$Type (p)}

- $M_0 \in \mu$PLACE est un multiset appelé le marquage initial du réseau.

3.2 Réseaux de Petri Algébriques de haut niveau

Le concept des réseaux de Petri algébrique de haut niveau (AHL nets) constitue une variante spécifique des réseaux de Petri de haut niveau, qui combinent les réseaux de Petri et les spécifications algébriques. Les réseaux de Petri sont connus comme des modèles fondamentaux de concurrence, les spécifications algébriques sont bien établies pour la spécification formelle de types abstraits de données et des systèmes logiciels. Les HLPN sont très utiles pour la spécification des systèmes concurrents et distribués [42]. Formellement un réseau de Petri algébrique est définit comme suit [52] :

Un tuple AN = [Σ; P, T, F; ψ, ξ, λ, m_0] est un réseau de Petri algébrique Ssi :

1) $\Sigma = [S, \Omega]$ est une spécification algébrique, où S est un ensemble de sortes et Ω un ensemble de symbole de fonctions. Σ est appelée signature;
2) [P, T, F] est un réseau i.e : P et T sont des ensembles finis est disjoints appelés respectivement places et transitions, et F est une relation $F \subseteq (P \times T) \bigcup (T \times P)$ où ses éléments sont appelés arcs ;
3) ϕ est une affectation de sorte $\psi : P \rightarrow S$;
4) ξ affecte un ensemble de Σ-variables ξ(t) pour chaque transition t$\in$ T ;
5) λ est l'inscription d'arcs tel que pour f = [p, t] $\in F$, $\lambda(f)$ est un multi-terme ;
6) m_0 est le marquage initial.

[14] Nous utiliserons dans le reste de ce mémoire le terme anglais Multisets

4. Logique de réécriture et Maude

Nous allons présenter dans cette section la logique de réécriture d'une manière non détaillée, mais avant cela il est nécessaire de comprendre ce que veut dire la réécriture ainsi que les éléments de base qui définissent cette logique. Ensuite nous allons présenter le langage Maude comme étant un langage de spécification formel basé sur la logique de réécriture.

4.1 Définitions de base

Définition 1. Une signature F est un ensemble d'opérateurs (symboles de fonction), chacun étant associé à un entier naturel par la fonction arité : $F \rightarrow N$, associant à chaque opérateur son arité. F_n désigne le sous ensemble d'opérateurs d'arité n [43].

On peut alors définir les termes construits sur ces opérateurs et un ensemble de Variables X.

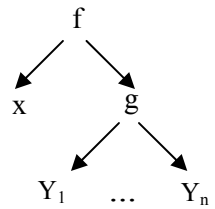
Définition 2. Les termes sur F sont définis par clôture comme le plus petit ensemble $T(F, X)$ tel que :

- $X \subset T(F, X)$: toute variable de X est un terme de $T(F, X)$
- Pour tous $t_1, \ldots, t_n$ termes de $T(F, X)$ et pour tout opérateur f d'arité n, $f(t_1, \ldots, t_n)$ est un terme de $T(F, X)$.

On peut définir l'ensemble $Var(t)$ des variables d'un terme t de manière inductive [43]:

- $Var(t) = \emptyset$ si $t \in F_0$,
- $Var(t) = \{t\}$ si $t \in X$
- $Var(t) = \bigcup_{i=1}^{n} Var(t_i)$ si $t = f(t_1, \ldots, t_n)$.

L'ensemble des termes clos (termes dans lesquels il n'y a pas de variables) est noté $T(F)$. On peut considérer un terme comme un arbre dont les feuilles sont des constantes (que l'on peut définir comme des opérateurs d'arité 0) ou des variables (si le terme n'est pas clos) et les nœuds sont des opérateurs d'arité strictement positive. Par exemple, le terme f(x, g(y1, . . . , yn)) peut être représenté par l'arbre :



La position d'un nœud dans cet arbre est définie :

Définition 3. Une position (ou occurrence) dans un terme t est représentée par une suite ω d'entiers naturels positifs décrivant le chemin de la racine du terme jusqu'à la racine du sous terme à cette position, noté $t_{|\omega}$. Un terme *u* a une occurrence dans t si $u = t_{|\omega}$ pour une position ω dans t. On note t $[t']_\omega$ pour indiquer le fait que le terme t a le terme t' à la position ω. Par exemple soit t=f (a, g (f(y, a))), $t_{|2}$=g (f(y, a)).

Nous allons présenter la définition d'une opération sur les termes nommée la substitution. Une substitution est une opération de remplacement spéciale, uniquement définie par une fonction des variables vers les termes. Effectuer une substitution consiste à remplacer une variable d'un terme par un autre terme [44].

Définition 4. La substitution de la variable x par le terme u dans le terme t, notée <x→*u*>t est la composition de chacun des remplacements du terme u à chacune des positions p telle que t(p) = x.

Définition 5. Une substitution est une fonction accomplissant en simultanées plusieurs substitutions de différentes variables par des termes. L'application d'une substitution à un terme t sera notée < $x \rightarrow u_1, \ldots, x_n \rightarrow u_n > t$ [44].

- Une paire de terme (l, r) est appelée égalité ou équation est notée (l=r). Par exemple, l'opération suivante définit sur une pile : *top (puch(x, y)) = x*.
- Étant donné un ensemble d'équations E sur un ensemble de terme *T*, la théorie équationnelle de E, *Th*(E), est l'ensemble d'équations qui peuvent être obtenues en prenant la réflexivité, la symétrie, transitivité, et l'application de contexte (ou la réflexivité fonctionnelle) comme règles d'inférences et toutes les instances d'équations dans E comme axiomes [45].

4.2 Systèmes de réécriture

L'idée centrale de la réécriture est d'imposer une direction dans l'utilisation d'axiomes, par la définition de règles de réécriture [44]. Contrairement aux équations, qui ne sont pas orientées, une règle sur un ensemble de termes T est une paire ordonnée <l, r> de termes, que l'on note l → r [45]. Les règles se différent des équations par leurs utilisation. Comme les équations les règles sont utilisées pour remplacer les instances de l par leurs correspondantes de r, en revanche, elles ne sont pas utilisées pour remplacer les instances de la partie droite r.

Définition 6. Une règle de réécriture est une paire ordonnée <l, r> de termes dans *T* (*F*, *X*), notée l → r. l est le membre gauche de la règle, et r son membre droite.

Un système de réécriture sur les termes est un ensemble de règles de réécriture. On impose généralement des conditions sur la construction des règles de réécriture [44]:

- les variables du membre droite de la règle font partie des variables du membre gauche (V ar(r) ⊆ V ar(l)).

- le membre gauche d'une règle n'est pas une variable ($l \notin X$)

La réécriture, c'est-à-dire le processus de remplacer les instances d'un motif (pattern) membre gauche d'une règle par l'instance correspondante au motif (pattern) du membre droite de cette règle, est un paradigme de calcul intrinsèquement concurrent [43]. Ainsi, une même règle peut être appliquée simultanément à différentes positions.

Exemple

L'exemple suivant montre la réécriture d'un terme en utilisant un système de réécriture *R* construit par des opérations sur une pile. Les règles de réécriture qui composent le système sont les suivantes [45]:

top (*puch*(*x,* y)) → *x*
pop (*puch*(*x,* y)) → *y*
alternate (*Λ, z*) → *z*
alternate (*puch*(*x,* y*), z*) → *puch* (*x, alternate* (*z, y*)

Une réécriture :

alternate (*puch*(*top*(*puch*(0, *z*)), *z*), *Λ*) →$_R$ *alternate* (*puch*(0,*z*), *Λ*)

→$_R$ *puch*(0, *alternate* (*Λ, z*)) →$_R$ *puch*(0, *z*).

La première étape est l'application de la règle *top-puch* à l'occurrence *top*(*puch*(0, *z*) ; la deuxième est celle de la règle *alternate-puch* avec 0 pour *x* et *z* pour *y* et *Λ* pour *z ;* la troisième est celle de la règle *alternate* (*Λ, z*) → *z*.

Après avoir présenté les concepts de base qui permettent de comprendre la logique de réécriture, nous allons introduire dans la section suivante ce que c'est cette logique.

4.3 Logique de réécriture

La logique de réécriture est destiné à servir comme modèle mathématique unifié et utilise les notions des systèmes de réécriture sur les théories équationnelles. Elle permet de séparer entre la description des aspects statiques et dynamiques d'un système distribué [53]. Plus précisément, elle distingue entre les lois décrivant la structure des états du système et les règles qui précisent ses éventuelles transitions. Cette logique, dotée d'une sémantique saine et complète, a été introduite par 'Meseguer'. Elle permet de décrire les systèmes concurrents. Plusieurs modèles formels qui expriment la concurrence peuvent être représentés naturellement dans cette logique.

La logique de réécriture est une logique qui permet de raisonner d'une manière correcte sur les systèmes concurrents ayant des états et évoluant en termes de transitions [54]. Elle a été proposée comme une manière d'interpréter les systèmes de réécriture [44]. La syntaxe nécessaire pour définir une logique est spécifiée par sa signature qui nous permet des construire des formules.

Une signature dans la logique de réécriture est une théorie équationnelle (Σ, E), où Σ est une signature équationnelle et E un ensemble de Σ-équation [54]. Elle exprime une structure particulière pour les états d'un système (par exemple : un multiset, arbre binaire…etc.) de sorte que ses états peuvent être distribué selon cette structure.

Etant donné une signature (Σ, E), les formules de la logique de réécriture sont des séquents de la forme [54] : $[t]_E \rightarrow [t']_E$, où t et t' sont des Σ-termes qui peuvent impliqués des variables de l'ensemble $X= \{x_1,\ldots, x_n,\ldots\}$ et $[t]_E$ dénote la E classe d'équivalence de *t*. L'ensemble Σ des symboles de fonction est un alphabet d'arité $\Sigma= \{\Sigma_n \mid n \in \mathbb{N}\}$.

Formellement la logique de réécriture est définit comme suit [55]:

Définition 7. Une théorie de réécriture $\mathfrak{R}$ est un 4-tuple $\mathfrak{R}= (\Sigma, E, L, R)$, où Σ est un alphabet de symboles de fonction, E un ensemble de Σ-équation, L est un ensemble d'étiquettes, et R est l'ensemble des paires $R \subseteq L \times (T_{\Sigma,E}(X))^2$ tel que le premier composant est une étiquette et le second est une paire des classes d'équivalence des termes modulo les équations E avec $X = \{x_1,\ldots, x_n,\ldots\}$ un ensemble comptable et infini des variables. Les éléments de R s'appellent les règles de réécriture, qui peuvent être conditionnelles ou inconditionnelles (si la partie condition est vide). On utilise la notation suivante pour décrire une règle de réécriture, r : $[t] \rightarrow [t']$. La forme la plus générale d'une règle de réécriture est comme suit : $r : [t] \rightarrow [t']\ \textit{if}\ [u_1] \rightarrow [v_1] \wedge\ldots\wedge [u_k] \rightarrow [v_k]$

Pour indiquer que $\{x_1,\ldots, x_n\}$ est l'ensemble de variables qui se produisent dans *t* ou *t'*, on écrit, r : $[t\,(x_1,\ldots, x_n)] \rightarrow [t'\,(x_1,\ldots, x_n)]$, on notation abréger on écrit, r :$[t\,(\bar{x})] \rightarrow [t\,(\bar{x})]$.

Les règles de réécriture décrivent les transitions élémentaires locales qui sont possibles dans l'état distribué d'un système, par des transformations locales concurrentes. Elle représente donc l'aspect dynamique d'un système. La réécriture fonctionne sur les classes d'équivalence de termes modulo E.

Étant donnée une théorie de réécriture $\mathfrak{R}$, on dit que le séquent $[t] \rightarrow [t']$ se déduit à partir de $\mathfrak{R}$ et on écrit : $\mathfrak{R} \vdash [t] \rightarrow [t']$ Ssi $[t] \rightarrow [t']$ (t deviens t') peut être obtenue en appliquant un nombre fini de fois les règles de déduction suivantes [55] :

1- Réflexivité. Pour chaque $t \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \rightarrow [t]}$$

2- Congruence. Pour chaque $f \in \sum_n, n \in \mathrm{N}$

$$\frac{[t] \rightarrow [t'] \ldots [t_n] \rightarrow [t'_n]}{[f(t_1,\ldots t_n)] \rightarrow [f(t'_1,\ldots t'_n)]}.$$

3- Remplacement. Pour chaque règle de réécriture $r : [t(x_1,\ldots,x_n)] \rightarrow [t'(x_1,\ldots,x_n)]$ dans R,

$$\frac{[w_1] \rightarrow [w_1'] \quad \ldots \quad [w_n] \rightarrow [w_n']}{[t(\bar{w}/\bar{x})] \rightarrow [t'(\bar{w}'/\bar{x})]}$$

Tel que $t(\bar{x}/\bar{w})$ indique la substitution simultanée des w_i pour x_i dans t.

4- Transitivité.

$$\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Une logique équationnelle (modulo un ensemble d'axiomes E) est obtenue en ajoutant la règle suivante :

5- Symétrie. $\frac{[t_1] \rightarrow [t_2]}{[t_2] \rightarrow [t_1]}$, dans ce cas on adopte la notation $[t] \leftrightarrow [t']$ est appeler cette séquence bidirectionnelle équation.

Définition 8. Étant donnée une théorie de réécriture $\mathfrak{R}$= (Σ, E, L, R), le séquent [*t*] → [*t'*] est dit $\mathfrak{R}$-réécriture concurrente Ssi, il peut être dérivé à partir de $\mathfrak{R}$ par une application finie des règles 1-4 [55].

4.3.1 Réseau de Petri dans la logique de réécriture

La logique de réécriture est essentiellement une logique de changement dans laquelle la déduction correspond directement aux changements [54]. Cette logique comme déjà mentionner peut naturellement exprimer et unifier divers modèles de concurrence. Les réseaux de Petri représentent un exemple simple mais assez riche des systèmes présentant des changements concurrents. Généralement, un RdP est présenté comme un ensemble de places, un ensemble disjoint de transitions et une relation entre eux qui associe à chaque transition l'ensemble des ressources consommées ainsi que ceux produit par son franchissement.

Cette représentation a été proposée dans un Framework algébrique [54]. Dans ce contexte les ressources sont représentées par des multisets de places, et donc nous avons une opération binaire (union ensembliste dénotée $\otimes$) qui est associative, commutative, est qui a le multiset vide comme élément identité. Un réseau de Petri est vu donc, comme un graphe où les arcs sont des transitions et les nœuds sont des multisets sur l'ensemble des places nommés généralement marquage. L'exemple suivant (figure 10) montre un RdP qui modélise une machine concurrente pour vendre des tartes et des pommes.

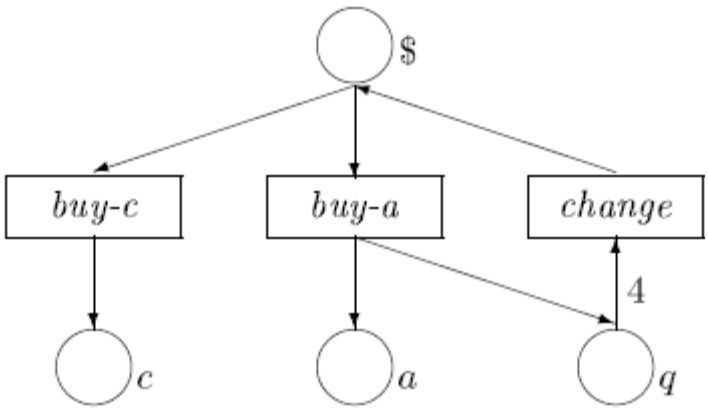


Fig. 10 RdP d'une machine vendeuse de tartes et de pommes [54]

Une tarte coûte un dollar et une pomme coûte trois quarts de dollar. L'achat d'une pomme (*buy-a*) par exemple provoque le dépôt d'un jeton a dans la place *a* (une pomme) et un quart de dollar dans la place *q* (un quart de dollars). Par défaut de fabrication la machine n'accepte que des pièces d'un dollar que l'utilisateur doit déposer pour acheter une pomme. Ainsi en ayant 4 quarts de dollar dans la place *q* cela provoque leurs changement (*change*) en un dollar dans la place $ (une pièce de 1 dollars) pour surmonter le problème.

Les réseaux de Petri ont une expression naturelle et simple comme théories de réécriture [56]. Leurs états distribués correspondent aux marquages des places représentés par des multisets finis.

Pour notre exemple, l'RdP peut être vu comme un graphe ayant les arcs suivants :

- *buy-c* : $\$ \rightarrow c$
- *buy-a* : $\$ \rightarrow a \otimes q$
- *change* : $q \otimes q \otimes q \otimes q \rightarrow \$$

On peut remarquer que la représentation de cet RdP en logique de réécriture est évidente. Une transition t est simplement une règle de réécriture étiquetée $t : M \rightarrow M'$ (1) entre deux marquages (M et M') de multiset. Ainsi, un RdP N peut être vu comme une théorie de réécriture N avec une axiomatisation algébrique (1) pour son marquage et avec une règle de réécriture par transition. Le franchissement d'une transition correspond alors à une réécriture modulo ACI par la règle de réécriture correspondante [56]. La machine modélisée dans notre exemple est concurrente, car plusieurs règles de réécriture peuvent être appliquées en parallèle suite au fait d'appuyer en même temps sur plusieurs boutons.

La logique de réécriture fournit une sémantique complète pour exprimer l'aspect dynamique d'un RdP, et réponds parfaitement sur les questions concernant les états que ce dernier peut atteindre. Cela a été précisé comme suit [55] :

Etant donné un RdP N avec un ensemble de place S et un ensemble de transition T, N' est la théorie de réécriture qui représente cet RdP, et étant donné M et M' des marquages sur S. Alors, le marquage M' est atteignable depuis M dans l'RdP N Ssi la réécriture $M \rightarrow M'$ est une séquence dérivable de la théorie N' en utilisant les règles de déduction de la logique de réécriture.

La théorie de réécriture correspondante a l'RdP de notre exemple peut être exprimer en langage Maude comme suit :

```
mod PETRI-MACHINE is
    sort Marking .
    ops null $ c a q : -> Marking [ctor].
    ops _ _ : Marking Marking  -> Marking [ctor assoc comm id: null].
    rl [buy-c] : $ => c.
    rl [buy-a] : $ => a q.
    rl [chng] : q q q q  => $.
endm
```

Pour comprendre l'exemple ci-dessus, nous allons essayer de présenter plus de détail sur ce langage et montré son intérêt comme étant un langage formel basé sur la logique de réécriture.

4.3.2 Maude

Le langage Maude est un langage de programmation formel et déclaratif basé sur la théorie mathématique de la logique de réécriture [47]. Il a été développé ainsi que la logique de réécriture par 'José Meseguer' et son groupe dans le laboratoire d'informatique en SRI International. C'est un langage de haut niveau et un système de haute performance qui supporte deux types de calcul celui de la logique équationnelle et de la logique de réécriture pour une large gamme d'applications [57].

Le langage Maude spécifie des théories de la logique de réécriture. Les types de données sont définies algébriquement par des équations et le comportement dynamique d'un système est défini par des règles de réécriture qui décrivent comment une partie d'un état du système peut changer on un seul pas.

Maude est un langage de programmation très puissant qui modélise les systèmes et les actions à l'intérieur de ces systèmes. Il peut presque tout modéliser, de l'ensemble des nombres rationnels aux systèmes biologiques, ou toute chose décrite avec le langage humain peut être exprimée par un code Maude.

Maude a été influencé de manière importante par le langage OBJ3[15], il contient ce dernier comme sous langage de la logique équationnelle. Il se diffère de OBJ3 sur le niveau équationnel par une meilleure performance et une logique équationnelle plus riche, à savoir, la logique équationnelle d'appartenance[16] [57].

Maude et un interpréteur de haute performance, il exécute des programmes équationnels écrit en Maude en commençant par une expression initiale et en appliquant les équations "de gauche à droite" jusqu'à ce que aucune équation ne peut être appliquée. L'interpréteur exécute des programmes de réécriture par une application "arbitraire" des règles de réécriture (aussi de gauche à droite) sur l'expression ou l'état initial jusqu'à ce qu'aucune règle ne soit applicable, ou une limite supérieure sur le nombre de réécriture donnée par l'utilisateur a été atteinte [47].

Une théorie de réécriture est souvent non déterministe et peut exhiber différents comportements. La commande **rewrite** de Maude peut être utilisée pour exécuter l'un de ces comportements à partir d'un état initial. Pour analyser tous les comportements possibles à partir d'un état initial donné, on peut utiliser la commande **search** de haute performance.

Le module est le concept clé de Maude. Il s'agit essentiellement d'un ensemble de définitions. Il définit un ensemble d'opérations et comment elles interagissent, ou, mathématiquement parlant, ils définissent une algèbre. Une algèbre est un ensemble d'ensembles et les opérations sur eux. En Maude, un module fournira une collection de sortes et une collection d'opérations sur ces sortes, ainsi que les informations nécessaires pour réduire et de réécrire des expressions entrées par l'utilisateur dans l'environnement Maude.

Il y a trois types de module dans Maude pour la spécification des systèmes [55] :

1. Modules fonctionnels, introduit par le mot clé 'fmod',
2. Modules systèmes, introduit par le mot clé 'mod', et
3. Modules orientés objet, introduit par le mot clé 'omod'.

Nous allons essayer de présenter brièvement chaque module et cela dans le cadre de son utilisation et son application dans ce mémoire, pour la vérification des propriétés comportementales des logiciels à base de composants.

4.3.2.1 Modules fonctionnels

Les modules fonctionnels sont des théories équationnelles. Ces derniers définissent des types de données et des opérations applicables sur elles. Les types de données consistent en des éléments qui peuvent être nommés par 'termes clos' (ground terms) qui

[15] C'est un langage de programmation et de spécification algébrique de la famille OBJ basé sur une logique équationnelle de sorte ordonnné (Order sorted equational logic).
[16] En anglais 'membership equational logic'.

sont des termes sans variables [58]. Deux termes clos désignent le même élément si et seulement si elles appartiennent à la même classe d'équivalence tel que déterminé par les équations. Les équations sont considérées comme des règles de simplification orientées et sont utilisées seulement de gauche à droite.

Le calcul dans un module fonctionnel est accompli en utilisant les équations comme règles de réécriture jusqu'à ce qu'une forme canonique soit trouvée [57]. Le résultat du calcul (forme canonique) est le même quelque soit l'ordre dans lequel les équations sont appliquées. Les équations sont déclarées en utilisant les mots clés **eq** ou **ceq** (pour les équations conditionnelles). Maude supporte des simplifications équationnelles modulo n'importe quel combinaison des opérateurs d'associativité (spécifié par le mot clé : *assoc*) et/ou de commutativité (le mot clé : *comm*) et qui peut avoir aussi un élément identité (le mot clé : *id :*). Ce qui veut dire que la simplification a lieu pas seulement entre les termes, mais entre les classes d'équivalence de termes modulo de tels axiomes équationnels [59].

La logique équationnelle sur laquelle les modules fonctionnels de Maude sont basés est une extension de la logique équationnelle de sorte ordonné nommée logique équationnelle d'appartenance (membership equational logic). Ainsi, les modules fonctionnels supportent la déclaration de sortes (types) multiples, la relation entre les sous sortes (subsorts), et la surcharge des opérateurs [58]. Les axiomes dans cette logique peuvent être conditionnels ou inconditionnels.

Un module fonctionnel est de la forme : fmod ε endfm, où ε représente une théorie équationnelle. L'exemple suivant illustre un module fonctionnel d'un type abstrait de donnée celui des naturels :

```
fmod BASIC-NAT is                  *** BASIC-NAT est le nom du module
sort Nat .                          *** une sorte (type) Nat
op 0 : -> Nat [ctor] .              ***  une opération
op s_ : Nat -> Nat [ctor] .        ***le mot clé ctor désigne un constructeur
op _+_ : Nat Nat -> Nat .           *** opération d'addition sur deux naturels
op max : Nat Nat -> Nat .
vars N M : Nat .                    *** déclaration de deux variables de type Nat
eq s M + N = s (M + N) .            *** une équation
eq 0 + N = N .              *** N + 0 et N dénotent le même terme (même classe d'équivalence) qui est [N]
eq max(0, M) = M .
eq max(N, 0) = N .
eq max(s N, s M) = s max(N, M) .
endfm
```

L'évaluation des expressions dans les modules fonctionnels se fait à l'aide de la commande **reduce** (en abréger **'red'**).

4.3.2.2 Modules systèmes

Un module système spécifie une théorie de réécriture. De la même manière qu'un module fonctionnel à principalement la forme, fmod (Σ, E) endfm, tel que (Σ, E) une théorie de la logique équationnelle d'appartenance, un module système à essentiellement la forme mod (Σ, E, R) endm, tel que (Σ, E, R) est une théorie de réécriture [59]. Une théorie de réécriture inclut des sortes et des opérateurs et peut avoir trois types d'instructions : les équations, l'appartenance, et les règles de réécriture, qui peuvent tous être conditionnel [58].

Une règle de réécriture conditionnelle est spécifiée en Maude avec la syntaxe :

crl [l] : *t* => *t*' if *cond*, de même, une règle de réécriture inconditionnelle est spécifiée comme suit : rl [l] : *t* => *t*'.

