THESE

présentée par

Thomas BOLUSSET

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE DE SAVOIE

(Arrêté ministériel du 30 mars 1992)

Spécialité :

Informatique

β-SPACE : Raffinement de descriptions architecturales en machines abstraites de la méthode formelle B

Soutenue publiquement le 30 septembre 2004, devant le jury composé de :

M. Alain HAURAT	Président du jury	Professeur à l'Université de Savoie
M. Jean-Jacques CHABRIER	Rapporteur	Professeur à l'Université de Bourgogne
M. François JACQUENET	Rapporteur	Professeur à l'Univesité Jean Monnet
M. Noureddine BELKHATIR	Examinateur	Professeur à l'Université de Grenoble II
M. Flavio OQUENDO	Directeur de thèse	Professeur à l'Université de Savoie

Thèse préparée au sein du Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance – Ecole Supérieure d'Ingénieurs d'Annecy (ESIA) – Université de Savoie.

A MES PARENTS

REMERCIEMENTS

Je tiens à remercier tous les membres du jury qui ont bien voulu, malgré les responsabilités et engagements multiples de la rentrée universitaire, consacrer un peu de leur temps à la lecture et l'évaluation de ce travail :

> M. Jean-Jacques Chabrier et M. François Jacquenet, pour avoir accepté d'être rapporteurs de cette thèse, pour leur lecture minutieuse de ce mémoire pendant leurs vacances estivales, de même que leurs critiques judicieuses et constructives ;
>
> M. Nourredine Belkhatir et M. Alain Haurat pour avoir accepté de faire partie du jury, pour la lecture de ce document et l'intérêt qu'ils ont porté à ce travail ;
>
> M. Flavio Oquendo pour m'avoir encadré durant ces années, pour m'avoir fait profiter de ses connaissances, de son expérience, sa perspicacité, et pour la confiance qu'il m'a accordée.

Je souhaite également exprimer ma reconnaissance à tous les membres du LISTIC, et plus particulièrement ceux de l'ancien LLP, pour leur accueil, l'ambiance agréable, et pour m'avoir donné les moyens de faire aboutir ce travail.

Merci notamment à Valérie, indispensable et généreuse, à Ilham, Sorana et Hervé pour les bons moments passés et les discussions intéressantes, à Claire, Philippe, Sylvain de précieux compagnons de bureaux, ainsi qu'à Lamia, Bethsabée et bien sûr tous mes autres collègues doctorants (David, Fabien, Frédéric, Gülçin Jérôme, Lionel, Luciano Karim, Sébastien, Selma, Vincent, Yoann …) pour leur amitié, leur soutien et l'esprit convivial qu'ils ont su cultiver.

Je n'oublie pas les membres du CRID de Dijon (devenu depuis le LERSIA), dans lequel j'ai appris à faire et à aimer la recherche, sous la bienveillance de Mme et M. Chabrier.

Un immense Merci à mes parents pour leur soutien sans faille, à ma famille (Sylvain, Jean-michel et Laurence), à mes amis proches ou éloignés, de tous âges et tous temps, pour leurs encouragements et leurs marques d'affection (Alain, Bertrand, Carine, Céline, Claire, François, Frédéric, Marie, Roxane, Sébastien, Stéphane, Virginie, Xavier et tous les autres).

ABSTRACT

β-SPACE: Refinement of architectural descriptions into abstract machines of the B formal method

A software architecture describes its structure and behaviour in terms of components and connectors. Nevertheless, the architecture description languages do not support the complete development of complex software systems, from architectural design to the executable code. On the other hand, some formal development methods permit to refine a software specification in order to obtain another one closer to the implementation, or even to generate code, but without taking into account the system architectural description.

We propose, in this thesis, to use a refinement mechanism to transform the architectural description into a "classical" formal specification, which is already supported by tools allowing the development achievement.

The research problem that we are treating is the refinement of a software architecture description in π-SPACE, a software architecture description language based on a process algebra, towards an abstract specification, formed by abstract machines of the B method, which is supported by tools to help the formal development and the code generation.

We develop a formal system – named β-SPACE – to bring successive refinements into operation, leading from the starting architectural description (in π-SPACE) to a formal specification (a set of abstract machines of the B method) such as to make a formal development of the application possible, in the B method framework, while guaranteeing that each refinement step preserves the initial architectural description properties.

The formal definition of the software architecture refinement is based on the rewriting logic, in which the abstract architectural elements are represented, but the constructs of the target specification language too. This logic is also supported by a development and execution tool which permits to automate the transformations.

Our approach of the architectural refinement differs from the other existing methods, by being interested not only in the addition of details to the formal description, but also in the transformation of its control structure: the composition of components and connectors in the architecture is transformed to obtain a hierarchy of B abstract machines. This requires to change the way to control actions inside these different descriptions of the same system: from concurrent behaviours of components and connectors, synchronized by communications, to hierarchically organized abstract machines, related together by operation calls. However, we ensure the conservation of the interesting architectural properties.

This is an original approach both concerning its architectural range (structure but also behaviour), its formalisation and its connection with the classical formal methods.

Keywords: Software Engineering, Formal Refinement, Software Architecture, Formal Method.

TABLE DES MATIERES

ANNEXE A : CLASSIFICATION DES DIFFERENTES APPROCHES DE RAFFINEMENT

ANNEXE B : PATRONS DE TRANSFORMATION (1ER, 2EME ET DEBUT DU 3EME)

LISTE DES FIGURES

LISTE DES TABLEAUX

Chapitre 1 : Introduction

Chapitre 1 : Introduction

1. Contexte général

Une approche en plein essor dans l'ingénierie des logiciels est celle des architectures logicielles. L'architecture d'un logiciel décrit la structure et le comportement du système dans son ensemble, et repose sur la description de composants, unités de calcul localisées et indépendantes, de connecteurs, unités d'interactions entre les composants, et de configurations, ensembles de composants combinés via des connecteurs.

Les descriptions d'architectures logicielles permettent de modéliser de façon assez intuitive les différentes unités de calcul et les interactions ou liens de communication entre elles. Elles ont donné lieu à de nombreux travaux ces dernières années, par exemple : [Perry et Wolf 1992], [Garlan et Perry 1995], [Perry 1997a], [Allen et Garlan 1997], [Compare et al. 1999], [Oreizy et al. 1999] …

D'autre part, un développement formel fournit un moyen sûr pour passer d'une spécification abstraite d'un système à un niveau de spécification plus détaillé, dit concret, afin de pouvoir, par la suite, générer automatiquement une implémentation, le code, pour une plate-forme et un langage de programmation donnés.

Néanmoins, une méthode formelle de développement est à forte composante mathématique et son utilisation nécessite habituellement une formation spécifique.

Certaines méthodes formelles permettent de raffiner des spécifications abstraites pour obtenir des spécifications concrètes, voire générer des implémentations. On appelle **raffinement** une spécialisation apportée à une modélisation du système. Raffiner consiste alors à offrir une nouvelle formulation de la représentation du système, qui ne doit pas en contredire les propriétés ; cela se traduit souvent par une diminution du niveau d'abstraction ou d'indéterminisme, par exemple en ajoutant des détails à une spécification abstraite, de façon à s'approcher d'une implémentation dans un langage de programmation. Mais, ces approches ne prennent pas en compte la description architecturale du système.

Par ailleurs, certains langages de description d'architecture peuvent autoriser une compilation en code exécutable. Cependant, ils contraignent la spécification architecturale dès le départ avec des détails d'implémentation [Ziane 2001].

2. Introduction à la problématique de recherche

Le problème de recherche posé est celui de la mise en place d'un moyen pour passer d'une description architecturale donnée à une spécification dans une méthode formelle classique.

Notre objectif est d'automatiser cette transformation, afin de l'opérer de façon systématique. Ce raffinement doit se faire relativement à une "plate-forme formelle" et un langage de description d'architectures existants. De plus, il doit garantir la conservation de propriétés entre les descriptions (cf. figure I.1).

Dans ce contexte, nous proposons une approche qui s'appuiera, en amont d'un développement formel classique, sur une description architecturale de haut niveau que nous raffinerons dans le langage formel cible d'une méthode formelle classique. A l'issue de la phase d'analyse des besoins, l'architecte, utilisateur de β-SPACE, décrira le système sous la forme d' une architecture logicielle en se servant du langage choisi. Différents travaux ont porté sur les descriptions d'architecture de haut niveau ([Garlan et al. 1995], [Shaw et Garlan 1995], [Perry 1997b], [Perry 1998],

[Melton et Garlan 1997], [Garlan et Wang 1998], [Osterweil 1998], [Allen et al. 1998], [Bolusset et al. 1999a], [Egyed et al. 1999], [Egyed et Medvidovic 2000], [Egyed et Medvidovic 2001]).

A la suite de cette phase de spécification architecturale, il n'aura qu'à "appuyer sur un bouton" afin qu'elle soit raffinée et que soit générée une spécification formelle conforme à la méthode de développement formel choisie [Riemenschneider 1998]. La spécification abstraite obtenue pourra à son tour être traitée de façon conventionnelle, dans le cadre d'un développement formel classique, mais jusqu'à l'implémentation cette fois.

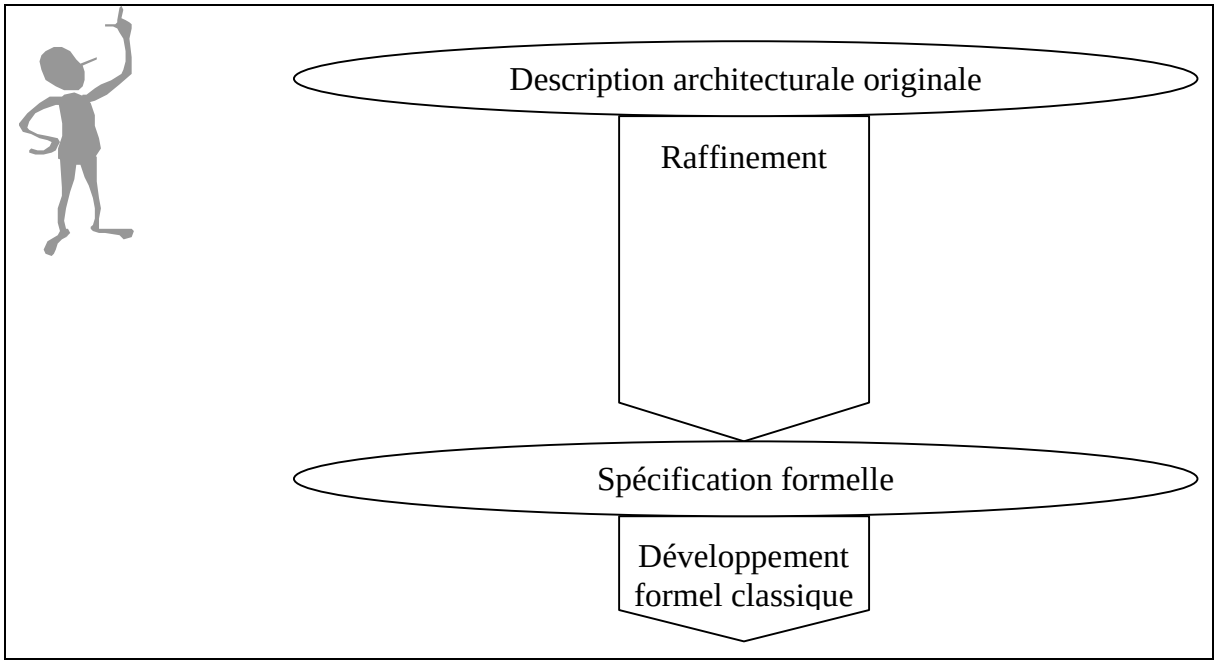

Figure I.1. : illustration du raffinement à mettre en place

Le fait d'opérer cette phase de raffinement entre deux spécifications écrites dans des formalismes avec des pouvoirs d'expression comparables nous permettra, de plus, de garantir leur compatibilité par le raffinement et la conservation de propriétés ou du comportement. Les propriétés que l'architecte souhaitera voir conservées pourront être, par exemple, la vivacité, la sûreté, l'atteignabilité ou encore la terminaison.

3. Quel raffinement ?

Peu de langages de description d'architecture logicielle supportent explicitement un processus de "raffinement" en plusieurs étapes distinctes, à la différence de ceux qui ne permettent qu'une phase de compilation en code. Pourtant, il est courant pour un élément architectural de pouvoir être décomposé en un assemblage d'autres composants et connecteurs [Clements 1996] [Garlan 1996] [Clarke 1998] [Bolusset et al. 1999b] [Medvidovic et Taylor 2000].

Pour notre approche du raffinement, nous sommes amenés à distinguer ces deux types de mécanismes :

- la *décomposition*, ou *raffinement horizontal*, qui ne provoque pas de changement de niveau d'abstraction, mais qui ajoute à la spécification de nouveaux composants, de façon plus détaillée (par exemple la décomposition d'un composant effectuant une opération complexe pour détailler les opérations élémentaires mises en oeuvre),
- la *transformation*, ou *raffinement vertical*, de certaines parties de la spécification, décidées par des choix liés à la plate-forme d'implémentation ultérieure, et qui, de ce fait, diminue le niveau d'abstraction, sans remettre en cause la spécification (seules des précisions sont

éventuellement ajoutées, par exemple pour dire que l'ensemble des entiers naturels est noté "INTEGER").

Ainsi, ce sont plutôt des mécanismes de raffinement vertical que nous devons mettre en place afin, notamment, de transformer la structure de contrôle d'une description architecturale (des composants communiquant via des connecteurs) pour l'expliquer en des termes compréhensibles par la méthode de développement formel.

4. Quels formalismes ?

Si nous nous positionnons dans le cadre du développement de logiciels centré architecture, nous proposons de définir un système formel – appelé β-SPACE – pour la description de raffinements d'architectures logicielles. Ce système doit permettre de mettre en place un "processus formel" de raffinements successifs, menant d'une description architecturale abstraite à une spécification suffisamment concrète pour qu'un développement formel de l'application soit possible. Du point de vue de ce dernier, l'étape de modélisation est simplifiée par l'emploi d'un langage de plus haut niveau. Chacune de ces étapes de raffinement correspond à un aspect bien précis de la transformation, en relation avec soit le langage de description d'architecture choisi, soit la méthode de développement formel. Ces niveaux d'abstraction intermédiaires facilitent la mise en œuvre du raffinement, mais resteront "invisibles" aux yeux de l'architecte.

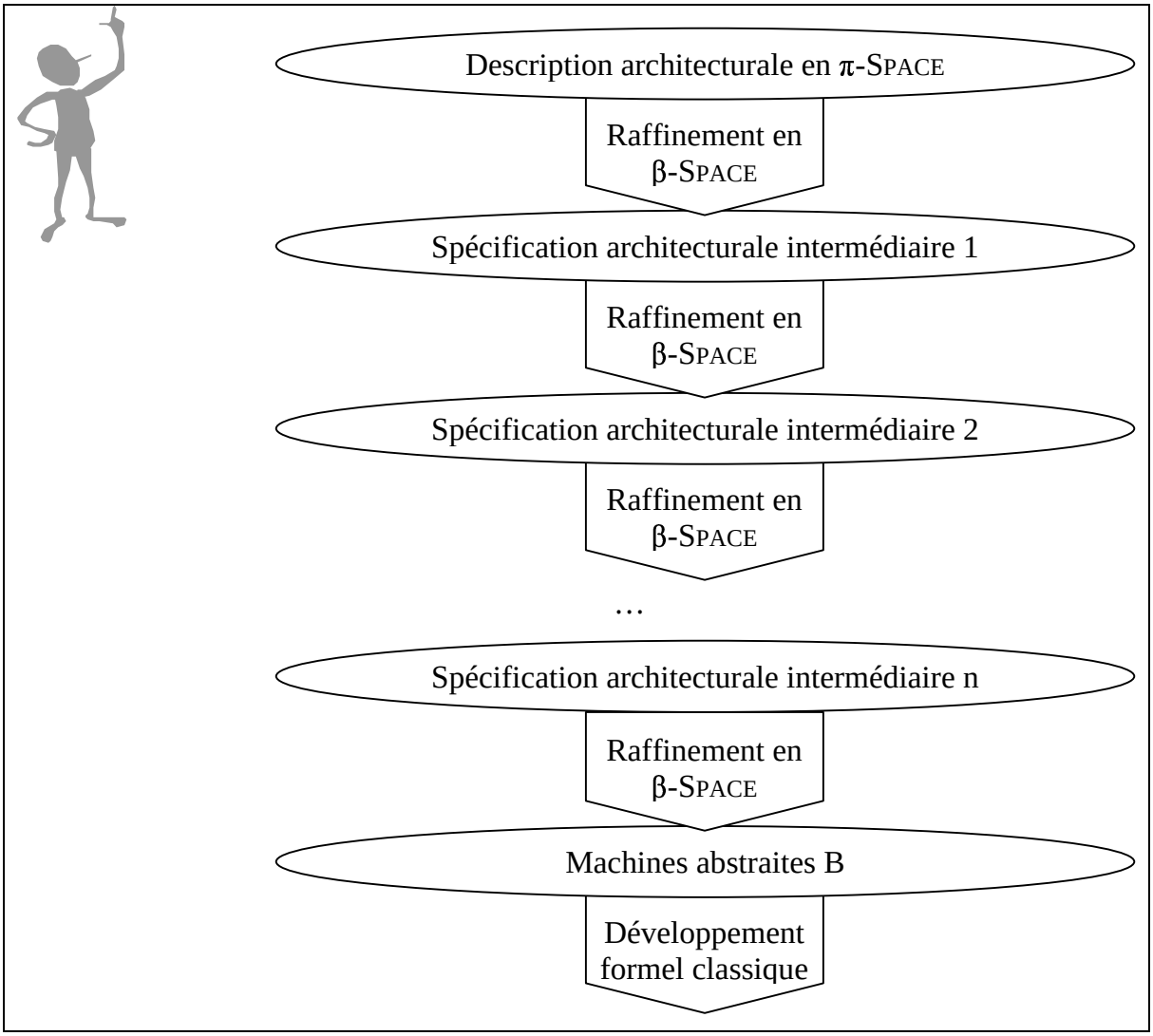

Figure I.2. : illustration du raffinement en β-SPACE

Cette suite de raffinements doit être décrite formellement. Certains langages de description d'architecture logicielle, tels SADL ou Rapide, prennent en compte le raffinement sur plusieurs niveaux d'abstraction, mais ils ne permettent pas d'achever le développement complet de systèmes logiciels complexes car ils n'offrent pas de support pour la génération du code. Actuellement, les méthodes utilisant le raffinement se classent en deux catégories. La première nous oblige à prouver, après chaque étape du raffinement de chaque système, la préservation de propriétés avec le niveau d'abstraction précédent (le raffinement est vérifié a posteriori). La seconde nous permet de construire la nouvelle spécification concrète à partir de la spécification abstraite (le raffinement s'inscrit dans un développement génératif). Cette dernière catégorie est la plus intéressante, puisqu'en garantissant que chaque étape de raffinement conserve les propriétés de la description architecturale de départ, indépendemmant du système logiciel en cours de développement, nous pouvons nous abstenir d'une phase de vérification a posteriori.

Le langage de description d'architecture logicielle qui nous intéresse, π-SPACE, est fondé sur une algèbre de processus, le π-calcul ; ceci octroie à π-SPACE un pouvoir expressif important, qui permet notamment l'expression d'architectures dynamiques (cf. chapitre 3). Le langage de spécification formelle est celui de la méthode B : elle a en effet le grand avantage de disposer d'outils pour aider au développement formel et à la génération du code. Notre approche s'effectue naturellement en traitant le raffinement par des réécritures successives de la spécification abstraite pour substituer aux constructions architecturales des éléments de machines abstraites B (cf. figure I.2). Notre choix se porte sur l'utilisation d'une logique supportée, elle aussi, par un outil de développement formel : la logique de réécriture.

5. Plan de la thèse

L'état de l'art du raffinement en génie logiciel est exposé dans le *deuxième chapitre* de ce mémoire.

Le *troisième chapitre* introduit de façon générale les trois formalismes utilisés pour notre travail, à savoir le langage de description d'architecture π-SPACE, la méthode formelle B utilisée pour le développement ultérieur du système, mais aussi la logique de réécriture pour l'expression de notre raffinement. En effet, notre processus de raffinement doit, à chaque étape, assurer la construction correcte du système, avec notamment la conservation des propriétés sémantiques de la description d'architecture, tout en nécessitant le moins possible d'interventions de la part de l'utilisateur. La représentation formelle du raffinement d'architectures logicielles requiert un formalisme dans lequel on puisse représenter simplement les éléments architecturaux abstraits et les interactions entre les différents composants et connecteurs qui constituent l'architecture du système, mais aussi les constructions du langage de spécification concret. Une telle approche s'effectue naturellement en traitant le raffinement par des réécritures successives de la description d'architecture pour substituer aux constructions architecturales des éléments du langage de la méthode formelle. Notre processus de raffinement est donc contraint par les deux langages de spécification formelle choisis comme source et cible.

Dans le *quatrième chapitre*, nous présentons notre nouvelle approche de raffinement, en présentant d'abord notre démarche basée sur des règles de réécriture, puis sa mise en œuvre sur une étude de cas.

Le *cinquième chapitre* montre comment l'implémenter dans la logique de réécriture et valider les spécifications formelles obtenues, qui doivent être correctes par construction, dans le cadre de la méthode de développement formel B.

Enfin, après un bilan de notre travail, le *sixième chapitre* nous présente quelques perspectives prometteuses.

Chapitre 2 : Etat de l'art

Chapitre 2 : Etat de l'art

1. Etat de l'art du raffinement dans le génie logiciel

Le raffinement est étroitement lié au développement de logiciels, tout au long du cycle de vie. De ce fait, il est possible de voir dans de nombreuses méthodologies des approches de raffinement plus ou moins explicites.

En fait, même si les langages de description d'architectures permettent habituellement de décomposer des architectures logicielles, peu d'entre eux comportent des mécanismes appropriés pour un raffinement d'architecture couvrant plusieurs niveaux d'abstraction. Par conséquent, nous ne considérons pas seulement les méthodes de raffinement d'architectures logicielles.

Tout d'abord, des techniques de développement complètement informelles, tel l'établissement d'un cahier des charges en langage naturel, ne peuvent pas nous intéresser ici, vu qu'elles ne permettent ni d'établir une description complète du système, ni de vérifier la présence de propriétés spécifiques. Cependant, certaines autres méthodes du génie logiciel, bien qu'elles ne manipulent pas vraiment les mêmes concepts que les langages de description d'architectures logicielles, utilisent le raffinement pour assurer le lien entre les spécifications de systèmes à différentes étapes du développement. Elles incluent à la fois des approches formelles et semi-formelles.

Le raffinement peut également être abordé de façons tout à fait différentes, selon les caractéristiques et les buts de la méthode de développement de logiciels considérée (comme présenté par exemple dans [Gorrieri et Rensink 1999], [Egyed et al. 2000], [Kerschbaumer 2002] ou [Vos et al. 2002]). D'une part, certains systèmes ne font intervenir qu'une unique étape de raffinement, au moment de la phase de compilation. Mais cette étape se prête mal à la vérification de la correction, c'est-à-dire de la conservation des propriétés. La notion de raffinement est alors associée à la phase de conception ; elle peut ainsi être employée plusieurs fois de suite. D'autre part, une partie des méthodes fournit des mécanismes propres à construire la description concrète pour raffiner une spécification, à partir de l'originale, tandis que les autres se contentent d'effectuer des vérifications sur des spécifications déjà existantes. Enfin, la relation de raffinement elle-même peut s'avérer être un critère discriminant entre les différentes approches, étant donné qu'elle ne repose pas toujours sur les mêmes concepts, voire sur les mêmes objectifs.

Dans la suite de ce chapitre, nous décrivons plusieurs méthodes du génie logiciel, représentant les différentes relations de raffinement vertical. Nous les présentons dans un premier temps par "familles", afin d'être capable de comparer, par la suite, leurs approches respectives du raffinement. Les premières concernent des méthodes formelles orientées modèles, dans l'ordre alphabétique :

- la méthode formelle B,
- une combinaison de B et CSP,
- CSP2B, deuxième exemple traitant une combinaison différente,
- le B événementiel ("event-driven B"), extension de la méthode B classique,
- la réduction dans la méthode B, variante fondée sur la méthode B,
- VDM (son langage de spécification),
- la notation Z.

L'approche suivante, Catalysis, est semi-formelle, basée sur UML. Viennent ensuite différentes catégories de langages de description d'architecture, chacune supportant différemment le raffinement ; elles sont illustrées par :

- Darwin, MetaH, UniCon et Weaves,
- Rapide,
- SADL.

Enfin, le calcul de raffinement est un formalisme mathématique spécialisé dans le raffinement de spécifications pour obtenir des programmes.

Dans un deuxième temps, les critères servant à la classification de ces travaux seront décrits, et nous serviront à classer les approches présentées. Nous discuterons alors du raffinement d'architecture logicielle, lorsqu'il est traité simplement comme une phase d'implémentation. Puis, les approches traitant uniquement de relations de raffinement basées sur des spécifications déjà existantes seront présentées, avant d'aborder les méthodes plus complètes et formelles qui permettent réellement de construire un nouveau raffinement. Enfin, ce chapitre se conclura par un bilan des différents types de raffinement qui peuvent être rencontrés dans le domaine plus large du génie logiciel, et des problèmes ouverts qui méritent d'être examinés et résolus.

2. Présentation des travaux existants

Ainsi, différents types de raffinement, qui peuvent être rencontrés dans le domaine plus large du génie logiciel, sont étudiés plutôt que de se contenter de ce qu'a déjà pu proposer la communauté de l'architecture logicielle. Chacune des approches présentées est illustrée par un tableau récapitulant d'abord son nom, puis la description de ses principales caractéristiques – essentiellement son objectif – avant de donner éventuellement le lieu de son développement, et finalement quelques références bibliographiques la décrivant.

Dans ce cadre des approches du génie logiciel utilisant le raffinement, la méthode B (tableau II.1) tient une place tout à fait particulière puisqu'elle bénéficie d'outils commerciaux utilisés dans l'industrie du logiciel. Il s'agit là d'un point d'autant plus important que le monde économique est généralement réticent à l'utilisation des méthodes formelles.

Nom	méthode B
Description de l'objectif	• La méthode B est une méthodologie sémantiquement bien formée et structurellement riche pour le développement formel de logiciels à travers le cycle de vie entier ; • c'est une approche formelle basée sur des machines abstraites, qui constituent les blocs de construction de spécifications et sur des obligations de preuve ; • des spécifications de grande ampleur peuvent être construite de façon confiante à partir de plus petites, en utilisant quelques mécanismes d'assemblage pour combiner des machines abstraites ensemble ; • l'approche est compositionnelle, vu qu'une telle combinaison de machines abstraites peut aussi être considérée comme une machine abstraite, permettant ainsi une spécification hiérarchique ; • chaque machine doit avoir un nom propre, de telle sorte que les autres parties d'une large spécification puisse s'y référer ; lorsqu'il faut plus d'une instance d'une machine, son nom, tout comme celui de chacun de ses constituants, est préfixé par un nom d'instance ; • chaque machine abstraite renferme la spécification d'au moins une partie du système, centrée autour d'une notion d'état, résultant de variables modifiables via des opérations ; • les opérations représentent les fonctions apportées par le composant ; elles peuvent prendre des paramètres en entrée, fournir des résultats en sortie, effectuer un changement d'état ou toute combinaison de ces différentes possibilités ; • comme elles constituent l'interface du composant correspondant, les opérations d'une machine abstraite ne peuvent pas se référencer entre elles ; • les types des variables et toute autre information concernant ces dernières sont regroupés pour

	former l'invariant : ces contraintes doivent toujours être vérifiées, que ce soit avant ou après toute application d'une opération ; leur état initial doit également être spécifié ; • le raffinement est une part essentielle de la méthode B, comme des constructions spéciales permettent de construire une spécification plus déterministe : des structures syntaxiques spécifiques permettent d'exprimer une relation de raffinement entre une spécification et son implémentation sous la forme d'une partie de l'invariant de cette dernière ; • il est possible d'avoir un raffinement multiple, en utilisant une unique machine de raffinement pour plusieurs abstractions en même temps (même si une seule machine abstraite pourra être référencée explicitement).
Bibliographie	J.-R. Abrial, "The B-Book: Assigning programs to meanings", Cambridge University Press, 1996. http://www.afm.lsbu.ac.uk/b/ ou http://vl.fmnet.info/b/

Tableau II.1. : caractéristiques de la méthode B

Le raffinement dans la méthode B est décrit explicitement par l'usage de machines introduites par les mots-clés "REFINEMENT" ou "IMPLEMENTATION", et dans lesquelles sont décrites de façon plus poussée les mêmes opérations que dans la machine abstraite raffinée. Chaque raffinement repose alors sur une relation bien précise, établie par l'invariant de la machine qui raffine. Néanmoins, chaque étape nécessite que les préconditions des opérations s'affaiblissent et que leurs postconditions se renforcent : le comportement qui résultera du raffinement doit donc être un comportement possible de l'abstraction. Des outils apportent une aide pour effectuer les preuves ou automatiser la génération du code à partir des implémentations.

Cet intérêt pour la méthode B ne s'arrête cependant pas là, étant donné que d'autres approches l'ont prises comme base de départ, essentiellement dans l'optique de profiter de ses aspects formels. Par exemple, l'approche présentée dans le tableau II.2 cherche à combiner la méthode B avec l'algèbre de processus CSP : la spécification du système comprend des processus CSP en plus des machines abstraites B, pour décrire comment est exécuté le contrôle des actions de ces machines abstraites.

Nom	B et CSP
Description de l'objectif	• CSP est utilisé pour l'exécution du contrôle d'un système abstrait B ; • CSP et B sont concernés par des aspects différents de la description d'un système : CSP spécifie les séquences possibles de transitions et B spécifie les transitions individuellement ; • les 2 vues sont cohérentes si le cadre de travail du contrôle en CSP ne peut jamais appeler un module B en dehors de ses préconditions ; • l'environnement du module B est séparé en 2 couches : la couche de contrôle en CSP et l'environnement extérieur ; • la partie CSP est conservée à travers les niveaux d'abstraction par raffinement (mais cela ne concerne qu'un sous-ensemble de la méthode B).
Où	Department of Computer Science, Royal Holloway, University of London
Bibliographie	Helen Treharne et Steve Schneider, "Using a Process Algebra to control B OPERATIONS", *IFM'99*, Springer, York, 1999. Helen Treharne et Steve Schneider, "How to Drive a B Machine", *ZB'2000: Formal Specification and Development in Z and B*, LNCS 1878, 2000.

Tableau II.2. : caractéristiques de l'approche combinant B et CSP

Ce type d'approche, s'il rajoute une couche de contrôle aux machines abstraites B, ne s'écarte pourtant pas du processus classique de raffinement en B. Par contre, il exclut l'utilisation des outils dédiés au B classique, comme la génération automatique du code de l'application.

CSP2B (dans le tableau II.3) est une autre approche basée sur l'algèbre de processus CSP et la méthode B. A la différence de la précédente qui utilisait une combinaison des deux mêmes formalismes, CSP2B traite la partie en CSP, non plus comme faisant partie de l'environnement de la spécification, mais comme un sucre syntaxique qui peut être traduit en B le moment venu.