Sémantiquement, les règles de réécriture expriment le comportement dynamique d'un système. Une règle spécifie une 'transition concurrente locale' qui peut se produire dans un système si la partie gauche de la règle correspond à un fragment de l'état du système avec la condition étant valide. Dans un module système ou orienté objet les équations sont déclarées par les mots clé **ax** ou **cax**. Un module système est de la forme :

mod <ModuleName> is <DeclarationsAndStatements> endm

Revenant maintenant à notre exemple présenté dans la section 4.3.1, qui modélise la machine vendeuse de pommes et de tartes pour quelques explications.

```
mod PETRI-MACHINE is
    sort Marking .
    ops null $ c a q : -> Marking [ctor].
    ops _ _ : Marking Marking  -> Marking [ctor assoc comm id: null].
    rl [buy-c] : $ => c.
    rl [buy-a] : $ => a q.
    rl [chng] : q q q q  => $.
endm
```

La spécification de cette machine est donnée par le type (sort) 'Marking' représentant le marquage dans un RdP. Les opérations sur ces marquages sont définit à l'aide du mot clé *op* ou *ops* (pour plusieurs opérations). Chaque place dans un RdP constitue un marquage, cela est définit par le constructeur *ctor* indiquant que c, a, $, q, et null sont des marquages de places. Les mots clés *assoc* et *comm* signifient les opérations d'associativité et de commutativité entre deux multisets de marquages, ayant null comme éléments identité.

La dynamique de ce système est exprimée en termes de règles de réécriture montrant les changements élémentaires se produisant durant son exécution. L'exécution d'un module système Maude se fait par la commande **rewrite** (**rew**).

4.3.2.3 Modules orientés objets

Il y a le 'Core Maude' qui contient principalement les modules systèmes et le modules fonctionnels, et le 'Full Maude' qui est une extension du 'Core Maude' avec les modules orientés objets pour permettre la définition des systèmes concurrents orientés objets.

Dans les systèmes concurrents orientés objets l'état concurrent, qui est appelé habituellement 'une configuration', a typiquement la structure d'un multiset constitué d'objets et de messages. Ces derniers évoluent par réécriture concurrente modulo l'associativité, commutativité, et l'élément identité en utilisant des règles décrivant les effets des événements de communication entre les objets et les messages [57]. Tout les modules orientés objets incluent un module nommé 'CONFIGURATION' qui définit les concepts de base des systèmes à objets concurrents. Ce dernier inclus la déclaration de sortes suivantes :

- Oid : identificateurs d'objets,
- Cid : identificateurs de classes,
- Objsect : pour les objets, et
- Msg : pour les messages.

L'état d'un objet est représenté dans Maude par un terme $<O : C \mid a_1 : v_1, \ldots, a_n : v_n>$, tel que O est le nom de l'objet ou l'identificateur, C est l'identificateur de sa classe, les a_i sont les noms des identificateurs des attributs de l'objet et les v_i leurs valeurs correspondantes. Les classes sont définies par le mot clé 'class' suivit du nom de la classe C, et d'une liste d'attributs séparés par des virgules. Chaque attribut est de la forme $a : S$, a est un attribut, S est la sorte (type) des valeurs de l'attribut ; $C \mid a_1 : S, \ldots, a_n : S_n$. Les règles de réécriture dans un module orienté objet nous spécifient le comportement associé avec les messages. L'exemple suivant illustre un module orienté objet modélisant des comptes bancaires :

```
(omod ACCOUNT is
protecting QID . *** importation du module 'quoted identifiers'
protecting INT . *** importation du module des entiers.
subsort Qid < Oid .  *** déclarer Qid comme sous sorte de Oid
class Account | bal : Int . *** déclaration d'une classe Account avec un attribute de balance'bal' de typr int.
msgs credit debit : Oid Int -> Msg .  déclarer deux messages 'credit' et 'debit' de type Oid et Int respectivement.
msg from_to_transfer_ : Oid Oid Int -> Msg .
vars A B : Oid .
var M : Nat .
vars N N' : Int .
```

```
rl [credit] :
credit(A, M)
< A : Account | bal : N >
=> < A : Account | bal : N + M > . *** une règle de réécriture inconditionnelle
crl [debit] :
debit(A, M)
< A : Account | bal : N > => < A : Account | bal : N - M > if N >= M . *** une règle de réécriture conditionnelle
crl [transfer] :
(from A to B transfer M)
< A : Account | bal : N > < B : Account | bal : N' >
=> < A : Account | bal : N - M > < B : Account | bal : N' + M > if N >= M .
endom)
```

Nous pouvons maintenant réécrire une simple configuration composée d'un compte et un message comme suit:

```
Maude> (rew < 'A-06238 : Account | bal : 2000 >
debit('A-06238, 1000) .)
result Object :
< 'A-06238 : Account | bal : 1000 >
```

Si on utilise le Full Maude, tel est notre cas, on met le tout entre parenthèses.

5. ECATNets

Les réseaux de Petri sont des modèles de concurrence largement utilisés. Ce modèle est attractif d'un point de vue théorique, en raison de sa simplicité attrayante est de sa nature intrinsèquement concurrente [60]. L'interprétation d'un réseau de Petri simple dans la logique de réécriture a été donnée plus haut, où pour chaque transition on associe la règle de réécriture appropriée, et deuxièmement le marquage dans le réseau est représenté par un terme composé de places (en utilisant l'opérateur $\otimes$) contenant des jetons.

La logique de réécriture offre un Framework naturel pour donner une sémantique a différents types de réseaux de Petri algébrique [56]. Plusieurs propositions ont été introduite dans ce contexte, parmi elles dont nous nous somme intéressés dans ce mémoire pour la modélisation des logiciels à base de composants sont les ECATNets (Extended Concurrent Algebraic Term Nets). Ces derniers constituent une catégorie des réseaux de pétri algébrique de hauts niveaux basés sur une combinaison saine des réseaux de Petri de haut niveau et des types abstraits algébriques.

Les ECATNets ont été proposés comme une extension des CATNets (Concurrent Algebraic Term Nets) initialement introduits et définis par Bettaz et Maouche dans [48]. Ils combinent (ECATNets) la force des RdPs avec celle des types abstraits de données. Les

réseaux de Petri sont utilisés pour leur puissance d'exprimer la concurrence et leurs dynamiques, alors que les types abstraits de données sont motivés pour leurs pouvoirs d'abstraction de données et leurs bases mathématiques solides [61]. La sémantique des ECATNets est donnée en termes de la logique de réécriture, permettant ainsi d'étudier et de spécifier formellement le comportement des systèmes modélisés.

La logique de réécriture dans ces modèles consiste en un ensemble d'axiomes et un ensemble de règles de déduction [61]. Les axiomes sont des règles de réécriture décrivant l'effet des transitions comme étant des types élémentaires de changement d'état. Les règles de déduction permettent de 'propager' l'effet des changements élémentaires sur l'état global du système. Sous ces hypothèses le comportement d'un système (défini en termes de changements d'état) peut être décrit par une preuve (en utilisant la logique en question) qui nous permet de déduire un état à partir d'un autre [62].

La définition de base des ECATNets est fondée sur celle de leurs prédécesseurs les CATNets définit pour la première fois dans [48], où on peut trouvé plus de détails. Nous allons présenter dans la section suivante la définition formelle des ECATNets.

5.1 Définition formelle d'un ECATNets

Etant donné CATNas(X) la structure syntaxique de CATNet qui constitue une structure algébrique (pour plus de détails voir [48]). La structure syntaxique d'un ECATNet dénoté $CATNas(X)^+$ est définit par induction comme suit [61] :

- $CATNas(X) \subset CATNas(X)^+$
- Empty (L'élément vide)$\in CATNas(X)^+$
- Si $[m]_{\oplus} \in CATNas(X)$ alors $\tilde{}[m]_{\oplus} \in CATNas(X)^+$

Dans la suite, CATNas(X) sera appelé la structure syntaxique d'un ECATNet.

Un ECATNet est un réseau de Petri de haut niveau d'une structure [61] :

(P, T, s, IC, DT, CT, C, TC) où,

- P est l'ensemble de places et T est l'ensemble de transitions ;
- $s : P \rightarrow S$ est une fonction qui associe un sorte a chaque place ;
- $IC : (P \times T) \rightarrow CATNas(X)^+$ est une fonction partielle tel que pour tout $(p,t) \in$ domaine(IC), si $IC(p, t) \in CATNas(X)$ alors $IC(p, t) \in CATNas(X)_{s(p)}$ sinon, si $IC(p, t) \neq$ empty alors $\exists [m]_{\oplus} \in CATNas(X)_{s(p)}$ tel que $IC(p, t) = \tilde{}[m]_{\oplus}$;
- $DT : (P \times T) \rightarrow CATNas(X)$ est une fonction tel que pour tout $(p, t) \in (P \times T)$, $DT(p, t) \in CATNas(X)_{s(p)}$;
- $CT : (P \times T) \rightarrow CATNas(X)$ est une fonction tel que pour tout $(p, t) \in (P \times T)$, $CT(p, t) \in CATNas(X)_{s(p)}$;

- C : P → CATNas(Ø) est une fonction partielle tel que pour tout p∈ domaine(C), C(p) ∈ CATNas(Ø) ; si une place p n'appartiens pas a domaine(C) elle est considérée comme place avec une capacité infinie.
- TC : T → $CATNas(X)_{bool}$ est une fonction tel que pour tout t∈T, TC(t)∈ $CATNas(X(t))_{bool}$ où X(t) est l'ensemble de variable qui se produisent dans IC(p,t), DT(p, t) et CT(p, t) pour tout p∈P.

5.2 Syntaxe des ECATNets

Les ECATNets permettent la construction de modèles hautement compacte grâce à leurs notations syntaxique puissante. D'un point de vue syntaxique, la différence entre les RdPs ordinaire et les ECATNets est que les places et les arcs sont inscrits par des multisets (multi-ensembles) de termes algébriques.

Un ECATNet est une paire $\varepsilon = (Spec, N)$ où $Spec = (\Sigma, E)$ est une spécification algébrique d'un type abstrait de donnée, et *N* est un réseau de Petri dans lequel les marquages des places sont des multisets de Σ-*terms* [63]. Le marquage d'une place p par un ensemble de jetons est dénoté M(p). Le marquage d'un ECATNets est donc des multisets de termes algébriques spécifiés par l'utilisateur. La représentation graphique d'un ECATNet est donnée par la figure 11 :

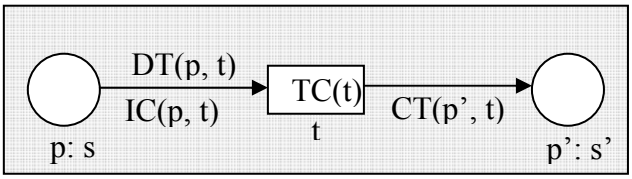


Fig. 11 Représentation générique d'un ECATNet

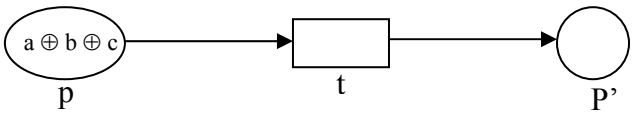
On distingue entre trois types de multisets utilisés pour inscrire les arcs dans un ECATNet qui sont IC, DT et CT. Les arcs entrants de chaque transition t i.e. (p, t) sont étiquetés par deux inscriptions IC(p, t) (Input Conditions) et DT(p, t) (Destroyed Tockens). Les arcs sortants de chaque transition t i.e. (t, p') sont étiquetés par CT(p', t) (Created Tockens). Finalement, chaque transition t peut être étiquetée par TC(t) (Transition Condition) qui prend comme valeur par défaut le terme 'true' (quand TC(t) est omis).

IC(p, t) spécifie la condition de franchissement de la transition t, DT(p, t) spécifie les jetons (un multiset) qui doivent être enlever depuis p quand t est franchie. Il est évident que le multiset DT(p, t) doit être inclut dans le marquage de la place p (i.e. DT(p, t) ⊆ IC(p, t)). IC(p, t) ou DT(p, t) sont omis dans la représentation graphique des ECATNets lorsque IC(p, t) = DT(p, t). CT(t, p') spécifie les jetons qui doivent être ajoutés a 'p' quand t est franchie, et finalement TC(t) représente un terme booléen qui spécifie une condition de franchissement additionnelle pour la transition t, elle précise quelques contraintes sur les valeurs prises par les variables locales de t.

Une caractéristique importante des ECATNets est qu'ils font la distinction entre la condition de franchissement et les jetons qui devrons être enlevés durant le franchissement de la transition t (représentés respectivement par IC(p, t) et DT(p, t)), alors que dans plusieurs type de réseaux de Petri de haut niveau les jetons (multiset) a enlever sont exactement les jetons qui satisfassent la condition d'entrée [63].

L'état courant d'un ECATNet représenté par un marquage est donné par l'union des termes ayant la forme (p, M(p)) [60].

L'exemple suivant montre l'état distribué d'un ECATNet représenté par une transition, une place d'entrée p, et une place de sortie vide p'.



s= (p, a ⊕ b ⊕ c) ⊗ (p', Ø) où ⊗ dénote l'union sur les multisets, et ⊕ dénote l'opération interne sur les termes considérés comme singleton (a ⊕ b ⊕ c). Le symbole ⊗ est distributif par rapport à l'opérateur ⊕.

5.3 Sémantique des ECATNets

Le comportement d'un ECATNet comme mentionner plus haut est donnée en terme de la logique de réécriture. Elle consiste dans ce contexte en un ensemble de règles de réécritures qui constituent les axiomes, et un ensemble de règles de déduction. Le comportement de ce type de réseau tel est le cas pour les autres types aussi, est traduit par un changement d'état dû au franchissement successif des transitions qui font passé le système d'un état à un autre.

Une transition dans un ECATNet est franchissable a n'importe quel moment si plusieurs conditions sont satisfaites simultanément [61] :

- La première est que tout IC(p, t) pour chaque place d'entrée p est satisfaite.
- La deuxième est que TC(t) est vraie.
- Et enfin, l'addition de CT(t, p') à chaque place de sortie p' ne doit pas avoir comme conséquence d'excéder sa capacité quand cette capacité est finie.

Lors du franchissement d'une transition t, DT (p, t) est enlevé totalement (cas positif) de la place d'entrée p et simultanément CT(t, p') est ajouté a la place de sortie p'. Notons que dans le cas non positif, on enlève les éléments en commun entre DT(p, t) et M(p). Le franchissement d'une transition et les conditions de ce franchissement sont formellement exprimés par des règles de la logique de réécriture. Les axiomes sont en réalité des règles de réécriture conditionnelles décrivant les effets des transitions comme des types élémentaires de changement. Les règles de déduction nous permettent d'avoir des conclusions valides des ECATNets à partir des changements.

Une règles de réécriture est de la forme ‘‘t: u → v if boolexp’’; où u et v sont respectivement les côtés gauche et droit de la règle, t est la transition liée à cette règle, et ‘‘boolexp’’ est un terme booléen. Plus précisément, u et v sont des multisets des pairs de la forme (p, $[m]_{\oplus}$), telle que p est une place du réseau, $[m]_{\oplus}$ est un multiset des termes algébriques. L'union des multiset sur les pairs (p, $[m]_{\oplus}$) est dénoté par ⊗. $[x]_{\otimes}$ dénote la classe d'équivalence de x, selon les axiomes ACI (Associativity Commutativity Identity = ϕ_B) pour ⊗ [60]. Nous allons présenter les règles de réécriture (les métarègles) associées aux ECATNets, qui définissent et capture le comportement concurrent et distribué de ces derniers. La forme de ces règles dépends fortement sur la relation entre IC(p, t) et DT(p, t).la forme générale des règles de réécriture est la suivante [62]:

- **IC(p,t) est de la forme $[m]_{\oplus}$**

Cas 1. $[IC(p,t)]_{\oplus} = [DT(p,t)]_{\oplus}$

C'est la relation la plus simple est corresponds aux cas habituellement traités dans les RdPs simple ainsi que dans les CATNets (prédécesseur des ECATNets).

La métarègle a La forme suivante : t : (p, $[IC(p, t)]_{\oplus}$) → (p', $[CT(t, p')]_{\oplus}$) **(R1 : règle1)**.

Cas 2. $[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} = \phi_M$

Cette situation correspond à vérifier que IC(p, t) est inclus dans M(p) et, retirer DT(p, t) de M(p). Il est clair qu'il n'y a pas d'élément en commun entre IC(p, t) et DT(p, t) et que IC(p, t) est juste utilisé pour jouer le rôle d'un jeton déclencheur permettant le franchissement de la transition en question.

La forme de la règle est : t : (p, $[IC(p, t)]_{\oplus}$) ⊗ (p, $[DT(p, t)]_{\oplus}$) → (p, $[IC(p, t)]_{\oplus}$) ⊗ (p', $[CT(t, p')]_{\oplus}$) **(R2)**.

Cette forme simple et inconditionnelle peut conduire à une description non correcte de la nature du parallélisme désiré par le spécifieur, l'indéterminisme ou le vrai parallélisme [64]. Un cas de figure a été discuté dans [64] montrant une situation où on veut exprimer le vrai parallélisme mais qui ne peut être représenté avec la forme simple de la règle, il s'agit d'une place p partagée comme place d'entrée de deux transitions t_1 et t_2, et ayant chacune une place de sortie. Une deuxième forme a été proposée, pour montrer une sémantique de vrai parallélisme au lieu d'une sémantique d'entrelacement. Dans ce derniers cas le tirage d'une transition est non déterministe. La forme de la règle est la suivante :

(p, $[DT(p, t)]_{\oplus}$) → (p', $[CT(t, p')]_{\oplus}$) if (p, $[IC(p, t)]_{\oplus}$) ∩ $[M(p)]_{\oplus}$ → (p, $[IC(p, t)]_{\oplus}$) **(R2')**

Cette règle montre que le tirage d'une transition est conditionner par la présence de IC(p,t) et se traduit par le retrait de DT(p, t) de p suivit du dépôt de CT(t, p')dans p'.

Cas 3. $[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} \neq \phi_M$ **(R3)**

Cette situation correspond au cas le plus général. Elle peut cependant être résolue d'une manière élégante en remarquant qu'elle pourrait être apportée aux deux cas précédents. Ceci est réalisé en remplaçant la transition t de ce cas par deux transitions t1 et t2. Ces transitions, une fois elles sont franchises concurremment, donnent le même effet global que notre transition. Nous éclatons IC(p, t) en deux multisets IC1(p, t1) et IC1(p, t2). Nous éclatons également DT(p, t) en deux multisets DT1(p, t1) et DT1(p, t2) :

$IC(p, t) = IC1(p, t1) \cup IC1(p, t2)$, $DT(p, t) = DT1(p, t1) \cup DT1(p, t2)$.

Les quatre multisets obtenus doivent réaliser $IC1(p, t1) = DT1(p, t1)$ et $IC2(p, t2) \cap DT2(p, t2) = \phi M$. Le franchissement de la transition t est identique au franchissement en parallèle des deux transitions t1 et t2.

- **IC(p, t) est de la forme $\tilde{}[m]_{\oplus}$**

Dans quelques situations on s'intéresse au franchissement d'une transition lorsque sa place d'entrée ne contient pas un multiset de jetons précis **M**. Dans ce cas, on utilise la notation $\tilde{}$**M** a la place du multiset IC(p, t). La forme de la règle correspondante est:

$t : (p, [DT(p, t)]_{\oplus} \cap [M(p)]_{\oplus}) \rightarrow (p', [CT(t, p')]_{\oplus})$ if $([IC(p, t)]_{\oplus} \setminus ([IC(p, t)]_{\oplus} \cap [M(p)]_{\oplus})) = \phi M \rightarrow$ [false]. **(R4)**.

La partie conditionnelle traduise le fait que la transition t est franchissable si le marquage de sa place d'entrée ne contient aucun jeton appartenant à **M**.

- **IC(p, t) = empty**

Dans d'autres situations on s'intéresse par le franchissement d'une transition lorsque sa place d'entrée est vide. On utilise dans ce cas la notation **empty** a la place du multiset IC(p, t). La forme de la règle correspondante est:

$t : (p, [DT(p, t)]_{\oplus} \cap [M(p)]_{\oplus}) \rightarrow (p', [CT(t, p')]_{\oplus})$ if $[M(p)]_{\oplus} \rightarrow \phi_M$ **(R5)**.

Si la capacité de la place de sortie C(p) est finie, la partie conditionnelle de la règle de réécriture doit inclure le composant suivant :

$[CT(p, t)]_{\oplus} \oplus [M(p)]_{\oplus} \cap [C(p)]_{\oplus} \rightarrow [CT(p, t)]_{\oplus} \oplus [M(p)]_{\oplus}$ (Cap).

Dans le cas où il y a une condition de transition TC(t), chaque métarègle doit contenir dans sa partie conditionnelle l'expression conditionnelle : 'and TC(t) $\rightarrow$ true' **(R6)**.

La forme d'une règle qui correspond à une transition avec une place d'entrée et plusieurs places de sorties est obtenue par l'union de tous les multisets CT(t, p') des arcs sortants [64]. La situation où une transition a plusieurs places d'entrées peut être considérée comme la composition de l'application des règles sur tous les arcs de bases.

Nous allons maintenant citer l'ensemble des règles de déduction définit dans les ECATNets. Soit R la théorie de réécriture d'un ECATNet, on dit que la séquence s→ s' est prouvable dans R, où (s, s') sont deux état différents, Ssi s → s' peut être obtenu par des applications finies et concurrentes des règles de déduction suivantes : Réflexivité (Reflexivity), Congruence (Congruence), Remplacement (Replacement) [55]; Décomposition (Splitting), Recombinaison (Recombination) et l'Identité (Identity) [48].

Les trois premières règles ont été déjà abordées dans la logique de réécriture, les trois dernières règles sont inhérentes aux ECATNets :

- **Décomposition:** $\forall p \in P$, $\forall [mi]$ et $[M]$ des éléments de type CATNas(X),

$$\frac{1 \leq i \leq k, [M] = \bigoplus_{i=1}^{k} [m_i]}{[\langle p, [M] \rangle] \rightarrow [\bigoplus_{i=1}^{k} \langle p_i, [M_i] \rangle]}$$

Exemple : (p, a ⊕ b) = (p, a) ⊗ (p, b)

- **Recombinaison**

$\forall p \in P$, $\forall [m]$ et $[m']$ des éléments de type CATNas(X),

$$\frac{}{[\langle p, [m] \rangle] \otimes [\langle p, [m'] \rangle] \rightarrow [\langle p, [m] \oplus [m'] \rangle]}$$

Exemple : (p, a) ⊗ (p, b)= (p, a ⊕ b)

- **Identité**

$\forall p \in P$, $\dfrac{}{(p, \phi_M) = \phi_B}$

Nous rappelons que la règle de réflexivité décrit que chaque état se transforme en lui-même. La règle de congruence que les changements élémentaires doivent être correctement propagés. La règle de remplacement est utilisée quand les instanciations de variables deviennent nécessaires. Les règles de décomposition et de recombinaison nous permettent, en décomposant et recombinant "judicieusement" les multi-ensembles des classes d'équivalences des termes, pour détecter les calculs montrant un maximum de concurrence. La règle d'identité permet de lier ϕ_M (l'élément identité de ⊕) avec ϕ_B (l'élément identité de ⊗).

La logique de réécriture, i.e., la théorie de réécriture et les règles de déduction, signifie que le comportement d'un ECATNet est donné par un système de déduction, qui pour un état donné 's', il va cherché tout les changements de base possible (application des axiomes) est essaye de les composés (application des règles de déduction) de manière à atteindre l'état suivant s', qui correspond au calcul concurrent prévu par le réseau [48]. Un exemple détaillé peut être trouvé dans [65].

5.4 Exemples de modélisation avec les ECATNets

Nous allons essayer à travers un exemple illustratif de montrer l'utilisation du formalisme des ECATNets dans la modélisation des systèmes réels. L'exemple suivant, a été modélisé en utilisant les ECATNets dans [64].

5.4.1 Présentation de l'exemple 1

L'exemple est à propos d'une cellule de production qui fabrique des pièces forgées de métal à l'aide d'une pression. Cette cellule se compose de table A qui sert à alimenter la cellule en pièces brutes, d'un robot de manutention, d'une presse et d'une table B qui sert au stockage des pièces forgées. Le robot inclut deux bras, disposés perpendiculairement sur un même plan horizontal, interdépendant d'un même axe de rotation et sans possibilité verticale de mobilité. La figure 12 représente la disposition spatiale des éléments de la cellule. Le robot peut prendre une pièce brute de la table A et la déposer dans la presse à l'aide du bras 1. Il peut également prendre une pièce forgée de la presse et la déposer sur la table du stockage B à l'aide du bras 2. En bref, le robot peut faire deux mouvements de rotation. Le premier lui permet de passer de sa position initiale à sa position secondaire. Ce mouvement permet au robot de déposer une pièce brute dans la pression et probablement celui d'une pièce forgée sur la table du stockage B. Le second lui permet de passer de sa position secondaire vers sa position initiale et de poursuivre le cycle de la rotation.

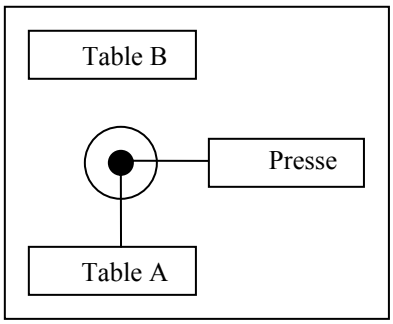


Fig. 12 Cellule de production [64]

- **Modèle ECATNet**

La figure 13 représente le modèle ECATNet de la cellule de production schématisée ci-dessus. Le symbole ϕ est utilisé pour dénoter le multiset (multi-ensemble) vide dans les inscriptions des arcs. Les multisets r sur les arcs dénote 'raw' (brute) et f dénote 'forge' (forgée). On a précédemment indiqué que ces derniers sont des spécifications algébriques

de type abstrait de données qui doivent être bien évidement définit par l'utilisateur. Si les inscriptions IC(p, t) et DT(p, t) sont identiques, alors nous présentons uniquement IC(p, t) sur l'arc (p, t).

- **Les places**

Ta : table A ; ensemble, potentiellement vide, des pièces brutes.

Tb : table B ; ensemble, potentiellement vide, des pièces forgées.

Ar1 : bras 1 du robot ; au plus une pièce brute.

Ar2 : bras 2 du robot ; au plus une pièce forgée.

Pr : presse ; au plus une pièce brute ou une pièce forgée.

PosI : position initiale du robot ; elle est marquée "ok"si elle est la position courante du robot.

PosS : position secondaire du robot ; elle est marquée "ok" si elle est la position courante du robot.

EA : cette place a été ajoutée pour tester si les deux bras du robot sont vides.

- **Les transitions**

T1 : prise d'une pièce brute par le bras 1 du robot.

T2 : prise d'une pièce forgée par le bras 2 du robot.

D1 : dépôt d'une pièce brute dans la presse.

D2 : dépôt d'une pièce forgée sur la table B.

TS1, **TS2** : rotation du robot de la position initiale vers la position secondaire.

TI : rotation du robot de la position secondaire vers la position initiale.

F : forge de la pièce brute introduite dans la presse.

E : dépôt d'une pièce brute sur la table A.

R : retire des pièces forgées de la table B.

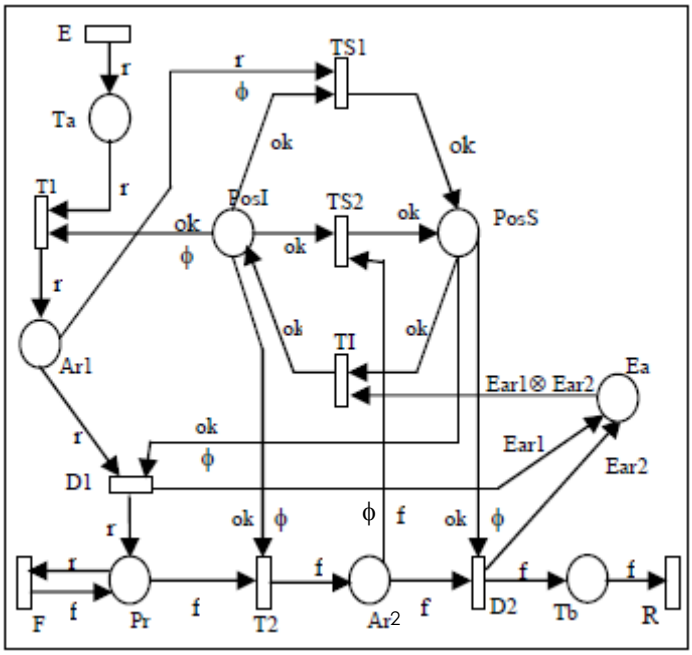


Fig. 13 Modèle ECATNet de la cellule de production [64]

- **Les règles de réécriture**

[T1]: appliquer **(R2')** % (Ta, raw) → (Ar1, raw) if (M(Pos-I) → ok)

and ((Ar1, raw) ⊗ M(Ar1) ∩ C(Ar1)) → (Ar1, raw) ⊗ M(Ar1) …..(1)

(1) ajoutée car la capacité de la place Ar1 est finie (une pièce brute au plus)

[T2]: appliquer **(R2')** % (Pr, forge) → (Ar2, forge) if (M(Pos-I) → ok)

and ((Ar2, raw) ⊗ M(Ar2) ∩ C(Ar2)) → (Ar2, raw) ⊗ M(Ar2)

[D1]: appliquer **(R2')** % (Ar1, raw) → (Pr, raw) ⊗ (Ea, Ear1) if (M(Pos-S) → ok)

[D2]: appliquer **(R2')** % (Ar2, forge) → (Tb, forge) ⊗ (Ea, Ear2) if (M(Pos-S) → ok))

[TS1]: (Pos-I, ok) → (Pos-S, ok) if (M(Ar1) → raw)

[TS2]: (Pos-I, ok) → (Pos-S, ok) if (M(Ar2) → forge)

[TI]: (Pos-S, ok) ⊗ (Ea, Ear1) ⊗ (Ea, Ear2)→ (Pos-I, ok) if (M(Pos-S) → ok)

[F] : (Pr, raw) → (Pr, forge)

[E] : ϕ → (Ta, raw)

[R] : (Tb, forge)→ ϕ

5.4.2 Présentation de l'exemple 2

L'exemple suivant [62] est tiré du domaine des protocoles de communication. Il consiste à modéliser le protocole Ethernet vu par une station émettrice. Cette dernière est composée de quatre modules : un module de construction de trames et début de transmission, un module qui a la fonction de transmission avec succès, le troisième traite la détection de collision, le dernier est relatif à la fonction de retransmission. Nous allons se limiter dans notre cas à présenter seulement le premier module. Expliquons maintenant le rôle de ce module. La construction de la trame commence lorsque un jeton '<d,s,data>'est déposé dans la place 'FROM_USER' (figure 14).

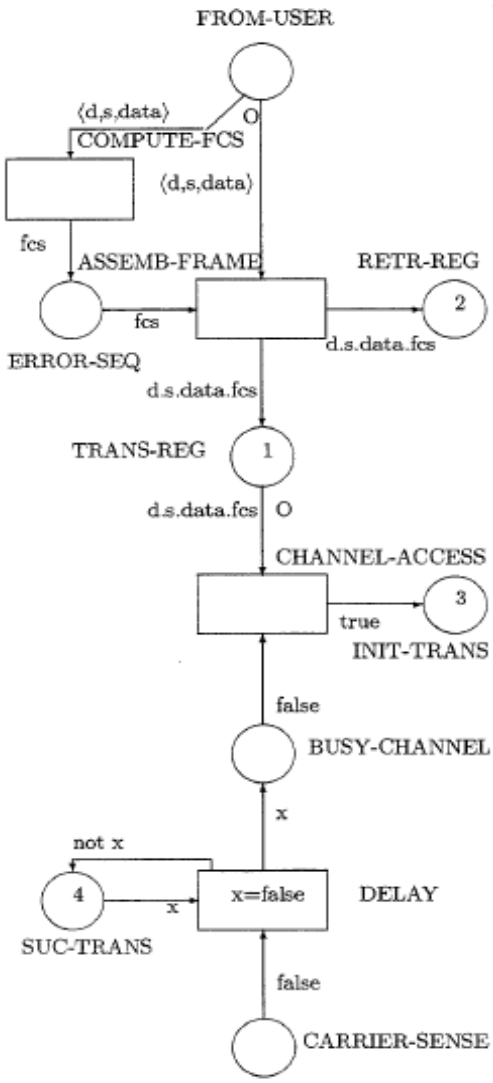


Fig. 14 Modèle ECATNet du module début de transmission d'une station émettrice Ethernet [62]

Ce jeton est considérer comme une primitive transférée depuis la couche utilisateur vers la sous couche MAC (Medium Access Control) demandant la transmission d'une valeur 'data' depuis la source 's' vers la destination 'd'. La place 'FROM_USER' est

considérée comme une interface entre les deux couches. Ensuite, un code de contrôle d'erreur sur la donnée est ajoutés à la trame fcs (Frame Check Sequence) 'd.s.data.fcs', qui sera composée dans un registre de transmission 'TRANS_REG'. Une copie de la trame est conservée dans un registre de retransmission 'RETR_REG'.

D'autre part, la sous couche MAC écoute le canal de transmission 'CARRIER_SENSE' pour éviter les collisions qui peuvent se produire avec la transmission en cours. La place 'CARRIER_SENSE' est une interface entre la sous couche MAC et la couche physique. Si le canal est libre (un jeton 'false' est présent dans la place 'CARRIER_SENSE') elle attend pendant une période de temps qui corresponds au parcours d'une trame entre les deux nœuds les plus éloignés dans le réseau. Ensuite, si la transmission a eut lieu avec succès (dépôt d'un jeton 'false' dans la place 'BUSY_CHANNEL', et un jeton 'true' dans la place 'SUC_TRANS') elle prend possession du canal 'CHANNEL_ACCESS' et la transmission commence (dépôt d'un jeton 'true' dans la place 'INIT_TRANS').

- **Les règles de réécriture**

COMPUTE_FCS : appliquer **(R2)**% (FROM_USER,<d,s,data>)→ (FROM_USER,<d,s,data>) ⊗ (ERROR_SEQ, fcs).

ASSEMB_FRAME : appliquer **(R1)**% (FROM_USER,<d,s,data>) ⊗ (ERROR_SEQ, fcs) → (TRANS_REG, d.s.data.fcs) ⊗ (RETR_REG, d.s.data.fcs).

CHANNEL_ACCESS: appliquer **(R2)** et **(R1)** % (TRANS_REG, d.s.data.fcs) ⊗ (BUSY_CHANNEL, false) → (INIT_TRANS, true) ⊗ (TRANS_REG, d.s.data.fcs).

DELAY: appliquer **(R1)** et **(R6)** % (CARRIER_SENSE, false) ⊗ (SUC_TRANS, x) → (BUSY_CHANNEL, x) ⊗ (SUC_TRANS, not(x)) if (x=false) → true.

6. Conclusion

Nous avons pu voir à travers ce chapitre les parties qui constituent les éléments de base de notre travail, il s'agit de proposer une approche de modélisation des logiciels à base de composants en utilisant les RdPs. On a commencé par un bref survol sur les réseaux de pétri dans lequel nous avons introduit ce formalisme. Les RdPs existent sur plusieurs formes, les HLPN en constituent l'une d'elles. Ils étaient introduits pour résoudre le problème d'explosion d'état, permettant ainsi une représentation plus compacte des systèmes complexes concurrent et distribués. On a présenté ensuite les RdPs algébriques (AHL nets) qui représente une variante spécifique des HLPN, ils combinent les réseaux de Petri et les spécifications algébriques. Ces derniers sont bien établis pour la spécification formelle de types abstraits de données et des systèmes logiciels. On a introduit par la suite la logique de réécriture qui constitue un Framework pour la spécification des systèmes distribués et concurrents ainsi que son langage Maude. Plusieurs modèles peuvent être

naturellement représentés dans cette logique, y compris les RdPs. Dans ce contexte, des efforts considérables ont été investit pour faire le couplage entre les RdPs algébrique et cette logique donnant naissance par exemple aux ECATNets. Notre choix a été porté dans ce mémoire sur ce formalisme, pour la modélisation des logiciels à base de composants. Nous allons expliquer nos motivations sur ce choix dans le prochain chapitre qui sera consacré à la présentation de notre approche.

CHAPITRE III :

Modélisation des logiciels à base de composants avec les ECATNets

1. Introduction

Dans ce chapitre nous allons présenter une approche formelle de modélisation des logiciels à base de composants en utilisant le formalisme des ECATNets. Les spécifications de composants logiciels utilisés dans les pratiques du développement logiciels aujourd'hui sont limitées aux spécifications syntaxiques. Ce type de spécification n'offre pas suffisamment d'informations sur la sémantique des opérations qui définissent les interfaces d'un composant, ce qui peut faciliter la compréhension du comportement de ces derniers. Pour pouvoir assurer cette sémantique et disposer de plus ample d'informations sur le comportement des composants, nous avons choisis l'alternative d'utiliser les méthodes formelles. Celles-ci, nous permettent de construire des modèles mathématiques ayant une sémantique bien définit, et rendent possible l'analyse et la vérification des systèmes en cours de spécification. Ceci peut aider à éviter les erreurs de composition qui peuvent mener assez souvent à des comportements bizarres d'une application à base de composant.

Une multitude de travaux ont été réalisés dans ce contexte. Nous présenterons quelques uns dans la deuxième section de ce chapitre. Dans le même contexte, et dans le cadre de travail de ce mémoire, nous nous somme proposés à contribué dans la discipline du CBSE et participer a la faire évoluer.

2. Travaux voisins

Peu de travaux en été consacré pour les fondements théoriques et formels dans le développement basé composants. Nous allons se limiter à présenter brièvement quelques travaux de formalisations des composants logiciels. Le premier [46] offre une manière solide pour spécifié un composant logiciel en se basant sur la théorie de la logique du premier ordre et les types abstraits de données (ADTs17). Les auteurs de ce travail ont proposés une approche de spécification basée sur la notion qu'ils ont nommée 'a priori correctness'. Celle-ci suppose que l'on peut prédire qu'un assemblage de composants est

[17] Abstract Data Types

correct (correctness) avant de faire la composition, i.e. qu'il satisfait la spécification dont il a été conçu pour remplir, à partir de l'exactitude (correctness) des composants élémentaires qui compose l'assemblage dans tous les contextes prévus pour le composant. La spécification d'un composant suit une structure définit formellement un contexte pour le composant représenté par une signature (axiomes, fonctions, relations), une interface qui définit des opérations et de contraintes, et un code.

De notre point de vue, ce travail offre un moyen pour spécifier et de vérifier formellement la spécification d'un ensemble de composants logiciels séparément et déduire que l'assemblage de ces composants est aussi correct. En revanche, il ne montre pas les interconnexions et les relations qui existent entre ces composants, i.e. quel composant doit être connecté avec tel ou tel composant, et quels sont les services échangés entre ces derniers via quels interfaces. Il ne modélise donc pas l'aspect de communication définit généralement par les modèles de composants actuellement connus.

Le travail de [66] représente une tentative pour la modélisation formelle d'un composant logiciel. Les auteurs ont proposés une syntaxe et une sémantique bien définit pour un model de composant logiciel qui capture les concepts essentiels dont les composants sont bâtis autours. Différents travaux de formalisations ont été cité dans ce travail parmi eux, la spécification formelle de CORBA en utilisant le langage de spécification Z pour assurer que les facilités ajoutés à CORBA sont correctes, une tentative pour la formalisation du model COM qui consiste en un model basé sur la théorie des ensembles du premier ordre exprimée en langage Z. Chacune de ces formalisations décrit un modèle de composants, et est destiné à un usage particulier. L'approche proposée par les auteurs consiste à définir un composant formellement comme étant un 4-tuple 'X' de la forme (**inports**, **outports**, **function**, **triggers**). Ils considèrent le composant par analogie comme une fonction qui à des entrées (**inports**) et des sorties (**outports**) réalisant une tâche précise (**function**) qui est réalisée en recevant un événement en provenance d'un autre composant (**triggers**). Les (**inports**) et (**outports**) fournissent une interface via laquelle le composant interagit avec les autres composants. Les auteurs renforcent le modèle par des représentations graphiques qui permettent la réalisation d'assemblage de composants. Une sémantique représentant l'état, l'exécution, la composition d'un composant est clairement définie.

Un autre travail, celui de [67] est similaire à notre approche du fait que les auteurs en utilisés une catégorie des réseaux de Petri de haut niveau qui sont les réseaux de Petri colorés, pour fournir une sémantique formelle pour les concepts de base des composants logiciels. Pour cette fin les auteurs ont suivis une approche dans laquelle ils ont commencé tout d'abord par la définition d'un model de composant qu'ils ont nommé 'CompoNet' une abréviation de 'Component and Petri Nets' avec une représentation graphique. Le modèle est largement inspiré du modèle abstrait du CCM de CORBA, puisque il garde le même vocabulaire utilisé par ce dernier. La notion d'assemblage de composants est alors bien décrite. Ils ont ensuite proposés une notation pour spécifier formellement le comportement interne d'un composant logiciel. A travers cette notation les caractéristiques et les

fonctionnalités d'un composant logiciel tel que, l'aspect de communication et les interfaces sont spécifiés. Cette notation est basée sur les RdPs colorés. Ces derniers étant bien adapté pour la spécification des systèmes distribués et concurrents. Le concepteur qui va utiliser l'approche doit en premier lieux définir l'architecture logicielle de son système sous forme d'un assemblage de composants en utilisant 'CompoNet'.Ensuite un mapping est définit depuis les constructions du modèle de composant (CompoNet) (facettes, réceptacles,…) vers les constructions des notations basée sur les RdPs colorés (places, transitions,…) et cela pour chaque composant. Les connexions inters composants sont aussi définit formellement en terme des RdPs colorés. Finalement, une spécification formelle du comportement de l'assemblage de composants est obtenue en reliant les composants entre eux.

Le travail s'est principalement focalisé sur la sémantique du comportement des activités des composants logiciels.

3. Présentation de l'approche de modélisation

Comme nous l'avons mentionné dans le premier chapitre, le développement et l'évolution d'un produit logiciel suit un processus appelé le processus logiciel. Plusieurs approches ont été proposées dans ce cadre, mais elles se basent toutes sur les mêmes activités et se différent seulement de la manière dont elles sont réalisées. L'enchaînement de ces activités déjà abordées dans la section 2.1 du premier chapitre constitue le cycle de vie d'un produit logiciel. La phase de conception est une activité dans laquelle les spécifications de l'architecture générale du logiciel sont élaborées. Les abstractions fondamentales du système logiciel et les relations entre elles sont décrites et identifiées dans cette phase. Dans une approche de développement basé composants, l'étape qui suit la spécification du système consiste en la recherche et la sélection des composants répondant aux besoins spécifiés.

L'approche de modélisation de logiciels à base de composants que nous proposons démarre à partir de l'architecture du système logiciel, montrant l'interconnexion de ses différents composants ainsi que leurs spécifications. Pour ce faire, on s'est basé sur le formalisme des ECATNets pour la spécification des composants ainsi que leurs connexions.

Nous distinguons trois étapes dans l'approche à suivre (figure 15) :

1) **Spécification du système** : lors de cette étape, les interfaces de chaque composant ainsi que les connexions inters composants sont spécifiées en utilisant la notation proposée qui est basée sur le formalisme des ECATNets. Le résultat de cette étape est un modèle ECATNet représentant la spécification d'un assemblage de composant.
2) **Génération des règles de réécriture** : à partir du modèle ECATNet construit dans la première étape, l'utilisateur génère les règles de réécriture associées au système modélisé comme il a été montré dans les exemples du deuxième chapitre.

3) **Vérification du système** : cette étape conclus la finalité et le but d'une modélisation, c'est de pouvoir analyser et étudier certains aspects du système spécifié. Cela est réalisé par le biais d'une spécification écrite en langage Maude, qui intègre dans son corps les règles de réécritures définies dans l'étape précédente. Le résultat de cette étape est une spécification exécutable introduite dans l'interpréteur du langage Maude pour être vérifié.

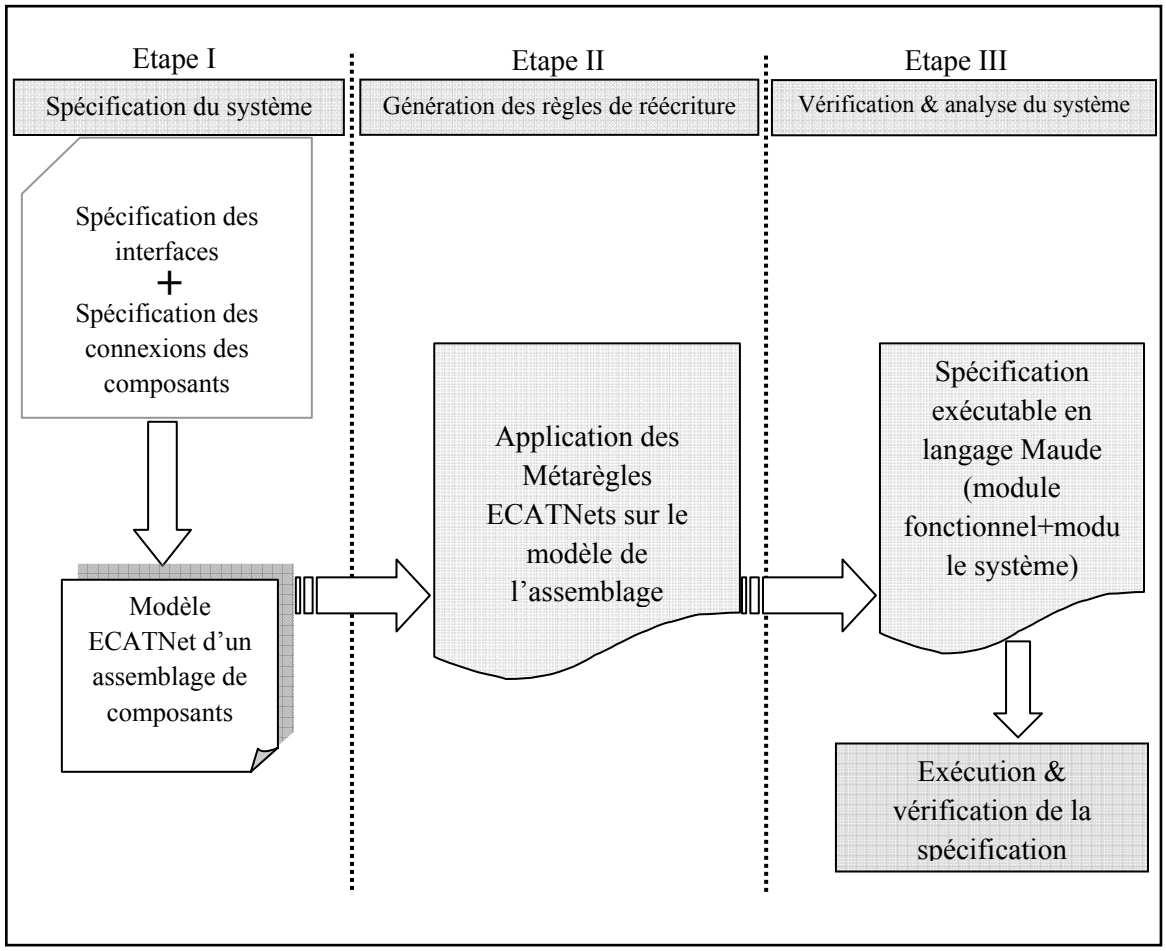


Fig. 15 Etapes de l'approche de modélisation des logiciels à base de composants

Nous aborderons dans ce qui suit chaque étape de notre approche en détail, dans lesquelles nous motivons implicitement et explicitement le choix porté sur les ECATNets.

3.1 Spécification du système

Dans une approche de développement basé composant la construction d'applications logicielles est réalisée par composition. Pour spécifier un système logiciel à base de composants, cela revient à spécifier les composants qui le constituent ainsi qu'aux interconnexions entre ces composants. Un composant logiciel est décrit en termes de ces interfaces à travers lesquelles il peut interagir avec l'environnement externe constitué des autres composants dans le système.

La plupart des approches actuelles pour la définition des interfaces des composants se limitent essentiellement aux spécifications syntaxiques. Cette forme de spécification comprend les spécifications utilisées dans des technologies comme COM, CORBA, et JavaBeans [1]. Les deux premières utilisent les IDLs, alors que la troisième utilise le langage de programmation Java pour spécifier les interfaces d'un composant. Or, les spécifications syntaxiques n'offrent pas des informations sur la sémantique des opérations qui définissent ces interfaces tel que, par exemple les contraintes imposées pour l'exécution des opérations.

Nous proposant dans ce qui suit (figure 16) une notation basée sur le formalisme des ECATNets. A travers cette notation nous apporterons un formalisme sémantique par lequel une sémantique formelle du système peut être spécifiée. Ainsi, nous espérons couvrir deux niveaux de spécification de propriétés dans un composant : le niveau comportemental et le niveau synchronisation. Le premier niveau exprime les conditions d'entrées et de sorties des opérations qui définissent les interfaces d'un composant logiciel. Alors que le deuxième, exprime les dépendances entre les services fournis par un composant, tel que le parallélisme.

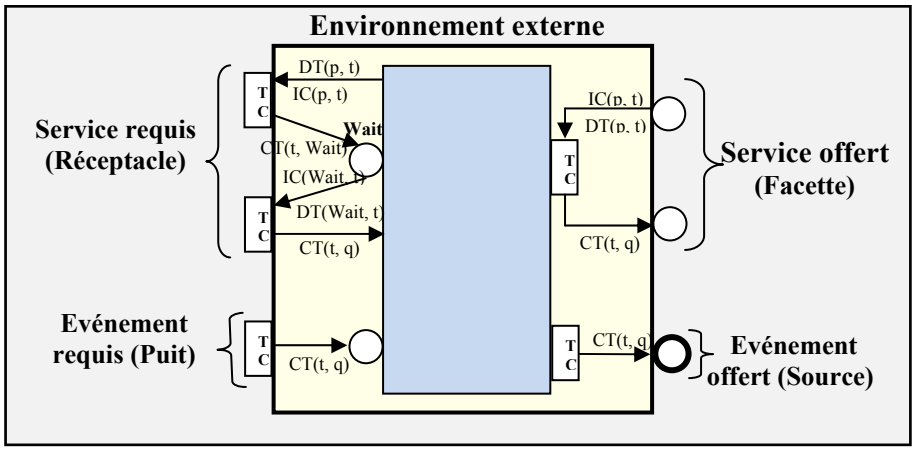


Fig.16 Spécification d'un composant logiciel

Cette notation graphique représente la spécification formelle des interfaces d'un composant logiciel. Une interface d'un composant peut être définie comme une spécification de ces points d'accès. Les composants clients accèdent aux services fournis par le composant en utilisant ces points. Si un composant à plusieurs points d'accès dont chacun représente un service offert par le composant, le composant est supposé avoir plusieurs interfaces.

En omettant les différences dans les détails techniques tels que, le langage d'implémentation, les mécanismes de paramétrage, méthodologies de définition d'interfaces,…etc, les technologies de composants (CORBA, COM, EJB,…) ont essentiellement des architectures et fonctionnalités similaires. Chacune de ces technologies définit un composant comme étant une boite noire qui émet et reçoit des messages vers et

depuis d'autres composants via des interfaces bien définies, et réalise des calculs en réponse à des événements [66].