Nom	CSP2B
Description de l'objectif	• CSP2B permet la combinaison de descriptions ressemblant à CSP avec des spécifications en B : la notation de CSP sert à décrire l'ordre des opérations d'une machine B ; • la méthode B est adaptée à la conception d'activités distribuées en termes d'événements, mais elle est moins appropriée pour la conception d'activités séquentielles telles que, par exemple, un "compteur de programme" abstrait pour organiser l'exécution des actions ; • la notation CSP est utilisée pour décrire des contraintes d'ordonnancement et améliorer la notation standard de B ; • un outil convertit les spécifications ressemblant à CSP en machines B standards : les spécifications abstraites et les raffinements sont spécifiés en utilisant soit CSP, soit une combinaison de CSP et de B ; • expressions, types et prédicats écrits dans la notation standard B sont copiés dans la machine B générée ; • la composition d'un processus CSP avec une machine B est compositionnelle vis-à-vis du raffinement ; • une description CSP peut être le raffinement d'une autre machine (CSP ou B) ; • les machines B et CSP sont raffinées indépendamment.
Où	Declarative Systems and Software Engineering Group, Electronics and Computer Science, University of Southampton http://www.dsse.ecs.soton.ac.uk
Bibliographie	Michael Butler, *csp2B: A Practical Approach To Combining CSP and B*, Declarative Systems and Software Engineering Group, Technical Report DSSE-TR-99-2, février 1999.

Tableau II.3. : caractéristiques de CSP2B

Le raffinement des machines classiques B est étendu en CSP2B pour couvrir également les machines décrites en CSP. Les différences avec la méthode B sont donc d'autant moins flagrantes que cette fois, les processus CSP, qui décrivent un peu le même genre de contrôle que dans les comportements des composants et connecteurs architecturaux, sont aussi destinés à être traduits en machines B standards à la fin de la phase de conception.

Mis en place dans l'idée de faciliter l'utilisation de la méthode B pour la conception d'applications complexes, le B événementiel (tableau II.4) a été conçu comme une extension au B classique avec des propriétés basées sur des événements. Le langage original de B est étendu avec de nouvelles caractéristiques facilement traduisibles en B standard, dont essentiellement une syntaxe basée événements. Le développement de modèle permet maintenant deux types d'observations : les états (amenant aux propriétés de sûreté) et les événements (pour ce que l'on peut observer, ou propriétés de vivacité). De plus, les raffinements (comme en B classique) peuvent être enchaînés avec des décompositions. Par conséquent, on peut voir dans ces dernières une espèce de raffinement horizontal explicite, même si elles ne s'intéressent pas à des composants et connecteurs architecturaux. Néanmoins, la relation de raffinement admet quelques différences respectivement à la méthode B : les obligations de preuves peuvent être séparées en deux ensembles, selon qu'elles concernent des états ou des événements. Fondamentalement, la spécification qui raffine l'autre génère un état plus précis et davantage d'événements.

Nom	B événementiel
Description de l'objectif	• L'approche du B événementiel a été conçue pour la modélisation de systèmes complexes, comme des extensions au B classique, en utilisant un développement suivant un modèle (ou aspect) ; • le B événementiel est donc une extension de la méthode B classique ; • la méthode B est appliquée successivement à la spécification du logiciel et son développement, mais une nouvelle approche est nécessaire pour concevoir des systèmes complexes (il faut au moins le même langage, avec un point de vue différent) ; • du fait de l'augmentation des complexités des systèmes distribués et des besoins de validation industriels qui y sont liés, la méthode B, basée sur du raffinement et des preuves, a été adaptée à la conception de systèmes, non pas en changeant le langage sous-jacent, mais en modifiant la façon de modéliser (une approche basée événements des systèmes clos) ; • certaines extensions de langage sont nécessaires pour pouvoir exprimer des propriétés basées sur les événements ; afin de conserver la compatibilité avec l'Atelier B, le langage B est étendu avec de nouvelles caractéristiques facilement traduisibles en B séquentiel, en utilisant des post-conditions, des modalités, une syntaxe basée événements et un raffinement explicite ; • la construction de modèle reflète ce qui peut être observé ; il y a deux types d'observations : les états (pour les propriétés de sûreté) et les événements (pour les propriétés de vivacité) ; des preuves sont requises pour valider ces propriétés ; • le paradigme du parachute repose sur le fait que d'autres choses intéressantes peuvent être observées à un niveau inférieur (un état plus précis et plus d'événements) ; • chaque étape de la construction globale doit assurer les preuves de ce que les parties de contrôle et de communication en construction préservent des lois élaborées uniquement à partir de l'état physique aux premières étapes du développement ; • lorsqu'il devient difficile de développer le modèle davantage, celui-ci est décomposé en plusieurs sous-modèles, basés sur du logiciel, du matériel ou en représentant un canal de communication ; pour les modèles basés logiciels, un ordonnanceur devrait être ajouté avant d'arriver au code, par des moyens traditionnels ou en achevant un developpement de logiciel en B ; • un autre outil B a été développé pour transformer un ensemble d'événements en un fragment de code en assemblant les événements ensemble ; il est donc capable de générer les algorithmes équivalents : les règles d'assemblage sont définies et appliquées sur un ensemble d'événements, que ce soit de façon automatique ou interactive ; • le B événementiel offre la possibilité d'enchaîner raffinements et décompositions.
Où	ClearSy http://www.clearsy.com
Bibliographie	Jean-Raymond Abrial, "Event Driven Sequential Program Construction", *EJC2000-JRA3 Ecole Jeunes chercheurs en programmation*, Ecole Normale Supérieure de Lyon, Lyon, mars 2000. Thierry Lecomte, "Event Driven B : methodology, language, tool support and experiments", *RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, Grenoble – IMAG, 22 janvier 2002.

Tableau II.4. : caractéristiques de l'approche du B événementiel

Quoi qu'il en soit, toutes les approches de raffinement de conception n'adoptent pas le même type de relation de raffinement. La réduction ("retrenchment"), présentée dans le tableau II.5, se base aussi sur des machines abstraites B au plus haut niveau d'abstraction, et étend la notation de la méthode B. Néanmoins, elle consiste dans le renforcement des préconditions des opérations et l'affaiblissement de leurs postconditions : la construction réduite est une simplification exagérée de la machine réduisante, c'est-à-dire que la réduction repose sur une relation de raffinement contraire à celle qu'il y a habituellement. Alors, certaines opérations nouvellement introduites dans la

machine qui réduit n'auraient pas de sens au niveau abstrait. De plus, la réduction permet une plus grande modification des types de données entre les opérations abstraites et concrètes, avec plus de flexibilité dans les paramètres en entrée et en sortie. Par conséquent, même si la réutilisation est plus facile, le résultat d'une réduction est la spécification d'un nouveau problème, décrite par une construction de haut niveau, c'est-à-dire une autre machine abstraite. Ainsi, alors qu'il est possible de considérer que le raffinement vertical est toujours présent, et aussi longtemps que cette relation de raffinement relie seulement une machine à une autre, ceci est vraiment discutable du fait que de nouvelles opérations peuvent être ajoutées, i.e. de nouvelles fonctionnalités, à la description d'un système. D'ailleurs, il n'est pas évident non plus qu'il puisse y avoir un raffinement compositionnel. Dans ce contexte, aucune génération de code ou de propriété compositionnelle comme celles de B ne peut être considérée, bien que la préservation de certaines propriétés inhérentes au système pourrait avoir lieu grâce à la présence de relations de recouvrement ou d'invariants.

Nom	réduction dans la méthode B
Description de l'objectif	• Le raffinement affaiblit la précondition et renforce la postcondition d'une opération : un comportement observable d'un raffinement est toujours un comportement possible de l'abstraction ; • B, Z et VDM placent l'explicitation des besoins au plus haut niveau d'abstraction et remplissent les détails manquant aux niveaux inférieurs ; • la réduction renforce la précondition et affaiblit la postcondition : la construction réduite est une simplification exagérée de la machine qui réduit ; • certaines opérations de la machine réduisante n'auraient pas d'intérêt au niveau abstrait ; • la réduction supporte la modification de type de données entre les opérations abstraites et concrètes, la flexibilité dans les composants d'entrée/sortie (les paramètres des opérations) ; elle introduit également une séparation entre les invariants locaux et la relation de recouvrement, ainsi que deux prédicats supplémentaires par opération de la machine réduisante paramétrée ; • la sémantique de la réduction tend à émuler le système abstrait ; • néanmoins les systèmes abstraits et concrets sont incompatibles ; • la réduction repose sur une relation très flexible entre les machines, pour construire des solutions complexes à des problèmes complexes en modifiant des parties très simplifiées mais plus compréhensibles ; • des extensions différentes ont été développées afin d'obtenir des propriétés de simulation différentes : ➢ le raffinement modulé ("modulated refinement") : cette extension introduit des signatures d'entrée/sortie différentes pour une notion conventionnelle de raffinement ; ➢ la réduction pointue ("sharp retrenchment") : plus les restrictions de la réduction ressemblent à du raffinement, plus les propriétés de simulation ressemblent à celles du raffinement modulé ; il en résulte la spécification d'un nouveau problème, amenant donc à une construction de haut niveau, c'est-à-dire une machine abstraite ; toutes les opérations doivent paraître ramifiées (la ramification concerne la façon avec laquelle l'opération concrète échoue à raffiner l'abstraite) ; il peut y avoir des opérations supplémentaires ; ➢ le raffinement modulé inverse ("reversed modulated refinement") : cette extension a été définie pour une notion conventionnelle de simulation forte ; ➢ la réduction $\neg$C : elle est proche du raffinement modulé ; ➢ la réduction $\exists$GVP : une condition de simulation pas à pas permet la complétude du système abstrait ;

	➢ la réduction décomposée : celle-ci est basée sur une re-mise en correspondance d'entrée/sortie.
Où	Computer Science Department, Manchester University, Royaume Uni
Bibliographie	R. Banach et M. Poppleton, "Retrenchment: An Engineering Variation on Refinement", *in Proceedings of B-98*, LNCS 1393, pp. 129-147, D. Bert (ed.), ISBN. 3540644059, 1998. R. Banach et M. Poppleton, "Sharp Retrenchment, Modulated Refinement and Simulation", *Form. Asp. Comp.*, 11, pp. 498-540, 1999.

Tableau II.5. : caractéristiques de l'approche de réduction pour la méthode B

La méthode de développement VDM (pour "Vienna Development Method"), dans le tableau II.6, a été conçue à l'origine pour la spécification formelle et le développement des aspects fonctionnels de systèmes, antérieurement à la méthode B. En fait, elle ne gère pas non plus les composants et connecteurs architecturaux. Par conséquent, aucun raffinement horizontal ne peut avoir lieu. C'est une méthode formelle orientée modèle basée sur une sémantique dénotationnelle, destinée à supporter le raffinement pas à pas de modèles abstraits en implémentations concrètes, mais sans mener à un code exécutable cependant.

Une spécification en VDM consiste en une description d'état (éventuellement augmentée avec des prédicats d'invariant et d'initialisation), une collection de définitions de domaines (auxquelles des invariants peuvent être ajoutés), une collection de définitions de constantes, une collection d'opérations et une collection de fonctions. La signification d'une spécification en VDM peut être vue comme un ensemble de modèles et la sémantique dénotationnelle est basée sur la construction de l'ensemble des modèles possibles.

La relation de raffinement de VDM établit qu'une implémentation doit tout simplement avoir une fonctionnalité dont il est possible de dire qu'elle implémente la construction spécifiée. Autrement dit, le comportement de l'abstraction doit être un comportement possible de la spécification concrète qui la raffine, comme dans le cas de la réduction étendant la méthode B. Cela ne peut pas véritablement être interprété comme un raffinement vertical, même s'il est basé sur le raffinement de fonctions et d'opérations, car des détails peuvent être ajoutés à la spécification de cette manière. D'ailleurs, les différents niveaux d'abstraction ne seraient pas clairement distincts. Néanmoins, ce raffinement comportemental et compositionnel préserve les propriétés internes des spécifications abstraites (à travers des invariants, …).

Nom	ISO/VDM-SL : la version standardisée du langage de spécification de VDM
Description de l'objectif	• Une méthode formelle est une approche mathématique pour la spécification d'un logiciel et son implémentation ultérieure, permettant des spécifications non ambiguës et des étapes de développement qui peuvent être prouvées comme étant correctes ; • par exemple, VDM (pour "Vienna Development Method") a été conçue pour la spécification formelle et le développement des aspects fonctionnels de systèmes ; • son élément central est VDM-SL, son langage de spécification formelle, utilisable pour des spécifications très abstraites, comme pour des spécifications à un très bas niveau d'abstraction ; un sous-ensemble exécutable du langage peut être défini ; • ISO/VDM-SL est la forme standardisée du langage de spécification formelle de VDM ; • la définition complète du standard pour VDM-SL peut être divisée en plusieurs composants principaux : la syntaxe (définie à différents niveaux d'abstraction), les représentations des symboles, la sémantique statique, la sémantique dynamique, la mise en correspondance ("mapping") de syntaxe ; • la syntaxe et la sémantique de ce langage formel ont été définies complètement et formellement ; • VDM est une méthode formelle orientée modèle basée sur une sémantique dénotationnelle : elle a été conçue pour supporter le raffinement pas à pas de modèles abstraits en

	implémentations concrètes ; • l'abstraction de la représentation est assurée au travers d'éléments de conception de données, basés sur six mécanismes de structuration de données mathématiques (ensembles, séquences, mises en correspondance – relations, objets composites, produits cartésiens et unions), sur de nombreux types numériques, les booléens, des jetons ("tokens"), sur des types énumérés et sur des types de données composées rassemblés sous le terme "domaines" (ceux-ci forment généralement des classes infinies d'objets, avec une structure mathématique spécifique ; le sous-typage est supporté en attachant des invariants de domaines aux définitions de domaines) ; • une abstraction opérationnelle est supportée par l'abstraction fonctionnelle, par des spécifications de fonctions (complètement transparentes pour les références) et une abstraction relationnelle par des spécifications d'opérations : fonctions et opérations peuvent être spécifiées soit implicitement en utilisant des pré- et postconditions, soit explicitement avec des constructions applicatives pour spécifier des fonctions et des constructions impératives pour spécifier des opérations ; • une spécification en VDM consiste typiquement en : une description d'état (éventuellement augmentée avec des prédicats d'invariant et d'initialisation), une collection de définitions de domaines (auxquelles des invariants peuvent être ajoutés), une collection de définitions de constantes, une collection d'opérations et une collection de fonctions ; • la sémantique dynamique est totale : un sens est attribué même aux constructions de spécification insignifiantes (⊥ est l'expression sans signification et une spécification entièrement insignifiante est représentée par un ensemble de modèles vide) ; • la syntaxe existe sous deux formes : des règles de réécriture EBNF (pour la syntaxe concrète d'un langage, elles permettent de générer automatiquement des analyseurs grammaticaux à partir d'une telle définition) et des définitions de types VDM-SL (la syntaxe abstraite externe, "Outer Abstract Syntax", est utilisable par des parties du standard défini en termes de fonctions VDM-SL et est utilisée par la sémantique statique ainsi que par les fonctions de correspondance syntaxiques) ; • la syntaxe abstraite centrale ("Core Abstract Syntax"), en plus de la syntaxe abstraite externe, donne une autre représentation de la syntaxe abstraite, moins compliquée, pour des spécifications VDM ; elle est utilisée pour la définition de la sémantique formelle (sémantique dynamique) ; • toute spécification VDM qui peut être représentée dans la syntaxe abstraite centrale reçoit une signification formelle par la sémantique dynamique ; la définition de la sémantique dynamique est basée sur la théorie des ensembles (des opérateurs sont utilisés pour construire un univers de domaine contenant toutes les "valeurs" valides qui peuvent être exprimées dans VDM-SL ; une collection de domaines sémantiques est définie sur cet univers de domaine) ; • la correspondance de syntaxe est due à la différence importante entre la représentation concrète d'une spécification en VDM-SL (syntaxe abstraite externe) et la représentation sous laquelle la signification d'une spécification a été définie (syntaxe abstraite centrale) ; le standard contient donc une définition formelle d'une correspondance de syntaxe depuis une spécification en termes de la syntaxe abstraite externe vers une spécification en termes de la syntaxe abstraite centrale ; • le raffinement n'est pas un point central : l'implémentation a juste besoin de fournir une fonctionnalité implémentant la construction abstraite correspondante (la sémantique dénotationnelle est basée sur la construction de l'ensemble des modèles possibles).
Où	Delft University of Technology, Faculty of Technical Mathematics and Informatics, Pays Bas IFAD, The Institute of Applied Computer Science, Odense, Danemark http://www.ifad.dk
Bibliographie	Nico Plat et Peter Gorm Larsen, "An Overview of the ISO/VDM-SL Standard", *ACM SIGPLAN Notices*, 1992.

Tableau II.6. : caractéristiques de VDM (son langage de spécification)

La notation Z (dans le tableau II.7) est également une approche orientée modèle mais elle utilise une relation de raffinement différente de la précédente. En réalité, en plus de ce que fait VDM, Z effectue une distinction entre les raffinements d'opérations, de données et de données fonctionnelles. Par conséquent, le raffinement recouvre un peu plus de choses, mais en fait ce raffinement est apparemment contraint par le cadre formel : comme dans la méthode B (qui serait développée ultérieurement à partir de l'expérience de Z), le raffinement vertical est plutôt limité à la modification des corps d'opérations, en exprimant plus précisément comment le résultat est obtenu. L'établissement de la relation de raffinement repose sur la vérification de conditions mathématiques (logiques) ; il s'agit à nouveau d'un test a posteriori, mais les obligations de preuve constituent indéniablement un moyen propice à la préservation de propriétés. Malgré tout, la relation de raffinement de Z, contrairement à VDM ou la réduction, suit la même idée que dans la méthode B classique : effectivement, elle s'exprime comme l'affaiblissement des préconditions de l'opération abstraite et le renforcement de ses postconditions (l'opération qui raffine l'autre est plus déterministe), à chaque fois que l'abstraction est en situation de terminer. Par conséquent, le raffinement fait montre d'un comportement possible de l'abstraction. Enfin, tout comme dans le cas de VDM, le raffinement de la notation Z est compositionnel et ne conduit pas non plus à la génération de code exécutable, sans distinction claire entre les niveaux d'abstraction.

Nom	Z
Description de l'objectif	• La notation de spécification formelle Z est basée sur la théorie des ensembles de Zermelo-Fraenkel et sur la logique des prédicats du premier ordre ; • Z supporte des raffinements d'opérations, de données et de données fonctionnelles, dans le même esprit que le raffinement en B : les préconditions sont affaiblies et les opérations deviennent plus déterministes : • un schéma Z est une signature, plus la propriété reliant les variables à la signature ; • des conditions (invariants) doivent être démontrées pour s'assurer de la terminaison et de la compatibilité des résultats ou des états initiaux (elles sont simplifiées lorsqu'il y a une fonction totale entre les états abstraits et concrets) ; • le développement d'un programme depuis une spécification obéit à deux types de décisions de conception : les opérations décrites par des prédicats dans la spécification doivent être implémentées par des algorithmes exprimés dans un langage de programmation ; les données décrites par des types de données mathématiques dans la spécification doivent être implémentées par des structures de données du langage de programmation ; • les règles pour un raffinement d'opération simple nécessitent de montrer qu'une opération est une implémentation correcte d'une autre opération avec le même espace d'états, lorsque les deux opérations sont spécifiées par des schémas ; ce genre de raffinement d'opération peut être étendu dans deux directions de façon à le rendre utilisable généralement, en introduisant des constructions du langage de programmation et par raffinement de données (introduction de structures de données orientées machine) ; • une opération concrète Cop, issue du raffinement d'opération d'une opération abstraite Aop, peut en différer de deux manières : la précondition de Cop peut être plus libérale que celle de Aop (Cop est garantie de terminer dans plus de cas que Aop), Cop peut être plus déterministe que Aop (pour certains états avant l'opération, il y aura moins d'états suivants possibles) ; • Cop doit être assurée de terminer lorsque Aop l'est ; si Aop est garantie de terminer, alors chaque état que Cop pourrait produire doit être un état que Aop pourrait produire (c'est-à-dire que si la précondition de Aop est satisfaite, alors tout résultat que Cop pourrait produire doit être un résultat possible de Aop) ; • le raffinement de données étend le raffinement d'opération en permettant à l'espace d'états des opérations concrètes d'être différent de l'espace d'états des opérations abstraites ; • il permet aux types de données mathématiques d'une spécification d'être remplacés par des types de données plus orientés machine dans la conception (une étape de raffinement de données relie un type abstrait de données, la spécification, à un type de données concret, la conception) ; le type de données concret peut alors être considéré comme un autre type de

	données abstrait (constitué d'un espace d'états et d'opérations décrites par des schémas) ; • afin de prouver que le type de données concret implémente correctement le type de données abstrait, il faut expliquer quels états concrets représentent quels états abstraits ; • un raffinement de données peut être correct seulement si deux conditions sont satisfaites pour chaque opération du type données abstrait (de façon analogue aux deux conditions du raffinement d'opération) : celles-ci doivent permettre la terminaison de l'opération concrète à chaque fois que l'opération abstraite est garantie de terminer et faire que l'état à la fin de l'opération concrète représente un des états abstraits dans lesquels l'opération abstraite peut terminer ; • une autre condition relie les états initiaux des types abstraits et concrets (chaque état initial possible du type concret doit représenter un état initial possible du type abstrait) ; • le schéma d'abstraction Abs entre les états abstraits et concrets définit une fonction totale des états concrets vers les états abstraits ; le raffinement de données fonctionnelles a un ensemble plus simple de conditions si le schéma d'abstraction est une fonction totale : la première condition est identique, la deuxième condition et la condition sur les états initiaux sont simplifiées (élimination du quantificateur existentiel) ; l'avantage est que la preuve que Abs est fonctionnel n'a besoin d'être faite qu'une seule fois par type (ce travail n'a pas besoin d'être répété pour chaque opération) ; • la relation de raffinement est similaire à celle de B ; • le raffinement est vérifié a posteriori, sur la base de preuves de conditions mathématiques (logiques) ; • il n'y a pas de construction de raffinement explicite ; • les différentes notions de raffinement ont quelques différences les unes avec les autres : le raffinement d'opération nécessite que les objets abstraits et concrets aient les mêmes espaces d'états et le raffinement de données a besoin d'espaces d'états différents.
Où	Programming Research Group, University of Oxford, Royaume Uni
Bibliographie	J. M. Spivey, *The Z Notation: A Reference Manual*, *Second Edition*, by Prentice Hall International (UK) Ltd, première publication 1992. Jim Davies et Jim Woodcock, *Using Z: Specification, Refinement and Proof*, Prentice Hall International Series in Computer Science, 1996. http://vl.zuser.org

Tableau II.7. : caractéristiques de la notation Z

A l'opposé de la méthode B, formelle et intégrant totalement le raffinement dans la spécification, Catalysis (tableau II.8) est une approche de conception semi-formelle qui, si elle permet de décrire bien plus de raffinements différents, ne le fait cependant que par le biais de documentations qui ne permettront pas de vérification de propriétés.

Nom	approche Catalysis basée sur UML
Description de l'objectif	• Catalysis est une approche conçue pour le développement UML, avec un raffinement a posteriori basé sur la vérification de la conformité d'une conception ou d'une implémentation relativement à sa spécification abstraite, grâce à une documentation plus ou moins formelle à propos des décisions liées au changement de niveau d'abstraction ; • la relation de raffinement doit être établie entre deux niveaux d'abstraction pour établir la traçabilité (des décisions de conception, …) depuis le cahier des charges jusqu'au code ; • une *conception* orientée-objet nécessite des techniques claires, conduites par des études de cas, pour transformer un modèle commercial en code orienté objet avec une approche centrée-interface et l'assurance d'une haute qualité ; la *réingénierie* demande des techniques pour comprendre le logiciel existant et pour en concevoir un nouveau à partir de lui ; • de façon à développer un ensemble cohérent de composants, il faut d'abord définir un

	ensemble commun de connecteurs (interfaces de haut-niveau) et des modèles communs de ce dont les composants discutent entre eux (le cadre de travail architectural) ; • Catalysis, pour pouvoir être utilisé dans un *développement basé-composants*, a besoin de savoir comment définir précisément des interfaces indépendamment de l'implémentation, comment construire l'architecture de l'ensemble de composants et les connecteurs de composants, ainsi que comment s'assurer qu'un composant est conforme aux connecteurs ; • une conception de haute intégrité demande des spécifications abstraites précises et une *traçabilité* non ambiguë depuis les objectifs industriels jusqu'au code du programme ; • *raffinement* et *conformité* sont des points essentiels dans une *revue de conception* de Catalysis ; • la relation de raffinement permet de dire si le code pour un composant est conforme à l'interface attendue par ses clients et de relier les besoins individuels aux spécificités d'une conception ; • la rigueur de la *traçabilité* a un degré variable : dans un contexte critique, les vérifications doivent être effectuées avec un niveau de détail mathématique ; dans des circonstances ordinaires, une documentation des points de correspondances principaux sert à la fois à guider et pour fournir une base pour la vérification, pour les revues de conception et les tests ; • les *tests* vérifient qu'une implémentation corresponde à sa spécification, avec le même objectif que le raffinement exception faite que les comportements à l'exécution sont suivis selon un ensemble dérivé systématiquement de jeux de tests ; • un *raffinement* est une description détaillée correspondant à une autre (son abstraction) : tout ce qui est vrai pour l'abstraction reste vrai, éventuellement sous une forme différente, aussi appelé réalisation ; le raffinement désigne aussi la relation entre les descriptions abstraites et détaillées ; "raffiner" signifie créer un raffinement, également appelé "concevoir", "implémenter" ou "spécialiser" ; • une *abstraction* est une description de quelque chose qui omet certains détails ne correspondant pas aux besoins de l'abstraction (la relation réciproque du raffinement) ; "abstraire" veut dire créer une abstraction, autrement appelé "généraliser", "spécifier" et quelquefois "analyser" ; • la *conformité* est définie par le fait qu'une description comportementale est conforme à une autre si (et seulement si) chaque objet se comportant comme décrit par l'une se comporte également comme l'autre le décrit (étant donnée une correspondance entre les deux descriptions) ; la conformité est la relation entre les deux descriptions, accompagnée d'une justification incluant la correspondance entre elles et le raisonnement qui a conduit aux choix effectués ; • raffinement et conformité forment les bases pour la traçabilité et documentent la réponse à la question : pourquoi la conception a-t'elle été faite de cette façon ? • le *recouvrement* est une fonction déterminant la valeur d'un attribut abstrait à partir des données enregistrées de l'implémentation et il est utilisé avec une conformité afin de montrer comment les attributs correspondent à l'abstraction et aux spécifications de comportement suivantes ; • une *implémentation* est un code de programme correspondant à une abstraction et qui ne nécessite pas davantage de raffinement (mais il se peut qu'elle doive passer par une compilation ou quelque chose d'équivalent) ; • un *sous-type* est défini a priori comme une extension du super-type avec toutes ses propriétés, voire plus ; le raffinement est un modèle autosuffisant même sans l'abstraction et il est supposé spécifier tout ce qui a été défini pour l'abstraction et même plus ; • un raffinement est documenté avec : les raisons menant au choix de cette réalisation parmi plusieurs alternatives ; les justifications de ce qui fait penser que le raffinement est correct ; le raffinement est une relation de plusieurs à plusieurs, donc la documentation est attachée à la relation de raffinement plutôt qu'à l'abstraction ou qu'au raffinement ; • Catalysis permet : la modélisation abstraite ; des instantanés pour animer la spécification (des

	diagrammes d'instances pour illustrer les effets de chaque opération, mais sans aucun message parce que ceux-ci sont décidés pendant le processus de conception) ; une implémentation, conçue pour faire plus, sous la forme d'une conception de haut niveau, d'une conception détaillée et de code par des raffinements successifs ; • la *comparaison* de deux spécifications d'opérations est constituée de : la conjonction des invariants pour les pré- et post-conditions (conjonction des paires de pré- et post-conditions de différents super-types pour une post-condition globale des besoins et leur composition est adéquate si elles ne viennent pas de super-types différents) ; la traduction du vocabulaire de l'implémentation ; la pré-condition des besoins doit impliquer celle de l'implémentation ; la post-condition de l'implémentation doit impliquer celle de la spécification (c'est la contravariance) ; • la justification associée au raffinement (*conformité*) peut être formelle ou informelle et elle concerne le recouvrement des attributs abstraits, la concordance du code séquentiel à une post-condition, la concordance de séquences d'actions détaillées à l'action abstraite, la concordance d'objets constituants à l'objet abstrait, les raisons des choix effectués ou rejetés, les spécifications et les données des tests (résultats compris) ; • Catalysis propose des directives pour l'application de techniques de raffinement, notamment : ➢ pour son patron supposant que le raffinement est une relation, et pas une séquence : pour une combinaison de développements descendants, ascendants, de l'intérieur vers l'extérieur ou basé sur de l'"assemblage" – respectivement "top-down", "bottom-up", "inside-out" et "assembly-based" – (cela n'implique pas un développement séquentiel descendant), il ne faut pas attendre qu'une phase complète soit achevée avant de passer à la suivante (il faut construire et utiliser ce pour quoi la technologie existe) ; la stratégie consiste à atteindre 80% de correction ("get it 80% right") de sorte à commencer la phase suivante lorsqu'il ne reste plus que 20% à terminer, car des résultats de la phase suivante peuvent servir de feed-back, pour utiliser des approches ascendantes et descendantes (l'important est d'avoir les bonnes relations de raffinement à la fin, pas l'ordre dans lequel elles ont été complétées), pour garder à l'esprit que même coder peut aider l'analyse (les gens connaissent mieux ce qu'ils ne veulent pas ; cela fournit rapidement une présentation, un prototype, … ; quelle information attendre ; que deviendra le code ; que deviendront les autres documents de conception) ; ➢ pour le patron basé sur un raffinement récursif : il faut établir une relation traçable entre la spécification plus abstraite et l'implémentation détaillée (le code) par une série de raffinements (la plupart étant des décompositions) ; la stratégie est de modéliser des vues séparées et de les composer en une spécification, de raffiner la spécification par un raffinement standard, de documenter chaque décomposition par un raffinement, d'utiliser une conception de base pour réduire le processus de raffinement, d'utiliser des cadres de travail pour construire les spécifications, les conceptions et les raffinement entre elles (parce qu'ils permettent de saisir les patrons récurrents de types, les collaborations et les raffinements, et de documenter à une place unique plutôt que partout) ; • la décomposition fait partie du raffinement ; • l'orthogonalité n'est pas expliquée de façon bien précise ; • il s'agit plutôt de recommandations dans le cadre d'un développement (orienté objet) ; • l'accent est mis sur la documentation à chaque étape de raffinement, à propos de toutes les décisions ou choix de conception, …
Bibliographie	Desmond Francis D'Souza et Alan Cameron Wills, Objects, Components, and Frameworks with UML the Catalysis Approach, Addison-Wesley, 1998. http://www.catalysis.org/books/ocf/index.htm

Tableau II.8. : caractéristiques de l'approche Catalysis

Certains langages de description d'architecture logicielle, tels que Darwin, MetaH, UniCon et Weaves, ne permettent qu'un type de raffinement limité, restreint à la génération automatique du code du système (cf. tableau II.9).