Notre travail est similaire à celui de [67], nous avons suivi le même vocabulaire du modèle à composant utilisé pour la spécification des interfaces du composant logiciel qui est le CCM de CORBA pour sa simplicité. Les fonctionnalités offertes (Facettes) sont considérées comme des services nécessaires à d'autres composants, ceux qui sont requis (Réceptacles) sont des services essentiels pour le composant pour accomplir ses fonctions et qui sont fournis par d'autres composants. L'interaction entre les composants est représentée par la communication par événement et par invocations de méthodes. Chaque composant peut envoyer (événements source) et recevoir (événements puits) des événements de son environnement externe.

Une caractéristique essentielle d'un composant est la séparation explicite entre les aspects interne et externe de l'entité à encapsuler pour représenter un composant. En d'autre terme, un composant possède un intérieur caché et des interfaces exposées [10]. Dans la notation proposée (fig.14) cet aspect dans les composants logiciels est pris en considération et est représenté par la zone nommée « vue interne ». Plus tard dans ce chapitre, nous allons voir que cette partie du composant à été spécifiée seulement dans le but de simuler le fonctionnement interne et non pas de spécifier l'implémentation du composant logiciel qui doit être entièrement encapsuler.

Pour modéliser donc un logiciel à base de composant en utilisant la notation proposée, nous devons spécifier les interfaces de chaque composant ainsi qu'aux interconnexions entre ces composants. Nous allons détailler dans la section suivante, outre la spécification de chaque interface, la spécification des interconnexions qui permettent de construire un assemblage de composants.

3.1.1 Service requis (Réceptacle)

Un service requis modélise une invocation de méthode qui réside dans un autre composant. L'invocation d'une méthode représente une communication de type synchrone entre le composant client et le composant serveur (fournisseur du service). Un 'Réceptacle' est spécifié par deux transitions qu'on a nommées (StartInvocation, EndInvocation) et une place (Wait) comme il est illustré par la figure 17.

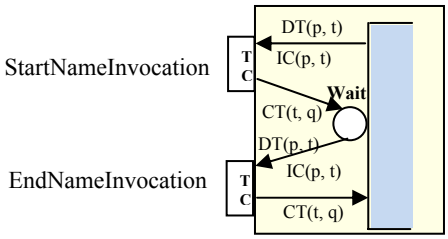


Fig.17 Spécification d'un service requis (Réceptacle)

La transition StartInvocation (SnameInv en abrégé) spécifie le début de l'invocation. L'arc entrant modélise un contrat (section 3.4.1.2 du chapitre 2) qui représente les pré-conditions (contraintes) qui doivent être assurées par le comportement interne du composant pour pouvoir invoquer une méthode. Le nom de la méthode est explicitement donné par le nom de la transition (exemple: Start**Extract**Invocation, Extract est le nom de la méthode).

L'arc sortant est connecté à la place 'Wait', cette dernière modélise le fait qu'un composant est dans un état bloqué en attente d'une réponse depuis le composant fournisseur du service. Cet arc est étiqueté par le multiset de terme CT (Created Token) qui indique l'état que le composant devrait atteindre.

La transition EndInvocation (ENameInv) spécifie la fin de l'invocation de méthode. Son arc entrant arrivant depuis la place 'Wait', modélise le contrat représentant les conditions pouvant faire basculer l'état du composant vers l'état non bloqué. L'arc sortant représente les post-conditions qui devront être valides pour le comportement interne du composant après l'exécution de l'action modélisée par la transition.

3.1.2 Service offert (facette)

Un service offert est une interface représentée par une opération nécessaire pour d'autres clients pour accomplir leurs tâches. Une facette véhicule les fonctionnalités spécifiques qu'offre le composant à ses clients et que le développeur doit implanter. On a modélisé une facette dans notre notation (figure 18) par deux places (NameRequest, NameResult) et une transition (Name : est le nom donné à l'opération associée à la transition).

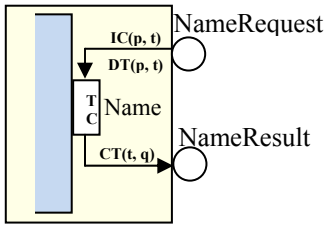


Fig. 18 Spécification d'un service offert (Facette)

La place 'NameRequest' modélise les invocations reçues depuis les composants clients existant dans l'environnement du composant offrant le service. Ainsi, un service fournit peut être invoqué par plusieurs composants simultanément. Le franchissement de la transition qui dépend des multisets de termes IC (Input Condition) et TC (Transition Condition) représente l'exécution de l'opération associée à cette transition. La place 'NameResult' modélise le service fournit comme résultat de la transition franchie.

Dans certains cas lorsqu'un composant client invoque un service offert situé dans un autre composant, ce dernier pour fournir le service doit faire appel à d'autres services situés chez d'autres composants qui eux même peuvent demandés à leurs tours des services

d'autres composants. Nous proposons une adaptation du modèle plus haut (fig.16), en éclatant la transition 'Name' en deux et en ajoutant une place 'Wait' qui modélise un état de blocage du composant. L'état de blocage représente l'attente d'une réponse d'un autre composant. La première modélise le début du service et la deuxième sa fin. Ce modèle (figure 19) représente le cas le plus général pour modéliser un service offert (Facette).

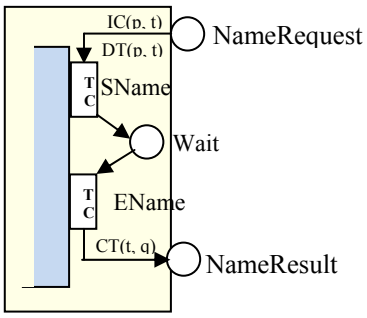


Fig. 19 Modèle générale d'un service offert (Facette)

3.1.3 Evénement offert (source)

Un événement offert ou bien événement source décrit un événement qui peut être émis par le composant. Cet événement peut être intercepté par d'autres composants qui ont manifesté leurs intérêts pour lui, il peut être alors envoyé en multicast à plusieurs récepteurs. On a modélisé ce type d'événement par une transition (NameSource) et une place qu'on a nommée 'signal' pour différencier entre la communication par invocation de méthode et la communication par événement dans un composant, et donner une manière signifiante pour représenter l'événement émis (figure 20).

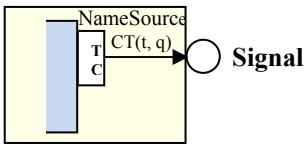


Fig. 20 Spécification d'un événement source

Le franchissement de la transition 'NameSource' aura comme conséquence le dépôt d'un jeton CT (Created Token) dans la place 'signal', pour être consommé par la suite par la transition puit en relation d'un autre composant qui à présenté son intérêt à l'événement source.

3.1.4 Evénement requis (puit)

Un événement puit est un événement que le composant est disposé à recevoir. On a conservé pour la modélisation d'un événement puit la même notation graphique pour la modélisation d'un événement source.

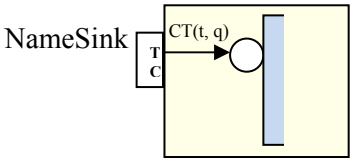


Fig. 21 Spécification d'un événement puit

La transition est interfacée pour recevoir un événement source. Le fait qu'une transition soit franchie modélise la réception d'un événement et le dépôt d'un jeton CT dans la place reliant l'arc sortant de la transition valide l'arrivée d'un événement source en provenance d'un autre composant.

3.1.5 Spécification des connexions

Une composition peut être obtenue entre les composants en reliant chaque fonctionnalité spécifiée à une autre, les services offerts avec les services requis et les événements offerts avec les événements requis de telle façon on obtiendra un assemblage de composants.

- **Spécification d'une connexion : Evénement source / puit**

Un événement source peut être connecté un n'importe quel nombre d'événements puits, c'est ainsi que le multicast d'événement vers plusieurs destinations (composants) est modélisé. Un événement puit (requis) peut être connecté de la même manière, ce qui implique qu'il peut recevoir le même événement de sources distinctes. La figure 22 montre le modèle de connexion entre un événement source et un événement puit en utilisant la notation graphique proposée.

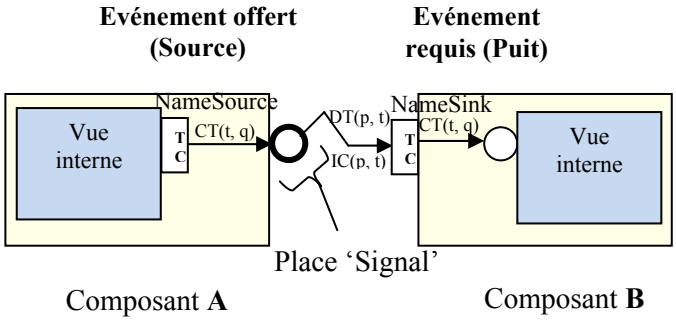


Fig. 22 spécification d'une connexion : événement Source /Puit

La connexion entre les deux types d'événement est réalisée simplement en reliant la place du composant émetteur 'A' avec la transition 'NameSink' du composant récepteur 'B' tout en respectant le formalisme des ECATNets.

Le franchissement de la transition 'NameSink' dépend de la présence d'un jeton CT dans la place de sortie de la transition 'NameSource'. On a nommé cette place 'Signal' pour illustrer la communication par événement.

- **Spécification d'une connexion : Services offerts / requis**

La spécification d'une connexion entre un service offert (facette) et un service requis (Réceptacle) modélisant une invocation de méthode unicast, est donnée par la réutilisation

du modèle client / serveur définit dans [68]. La connexion est réalisée par deux arcs (figure 23) :

- Le premier est un arc sortant depuis la transition ‘SNameInv’ du composant ‘B’ vers la place ‘NameRequest’ du composant ‘A’.
- Le second est un arc sortant depuis la place ‘NameResult’ du composant ‘A’ vers la transition ‘ENameInv’ du composant ‘B’.

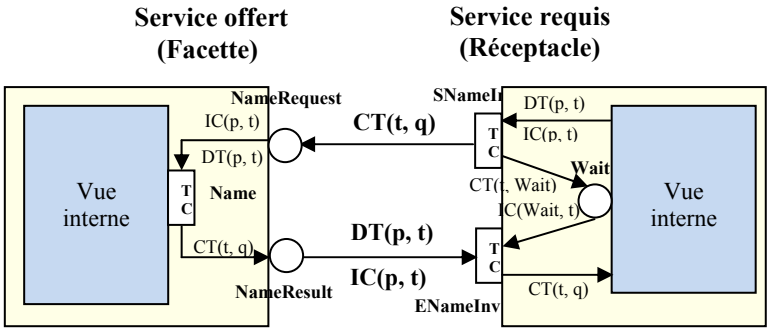


Fig.23 Spécification d’une connexion :
Services offerts / requis

La transition ‘SNameInv’ du composant ‘B’ est connectée avec la place ‘NameRequest’ du composant ‘A’ par un arc sortant. Le franchissement de cette transition représente l’invocation du service demandé et ajoute un jeton CT à la place ‘NameRequest’ ce qui aura comme conséquence le déclanchement du franchissement de la transition ‘Name’ du composant ‘A’. D’un autre côté, la transition ‘SNameInv’ du composant ‘B’ dépose un second jeton dans la place ‘Wait’, ce qui changera bien sûr l’état du composant.

Après le franchissement de la transition ‘Name’, un jeton est créé dans la place ‘NameResult’ et de cette manière le résultat du service est prêt à être reçus. La transition ‘EnameInv’ est maintenant franchissable, son tirage fait délivrer le résultat au composant et change son état d’un état d’attente bloquant vers un état non bloquant.

Maintenant, pour modéliser un assemblage de composants logiciels avec les notations proposées, il suffit d’identifier en premier lieu pour chaque composant le type d’interfaces qui expose aux autres composants et dégager les opérations qui les représentent. Le langage de description utilisé pour définir les interfaces du composant peut être par exemple de la famille des IDLs, le langage Java, une spécification en diagramme de composants UML, ou même une description en langage naturel qui spécifie toutes les opérations de chaque interface. L’intérêt de cette étape est de préciser l’enveloppe qu’un composant expose à l’environnement externe dont il fait partie, ainsi que les types de paramètres importés et exportés par ce composant. Ensuite, en utilisera bien sur les spécifications données plus haut pour modéliser chaque type d’interface.

Les types de paramètres des opérations de chaque interface représenteront les structures de données définies par le modèle de données des ECATNets qui est un formalisme algébrique (spécifications algébriques de types abstrait de données). Le résultat final de cette démarche est un modèle ECATNets modélisant une application à base de composants.

3.2 Génération des règles de réécriture

Ce qui caractérise les ECATNets des autres types de réseaux de Petri, est que leurs sémantique est donnée en terme de la logique de réécriture pour décrire le comportement dynamique des systèmes modélisés. Cette logique consiste en un ensemble d'axiomes sous la forme de règles de réécriture et un ensemble de règles d'inférence. Avec cette logique, le comportement du système peut être expliqué par un raisonnement formel.

Pour pouvoir raisonner correctement sur le système en conception, modélisé par le biais des notations basées sur le formalisme des ECATNets, la phase suivante de l'approche à suivre consiste en la génération des règles de réécriture spécifique au modèle ECATNet construit dans la phase antérieure. Pour chaque composant dans le modèle il faut identifier ses transitions, et appliquer ensuite pour chaque transition la métarègle appropriée (section 5.4 du chapitre 2) pour générer la règle de réécriture associée.

3.3 Vérification du système

Dans cette phase le concepteur du système doit préciser tout d'abord les propriétés à vérifié. Ces propriétés permettent d'analyser un aspect comportemental du système modélisé. La vérification se fait à travers une spécification exécutable écrite en langage Maude qui inclut dans son corps les règles de réécritures générées à partir du modèle ECATNet du système en cours de spécification. Cette spécification permet de vérifier formellement le comportement dynamique du système modélisé. L'apport de la vérification formelle est significatif en particulier dans les premières phases du processus de développement logiciel. Elle permet d'éliminer très tôt dans ce processus les erreurs qui peuvent se produire dans la spécification des besoins, et de résoudre les inconsistances découvertes évitant ainsi leurs propagations durant tous le cycle de développement.

L'usage des ECATNets regroupe à la fois, l'avantage d'une représentation graphique simple qui accroît la lisibilité et facilite la compréhension des modèles tel est le cas pour tous les types des réseaux de Petri, et détient la spécificité de posséder l'avantage de pouvoir vérifier formellement des propriétés via une spécification en logique de réécriture. Cette dernière possède un langage formel très puissant qui l'implémente, permettant ainsi de fournir une spécification exécutable du système. C'est le langage est Maude.

La spécification Maude doit contenir deux modules :

1) Un module fonctionnel qui capte l'aspect statique du système en définissant les types de données et les opérations qui les manipulent. Les données à déclarer dans ce module

représentent les paramètres faisant partie des signatures des opérations de chaque interface dans le composant. En d'autres termes c'est les entrées et les sorties de chaque opération. Le module suivant emprunté de [69] décrit les opérations de base pour un ECATNet :

```
fmod ECATNET is
sorts Place Marking GenericTerm.
op null : -> Marking .
op <_ | _> : place GenericTerm -> Marking .
op __ : Marking Marking -> Marking [assoc comm id: null] .
endfm
```

Comme il est illustré dans ce code, null est le marquage vide qui implémente l'élément identité pour ⊗. L'opération '<_ | _>' permet la construction des marquages élémentaires. Les deux caractères soulignés indiquent la position des paramètres de l'opération. Le premier paramètre est une place, le second est un terme algébrique (un marquage) dans la place. L'opération qui implémente l'opération ⊕ n'est pas définit dans ce code. L'opération '__' qui implémente l'opération ⊗ est suffisante tout en se basant sur la notion de décomposition. Si une place contient plusieurs termes, par exemple (p, a ⊕ b ⊕ c) alors en peut l'écrire sous la forme (p, a) ⊗ (p, b) ⊗ (p, c).

2) Un module système qui capte l'aspect dynamique en termes de règles de réécriture. Dans ce module on inclut les modules fonctionnels déjà définis, ainsi que les règles définies dans la deuxième phase.

On utilise le mot clé « including » pour l'importation la plus générale de modules. Cette opération peut également être faite on utilisant d'autres modes d'importation via « protecting » ou « extending ». Une fois le code est spécifié, la vérification du système se fera à l'aide des commandes de Maude '**rewrite**', '**frewrite**', '**search**', '**show path**', etc…

Nous préférons montrer comment cette tache peut être réalisée à travers un exemple qui sera développé dans la section 4 du présent chapitre.

4. Etude de cas

Nous allons essayer à travers deux exemples de montrer en détails l'application de notre approche de modélisation proposée dans le cadre d'une méthodologie de développement basée composants. Le premier exemple est tiré du domaine des interfaces utilisateurs, il a été présenté dans [67]. Nous avons réutilisé cet exemple pour présenter l'utilisation des notations proposées pour modéliser une application à base de composants (assemblage de composants). Nous nous sommes limités à présenter seulement la phase de spécification du système dans cet exemple. Le deuxième exemple que nous proposons, représente un système connu dans la littérature du domaine de gestion des processus métier (BPM[18]). Il modélise le processus de réservation d'une agence de voyage. On a effectué

[18] Business Process Management

de légère modification pour des raisons de simplicité. Dans cet exemple, toutes les étapes seront présentées.

4.1 Présentation de l'exemple1

Pour illustrer notre approche pour modéliser formellement des logiciels à base de composants, nous présentons un exemple emprunté de [67]. Dans cet exemple, une simple application du domaine des interfaces utilisateurs est montrée (figure 24) :

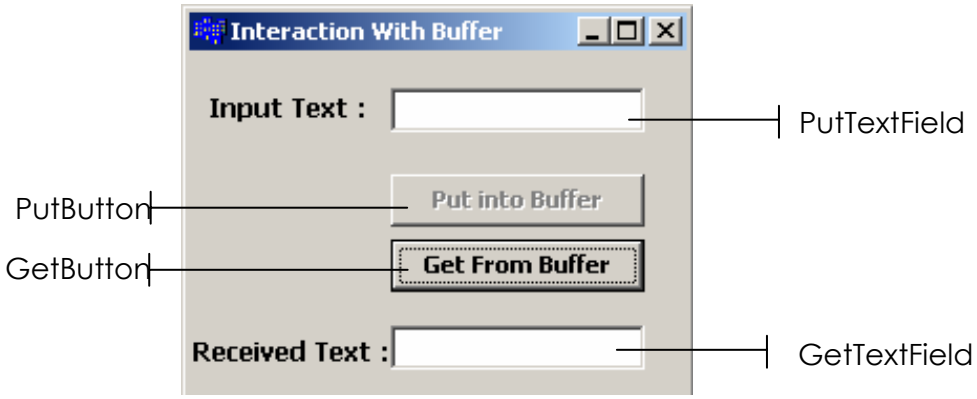


Fig.24 Interface utilisateur de l'exemple [67]

Cette application permet à l'utilisateur de saisir une chaîne de caractères (texte) dans une zone d'édition de texte 'PutTextField', et mettre le texte tapé dans une mémoire tampon (buffer) d'une capacité inconnue en cliquant sur un bouton 'PutButton'. En revanche, un autre bouton 'GetButton' permet à l'utilisateur d'obtenir une chaîne de caractère à partir de la mémoire tampon, et affiche le texte retiré dans une autre zone de texte. Les boutons doivent être activés ou désactivés en fonction de l'état de la mémoire tampon : si le buffer est plein, le bouton 'PutButton' doit être désactivé. En revanche, si le buffer est vide, le bouton 'GetButton' doit être désactivé.

L'exemple identifie quatre composants visuels (PutButton, GetButton, PutTexField et GetTextField) (figure 24), et trois composants non visuels : 'MyBuffer' qui modélise la mémoire tampon de capacité inconnue, 'GetAdapter' et 'PutAdapter' qui fournissent la logique nécessaire pour assembler les différentes pièces.

Dans le travail de [67], l'architecture du système montrant l'assemblage de composants a été donnée en terme d'une notation basée sur le modèle CCM de CORBA en gardant le vocabulaire utilisé par celui-ci. La spécification de chaque composant a été donnée en utilisant le langage CPN-ML, une variante du standard ML. Ce langage ne supporte pas la notion d'interface. De notre part nous avons utilisé le langage de description d'interface de CORBA l'IDL3 pour la spécification des interfaces de chaque composant. Ce choix est motivé pour sa simplicité et sa clarté, en revanche l'utilisation d'un autre langage comme Java est possible aussi. Il est à noter que la spécification des composants en IDL3 n'est pas incluse comme une étape de notre approche mais elle constitue le point de départ de celle-ci dont le concepteur va utiliser durant la modélisation.

Nous procédons dans ce qui suit à la spécification de chaque composant de notre exemple à part.

4.1.1 GetButton

C'est un composant visuel qui présente une enveloppe de quatre événements ; trois événements Puit (requis) et un événement Source (offert). La syntaxe en IDL3 pour la spécification d'un événement Source et un événement Puit dans un composant est donnée respectivement par les figures 25 et 26 :

```
Le cas d'un broadcast (plusieurs destinataires):
publishes <event_type> <name_identifier>

Le cas d'une connexion point à point:
emits <event_type> <name_identifier>
```

Fig.25 Syntaxe IDL3 pour un Source

```
consumes <event_type> <name_identifier>
```

Fig.26 Syntaxe IDL3 pour un Puit

La déclaration d'un événement se fait par le mot clé '*eventtype*'. Ce dernier définit un bloc qui contient des champs représentant les données de l'événement et éventuellement des opérations qui permettent de manipuler ces données.

```
eventtype EnableEvent
{
public string en ;
} ;

eventtype DisableEvent
{
public string dis ;
} ;

eventtype ClickPEvent
{
public string ck ;
} ;

eventtype ClickSEvent
{
public string text ;
} ;

        //syntaxe de déclaration d'un composant
        component <name> [ :<base>][supports<interface>],[,<interface>]
                {<attribute declaration>
                  <port declaration>
                }
component GetButton
{
  emits ClickSEvent to_GetAdapter; // événement offert au composant GetAdapter
  consumes EnableEvent from_MyBuffer : //événement consommé offert par le composant MyBuffer
     consumes DisableEvent from_MyBuffer :
  consumes ClickPEvent from_MyBuffer ;
};
```

Fig.27 Spécification du composant GetButton en IDL3

A partir de cette spécification nous pouvons construire le modèle équivalent en utilisant les notations proposées, en faisant un 'mapping' du modèle en IDL vers celui en ECATNet. Le résultat est illustré dans la figure 28 :

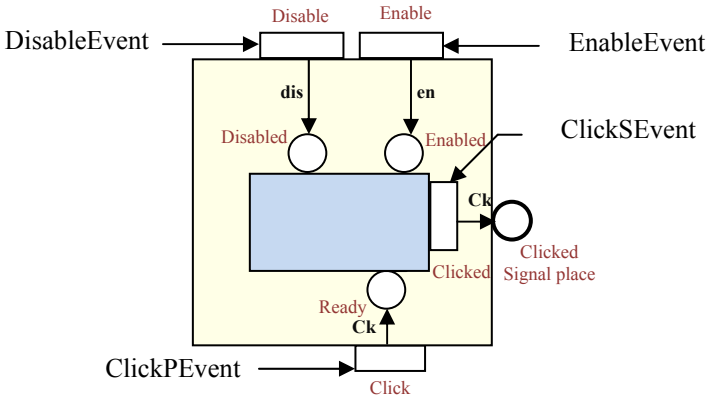


Fig.28 Modèle ECATNet du composant GetButton

Les inscriptions sur les arcs sont des termes algébriques (marquages) représentant les messages envoyés via une communication asynchrone par événement entre les composants. Ce composant véhicule une chaîne de caractère 'Ck' par l'événement source 'Clicked', et consomme trois chaîne de caractères 'dis','en', et 'Ck' respectivement par les puits d'événements 'DisableSink', EnableSink', et 'ClickSink'. Selon l'état du buffer 'plein' ou 'vide' le composant est soit dans un état actif ou inactif, il reçoit s'il est actif un événement click de souris, ce qui provoque l'envoi d'un événement source vers le composant 'GetAdapter' comme il est spécifié dans la déclaration du composant. Ce dernier contient la logique pour extraire un texte du buffer et de le remettre dans la zone de texte 'GetTextField'. Si le buffer est vide le bouton est signalé par un événement qui sera consommé et remettra son état vers inactif. Pour offrir une sémantique pour le comportement interne du composant 'GetButton' nous allons simuler le fonctionnement interne qui doit être masqué. La figure 29 montre le comportement dynamique du composant.

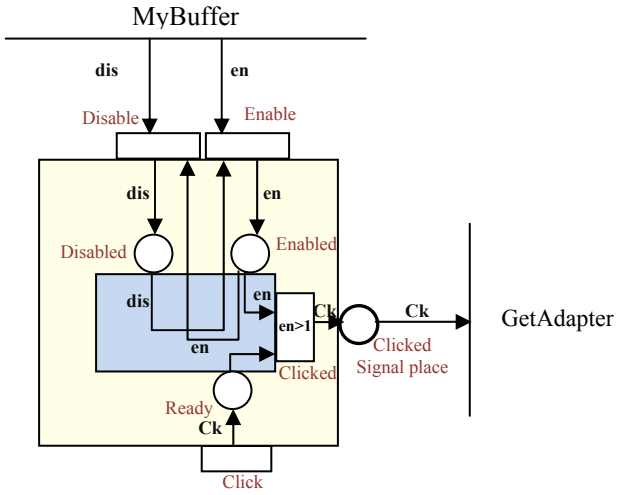


Fig.29 Spécification formelle du comportement du composant GetButton

La condition additive (en>1) imposée sur la transition 'Clicked' modélise la contrainte de la présence de deux jetons 'en' dans la place 'Enabled', pour que la transition qui représente l'envoi d'un événement source 'Clicked' soit franchissable. Le fait de recevoir un click de souris avec un seul jeton modélise la perte du signal (l'événement est ignoré) et la transition n'est pas franchissable. Cela montre bien la puissance des notations syntaxiques des ECATNets dans la modélisation des communications asynchrones par événement.