Nom	Darwin, MetaH, UniCon et Weaves
Description de l'objectif	• Ce sont des langages de description d'architecture logicielle ; • leur support au raffinement est limité à la génération automatique du code de l'application à partir de la spécification architecturale et de commentaires appropriés ; • il n'y a pas d'étape intermédiaire de raffinement ; • les spécifications sont déjà contraintes par des détails d'implémentation ; • seules des propriétés intrinsèques ("incorporées") des spécifications (orientées implémentation) peuvent être préservées.
Bibliographie	J. Magee, N. Dulay, S. Eisenbach et J. Kramer, "Darwin: An Architectural Description Language", Specifying Distributed Software Architectures, Department of Computing, Imperial College, Londres, septembre 1995. Paul C. Clements, "A Survey of Architecture Description Languages", Eight International Workshop on Software Specification and Design, Allemagne, mars 1996. *The Darwin Language*, version 3d, Department of Computing, Imperial College of Science, Technology and Medecine, Londres, 15 septembre 1997. N. Medvidovic et R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, janvier 2000.

Tableau II.9. : caractéristiques des langages de description d'architecture Darwin, MetaH, UniCon et Weaves

En fait, ces langages ne montrent pas de véritable étape intermédiaire distincte de raffinement : les spécifications sont déjà contraintes par des détails liés à l'implémentation. Dans ce cas, la relation de raffinement se réduit à une sorte de compilation, souvent assurée par un compilateur spécifique. Au lieu d'être effectué durant la phase de conception, le raffinement se prépare dans la spécification et est opéré de façon effective pendant la phase d'implémentation. Il pourrait alors être assimilé à un genre de raffinement vertical limité, qui préserverait des propriétés inhérentes des spécifications (orientées implémentation). Puisqu'aucun raffinement horizontal ne peut avoir lieu en même temps, il n'est pas possible d'affirmer qu'il existe une distinction claire entre raffinement horizontal et vertical dans ces approches.

Alors que les langages de description d'architecture logicielle présentés précédemment mettent en jeu un raffinement tardif pendant le processus de développement du système logiciel, Rapide (dans le tableau II.10) fournit un meilleur support pour le raffinement architectural à la phase de conception. Par ailleurs, il se sert d'événements, mais d'une façon différente que le B événementiel. En fait, il s'agit d'un langage de description d'architecture utilisant des patrons d'événements pour définir des correspondances entre architectures logicielles. Le raffinement dans Rapide est alors basé sur la comparaison de simulations d'architectures : les ensembles d'événements générés, ainsi que leur enchaînement (liens de cause à effet) et leur rythme. Par conséquent, ce genre de raffinement architectural concerne le comportement de l'architecture mais pas sa structure.

Nom	Rapide
Description de l'objectif	• Le langage de description d'architectures logicielles Rapide est un langage concurrent orienté objet basé sur les événements, conçu spécifiquement pour le prototypage d'architectures de systèmes, pour fournir des constructions pour la définition de prototypes exécutables et pour adopter un modèle d'exécution dans lequel des propriétés de concurrence, de synchronisation, de données et de flux temporels d'un prototype sont représentées explicitement ; • Rapide est basé sur les concepts d'architecture, d'interface et de connexion ;

	• une architecture est décrite en déclarant des interfaces de composants et les connexions entre elles, puis en spécifiant alors comment les interfaces réagissent aux événements qui leur sont présentés à travers les connexions ; • à ce niveau d'abstraction, l'exécution d'une architecture est représentée par l'ensemble de primitives échangées entre les interfaces au travers des connexions définies ; • une interface en Rapide est définie complètement par les primitives (actions et fonctions) – qui forment des services – qu'elle offre aux autres composants dans le système, et celles qu'elle requiert du reste du système pour effectuer les services offerts ; • le seul comportement à spécifier est celui qui relève de l'interaction d'interfaces, et pas le comportement du "module à venir" qui pourrait l'implémenter ; • le comportement d'une interface peut être exprimé dans Rapide par l'utilisation de règles réactives et en imposant des contraintes sur les événements générés ; • le module implémentant une interface donnée peut exécuter d'autres actions qui n'apparaissaient pas au niveau de détail de l'interface ; • la spécification du comportement de l'interface agit comme une ligne directrice et comme une contrainte pour le module à implémenter ; • Rapide fournit trois possibilités pour accomplir un raffinement, chacun correspondant à un niveau de détail et de raffinement différent : ➢ tout d'abord, au niveau de la spécification, des règles réactives associées aux interfaces constituent un moyen très concis de spécifier le comportement d'un module ; ➢ deuxièmement, afin de fournir plus de détails sur la façon dont le module fonctionne, une sous-architecture peut être associée à l'interface pour spécifier la structure interne, au moyen de sous-composants et de connexions ; ➢ enfin, Rapide dispose d'un ensemble complet de constructions de langage procédural qui permettraient d'implémenter complètement un module de façon conventionnelle ; • la correspondance entre des paires d'architectures s'intéresse à la façon dont les événements dans un système sont reliés aux événements dans un autre, même s'il s'agit de systèmes à des niveaux d'abstraction différents ; • elle permet également de vérifier la conformité d'un système à des standards ; • les patrons d'événements servent à définir des correspondances entre architectures en permettant des simulations comparatives et hiérarchiques ; • l'objectif d'une mise en correspondance est de définir comment les exécutions, d'une architecture ou plusieurs, prises ensemble (constituant le domaine de la correspondance), peuvent être interprétées comme des exécutions d'une autre architecture ou d'une interface (sa portée ou codomaine) ; • une mise en correspondance génère un nouvel ensemble partiellement ordonné ("poset") d'événements à partir d'un autre ensemble partiellement ordonné : elle est déclenchée par un ensemble partiellement ordonné du domaine, généré par son (ou ses) architecture(s) de domaine, et elle génère un nouvel ensemble partiellement ordonné cible, qui devrait être un comportement possible de l'architecture cible, mais généré par la correspondance plutôt que par l'architecture cible ; • ainsi, les dépendances entre événements de l'ensemble partiellement ordonné cible sont induites par celles de l'ensemble partiellement ordonné de domaine, même si certains événements de l'ensemble partiellement ordonné cible ne peuvent pas être comparés (sur l'aspect causal) aux événements de l'ensemble partiellement ordonné de domaine ; les ensembles partiellement ordonnés de codomaine, générés par une correspondance, doivent donc satisfaire les contraintes de leur architecture ou interface de codomaine ; • une mise en correspondance pose une contrainte sur ses paramètres de domaine, qui peuvent

	être des objets (généralement des architectures) ou d'autres mises en correspondance (dans le but de composer ensemble des mises en correspondance) ; • Rapide permet donc deux utilisations des mises en correspondance : comme contraintes ou comme domaines d'autres mises en correspondance ; • des applications particulières de correspondances de patrons d'événements incluent la diminution de la complexité de simulations de grande taille, en mettant en correspondance des ensembles partiellement ordonnés d'événements dans une simulation détaillée de bas niveau avec des événements séparés dans une simulation de haut niveau, et la comparaison pendant l'exécution d'une architecture avec une architecture standard (ou de référence) ; • la comparaison d'architectures est accomplie par la mise en correspondance des comportements de la (ou des) architectures de domaine avec des comportements de l'architecture de codomaine, et par la vérification de leur cohérence vis-à-vis des contraintes de l'architecture cible ; • lorsqu'une méthodologie de conception hiérarchique est utilisée pour développer une architecture détaillée de bas niveau à partir d'une spécification très abstraite, les mises en correspondance reliant des architectures de niveaux d'abstraction successifs peuvent être composées transitivement pour définir des mises en correspondance à travers plusieurs niveaux.
Où	Program Analysis and Verification Group, Computer Science Department and Electrical Engineering Department of Stanford University http://pavg.stanford.edu/rapide
Bibliographie	David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan et Walter Mann, "Specification and Analysis of System Architecture using Rapide", *IEEE Transactions on Software Engineering*, 21 (4) : 336-355, avril 1995. David C. Luckham et James Vera, “An Event-Based Architecture Definition Language”, *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734, 1995. David C. Luckham, James Vera, Doug Bryan, Larry Augustin et Frank Belz, “Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems”, *Journal of Systems and Software*, 21(3):253-265, juin 1993.

Tableau II.10. : caractéristiques du langage de description d'architecture logicielle Rapide

Le raffinement dans Rapide consiste alors en la comparaison a posteriori des deux traces d'événements : le raffinement est établi si la trace de la simulation de la spécification abstraite est incluse dans celle de la simulation de la spécification concrète ; il n'est donc pas réutilisable. De plus, la préservation des propriétés inhérentes aux systèmes, dans les correspondances de patrons d'événements, peut être vérifiée (seulement à un niveau d'abstraction d'écart), mais il n'est pas possible d'en vérifier d'autre. En outre, Rapide est composé de plusieurs sous-langages (pour la spécification abstraite, l'implémentation, etc.) ; la décomposition en sous-architectures pourrait être assimilée à du raffinement horizontal et l'implémentation pourrait être vue comme du raffinement vertical, bien qu'il n'y ait d'aménagement explicite ni pour le raffinement horizontal ni pour le vertical. Enfin, de telles comparaisons ne permettent pas la préservation de composition à travers un raffinement.

Alors que la plupart des approches proposent que le raffinement soit, ou bien un moyen d'obtenir un comportement possible de la spécification abstraite, ou alors au contraire une implémentation qui devrait couvrir plus de possibilités d'utilisation, une troisième alternative est d'obtenir un comportement similaire. Au moins un langage de description d'architecture fait partie de cette catégorie de relations de raffinement. SADL (pour "Structural Architecture Description Language") a été conçu pour décrire la structure architecturale d'un système logiciel (sa composition en termes de composants et connecteurs) et le raffinement de styles (cf. tableau II.11). Il ne s'intéresse pas au comportement du système, mais il permet de définir des patrons de raffinement réutilisables et qui peuvent être composés. Un système en SADL est complètement spécifié en utilisant un ou plusieurs styles et le raffinement sert fondamentalement à transformer un style en un autre. En fait,

les deux architectures mises en correspondance par le raffinement doivent permettre d'effectuer exactement les mêmes communications entre composants, ce qui a donc comme conséquence l'expression de hiérarchies de styles d'architecture logicielle et dans une certaine mesure la préservation de propriétés. Quoi qu'il en soit, cela ne peut pas conduire à de la génération de code. Bien que le raffinement de SADL soit limité à ce que les styles sont capables d'exprimer, il y a une distinction plutôt claire entre décomposition (en fait la description de composants composites statiques) et raffinement de style, donnant comme résultat ce qui pourrait être considéré comme des embryons de raffinements horizontal et vertical respectivement.

Nom	SADL (Structural Architecture Description Language)
Description de l'objectif	• SADL permet la description d'une structure architecturale (sa composition) et de raffinement de styles ; • il a été conçu pour la définition de hiérarchies d'architectures logicielles qui doivent être analysées formellement ; • la spécification et le raisonnement à propos de propriétés structurales de systèmes logiciels concernent la forme d'une architecture ainsi que les relations structurelles entre architectures qui sont supposées décrire la même famille de logiciels ; • SADL pourrait décrire à la fois une hiérarchie d'architecture verticale et une hiérarchie d'architecture horizontale ; • SADL effectue des liens entre différents niveaux de représentation en utilisant des mises en correspondance ; • SADL inclut un support pour des mises en correspondance explicites entre architectures : ces dernières comportent des architectures génériques, basées essentiellement sur des styles architecturaux (dont des contraintes de bonne formulation) et des patrons de raffinement d'architecture qui fournissent des solutions de routine à des problèmes de conception communs ; • ces patrons de raffinement décrivent des solutions réutilisables, qui peuvent être composées grâce à une propriété utile de mise en correspondance ; • SADL supporte des patrons de raffinement corrects : de nouveaux faits à propos de l'architecture abstraite composite ne doivent pas être inférés à partir de l'architecture concrète composite (l'interprétation pour le raffinement composite doit être fidèle à l'interprétation abstraite) ; • la correction du raffinement pourrait être prouvée en montrant qu'il s'agit d'une instance d'une composition de patrons, qui serait construite à partir de patrons de raffinement primitifs vérifiés, en utilisant des modes de composition prouvant la correction ; • un raffinement structural correct doit être fidèle : prouver la correction et trouver des modes de composition de raffinement préservant la correction deviennent d'autant plus difficiles ; • les transformations peuvent être vues comme étant appliquées à la hiérarchie tout entière (du même niveau d'abstraction) plutôt qu'à des descriptions individuelles à l'intérieur de la hiérachie (cela donne la même chose que la hiérachie produite par composition de raffinements) : le niveau concret complet est lui aussi un raffinement correct de la description du niveau abstrait ; • une approche incrémentale est utilisée à la place d'une approche compositionnelle : alors qu'une approche compositionnelle cherche à vérifier la correction d'une hiérarchie complète, l'objectif d'une approche incrémentale est d'éviter l'introduction d'erreurs au cours du processus de développement ; • dans une transformation incrémentale, à chaque décision de conception, la hiérarchie est tranformée pour refléter l'état actuel de la conception ; si toutes les transformations de la hiérarchie préservent sa correction et si le point de départ est une hiérarchie sans raffinement trivialement correcte, alors la hiérarchie complète en résultant est garantie comme étant correcte par construction ; • les preuves de correction des règles (de correspondance ou de transformation) et les preuves

	de correction de certaines étapes de raffinement sont donc étroitement reliées ; • une règle de transformation de hiérarchie d'architectures est correcte si et seulement si le résultat de chaque application à une hiérarchie correcte est une hiérarchie correcte : des règles de transformation correctes préservent la correction de la hiérarchie ; • prouver qu'une étape de raffinement est correcte devrait toujours garantir que la transformation effective qui étend une hiérarchie en utilisant ce raffinement sur une description complète homogène (basée sur un seul style) est également correcte ; • la composition est une méthodologie de développement évitant des descriptions architecturales non homogènes (c'est-à-dire utilisant des styles de niveaux d'abstraction différents), mais les remplacements doivent avoir lieu simultanément et le raffinement est complexe ; • des spécifications non homogènes ne peuvent donc pas être évitées, comme dans l'exemple de composition de raffinements en SADL suivant : deux éléments doivent être raffinés au niveau abstrait, l'un est d'abord raffiné et il en reste un autre à raffiner au niveau de raffinement 1, puis les éléments doivent être raffinés tous les deux au niveau de raffinement 2 ; les deux patrons de raffinement sont corrects, pris indépendamment l'un de l'autre, mais la hiérarchie ne l'est pas parce que le niveau 1 utilise les deux différents styles ; • une solution avec remplacement incrémental en SADL consiste à ne conserver aucun niveau intermédiaire : les deux éléments sont considérés comme raffinés à un niveau de raffinement 1' ; de cette manière, le raffinement correct implique une hiérarchie correcte.
Où	Computer Science Laboratory, SRI International, Etats-Unis http://www.sdl.sri.com/programs/dsa/sadl-main.html
Bibliographie	M. Moriconi, X. Qian et R. A. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, 21 (4) : 356-372, avril 1995. M. Moriconi et X. Qian, "Correctness and Composition of Software Architectures", *in Proceedings of ACM SIGSOFT'94: Symposium on the Foundations of Software Engineering*, pp. 164-174, New Orleans, Louisiane, décembre 1994. M. Moriconi et R. A. Riemenschneider, *Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies*, Technical report SRI-CSL-97-01, Computer Science Laboratory, SRI International, mars 1997. R. A. Riemenschneider, *Correct Transformation Rules for Incremental Development of Architecture Hierarchies*, Working Paper DSA-98-01, SRI CSL Dependable System Architecture Group, février 1998.

Tableau II.11. : caractéristiques du langage de description d'architecture logicielle SADL

Plus la méthode est formelle, plus son champ d'application est étendu. Le calcul de raffinement, présenté dans le tableau II.12, est une de ces méthodologies, qui peuvent être utilisées de façon appropriée soit durant la phase de conception, soit lors de l'implémentation. De plus, il peut servir pour la transformation de comportement, ou raffinement, mais aussi pour justifier une programmation si besoin.

Fondamentalement, le calcul de raffinement est une méthode pour développer des programmes impératifs, utilisée essentiellement pour transformer des spécifications écrites dans des extensions diverses du langage de Dijkstra en code assembleur. La compilation de programmes écrits dans un langage de haut niveau en code est alors représentée formellement dans le calcul de raffinement standard sous la forme de lois de compilation ou de raffinement. En fait, les règles de raffinement dérivées sont exprimées dans le calcul de raffinement usuel de Back et Morgan : basé sur le langage de commande gardé de Dijkstra, augmenté avec des énoncés de spécification, il sert de langage de modélisation général, à large spectre ; les deux langages, à la fois celui de départ de "haut niveau" et le langage "concret" cible, en sont alors purement et simplement des sous-ensembles distincts. Quelques règles bien connues de raffinement utilisées pour manipuler ce langage sont le renforcement de postcondition, l'introduction de variable locale (initialisée) ou

l'introduction de constante logique. Plusieurs autres règles de raffinement dérivées représentent des stratégies de compilation simples pour les énoncés du langage de haut niveau. La "compilation" de programme consiste alors en l'application répétée de telles règles. Elles ne permettraient pas pour autant un raffinement compositionnel. Des invariants peuvent également être définis ; ils comportent fréquemment des informations de types et sont maintenus automatiquement.

En fait, le raffinement pas à pas d'un programme est possible, même s'il est plutôt contraint par des détails d'implémentation. Par conséquent, un raffinement horizontal comme vertical peut avoir lieu, mais uniquement de façon occasionnelle comme des cas spéciaux de raffinements de données, fonctionnels ou comportementaux, puisque le calcul de raffinement ne gère pas les composants et les connecteurs architecturaux. En outre, l'application de lois de raffinement peut résulter en une génération de code et elles peuvent être réutilisées, éventuellement à travers plusieurs niveaux d'abstraction, mais les différents vocabulaires utilisés, représentant les différents niveaux d'abstraction, sont décrits dans le même langage. Enfin, la définition d'invariants peut assurer la préservation de certaines propriétés.

Nom	Calcul de raffinement
Description de l'objectif	• Le calcul de raffinement est une méthode pour développer des programmes impératifs ; • par exemple, le travail sur le calcul de raffinement à Oxford fut motivé par la recherche d'un moyen rigoureux et pratique de développement de programmes à partir de spécifications en Z ; les racines d'invariants locaux se retrouvent dans l'utilisation d'invariants dans un schéma Z (un invariant comporte fréquemment des informations de typage et est maintenu automatiquement) ; • tout comme cela se fait d'habitude avec l'utilisation du calcul de raffinement, le langage "de programmation" est défini en utilisant les constructions d'une extension du langage de Dijkstra ; • le but du développement est de raffiner un programme donné dans un sous-ensemble du langage, appelé code ; ceci est effectué par l'utilisation de lois de raffinement, tandis qu'il n'y a pas besoin d'utiliser directement les plus faibles préconditions – elles sont utilisées uniquement pour prouver les lois de raffinement (renforcement de postcondition, affectation, introduction de constante, …) ; • l'intention est de développer une stratégie de compilation vérifiée, digne de confiance, pour un langage de programmation simplifié, comme base pour, soit prouver (directement) la correction d'un code objet vis-à-vis de sa spécification, soit prouver (indirectement) la correction en vérifiant le compilateur ; • il est essentiel d'élargir le langage de modélisation pour englober le nouveau domaine d'application ; • les "lois de compilation" sont exprimées dans le calcul de raffinement habituel développé par Morgan ; • l'introduction d'invariants locaux fournit au calcul de raffinement un mécanisme pour décrire des prédicats qui sont maintenus à l'intérieur d'un bloc ; • une sémantique des invariants est utile pour l'extension du langage de Dijkstra ; • la signification d'une construction est donnée relativement à un invariant (appelé le contexte) ; • pour la plupart des constructions, la sémantique d'invariant est équivalente à la sémantique usuelle (non-invariant) si le contexte est supposé vrai ; • la sémantique d'invariant peut être étendue en donnant la signification du langage relative à un environnement ; une précondition la plus faible est établie dans le contexte I et l'environnement ρ d'un programme P pour établir la postcondition Φ ; • la relation de raffinement est établie entre P et Q si Q satisfait chaque spécification que P satisfait, quels que soient le contexte et l'environnement ; • si le raffinement est effectué dans un contexte, le renforcement de ce contexte ne peut pas

	invalider le raffinement ; • un raffinement de programme traduit habituellement une spécification abstraite en un programme dans un langage de haut niveau ; ce processus peut être mené plus loin en raffinant une telle "spécification" dans un langage (de programmation) de haut niveau en une "implémentation" en code assembleur, ce qui peut être effectué dans le cadre du calcul de raffinement, en utilisant plusieurs règles de raffinement dérivées pour modéliser la compilation de programme ; • le calcul de raffinement constitue un cadre théorique en treillis pour le calcul du raffinement de programme ; • spécifications et énoncés de programmes sont combinés en un langage unique qui permet l'utilisation d'énoncés miraculeux, angéliques et démoniaques dans la description d'un comportement de programme ; • l'approche de Back (son sous-ensemble "système d'action" du langage de commande gardé) peut être appliqué au problème de la compilation, pour développer une façon plus simple et plus uniforme de représenter la compilation comme un raffinement ; • des règles de raffinement dérivées sont exprimées dans le calcul de raffinement usuel de Back et Morgan : comme d'habitude dans le calcul de raffinement, le langage de modélisation général à large spectre est constitué du langage de commande gardé de Dijkstra, auquel sont ajoutés des énoncés de spécification (avec l'utilisation des lois et définitions habituelles du calcul de raffinement pour manipuler ce langage) ; les deux langages source de "haut niveau" et cible "assembleur" en sont alors simplement des sous-ensembles distincts ; • un interpréteur est nécessaire mais il introduit un écart significatif de paradigmes tout en essayant de modéliser la compilation comme un raffinement, puisque les langages source et cible sont représentés de façons très différentes ; • des règles de raffinement sont fournies pour traduire une "spécification", exprimée avec les énoncés de langage de haut niveau, en une "implémentation", exprimée en utilisant des instructions de bas niveau ; • en réalité, ces règles de raffinement définissent une stratégie de compilation de programme ; • des règles de raffinement connues pour la manipulation de ce langage permettent notamment de : renforcer la postcondition, introduire une variable locale (initialisée), introduire une constante logique, … ; • le calcul de la plus faible précondition est étendu pour couvrir cette classe plus large d'énoncés et une interprétation dans la théorie des jeux est affectée à ces constructions ; • le langage est complet, dans le sens où chaque fonction monotone transformant des prédicats peut être exprimée dans ce langage ; • les constructions de programme usuelles peuvent être définies comme des notions dérivées dans ce langage ; • la notion d'énoncés inverses est définie et utilisée dans la formalisation de la notion de raffinement de données.
Où	Department of Computer Science, Australian National University, Australie Programming Research Group, Oxford University, Royaume Uni Software Verification Research Centre, Department of Computer Science, University of Queensland, Australie California Institute of Technology, Etats-Unis Department Of Computer Science, Åbo Akademi, Finlande
Bibliographie	Trevor Vickers et Carroll Morgan, *Procedures and invariants in the refinement calculus*, technical report TRCS9404, Department of Computer Science, Australian National University, mai 1994. C. J. Fidge, *Modelling program compilation in the refinement calculus*, technical report No. 97-

	22, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1997. Karl Lermer et Colin J. Fidge, *Compilation as refinement*, technical report No. 97-29, Software Verification Research Centre, Department of Computer Science, University of Queensland, mai 1997. R.-J. R. Back et J. von Wright, *Refinement Calculus, Part I: Sequential Nondeterministic Programs*, 2 novembre 1989.

Tableau II.12. : caractéristiques du calcul de raffinement

3. Les critères de classification de raffinement

Les différentes méthodologies du génie logiciel présentées précédemment offrent autant de relations de raffinement. Nous en avons extrait seize critères, afin de différencier ces douze approches (cf. annexe A).

Trois critères fondamentaux suffisent à distinguer les principales catégories de relations de raffinement :

- la place du raffinement dans le cycle de vie,
- le type de processus de raffinement,
- la relation de raffinement.

Tout d'abord, chacune de ces approches va soit introduire une étape de raffinement durant la phase de conception – de façon à changer une spécification abstraite, architecturale ou pas, en une autre spécification plus concrète – soit conduire directement à une implémentation, se cantonnant donc à la phase d'implémentation ou de programmation. Dans un deuxième temps, le raffinement est quelquefois construit à partir de la spécification abstraite, en se basant sur des constructions spécifiques, et parfois il n'est vérifié qu'après coup, telle une propriété particulière du développement. Troisièmement, la relation de raffinement elle-même peut vouloir que le comportement concret soit un comportement possible du comportement abstrait (menant ainsi à une spécification plus détaillée du même problème), ou qu'il soit un comportement plus simple correspondant à un système plus général qui saurait gérer le problème abstrait spécifique, ou encore que le comportement reste inchangé.

En plus de ces principaux critères de discrimination, plusieurs autres peuvent servir à préciser de façon plus précise les différents aspects de la relation de raffinement. Par exemple, il peut y avoir un raffinement horizontal (décomposition) ou vertical (avec changement de niveau d'abstraction) et éventuellement une distinction claire entre les deux. Il est également possible qu'il y ait des constructions spécifiques pour un raffinement de données, un raffinement fonctionnel ou un raffinement comportemental, mais aussi que le raffinement soit compositionnel. En outre, toutes les méthodes ne conduisent pas d'une spécification abstraite jusqu'à la génération du code, tout comme elles pourraient ne pas permettre la distinction entre différents niveaux d'abstraction.

En dépit de ces aspects liés à l'action de raffiner, quelques critères supplémentaires concernent la réutilisation possible de parties du processus de raffinement et la préservation de propriétés, qu'elles soient inhérentes à la description du système ou définies par l'utilisateur. Finalement, de multiples langages ou vocabulaires peuvent être utilisés pour identifier les différents niveaux d'abstraction ou étapes de raffinement.

Par conséquent, ces critères peuvent servir à organiser les différentes approches dans des catégories en fonction de leurs relations de raffinement.

4. L'implémentation comme raffinement d'architecture logicielle

Comme cité précédemment, tout langage de description d'architecture supporte la décomposition d'une architecture logicielle – en ajoutant des détails concernant la façon dont un composant ou un connecteur fonctionne – mais peu d'entre eux sont capables de réaliser un raffinement d'architecture qui s'étend sur plusieurs niveaux d'abstraction [Medvidovic et Taylor 2000]. D'une part, certains langages, tels que Rapide et SADL, fournissent un support explicite au raffinement architectural, en établissant des correspondances entre des descriptions architecturales de différents niveaux d'abstraction. Cependant, ils ne permettent pas de générer automatiquement le code de l'application.

D'autre part, des langages comme Darwin, MetaH, UniCon et Weaves, présentés dans les tableaux II.9 et II.13, ne permettent qu'une forme limitée de raffinement, restreinte à une implémentation possible de la description du système qu'ils auront servi à spécifier. Cette étape de raffinement est de ce fait analogue à une compilation de la spécification.

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
implémentation	compilateur (excepté Weaves)	compilation

Tableau II.13. : critères principaux pour les langages de description d'architecture Darwin, MetaH, UniCon et Weaves

Par conséquent, cette unique étape de raffinement, souvent basée sur des commentaires de la spécification, peut être vue comme une étape de raffinement vertical. Cependant, plusieurs critères auxiliaires (tableau II.14) qui concernent le processus de raffinement n'ont pas vraiment de sens dans le contexte d'une implémentation (par exemple, la présence de constructions pour un raffinement horizontal, de données, fonctionnel ou comportemental, ou encore de langages de description différents).

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
NON	-	OUI	-	-	-	NON	OUI	NON	NON	OUI	NON	-

Tableau II.14. : critères auxiliaires pour Darwin, MetaH, UniCon et Weaves

5. Comparaison contre construction

Alors que les approches précédentes n'utilisent le raffinement qu'à la fin du processus de développement du système logiciel, d'autres méthodologies fournissent un meilleur support pour le raffinement d'architecture pendant la phase de conception. Néanmoins, elles ne procèdent pas toutes de la même manière.

Certaines de ces méthodes ou langages peuvent tirer avantage de constructions spécifiques pour le raffinement, permettant ainsi de construire la nouvelle conception de la spécification "concrète" qui raffinera la spécification originale "abstraite", à partir de celle-ci, en suivant certaines règles ou directives. Cependant, les plus anciennes adoptèrent une direction différente, en cherchant non pas à construire le raffinement mais plutôt à le vérifier.

Comme premier exemple, Rapide (tableaux II.10 et II.15) permet la simulation de descriptions d'architectures logicielles, en montrant les traces d'événements possibles pour l'exécution d'un système [Luckham et al. 1995]. Le raffinement est alors établi sur la base de la simulation de deux descriptions architecturales différentes, l'abstraite et la concrète, si la trace abstraite est incluse dans la trace concrète. Le raffinement en Rapide est donc effectué a posteriori, comme une comparaison de traces, après la définition des deux descriptions d'architectures logicielles.

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception	comparaison de traces a posteriori	inclusion de la trace abstraite dans la trace concrète

Tableau II.15. : critères principaux pour le langage de description d'architecture logicielle Rapide

Rapide ne fournit pas de mécanisme spécifique à un raffinement horizontal ou vertical (cf. tableau II.16). Les mises en correspondance indiquent uniquement quels patrons d'événements sont à comparer. Comme il n'y a aucune construction structurale, il est seulement possible d'observer un raffinement comportemental et des propriétés basées sur les ensembles partiellement ordonnés observés d'événements. Cette méthode utilise plusieurs sous-langages pour décrire des aspects différents de la spécification architecturale, tels que les interfaces ou les modules objets, mais Rapide ne fournit de moyen ni de générer du code, ni de réutiliser d'étape de raffinement. Par ailleurs, il n'est pas possible de distinguer des niveaux d'abstraction différents.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multi-langages**
-	-	-	NON	NON	OUI	NON	NON	-	NON	OUI mais juste 1 niveau	NON	plusieurs sous-langages

Tableau II.16. : critères auxiliaires pour Rapide

Parmi les autres méthodologies non-architecturales du génie logiciel disposant de raffinement, il en existe également certaines qui n'établissent une relation de raffinement qu'après que la spécification "concrète" ait été écrite.

Catalysis [D'Souza et Wills 1998], illustrée dans les tableaux II.8 et II.17, y tient une place spéciale en tant qu'approche semi-formelle pour un développement basé sur UML. Quoi qu'il en soit, cette méthode consiste essentiellement en recommandations pour pouvoir justifier, grâce à une documentation suffisante, le raffinement (soit horizontal, soit vertical) des spécifications (orientées objet).

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception	documentation a posteriori	documentation

Tableau II.17. : critères principaux pour l'approche Catalysis

Comme Catalysis repose essentiellement sur UML, le raffinement de comportement pourrait paraître assez limité, mais des raffinements de données et fonctionnels peuvent être facilement documentés à travers de multiples niveaux d'abstraction (cf. tableau II.18), bien que réduits à de la documentation. Pourtant, les raffinements verticaux et horizontaux ne sont pas distingués explicitement, et cette méthode ne permet pas d'automatiquement conserver les compositions, générer le code de l'application, réutiliser le raffinement ou préserver des propriétés. Enfin, Catalysis est une méthode semi-formelle, et ne dispose pas d'un langage spécifique.

Distinction RH / RV	Raffinement Horizontal (RH)	Raffinement Vertical (RV)	Raffinement de données	Raffinement fonctionnel	Raffinement comportemental	Raffinement compositionnel	Génération de code	Multiples niveaux d'abstraction	Réutilisation	Préservation de propriétés "inhérentes"	Préservation de propriétés "définies par l'utilisateur"	Multilangages
NON	OUI	OUI	OUI	OUI	limité	NON	NON	OUI	NON	NON	NON	-

Tableau II.18. : critères auxiliaires pour Catalysis

VDM ("Vienna Development Method", dans les tableaux II.6 et II.19) est une approche formelle pour le développement de logiciels, mais elle ne gère pas des composants et connecteurs architecturaux [Plat et Larsen 1992]. Le raffinement en VDM est essentiellement constitué de la comparaison de l'ensemble des modèles possibles de deux spécifications différentes. Par conséquent, les deux descriptions comparées doivent déjà exister. Fondamentalement, une fonctionnalité qui en raffine une autre n'a pas besoin d'implémenter tous les comportements possibles de son abstraction ; au contraire, elle devrait pouvoir faire au moins la même chose, mais dans éventuellement moins de cas (des préconditions plus fortes et des postconditions plus faibles).

Place du raffinement	Type de processus de raffinement	Relation de raffinement
conception	comparaison a posteriori des modèles possibles	l'implémentation doit avoir une fonctionnalité vue comme implémentant les constructions spécifiées de façon moins précise

Tableau II.19. : critères principaux pour VDM

Bien que de la décomposition puisse apparaître dans le développement de constructions de base de VDM, elle ne concerne pas des éléments architecturaux. Puisqu'elle n'a pas à gérer des composants et connecteurs, nous ne la considérerons pas comme un raffinement horizontal (cf. tableau II.20). En outre, la relation de raffinement de VDM permet l'ajout d'opérations et des changements de paramètres. Vu que la spécification est modifiée, cela n'est pas vraiment un raffinement vertical non plus. En fait, ces deux aspects semblent étroitement liés. D'autre part, les structures de données ne sont pas raffinées directement, alors que les fonctions et opérations qui les manipulent peuvent l'être.

En réalité, le raffinement dans VDM est essentiellement comportemental : il est basé sur la comparaison des modèles possibles pour les spécifications concernées (il est attendu de ces modèles possibles qu'ils soient préservés). Ainsi, il serait aussi compositionnel. Par ailleurs, les différents niveaux d'abstraction ne sont pas clairement distincts et il n'y aura pas de génération de code : le langage est plutôt homogène. Finalement, les étapes de raffinement, étant donné qu'elles ne concernent que des spécifications VDM particulières, ne peuvent pas être réutilisées pour alléger le travail de développement d'un autre système logiciel.

Distinction RH / RV	Raffinement Horizontal (RH)	Raffinement Vertical (RV)	Raffinement de données	Raffinement fonctionnel	Raffinement comportemental	Raffinement compositionnel	Génération de code	Multiples niveaux d'abstraction	Réutilisation	Préservation de propriétés "inhérentes"	Préservation de propriétés "définies par l'utilisateur"	Multilangages
-	-	NON pas réellement	NON	OUI	OUI (modèles possibles)	OUI	NON	pas très distincts	NON	OUI (modèles possibles)	NON	NON

Tableau II.20. : critères auxiliaires pour VDM

La notation Z [Spivey 1992] est également une approche orientée modèle. Sa relation de raffinement, décrite dans les tableaux II.7 et II.21, repose, comme pour la méthode B, sur l'affaiblissement des préconditions et le renforcement des postconditions (les opérations issues du raffinement doivent être plus déterministes, lorsque l'abstraction se termine). Le raffinement en Z devrait être établi sur la base de la preuve que les propriétés reliant les variables abstraites sont encore vérifiées au niveau concret.

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception	vérification a posteriori de conditions mathématiques (logiques)	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme), lorsque l'abstraction se termine

Tableau II.21. : critères principaux pour la notation Z

Comme pour VDM, il n'est pas vraiment possible de dire que Z supporte le raffinement horizontal (cf. tableau II.22) car elle traite le système dans son ensemble, et pas au niveau des composants et connecteurs. Néanmoins, le raffinement est principalement établi sur la base de corps d'opérations, arrivant ainsi à une sorte de raffinement vertical. En réalité, il affecte plusieurs aspects de la spécification, comme les structures de données ou, à travers la modification des corps d'opérations, les fonctions et, de façon plus limitée, les comportements. Les étapes de raffinement peuvent être composées, mais les niveaux d'abstraction ne sont pas clairement distincts : le formalisme ne permet pas de les différencier. Par conséquent, même si le développement en Z cherche à produire une implémentation, il n'y aura pas de génération directe du code. Enfin, puisque le raffinement est établi à partir de la preuve que les propriétés d'une spécification donnée sont préservées dans une autre, la réutilisation n'est pas envisageable.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
-	-	OUI	OUI	OUI	limité	OUI	NON	pas très distincts	NON	OUI (obligations de preuves)	NON	NON

Tableau II.22. : critères auxiliaires pour la notation Z

6. Différents raffinements constructifs pendant la conception

La troisième catégorie de systèmes de raffinement concerne les approches mettant à disposition des constructions spécifiques pour construire une spécification concrète à partir de la spécification abstraite : le raffinement n'a pas à être justifié a posteriori, il constitue en lui-même une phase de développement.

En tant que successeur de la notation Z, la méthode B [Abrial 1996] (également présentée dans [Ait-Ameur et al. 1998], [Bert et al. 1996] ou [Potet et Rouzaud 1998]) utilise la même relation de raffinement (cf. tableaux II.1, II.23 et II.24). Les différences proviennent essentiellement de la syntaxe du formalisme et du mécanisme de preuve. La méthode B est une approche formelle orientée modèle basée sur des machines abstraites et des obligations de preuve, avec des structures syntaxiques explicites pour exprimer une relation de raffinement entre une spécification et son implémentation. Les descriptions de modèles reposent essentiellement sur les invariants, qui caractérisent les attributs des variables et sur les différentes opérations qui modifient ces variables. D'une part, les machines abstraites constituent le niveau de spécification le plus abstrait de la méthode B, alors que les implémentations permettent la génération de code, qui peut être généré automatiquement en utilisant des outils commerciaux comme l'Atelier B ou le B Toolkit. D'autre

part, un module de raffinement en B est défini comme un différentiel à ajouter à un module B, soit une machine abstraite, soit un autre raffinement, jusqu'à ce qu'une implémentation soit générée. Plus précisément, un raffinement peut avoir des variables propres reliées aux variables du module raffiné par un invariant de recollement ; les opérations doivent dans ce cas être énoncées en fonction de ces nouvelles variables, mais en conservant exactement la même signature et en respectant les invariants de la spécification abstraite. Dans ce contexte, la relation de raffinement de B semble donc très proche de celle de Z : elle est basée sur l'affaiblissement des préconditions et le renforcement des postconditions.

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception	différentiel	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme) : comportement possible de l'abstraction

Tableau II.23. : critères principaux pour la méthode B

En outre, les obligations de preuve, du fait des invariants de recollement reliant la spécification raffinée à celle qui la raffine, permettent la préservation de propriétés inhérentes – qui sont en fait essentiellement les invariants des spécifications raffinées. Vu que le niveau des implémentations a un fonctionnement spécifique, avec des mots-clés réservés, le raffinement peut bien être compositionnel, excepté pour l'étape d'implémentation. En effet, la génération de code requiert l'utilisation d'un outil de développement approprié. Il en va de même pour le test de propriétés additionnelles, même si les outils standard n'ont pas vraiment été conçus pour cela. La méthode B ne permet pas l'ajout d'élément à la spécification, aussi ne peut-il pas y avoir de raffinement horizontal. En tout état de fait, le raffinement en B est purement vertical : il se cantonne essentiellement aux corps d'opérations et aux variables manipulées (limité ici à l'utilisation d'un invariant de recouvrement). Le raffinement d'un corps d'opération peut différer du corps abstrait ; néanmoins, l'interface doit rester la même et des obligations de preuve assurent de façon globale de conserver un comportement similaire. Enfin, la réutilisation d'un raffinement B est juste attachée à un raffinement multiple (mais implicite) de plusieurs modules simultanément : c'est trop restrictif pour être considéré comme une véritable possibilité de réutilisation.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
-	-	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	limité	OUI (excepté les implémentations)	OUI	OUI	NON	OUI (obligations de preuve)	NON (pas automatisée)	mots-clés réservés en fonction du niveau d'abstraction

Tableau II.24. : critères auxiliaires pour la méthode B

Par ailleurs, la méthode B a été le sujet de nombreux travaux différents ([Bon et al. 2000], [Marcano et al. 2000], [Sanlaville 1997] …). Le B événementiel [Abrial 2000], [Lecomte 2002], par exemple, est une extension du langage B classique avec un sucre syntaxique supplémentaire, utilisant des postconditions, des modalités, une syntaxe basée événement et des constructions de raffinement explicites. Néanmoins, cela ne change pas de façon drastique la relation de raffinement, comme illustré dans les tableaux II.4 et II.25 : le formalisme dispose d'un pouvoir expressif amélioré pour les spécifications, mais les opérations après raffinement doivent encore contenir des préconditions plus faibles et des postconditions plus fortes que dans leurs versions abstraites.

Place du raffinement	Type de processus de raffinement	Relation de raffinement
conception	différentiel (extension du B classique)	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme)

Tableau II.25. : critères principaux pour l'approche du B événementiel

Les propriétés ne sont plus établies que sur des états seulement, mais elles peuvent être maintenant établies sur des événements également. Un changement majeur repose sur le fait que les événements d'un module abstrait peuvent être décomposés dans un module plus concret. Cela constitue, d'une certaine manière, un genre de raffinement horizontal, bien que cela ne concerne pas des éléments architecturaux (cf. tableau II.26). D'autre part, cette version de la méthode de développement permet d'enchaîner des étapes de raffinement et de décomposition. Malgré tout, et puisque tout peut être traduit en B classique, les différentes étapes de raffinement peuvent toujours être vues comme compositionnelles.

Distinction RH / RV	Raffinement Horizontal (RH)	Raffinement Vertical (RV)	Raffinement de données	Raffinement fonctionnel	Raffinement comportemental	Raffinement compositionnel	Génération de code	Multiples niveaux d'abstraction	Réutilisation	Préservation de propriétés "inhérentes"	Préservation de propriétés "définies par l'utilisateur"	Multi-langages
NON	OUI (mais pas sous forme de composant et connecteur)	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	limité	OUI (excepté les implémentations) : possibilité d'enchaîner raffinements et décompositions	OUI	OUI	NON	OUI (obligations de preuve sur des états et des événements)	NON (pas automatisée)	mots-clés réservés en fonction du niveau d'abstraction

Tableau II.26. : critères auxiliaires pour l'approche du B événementiel

D'autres travaux intéressants autour de la méthode B se servent du même type de relation de raffinement ([Klenov et Pierre 2002], …). Les publications [Treharne et Schneider 1999] et [Treharne et Schneider 2000] proposent de décrire le contrôle des opérations B en utilisant un autre formalisme tel que l'algèbre de processus CSP : CSP permet d'exprimer le mode d'exécution du contrôle d'un système abstrait B (dans le tableau II.2). Effectivement, CSP et B sont concernés par des aspects différents de la description d'un système : CSP spécifie les séquences de transitions possibles et B spécifie individuellement chaque transition. L'environnement d'un module B est alors séparé en deux couches : la couche de contrôle CSP et l'environnement externe. Dans cette approche, la partie CSP est conservée dans tous les niveaux d'abstraction. Puisqu'elle n'est pas impliquée dans le raffinement, l'unique différence avec le raffinement classique en B est que les outils commerciaux pour le développement en B ne pourront pas être utilisés pour la génération automatique de code (cf. tableaux II.27 et II.28).

Place du raffinement	Type de processus de raffinement	Relation de raffinement
conception	différentiel (raffinement classique en B : la partie CSP est conservée dans les différents niveaux d'abstraction)	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme) : comportement possible de l'abstraction

Tableau II.27. : critères principaux pour l'approche combinant B et CSP

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
-	-	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	limité	OUI (excepté les implémentations)	NON	OUI	NON	OUI (obligations de preuve)	NON	OUI : CSP et un sous-ensemble de B

Tableau II.28. : critères auxiliaires pour l'approche combinant B et CSP

[Butler 1999] présente une autre approche basée sur les deux mêmes formalismes. CSP2B (cf. tableaux II.3 et II.29) fournit des descriptions ressemblant à du CSP, accompagnant des spécifications B. La notation CSP y sert uniquement à décrire l'ordre des opérations dans une machine B et elle peut être convertie en machines B standard. Comme toute machine CSP est équivalente à une certaine machine B standard, la relation de raffinement n'en est pas réellement affectée. Néanmoins, une telle description CSP peut être le raffinement d'une autre machine, en CSP ou en B : réciproquement, les spécifications abstraites et les raffinements peuvent être spécifiés soit avec CSP, soit avec CSP et B conjointement.

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception	différentiel (raffinement classique en B : les machines B et CSP sont raffinées indépendamment)	affaiblissement des préconditions et renforcement des postconditions

Tableau II.29. : critères principaux pour CSP2B

De plus, les machines CSP et B sont raffinées indépendamment l'une de l'autre, et la composition d'un processus CSP avec une machine B est donc compositionnelle vis-à-vis du raffinement (tableau II.30). D'autre part, CSP2B permet au moins une amélioration notable concernant le raffinement comportemental : l'association de CSP et B augmente le pouvoir expressif du formalisme et le raffinement conservera le comportement du module abstrait.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
-	-	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	OUI	OUI (excepté les implémentations)	OUI (conversion des machines CSP en B)	OUI	NON	OUI (obligations de preuve)	NON (pas automatisée)	OUI (CSP et B)

Tableau II.30. : critères auxiliaires pour CSP2B

Quoi qu'il en soit, toutes les approches constructives de raffinement à la conception n'adoptent pas la même relation de raffinement.

La réduction, comme présentée dans [Banach et Poppleton 1998] et [Banach et Poppleton 1999], en est une bonne illustration. Cette méthode est elle-aussi basée sur des machines abstraites B à son plus haut niveau d'abstraction, et son raffinement est encore défini en établissant la différence entre les modules raffinés et raffinants, mais avec une intention opposée (cf. tableaux II.9 et II.31).

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception	différentiel mais le résultat est la spécification d'un nouveau problème	renforcement des préconditions et affaiblissement des postconditions

Tableau II.31. : critères principaux pour l'approche de réduction pour la méthode B

Une conséquence majeure, visible aussi dans le tableau II.32, est que le module résultant de la réduction d'une machine abstraite est également une construction de haut niveau, c'est-à-dire une autre machine abstraite. Il n'y a alors qu'un seul niveau d'abstraction (en comparaison avec la méthode B classique), sans sous-langage spécifique ni génération de code. La réduction n'utilise de raffinement ni horizontal ni vertical : en effet, elle permet d'ajouter de nouvelles opérations ou de changer l'interface des autres. Des relations de recouvrement établissent, quant à elles, les liens entre les machines abstraites, mais ce raffinement n'est plus compositionnel.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
-	-	NON pas vraiment (ajout d'opération et changement de paramètres)	limité	OUI	limité	NON	NON	NON	NON	OUI mais limitée aux relations de recouvrement	NON	NON

Tableau II.32. : critères auxiliaires pour l'approche de réduction pour la méthode B

Une autre famille d'approches de raffinement importante est formée autour du calcul de raffinement, pour une programmation fine.

Le calcul de raffinement [Back et von Wright 1989], représenté dans les tableaux II.12 et II.33, est une méthode formelle qui fournit un cadre de travail mathématique pour le calcul de raffinement de programme. Le raffinement dans ce cas n'est pas limité à une phase spécifique du développement (telle que la conception ou l'implémentation). Il peut être utilisé soit pour la transformation d'un comportement [Lowe et Zedan 1995], soit pour la justification (a posteriori) d'une étape de raffinement ou de programmation. En réalité, le calcul de raffinement a de nombreuses applications possibles ; celles-ci sont définies par des invariants et par des lois de raffinement (établissant typiquement le renforcement de postconditions).

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception / implémentation	transformation et raffinement de comportement ou justification de programmation	lois de raffinement (renforcement de postconditions, …) et invariants

Tableau II.33. : critères principaux pour le calcul de raffinement

Les spécifications et les énoncés de programme sont combinés en un unique langage de commandes (infini) qui permet d'utiliser des énoncés miraculeux, angéliques et démoniaques pour la description du comportement d'un programme. Le calcul de la plus faible précondition est étendu pour couvrir cette classe plus large d'énoncés et une interprétation dans la théorie des jeux est

associée à ces constructions. Le langage qui en résulte est complet, dans le sens qu'il peut exprimer tout élément transformant un prédicat de façon monotone, et les constructions de programmation usuelles peuvent être définies comme des notions dérivées dans ce langage. La description de la notion d'énoncé inverse est également possible pour être utilisée dans la formalisation de la notion de raffinement de données. Au départ, le calcul de raffinement est une méthode pour le développement de programmes impératifs. Le raffinement pas à pas d'un programme est possible, mais essentiellement contraint par des détails liés à l'implémentation. Par conséquent, peuvent avoir lieu du raffinement horizontal ou du raffinement vertical, mais seulement à l'occasion de cas particuliers de raffinement de données, de fonctions ou de comportement (cf. tableau II.34). D'autre part, l'emploi de lois de raffinement peut avoir comme résultat de la génération de code, et elles peuvent être réutilisées, éventuellement à travers plusieurs niveaux d'abstraction, mais les différents vocabulaires utilisés sont finalement représentés dans le même langage. Enfin, la présence d'invariants assure la préservation de certaines propriétés.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
NON	possible	possible	OUI	OUI (utilisation d'invariants locaux)	OUI	NON	OUI (lois de raffinement)	OUI	OUI (lois de raffinement)	OUI (invariants)	NON	NON (mais différents vocabulaires)

Tableau II.34. : critères auxiliaires pour le calcul de raffinement

Au bout du compte, le calcul de raffinement est surtout utilisé pour transformer des spécifications écrites dans diverses extensions du langage de Dijkstra en code assembleur. Les "lois de compilations" dans le calcul de raffinement standard sont la représentation formelle de la compilation des programmes de langages de haut niveau en code assembleur, et de nouveaux opérateurs sont introduits pour élargir le langage de modélisation afin qu'il englobe le code assembleur. Par exemple, [Vickers et Morgan 1994] propose une sémantique sous forme d'invariants pour une extension de ce type qui ajoute des procédures : la signification d'une construction est également donnée relativement à un invariant (appelé le contexte), mais la sémantique de cet invariant est étendue en donnant la signification du langage en fonction d'un environnement. Le raffinement fonctionne encore dans un tel cadre : renforcer le contexte ne peut pas invalider le raffinement. L'objectif du développement reste toujours de raffiner un programme donné, en utilisant des lois de raffinement (renforcement de postcondition, affectation, introduction de constante, …), vers un sous-ensemble du même langage, appelé code. Cependant, les plus faibles préconditions sont uniquement utilisées pour prouver les lois de raffinement. Les invariants portent fréquemment en eux des informations de typage, et sont maintenus automatiquement ; l'introduction d'invariants locaux confère alors au calcul de raffinement un mécanisme pour décrire des prédicats qui ne sont maintenus qu'à l'intérieur d'un bloc d'instructions.

Dans [Fidge 1997], le but général poursuivi est le développement de techniques formelles de compilation en code assembleur pour des programmes temps-réels, en utilisant le calcul de raffinement. En fait, une stratégie de compilation est imbriquée sous la forme d'un ensemble de règles de raffinement dérivées. Habituellement, le calcul de raffinement traduit une spécification abstraite des besoins en une implémentation dans un langage de programmation, en utilisant le langage de commandes gardé de Dijkstra amélioré avec des énoncés de spécification pour en faire la notation de modélisation (les lois et définitions usuelles du calcul de raffinement sont utilisées pour manipuler ce langage). Dans le contexte de la compilation, une "spécification" est un programme dans un langage de haut niveau et une "implémentation" est un bloc de code assembleur. Le langage de modélisation est donc élargi pour prendre en compte le nouveau

domaine d'application. La présence d'un interprète introduit un changement de paradigme non négligeable lors de tentatives pour modéliser la compilation comme un raffinement, vu que les langages sources et cibles sont représentés de façons très différentes.

De la même manière, [Lermer et Fidge 1997] considère que le raffinement de programme traduit généralement une spécification abstraite en un programme dans un langage de haut niveau, et que ce processus peut être conduit plus loin en raffinant une "spécification" dans un langage de haut niveau en une "implémentation" de code assembleur. L'objectif de départ est de développer une stratégie de compilation vérifiée et fiable pour un langage de programmation simplifié, en tant que base pour soit prouver (directement) la correction du code relativement à sa spécification, soit prouver (indirectement) la correction en vérifiant le compilateur. Par conséquent, modéliser la compilation de programme comme un processus de développement formel devient un sous-objectif. Néanmoins, même si un interprète s'avère utile, sa présence implique un changement de paradigme non négligeable lors de tentatives pour modéliser la compilation comme du raffinement – les langages sources et cibles sont représentés de façons très différentes à nouveau. Cependant, Back démontra à travers une étude de cas que son sous-ensemble du langage de commandes gardé en "système d'actions" est capable de représenter les programmes de type assembleur. Ainsi, les règles de raffinement dérivées sont exprimées dans le calcul de raffinement usuel de Back et Morgan : le langage de commandes gardé de Dijkstra, auquel ont été ajoutés des énoncés de spécification, sert de langage de modélisation général, à large spectre ; alors, à la fois les langages sources de "haut niveau" et cible "assembleur" en sont des sous-ensembles à peine distincts. Les règles de raffinement habituelles sont également utilisées pour manipuler ce langage, essentiellement à des fins de : renforcement d'une postcondition, introduction (ou initialisation) d'une variable locale, introduction d'une constante logique. Plusieurs autres règles de raffinement dérivées représentent des stratégies de compilation simples pour les énoncés du langage de haut niveau. La "compilation" de programme consiste alors en une application répétée de telles règles. Il en résulte un modèle constitué de boucles imbriquées, qui peuvent être dépliées et optimisées afin d'obtenir le programme assembleur final. Enfin, les règles de compilation, via du raffinement de programme, pour les programmes temps-réel, sont vérifiées.

Les approches précédentes proposent de voir le raffinement comme un moyen d'obtenir soit une spécification concrète adoptant un comportement possible de la spécification abstraite, ou au contraire une implémentation qui pourrait offrir plus de fonctionnalités dans éventuellement moins de cas d'utilisation. Une troisième possibilité est d'obtenir un comportement similaire. Au moins un langage de description d'architecture appartient à cette catégorie particulière de relations de raffinement. SADL (pour "Structural Architecture Description Language") a été conçu pour la description de la structure architecturale d'un système logiciel (sa composition en termes de composants et connecteurs) et celle de raffinement de styles architecturaux [Moriconi et Qian 1994], [Moriconi et al. 1995], [Moriconi et Riemenschneider 1997]. Il ne s'intéresse pas, par contre, à l'expression du comportement des composants et connecteurs, mais il permet de définir des patrons de raffinement réutilisables et composables. Un système en SADL est complètement spécifié en utilisant un ou plusieurs styles, et le raffinement sert tout simplement à passer d'un style à un autre. En effet, les deux architectures mises en correspondance par le raffinement doivent autoriser de faire exactement les mêmes communications entre composants (voir dans les tableaux II.11 et II.35) ; il en découle donc l'expression de hiérarchies d'architectures logicielles.

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception	mise en correspondance de styles	traduction selon la mise en correspondance : exactement la même chose

Tableau II.35. : critères principaux pour le langage de description d'architecture logicielle SADL

Quoi qu'il en soit, cela ne peut pas conduire à la génération du code, comme cela est rappelé dans le tableau II.36. Même si le raffinement de SADL est limité à ce que les styles sont capables d'exprimer (tout spécialement en ce qui concerne les structures de données et les fonctions), la distinction entre la décomposition (en fait la description statique de composants composites) et le raffinement de styles est assez claire, ce qui pourrait finalement être interprété comme des embryons de raffinement horizontal et vertical, respectivement. Des étapes de raffinement

différentes peuvent être composées du moment qu'il n'apparaît pas de cycle dans les changements de styles. Un avantage important de la mise en correspondance de styles est la réutilisation aisée qui peut être faite des patrons qui la décrivent : à l'intérieur d'un même développement, un patron de raffinement peut être appliqué de nombreuses fois pour accomplir la même étape de raffinement, de sorte à aboutir à une description architecturale homogène. Enfin, le raffinement dans SADL préserve les quelques propriétés qui peuvent être exprimées par l'architecture abstraite.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
distinction entre décomposition et raffinement de style	composants composites statiques	changement de styles architecturaux entiers	pour ce qui peut être défini dans les styles	pour ce qui peut être défini dans les styles	NON	OUI	NON	si les styles sont considérés ainsi	OUI patrons	OUI	NON	NON

Tableau II.36. : critères auxiliaires pour SADL

7. Conclusion

En fait, le génie logiciel ne comporte qu'un nombre limité de relations de raffinement possibles. Certaines spécifications peuvent être compilées pour obtenir du code exécutable, mais elles sont, dès le départ, très liées à l'implémentation. Une deuxième catégorie de méthodologies, rassemble à la fois des approches formelles et semi-formelles, qui permettent de comparer des descriptions abstraites et concrètes entre elles au cours de la phase de conception pour vérifier leur conformité. Leur principal défaut réside dans le fait qu'elles doivent s'écarter d'un niveau de structure architecturale afin de se focaliser sur un grain plus fin de représentation. Enfin, une troisième famille d'approches, plus intéressante dans le cadre de nos travaux, est capable d'élaborer la spécification qui raffine, à partir de la spécification abstraite à raffiner, en se conformant à des règles explicites de construction correcte. Ces règles sont d'autant plus importantes qu'elles régissent la notion de conformité du "comportement raffinant" vis-à-vis du "comportement abstrait" : elles décident si le raffinement doit permettre de faire une action plus déterministe (dans éventuellement plus de cas possibles, comme par exemple dans le cas de la méthode B), une fonctionnalité plus complexe (mais qui couvre au moins ce que l'abstraction spécifie, comme par exemple pour la réduction dans la méthode B), ou bien exactement autant de choses qu'au niveau abstrait (comme dans l'exemple de SADL).

Nous pouvons encore noter que les relations de raffinement présentées ne montrent pas de distinction claire entre les vocabulaires utilisés dans la description des spécifications de niveaux d'abstraction différents, les mécanismes de décomposition et les constructeurs de raffinement.

D'un côté, la décomposition architecturale est indubitablement au cœur des intentions du raffinement horizontal : la spécification, une fois décomposée, contient de nouveaux détails. De l'autre côté, le raffinement vertical est supposé lui être orthogonal : il ne devrait pas être vu comme menant à une description plus précise, mais plutôt à une description plus "implémentable".

Bien que la décomposition soit un aspect assez répandu parmi les langages de description d'architectures logicielles, cette différenciation entre ces deux formes de raffinement n'y est pas encore supportée. Cela n'est pas sans conséquence : le raffinement est rarement réutilisable d'un système à l'autre. En effet, la plupart des méthodologies étudiées entendent par étape de

raffinement soit la décomposition de certaines parties de la spécification, soit un changement de niveau d'abstraction pour s'approcher de l'implémentation, soit un mélange des deux. Le calcul de raffinement (tableaux II.33 et II.34) permet de définir des lois de raffinement génériques qui peuvent servir pour le développement de n'importe quel système auquel elles peuvent s'appliquer ; SADL (tableaux II.35 et II.36) décrit également des patrons génériques pour le raffinement. La réutilisation est donc étroitement liée aux notions de règles de construction et de généricité.

Ces deux approches illustrent assez bien ce dont nous avons besoin pour compléter un développement formel. En effet, les méthodes formelles de développement manquent de flexibilité, notamment pour détailler certaines parties spécifiques de la spécification d'un système, par décomposition. A l'opposé, les langages de description d'architecture de haut niveau sont plus intuitifs à mettre en œuvre, et permettent de modéliser les interactions à l'intérieur du système ; toutefois, ils ne possèdent pas de base sémantique qui puisse en même temps garantir la cohérence d'une spécification, et la conservation de son comportement par des transformations menant à la production du code de l'application.

A la différence de méthodes orientées modèles, comme la méthode B, qui modélisent puis manipulent le système dans son intégralité, une spécification architecturale permet de s'intéresser à des constituants élémentaires de la description, indépendamment de leur contexte. L'expression de leurs transformations possibles sous forme de règles de raffinement, à l'instar du calcul de raffinement, nous autoriserait pareillement à ne traiter que ces éléments de base. Néanmoins, nous avons également besoin, comme dans le cas de SADL, d'avoir une cohérence à un niveau plus global, afin que le raffinement de la spécification du système ne soit pas mené de façon anarchique. C'est pourquoi nous proposons de regrouper ces règles de raffinement dans des ensembles, tels les patrons génériques et réutilisables de SADL. Chaque patron peut alors servir au raffinement d'une spécification écrite initialement dans le langage de description d'architecture source, pour obtenir une spécification dans le langage de modélisation de la méthode formelle cible, avec une base sémantique formelle assurant la vérification de la conservation de propriétés du système.

Ainsi, des patrons de règles de raffinement nous permettent à la fois de traduire les différentes constructions d'un langage de description d'architecture pour se rapprocher d'un niveau d'abstraction d'implémentation, tout en garantissant la conformité des raffinements, et de faire bénéficier une méthode formelle de développement d'une étape de conception plus "ergonomique".

Nous rappelons dans le chapitre 3 les caractéristiques des différents formalismes que nous sommes amenés à utiliser dans le cadre de la mise en œuvre de nos travaux :

- un langage de description d'architecture logicielle, π-SPACE,
- une méthode formelle de développement, B,
- un langage formel pour la représentation syntaxique et sémantique des règles des patrons de raffinement, la logique de réécriture.

Nous présentons ensuite, dans le chapitre 4, l'approche que nous proposons pour traiter le raffinement vertical d'une description d'architecture pour obtenir une spécification dans une méthode de développement formel, qui pourra par la suite mener à la génération de code.

Chapitre 3 : Formalismes utilisés

Chapitre 3 : Formalismes utilisés

1. Introduction

Avant d'entamer la présentation de notre approche, il est utile de rappeler les caractéristiques des différents formalismes existants sur lesquels nous nous appuyons. En effet, si le chapitre précédent présentait un état de l'art sur les différentes méthodologies de raffinement dans un cadre assez général, nous nous intéressons à un problème bien plus particulier.