En comparant les deux types de spécification celle de la figure 27 en IDL3 et celle de la figure 28 utilisant le formalisme des ECATNets, on remarque que la première n'offre aucune sémantique sur les opérations mais simplement une description de celles-ci. A l'inverse celle de la figure 29 offre une sémantique formelle des opérations fournies par le composant. Les pré / post conditions précisant les contraintes d'exécution de chacune d'elles sont bien spécifiées.

4.1.2 PutButton

Ce composant est la copie identique du composant 'GetButton'. Il représente une instance du même type de composant, à l'exception de l'événement source qu'il émet vers le composant 'PutAdapter'. Pour cela, nous présenterons seulement la déclaration du composant (figure 30) :

```
component PutButton
{
 emits ClickSEvent to_PutAdapter ; //événement offert au composant PutAdapter
 consumes EnableEvent from_MyBuffer ; //événement consommé offert par le composant MyBuffer
 consumes DisableEvent from_MyBuffer ;
 consumes ClickPEvent from_MyBuffer ;
}
```

Fig.30 Spécification du composant PutButton en IDL3

Le composant PutAdapter contient la logique pour extraire une chaîne de caractère de la zone de texte 'PutTextField' puis de la remettre dans la mémoire tampon. Si ce dernier est plein le bouton est signalé par un événement qui sera consommé est remettra son état vers inactif.

4.1.3 MyBuffer

Ce composant non visuel expose quatre sources d'événements et deux services offerts. Il modélise une mémoire tampon. Dans notre cas on considère qu'il est de capacité N, il peut être dans deux états plein ou vide. La définition d'une facette se fait par le mot clé '*provides*' (figure 31). Sa spécification est donnée dans la figure 32 :

provides <Interface > <name_identifier>

Fig.31 Syntaxe IDL3 pour une facette

```
eventtype NotFull
{
 public string Nful ;
}

eventtype Full
{
 public string ful ;
}

eventtype empty
{
 public string em ;
}

eventtype Notempty
{
 public string Nem ;
}

interface Buffer
{
 void get (out string text) ; //service offert au composant GetAdapter
 void put (in string text, out string text) ;// service offert au composant PutAdapter
}

  component MyBuffer supports Buffer
  {
   provides  Buffer buf ; //interface fournit a GetAdapter et PutAdapter
   emits NotFull to_PutButton ; //événement offert au composant PutAdapter
   emits Full to_ PutButton ; //événement consommé offert par le composant MyBuffer
   emits empty to_GetButton ;
   emits Notempty to_ GetButton ;
  }
```

Fig.32 Spécification du composant MyBuffer en IDL3

On remarque bien que ce composant communique de manière asynchrone avec deux composants, 'PutButton' et 'GetButton' les notifiant de son état lors d'une extraction ou insertion d'un message. Cela est réalisé via les événements sources qu'il émet. Les conditions de son fonctionnement n'apparaissent pas dans la spécification fournie. Lorsque par exemple le buffer est rempli par l'insertion d'un message et qu'aucune place n'est vide, le buffer notifie les deux boutons. 'PutButton' par l'événement 'Full' indiquant qu'il est plein et qu'il ne peut pas recevoir des messages ce qui remettra son état vers inactif. 'GetButton' par l'événement 'NotEmpty' signalant qu'il y a une place vide et qui peut contenir un message ce qui remettra son état vers actif.

Le composant offre aussi une interface 'Buffer' définit par deux opérations qui constituent des services offerts par le composant lui permettant de jouer un rôle précis. Ces derniers représentent des services requis par les composants 'PutAdapter' et 'GetAdapter'. Le premier fait usage de l'opération 'put', celle-ci permet le dépôt d'un message dans la mémoire tampon. Le second invoque l'opération 'get' pour extraire un message de la mémoire tampon. La définition d'interfaces de type Facette et Réceptacle permet une communication synchrone par invocation de méthode où l'appelant et l'appelé agissent selon le modèle client/serveur. Le client demande le service et se met en état de blocage, jusqu'à ce que le fournisseur du service (serveur) lui renvoi le résultat. Le modèle

ECATNet spécifique au composant 'MyBuffer' est donné dans la figure 33 après un mapping à partir de la spécification fournie plut haut :

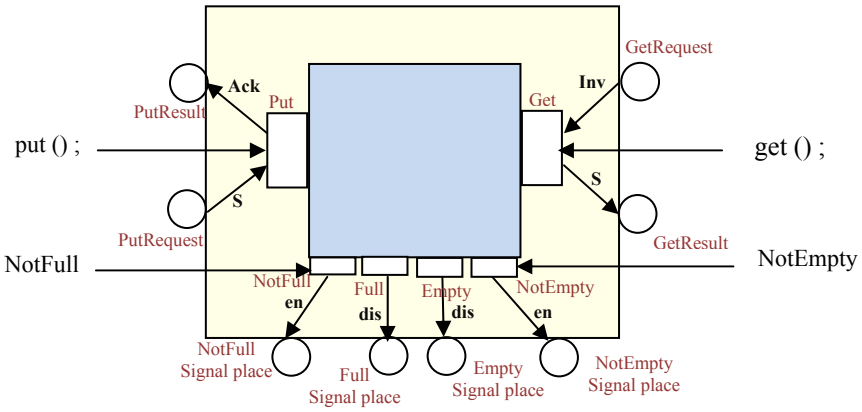


Fig.33 Modèle ECATNet du composant MyBuffer

La sémantique du comportement dynamique du composant est spécifiée formellement dans la figure 34. La spécification du comportement interne des composants est donnée dans le but d'une vérification du système suivant l'approche proposée.

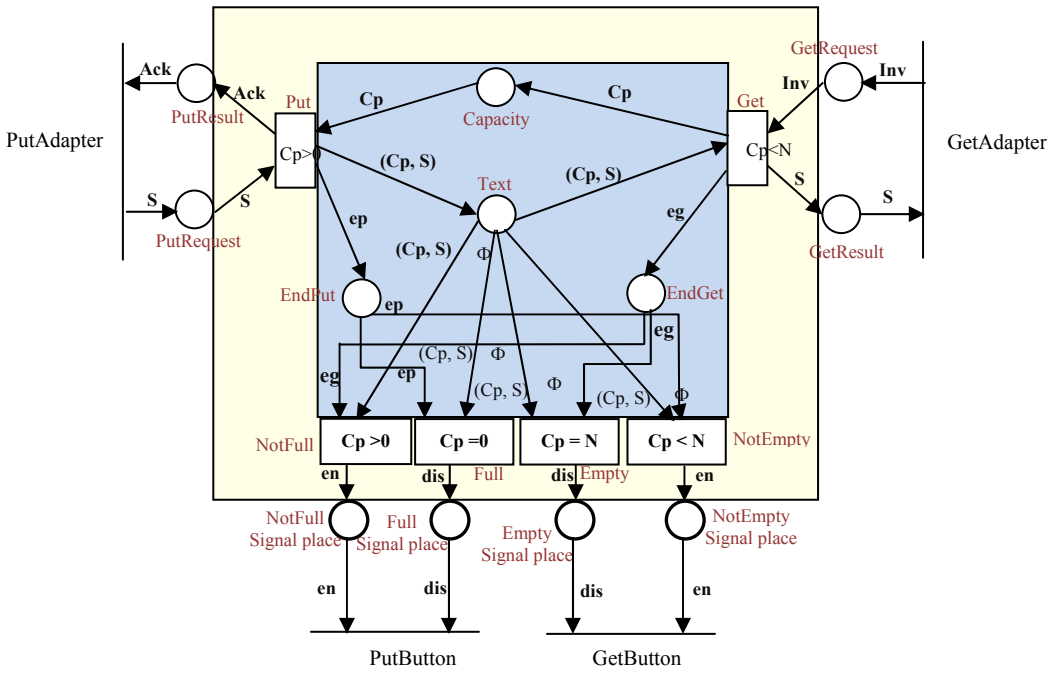


Fig.34 Spécification formelle du comportement du composant MyBuffer

Nous avons effectué une légère modification sur la modélisation du composant 'MyBuffer' par rapport à l'exemple d'origine de tel sorte il pourra contenir N message en vue d'exprimer plus de parallélisme dans notre modèle. La variable 'N' dans le modèle ci-dessus représente la capacité du buffer, 'Cp' est le nombre de places vides qui peut avoir initialement une valeur inférieur ou égale à N contenue dans la place 'Capacity'. Le terme algébrique composé (Cp, S) nous indique la présence d'une chaîne de caractère 'S'avec l'état courant du buffer en nombre de places vides. La présence d'un jeton de ce type dans la place 'text' permettra au buffer de décider sur le comportement à suivre, activer ou

désactiver les composants 'PutButton' et 'GetButton'. Les places 'EndPut' et 'EndGet' indiquent respectivement la fin d'une opération d'insertion et d'extraction de messages.

4.1.4 PutTextField

C'est un composant visuel qui permet la saisie d'une chaîne de caractère. Il offre une enveloppe constituée d'une facette et d'un événement Puit. Sa spécification est donnée dans la figure 35 :

```
eventtype KeyPressed
{
  public string KeyPr ;
}
interface GetText
{
  void get (out string text) ; // service fournis au composant  PutAdapter
}
component PutTextField  supports Text
{
  provides GetText My_GetText ; //interface fournit au composant PutAdapter
  consumes KeyPressed id_keypressed; //événement consommé par frappe de clavier
}
```

Fig.35 Spécification du composant PutTextField en IDL3

Le mapping entre cette spécification et celle proposée nous donne le modèle ECATNet de la figure 36 :

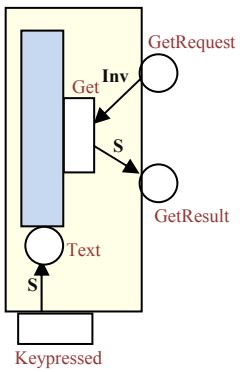


Fig.36 Modèle ECATNet du composant PutTextField

Ce composant consomme un événement Puit suite à une frappe sur le clavier ce qui fait véhiculé un caractère vers la zone de texte. Il offre éventuellement un service a utilisé par le composant non visuel 'PutAdapter' pour insérer la chaîne de caractère saisie dans la zone de texte dans la mémoire tampon. L'opération de dépôt d'un message réalisé par 'PutAdapter' ne se lancera que si ce dernier a été notifié par le composant 'PutButton' via une source d'événement, comme il a été montré dans sa spécification. La figure 37 montre le comportement dynamique du composant.

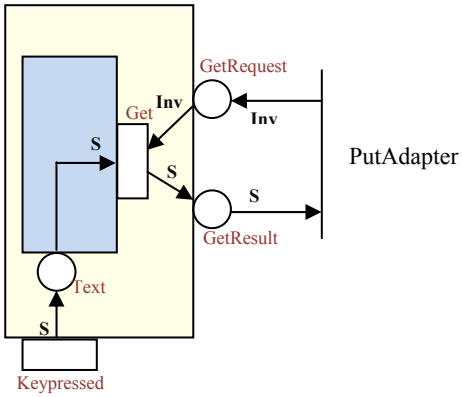


Fig.37 Spécification formelle du comportement du composant PutTextField

4.1.5 GetTextField

Ce composant visuel possède un service à fournir pour le composant 'GetAdapter'. Il modélise une zone de texte pour afficher le texte reçu du buffer. La spécification en IDL3 est donnée dans la figure 38 :

```
interface PutText
{
  void put (in string text) ; // service fournis au composant  PutAdapter
}
component GetTextField  supports Text
{
  provides  PutText My_PutText ; //interface fournit au composant GetAdapter
}
```

Fig.38 Spécification du composant GetTextField en IDL3

Le modèle ECATNet équivalent (figure 39) :

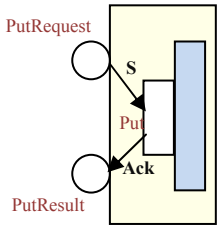


Fig.39 Modèle ECATNet du composant GetTextField

En la présence d'un message dans le buffer, le composant 'GetAdapter' le fait récupérer suite à un click du bouton 'GetButton' et le dépose dans la zone de texte représenté par 'GetTextField' en invoquant l'opération 'put' fournie par l'interface 'PutText'. Cette dynamique est exprimée dans la figure 40 :

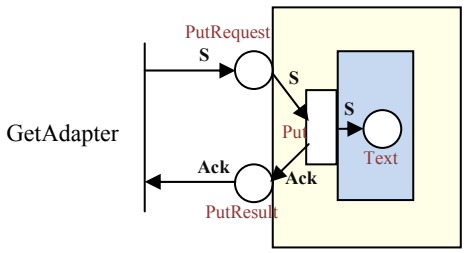


Fig.40 Spécification formelle du comportement du composant PutTextField

4.1.6 PutAdapter et GetAdapter

Ces deux composants non visuels représentent deux instances identiques du même type de composant. Leurs rôles sont d'adapter les événements en provenance des boutons aux services fournis par le buffer et les zones de texte. Le mapping d'un service requis (Réceptacle) est montré à travers la modélisation de ces composants. La spécification en IDL3 des deux composants (figure 42, 43). La syntaxe pour définir un Réceptacle (figure 41). On utilise le mot clé '*uses*'.

uses <Interface > <name_identifier>

Fig.41 Syntaxe IDL3 pour une facette

```
eventtype PerformGet
{
  public string PerGt ;
}
component GetAdapter
{
  uses  PutText id_PutText ; //interface requise offerte par GetTextField
  uses Buffer id_Buffer;  // interface requise offerte par MyBuffer
  consumes PerformGet from_GetButton; //événement offert par GetButton
}
```

Fig.42 Spécification du composant GetAdapter en IDL3

```
eventtype PerformPut
{
  public string PerPt ;
}
component PutAdapter
{
  uses  GetText id_PutText ; //interface requise offerte par PutTextField
  uses Buffer id_Buffer;  // interface requise offerte par MyBuffer
  consumes PerformPut from_PutButton; //événement offert par PutButton
}
```

Fig.43 Spécification du composant PutAdapter en IDL3

Pour modéliser une interface requise, nous devons voir les opérations déclarées dans sa spécification. Par exemple, l'interface buffer contient deux services offerts représentés par les opérations get() pour le composant 'GetAdapter' et put() pour le composant 'PutAdapter'. Ensuite, représenter chaque service requis par la notation proposée. La figure 44 est le résultat de la modélisation.

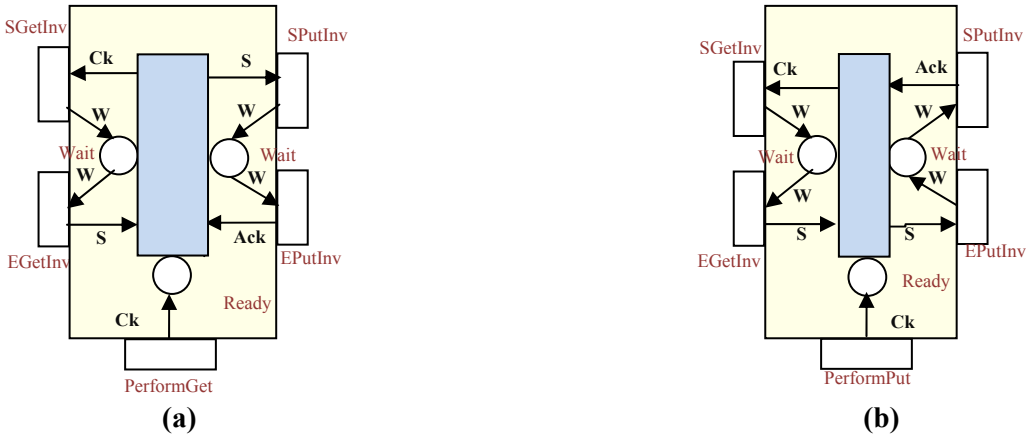


Fig.44 Modèle ECATNet : (a) GetAdapter, (b) PutAdapter

Le composant 'PutAdapter' s'occupe d'extraire une chaîne de caractère de la zone de texte puis de la déposée dans le buffer en invoquant successivement les opérations get() de 'PutTextFiels' et put() de 'MyBuffer'. De même, 'GetAdapter' récupère le texte déposé à partir du buffer puis le remet dans la zone de texte 'GetTextField' en invoquant successivement les opérations get() de 'MyBuffer' et put() de 'GetTextField'. La figure 45 illustre ce comportement.

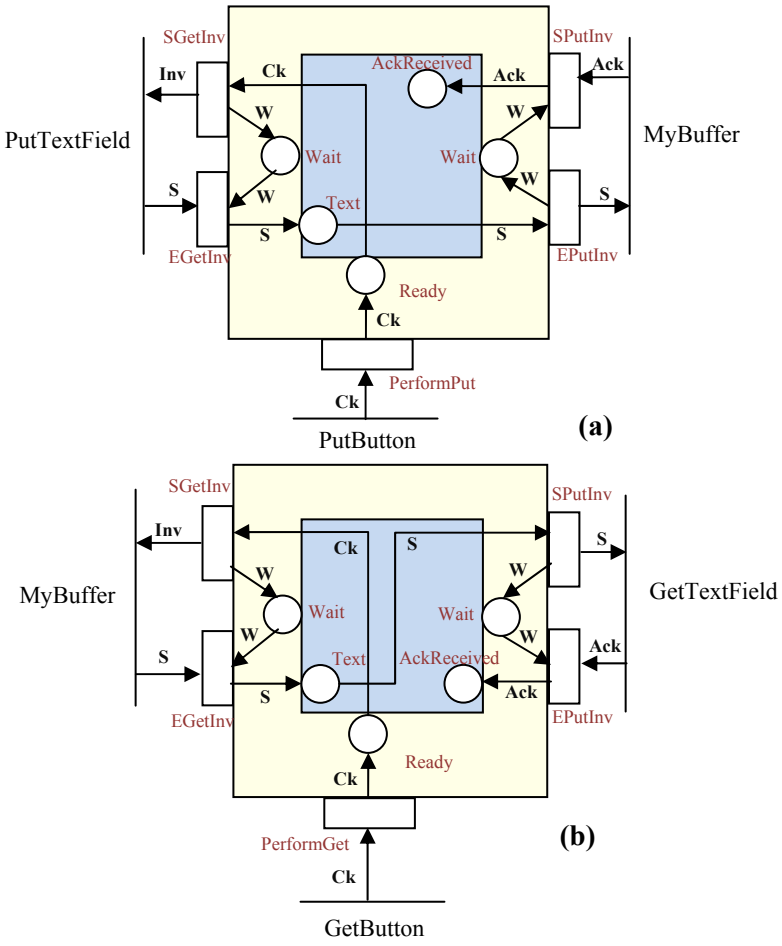


Fig.45 Spécification formelle du comportement du composant :
(a) PutAdapter, (b) GetAdapter

L'assemblage des composants 'ECATNets' est maintenant évident. Une vue externe de chaque composant de l'application de notre exemple et leur assemblage est formellement spécifiée par notre approche dans la figure 46.

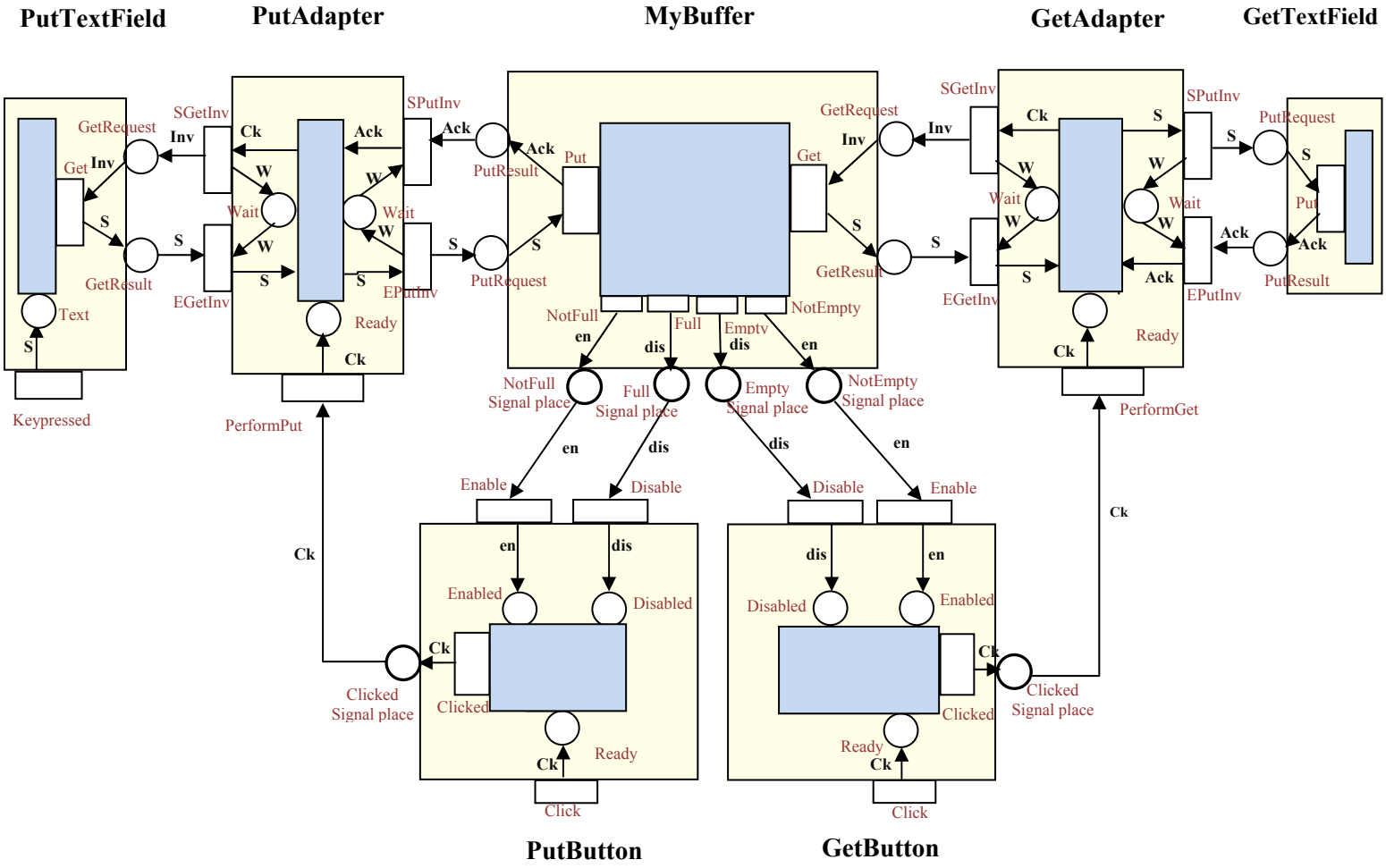


Fig.46 Assemblage formel des composants

Après avoir représenté la spécification formelle des connexions entre les interfaces de composants en utilisant les ECATNets, le détail du comportement interne de chaque composant dans l'assemblage est donné par la figure 47. Celle-ci illustre bien et offre une sémantique formelle du comportement de tout le système à base de composants. Le déroulement de l'exécution de ce système se fait on supposant un état initial. Le choix de cet état est arbitraire ; nous pouvons commencer par exemple avec le composant 'PutButton' inactif, un message dans le composant 'MyBuffer', et 'GetButton' actif. La place 'Enabled' doit contenir un jeton 'en'.

C'est ainsi que la phase de spécification du système est achevée, et l'application constituée d'un assemblage de composants a pût être modélisée.

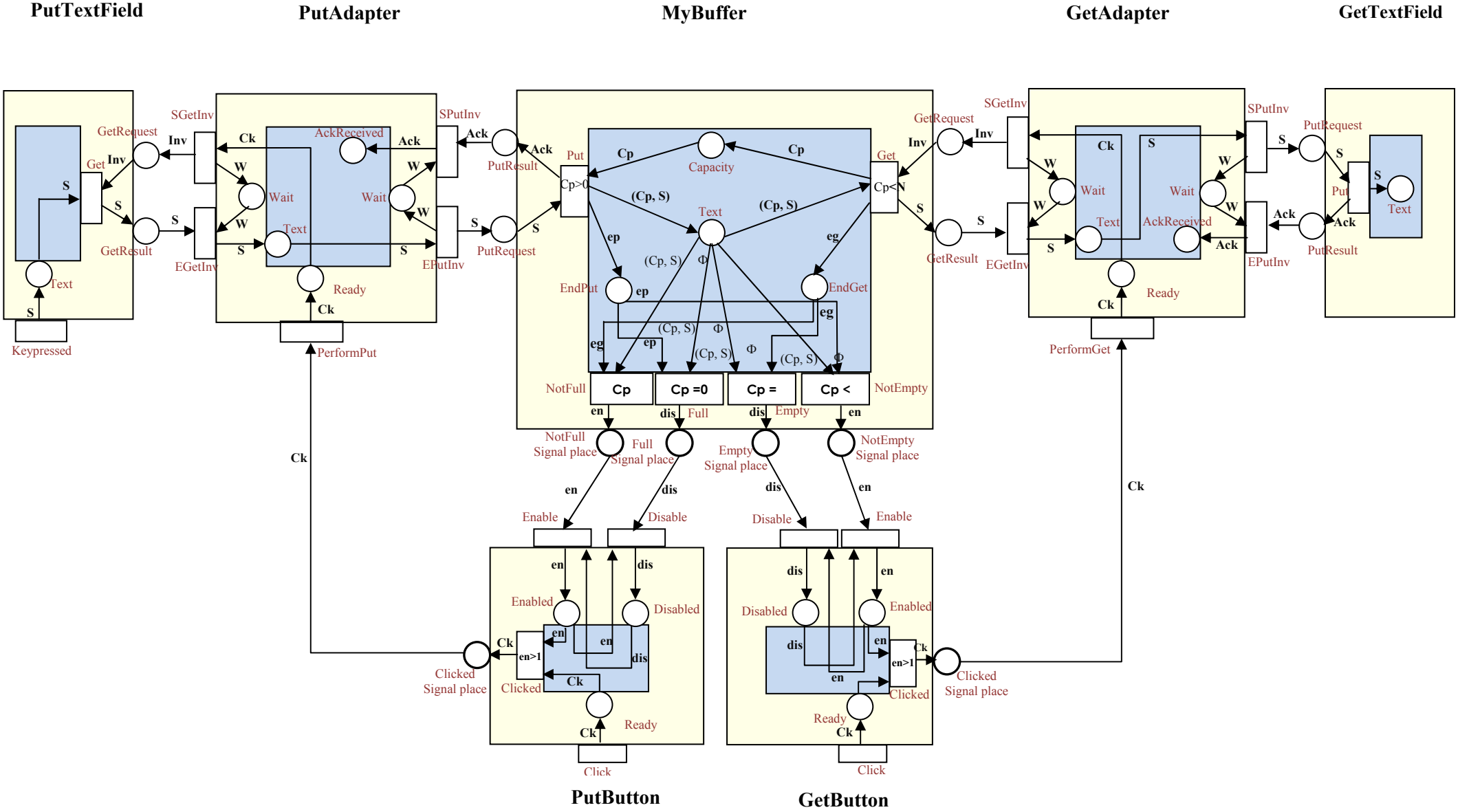


Fig.47 spécification comportementale formelle de l'assemblage de composants

4.2 Présentation de l'exemple 2

A travers cet exemple nous allons examiner une application de réservation de voyage, qui modélise un processus métier (Business Process). La description informelle du processus de l'application est comme suit [71] :

1) Obtenir l'itinéraire du client.
2) Pour chaque point de l'itinéraire, tenter de réserver avec la société cible. Plus précisément, réserver un vol avec la compagnie aérienne, une chambre à l'hôtel. Ces réservations peuvent être effectuées en parallèle.
3) Si toutes les réservations réussissent, obtenir le paiement du client et envoyer au client une confirmation. Processus terminé normalement.
4) Si au moins une réservation échoue, annuler les réservations effectuées et signaler le problème au client.

Pour un développeur de logiciels, cette liste est interprétée comme un algorithme: il s'agit d'une séquence d'étapes avec des conditions, des boucles et un peu de parallélisme. En effet, un processus soigneusement conçu est algorithmique et est souvent représenté sous forme de diagramme. La figure 48 représente un diagramme d'activité UML modélisant le processus de réservation de l'agence de voyage. Le diagramme inclus la location de voiture qu'on a omis dans notre modélisation pour simplifier.

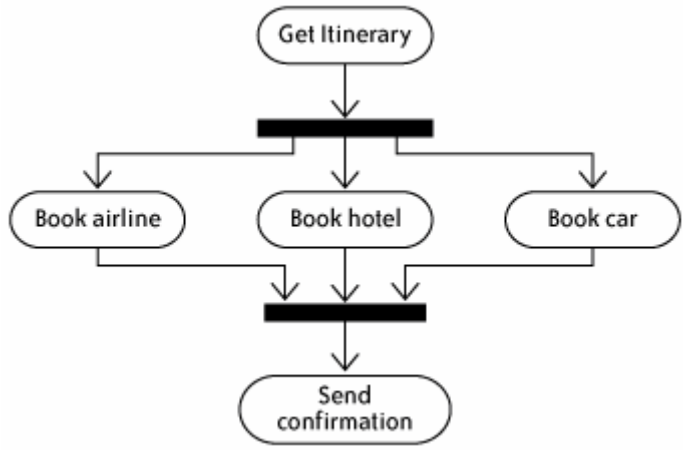


Fig.48 Diagramme d'activité UML du processus de réservation [50]

Nous avons identifié trois composants dans l'application qui modélise un système distribué. Un composant 'AgenceVoyage' qui joue le rôle de coordinateur, un composant 'ReservVol' qui assure la réservation d'un vol, et finalement un composant 'ReservHotel' qui assure la réservation d'une chambre d'hôtel. Les deux derniers composants représentent respectivement des agences de réservation de vols et d'hôtels. Le client soumet sa demande via l'application Web de l'agence de voyage qui va invoquée le composant coordinateur. Ce dernier invoque deux services en parallèle depuis les composants 'ReservVol' et 'ReservHotel'. Les interfaces qu'expose chaque composant doivent être identifiées à partir par exemple, d'une spécification en IDL3, une spécification

en langage Java, un diagramme de composants UML qui montre les interfaces de chaque composant, ou même une description intuitive en langage naturel.

4.2.1 Spécification du système

A. AgenceVoyage

Ce composant reçoit les requêtes de réservation de voyages des clients soumises via l'application Web de l'agence, il peut recevoir plusieurs demandes à la fois modélisées par la place 'ResRequest'. Le service est rendu par le biais d'une interface fournie par le composant. Le service est réellement la combinaison du résultat de deux services invoqués en parallèle (réceptacles) et rendu par les deux autres composants. Les clients sont notifiés du résultat de l'opération de réservation positivement ou négativement (événements sources) par des messages électroniques. La réponse de ce composant dépend des résultats signalés par les composants 'ReservHotel' et 'ReservVol' (événements puits). Le client fourni dans sa requête un identificateur, la destination de son voyage, et son numéro de carte bancaire ou carte de crédit (ce système est bien sur pas utilisé dans notre pays). Le composant fera usage de ces paramètres pour formuler des requêtes pour les deux autres composants, en invoquant leurs services 'SResVInv' et 'SResHInv' et en fournissant l'identificateur et la destination du client comme paramètres. D'après la description des fonctionnalités du composant qu'on a présenté, on pourra donner son modèle ECATNet spécifique (figure 49) :

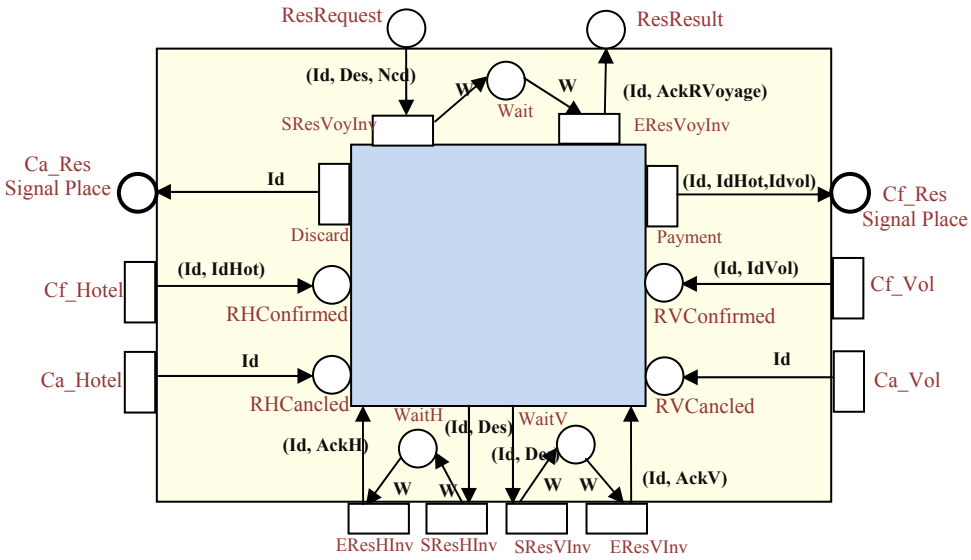


Fig.49 Modèle ECATNet du composant AgneceVoyage

On a utilisé dans cet exemple la notation générale pour modéliser le service offert par ce composant, puisqu'il fait appel à d'autres services pour accomplir sa tâche. Il se met en état de blocage jusqu'à réception d'acquittements des composants 'ReservHotel' et 'ReservVol' disant que l'opération de réservation du vol et de l'hôtel est en cours. A ce moment la, le composant 'AgenceVoyage' se débloque et retourne un acquittement au client pour lui dire que la réservation est en cours modélisé par la place 'ResResult'. Dans

le cas d'une réponse positive, le composant reçoit deux événements puits de confirmation 'Cf_Hotel' et 'Cf_Vol' et émet un événement source de notification du service de réservation rendu au client, modélisé par le franchissement de la transition 'Payement' et le dépôt d'un jeton dans la place signal 'Cf_Res'. Dans le cas échéant il reçoit 'Ca_Hotel' et 'Ca_Vol' et émet un événement source pour notifié un service non rendu pour le client représenté par le franchissement de la transition 'Discard' et le dépôt d'un jeton dans la place signal 'Ca_Res'.

Nous allons se restreindre à présenter seulement les modèles ECATNets de chaque composant. Leurs modèles spécifiant leurs comportements seront illustrés dans le modèle spécifiant l'assemblage des composants, puisqu'on a montré comment composé les composants dans le premier exemple.

B. ReservHotel

C'est un composant implémenté au niveau d'une agence de réservation d'hôtel. Il assure la réservation d'une chambre d'hôtel pour un client particulier en se basant sur les paramètres fournis par le composant 'AgenceVoyage' (un identificateur du client et la destination demandée). La demande du service est modélisée par la place 'ResHRequest'. L'acquittement est transmis informant le composant qui a invoqué le service que l'opération est en cours. Le composant consulte une base de données pour voir la disponibilité d'une chambre d'hôtel dans la destination sollicitée par le client. Dans le cas positif, une confirmation est envoyée par un événement source modélisé par le franchissement de la transition 'SuccessH' et le dépôt d'un jeton dans la place signal 'Confirm_H'. Le cas négatif (pas de chambre) est modélisé par l'envoi d'un événement source représenté par le franchissement de la transition 'FailH' et le dépôt d'un jeton dans la place signal 'Cancel_H'.

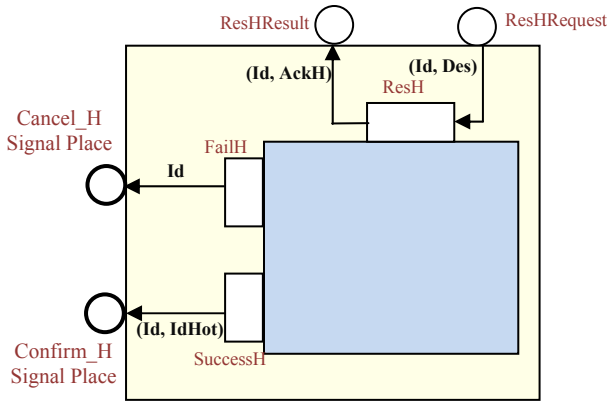


Fig.50 Modèle ECATNet du composant ReservHotel

C. ReservVol

C'est un composant implémenté au niveau d'une agence de réservation de vol. Il assure la réservation d'un billet de vol pour un client, dans une compagnie aérienne quelconque qui possède des places vides pour la destination souhaité. L'invocation du

service est modélisée par la place 'ResVRequest' ; le franchissement de la transition 'ResV' constitue le rendement du service acquitté dans la place 'ResHResult' pour le composant 'AgenceVoyage'. Le composant consulte la aussi la présence d'un place libre pour un vol quelconque dans une base de données. Le comportement de ce composant est identique à celui du composant 'ReservHotel'. En la présence d'une place libre dans un vol, une confirmation est envoyée par un événement source modélisé par le franchissement de la transition 'SuccessV' et le dépôt d'un jeton dans la place signal 'Confirm_V'. Le cas négatif est modélisé par l'envoi d'un événement source représenté par le franchissement de la transition 'FailV' et le dépôt d'un jeton dans la place signal 'Cancel_V' (figure 51).

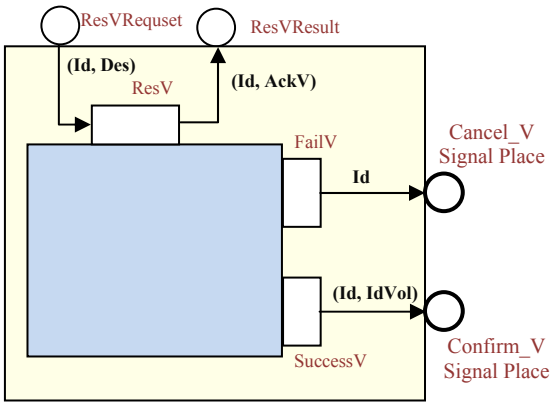


Fig.51 Modèle ECATNet du composant ReservVol

La spécification de l'assemblage des composants est donnée dans la figure 52, et la sémantique du comportement dynamique du système modélisé est représentée formellement dans la figure 53.

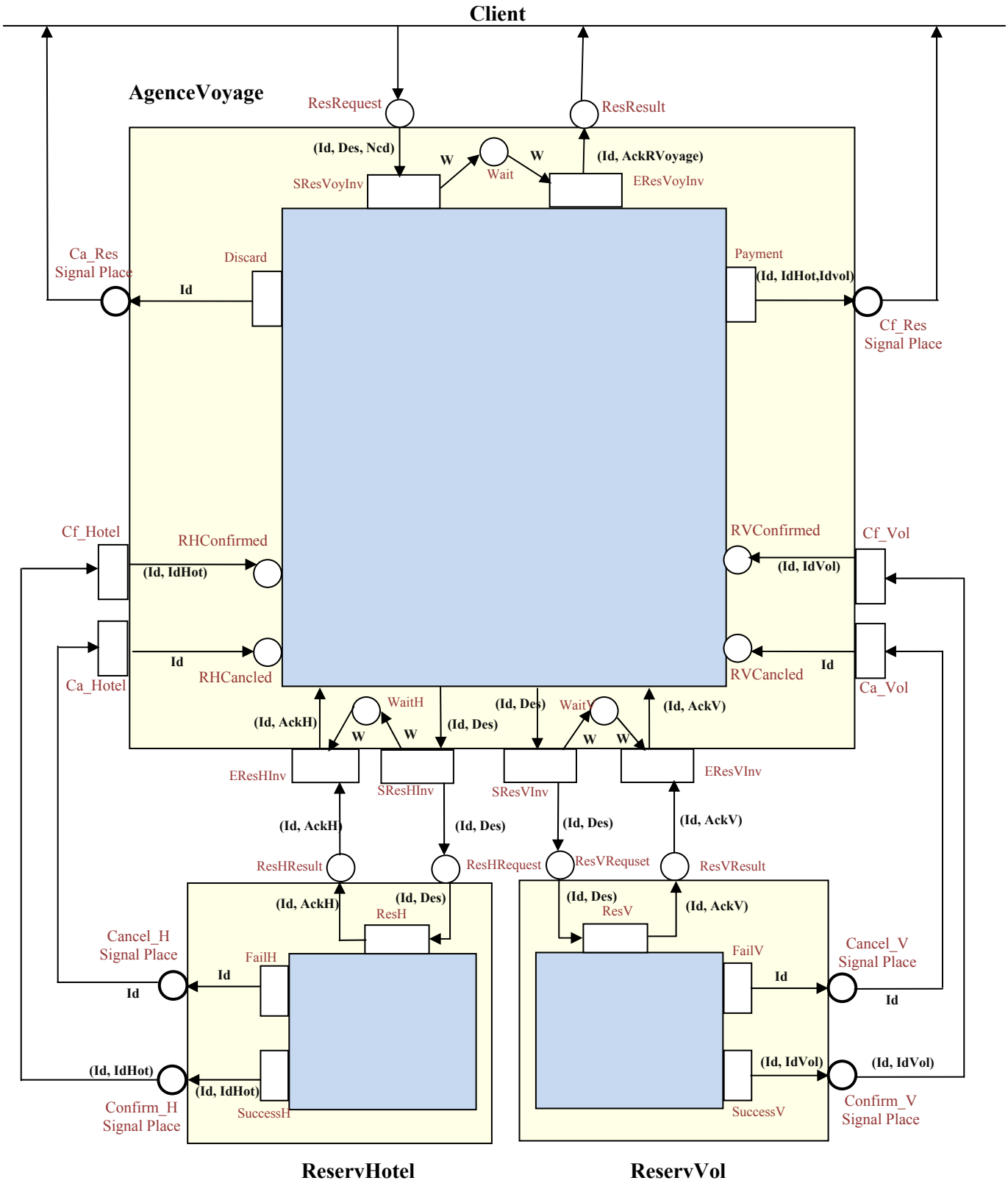


Fig.52 Assemblage formel des composants

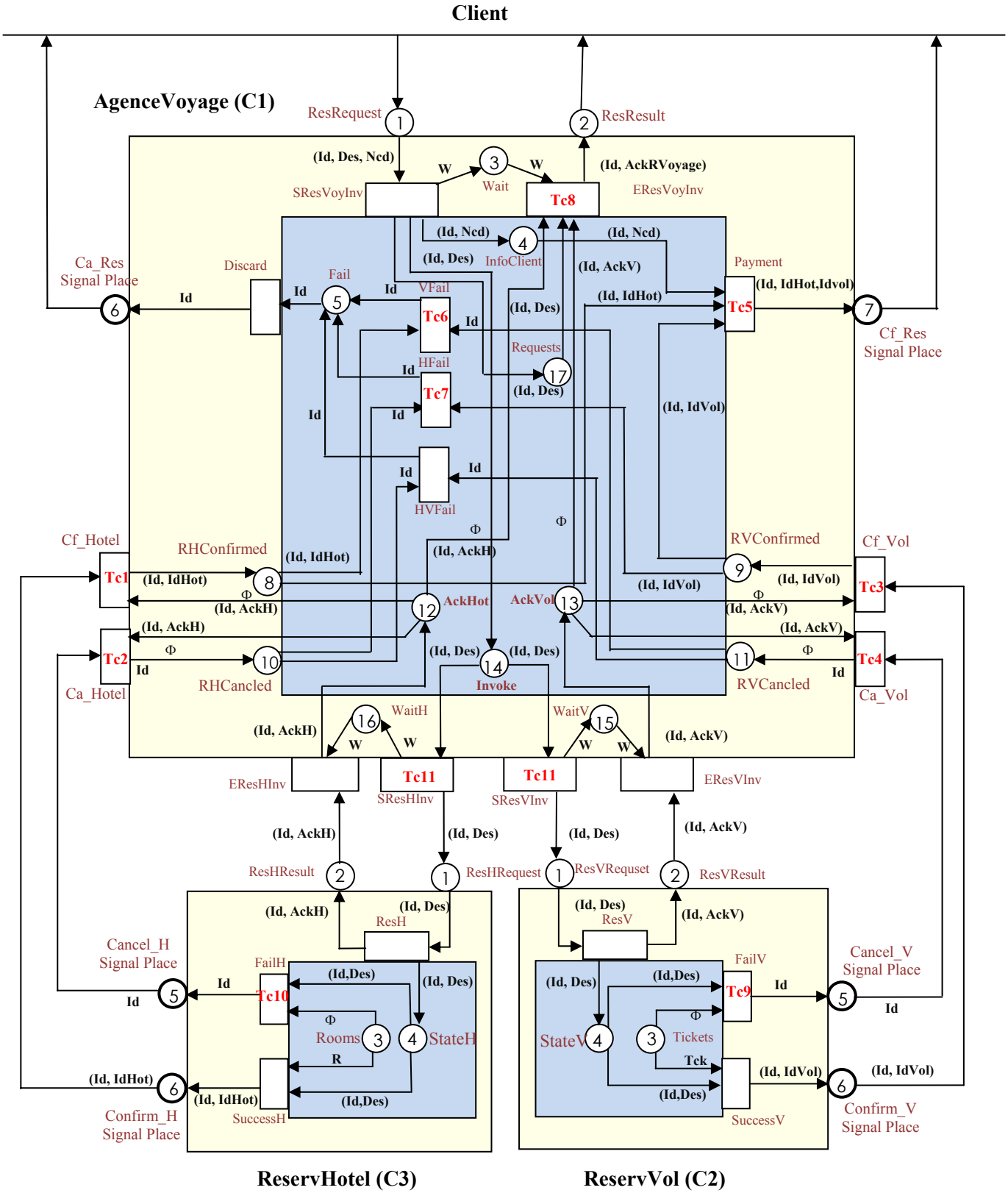


Fig.53 Spécification comportementale formelle de l'assemblage de composants

Le modèle donné dans la figure 53 modélise les requêtes du service de réservation de voyages de plusieurs clients en même temps. Celles-ci sont représentées par la présence de jetons <Id,Des,Ncd> dans la place 'ResRequest' ce qui provoque le déclanchement de l'opération de réservation modélisée par le franchissement de la transition 'SResVInv'. Le composant extrait ensuite les informations de chaque client et les stockent dans une base de données représentée par la place 'InfoClient', puis invoque en parallèle les deux services de réservation d'hôtel et de vol et se met en état de blocage en attente du résultat.

Grâce au formalisme des ECATNets on a pu exprimer le vrai parallélisme dans le franchissement des deux transitions 'SResVInv' et 'SResHInv' qui modélise les services requis par le composant 'AgenceVoyage'. Cela est exprimé sémantiquement par la métarègle **(R2')** abordée dans le chapitre 2 (section 5.3, cas 2). La présence d'un jeton (Id, Des) dans la place 'Invoke' suffira pour exécuter les deux transitions en parallèle est d'invoqué les services de réservation de l'hôtel et de vol. La présence de jetons (Id, Des) dans les places 'ResHRequest' et 'ResVRequest' déclanche le processus de réservation de vol et de l'hôtel par le franchissement des transitions 'ResV' et 'ResH'. Le composant 'AgenceVoyage' est acquitté via les places 'ResHResult' et 'ResVResult' du composant 'ReservHotel' et 'ReservVol' par deux jetons de termes algébriques composés (Id, AckH) et (Id, AckV). Le dépôt de ces derniers dans les places 'AckHot' et 'AckVol' respectivement permet le franchissement de la transition 'EResVoyInv' est débloque le composant 'AgenceVoyage'. Les clients sont acquittés par des messages leurs indiquant que leurs requêtes sont en cours, cela est modélisé par le dépôt des jetons dans la place 'ResResult'.

Parallèlement et indépendamment, les composants 'ReservHotel' et 'ReservVol' consulte la disponibilité de chambres d'hôtel et de place vide dans un vol dans leurs bases de données modélisées respectivement par les places 'Rooms' et 'Tickets'. Suite à la réception d'événements sources de confirmation, le composant 'AgenceVoyage' fait le payement à partir des informations préalablement stockées sur les clients, puis notifie les clients par des messages électroniques. Le cas échéant, i.e. la réception d'événements sources d'annulation est modélisé par l'absence de jetons dans les places 'Rooms' et 'Tickets'. Le comportement du composant 'AgenceVoyage' qui en résulte est l'annulation du service de réservation de voyage, par l'envoi d'un événement source représenté par la transition 'Discard' et la place 'Ca_Res'. Ce dernier véhicule un message au client concerné parmi ceux qui ont soumis des demandes de réservation pour l'aviser que le service n'a pas pu aboutir.

Les conditions de transitions (TC) on été ajouté pour assuré que seules les requêtes de clients qui ont été traitées et acquittées par les composants 'ReservHotel' et 'ReservVol' peuvent être traitées par le composant 'AgenceVoyage'. De cette façon on pourra modéliser correctement un nombre important de demandes de clients de manière très compacte. Le modèle de données du formalisme des ECATNets qui est un formalisme algébrique permet aussi de définir des fonctions et enrichir la sémantique du modèle. head (param1, param2) est une fonction qui retourne le premier paramètre, et permettra ainsi de

comparer les identificateurs de clients ayant des demandes acquittés et traités pour pouvoir par la suite les notifiés.

Tc1: If (head(Id, IdHot) = head(Id, AckH))
Tc2: If (Id = head(Id, AckH))
Tc3: If (head(Id, IdVol) = head(Id,AckV))
Tc4: If (Id = head(Id, AckV))
Tc5: If (head(Id, IdHot) = head(Id, IdVol) and head (Id, IdHot) = head(Id, Ncd))
Tc6: If (Id= head(Id, IdHot))
Tc7: If (Id= head(Id, IdVol))
Tc8: If (head(Id, AckV) = head(Id, AckH) and head (Id, AckV) = head(Id, Des))
Tc9: If (M(Tickets) →ϕ)
Tc10: If (M(Rooms) →ϕ)
Tc11: If (M(Invoke) → <Id, Des>)
head (param1,param2): une fonction qui retourne le premier paramètre.

4.2.2 Génération des règles de réécriture

Cette phase consiste à exprimé le comportement dynamique du système et capturer le parallélisme inhérent dans les composants distribués. Cela est spécifié en termes de règles de réécritures générées par l'application des métarègles abordée dans le deuxième chapitre.