Comme nous l'avons déjà évoqué dans le chapitre 1, nous proposons de nous appuyer, en amont d'un développement formel classique, sur une description architecturale, dans un langage de haut niveau, que nous raffinerons dans le langage formel cible de la méthode formelle classique choisie. Dans un premier temps, une description architecturale du système est issue de la phase de l'analyse des besoins. Plutôt que de raffiner cette description architecturale jusqu'à l'implémentation dans un cadre purement architectural, nous la raffinerons pour obtenir une spécification abstraite dans un langage formel qui conduira par la suite à l'implémentation en utilisant un développement formel classique.

Le langage de description d'architecture logicielle qui nous intéresse est fondé sur une algèbre de processus, le π-calcul [Milner 1999] ; ceci confère à π-SPACE un pouvoir expressif important, qui permet notamment l'expression d'architectures dynamiques. Le langage de spécification formelle est celui de la méthode B ; cette méthode a en effet le grand avantage de disposer d'outils pour aider au développement formel et à la génération du code.

Par ailleurs, il nous faut également définir comment passer d'un langage à l'autre, c'est-à-dire comment la description d'architecture abstraite va être raffinée pour donner la spécification formelle concrète. C'est pourquoi nous proposons de mettre en place un processus formel de raffinements successifs, écrit dans un langage formel approprié. Nous devons décrire formellement cette série de raffinements par un modèle de processus.

D'une part, nous avons vu dans le chapitre précédent que certains langages de description d'architecture logicielle, tels SADL ou Rapide, prennent en compte le raffinement mais ils n'offrent pas un support suffisant pour le développement complet de systèmes logiciels complexes. Actuellement, les méthodes utilisant le raffinement nous restreignent, soit à l'obligation de prouver après chaque étape la préservation de propriétés avec le niveau d'abstraction précédent, soit à la construction de la nouvelle spécification concrète à partir de la spécification abstraite. Cette dernière catégorie est la plus intéressante, puisqu'en garantissant que chaque étape de raffinement conserve les propriétés de la description architecturale de départ, nous pouvons nous abstenir d'une phase de vérification a posteriori.

D'autre part, rappelons que notre processus de raffinement doit, à chaque étape, assurer de façon systématique la construction correcte du système et la conservation des propriétés sémantiques, afin de nécessiter le moins possible d'interventions de la part de l'utilisateur. La représentation formelle du raffinement d'architectures logicielles requiert un formalisme dans lequel il soit possible de représenter simplement les éléments architecturaux abstraits et les interactions entre les différents composants et connecteurs qui constituent l'architecture du système, mais aussi les constructions du langage de spécification concret.

Nous avons déjà constaté, dans le chapitre d'introduction à cette thèse, qu'une telle approche s'effectue de façon naturelle en traitant le raffinement par des réécritures successives de la spécification abstraite pour substituer aux constructions architecturales des éléments de machines abstraites B. Notre choix se porte sur l'utilisation d'une logique supportée, elle aussi, par un outil de développement formel : la logique de réécriture.

Dans la suite de ce chapitre, nous présenterons tout d'abord le langage de description d'architecture logicielle π-SPACE. Nous poursuivrons avec la méthode de spécification formelle B avant d'introduire la logique de réécriture. Un lecteur familier avec l'une ou l'autre de ces approches peut s'abstenir de lire les parties qui ne lui apporteraient rien.

2. Le langage de description d'architecture logicielle π-SPACE

2.1. Introduction

Le langage de description d'architecture logicielle π-SPACE fut conçu pour prendre en compte, d'une part les aspects dynamiques, et d'autre part les aspects évolutifs d'une architecture.

Ce langage se place dans un cadre compositionnel et évolutif pour la description et la décomposition d'architectures dynamiques à composants :

- le cadre compositionnel permet d'utiliser des bases de composants qui peuvent être assemblés pour construire une architecture ;
- le cadre évolutif permet de décomposer une architecture et de la recomposer avec l'ajout ou le retrait de composants, sans interrompre l'utilisation de l'application ;
- la décomposition permet de détailler un comportement ;
- enfin, une architecture dynamique comprend des composants dynamiques, c'est-à-dire des composants capables d'être créés et attachés aux composants existants au cours de l'utilisation de l'application.

Le développement d'une application logicielle utilise soit des composants spécifiques au cadre de l'application, soit des composants décrits préalablement et provenant d'une base d'éléments architecturaux. En effet, ce sont des éléments autonomes qui peuvent être incorporés à n'importe quel niveau d'abstraction – plus ou moins détaillé – de l'architecture. Il est possible, dans un premier temps, de réaliser cette spécification seulement avec les fonctions principales du système, avant de détailler les différentes sous-parties de l'architecture.

A tout moment de cette spécification, des tests de comportement de l'architecture peuvent être effectués, dès la prise en compte des premières fonctions, permettant ainsi de valider les besoins qu'elle doit satisfaire.

Après une présentation du π-calcul, l'algèbre de processus qui sert de base sémantique au langage, nous introduirons les différents éléments d'une description d'architecture en π-SPACE. Pour plus de détails, le lecteur peut se reporter à [Chaudet 2002].

2.2. Le π-calcul

2.2.1. Introduction sur le π-calcul

Les algèbres de processus constituent des méthodes formelles pour modéliser des interactions entre processus. Elles permettent, en construisant un modèle mathématique reprenant certaines descriptions du modèle d'analyse de l'application (celles relatives aux interactions), de garantir la cohérence du modèle et la conformité du programme.

π-SPACE a été fondé sur le π-calcul, plutôt qu'une autre algèbre de processus, car cette algèbre introduit le concept de mobilité, c'est-à-dire la possibilité de faire évoluer dynamiquement la topologie des applications. Il existe différentes versions du π-calcul. Celle choisie pour π-SPACE est le π-calcul synchrone, polyadique. Ces nuances seront expliquées dans le paragraphe 2.2.4.

Le π-calcul permet de représenter des communications entre des processus. Par exemple, un processus *P* émet une valeur *v* au processus *Q* à travers un canal de transmission *α*.

2.2.2. Les opérateurs du π-calcul retenus pour π-SPACE

2.2.2.1. Convention d'écriture et syntaxe des opérateurs

Les lettres majuscules définissent des processus et un agent représente un ensemble de processus. Les lettres minuscules définissent des variables, des valeurs ou des canaux de transmission.

Les opérateurs suivants sont utilisés :

- la préfixation d'un processus ***P*** par une action peut dénoter la réception de la variable **y** sur le canal **x** (**x*(y)*** ¤ ***P***), l'émission de la valeur **y** sur la canal **x** (**x<y>** ¤ ***P***), ou une action inobservable – ou action interne (***τ*** ¤ ***P***) ;
- le parallélisme (***P*** **|** ***Q***) met en parallèle deux processus ;
- la restriction indique qu'un processus ***P*** ne peut pas communiquer sur un canal **x** avec son environnement – seule la communication interne sur ce canal est autorisée (***(ν*** **x)** ***P***) ;
- la somme indéterministe permet le choix entre deux processus (***P*** **+** ***Q***) ;
- l'appariement de forme "matching" – ou gardé – décrit dans l'exemple **[x = y]** ***P*** que cet agent se comporte comme ***P*** si **x** et **y** sont identiques ;
- la définition d'un processus permet notamment d'exprimer la récursivité (***A*****[x_1, …, x_n]** **=** ***P***).

La structure du π-calcul peut être définie de manière inductive, c'est-à-dire qu'elle va être construite autour du processus ***0***. Puis, nous allons pouvoir préfixer ce processus par des actions et le mettre en parallèle avec un autre processus…

La syntaxe d'un processus ***P*** peut donc être résumée par :

P ::= ***0*** | **x*(y)*** ¤ ***P*** | **x<y>** ¤ ***P*** | ***τ*** ¤ ***P*** | P_1 | P_2 | ***(ν*** **x)** ***P*** | P_1 + P_2 | **[x = y]** ***P*** | ***A*****[x_1, …, x_n]** **=** ***P***

L'opérateur de choix est prioritaire sur l'opérateur de parallélisme, qui lui-même est prioritaire sur les opérateurs de restriction, de préfixation et d'appariement de forme ; par exemple :

(ν **x)** ***P*** **|** ***τ*** ¤ ***Q*** **+** ***R*** se lit : (((***(ν*** **x)** ***P***) | (***τ*** ¤ ***Q***)) + ***R***).

2.2.2.2. Sémantique

Le processus inactif est représenté par ***0***. C'est un constituant de base de tout processus, car un processus finira par ne plus rien faire (sauf dans le cas de la récursivité).

La préfixation peut prendre plusieurs formes. L'expression ***P*** ≡ **α<v>** ¤ ***0*** décrit un processus ***P*** qui émet la valeur **v** par le canal **α**, puis s'arrête ; l'expression "puis" est traduite par l'opérateur ¤.

L'expression ***Q*** ≡ **α*(x)*** ¤ ***R*** décrit un processus ***Q*** qui reçoit la variable **x** par le canal **α**, puis continue en se comportant comme un processus ***R*** (pouvant contenir des occurrences de **x**) ; l'expression "puis" est traduite par l'opérateur ¤.

Le parallélisme est le seul moyen pour que deux processus puissent communiquer. Ceci est noté ***P*** **|** ***Q***.

A un moment donné, une valeur ***v*** va être transmise par le canal ***α*** :

α<v> ¤ ***0*** **|** **α*(x)*** ¤ ***R*** ¤ ***0*** $\rightarrow^{\tau}$ ***0*** **|** ***R*****{v/x}** ¤ ***0***

Après une action inobservable ***τ***, le système a évolué en transmettant **v** du processus ***P*** au processus ***Q***. En effet, les **x** libres qui se trouvent dans le processus ***R*** seront substitués par des **v**.

Les propriétés suivantes sont observées :

$P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $(P \mid Q) \mid R \equiv P \mid (Q \mid R) \equiv P \mid Q \mid R$.

Pour introduire la restriction, considérons le système suivant : $P \mid Q \mid R$, avec :

$P \equiv \alpha\langle v\rangle \,¤\, 0$ et $Q = R \equiv \alpha(x) \,¤\, 0$.

Alors, P peut transmettre la valeur **v** aussi bien à Q qu'à R.

Afin de restreindre la communication entre P et Q uniquement, il faut faire la restriction ν sur le canal α, de la manière suivante : $(\nu\, \alpha)\ (P \mid Q) \mid R$.

La communication sur le canal α ne pourra se faire qu'entre P et Q, c'est-à-dire que R n'a pas accès au même canal α que celui utilisé par P et Q. Dans cet exemple, les occurrences du canal α dans les processus P et Q sont dites "variables liées" et, dans R, α est une "variable libre".

Une extension de portée est possible, par exemple si P possède un canal privé **x** avec Q : $(\nu\, x)\ (P \mid Q)$, et souhaite le transmettre à R : $(\nu\, x)\ (P \mid Q \mid R)$, alors il y a une extension de portée. Par exemple :

$(\nu\, x)\ (\alpha\langle x\rangle \,¤\, P' \mid Q) \mid \alpha(x) \,¤\, R' \rightarrow^{\tau} (\nu\, x)\ (P' \mid Q \mid R'\{x/y\})$

Toutefois, la substitution de **y** par **x** – $R'\{x/y\}$ – n'est possible que si R' ne possède pas déjà un canal **x**, sinon il faut renommer le canal privé **x** pour préserver la différence avec le canal public. La portée de **x** est alors étendue aux trois processus P', Q et R'' (avec $R'' = R'\{x/y\}$: le processus résultant de la substitution de **y** par **x** dans R').

La somme indéterministe est introduite par l'opérateur de choix +. Soit l'exemple :

$(\alpha(x) \,¤\, R + \beta(y) \,¤\, S) \mid \alpha\langle v\rangle \,¤\, 0 \mid \beta\langle w\rangle \,¤\, 0$

Ce système peut évoluer :

- soit vers : $R\{v/x\} \mid 0 \mid \beta\langle w\rangle \,¤\, 0$
- soit vers : $S\{w/y\} \mid \alpha\langle v\rangle \,¤\, 0 \mid 0$

Les propriétés suivantes peuvent être observées :

$P + P \equiv P$, $P + Q \equiv Q + P$, $(P + Q) + R \equiv P + (Q + R) \equiv P + Q + R$.

Le choix peut également être spécifié en utilisant l'appariement de forme (ou "matching") :

$[x = a]\ P + [x = b]\ Q$

L'agent se comportera comme P si **x** et **a** sont identiques, comme Q si **x** et **b** sont identiques et dans les autres cas comme 0.

Il existe aussi le non appariement de forme (ou "mismatching") :

$[x \neq a]\ P$

L'agent se comportera comme P si **x** et **a** sont différents.

Ainsi, le "si, alors, sinon" peut facilement être traduit :

$[x = a]\ P + [x \neq a]\ R$ (si **x** = **a** alors P sinon R)

Un processus peut être exprimé de la façon suivante :

$A[x_1, \ldots, x_n] = P$

Un processus peut ainsi être remplacé par un nom avec ses canaux libres en paramètres.

L'avantage principal de cette notation est la possibilité de décrire des processus récursifs. Par exemple, un processus qui ne ferait qu'émettre indéfiniment la valeur **a** sur le canal **x** serait :

$Emission[x] = \alpha\langle v\rangle \,¤\, Emission[x]$

2.2.3. La particularité du π-calcul

Ce qui fait la force du π-calcul est qu'un nom de canal peut être transmis à travers un autre canal, contrairement aux autres algèbres de processus telles que CSP [Hoare 1985] ou CCS [Milner 1989].

Par exemple, si les processus ***P***, ***Q*** et ***R*** sont en parallèle : ***P*** **|** ***Q*** **|** ***R***.,

avec $P \equiv \alpha\langle c\rangle \,¤\, P'$, $Q \equiv c(y) \,¤\, Q'$, et $R \equiv \alpha(x) \,¤\, x\langle a\rangle \,¤\, R'$,

alors $P \mid Q \mid R \equiv \alpha\langle c\rangle \,¤\, P' \mid c(y) \,¤\, Q' \mid \alpha(x) \,¤\, x\langle a\rangle \,¤\, R'$

$\rightarrow^{\tau} P' \mid c(y) \,¤\, Q' \mid c\langle a\rangle \,¤\, R'$

$\rightarrow^{\tau} P' \mid Q' \mid R'$

Dans ce cadre, la topologie du système est modifiée en cours d'exécution.

C'est là la principale raison du choix du π-calcul comme base formelle pour le langage de description d'architecture logicielle π-SPACE.

2.2.4. Version du π-calcul

Le π-calcul synchrone est celui décrit jusuq'à présent, c'est-à-dire que le communication se fait sous forme de rendez-vous : soient $P \equiv \alpha\langle v\rangle \,¤\, 0$ et $Q \equiv \alpha(x) \,¤\, 0$.

Si ***P*** arrive le premier au rendez-vous, il va attendre que ***Q*** arrive à la réception de la variable **x**, pour émettre la valeur **v**. Si ***Q*** était arrivé le premier, il aurait attendu de la même façon ***P***.

Le π-calcul asynchrone se veut non bloquant pour l'émission. Dans l'exemple ci-dessus, ***P*** aurait continué son chemin après avoir émis **v**. Par contre, ***Q*** aurait réagi de la même manière que précédemment.

Le π-calcul polyadique permet de transmettre un tuple (ou vecteur) de valeurs lors d'une interaction.

Par exemple, avec $P \equiv \alpha\langle x, y, z\rangle \,¤\, 0$ et $Q \equiv \alpha(u, v, w) \,¤\, v\langle u, w\rangle \,¤\, 0$,

$P \mid Q \equiv \alpha\langle x, y, z\rangle \,¤\, 0 \mid \alpha(u, v, w) \,¤\, v\langle u, w\rangle \,¤\, 0 \quad \rightarrow^{\tau} 0 \mid y\langle x, z\rangle \,¤\, 0$

2.3. Définition de π-SPACE

π-SPACE est un langage de description d'architectures qui fournit des abstractions architecturales par rapport au π-calcul.

Le noyau de π-SPACE est basé sur la notion de composant. Un composant est la plus petite partie d'une architecture avec les caractéristiques suivantes :

- il possède un comportement ;
- il communique avec l'extérieur via des ports.

Pour permettre la description du comportement et des communications, π-SPACE fournit des opérateurs fondés sur le π-calcul.

Ce noyau fournit des mécanismes de composition permettant à ces composants d'être attachés entre eux pour former un composite. Un composite peut être lui-même attaché à d'autres composants ou composites. Le composite englobant l'ensemble de tous les composants et composites de l'application définit l'architecture. Les éléments formant un composant (le comportement et les ports), les composants et les composites, sont des éléments architecturaux.

Ce paragraphe présente, dans un premier temps, les opérateurs de π-SPACE permettant la description des processus. Dans un deuxième temps, les différents éléments architecturaux de π-SPACE seront définis : les données et les canaux de communication, les composants (formés de

ports et de comportements, avec leurs spécialisations en opérations et connecteurs), les composites ainsi que les architectures (statiques ou dynamiques). Enfin, nous décrirons les mécanismes de changement dynamique de ces éléments et le mécanisme d'évolution permettant de changer une architecture d'une application en cours d'exécution.

2.3.1. Les opérateurs de π-SPACE

La définition d'un port ou d'un comportement en π-SPACE est donnée par une expression de processus. Le processus de terminaison avec succès **$** représente un processus élémentaire en π-SPACE. Il équivaut à un événement "*terminate*" suivi du processus inactif ***0***.

Les opérateurs pour construire une expression de processus sont des opérateurs :

- de **séquence**, noté '¤',
- de **parallélisme**, noté '|',
- de **choix**, noté '+',
- de **définition de processus**, noté '=',
- de "**matching**", noté '**[x = y]** ***P***',
- de **composition**, noté '||',
- de **renommage**, noté '***{x/y}***'.

Comme pour le π-calcul, les processus contenus par les composants de π-SPACE sont définis de manière inductive, c'est-à-dire qu'ils vont être construits autour du processus **$**, puis celui-ci sera préfixé par des émissions ou des réceptions d'événements, etc. C'est un constituant de base de tout processus, puisqu'un processus finira par se terminer avec succès (sauf dans le cas de la récursivité).

Les opérateurs de π-SPACE sont de même priorité. Seule l'utilisation de parenthèses change l'ordre d'évaluation du processus.

2.3.1.1. La séquence

L'expression **α<v>** ¤ **$** décrit un comportement qui émet < > un événement contenant la donnée[1] **v** par le canal **α** puis se termine avec succès ; l'expression "puis" est traduite par l'opérateur de séquence ¤.

L'expression **α*(v)*** ¤ **$** décrit un comportement qui reçoit ***( )*** par le canal **α** un événement contenant une donnée qui se substituera au paramètre formel[2] **v**, puis se termine avec succès ; comme en π-calcul, l'expression "puis" est traduite par l'opérateur de séquence ¤.

2.3.1.2. Le parallélisme

Le parallélisme utilisé pour la description des comportements et des ports est restreint (de manière syntaxique) par rapport au parallélisme du π-calcul pour la modélisation d'un ordre indéterminé. En effet, il ne peut pas y avoir de transmission d'événement à l'intérieur d'un même composant – c'est la communication entre les composants qui nous intéresse. Le parallélisme est donc limité à la description d'émissions ou de réceptions d'événements dans un ordre indéterminé.

Par exemple, l'expression ((**α<v>** ¤ **$**) | (**β*(w)*** ¤ **$**)) ¤ **$** décrit un comportement qui émet un événement contenant la donnée **v** par le canal **α**, et reçoit par le canal **β** un événement contenant une donnée qui sera substituée au paramètre formel **w**, dans un ordre indéterminé, puis se termine avec succès.

---

[1] Une donnée en π-SPACE correspond à une valeur en π-calcul.

[2] Un paramètre formel en π-SPACE correspond à une variable en π-calcul.

2.3.1.3. Le choix

Le choix est équivalent à la somme indéterministe du π-calcul. Par contre, comme il n'a pas de priorité sur les autres opérateurs, l'utilisation de parenthèses est indispensable. L'expression suivante ((**α<v>** ¤ **$**) + (**β*(w)*** ¤ **$**)) ¤ **$** décrit un comportement qui, soit émet un événement contenant la donnée **v** par le canal **α**, soit reçoit par le canal **β** un événement contenant une donnée qui sera substituée au paramètre formel **w**, puis se termine avec succès.

2.3.1.4. La définition d'un processus

Dans la description d'un comportement, nous pouvons utiliser la définition de processus du π-calcul, permettant notamment d'exprimer la récursivité. L'expression ***P*[a, b] = α<v>** ¤ **β*(w)*** ¤ **$** décrit un processus ***P*** utilisant les canaux **α** et **β**. Ce processus émet un événement contenant la donnée **v** par le canal **α**, puis reçoit par le canal **β** un événement contenant une donnée qui sera substituée au paramètre formel **w**, puis se termine avec succès. Un appel de processus, dans π-SPACE, peut être récursif, comme dans l'exemple suivant, dans lequel un processus ne ferait qu'émettre indéfiniment la donnée **a** sur le canal **v** : ***P*[α, β] = α<v>** ¤ ***P*[α, β]**.

2.3.1.5. Le "matching"

Dans la description d'un comportement, nous pouvons utiliser de manière identique l'appariement de forme du π-calcul. Dans π-SPACE, les comparaisons selon la donnée reçue peuvent non seulement être l'égalité ("matching") ou la différence ("mismatching"), mais aussi la supériorité ou l'infériorité.

2.3.1.6. La composition

Cet opérateur permet de mettre en communication les processus contenus dans les composants. Il s'agit de l'opérateur de parallélisme du π-calcul.

Par exemple, un composant *C1* contient le processus **α<v>** ¤ **$**, et un composant *C2* contient le processus **α*(w)*** ¤ **β<w>** ¤ **$**. Si ces deux composants sont composés, alors après une action inobservable ***τ***, **v** peut être transmis par le canal **α** du composant *C1* au composant *C2* :

α<v> ¤ **$** || **α*(w)*** ¤ **β<w>** ¤ **$** $\rightarrow^{\tau}$ **$** || **β<v>** ¤ **$**

2.3.1.7. Le renommage

Pour que deux composants puissent communiquer, il est parfois nécessaire de renommer les canaux du processus qu'ils contiennent. Par exemple, un composant *C1* contient le processus **α<v>** ¤ **$**, et un composant *C2* contient le processus **δ*(w)*** ¤ **β<w>** ¤ **$**. Pour permettre aux processus contenus dans ces deux composants de communiquer, il faut renommer le canal **δ** du composant *C2* :

{α/δ}* δ*(w) ¤ **β<w>** ¤ **$** $\rightarrow^{\tau}$ **α*(w)*** ¤ **β<w>** ¤ **$**

2.3.2. Les données et les canaux

Comme nous l'avons vu dans la description des opérateurs de π-SPACE, les processus contenus dans les composants utilisent des données qui sont émises ou reçues par des canaux. Avant de nous intéresser aux constituants d'un composant nous allons définir les données et les canaux de π-SPACE.

2.3.2.1. Les données

Les données de π-SPACE sont typées. Les types de base sont :

- caractères "*Char*" (ex. : 'a', 'b', 'c'),
- chaînes de caractères "*String*" (ex. : "je suis une chaîne de caractères"),

- entiers "*Integer*" (ex. : -3, +12, 3),
- réels "*Real*" (ex. : 3e+10, -6e11, +2e-4),
- booléens "*Boolean*" (*%true*, *%false*).

A partir de ces types, π-SPACE permet de créer des types de données propres à l'utilisateur à travers quatre constructeurs de types.

Un type peut être créé à partir d'une énumération, d'un alias, d'une liste ou d'une structure de données ou de types.

L'**énumération** permet de donner le choix entre plusieurs types ou données.

Exemples de types construits par le constructeur "Enumeration", le type *Mois* est une énumération des types *MoisAlphanumérique* ou *MoisNumérique* selon qu'il soit écrit en alphanumérique ou en numérique. Le type *MoisAlphanumérique* est une énumération de données entre les douze mois de l'année. La déclaration de ces types en π-SPACE est (figure III.1) :

```
define data type MoisAlphanumérique {Enumeration[
    "Janvier", "Février", "Mars", "Avril",
    "Mai", "Juin", "Juillet", "Août", "Septembre", "Octobre", "Novembre", "Décembre"]}

define data type MoisNumérique {Enumeration[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]}

define data type Mois {Enumeration[MoisAlphanumérique, MoisNumérique]}
```

Figure III.1. : exemple de définitions de types de données par énumération

Exemples de données du type :

- *MoisAlphanumérique* : "Février", "Novembre" ;
- *MoisNumérique* : 2, 11 ;
- *Mois* : "Mai", 5.

L'**alias** permet de renommer un type de données.

Exemple de type construit par le constructeur "Alias", le type *Nom* est un alias du type *String* (cf. figure III.2).

```
define data type Nom {Alias[String]}
```

Figure III.2. : exemple de définition d'un type de données par alias

Exemples de données du type *Nom* : "Mozart", "Hugo".

La **liste** permet de définir une liste de données d'un même type de données.

Exemple de type construit par le constructeur "Liste", le type *Prénoms* est une liste du type *String* (cf. figure III.3).

define data type Prénoms **{**Liste**[**String**]}**

Figure III.3. : exemple de définition d'un type de données par liste

Exemples de données du type *Prénoms* : List("Wolfgang", "Amadeus"), List("Victor").

La **structure** permet de définir un type de données composé lui-même de plusieurs types de données.

Exemple de type construit par le constructeur "Structure", le type *Identification* est une structure des types *Nom* et *Prénoms* (cf. figure III.4).

define data type Identification **{**Structure**[**Nom, Prénoms**]}**

Figure III.4. : exemple de définition d'un type de données par structure

Exemples de données du type *Identification* : Structure("Mozart", List("Wolfgang", "Amadeus")), Structure("Hugo", List("Victor")).

2.3.2.2. Les canaux

Les canaux qui permettent à deux processus de communiquer sont typés, c'est-à-dire que chaque canal est prévu pour transmettre un type d'événement, ce type pouvant être complexe (énumération, structure, alias, liste).

Par exemple, la spécification ci-dessous (figure III.5) spécifie en π-SPACE deux types de canaux :

define channel type Event **{ [] }**

define channel type ChannelResult **{ [**Result**] }**

avec :

define data type NomFichier **{**Alias**[**String**]}**

define data type Result **{**Structure**[**NomFichier, Boolean, String**]}**

Figure III.5. : exemple de définition de types de canaux

Le type de canal *Event* permet de créer des canaux transportant un événement ***[]***. Cet événement ne contient pas de donnée ; il servira donc simplement à synchroniser deux processus.

Exemple d'utilisation d'un canal de type *Event* : canalEvent<>.

Le type de canal *ChannelResult* permet de créer des canaux transportant un événement contenant des données de type *Result*.

Exemple d'utilisation d'un canal de type *ChannelResult* :

canalResult<Structure("Fichier.doc", *%true*, "corrigé le 26/06/2004")>.

2.3.3. Les composants

Comme il a été défini plus haut, un composant est la plus petite partie d'une architecture avec les caractéristiques suivantes :

- il possède un **comportement** spécifiant un processus séquentiel ;
- il communique avec l'extérieur via des **ports**.

Pour pouvoir utiliser un composant dans une architecture, il n'est pas nécessaire de connaître comment il fonctionne mais ce qu'il produit. Par exemple, si un composant est utilisé pour additionner deux chiffres et en soustraire un troisième, alors il n'est pas nécessaire de savoir s'il commence par effectuer l'addition ou la soustraction. Il est intéressant de connaître le fait qu'il faut lui fournir trois chiffres et qu'il nous renverra le résultat. Son fonctionnement interne est décrit par son comportement, alors que ce qu'il produit est décrit par ses ports. Un composant peut alors être vu comme une boîte noire dont les ports représentent les interactions qu'il peut avoir avec les autres composants. Par le port, le composant reçoit et émet les événements nécessaires à son comportement.

Un composant étant une composition de ports et d'un comportement, nous allons, dans un premier temps, spécifier ces deux éléments architecturaux avant de spécifier le composant.

2.3.3.1. Les ports

Un port représente les interactions possibles qu'un composant peut avoir avec d'autres composants, c'est-à-dire les émissions et les réceptions d'événements. Ces ports possèdent des canaux, où chaque canal est attaché à un port d'un autre composant, permettant ainsi la transmission d'événements.

Un port est toujours une instance d'un type de port.

Un type de port est défini par son nom et les canaux typés qu'il utilise.

Il peut étendre un autre type de port, c'est-à-dire qu'il peut reprendre la spécification d'un type de port déjà existant et l'étendre avec ses propres spécifications. Pour cela le mot-clé ***extends*** est utilisé.

Dans sa spécification, un port décrit l'enchaînement des émissions et des réceptions d'événements par chacun de ses canaux.

L'enchaînement est spécifié par les opérateurs π-SPACE :

- de séquence, noté '¤',
- de parallélisme, noté '|',
- de choix, noté '+'.

Prenons l'exemple, spécifié en π-SPACE, d'un port défini ci-dessous (figure III.6) :

define port type Request **[**request:**[**Service**]**, reply:**[**Result**]] {**

service:Service, result:Result,

Request**[**request, reply**]** = ((request<service> ¤ reply***(***result***)*** ¤ Request**[**request, reply**]**) + **$**) **}**

Figure III.6. : exemple de définition d'un type de port

Cet exemple définit un type de port *Request* ayant comme paramètres d'entrée :

- *reply*, un canal transportant des événements de type *Result*,

- *request*, un canal transportant des événements de type *Service*.

Ce port utilise deux paramètres formels, dont la portée est celle du port :

- *service*, qui est de type *Service*,
- *result*, qui est de type *Result*.

Les paramètres formels représentent le type de la donnée utilisée.

Ce port permettra au composant auquel il appartiendra, d'émettre une donnée de type *Service* (représentée par le paramètre formel *service*) par le canal *request* (request<service>), puis (¤) de recevoir un paramètre de type *Result* par le canal *reply* (reply***(***result***)***). Ensuite le composant peut soit :

- utiliser la récursivité (Request **[**request, reply**]**) et recommencer l'émission du paramètre de type *Service* par le canal *request*, …,
- terminer avec succès **$**.

2.3.3.2. Les comportements

Un comportement représente le calcul interne d'un composant. Ces comportements peuvent émettre ou recevoir des événements grâce aux ports qu'ils reçoivent en paramètre d'entrée.

Un comportement est toujours une instance d'un type de comportement.

Un type de comportement est défini par son nom et les ports utilisés dans sa spécification. Comme pour les ports, un type de comportement peut étendre un autre type de comportement avec le mot-clé ***extends***.

Dans sa spécification, un comportement utilise les canaux des ports pour recevoir ou émettre des événements typés.

Les enchaînements utilisés par les comportements sont spécifiés par les opérateurs π-SPACE :

- de séquence, noté '¤',
- de parallélisme, noté '|',
- de choix, noté '+'.

Un comportement peut utiliser :

- le "matching", permettant de prendre différents chemins selon la donnée reçue,
- la définition de processus, permettant de simplifier l'écriture du comportement et de procéder à des appels récursifs.

Prenons l'exemple, spécifié en π-SPACE, d'un comportement ci-dessous (figure III.7) :

define behaviour component type ExempleComportement **[**pRequest: Request **[**request:**[**NomFichier**]**, reply:**[**Result**]]] {**

nomFichier:NomFichier, acceptation:Boolean, commentaire:String,

ExempleComportement**[**pRequest**]** = (pRequest **@** request<"FichierExemple"> ¤ pRequest **@** reply***(***Structure(nomFichier, acceptation, commentaire)***)*** ¤ (**[**acceptation = *%true***]** **$** + **[**acceptation = *%false***]** ExempleComportement**[**pRequest**]**)) **}**

Figure III.7. : exemple de définition d'un type de comportement de composant

Cet exemple définit un type de comportement de composant *ExempleComportement* ayant le port *pRequest* de type *Request* comme paramètre d'entrée. Les canaux qui lui sont associés sont :

- le canal *request* transportant des événements de type *NomFichier*,
- le canal *reply* transportant des événements de type *Result*.

Ce comportement utilise trois paramètres formels :

- *nomFichier* qui est de type *NomFichier*,
- *acceptation* qui est de type *Boolean*,
- *commentaire* qui est de type *String*.

La portée de ces paramètres formels est celle du comportement. Ils permettent de donner le type d'une donnée à recevoir.

Ce type de comportement ne comporte qu'un paramètre d'entrée. De ce fait, un seul port de nom *pRequest* est utilisé. Ce port possède deux canaux *request* et *reply*. L'écriture spécifiant l'utilisation du canal *request* du port *pRequest* est : *pRequest* **@** *request* ; et l'utilisation du canal *reply* du port *pRequest* est : *pRequest* **@** *reply*.

Ce type de comportement spécifie un processus qui émet un événement contenant la donnée "*FichierExemple*" par le canal *request* du port *pRequest*.

Ensuite (opérateur séquentiel '¤'), il reçoit sur le canal *reply* de son port *pRequest* l'événement contenant une structure de trois données qui sont substituées aux trois paramètres formels *nomFichier*, *acceptation* et *commentaire*.

Si la valeur de la donnée représentée par le paramètre *acceptation* est :

- "vrai", alors le comportement se termine avec succès ;
- "faux", alors le comportement se poursuit avec la récursivité (ExempleComportement**[**pRequest**]**) : émission de l'événement contenant la donnée "*FichierExemple*" par le canal *request* du port *pRequest*, …

2.3.3.3. Les composants

Maintenant que nous avons spécifié les types de ports et les types de comportements, nous pouvons spécifier les types de composants. En effet, un composant est une composition de une ou plusieurs instances de types de ports et d'une instance de type de comportement.

Un type de comportement est défini par son nom et les ports utilisés dans son comportement. Dans sa spécification, le composant peut utiliser la composition, permettant de composer les ports et le comportement. Comme pour les autres types, un type de composant peut étendre un autre type de composant avec le mot-clé ***extends***.

Prenons l'exemple, spécifié en π-SPACE, d'un type de composant ci-dessous (figure III.8) :

```
define component type ExempleComposant
   [pRequest: Request [request:[NomFichier], reply:[Result] ]] {
port pRequest: Request ||
behaviour ExempleComportement[pRequest] }
```

Figure III.8. : exemple de définition d'un type de composant

Cet exemple définit un type de composant *ExempleComposant* ayant le port *pRequest* de type *Request* comme port de communication.

Ce composant est vu par les autres composants par son port pouvant transmettre des événements par :

- le canal *request* qui transporte des événements de type *NomFichier*,
- le canal *reply* qui transporte des événements de type *Result*.

Dans sa spécification, le composant instancie le port *pRequest* de type *Request*, et le type *ExempleComportement* en lui passant en paramètre le port par lequel il peut communiquer.

2.3.3.4. Spécialisations de composants : les opérations et les connecteurs

A partir des composants appartenant au noyau de π-SPACE, nous pouvons définir d'autres constructions architecturales permettant de prendre en considération des domaines d'applications spécifiques. Ces éléments n'appartiennent pas au noyau de π-SPACE, mais sont construits à partir de celui-ci.

Par rapport au vocabulaire et aux besoins des architectures, ont été construites à partir du noyau deux spécialisations de composant :

- les opérations,
- les connecteurs.

Une **opération** π-SPACE est définie comme étant un composant dont seuls les paramètres d'entrée et de sortie sont connus. Prenons comme exemple l'utilisation de COTS ("Component Of The Shelf") : un composant pourrait contenir une opération capable d'exécuter l'ouverture d'une application de traitement de texte. L'utilisateur pourrait écrire à l'intérieur et enregistrer son document sous un nom de fichier pouvant être le paramètre de sortie de l'opération. Ce nom de fichier pourra être transmis à un autre composant qui, par une opération, pourra demander son ouverture afin de le lire.

π-SPACE permet de créer des types d'opérations qui seront instanciés dans le comportement d'un composant. Un type d'opération est défini par son nom et une liste de paramètres. Ces paramètres peuvent être :

- des paramètres d'entrée (événement qui sera reçu par le port de type *In*),
- des paramètres de sortie (événement qui sera émis par le port de type *Out*),
- des paramètres d'entrée/sortie (événement qui sera reçu/émis par le port de type *InOut*).

Prenons l'exemple, spécifié en π-SPACE, d'un type d'opération ci-dessous (figure III.9) :

```
define operation type InternalCompute [entree: In [service:Service], sortie: Out
    [result:Result]] { … }
```

Figure III.9. : exemple de définition d'un type d'opération

Cet exemple définit un type d'opération *InternalCompute* ayant comme paramètres d'entrée :

- *entree* de type ***In***, correspondant à un port d'entrée de l'opération et transportant un événement contenant un paramètre *service* de type *Service*,
- *sortie* de type ***Out***, correspondant à un port de sortie de l'opération et transportant un événement contenant un paramètre *result* de type *Result*.

Ces opérations sont utilisées dans un composant. Ces composants sont définis de la même manière que les composants du noyau, sauf que le comportement peut être étendu en intégrant des opérations.

Un comportement d'un composant peut, soit instancier un type d'opération, soit créer une opération directement dans sa spécification. Prenons comme exemple un comportement recevant un paramètre par son port. Ce paramètre est transmis à une opération *InternalCompute* qui peut réaliser une transformation et renvoyer un résultat. Ce résultat est émis par le port du composant qui, soit recommence (récursivité du comportement), soit s'arrête avec succès. Cet exemple est spécifié en π-SPACE ci-dessous (figure III.10) :

define behaviour component type ServerWork **[**pReply: Reply **[**reply:**[**Service**]**, request:**[**Result**]]] {**

service:Service, result:Result,

InternalCompute **[**entree: **In [**Service**]**, sortie: **Out [**Result**]] {** … **}**,

ServerWork**[**pReply**]** = ((pReply **@** reply***(***service***)*** ¤ InternalCompute **[**service, result**]** ¤ pReply **@** request<result> ¤ ServerWork**[**pReply**]**) + **$**) **}**

Figure III.10. : exemple de définition d'un type de comportement avec opération

Dans ce type de comportement de composant est spécifiée une opération *InternalCompute*. Cette opération possède :

- un port *entree* de type ***In***, transportant un paramètre d'entrée de type *Service*,
- un port *sortie* de type ***Out***, avec un paramètre de sortie de type *Result*.

Dans ce même comportement, le type d'opération *InternalCompute* spécifié dans le paragraphe précédent aurait pu être utilisé. Ce type aurait été instancié de la manière suivante :

*internalCompute : InternalCompute **[**entree: **In [**service**]**, sortie: **Out [**result**]] { … }*,

Ainsi, dans un même comportement, un type identique d'opérations peut être instancié plusieurs fois. De même, une opération apparaissant dans plusieurs composants ne peut être spécifiée par un type qu'une seule fois.

Dans un autre exemple, il aurait pu être nécessaire de transmettre les paramètres de l'opération de façon séparée. Pour spécifier la transmission du paramètre d'entrée seul, nous aurions noté : *internalCompute **@** entree**[**service**]*** ; et pour la transmission du paramètre de sortie seul : *internalCompute **@** sortie**[**result**]***.

Le **connecteur** est spécifié comme une autre spécialisation de composant. Il n'a pour fonction que d'aiguiller des événements d'un composant à l'autre. Cet élément architectural ne peut donc pas contenir d'opération.

Nous allons, dans cette partie, spécifier deux sortes de connecteurs : la première permettant d'aiguiller des événements entre des composants et la deuxième permettant d'aiguiller des événements entre des composants internes à un composite (cf. paragraphe 2.3.4 juste après) et des composants externes.

Un **connecteur composant-composant** a pour fonction d'aiguiller les événements entre les composants. Il représente leur protocole de communication.

La fonction d'un **connecteur composite-composant** est d'aiguiller l'ensemble des événements entre les composants internes d'un composite et les composants externes. Son instanciation ne peut être faite qu'à l'intérieur d'un composite. En effet, cet élément portera les ports non attachés d'un composite. Tous les événements sortant d'un composite passeront forcément par ses connecteurs.

2.3.4. Les composites

Un composite est une composition de composants. Il existe deux utilisations possibles d'un composite : utilisation en tant que composant, ou utilisation en tant que modèle d'attachement. Dans un premier temps, nous spécifierons et expliquerons la syntaxe de π-SPACE se rapportant à la spécification d'un type de composite. Puis, dans un deuxième temps, nous nous intéresserons à ces deux utilisations possibles par des exemples spécifiés en π-SPACE.

Un type de composite est défini par son nom et :

- ses ports, si le composite est utilisé comme composant,
- les composants à attacher, si le composite est utilisé comme modèle d'attachement.

Comme pour les autres types, un type de composite peut étendre un autre type de composite avec le mot-clé ***extends***.

Une partie de déclaration des opérations d'évolution, dans la définition du type de composite, permet, quand le composite est utilisé comme modèle d'attachement, de déclarer les conditions d'évolution. Celles-ci seront décrites dans le paragraphe 2.3.7 ("l'évolution d'une architecture vers une autre").

Une partie de déclaration des composants, dans la définition du type de composite, permet d'instancier les composants de la composition. Pour cela, nous procédons comme pour l'instanciation des ports et des comportements dans le composant, c'est-à-dire que nous noterons le nom de l'instance, l'opérateur d'instanciation ':', suivis du type à instancier.

Pour permettre la communication entre deux composants, il est parfois nécessaire de renommer les ports et les canaux des composants instanciés. Pour cela, nous utilisons l'opérateur de renommage (cf. 2.3.1.7).

Par exemple, prenons un composite qui instancie un composant *EC* de type *ExempleComposant* (dans la figure III.11) :

EC: ExempleComposant **[**pRequest/pExemple **{**request/verifie, reply/reponse**}]**

Figure III.11. : exemple d'une instanciation de composant

Le type de composant *ExempleComposant* (cf. 2.3.3.3) est instancié en renommant le port *pRequest* par *pExemple*, et ses canaux *request* et *reply* par *verifie* et *reponse* respectivement.

Une partie de déclaration des attachements, dans la définition du type de composite, permet d'attacher les canaux des ports des composants de la composition. L'opérateur ***attach to*** permet de procéder à ces attachements selon les trois cas suivants :

1. si les canaux et les ports possèdent le même nom, alors il suffit de donner les noms des deux composants à attacher ;
2. si les canaux possèdent le même nom mais pas les mêmes ports, alors il faudra donner le nom des deux composants et des deux ports à attacher ;
3. enfin, si ni les canaux, ni les ports ne possèdent les mêmes noms, alors il faudra donner les noms des deux composants, des deux ports et des deux canaux à attacher.

Pour simplifier les attachements, il est préférable de faire les renommages adéquats pour n'avoir plus qu'à attacher les composants.

Une partie de déclaration des opérations dynamiques, dans la définition du type de composite, permet d'utiliser les opérations dynamiques (cf. paragraphe 2.3.6).

Maintenant que nous avons précisé la syntaxe de π-SPACE se rapportant à la spécification d'un type de composite, nous pouvons aborder les deux utilisations possibles d'un composite, c'est-à-dire soit en tant que composant, soit en tant que modèle d'attachement.

2.3.4.1. Les composites en tant que composants

Un composite en tant que composant permet la décomposition d'une architecture de π-SPACE. En effet, à un haut niveau d'abstraction, une fonction peut être spécifiée en tant que composite en spécifiant uniquement ses ports (ses entrées/sorties d'événements). Puis il peut être raffiné en spécifiant à l'aide de composants (et/ou de composites qui seront raffinés ensuite) son comportement.

Les ports des composants qui ne sont pas attachés avec d'autres composants sont les ports du composite.

L'exemple ci-dessous (figure III.12) propose un type de composite, décomposé en trois composants. Le composite instancie ces trois types de composant et les compose en utilisant les renommages appropriés et en attachant leurs ports.

define composite type ExempleComposite1 **[**pComposite: PortEmission **[**canalEmission:**[]]] {**

C1 : Component1 **||**

C2 : Component2 **||**

C3 : Composite3 **[**pC1/pC3 **{**canalEmission / canalReception **}]**

where

attach C1 **@** pC2 **@** canalReception

to C2 **@** pC1 **@** canalEmission,

attach C1 **to** C3 **}**

avec :

define component type Component1 **[**pComposite: PortEmission **[**canalEmission:**[]]**, pC2: portReception **[**canalReception:**[]]**, pC3: portReception **[**canalReception:**[]]] {** … **}**

define component type Component2 **[**pC1: PortEmission **[**canalEmission:**[]]] {** … **}**

define composite type Composite3 **[**pC1: PortEmission **[**canalEmission:**[]]] {** … **}**

Figure III.12. : exemple de définition d'un type de composite comme composant

Le composite *Composite3* possède un port *pC1* qui possède un canal transportant un événement. Ce composite est instancié dans le composite *ExempleComposite1* ; son port *pC1* est renommé par *pC3* et son canal *canalEmission* par *canalReception*. Ces renommages permettent de l'attacher au composant *C1* en notant : ***attach*** *C1* ***to*** *C3*.

N'ayant pas fait de renommage, le canal *canalReception* du port *pC2* du composant *C1* est attaché au canal *canalEmission* du port *pC1* du composant *C1* en notant : ***attach*** *C1* **@** *pC2* **@** *canalReception* ***to*** *C2* **@** *pC1* **@** *canalEmission*.

Le port *pComposite* du composant *C1* n'étant pas attaché dans le composite, il devient le port du composite *ExempleComposite1*.

Le composite ainsi défini peut être vu comme une boîte noire dont seul son port *pComposite* est connu. Il est alors assimilable à un composant et peut donc être instancié de la même façon dans un autre composite.

2.3.4.2. Les composites en tant que modèle d'attachement

Un composite en tant que modèle d'attachement permet de spécifier des attachements entre des types de composants. Ainsi, à partir d'instances de composants et d'un modèle d'attachement, nous obtenons une instance de composite.

Les ports qui n'ont pas été attachés par le modèle d'attachement devront être attachés dans la composition où le composite est instancié.

Prenons l'exemple, spécifié en π-SPACE, d'un type de composite (figure III.13).

```
define composite type ExempleComposite2 [C1 : Component1, C2 : Component2] {
C1 : Component1 ||
C2 : Component2
where
attach C1 @ pC2 @ canalReception to C2 @ pC1 @ canalEmission }
```

Figure III.13. : exemple de définition d'un type de composite comme modèle d'attachement

Ce composite prend en paramètres d'entrée deux composants de type *Component1* et *Component2*. Il compose ses deux composants en procédant à l'attachement spécifié dans la partie ***where***.

Le composite *ExempleComposite1* spécifié au paragraphe précédent (2.3.4.1), aurait pu être spécifié en π-SPACE comme ci-dessous (figure III.14) :

```
define composite type ExempleComposite1 [pComposite: PortEmission [canalEmission:[]] ] {
C1 : Component1 ||
C2 : Component2 ||
C3 : Composite3 [pC1/pC3 {canalEmission / canalReception }]
where
attach C1 to C3,
ExempleComposite2 [C1, C2] }
```

Figure III.14. : exemple de définition d'un type de composite à partir d'un modèle d'attachement

Le composite *ExempleComposite1* instancie les composants et le composite. Puis, il procède à l'attachement des canaux des ports des composants *C1* et *C3* portant le même nom. Enfin, il donne les instances *C1* et *C2* pour les attacher comme spécifié par le composite *ExempleComposite2*.

2.3.5. Les architectures

Une architecture est une composition de composants et/ou de composites. Elle peut être assimilée à un composite qui ne possède pas de port.

Une architecture est définie par son nom et, mise à part qu'elle ne possède pas de paramètre d'entrée, sa spécification est la même que celle d'un type de composite. Une fois l'architecture spécifiée, l'ensemble des éléments architecturaux qui la compose est instancié.

Une fois l'architecture spécifiée, elle pourra donner naissance, après raffinement, par génération automatique, à l'application dans le langage cible choisi.

2.3.6. Les architectures dynamiques

Dans les paragraphes précédents, nous avons vu :

- les différents éléments architecturaux de π-SPACE : port, comportement, composant, composite et architecture ;
- le cadre compositionnel de π-SPACE : la composition de ports et d'un comportement pour le composant, et la composition de composants et/ou de composites pour le composite.

Dans ce paragraphe, nous allons nous intéresser aux aspects dynamiques que nous avons laissés lors de la spécification des types de composites. En effet, π-SPACE permet de spécifier des éléments architecturaux dynamiques, c'est-à-dire pouvant se créer et s'attacher selon des règles prédéfinies dans l'architecture. Dans cette partie, nous reprendrons les éléments architecturaux définis précédemment en nous focalisant sur leurs aspects dynamiques, dans le but de décrire les connexions dynamiques et les composants dynamiques. Ces aspects seront traités au travers d'exemples.

2.3.6.1. Les connexions dynamiques

π-SPACE permet de changer dynamiquement les attachements entre les composants d'une architecture. Un composant peut être attaché à un autre composant, puis en être détaché pour être rattaché à un autre composant.

Un composant ne connaît pas la nature de son attachement (s'il est dynamique ou non) ; pour lui, cela reste totalement transparent. Il ne s'aperçoit pas des changements de ses attachements.

N'importe quel composant d'un composite peut provoquer le changement des attachements. Pour ce faire, π-SPACE met à disposition un type de port prédéfini relié directement au composite dans lequel il a été instancié. Ce type de port est : *AttachmentEvolutionPort*. Ce type possède deux canaux : *begin* et *end*. Ces canaux transportent des événements.

Le composant souhaitant procéder à un changement dynamique des attachements compose un port de type *AttachmentEvolutionPort* et un comportement, comme dans l'exemple spécifié ci-dessous (figure III.15) :

```
define component type ExempleComposant
    [pC: PortReception [canalReception:[] ], changementAttachement: AttachmentEvolutionPort
    ] {
port pC: PortReception ||
port changementAttachement: AttachmentEvolutionPort ||
behaviour ExempleComportement[pC, changementAttachement] }
```

Figure III.15. : exemple de définition d'un type de composant avec connexion dynamique

Dans le comportement *ExempleComportement*, le composant pourra transmettre un événement au composite où il a été instancié en utilisant l'opérateur ***evolvable*** comme dans l'exemple spécifié ci-dessous (figure III.16) :

```
define behaviour component type ExempleComportement [pC: PortReception
    [canalReception:[] ], changementAttachement: AttachmentEvolutionPort ] {
ExempleComportement[pC, changementAttachement] = (pC @ canalReception( ) ¤
    evolvable(changementAttachement) ¤ Boucle[pC] ),
Boucle[pC] = ( (pC @ canalReception( ) ¤ Boucle[pC] ) + $) }
```

Figure III.16. : exemple de définition d'un type de comportement de composant pour connexion dynamique

Lorsque le comportement *ExempleComportement* demande un changement d'attachement ***evolvable****(changementAttachement)*, il émet un événement sur le canal *begin* de son port *changementAttachement*. Le comportement de *C1* se bloque jusqu'à la réception d'un événement sur le canal *end* de son port *changementAttachement*.

Dans le composite, seront spécifiés les changements d'attachements à effectuer. Ces changements peuvent être effectués sur n'importe quel attachement de la composition. Le composite peut être celui spécifié en π-SPACE par l'exemple ci-dessous (figure III.17) :

```
compose ExempleComposition {
C1 : ExempleComposant ||
C2 : Component2 ||
C3 : Composite3
where
attach C1 @ pC @ canalReception to C2 @ pC1 @ canalEmission
whenever
evolvable(C1 @ changementAttachement) ⇒
    unattach C1 @ pC @ canalReception to C2 @ pC1 @ canalEmission
unattach C1 @ pC @ canalReception to C2 @ pC1 @ canalEmission ⇒
    attach C1 @ pC @ canalReception to C3 @ pC1 @ canalEmission ; }
```

Figure III.17. : exemple de composition d'architecture

Lorsque *C1* demande un changement d'attachement, la composition recevra un événement sur le canal *begin* du port *changementAttachement* du composant *C1* et déclenchera ('⇒') le détachement du canal *canalReception* du port *pC* du composant *C1* et du canal *canalEmission* du port *pC1* du composant *C2*.

Le détachement des canaux effectué déclenchera ('⇒') l'attachement du canal *canalReception* du port *pC* du composant *C1* et du canal *canalEmission* du port *pC1* du composant *C3*. Le point virgule signifie la fin du changement dynamique et un événement sera émis sur le canal end du port *changementAttachement* du composant *C1*.

2.3.6.2. Les composants dynamiques

π-SPACE permet de créer dynamiquement de nouvelles instances de ports, de composants ou de composites.

Ce paragraphe introduit les composants et les ports dynamiques à travers un exemple : un composant *C1* va demander au composite, dans lequel il a été instancié, l'instanciation d'un composant dynamique *C2π*. Lorsque sa demande est prise en compte par le composite, celui-ci va créer une nouvelle instance de port dans le composant *C1* et une nouvelle instance de composant *C2π*. Puis, il attachera le nouveau port du composant *C1* au port du nouveau composant dynamique *C2π*. Les instances du type de port et du type de composant vont être créées et attachées à la volée. Ainsi, le nombre d'instances de ports dynamiques et de composants dynamiques va changer en cours d'exécution du système.

Nous allons spécifier en π-SPACE, dans un premier temps, le composant *C1* ayant un port dynamique, ainsi que son comportement, et dans un deuxième temps le composite gérant le composant dynamique *C2π*.

La spécification en π-SPACE ci-dessous (figure III.18) décrit le type de composant *C1* :

```
define component type ExempleComposant
   [pCπ: PortReception [canalReception:[] ], creationComposant: ComponentEvolutionPort ] {
port pCπ: PortReception ||
port creationComposant: ComponentEvolutionPort ||
behaviour ExempleComportement[pCπ, creationComposant] }
```

Figure III.18. : exemple de définition d'un type de composant avec port dynamique

Cet exemple contient deux éléments nouveaux :

- un port dynamique *pCπ*,
- un port de type *ComponentEvolutionPort*.

L'opérateur permettant de rendre un élément architectural dynamique est *π*. Cet opérateur est accolé au nom du port qui doit être dynamique. Dans l'exemple ci-dessus, le port *pCπ* est un port dynamique, c'est-à-dire que ce port peut être créé dynamiquement et que son nombre d'occurrences n'est pas connu préalablement.

Dans l'exemple que nous traitons dans cette partie, c'est le composant auquel vont s'attacher les composants dynamiques qui vont demander leur instanciation. Mais, comme pour les attachements dynamiques, n'importe quel composant d'un composite aurait pu demander la création dynamique d'un composant. Pour ce faire, π-SPACE met à notre disposition un type de port prédéfini relié directement au composite dans lequel il a été instancié. Ce type de port est *ComponentEvolutionPort*. Ce type possède deux canaux : *begin* et *end*. Ces canaux transportent des événements.

Le comportement du composant est spécifié en π-SPACE ci-dessous (figure III.19) :

```
define behaviour component type ExempleComportement [pCπ: PortReception
    [canalReception:[Boolean] ], creationComposant: ComponentEvolutionPort ] {

reponse: Boolean,

ExempleComportement[pCπ, creationComposant] = (evolvable(creationComposant) ¤
    pCπany=i @ canalReception(reponse) ¤ ( [reponse = %true] Boucle[pCπi] + [reponse =
    %false] ExempleComportement[pCπ, creationComposant] ) ),

Boucle[pCπi] = ( (pCπi @ canalReception(reponse) ¤ Boucle[pCπi] ) + $) }
```

Figure III.19. : exemple de définition d'un type de comportement de composant avec port dynamique

Lorsque le comportement *ExempleComportement* demande une création d'une nouvelle instance de composant dynamique ***evolvable****(creationComposant)*, il émet un événement sur le canal *begin* de son port *creationComposant*. Le comportement de *C1* se bloque jusqu'à la réception d'un événement sur le canal *end* de son port *creationComposant*. Ensuite, il va attendre un événement sur un de ses ports dynamiques, ce port sera gardé en mémoire dans la variable *i*. Si la réponse est égale au booléen *%true*, il attendra alors un événement sur le port mémorisé.

Ne connaissant pas combien de ports dynamiques un composant a, à un instant donné, π-SPACE met à la disposition du comportement, des opérateurs permettant de gérer les transmissions par des ports dynamiques. Ces opérateurs utilisés dans les comportements sont les opérateurs *any* et *all*. L'opérateur *any*, utilisé dans l'exemple précédent, permet de transmettre sur un nombre de ports dynamiques prédéfinis. Le nombre sera juxtaposé, entre parenthèses, à l'opérateur. Par exemple, *any(3)* permet d'effectuer trois transmissions. La transmission se fera avec les trois premiers ports prêts à communiquer. Par défaut, si aucun nombre n'est précisé, la transmission se fera avec un des ports dynamiques. L'opérateur *all* permet de transmettre sur l'ensemble des ports dynamiques.

Avec ces deux opérateurs, un événement peut être émis par tous les ports dynamiques connectés. Par contre, à part pour *any(1)*, la réception passera obligatoirement par une liste d'événements, chaque événement provenant d'un des ports dynamiques.

Un opérateur "mémoire", utilisé dans l'exemple précédent, et permettant de repérer quels ports ont pris part à la transmission, peut être associé à ces opérateurs. Par exemple, si un événement est reçu par un port dynamique et qu'un événement doit lui être renvoyé, il faut pouvoir le garder en mémoire. Pour cela, l'opérateur de mémoire '=', suivi d'une variable, est juxtaposé aux ports dynamiques impliqués dans la transmission. Pour effectuer une transmission avec ces ports dynamiques en mémoire, la variable sera juxtaposée à leurs noms. Nous pouvons aussi vouloir, non pas renvoyer aux ports dynamiques en mémoire, mais émettre à un nombre identique de ports que ceux mémorisés. Pour cela, l'opérateur *card* suivi, entre parenthèses, du nom de la variable, est juxtaposé au nom du port dynamique.

Intéressons-nous maintenant au composite contenant les composants *C1* et *C2π* (cf. figure III.20) :

```
compose ExempleComposition {
C1 : ExempleComposant ||
C2π : Component2
whenever
   evolvable(C1 @ creationComposant) ⇒ new C2π,
   new C2π ⇒ new C1 @ pCπ,
   new C1 @ pCπ ⇒ attach C1 @ pCπ @ canalReception to C2 @ pC1 @ canalEmission ; }
```

Figure III.20. : exemple de composition d'architecture avec port dynamique

Comme pour les ports, pour spécifier qu'un composant est dynamique, un π est accolé à son nom. Dans l'exemple ci-dessus, le composant *C2π* est un composant dynamique, c'est-à-dire que ce composant peut être créé dynamiquement et que son nombre d'occurrences n'est pas connu préalablement.

Les règles de création des instances dynamiques sont décrites dans la partie ***whenever*** : lorsque *C1* demande une création de composants, la composition recevra un événement sur le canal *begin* du port *creationComposant* du composant *C1* et déclenchera ('⇒') la création d'une nouvelle instance du composant dynamique *C2π*. La création de la nouvelle instance effectuée déclenchera ('⇒') la création d'une nouvelle instance de port dynamique *pCπ* du composant *C1*. La création de la nouvelle instance effectuée déclenchera ('⇒') l'attachement du canal *canalReception* de la nouvelle instance du port *pCπ* du composant *C1* et du canal *canalEmission* du port *pC1* du nouveau composant dynamique *C2π*. Le point virgule signifie la fin des créations dynamiques et un événement sera émis sur le canal *end* du port *creationComposant* du composant *C1*.

2.3.7. L'évolution d'une architecture vers une autre

Dans les paragraphes précédents, nous avons vu :

- les différents éléments architecturaux de π-SPACE : port, comportement, composant, composite, architecture ;
- le cadre compositionnel de π-SPACE : la composition des ports et d'un comportement pour le composant et la composition de composants et/ou de composites pour le composite ;
- la spécification d'architectures dynamiques, c'est-à-dire le changement des attachements et la création de nouveaux éléments architecturaux selon des règles prédéfinies dans l'architecture.

Or, π-SPACE propose aussi un cadre évolutif, c'est-à-dire qu'une architecture peut être décomposée, puis être recomposée avec de nouveaux éléments architecturaux, tout cela en cours d'exécution de l'application. Dans cette partie, nous reprendrons les éléments architecturaux définis précédemment en nous focalisant sur leurs aspects évolutifs dans le but de décrire l'évolution d'une architecture vers une autre.

Cette partie présentera, dans un premier temps, une évolution d'architecture concernant le changement d'un élément architectural. Dans un deuxième temps, nous verrons comment faire évoluer l'intérieur d'un composite dans le cadre d'un changement d'architecture. Ces aspects seront traités au travers d'exemples.

2.3.7.1. Changement d'éléments architecturaux

Grâce à la mobilité du π-calcul, π-SPACE permet de faire évoluer une architecture.

Soit une architecture *A1* possédant deux composants *C1* et *C2*. Cette architecture a été raffinée suffisamment pour permettre la génération automatique de l'application correspondante. L'application est donc déployée et donne entière satisfaction au client. Seulement, au bout d'un certain temps, l'application n'est plus suffisamment performante et le client souhaite la faire évoluer, sans l'arrêter, pour ne pas stopper sa production. L'évolution souhaitée passe par le remplacement du composant *C1* par un nouveau composant plus performant *C3*. Pour effectuer cette évolution, π-SPACE propose divers opérateurs d'évolution comme la **décomposition** et la **recomposition**.

L'application issue de l'architecture *A1* est en train de s'exécuter. Nous décidons de changer le composant *C1*. Ce composant ne doit pas être enlevé à n'importe quel moment. Par exemple, si le composant *C2* émet une requête au composant *C1* et attend une réponse, et qu'au même moment *C1* est changé, *C2* sera bloqué en attente d'une réponse qui ne viendra pas. Pour cela, π-SPACE propose d'utiliser des points d'arrêt qui permettent d'arrêter le comportement d'un composant à un point donné, appelé **point d'évolution** (la spécification de ce point d'évolution sera donnée plus loin). Une fois le composant *C1* arrêté, l'architecture *A1* sera décomposée. Les composants qui ne sont pas stoppés à leur point d'évolution peuvent poursuivre leur comportement, mais leurs transmissions d'événements sont momentanément interrompues. Ensuite, de nouveaux composants et/ou composites pourront être créés. Dans notre exemple, le composant *C3* sera créé. Puis, les composants seront recomposés en remplaçant le composant *C1* par le composant *C3*. Tous les composants pourront transmettre leurs événements, l'architecture *A2* étant opérationnelle.