Les règles de réécriture

[SResVoyInv] :(ResRequest,<Id,Des, Ncd>)→ (InfoClient,<Id, Ncd>) ⊗ (Invoke,<Id, Des>) ⊗ (Wait,W) ⊗ (Requests,<Id, Des>) ***R1***

[EResVoyInv]:(AckHot,<Id,AckH>)⊗(AckVol,<Id,AckV>)⊗(Requests,<Id,Des>)⊗ (Wait,W)
→ (ResResult,<Id,AckRVoyage>)⊗(AckHot,<Id,AckH>)⊗(AckVol,<Id,AckV>)
if [(head(Id, AckV)= head(Id, AckH) and head (Id, AckV)=head(Id, Des))] → true.
R1 et R2 et R6
[Discard]: (Fail, Id)→ (Ca_Res, Id) ***R1***

[Payement]:(InfoClient,<Id,Ncd>)⊗(RVConfirmed,<Id,IdVol>)

⊗(RHConfirmed,<Id,IdHot>)→(Cf_Res,<Id, IdHot, IdVol>)

if [(head(Id, IdHot) = head(Id, IdVol) and head (Id, IdHot) = head(Id, Ncd))] → true. ***R1 et R6***

[VFail]: (RHConfirmed,<Id, IdHot>) ⊗(RVCancled, Id) → (Fail, Id)

if [(Id= head(Id, IdHot))] → true . ***R1 et R6***

[HFail]: (RVConfirmed,<Id, IdVol>) ⊗(RHCancled, Id) → (Fail, Id)

if [(Id= head(Id, IdVol))] → true . ***R1 et R6***

[HVFail]: (RVCancled,Id) ⊗(RHCancled, Id) → (Fail, Id) ***R1***

[Cf_Hotel]:(AckHot,<Id, AckH>) ⊗(Confirm_H,<Id, IdHot>) →(AckHot,<Id, AckH>) ⊗ (RHConfirmed,<Id,IdHot>) if [(head(Id, IdHot) = head(Id, AckH))] → true. ***R1, R2 et R6***

[Ca_Hotel]: (AckHot,<Id, AckH>) ⊗(Cancel_H, Id) → (AckHot,<Id, AckH>) ⊗ (RHCancled, Id) if [(Id = head(Id, AckH))] → true. ***R1, R2 et R6***

[Cf_Vol]: (AckVol,<Id, AckV>) ⊗(Confirm_V,<Id, IdVol>) →(AckVol,<Id, AckV>) ⊗ (RVConfirmed,<Id, IdVol>)
if [(head(Id, IdVol) = head(Id,AckV))] ***R1, R2 et R6***

[Ca_Vol]: (AckVol,<Id, AckV>) ⊗(Confirm_V,<Id, IdVol>) → (AckVol,<Id, AckV>) ⊗ (RVCancled, Id) if (Id = head(Id, AckV)) ***R1, R2 et R6***

[SResVInv]:(Invoke, ϕ)→(WaitV,W)⊗(ResVRequest,<Id, Des>) if (M(Invoke)→<Id,Des>)***R2'***

[SResHInv]:(Invoke, ϕ)→(WaitH,W)⊗(ResHRequest,<Id, Des>) if (M(Invoke)→<Id,Des>)***R2'***

[EResVInv]: (WaitV, W) ⊗ (ResVResult,<Id, AckV>) → (AckVol,<Id, AckV>) ***R1***

[EResHInv]: (WaitH, W) ⊗ (ResHResult,<Id, AckH>) → (AckHot,<Id, AckH>) ***R1***

[ResV]: (ResVRequest, <Id, Des>) → (ResVResult,<Id, AckV>) ⊗ (StateV,<Id, Des>) ***R1***

[FailV]: (StateV, <Id,Des>) ⊗ (Tickets, ϕ) → (Cancel_V, Id) if (M(Tickets) → ϕ) ***R5 et R1***

[SuccessV]: (StateV, <Id,Des>) ⊗ (Tickets, Tck) → (Confirm_V, <Id, IdVol>) ***R1***

[ResH]: (ResHRequest, <Id, Des>) → (ResHResult,<Id, AckH>) ⊗ (StateH,<Id, Des>) ***R1***

[FailH]: (StateH, <Id,Des>) ⊗ (Rooms, ϕ) → (Cancel_H, Id) if (M(Rooms) → ϕ) ***R5 et R1***

[SuccessH]: (StateH, <Id,Des>) ⊗ (Rooms, R) → (Confirm_H, <Id, IdHot>) ***R1***

4.2.3 Vérification du système

Cette phase consiste en la vérification des propriétés spécifiées par le concepteur du système modélisé. Une diversité de propriétés peut être vérifiée dans une application à base de composants. Nous avons choisit de présenté les propriétés qui nous on semblé les plus intéressantes. Pour réaliser la vérification de notre système présenté dans le deuxième exemple, nous avons modifié et enrichi le module fonctionnel présenté dans la section 3.3, avec des opérations supplémentaires (figure 54).

```
fmod ECATNET is
protecting STRING .
sorts Place Marking TOKENS token elem .
subsort String < Place .
subsort String < elem .
subsort token < TOKENS .
op none : -> elem  [ctor] .
op _-_ : elem elem -> elem [ctor comm assoc ] .
op T(_) : elem -> token .
op nill : -> token [ctor] .
op _;_ : token token -> token [ctor comm assoc id: nill] .
op mt : -> Marking [ctor] .
op __ : Marking Marking -> Marking [ctor comm assoc id: mt] .
op <_|_> : Place TOKENS -> Marking .
op init : -> Marking .
endfm
```

Fig.54 Spécification en Maude du module fonctionnel d'un composant ECATNets

Nous avons suivis la notation suivante dans la représentation de la sorte 'place' pour avoir une représentation plus lisible de notre spécification et moins encombrée : 'PiCi', ce qui signifie, la place i du composant i; tel que i ∈ N. Nous devons donc numéroter les

composants de notre exemple ainsi que leurs places. Dans notre cas nous avons trois composants (C1, C2, C3) représentant respectivement 'AgenceVoyage', 'ReservVol', 'ReservHotel'. Nous avons représenté donc l'aspect statique du système, il reste maintenant à représenter l'aspect dynamique par un module système.

```
mod ECATNET-COMPONENT is
        including ECATNET
vars X X1 X2 X3 X4 X5 : TOKENS . vars  Id Des Ncd IdHot IdVol Tck R : elem . var M : Marking . var T : String .
rl [SResVoyInv] : < "P17C1" | X4 > < "P1C1" | T( Id - Des - Ncd ) ; X > < "P4C1" | X1 > < "P14C1" | X2 > < "P3C1"
| X3 > => < "P1C1" | X > < "P4C1" | T( Id - Ncd ) ; X1 > < "P14C1" | T( Id - Des ) ; X2 > < "P3C1" | T("W") ;  X3 >
 < "P17C1" | T( Id - Des ) ; X4 > .
rl [EResVoyInv] : < "P17C1" | T( Id - Des ) ; X2 > < "P12C1" | T( Id - "AckH" ) ; X > < "P13C1" | T( Id - "AckV" ) ;
X1 > < "P3C1" | T("W") ; X3 > < "P2C1" | X4 > => < "P2C1" | T( Id - "AckRVoyage" ) ; X4 >
< "P12C1" | T( Id - "AckH" ) ; X > < "P13C1" | T( Id - "AckV" ) ; X1 > < "P3C1" | X3 > < "P17C1" | X2 > .
rl [VFail] : < "P8C1" | T( Id - IdHot ) ; X > < "P11C1" | T(Id) ; X1 > < "P5C1" | X2 > => < "P5C1" | T(Id) ; X2 >
 < "P8C1" | X > < "P11C1" | X1 > .
rl [Discard] : < "P5C1" | T(Id) ; X > < "P6C1" | X1 > => < "P6C1" | T(Id) ; X1 > < "P5C1" | X > .
rl [Payement] : < "P4C1" | T( Id - Ncd ) ; X > < "P9C1" | T( Id - IdVol ) ; X1 > < "P8C1" | T( Id - IdHot ) ; X2 >
 < "P7C1" | X3 > => < "P4C1" | X > < "P9C1" | X1 > < "P8C1" | X2 > < "P7C1" | T( Id - IdHot - IdVol ) ; X3 > .
rl [HFail]: < "P9C1" | T( Id - IdVol) ; X > < "P10C1" | T(Id) ; X1 > < "P5C1" | X2 > =>
 < "P9C1" | X > < "P10C1" | X1 > < "P5C1" | T(Id) ; X2 > .
rl [HVFail]: < "P11C1" | T(Id) ; X > < "P10C1" | T(Id) ; X1 > < "P5C1" | X2 > => < "P11C1" | X > < "P10C1" | X1 >
< "P5C1" | T(Id) ; X2 > .
rl [Cf_Hotel]: < "P12C1" | T( Id - "AckH" ) ; X > < "P6C3" | T( Id - IdHot ) ; X1 > < "P8C1" | X2 > =>
 < "P12C1" | T( Id - "AckH" ) ; X > < "P6C3" | X1 > < "P8C1" | T( Id - IdHot ) ; X2 > .
rl [Ca_Hotel]: < "P12C1" | T( Id - "AckH" ) ; X > < "P5C3" | T(Id) ; X1 > < "P10C1" | X2 >  =>
 < "P12C1" | T( Id - "AckH" ) ; X > < "P5C3" | X1 > < "P10C1" | T(Id) ; X2 > .
rl [Cf_Vol]: < "P13C1" | T( Id - "AckV" ) ; X > < "P6C2" | T( Id - IdVol ) ; X1 > < "P9C1" | X2 > =>
 < "P13C1" | T( Id - "AckV" ) ; X > < "P6C2" | X1 > < "P9C1" | T( Id - IdVol ) ; X2 > .
rl [Ca_Vol]: < "P13C1" | T( Id - "AckV" ) ; X > < "P5C2" | T(Id) ; X1 > < "P11C1" | X2 > =>
 < "P13C1" | T( Id - "AckV" ) ; X > < "P5C2" | X1 > < "P11C1" | T(Id) ; X2 > .
rl [SResVInv-SResHInv]: < "P14C1" | T( Id - Des ) ; X > < "P15C1" | X1 > < "P16C1" | X2 > < "P1C2" | X3 >
 < "P1C3" | X4 >  => < "P14C1" | X > < "P15C1" | T("W") ; X1 > < "P16C1" | T("W") ; X2 >
 < "P1C2" | T( Id - Des ) ; X3 > < "P1C3" | T( Id - Des ) ; X4 > .
rl [EResVInv]: < "P15C1" | T("W") ; X > < "P2C2" | T( Id - "AckV" ) ; X1 > < "P13C1" | X2 > =>
 < "P15C1" | X > < "P2C2" | X1 > < "P13C1" | T( Id - "AckV" ) ; X2 > .
rl [EResHInv]: < "P16C1" | T("W") ; X > < "P2C3" | T( Id - "AckH" ) ; X1 > < "P12C1" | X2 > =>
< "P16C1" | X > < "P2C3" | X1 > < "P12C1" | T( Id - "AckH" ) ; X2 > .
rl [ResV]: < "P1C2" | T( Id - Des ) ; X > < "P2C2" | X1 > < "P4C2" | X2 > =>
< "P1C2" | X > < "P2C2" | T( Id - "AckV" ) ; X1 > < "P4C2" | T( Id - Des ) ; X2 > .
rl [FailV]: < "P4C2" | T( Id - Des ) ; X > < "P3C2" | nill > < "P5C2" | X1 > => < "P4C2" | X > < "P3C2" | nill >
 < "P5C2" | T(Id) ; X1 > .
rl [SuccessV]: < "P4C2" | T( Id - Des ) ; X > < "P3C2" | T(Tck) ; X1 > < "P6C2" | X2 > =>
 < "P4C2" | X > < "P3C2" | X1 > < "P6C2" | T( Id - "IdVol" ) ; X2 > .
rl [ResH]: < "P1C3" | T( Id - Des ) ; X > < "P2C3" | X1 > < "P4C3" | X2 > =>
 < "P1C3" | X > < "P2C3" | T( Id - "AckH" ) ; X1 > < "P4C3" | T( Id - Des ) ; X2 > .
rl [FailH]: < "P4C3" | T( Id - Des ) ; X > < "P3C3" | nill > < "P5C3" | X1 > => < "P4C3" | X > < "P3C3" | nill >
< "P5C3" | T(Id) ; X1 > .
rl [SuccessH]: < "P4C3" | T( Id - Des ) ; X > < "P3C3" | T(R) ; X1 > < "P6C3" | X2 > =>
 < "P4C3" | X > < "P3C3" | X1 > < "P6C3" | T( Id - "IdHot" ) ; X2 > .
eq init = < "P1C1" | T( "id" - "Des" - "Ncd" ) ; T( "id2" - "Des2" - "Ncd2" ) ; T( "id3" - "Des3" - "Ncd3" ) >
< "P3C2" | T("Tck1") ; T("Tck2") > < "P3C3" | T("R1") ; T("R2") > < "P4C1" | nill > < "P14C1" | nill > < "P3C1" | nill
> < "P10C1" | nill > < "P11C1" | nill > < "P12C1" | nill > < "P13C1" | nill > < "P14C1" | nill > < "P15C1" | nill >
 < "P16C1" | nill > < "P17C1" | nill > < "P1C2" | nill > < "P1C3" | nill > < "P2C1" | nill > < "P2C2" | nill > < "P2C3" |
nill > < "P4C2" | nill > < "P4C3" | nill > < "P5C1" | nill > < "P5C2" | nill > < "P5C3" | nill > < "P6C1" | nill >
< "P6C2" | nill > < "P6C3" | nill > < "P7C1" | nill > < "P8C1" | nill > < "P9C1" | nill > .
endm
```

Fig.55 Spécification en Maude du module système de l'application

Ce dernier inclus outre le module fonctionnel défini plus haut, les règles de réécriture générées à partir du modèle de composants ECATNets.

Maude offre différentes façons d'exécuter une spécification tels que :

- Les commandes Maude **rew** (rewrite) et **frew** (frewrite) exécute (ou simule) l'un des nombreux comportements à partir d'un état initial donné, par l'application des règles de réécriture à l'état initial.
- Maude fournit une commande de recherche **search** pour la recherche de tous les comportements possibles à partir d'un état initial donné. Cela se fait par la recherche des termes qui peuvent être atteints depuis l'état initial et qui satisfont une condition donnée.
- Maude est doté d'un modèle Checking de logique temporel pour vérifier si tous les comportements à partir d'un certain état initial satisfait une propriété de logique temporelle.

Nous allons montrer dans ce qui suit comment peut on vérifiés des propriétés comportementales du système qu'on a modélisé, en utilisant le model de spécification proposé plus haut dans ce chapitre. La vérification se fera évidemment par la spécification en Maude représentée par la figure (56, 57). Nous avons mentionné dans le deuxième chapitre que le problème d'atteignabilité consiste à vérifié s'il existe une séquence de transitions qui transforme M_0 (marquage initial) en un marquage M. Nous allons utiliser cette propriété pour montrer que le système va répondre exactement comme prévus suite à la réception de requêtes de clients. Considérons l'état initial suivant :

- Trois demandes de réservations représentées dans le modèle par trois jetons de type (Id, Des, Ncd) dans la place 'P1C1'.
- Deux billets d'avion dans les vols désirés (deux jetons de type (Tck) dans la place 'P3C2').
- Deux chambres d'hôtel dans les destinations souhaitées (deux jetons de type (R) dans la place 'P3C3').
- Les autres places resteront vides, elles contiennent donc des marquages 'nill'.

L'état initial 'M_0' est représenté en Maude par l'équation : eq init = < "P1C1" | T("id" - "Des" - "Ncd") ; T("id2" - "Des2" - "Ncd2") ; T("id3" - "Des3" - "Ncd3") > < "P3C2" | T("Tck1") ; T("Tck2") > < "P3C3" | T("R1") ; T("R2") > < "P4C1" | nill > < "P14C1" | nill > < "P3C1" | nill > < "P10C1" | nill > < "P11C1" | nill > < "P12C1" | nill > < "P13C1" | nill > < "P14C1" | nill > < "P15C1" | nill > < "P16C1" | nill > < "P17C1" | nill > < "P1C2" | nill > < "P1C3" | nill > < "P2C1" | nill > < "P2C2" | nill > < "P2C3" | nill > < "P4C2" | nill > < "P4C3" | nill > < "P5C1" | nill > < "P5C2" | nill > < "P5C3" | nill > < "P6C1" | nill > < "P6C2" | nill > < "P6C3" | nill > < "P7C1" | nill > < "P8C1" | nill > < "P9C1" | nill > .

Le système ne peut satisfaire que deux demandes de réservations. L'état que nous devons atteindre est évidemment, deux réponses de confirmations, une réponse

d'annulation ainsi que trois accusés de réception envoyés aux clients qui signifient que leurs requêtes sont en cours. Deux jetons de type (Id, IdHot, Idvol) dans la place 'P7C1', un jeton dans la place 'P6C1' de type (Id), et trois jetons de type (Id - "AckRVoyage") doivent être présent.

Nous avons réunis la spécification des deux modules fonctionnel et système représentés respectivement par les figures (54, 55), dans un module appelé 'ECATNET-COMPONENT'. Ce dernier est sauvegardé dans un fichier nommé ECATNET. Pour exécuter la spécification nous devons tout d'abord charger le fichier qui la contient en utilisant la commande '**load**' (figure 56).

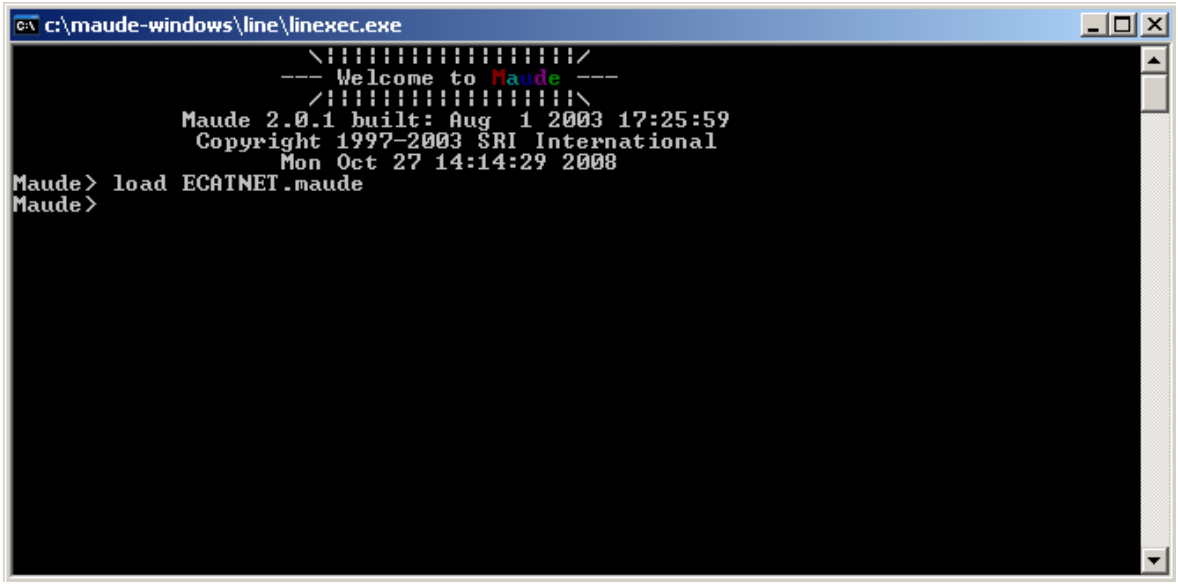


Fig.56 Chargement de la spécification par la commande 'load'.

L'exécution de la spécification se fait à l'aide de la commande '**rew**' ou '**frew**' en partant de l'état initial '**init**'.

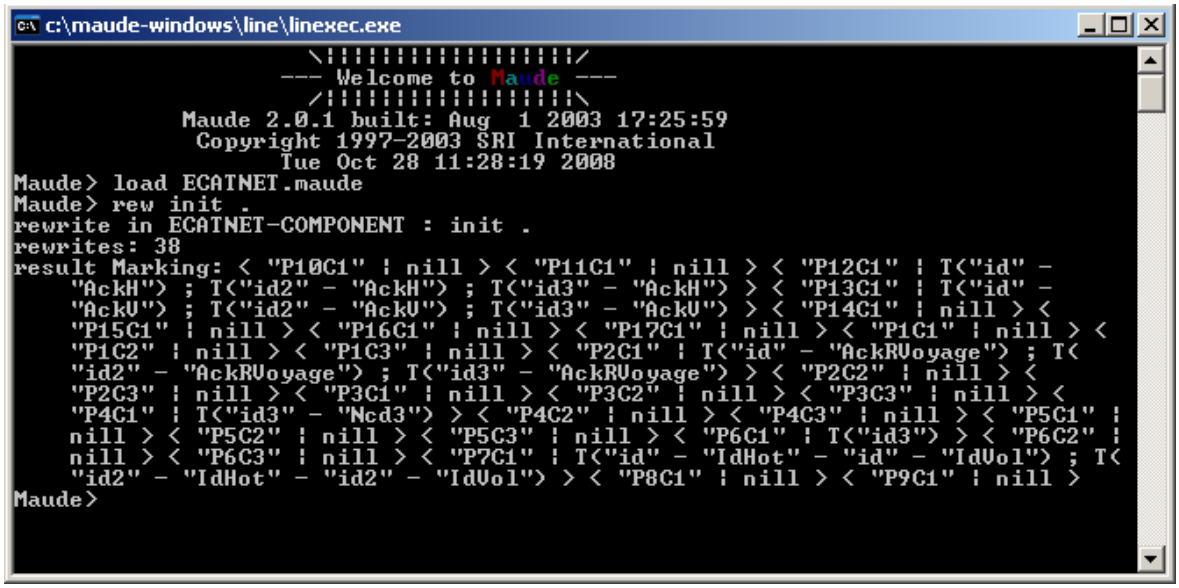


Fig.57 Résultat d'exécution de la spécification.

La commande '**rew**' (abréviation de 'rewrite') exécute les règles qui s'appliquent avec l'état initial '**init**', en substituant la partie gauche de ces règles par la partie droite. Prenant par exemple la règle :

rl [SResVoyInv] : < "P17C1" | X4 > < "P1C1" | T(Id - Des - Ncd) ; X > < "P4C1" | X1 > < "P14C1" | X2 > < "P3C1" | X3 > => < "P1C1" | X > < "P4C1" | T(Id - Ncd) ; X1 > < "P14C1" | T(Id - Des) ; X2 > < "P3C1" | T("W") ; X3 > < "P17C1" | T(Id - Des) ; X4 > .

La partie gauche de cette règle contient une place initialisée qui est 'P1C1'. Maude fait se qu'on appel le 'matching' de l'état initial avec cette règle, est fait les remplacement suivant : 'X' arbitrairement par 'T ("id2" - "Des2" - "Ncd2") ; T("id3" - "Des3" - "Ncd3")' ; Id : "id" ; Des : "Des" ; Ncd : "Ncd" ; X1, X2, X3, X4 : 'nill'. Le résultat de l'application de la règle constitue une réécriture et sera comme suit :

< "P17C1" | nill > < "P1C1" | T("id" - "Des" - "Ncd") ; T("id2" - "Des2" - "Ncd2") ; T("id3" - "Des3" - "Ncd3") > < "P4C1" | nill > < "P14C1" | nill > < "P3C1" | nill > => < "P1C1" | T("id2" - "Des2" - "Ncd2") ; T("id3" - "Des3" - "Ncd3") > < "P4C1" | T("id" - "Ncd") ; nill > < "P14C1" | T("id"- "Des") ; nill > < "P3C1" | T("W") ; nill > < "P17C1" | T("id"- "Des") ; nill >. Et ainsi de suite pour le reste des règles. Nous remarquons que Maude a réalisé '38' réécriture pour atteindre l'état final 'M' illustré par la figure 57. Nous avons pu donc, vérifier qu'il existe un franchissement d'une séquence de transition qui a transformé le marquage initial 'M_0' en un marquage final 'M'. Pour pouvoir explorer tous les chemins possibles qui ont mené à l'état final, et analyser les éventuels comportements qui peuvent se produire nous pouvons utiliser la puissance de la commande '**search**'. La figure 58 montre le résultat d'exécution de cette commande. Nous avons démarré avec un état initial très simple '**init**' qui est le suivant :

- Une demande de réservation représentée dans le modèle par un jeton de type (Id, Des, Ncd) dans la place 'P1C1'.
- Un billet d'avion dans le vol désiré (Un jeton de type (Tck) dans la place 'P3C2').
- Une chambre d'hôtel dans la destination souhaitée (Un jeton de type (R) dans la place 'P3C3').
- Les autres places resteront vides avec des marquages 'nill'.

L'état final que l'on prévoit atteindre est exprimé après le signe '=>' de la commande. Le signe '*' signifie que l'on cherche tous les chemins possibles à partir de l'état initial.

La commande a appliqué est comme suit :

search init =>* (< "P12C1" | T("id" - "AckH") > < "P13C1" | T("id" - "AckV") > < "P7C1" | T("id" - "IdHot" - "id" - "IdVol") > < "P2C1" | T("id" - "AckRVoyage") > X:Marking) .

Nous voulons étudier les comportements possibles que le système peut exhiber. Dans notre cas le système montre un seul comportement exprimé par la solution 1 (figure 58).