Dans un premier temps, nous spécifierons le comportement du composant *C1* comportant un point d'évolution. Puis, nous spécifierons les architectures *A1* et *A2* en insistant sur les opérateurs d'évolution.

Le composant possédant un point d'évolution est spécifié par la composition d'un port du type *EvolutionPort*, et d'un comportement, comme dans l'exemple spécifié ci-dessous (figure III.21) :

```
define component type ExempleComposant
   [pC: PortReception [canalReception:[], canalEmission:[] ], pointEvolution: EvolutionPort ] {
port pC: PortReception ||
port pointEvolution: EvolutionPort ||
behaviour ExempleComportement[pC, pointEvolution] }
```

Figure III.21. : exemple de définition d'un type de composant avec point d'évolution

Dans le comportement *ExempleComportement*, le composant pourra être arrêté à son point d'évolution en utilisant l'opérateur ***evolvable***, comme dans l'exemple spécifié ci-dessous (figure III.22) :

define behaviour component type ExempleComportement **[**pC: PortReception **[**canalReception:**[]**, canalEmission:**[]]**, pointEvolution: EvolutionPort **] {**

ExempleComportement**[**pC, pointEvolution**]** = (**evolvable**(pointEvolution) ¤ pC **@** canalReception***()*** ¤ pC **@** canalEmission<> ¤ ExempleComportement**[**pC, pointEvolution**]**), **}**

Figure III.22. : exemple de définition d'un type de comportement de composant avec point d'évolution

Lorsque le comportement *ExempleComportement* reçoit une demande d'arrêt à son point d'évolution, ***evolvable****(pointEvolution)*, il reçoit un événement sur le canal *begin* de son port *pointEvolution*. Le comportement de *C1* se bloque jusqu'à la réception d'un événement sur le canal *end* de son port *pointEvolution*. Contrairement au type de port précédent, ce n'est pas le comportement qui initie le transfert d'événements. De plus, le point d'arrêt, s'il n'est pas activé, n'est pas bloquant, c'est-à-dire que si le canal *canalReception* du port *pC* reçoit un événement, le point d'évolution est passé et ne peut plus être actionné.

Les spécifications des deux architectures sont les suivantes (figure III.23) :

compose A1 **{**

C1 : ComponentType1 ||

C2 : ComponentType2 **[{**canalReception / canalEmission, canalEmission / canalReception **}]**

where

attach C1 **to** C2 **}**

compose A2 **{**

evolvable(A1 **@** C1 **@** pointEvolution) ⇒ **decompose** A1,

C3 : ComponentType3

where

replace C1 **by** C3,

recompose (C3, C2) **}**

Figure III.23. : exemple de compositions d'architectures avec point d'évolution

L'architecture *A2* demande au composant *C1* de l'architecture *A1* de s'arrêter à son point d'évolution, en émettant un événement sur le canal *begin* de son port *pointEvolution*. Lorsque l'événement a été reçu, l'architecture *A1* est décomposée. L'architecture *A2* crée un nouveau composant *C3* de type *ComponentType3*. Ce composant remplace le composant *C1* (***replace*** *C1* ***by*** *C3*). Le remplacement d'un composant ou d'un composite par un autre composant ou composite n'est possible que s'ils ont au moins un port de même nom et de même type, contenant des canaux de même nom. Puis, le composant *C3* sera recomposé avec le composant *C2* (***recompose*** *(C3, C2)*), grâce aux attachements de l'architecture *A1*. Le composant *C1* quitte son point d'évolution en émettant un événement sur le canal *end* du port *pointEvolution*.

2.3.7.2. Evolution d'une architecture permettant le changement d'éléments architecturaux dans un composite

La construction d'une architecture débute par la spécification des principaux composites. Puis, ceux-ci sont décomposés, c'est-à-dire que les composites vont être raffinés par des composants et des composites. Ces derniers seront eux-mêmes décomposés jusqu'à ce que l'ensemble des composites de l'architecture soit raffiné par des composants. Or, le changement, au cours d'une évolution d'architecture, peut porter sur un composite, ou un composant encapsulé dans un composite.

Soit une architecture *A1*, possédant un composite *C1* et un composant *C2*. Le composite *C1* est raffiné par deux composants *C11* et *C12*. L'application issue de l'architecture *A1* ne correspond plus au besoin et nous souhaitons faire évoluer son architecture vers l'architecture *A2*. Cette architecture possède le même composite *C1* et le même composant *C2*. Il s'agit du même composite *C1* car son port, qui est aussi celui du composant *C11*, est resté le même. Par contre, le choix de son raffinement a changé. Le composite *C1* contient le même composant *C11*, mais le composant *C12* est changé par le composant *C13*, ce dernier étant relié à un nouveau composant *C14*. Pour effectuer le changement du composant *C12* et l'ajout du composant *C14* à l'intérieur du composite *C1*, nous utiliserons le composite spécifié ci-dessous (figure III.24) :

```
define composite type EvolueC [C1: Composite] {
evolvable(C1 @ C11 @ portEvolution) ⇒ decompose C1,
C13 : ComponentType3 ||
C14 : ComponentType4
where
replace C12 by C13,
recompose (C11, C13),
attach C13 to C14 }
```

Figure III.24. : exemple de définition d'un type de composite avec évolution

Ce composite reçoit un composite *C1* en paramètre d'entrée. Il arrête le composant *C11* à son point d'évolution en émettant un événement sur le canal *begin* de son port *portEvolution*. Lorsque le composant *C11* est arrêté, le composite est décomposé. Puis, deux instances de composants, *C13* et *C14*, de types *ComponentType3* et *ComponentType4*, sont créées. Le composant *C12* est remplacé par le composant *C13* (***replace*** *C12* ***by*** *C13*). Les composants *C11* et *C13* sont recomposés (***recompose*** *(C11, C13)*). Enfin, les composants *C13* et *C14* sont attachés.

Les spécifications des deux architectures sont les suivantes (figure III.25) :

```
compose A1 {
C1 : ComponentType1 ||
C2 : ComponentType2 [{canalReception / canalEmission, canalEmission / canalReception }]
where
attach C1 to C2 }

compose A2 {
decompose A1,
where
recompose (C1, C2),
EvolueC[C1] }
```

Figure III.25. : exemple de compositions d'architectures avec évolution

L'architecture *A2* demande la décomposition de l'architecture *A1*. Puis, elle recompose le composite *C1* et le composant *C2*. Le composite *C1* est donné comme paramètre d'entrée au modèle de composition *EvolueC*, décrit précédemment figure III.24, permettant l'évolution du composite *C1*.

2.4. Conclusion

Dans cette partie, nous avons présenté un langage de description d'architectures π-SPACE. Ce langage comporte :

- différents éléments architecturaux : ports, comportements, composants, composites, architectures ;
- un cadre compositionnel : la composition des ports et d'un comportement pour le composant, et la composition de composants et/ou de composites pour le composite ;
- la spécification d'architecture dynamique, c'est-à-dire le changement des attachements et la création de nouveaux éléments architecturaux, selon des règles prédéfinies dans l'architecture ;
- un cadre évolutif : le changement de l'architecture d'une application en cours d'exécution.

Du point de vue de sa dynamicité, l'apport de ce langage, par rapport aux autres langages de description d'architecture logicielle, est qu'il permet non seulement de définir le caractère dynamique des architectures (attachements dynamiques et composants dynamiques), mais d'offrir tous les mécanismes nécessaires à la création et à la gestion de composants (cf. paragraphe 2.3.6). De plus, l'originalité de ce langage de description d'architecture est sa capacité à spécifier l'évolution d'une architecture.

Différents outils ont été développés pour π-SPACE. La démarche pour la conception, la simulation et la génération de l'application client-serveur est définie de la façon suivante :

- grâce à l'extension de Rational Rose pour π-SPACE, la composition est décrite en utilisant la notation graphique. Puis, chaque élément architectural est renseigné en utilisant la notation textuelle ;

- à partir de Rational Rose, des fichiers créés à partir de la double notation permettent d'obtenir la description de l'architecture en π-SPACE. Chaque fichier contient un type d'éléments ;
- toujours dans le même outil, ces fichiers peuvent être compilés afin de vérifier la syntaxe et la sémantique de l'application ;
- à partir de la description d'architectures en π-SPACE contenue dans les fichiers, l'architecture peut être simulée avec l'outil développé en Pict.

Dans le cadre de cette thèse, tous les aspects du langage π-SPACE furent pris en considération. Cependant, certains d'entre eux, tels que les aspects dynamiques et évolutifs, ne possèdent pas de contrepartie dans le langage de spécification formel cible de notre travail, les machines abstraites de la méthode B.

3. La méthode B comme langage de spécification formelle

3.1. Introduction

La méthode B est une méthode formelle permettant le développement de logiciels sûrs. Elle a été conçue par Jean-Raymond Abrial [Abrial 1996].

Le développement d'un projet selon la méthode B comporte deux activités étroitement liées : l'**écriture** de textes formels et la **preuve** de ces mêmes textes.

L'activité d'écriture consiste à rédiger les spécifications formelles de *machines abstraites* à l'aide d'un formalisme mathématique de haut niveau. Ainsi, une spécification B comporte des données (qui peuvent être exprimées entre autres par des entiers, des booléens, des ensembles, des relations, des fonctions ou des suites), des *propriétés invariantes* portant sur ces données (exprimées à l'aide de la logique des prédicats du premier ordre), et enfin des *services* permettant d'initialiser puis de faire évoluer ces données (les transformations de ces données sont exprimées à l'aide de substitutions). L'activité de preuve d'une spécification B consiste alors à réaliser un certain nombre de démonstrations afin de prouver l'établissement et la conservation des propriétés invariantes en question (par exemple, il faut prouver que l'appel d'un service conserve bien les propriétés invariantes). La génération des assertions à démontrer est complètement systématique. Elle s'appuie notamment sur la transformation de prédicats par des substitutions.

Le développement d'une machine abstraite se poursuit par une extension de l'activité d'écriture lors d'étapes successives de *raffinement*. Raffiner une spécification consiste à la reformuler en une expression de plus en plus concrète, mais aussi à l'enrichir. L'activité de preuve concernant les raffinements consiste également à réaliser un certain nombre de vérifications statiques et à prouver que le raffinement constitue bien une reformulation valide de la spécification. Le dernier niveau de raffinement d'une machine abstraite se nomme l'*implantation* ou *implémentation*. Il est assujetti à quelques contraintes supplémentaires : par exemple, il ne peut plus manipuler que des données ou des substitutions ayant un équivalent informatique, c'est-à-dire dans les langages de programmation classiques. Les données et les substitutions de l'implémentation constituent un langage informatique similaire à un langage impératif. A ce titre, il peut donc s'exécuter sur un système informatique après fabrication d'un exécutable, soit à l'aide d'un compilateur dédié, soit en passant par une étape intermédiaire de traduction automatique vers Ada, C++ ou C [Steria 1998].

Nous présenterons tout d'abord la notation formelle de la méthode B sous forme de machines abstraites. Puis, nous introduirons les constructions de raffinement et d'implémentation.

3.2. La méthode formelle B : notation et formalisation de modèles

3.2.1. Spécifications formelles en B

Le développement d'une spécification formelle de la méthode B débute par la construction d'un modèle mathématique reprenant toutes les descriptions du modèle d'analyse du système. Le langage utilisé, la notation formelle de la méthode B, est la "*B Abstract Machine Notation*" (B/AMN ou notation de machine abstraite B). Un modèle B constitue une spécification de ce que devra réaliser le système (le quoi).

Un modèle de spécification B peut être raffiné, c'est-à-dire spécialisé, jusqu'à obtenir une implémentation complète du système logiciel (le comment).

La correction d'un modèle de spécification B et la conformité du programme par rapport à ce modèle sont garanties par des preuves mathématiques. Les preuves ne peuvent s'envisager que grâce à l'utilisation d'outils de preuves automatiques.

Dans la suite de cette partie, nous introduirons, tout d'abord, le concept de machine abstraite B. Dans un deuxième temps, nous aborderons les différents types de relations entre machines abstraites. Ensuite, nous décrirons la notation mathématique utilisée par la méthode B. Enfin, dans le paragraphe suivant, nous présenterons les substitutions généralisées, constituants essentiels des opérations des machines abstraites.

3.2.2. Machines abstraites

La machine abstraite est le concept de base de la méthode B. Il s'agit d'un concept très proche des types abstraits de données : une machine contient des variables et des opérations. Elle permet d'encapsuler les variables ; les opérations permettent d'accéder aux variables et de les manipuler.

Les variables d'une machine abstraite sont décrites par des expressions qui utilisent des concepts mathématiques comme les ensembles, les relations, les fonctions, les séquences, etc. Les lois statiques, que les variables doivent suivre, sont définies au moyen de la logique des prédicats et constituent l'invariant de la machine abstraite.

La spécification du comportement des opérations est basée sur une construction mathématique particulière : les substitutions généralisées. Une opération d'une machine abstraite peut contenir une précondition : il s'agit d'un prédicat qui exprime les conditions indispensables pour pouvoir invoquer l'opération. Une opération contient aussi une action : il s'agit d'une substitution généralisée qui exprime comment sont manipulées les variables de la machine.

La figure III.26 présente la structure de la définition d'une machine abstraite :

```
MACHINE
    /* Déclaration du nom de la machine abstraite et de la liste des paramètres formels
    éventuels : ensembles ou valeurs */
    CONSTRAINTS
    /* Déclaration des propriétés logiques des paramètres ensembles ou valeurs de la machine */

    /* Déclaration des relations entre machines */
    PROMOTES
    /* Déclaration des opérations "incluses" qui sont reprises telles quelles dans la machine */

    SETS
    /* Déclaration des définitions des ensembles abstraits ou énumérés de la machine */
    ABSTRACT_CONSTANTS
    [ CONCRETE_ ] CONSTANTS
    /* Déclaration des identificateurs des constantes */
```

```
PROPERTIES
/* Déclaration des propriétés logiques des ensembles et constantes déclarés, avec typage et
expression de valuation des constantes */

[ ABSTRACT_ ] VARIABLES
CONCRETE_VARIABLES
/* Déclaration des identificateurs des variables */
INVARIANT
/* Déclaration des propriétés logiques invariantes des variables déclarées, avec typage des
variables */
ASSERTIONS
/* Déclaration des définitions de nouvelles assertions logiques de la machine */

INITIALISATION
/* Déclaration de la substitution généralisée initialisant les variables de la machine */

OPERATIONS
/* Déclaration des définitions des opérations de la machine sous la forme d'un en-tête et d'un
corps */
END /* Fin de la définition de la machine */
```

Figure III.26. : structure d'une machine abstraite de la méthode B

La spécification d'une machine abstraite est divisée en différentes clauses, chacune étant identifiée par un mot-clé spécifique. Il n'est pas nécessaire que toutes les clauses soient utilisées dans l'écriture d'une machine abstraite, mais chacune d'entre elles ne peut apparaître qu'une seule fois, dans un ordre quelconque.

- La clause "*MACHINE*" est particulière car elle marque le début de la spécification de la machine abstraite ; elle permet aussi de l'identifier par son nom, qui doit être unique, et la liste des paramètres formels qu'elle peut nécessiter. Ces paramètres peuvent être des valeurs scalaires des types prédéfinis, ou bien des ensembles. Le texte de la machine abstraite se termine au "*END*" final.
- La clause "*CONSTRAINTS*" regroupe les propriétés, par exemple le type de ces paramètres, exprimées dans la logique des prédicats du premier ordre.
- Les relations entre machines seront introduites plus en détail dans le paragraphe 3.2.3. La clause "*PROMOTES*", par exemple, indique dans une machine abstraite quelles opérations sont reprises d'une autre machine abstraite incluse.
- Les clauses "*SETS*", "*ABSTRACT_CONSTANTS*" et "*CONCRETE_CONSTANTS*" (ou "*CONSTANTS*") permettent la déclaration d'ensembles et de constantes, dont les propriétés sont définies, dans la logique des prédicats du premier ordre, sous la clause "*PROPERTIES*".
- De façon assez similaire, les clauses "*ABSTRACT_VARIABLES*" (ou "*VARIABLES*") et "*CONCRETE_VARIABLES*" servent à déclarer les identificateurs des variables utilisées dans la spécification de l'état de la machine abstraite. Leurs propriétés logiques invariantes, définies elles aussi dans la logique des prédicats du premier ordre, sont regroupées dans la clause "*INVARIANT*". La clause "*ASSERTIONS*" offre la possibilité d'établir des propriétés supplémentaires, dans le but de faciliter les preuves.
- La clause "*INITIALISATION*" contient une substitution généralisée qui doit affecter une valeur initiale à toutes les variables de la machine abstraite.
- Enfin, la clause "*OPERATIONS*" contient les définitions des différentes opérations de la machine abstraite. Ces opérations permettent de modifier les valeurs des variables, tout en

préservant les propriétés de l'invariant. Chacune d'entre elles est composée d'un en-tête (son nom et les listes de paramètres en entrée et en sortie) et d'un corps (sous la forme d'une substitution généralisée).

Les opérations d'une machine abstraite B servent d'interface pour changer l'état de la machine abstraite, c'est-à-dire les valeurs des variables (les données de la machine). De ce fait, elles ne sont pas autorisées à faire appel l'une à l'autre et, notamment, il est exclu d'avoir recours à une définition récursive du comportement d'une opération.

3.2.3. Relations entre machines abstraites

Les relations suivantes définissent la visibilité des données (variables, ...) et opérations :

- la relation "*INCLUDES*", entre deux machines abstraites ou entre un raffinement et une machine abstraite, est exclusive et transitive pour les données ;
- la relation "*IMPORTS*", uniquement entre une implémentation et une machine abstraite, est exclusive ;
- la relation "*EXTENDS*" étend les relations "*INCLUDES*" ou "*IMPORTS*" ;
- la relation "*USES*", entre deux machines abstraites, peut être partagée, et n'est pas transitive ;
- la relation "*SEES*", entre deux machines abstraites, entre un raffinement et une machine abstraite, ou entre une implémentation et une machine abstraite, peut être partagée, et n'est pas transitive.

Chacune de ces relations implique une dépendance entre les machines concernées. Néanmoins, il ne doit pas y avoir de cycle dans le graphe de dépendance engendré : en effet, il s'agit essentiellement d'inclusion de texte.

Lorsqu'une machine abstraite (ou un raffinement) ***inclut*** (avec la relation "*INCLUDES*") une autre machine abstraite, elle intègre les données de la machine incluse, mais en lecture seulement, et en instanciant les paramètres formels, c'est-à-dire en leur affectant des valeurs. Cette machine peut, en plus de la clause "*INCLUDES*", utiliser la clause "*PROMOTES*" afin de promouvoir les opérations de la machine abstraite incluse : par la suite, ces opérations seront considérées comme faisant également partie de la machine incluante, même si ses opérations pourront encore y faire appel. La relation d'inclusion a pour objectif principal de réutiliser du texte de spécification déjà écrit.

Lorsqu'une machine abstraite, un raffinement ou une implémentation, ***étend*** (par une relation "*EXTENDS*") une machine abstraite *M*, elle hérite les données et toutes les opérations de la machine étendue. Cela revient en fait à une relation "*INCLUDES*" (ou "*IMPORTS*", suivant la nature du module), pour laquelle la clause "*PROMOTES*" porterait sur l'ensemble des opérations de la machine abstraite incluse (ou importée). Par ailleurs, les opérations promues peuvent être appelées dans les clauses "*INITIALISATION*" et "*OPERATIONS*".

Une implémentation qui ***importe*** (avec la relation "*IMPORTS*") une autre machine abstraite peut librement utiliser ses opérations (mais les données ne sont directement accessibles qu'en lecture seulement). Elle peut toutefois utiliser la clause "*PROMOTES*" (ou "*EXTENDS*") pour promouvoir les opérations de la machine abstraite importée. Contrairement à l'inclusion qui ne présuppose rien de particulier à propos de la machine abstraite incluse, l'importation requiert que la machine abstraite ciblée possèdera une implémentation à la fin du développement en B du système. L'implémentation diffère en cela de l'inclusion, car elle ne tend pas à simplement s'économiser des efforts d'écriture de spécification, mais bien cette fois à réutiliser du "code".

En outre, une instance de machine abstraite ne peut pas être importée plus d'une fois dans tout le projet. De plus, une et une seule machine (ou de façon plus général module) ne sera pas instanciée par importation : ce sera le module principal du projet qui permettra, lors de la génération de code, d'obtenir le module principal du programme.

Afin de pouvoir inclure, ou importer, plusieurs fois le même texte de machine abstraite, la méthode B s'est dotée d'un mécanisme de renommage de machines : lors de l'inclusion, de l'importation ou de l'extension d'une machine abstraite, le nom de la machine peut être identifié de façon unique en le préfixant par un nom d'instance, par exemple *nom_instance.Nom_machine*. Toutes les opérations, variables, … sont alors préfixées par le même nom d'instance (cf. figure III.27). Malgré tout, il ne doit y avoir au plus qu'un seul préfixe de renommage pour l'inclusion et l'importation.

```
MACHINE N
    INCLUDES a.M, b.M
    …
    x := a.vm … y := b.vm … z <-- a.opm(p) …
END
```

Figure III.27. : exemple d'utilisation de données et opérations de plusieurs instances d'une même machine abstraite incluse

Lorsqu'une machine abstraite ***utilise*** (avec la relation "*USES*") une autre machine abstraite, elle peut en utiliser les données (y compris les paramètres formels) dans ses invariants et ses opérations (sans les modifier). Cependant, il doit exister dans le projet une machine abstraite tierce qui inclut une instance de chaque machine (utilisée et utilisante). Cette relation permet donc de consulter l'état d'une autre machine, sans pouvoir pour autant l'altérer. Par conséquent, ne pouvant se suffire à elle-même, une machine abstraite qui en utilise une autre ne doit être ni raffinée, ni vue, ni importée dans tout le reste du projet de développement en B.

Une machine abstraite, un raffinement ou une implémentation, qui en ***voit*** une autre (par la relation "*SEES*"), peut en consulter les données et en utiliser les opérations qui ne modifient pas les données. Néanmoins, elle permet de voir une instance donnée de la machine abstraite visée, mais elle n'en aura qu'une vue générale, et somme toute assez limitée. D'autre part, les raffinements ultérieurs de la machine abstraite qui en voit une autre doivent voir la même instance.

Par ailleurs, toute instance de machine vue doit être importée quelque part dans le projet : en effet, il faut s'assurer que ce qui est regardé existe bien au final. Par contre, un composant d'un module (machine abstraite, raffinement ou implémentation) ne peut pas voir une instance importée par une dépendant de ce module, essentiellement parce que ce lien devient alors inutile à ce niveau de la hiérarchie de l'arbre de dépendance, et cela ne ferait que le compliquer.

Enfin, tout comme les cycles sont interdits dans le graphe de dépendance, un composant ne peut pas posséder plusieurs liens, de même nature ou de natures différentes, sur une même instance.

3.2.4. Notation mathématique pour B

La boîte à outils mathématiques de la méthode B comprend de nombreux compartiments, parmi lesquels :

- les enregistrements (ou "*records*"),
- la théorie des ensembles,
- les relations,
- les fonctions,
- les nombres,
- les séquences (ou suites),
- les arbres,
- la logique des prédicats,

- les substitutions généralisées (développées dans le paragraphe suivant 3.2.5).

Tout d'abord, quelques éléments ne correspondent pas à une catégorie bien précise citée précédemment, mais ont toutefois leur importance.

"==" signale une définition en B, par exemple pour donner une valeur à une constante. "/*" et "*/" servent, de façon générale, de délimiteurs, respectivement de début et de fin, de commentaires.

L'ensemble de toutes les chaînes de caractères possibles est noté "*STRING*", une chaîne étant délimité par des guillemets (par exemple *"je suis une chaine"*).

De façon à faire référence à la valeur précédente d'une variable (notamment lors des preuves), la méthode B propose d'utiliser "*$0*" comme suffixe au nom de la variable.

Les **enregistrements** peuvent être définis en extension avec "*rec*" ; chaque champ peut être identifié par un nom, placé devant la valeur. Par exemple, *rec(I1:x1, I2:x2)* définit un enregistrement comprenant deux champs, dont le premier champ *I1* a pour valeur *x1*, et le deuxième champ *I2* a pour valeur *x2*. Le mot-clé "*struct*" permet de définir un ensemble d'enregistrements, comme dans *struct(I1:E1, I2:E2)*, où *E1* et *E2* sont des ensembles. L'accès à la valeur d'un champ d'un enregistrement se fait par l'utilisation d'une apostrophe entre le nom de l'enregistrement et celui du champ désigné : ainsi, *R'I* dénote le champ *I* de l'enregistrement *R*.

La méthode B permet de définir un **ensemble** soit par énumération (en extension) en donnant entre accolades tous les éléments de l'ensemble, séparés par des virgules (par exemple : *{e1, ..., en}*), soit par intention : *{x | x : E & P}* représente l'ensemble des éléments de *E* qui satisfont le prédicat *P*. En effet, l'appartenance à un ensemble est indiquée par l'opérateur ":", tandis que la non appartenance l'est par "/:". L'égalité de deux ensembles est simplement assurée par "=", l'inégalité par "/=". L'ensemble vide est noté "*{}*". L'inclusion au sens large du sous-ensemble *SE* dans l'ensemble *E* s'écrit "*SE <: E*", et son contraire "*SE /<: E*" ; au sens stricte, cela devient respectivement "*SE <<: E*" et "*SE /<<: E*". L'union des ensembles *E1* et *E2* se note "*E1 \/ E2*", leur intersection "*E1 /\ E2*", leur différence "*E1 - E2*" et leur produit cartésien "*E1 * E2*". Le cardinal d'un ensemble *E* est "*card(E)*", l'ensemble de ses parties "*POW(E)*", l'ensemble des parties non vides "*POW1(E)*", l'ensemble des parties finies "*FIN(E)*", et l'ensemble des parties finies non vides "*FIN1(E)*". L'union quantifiée (introduite par "*UNION*") est l'union d'ensembles vérifiant un prédicat donné ; l'intersection quantifiée ("*INTER*") représente également l'intersection des ensembles vérifiant un certain prédicat. L'union généralisée ("*union(E)*") et l'intersection généralisée ("*inter(E)*") s'appliquent à un ensemble *E* dont les éléments sont eux-mêmes des ensembles dont on souhaite faire l'union ou l'intersection, respectivement.

L'ensemble des **relations** binaires *r* définies sur *E1*E2* est définit en B par "*E1 <-> E2*". Un élément spécifique de la relation *r* est noté "*e1 |-> e2*". Le domaine de *r* est "*dom(r)*", son codomaine "*ran(r)*", tandis que la restriction du domaine de *r* sur l'ensemble *E* est spécifiée par "*E <| r*", la restriction du codomaine de *r* sur *E* par "*r |> E*", la soustraction (ou antirestriction) du domaine de *r* par *E* est "*E <<| r*", et la soustraction du codomaine de *r* par *E* est "*r |>> E*".

La relation inverse de *r* est notée "*r~*". L'image d'un ensemble *E* par *r* est "*r[E]*" et la surcharge de *r* avec un élément *e1 |-> e2* s'écrit "*r <+ {e1|->e2}*". La fermeture réflexive et transitive de *r* est "*closure(r)*", alors que la fermeture transitive est "*closure1(r)*". En fait, deux relations *r1* et *r2* sont composées par "*(r1 ; r2)*", alors que leur produit direct (un antécédent commun dans les deux relations aura pour image le couple formé par ses deux images dans les relations) est noté "*r1 >< r2*", et leur produit parallèle (chaque couple possible de deux antécédents, le premier de la première relation et le second de la deuxième relation, est associé au couple de leurs images) est "*(r1 || r2)*". L'itération *n* fois d'une relation *r* est "*iterate(r,n)*".

La relation identité sur un ensemble *E* est définie comme "*id(E)*". La première projection de deux ensembles *E* et *F*, "*prj1(E,F)*", est la relation associant à chaque couple composé d'un élément de *E* et d'un élément de *F*, l'élément de *E*. De la même façon, la deuxième projection de deux ensembles *E* et *F*, "*prj2(E,F)*", est la relation associant à chaque couple composé d'un élément de *E* et d'un élément de *F*, l'élément de *F* correspondant.

Enfin, la transformée en fonction de la relation *r* est "*fnc(r)*".

La méthode B permet donc également de manipuler les **fonctions**. Pour ce faire, nous avons la possibilité de définir une fonction *f* comme étant :

- une fonction partielle d'un ensemble *E1* vers un ensemble *E2* : "*f : E1 +-> E2*" ;
- une fonction totale : "*f : E1 --> E2*" ;
- une injection partielle : "*f : E1 >+> E2*" ;
- une injection totale : "*f : E1 >-> E2*" ;
- une surjection partielle : "*f : E1 +->> E2*" ;
- une surjection totale : "*f : E1 -->> E2*" ;
- une bijection partielle : "*f : E1 >+>> E2*" ;
- une bijection totale : "*f : E1 >->> E2*".

La fonction inverse de *f* est notée "*f~*". Evaluer la fonction sur un élément donné se fait par la notation usuelle "*f(…)*". En outre, nous avons la possibilité de revenir d'une fonction à une relation, et la transformée en relation de *f* est "*rel(f)*".

Enfin, une lambda expression peut être écrite sous la forme "*%x.(...)*".

Les **nombres** en B sont restreints aux entiers. Néanmoins, une distinction est faite entre les ensembles de nombres entiers qui seront utilisés pour des données abstraites, et ceux qui serviront pour des données concrètes. Ainsi, "*NATURAL*", "*NATURAL1*" et "*INTEGER*" désignent respectivement les ensembles des entiers naturels, des entiers naturels non nuls et des entiers relatifs, pour les données abstraites. Au niveau concret, ces ensembles deviennent respectivement "*NAT*", "*NAT1*" et "*INT*". Deux constantes, "*MAXINT*" et "*MININT*", viennent compléter ces définitions des entiers utilisables, en représentant le plus grand et le plus petit entiers implémentables. Il est possible de demander le prédécesseur ou le successeur d'un entier *x*, par "*pred(x)*" et "*succ(x)*", mais aussi de décrire un intervalle entre deux nombres *inf* et *sup* par "*inf .. sup*". Par ailleurs, "*min(E)*" et "*max(E)*" retournent respectivement le minimum et le maximum d'un ensemble de nombres entiers *E*.

Pour manipuler ces entiers, nous disposons des opérations arithmétiques usuelles, addition "+", soustraction "-", multiplication "*", division entière "/" et reste de la division entière "*mod*", mais aussi des opérateurs de comparaisons numériques ("=", "/=", "<", "<=", ">" et ">="). De plus, il est possible de définir la puissance *y* de *x* avec "*x ** y*". La somme quantifiée, *Σ(X).(P | E)*, représente la somme des expressions de *E* correspondant aux valeurs des variables *X* qui établissent *P*, et elle est notée "*SIGMA*". Le produit quantifié, *Π(X).(P | E)*, représente le produit des expressions de *E* correspondant aux valeurs des variables *X* qui établissent *P*, et il est noté "*PI*".