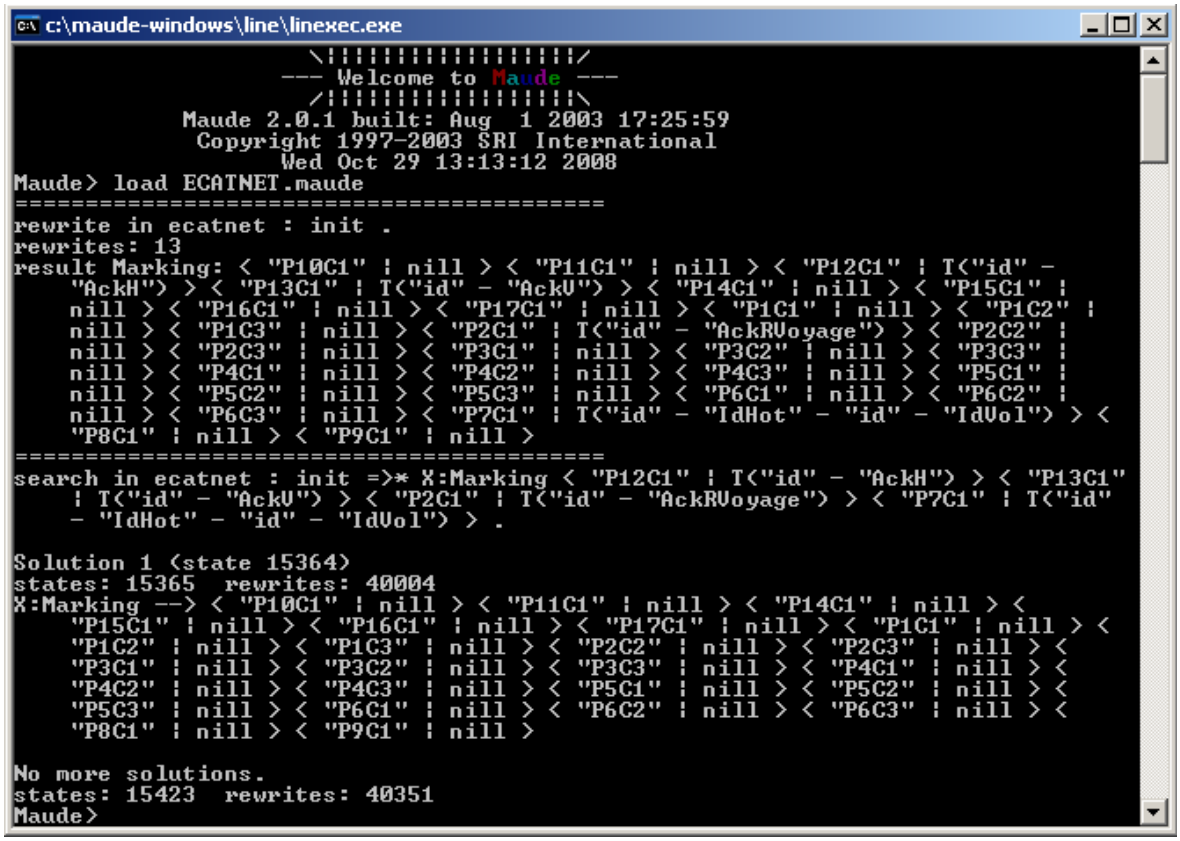


Fig.58 Résultat d'exécution de la commande 'search'.

Pour analyser le chemin emprunter par le système et voir la séquence de réécriture qui a permis d'arriver à l'état affiché, nous utilisons la commande '**show path**' en fournissant l'état '15364'. La séquence de franchissement emprunter par le système est représentée par la figure 59 :

[SResVoyInv], [SResVInv-SResHInv], [ResV], [EResVInv], [SuccessV], [Cf_Vol], [ResH], [EResHInv], [EResVoyInv], [SuccessH], [Cf_Hotel], [Payement].

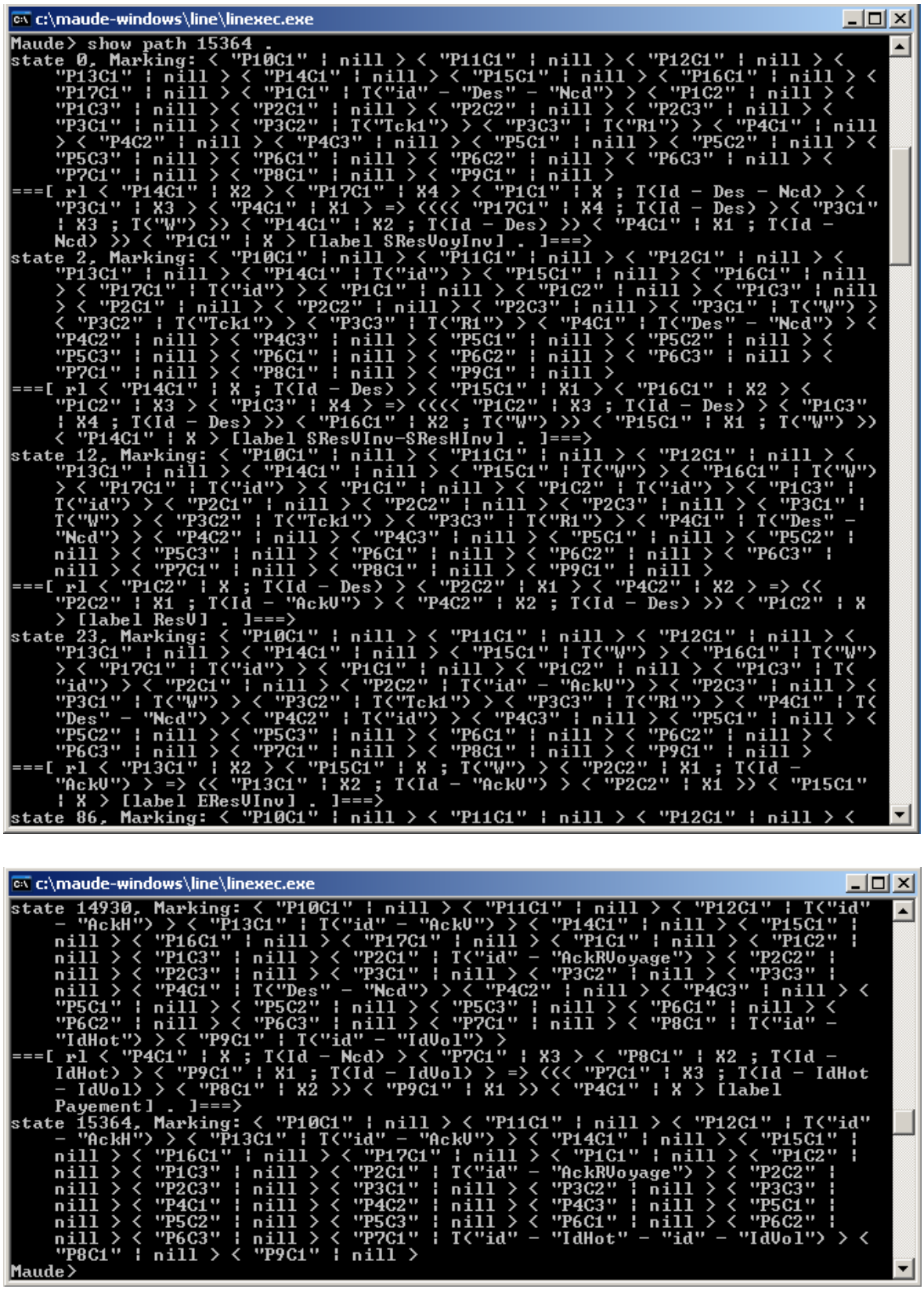


Fig.59 Résultat d'exécution de la commande 'show path'.

5. Conclusion

Nous avons présenté dans ce chapitre l'application de notre approche pour la modélisation des logiciels à base de composants. Cela a été réalisé en utilisant deux exemples différents issus de deux pratiques différentes. A travers le premier qui est tiré du domaine des interfaces utilisateurs, nous avons montré la manière de spécifier les composants logiciels ainsi que de leurs assemblages en faisant usage du model de spécification proposé. L'exemple modélise une simple application réalisée en assemblant des composants visuels et non visuels.

Le deuxième exemple modélise un processus métier d'une agence de voyage. On a montré comment spécifié le comportement dynamique des composants, ainsi que celui de l'assemblage des composants par des règles de réécritures générées à partir du modèle ECATNet qui modélise le système entier. On a aussi présenté comment vérifier les propriétés du système que le concepteur souhaite vérifier, en utilisant le langage Maude. Nous avons choisi la propriété d'atteignabilité pour montrer comment analyser et étudier le comportement du système. La vérification représente la finalité de notre travail de modélisation.

Conclusion générale et perspectives

A travers ce mémoire, notre objectif était de proposer une approche de modélisation des logiciels à base de composants en utilisant les réseaux de Petri. Notre choix a été porté sur les ECATNets (Extended Concurrent Algebraic Term Nets) qui sont considérés comme une catégorie de réseaux de Petri algébrique de haut niveau. Ce choix est motivé par leurs capacités de fournir des modèles hautement compacts des systèmes concurrents et distribués, grâce aux notations syntaxiques riches qui possèdent et qui leurs donnent une puissance d'expression particulière. En outre, ils sont caractérisés par une sémantique saine et non ambiguë exprimée en terme de la logique de réécriture, celle-ci constitue un Framework logique et sémantique particulièrement adapté pour la spécification du comportement dynamique des systèmes parallèles et distribués. Les logiciels à base de composants sont des systèmes connus par leurs parallélismes inhérents, du fait qu'ils sont essentiellement conçus par l'interconnexion d'un ensemble de composants logiciels préfabriqués non autonomes qui peuvent être locales ou distribués. Le développement de ce genre de systèmes suit donc une approche bien adaptée connue sous le nom de CBD, ou le développement basé composants.

Nous avons présenté dans le premier chapitre l'évolution des approches de développements logiciels traditionnelles qui n'étaient pas bien adaptés pour suivre le cours de l'évolution que les systèmes logiciels ont connue. La complexité croissante de ces derniers a fait que leurs échecs étaient incontournables. Ensuite, nous avons présenté le CBD comme une solution basée sur la notion de réutilisation pour palier aux problèmes des approches classiques. Son principe est de construire des applications logicielles par assemblage de briques logicielles préexistant. Le cœur de cette approche est le composant, et toute l'ingénierie logicielle à base de composants (CBSE) est basée autour. Nous avons présenté quelques standards de composants logiciels, et on a mentionné qu'ils sont tous fondés sur les mêmes concepts. Ils offrent et requièrent des services définis par des interfaces de leurs environnements externes.

Le deuxième chapitre a été le sujet de détail de la logique de réécriture et les ECATNets. Des exemples illustratifs montrant comment ces derniers sont utilisés pour la modélisation de différents types de systèmes ont été présentés. Le premier modélise une cellule de production, et le deuxième modélise la fonction de début de transmission d'une station émettrice Ethernet.

Dans le troisième chapitre, nous avons présenté une approche formelle pour la modélisation des logiciels à base de composant qui est réalisée en termes de spécifications formelles des composants individuels et de leurs connexions. Nous avons proposé pour cette fin, une notation basée sur le formalisme des ECATNets pour spécifier les différents services d'un composant logiciel et fournir une sémantique plus riches pour les opérations définissant les interfaces du composant. Nous avons vu que cette approche se divise en

trois étapes ; la première spécifie le système en cours de conception en utilisant les notations proposées, et vise à fournir un modèle ECATNet qui spécifie un assemblage formel de composants. Le comportement interne de chaque composant est aussi définit dans le modèle. On a vu dans la deuxième étape comment générer les règles de réécriture définissant le comportement dynamique de l'assemblage de composants, en appliquant les métarègles des ECATNets. On a vu aussi dans la troisième étape comment vérifier les propriétés souhaitées par le concepteur du système tel que l'atteignabilité. On peut par exemple vérifier aussi la propriété de vivacité, ou même de traiter certains problèmes tel que le problème d'incompatibilité d'interfaces. Ce dernier peut être expliqué par exemple, par une incompatibilité de paramètres où les paramètres peuvent avoir les mêmes noms mes avec différents types. Ou par une incompatibilité d'opération où les noms d'opérations dans les interfaces composées sont différents. La vérification a été réalisée par le langage Maude qui implémente la logique de réécriture. Nous avons montré l'application de l'approche sur deux exemples, le premier est du domaine des interfaces utilisateurs alors que le second est du domaine des processus métier.

Notre travail offre un modèle de spécification formelle d'un composant logiciel qui permet de fournir une sémantique plus facile à comprendre du comportement. Il fourni éventuellement plus de détails sur les interfaces du composant ce qui permet de mieux les inspecter, et les évaluer et par conséquent permet des constructions ou des assemblages non ambiguës.

Comme un travail futur et pour la concrétisation d l'approche proposée, nous prévoyons de construire un outil graphique pour la modélisation de logiciels à base de composants avec les ECATNets en utilisant la notation proposée. L'outil doit intégrer un module de translation qui à partir du modèle généré, des règles de réécriture écrit en langage Maude sont automatiquement générer pour la vérification formelle de l'assemblage des composants.

Références

[1] I. Crnkovic, M. Larsson, **Building Reliable Component-based Software Systems**, Artech House, 2002.

[2] M. O'Docherty, **Object-Oriented Analysis and Design: Understanding System Development with UML 2.0**, John Welly & sons, USA, 2005.

[3] Nicolas Belloir, **Composition conceptuelle basée sur la relation Tout-Partie**, Doctorat de l'université de Pau et des de l'Adour, soutenue le 9 décembre 2004.

[4] Boehm, B. 2000, **Spiral Development: Experience, Principles, and Refinements**. Report No. CMU/SEI-2000-SR-008. Spiral Development Workshop, February 9, 2000.Software Engineering Institute, Carnegie-Mellon University, Pittsburgh. PA.

[5] I. Jacobson, G. Booch, J. Rumbaugh, **The Unified Process**, IEEE Software, vol. 16, no. 3, pp. 96-102, May/Jun, 1999.

[6] Alfred Strohmeier, **CYCLE DE VIE DU LOGICIEL**, Laboratoire de Génie Logiciel - Département d'Informatique Ecole Polytechnique Fédérale de Lausanne, 13 mars 2000.

[7] Robin Passama, **Conception et développement de contrôleurs de robots, Une méthodologie basée sur les composants logiciels**, Thèse préparée au sein du Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, soutenue le 30 Juin 2006.

[8] X. Cai, M.R. Lyu, K. Wong, Roy Ko, **Component-Based Software Engineering: Technologies, Development Frameworks and Quality Assurance Schemes**, in Proceedings APSEC 2000, Seventh Asia-Pacific Software Engineering Conference, Singapore, December 2000, pp372-379.

[9] Alan W. Brown, **Large-Scale, Component-Based Development**, Prentice Hall PTR, 30 Mai 2000.

[10] Z .Stojanović, **A Method for Component-Based and Service-Oriented Software Systems Engineering**, Doctoral Dissertation, Delft University of Technology, The Netherlands 2005.

[11] Jerry Zeyu Gao H.-S, Jacob Tsao Ye Wu, **Testing and Quality Assurance for Component-Based Software**, Artech House, 2003.

[12] Peter Herzum, Oliver -Sims, **Business component Factory, A Comprehensive Overview of Component-Based Development for the Enterprise**, Wiley Computer Publishing 2000.

[13] Theo Dirk Meijler and Oscar Nierstrasz, **Beyond Objects: Components**, In Cooperative Information Systems: Current Trends and Directions, M.P. Papazoglou, G. Schlageter (Ed.), Academic Press, Nov., 1997.

[14] Ivica Crnkovic, **Component-Based Software Engineering—New Challenges in Software Development**, Journal of Computing and Information Technology - CIT 11, 2003, 3, 151–161.

[15] Ouafa Hachani, **Patrons de conception à base d'aspects pour l'ingénierie des systèmes d'information par réutilisation**, thèse de doctorat au sein de l'université JOSEPH FOURIER-Grenoble I, soutenue le 4 Juillet 2006.

[16] Hedley Apperly, Ralph Hofman, Steve Latchem, Barry Maybank, Barry McGibbon, David Piper, Chris Simons, **Service- and Component-based Development: Using Select Perspective and UML**, Addison Wesley January 24, 2003.

[17] Jan Bosch Charles Krueger (Eds.), **Software Reuse: Methods, Techniques, and Tools**, 8th International Conference, ICSR 2004 Madrid, Spain, July 5-9, 2004, Proceedings Springer.

[18] Danielle Ribot, Blandine Bongard, Claude Villermain, **Development Life-cycle WITH Reuse**, Proceedings of the 1994 ACM symposium on Applied computing, Arizona, United States, Pages: 70 – 76.

[19] Philippe RAMADOUR**, Modèles et langage pour la conception et la manipulation de composants réutilisables de domaine**, thèse de doctorat au sein de l'université d'Aix-Marseille III, soutenue le 17 Décembre 2001.

[20] Luiz Fernando Capretz, **Y: A New Component-Based Software Life Cycle Model**, Journal of Computer Science 1 (1): 76-82, 2005.

[21] Kuljit Kaur, Parminder Kaur, Jaspreet Bedi, and Hardeep Singh, **Towards a Suitable and Systematic Approach for Component Based Software Development**, PROCEEDINGS OF WORLD ACADEMY OF SCIENCE, ENGINEERING AND TECHNOLOGY VOLUME 21 MAY 2007.

[22] Ivica Crnkovic, Stig Larsson and Michel Chaudron, **Component-based Development Process and Component Lifecycle**, Journal of Computing and Information Technology - CIT 13, 2005, 4, 321-327.

[23] Ivica Crnkovic, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan**, Specification, Implementation, and Deployment of COMPONENTS**, Communications of the ACM Volume 45 , Issue 10 (October 2002) Pages: 35 – 40.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison Wesley, Reading, MA, 1995.

[25] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda,Fred Long, John Robert, Robert Seacord, Kurt Wallnau, **Volume II: Technical Concepts of Component-Based Software Engineering**, 2nd Edition TECHNICAL REPORT, CMU/SEI-2000-TR-008, ESC-TR-2000-007 May 2000.

[26] Oualid KHAYATI, **Modèles formels et outils génériques pour la gestion et la recherche de composants**, Thèse de doctorat de l'institut national polytechnique de Grenoble (INPG), Le 17 décembre 2005.

[27] Abdelmadjid KETFI, **Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels**, Thèse au sein de l'Université Joseph Fourier de Grenoble, soutenue le 10 décembre 2004.

[28] Barbara PERNICI and al, **Conceptual Modelling and Software Components Reuse: Towards the Unification**, In A. Sølvberg, S. Brinkkemper, E. Lindencrona (eds.): Information Systems Engineering: State of the Art and Research Themes. Springer Verlag, London, 2000.

[29] Ambler S. W, **An Introduction to Process Pattern**, AmbySoft White paper, 1998. http://www.Ambysoft.com.

[30] Bertrand Meyer: **Applying "Design by Contract"**, in Computer (IEEE), vol. 25, no. 10, October 1992, pages 40-51.

[31] Microsoft Component object model (COM) specification, http://www.microsoft.com/com/resources/comdocs.asp.

[32] Humberto CERVANTES, **Vers un modèle a composants orienté services pour supporter la disponibilité dynamique**, Thèse au sein de l'Université Joseph Fourier de Grenoble, soutenue Le 29 Mars 2004.

[33] OMG Business Application Architecture, White Paper ,http://jeffsutherland.com/oopsla/bowp2.html.

[34] Cory Casanave, **Business-Object Architectures and Standards**, in Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Addendum to the Proceedings, OOPSLA: ACM/SIGPLAN October, 1995.

[35] TADAO MURATA, **Petri Nets: Properties, Analysis, and Applications**, in Proceedings of the IEEE, Vol. 77, N80. 4, pages 541-580. April 1989.

[36] Jean Marie PROTH et Xiaolan XIE, **Les réseaux de Petri pour la conception et la gestion des systèmes de production**, Paris 1995, Edition Masson.

[37] G. Scorletti et G. Binet, **Réseaux de Petri**, Cours EL401T2, Gérard Scorletti, France, 2006.

[38] Final Draft International Standard ISO/IEC 15909, **High-level Petri Nets - Concepts, Definitions and Graphical Notation**, Version 4.7.1, October 28, 2000.

[39] Guy VIDAL-NAQUET, Annie CHOQUET-GENIET, **Réseaux de Pétri et systèmes parallèles**, Edition ARMAND COLIN, Paris 1992.

[40] J. Padberg, M. Gajewsky and C. Ermel, **Refinement versus Verification: Compatibility of Net InVariants and Stepwise Development of High-Level Petri Nets**, 1997. http://citeseer.ist.psu.edu/cache/papers/cs/1596/http:zSzzSzw3.cs.tu-berlin.dezSzcszSzifbzSzTeBerichtzSz97zSzTR97-22.pdf/padberg97refinement.pdf.

[41] Gajewsky, C. Ermel, **Transition InVariants in Algebraic High-Level Nets**, http://citeseer.ist.psu.edu/cache/papers/cs/15875/http:zSzzSzuser.cs.tu-berlin.dezSz~magdazSzPaperszSzIDPT99.pdf/gajewsky99transition.pdf

[42] C. Ermel, J. Padberg, **Formalization of Variables in Algebraic High-Level Nets, Comparison of Different Approaches (1998)**, http://citeseer.ist.psu.edu/cache/papers/cs/1596/http:zSzzSzw3.cs.tu-berlin.dezSzcszSzifbzSzTeBerichtzSz97zSzTR97-19.pdf/ermel98formalization.pdf

[43] Antoine Reilles, **Réécriture dans un cadre concurrent**, DEA de l'Institut National Polytechnique de Lorraine, 9 juillet 2003.

[44] Horatiu Cirstea, **Calcul de réécriture : fondements et applications**, thèse de Doctorat au sein de l'université Henri Poincaré Nancy 1, soutenue le 25 octobre 2000.

[45] Nachum Dershowitz, Jean-Pierre Jouannaud, **REWRITE SYSTEMS**, chapter 6 of handbook of theoretical computer science; volume B: Formal Methods and Semantics, pages 243-320 North Holland Amsterdam, 1990.

[46] Kung-Kiu Lau, Mario Ornaghi, **A Formal Approach to Software Component Specification**, In G.T. Leavens D. Giannakopoulou and M. Sitaraman, editors, Proc. of Specification and Verification of Component-based Systems Workshop at OOPSLA2001, pages 88–96, 2001.

[47] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. **All About Maude -A High-Performance Logical Framework**, volume 4350 of Lecture Notes in Computer Science. Springer, 2007.

[48] M. Bettaz and M. Maouche, **How to specify non determinism and true concurrency with algebraic term nets**, Lecture Notes in Computer Science, Vol. 655, Springer-Verlag, 1993, pp. 164–180.

[49] Ian Sommerville, **Le génie logiciel et ses applications**, Addison-Wesley, 1988.

[50] Michael Havey, **Essential Business Process Modelling**, O'Reilly Media, Inc., 2005.

[51] M. Bettaz, K. Djemame, C. Gilles, M. Mackenzie, **Performance comparison of high level algebraic nets distributed simulation protocols**, Winter Simulation Conference, 1996.

[52] N. Aoumeur, K. Barkaoui, and G. Saake, **On the Benefits of Rewrite Logic as a Semantics for Algebraic Petri Nets in Computing Siphons and Traps**. In Proc of the 10th International Conference on Computing and Information (ICCI '2000), Kuweit, to appear in LNCS, Springer.

[53] Thomas Noll, **On coherence properties in term rewriting models of concurrency**. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 478–493. Springer, Heidelberg (1999).

[54] José Meseguer, **Research Direction in Rewriting Logic**, In U. Berger and H. Schwichtenberg, editor, Computational Logic, NATO, Advanced Study Institute, Marktoberdorf, Germany, July 29 - August 6, 1997. Springer Verlag, 1998.

[55] Narciso Marti-Oliet and José Meseguer, **Rewriting Logic as a Logical and Semantic Framework**, In J. Meseguer, editor, Proc. First Intl Workshop on Rewriting Logic and its Applications, volume 4 in Electronics Notes in Theoretical Computer Science. Elsevier, 1996.

[56] José Meseguer, **Rewriting Logic as a semantic Framework for Concurrency: a Progress Report**, In U. Montanari and V. Sassone, editors, Proc. 7th Intern. Conf. on Concurrency Theory: CONCUR '96, Pisa, August 1996, pages 331-372, 1996. LNCS 1119.

[57] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada, **Maude: specification and programming in rewriting logic**, SRI International, June 2001, http://maude.cs.uiuc.edu/papers.

[58] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. **All About Maude - A High-Performance Logical Framework**, volume 4350 of Lecture Notes in Computer Science. Springer, 2007.

[59] José Meseguer, **Software Specification and Verification in Rewriting Logic**, Lectures given at the NATO Advanced Study Institute International Summer School, Marktoberdorf. Germany, 2002. Available from http://maude.cs.uiuc.edu, 2003.

[60] H. Sebih, K. Barkaoui, M. Bettaz, F. Belala et Z. Sahnoun, **Towards a Temporal Logic τLTL for the Verification of Rewriting Theories Denoting ECATNets** In 1st Int'l Workshop on Automated Technology for Verification and Analysis (ATVA'03), Taiwan, 2003.

[61] Bettaz M., Maouche M., Soulami M. and Boukebeche M, **Using ECATNets for specifying communication software in the OS1 framework**. In *Proceedings* of *ICCI'92,* pages -410-413. IEEE, 1992.

[62] M. Bettaz, M. Maouche, K. Barkaoui, **Formal Specification of Communication Protocols with Object-Based ECATNets**, Proceedings of EUROMICRO-22, Prague, Czech Republic, IEEE, pages 492-499,1996.

[63] N. Zeghib, K. Barkaoui, M. Bettaz, **Contextual ECATNets Semantics in Terms of Conditional Rewriting Logic**, aiccsa,pp.936-943, IEEE International Conference on Computer Systems and Applications, 2006.

[64] M. Maouche, M. Bettaz, G. Berthelot and L. Petrucci, **Du vrai Parallélisme dans les Réseaux Algébriques et de son Application dans les Systèmes de Production**, Conférence Francophone de Modélisation et Simulation (MOSIM'97), Hermes, 1997, pp. 417-424.

[65] M. Bettaz and A. Mehemmel, **Modeling and proving of truly concurrent systems with CATNets**, In Proceedings of Euromicro Workshop on Parallel and Distributed Processing, pages 265-272, IEEE, 1993.

[66] P .T. Cox, B .Song, **A Formal Model for Component-Based Software**, IEEE 2001 Symposium on Human Centric Computing Languages and Environments (HCC'01), 2001.

[67] R. Bastide and E. Barboni, **Software Components: a Formal Semantics Based on Coloured Petri Nets**, Electronic Notes in Theoretical Computer Science 160 (2006) 57–73, ScienceDirect.

[68] Ramamoorthy, C. V. and G. S. Ho, **Performance Evaluation of Asynchronous Concurrent systems Using Petri Nets**, IEEE Trans. Software Eng., 1980, pp. 440-449.

[69] N. Boudiaf, and A. Chaoui, **Double Reduction of Ada-ECATNet Representation using Rewriting Logic**, proceedings of world academy of science, engineering and technology volume 15 october 2006 pages 278/ 284.