Différents opérateurs permettent de manipuler des **séquences** ou suites d'éléments en B. Ainsi, "*seq(E)*" représente l'ensemble des séquences possibles d'éléments de l'ensemble *E*. "*seq1(E)*" est l'ensemble des séquences non vides définies sur *E*, tandis que "*iseq(E)*" et "*iseq1(E)*" sont respectivement les ensembles des suites injectives (pas deux fois le même élément), et des suites injectives non vides. "*perm(E)*" est l'ensemble des permutations (ou suites bijectives) qui peuvent être définies sur *E*.

Une séquence peut être définie directement en extension : par exemple "*[s1, ..., sn]*" désigne la suite dont les *n* éléments sont, dans l'ordre, *s1, ..., sn*. La suite vide est notée "*[]*". Deux séquences *S1* et *S2* peuvent être concaténées, avec "*S1 ^ S2*", et d'avantage avec la concaténation généralisée, "*conc(S)*", où *S* est l'ensemble de séquences à concaténer.

L'inverse d'une séquence *S* est défini par "*rev(S)*", sa taille par "*size(S)*", son premier élément par "*first(S)*", sa tête (la séquence sans le dernier élément) par "*front(S)*", son dernier élément par "*last(S)*", et sa queue (la séquence sans le premier élément) par "*tail(S)*". L'insertion d'un élément *x* en tête de la séquence *S* s'écrit "*x->S*" et son insertion en fin "*S<-x*". La restriction de *S* aux *n*

éléments de tête est "*S/|\n*" et la restriction de *S* à la queue, après les *n* premiers éléments, est "*S\|/n*".

En plus des ensembles, relations binaires, fonctions et séquences, la méthode B s'est également doté de primitives sur une autre structure de données : les **arbres**. Ainsi, "*tree(E)*" représente l'ensemble des arbres qui peuvent être définis sur l'ensemble *E*. La construction d'un arbre donné peut se faire avec "*const(x,q)*", qui représente l'arbre dont la racine est associée à *x*, et dont les fils sont les éléments de la liste *q*. La taille d'un arbre *t* (son nombre de nœuds) est "*sizet(t)*" ; la valeur associée à sa racine est "*top(t)*", et la séquence des fils de la racine est "*sons(t)*". "*subtree(t,n)*" représente le sous arbre de l'arbre *t* dont la racine est le nœud *n*. Le symétrique d'un arbre *t* est "*mirror(t)*". "*prefix(t)*" et "*postfix(t)*" donnent les aplatissements sous forme de suites préfixées et postfixées de l'arbre *t*. L'arité du nœud *n* de l'arbre *t*, c'est-à-dire son nombre de fils, est "*arity(t,n)*" ; son rang, ou plutôt celui de la branche le reliant à son nœud père "*father(t,n)*", est "*rank(t,n)*". Le $i^{ème}$ fils du nœud est "*son(t,n,i)*".

Certains opérateurs ne s'appliquent qu'aux arbres binaires. L'ensemble des arbres binaires qui peuvent être définis sur l'ensemble *E* est "*btree(E)*", alors que "*bin*" permet de créer un arbre binaire en extension et prend pour cela soit un paramètre (la valeur du nœud), soit trois : le fils gauche, la valeur de la racine et le fils droit. D'ailleurs, il est possible de retrouver le fils gauche d'un arbre binaire *t* avec "*left(t)*" et le fils droit avec "*right(t)*". "*infix(t)*" permet d'obtenir l'aplatissement infixé d'un arbre binaire *t*.

La **logique des prédicats** du premier ordre sert en de maintes occasions lors d'une spécification avec la méthode B, depuis les contraintes sur les paramètres d'une machine abstraite, jusque dans l'écriture des substitutions généralisées. Les opérateurs logiques principaux sont définis, comme : la conjonction de deux prédicats *P1* et *P2* avec "*P1 & P2*", leur disjonction "*P1 or P2*", l'implication "*P1 => P2*", l'équivalence "*P1 <=> P2*" ou la négation du prédicat *P* : "*not(P)*". Le parenthésage des formules est possible, tout comme l'écriture des quantificateurs universels (par exemple "*!x.(P)*") ou existentiels ("*#x.P*"). Par ailleurs, l'ensemble des booléens est défini sous l'appellation "*BOOL*", et contient la constante booléenne littérale *vrai*, notée "*TRUE*", et la constante booléenne littérale *faux*, notée "*FALSE*". La conversion d'un prédicat en booléen est assurée par l'opérateur "*bool*".

3.2.5. Substitutions généralisées

Une *substitution généralisée* est une construction mathématique pour l'expression de spécifications formelles. Dans les spécifications de la méthode B, les substitutions généralisées sont utilisées dans le cadre de l'initialisation des variables, et pour la description du comportement des opérations.

La plupart des *substitutions généralisées* sont utilisables dans les machines abstraites :

- substitution bloc ("*BEGIN*"),
- substitution (multiple) simultanée ("||"),
- substitution choix borné ("*CHOICE*"),
- substitution (avec) précondition ("*PRE*"),
- substitution avec garde ou sélection ("*SELECT*"),
- substitution non-opération ou identité ("*skip*"),
- substitution non-déterministe choix non borné ("*ANY*"),
- substitution définition locale ("*LET*"),
- substitution condition par cas ("*CASE*"),
- substitution conditionnelle ("*IF*"),
- substitution appel d'opération (" <-- "),

- substitution "devient égal" (" *:=* "),
- substitution "devient élément de" (" *::* "),
- substitution "devient tel que" (" *:(…)* "),
- substitution assertion ("*ASSERT*").

La substitution bloc correspond à un parenthésage d'une autre substitution *Substitution_S* ; sa syntaxe est illustrée par la figure III.28.

```
BEGIN
    Substitution_S
END
```

Figure III.28. : schéma d'utilisation de la substitution bloc

La substitution généralisée simultanée (multiple) sert à mettre en parallèle deux substitutions, qui seront en fait effectuées dans un ordre quelconque ; la syntaxe en est simplement :
Substitution_S1 || *Substitution_S2*.

La substitution de choix borné permet de choisir entre deux substitutions, et ainsi de faire soit l'une, soit l'autre (cf. figure III.29).

```
CHOICE Substitution_S1
    OR Substitution_S2
END
```

Figure III.29. : schéma d'utilisation de la substitution choix borné

La substitution (avec) précondition permet, si *Prédicat_P* est vrai, alors se comporter comme *Substitution_S* (cf. figure III.30). Par contre, si *Prédicat_P* est faux, alors *Substitution_S* ne doit pas se faire.

```
PRE Prédicat_P
    THEN Substitution_S
END
```

Figure III.30. : schéma d'utilisation de la substitution précondition

La substitution avec garde (ou sélection) possède plusieurs fonctionnements possibles (cf. figure III.31). Tout d'abord, le cas le plus simple est basé sur une seule substitution gardée *Substitution_S* : si le prédicat *Prédicat_P* est vrai, alors seulement *Substitution_S* sera faite (si *Prédicat_P* est faux, alors l'exécution de *Substitution_S* ne peut pas se faire). Dans le cas où plusieurs substitutions sont concernées, chacune est gardée : si *Prédicat_P1* est vrai, alors faire *Substitution_S1* et si *Prédicat_P2* est vrai, alors faire *Substitution_S2* et … et si *Prédicat_Pn* est vrai, alors faire *Substitution_Sn*. Enfin, il est possible de définir une substitution par défaut pour couvrir les autres cas, en complément de ceux définis par les gardes : si *Prédicat_P1* est vrai, alors faire *Substitution_S1* et … et si *Prédicat_Pn* est vrai, alors faire *Substitution_Sn* et si tous les prédicats sont faux faire *Substitution_S.*

SELECT *Prédicat_P*
THEN *Substitution_S*
END

SELECT *Prédicat_P1* THEN *Substitution_S1*
WHEN *Prédicat_P2* THEN *Substitution_S2*
… WHEN *Prédicat_Pn* THEN *Substitution_Sn*
END

SELECT *Prédicat_P1* THEN *Substitution_S1*
… WHEN *Prédicat_Pn* THEN *Substitution_Sn*
ELSE *Substitution_S* END

Figure III.31. : schéma d'utilisation de la substitution avec garde

La substitution identité (non opération) permet de ne spécifier aucune opération spéciale. Simplement écrite ***skip***, elle peut être, par exemple, utilisée dans une machine abstraite, puis raffinée par n'importe quelle autre substitution dans la suite du développement du projet.

La substitution de choix non borné, non déterministe, de spécification, définit une variable locale, sans avoir forcément à lui donner une valeur précise pour son initialisation (cf. figure III.32) : pour une valeur quelconque de *variable_v* qui satisfait *Prédicat_P*, alors faire *Substitution_S* (*variable_v* doit être une liste non vide de variables locales).

ANY *variable_v*
WHERE *Prédicat_P*
THEN *Substitution_S*
END

Figure III.32. : schéma d'utilisation de la substitution de choix non borné, non déterministe de spécification

La substitution de définition locale permet au contraire de définir une constante locale, c'est-à-dire une donnée initialisée, dont la valeur ne pourra pas être modifiée (cf. figure III.33) : pour la valeur de *constante_c* définie par le *Prédicat_P*, alors faire *Substitution_S* (*constante_c* doit être une liste non vide de données abstraites et *Prédicat_P* est une conjonction de prédicats donnant à chacune sa valeur).

LET *constante_c1 , constante_c2*
BE *constante_c1 = E1* $\wedge$ *constante_c2 = E2*
IN *Substitution_S*
END

Figure III.33. : schéma d'utilisation de la substitution de définition locale

La substitution de condition par cas, assez proche de la substitution avec garde, est chargée de définir différents comportements possibles en fonction de la valeur d'une expression. Comme pour la substitution avec garde, une "action" par défaut peut être définie, comme décrit dans la figure III.34. Néanmoins, la syntaxe est plus contraignante car elle nécessite que les listes de constantes des différentes branches, "*EITHER*" et "*OR*", doivent être deux à deux disjointes.

```
CASE E OF
    EITHER L1 THEN Substitution_S1
    OR L2 THEN Substitution_S2
    OR …
    OR Ln THEN Substitution_Sn
    END
END


CASE E OF
    EITHER L1 THEN Substitution_S1
    OR L2 THEN Substitution_S2
    OR …
    OR Ln THEN Substitution_Sn
    ELSE Substitution_S
    END
END
```

Figure III.34. : schéma d'utilisation de la substitution de condition par cas

La substitution conditionnelle peut également être vue comme étant composée à partir d'une substitution avec garde. Son schéma de fonctionnement correspond au "si … alors … sinon …" habituel (cf. figure III.35) : si *Prédicat_P* est vrai, alors faire *Substitution_S1* sinon faire *Substitution_S2*.

```
IF Prédicat_P
    THEN Substitution_S1
    ELSE Substitution_S2
END
```

Figure III.35. : schéma d'utilisation de la substitution conditionnelle

La substitution d'appel d'opération permet d'appliquer la substitution d'une opération, en remplaçant les paramètres formels par des paramètres effectifs, et en recueillant les paramètres de sortie dans des données accessibles en écriture. La syntaxe est très proche de celle de l'en-tête de l'opération lui-même (cf. paragraphe 3.2.6), sauf qu'ici les paramètres entre parenthèses sont des paramètres effectifs :
variables_v <-- *opération_O* **(***expressions_E***)** .

La substitution "devient égal" sert pour l'affectation : il est possible de remplacer une variable, ou plusieurs, ou un élément de fonction, ou un champ de record, par autant d'expressions. La syntaxe dans ces différents cas reste la même : *variables_v* **:=** *expressions_E*
ou *fonction_f*(*paramètres_p*) **:=** *expression_E* ou *record_r'champ_c* **:=** *expression_E* .

La substitution "devient élément de" est assez proche de la précédente, étant donné qu'elle affecte à une variable une valeur quelconque d'un ensemble, un peu dans le même esprit que la substitution de choix non borné non déterministe ("*ANY*"), mais sans la définition à caractère local :
variable_v **::** *ens_E* .

La substitution "devient tel que", elle, va chercher des valeurs en fonction d'un prédicat plutôt que d'un ensemble. En outre, elle peut être complètement définie à partir de la substitution de choix non borné non déterministe ("*ANY*"). Il s'agit en fait de remplacer une liste de variables *variables_X* par autant de valeurs qui satisfont un prédicat *Prédicat_P* (les variables doivent être typées au début de *Prédicat_P*) : *variables_X* **:(** *Prédicat_P* **)** .

La substitution assertion (cf. figure III.36) va appliquer *Substitution_S* sous l'assertion que *Prédicat_P* est vrai ; tout comme pour la substitution précondition, si *Prédicat_P* est faux, alors l'exécution de *Substitution_S* ne doit pas se faire.

```
ASSERT Prédicat_P
    THEN Substitution_S
END
```

Figure III.36. : schéma d'utilisation de la substitution assertion

Certaines substitutions (ou éléments du langage) ne sont pas autorisées dans les machines abstraites. Cependant, elles peuvent être introduites par raffinement, soit dans les modules de raffinement ("*REFINEMENT*"), soit dans les modules d'implémentation ("*IMPLEMENTATION*").

Les substitutions généralisées de raffinement regroupent les substitutions généralisées de spécification, plus les :

- substitution variable locale ("*VAR*"),
- substitution séquencement (";").

Par ailleurs, les substitutions suivantes sont interdites dans les spécifications :

- substitution variable locale ("*VAR*"),
- substitution séquencement (";"),
- substitution boucle "tant que" ("*WHILE*"), qui est également interdite dans les raffinements.

La substitution de variable locale non déterministe et non bornée d'implémentation (figure III.37) peut être utilisée pour raffiner la substitution de choix non borné, non déterministe, de spécification ("*ANY*") : pour une valeur quelconque de *variable_v* (*variable_v* doit être une liste non vide de variables locales), on effectue *Substitution_S*.

```
VAR variable_v
    IN Substitution_S
END
```

Figure III.37. : schéma d'utilisation de la substitution de variable locale non déterministe et non bornée d'implémentation

La substitution de séquencement (ou de composition séquentielle) permet d'enchaîner deux substitutions, et ainsi de faire *Substitution_S1* puis *Substitution_S2*, selon la syntaxe :

Substitution_S1 **;** *Substitution_S2* (on peut noter que ce séparateur d'opérations "**;**" ne doit jamais apparaître devant un "*END*").

Les substitutions généralisées d'implémentation correspondent à un niveau d'abstraction encore plus restrictif que pour les machines abstraites. En effet, dans l'optique de pouvoir générer du code dans un langage de programmation impératif, seules les constructions déterministes sont autorisées. Les substitutions généralisées d'implémentation comprennent certaines substitutions généralisées de raffinement, plus la substitution boucle "tant que" ("*WHILE*").

Les substitutions interdites dans les implémentations finales sont les :

- substitution (multiple) simultanée ("||"),
- substitution de choix borné ("*CHOICE*"),
- substitution (avec) précondition ("*PRE*"),
- substitution avec garde ("*SELECT*"),

- substitution de choix non déterministe non borné ("*ANY*"),
- substitution de définition locale ("*LET*"),
- substitution "devient élément de" (" *::* "),
- substitution "devient tel que" (" *:(…)* ").

La substitution de boucle permet d'itérer plusieurs fois une même substitution (cf. figure III.38) : tant que *Prédicat_TQ* est vrai, on fait *Substitution_S*. L'invariant *Prédicat_I* doit être vrai au début de chaque itération de boucle et à la terminaison de la boucle ; le variant *expression_var* est une expression de type entier naturel qui doit décroître à chaque itération.

```
WHILE Prédicat_TQ
    DO Substitution_S
    INVARIANT Prédicat_I
    VARIANT expression_var
END
```

Figure III.38. : schéma d'utilisation de la substitution de boucle

3.2.6. Opérations de machine abstraite

Une définition d'opération dans une machine M peut être vue comme l'expression d'un "contrat" entre M, qui fournit l'opération (côté serveur), et les machines qui appellent l'opération (les clients) : *si* un client appelle l'opération avec la précondition *Précondition* satisfaite, *alors* le serveur exécute une action qui vérifie la spécification exprimée par *Substitutions*. La définition d'une opération de machine abstraite débute par la liste (éventuelle) des paramètres de sortie *res* suivie d'une flèche ("<--"), puis le nom de l'opération *opération* et la liste (éventuelle) des paramètres d'entrée (*args*) ; le signe "=" sépare l'en-tête du corps de l'opération, constitué d'une substitution généralisée (cf. figure III.39).

```
res <-- opération (args ) =
    PRE Précondition
    THEN Substitutions
END
```

Figure III.39. : exemple de définition d'opération

Si plusieurs opérations sont déclarées successivement dans la machine abstraite, leurs spécifications sont séparées par un point virgule ("**;**").

3.3. Raffinement et implémentation de machines abstraites

3.3.1. Raffinement en B

Le raffinement en B est l'activité de développement d'une spécification moins abstraite (plus concrète) à partir d'une spécification abstraite : chaque machine abstraite peut être raffinée jusqu'à obtenir une implémentation. Une seule machine (ou raffinement) est raffinée à la fois, mais un même raffinement peut rester équivalent à plusieurs machines plus abstraites (mais de façon implicite).

Le raffinement se fait par transformation des variables et des opérations de la machine abstraite "à raffiner" : la machine "raffinée" doit pouvoir simuler le comportement observable de la machine abstraite "à raffiner". La correction des raffinements est garantie par des obligations de preuve (preuves mathématiques).

Lors d'un raffinement, les préconditions des substitutions (essentiellement les opérations, mais éventuellement aussi les initialisations) sont affaiblies, le non-déterminisme des résultats est réduit, et les structures de données qui ne l'étaient pas sont définies.

Le raffinement dans la méthode B peut s'appliquer :

- d'une "*MACHINE*" abstraite vers un raffinement ("*REFINEMENT*"),
- d'un raffinement vers un raffinement ("*REFINEMENT*"),
- d'une "*MACHINE*" abstraite vers une "*IMPLEMENTATION*",
- d'un raffinement ("*REFINEMENT*") vers une "*IMPLEMENTATION*".

Les paragraphes suivants présenteront tout d'abord la construction d'un "*REFINEMENT*", puis celle d'une "*IMPLEMENTATION*".

3.3.2. Modules de raffinement

Un module de raffinement B est distinct d'une machine abstraite en plusieurs points. La première différence apparaît dans la déclaration même du module, dans laquelle "*MACHINE*" est remplacée par "*REFINEMENT*". De plus, un raffinement doit posséder une clause "REFINES" qui référence la machine abstraite raffinée (cf. figure III.40), et les mêmes signatures d'opération (les mêmes noms d'opérations avec les mêmes noms de paramètres) que sa machine abstraite.

```
REFINEMENT N
   REFINES M
   …
END
```

Figure III.40. : en-tête d'un raffinement dans la méthode B

La seule **substitution généralisée**, que les opérations et l'initialisation de variables dans un raffinement ne peuvent pas utiliser, est la substitution de boucle ("*WHILE*"), comme énoncé dans le paragraphe précédent 3.2.5.

Une relation de raffinement, qui donne la correspondance entre les niveaux abstrait et concret, sur la base des variables raffinées, est intégrée à l'invariant du raffinement.

Le **raffinement d'une opération** répond à des critères bien précis. En effet, soient *m* l'ensemble des variables d'une machine abstraite (ou un raffinement) *M*, et *n* l'ensemble des variables d'un raffinement *N* de *M*. Alors, une opération *opN* raffine une opération *opM*, sous une relation de raffinement *R* si, pour toute variable de *m* et *n* (avec *m* correspondant par *R* à *n*), toute exécution possible de *opN*, à partir d'un état *n*, mène à un état *n'*, correspondant par *R* avec *m'*, qui peut être atteint par l'exécution de *opM* à partir d'un état *m*. Ainsi, un changement d'état dans le raffinement doit correspondre à un changement d'état dans la machine abstraite. Nous retrouvons dans ce cas la relation d'affaiblissement des préconditions et de renforcement des postconditions du paragraphe 3.3.1.

Finalement, les signatures de *opM* et *opN* doivent être les mêmes, mais pas forcément les substitutions définissant les opérations.

La figure III.41 présente la **structure** de la définition d'un raffinement :

```
REFINEMENT /* Déclaration du nom du raffinement "concret" */

    REFINES /* Déclaration du nom de la machine ou du raffinement "abstrait" */

    SEES /* Déclaration des machines vues */
    INCLUDES /* Déclaration des machines incluses */
    PROMOTES /* Déclaration des opérations promues à partir des machines incluses et qui
    sont reprises telles quelles dans le raffinement */
    EXTENDS /* Déclaration des machines étendues */

    SETS
    /* Déclaration des définitions des ensembles abstraits ou énumérés du raffinement */
    ABSTRACT_CONSTANTS
    [ CONCRETE_ ] CONSTANTS
    /* Déclaration des identificateurs des constantes */
    PROPERTIES
    /* Déclaration des propriétés logiques des ensembles et constantes déclarés, avec typage et
    expression de valuation des constantes */

    [ ABSTRACT_ ] VARIABLES
    CONCRETE_VARIABLES
    /* Déclaration des identificateurs des variables */
    INVARIANT
    /* Déclaration des propriétés logiques invariantes des variables déclarées, avec typage des
    variables, et déclaration de la relation de raffinement */
    ASSERTIONS
    /* Déclaration des définitions de nouvelles assertions logiques du raffinement */

    INITIALISATION
    /* Déclaration de la substitution généralisée initialisant les variables du raffinement */

    OPERATIONS
    /* Déclaration des définitions des opérations du raffinement */

END /* Fin de la définition du raffinement */
```

Figure III.41. : structure d'un raffinement de la méthode B

Nous retrouvons la division de la spécification en différentes clauses. Toutefois, la clause "*CONSTRAINTS*" disparaît : effectivement, même si les paramètres de la machine abstraite continuent à être utilisés, ils ont été définis une fois pour toute dans cette machine abstraite, et tous les raffinements successifs ne peuvent pas les remettre en question. Parmi les relations possibles entre un raffinement et des machines abstraites, le lien "*USES*" n'est plus permis, et le lien "*IMPORTS*" n'est pas encore autorisé. Néanmoins, le raffinement doit conserver au moins les liens "*SEES*" des niveaux d'abstraction précédents.

Pour ce qui est du reste de la structure du raffinement, nous pouvons juste noter que l'"*INVARIANT*" doit contenir au moins la relation de raffinement, et que les "*OPERATIONS*" doivent être les mêmes que dans les niveaux d'abstraction antérieurs, bien que leurs substitutions et celle de l'"*INITIALISATION*" puissent changer.

Les figures III.42 et III.43 nous montre un **exemple** de machine abstraite, appelée *Minimal*, et d'un raffinement possible, *Minimal_r1*.

```
MACHINE Minimal
    VARIABLES       x
    INVARIANT       x : NAT
    INITIALISATION  x :: NAT
    OPERATIONS      up(v) =
                    PRE v : NAT
                    THEN x := x + v
                    END

END /* Minimal */
```

Figure III.42. : exemple de machine abstraite

```
REFINEMENT Minimal_r1
    REFINES Minimal
    VARIABLES       y
    INVARIANT       y : NAT &
                    y = x /* relation de raffinement */
    INITIALISATION  y := 0
    OPERATIONS      up(v) =
                    PRE v : NAT
                    THEN y := y + v
                    END

END /* Raffinement Minimal_r1 */
```

Figure III.43. : exemple de raffinement

Le raffinement peut **porter sur** les données, les substitutions généralisées (le comportement) ou sur ces deux aspects à la fois.

Le raffinement des données se fait essentiellement par la concrétisation des structures de données, c'est-à-dire par la concrétisation des constantes et ensembles (valuation des constantes, valuation des ensembles par énumération, définition des ensembles différés), ou par la concrétisation des variables.

Toutefois, le raffinement des substitutions généralisées peut revêtir différentes formes. D'une part, il peut être focalisé sur l'affaiblissement de la précondition d'une opération : la précondition de l'opération abstraite doit impliquer la précondition de l'opération concrète ; par exemple, la substitution avec précondition ("*PRE*") peut être raffinée par une substitution conditionnelle ("*IF*").

D'autre part, l'objectif peut également être la réduction du non déterminisme du résultat d'une opération : le résultat de l'opération concrète doit impliquer le résultat de l'opération abstraite ; la substitution de choix non borné ("*CHOICE S1 OR S2 END*") peut être dans ce cas raffinée par une des alternatives ("*S1*"), et la substitution "devient élément de ("*v* :: *E*") par "*v* := *exp*", où *exp* serait une expression retournant une valeur de l'ensemble E.

Enfin, la réduction du non déterminisme peut se faire par l'initialisation des variables locales, comme en raffinant une substitution de choix non déterministe non borné ("*ANY*") par une substitution variable locale ("*VAR*").

L'**exemple** de la figure III.44 présente une machine abstraite *Array*, permettant de stocker (*store*) des valeurs dans un tabeau d'entiers naturels *contents*, et de les consulter (*value*) ; *ptr* est l'indice du dernier élément accédé par les opérations *store* (écriture) ou *value* (lecture).

```
MACHINE Array(maxsize)
    CONSTRAINTS maxsize >= 1
    VARIABLES contents, ptr
    INVARIANT contents : 1..maxsize --> NAT & ptr : 1..maxsize
    INITIALISATION contents := %i.(i : NAT & i : 1..maxsize | 0) || ptr :: 1..maxsize
    OPERATIONS
        store (i, v) =
        PRE v : NAT & i : 1..maxsize
        THEN contents(i) := v || ptr := i
        END;

        v <--  value (i) =
        PRE i : 1..maxsize
        THEN v := contents(i) || ptr := i
        END
END /* Machine Array */
```

Figure III.44. : exemple de machine abstraite

La figure III.45 propose un raffinement *Array_r1* de la machine abstraite *Array*. Nous nous apercevons que les deux variables ont été raffinées, et que notamment l'initialisation de *ptr_r1*, raffinement de *ptr*, est plus précise (la substitution "devient élément de" est remplacé par une substitution d'affectation "devient égal"). Ce raffinement est construit par affaiblissement des préconditions des opérations *store* et *value* : le test sur *i : 1..maxsize* est transféré de la substitution précondition à une substitution conditionnelle, ce qui peut permettre de gérer les cas d'exceptions (*ELSE v := 0 /* cas i /: 1..maxsize */*).

```
REFINEMENT Array_r1(maxsize)
    REFINES Array
    VARIABLES contents_r1, ptr_r1
    INVARIANT       contents_r1 : 1..maxsize --> NAT & ptr_r1 : 1..maxsize &
                    contents_r1 = contents & ptr_r1 = ptr /* relation de raffinement */
    INITIALISATIONcontents_r1 := %i.(i : NAT & i : 1..maxsize | 0) ||
                    ptr_r1 := 1 /* choix de la valeur d'initialisation */
    OPERATIONS
        store (i, v) =
        PRE v : NAT & i : NAT
        THEN  IF i : 1..maxsize
              THEN  contents_r1(i) := v ; /* séquence ; */
                    ptr_r1 := i
              END
        END;

        v <-- value (i) =
        PRE i : NAT
        THEN  IF i : 1..maxsize
              THEN  v := contents_r1(i) ; /* séquence ; */
                    ptr_r1 := i
              ELSE v := 0 /* cas i /: 1..maxsize */
              END
        END
END /* Raffinement Array_r1 */
```

Figure III.45. : exemple de raffinement

3.3.3. Modules d'implémentation

Le dernier raffinement mène à l'implémentation. C'est la dernière étape avant la génération du code exécutable du système logiciel.

L'implémentation importe ("*IMPORTS*") des spécifications de machines abstraites existantes, qui sont soit implémentées dans le projet, soit déjà codées et présentes dans l'environnement (stockées dans les bibliothèques de machines réutilisables des environnements logiciels B, comme l'Atelier B ou le B Toolkit). Cependant, elle ne peut plus inclure ("*INCLUDES*") une autre machine abstraite.

Tout comme un raffinement ("*REFINEMENT*"), une implémentation a un en-tête particulier ("*IMPLEMENTATION*"), et elle doit posséder une clause "*REFINES*" (par exemple : *IMPLEMENTATION I REFINES N … END*), ainsi qu'une relation d'implémentation (dans l'invariant), qui donne la correspondance entre les niveaux abstrait/concret pour les variables à implémenter en terme de variables de machines importées et de variables concrètes.

Les **substitutions généralisées** non déterministes sont interdites dans les implémentations. De ce fait, seules les substitutions suivantes peuvent être utilisées dans une implémentation :

- substitution bloc ("BEGIN"),
- substitution non-opération ou identité ("skip"),
- substitution condition par cas ("CASE"),
- substitution conditionnelle ("IF"),
- substitution appel d'opération (" <-- "),
- substitution "devient égal" (" := "),
- substitution assertion ("ASSERT"),
- substitution variable locale ("VAR"),
- substitution séquencement (";"),
- substitution boucle "tant que" ("WHILE").

De plus, les constantes et variables abstraites ne sont plus autorisées non plus. Une clause "VALUES" vient s'ajouter pour définir la valeur des constantes et des ensembles pour lesquels la décision avait été différée jusque là.

La figure III.46 présente la **structure** de la définition d'une implémentation :

```
IMPLEMENTATION /* Déclaration du nom de l'implémentation */
    REFINES /* Déclaration du nom de la machine abstraite ou du raffinement à implémenter */

    SEES /* Déclaration des machines vues */
    IMPORTS /* Déclaration des machines importées */
    PROMOTES /* Promotion des opérations des machines importées qui sont utilisées sans
    modification pour implémenter des opérations de même signature du raffinement ou de la
    machine à implémenter */
    EXTENDS /* Déclaration des machines étendues */

    SETS
    /* Définition des ensembles de l'implémentation */
     [ CONCRETE_ ] CONSTANTS
    /* Déclaration des identificateurs des constantes concrètes*/
    PROPERTIES
    /* Déclaration des propriétés des ensembles et constantes déclarés, avec typage et expression
    de valuation des constantes : définition des constantes, correspondance entre les constantes et
    ensembles locaux/importés */
```

```
    VALUES
    /* Valuation des constantes et des ensembles différés */

    CONCRETE_VARIABLES
    /* Déclaration des identificateurs des variables concrètes*/
    INVARIANT
    /* Déclaration des propriétés logiques invariantes des variables déclarées, avec typage des
    variables, et déclaration de la relation d'implémentation avec la machine à implémenter et
    les machines importées */
    ASSERTIONS
    /* Déclaration des définitions de nouvelles assertions logiques de l'implémentation */

    INITIALISATION
    /* Déclaration de la substitution généralisée initialisant les variables de l'implémentation */

    OPERATIONS
    /* Déclaration des définitions des opérations de l'implémentation : implémentation
    d'opérations de la machine à implémenter en terme d'opérations de machines vues ("SEES")
    et importées ("IMPORTS") */
END /* Fin de la définition de l'implémentation */
```

Figure III.46. : structure d'une implémentation de la méthode B

La figure III.47 propose une implémentation *Minimal_r2* du raffinement *Minimal_r1* de la figure III.43. Cet exemple importe une machine *MVar* de la librairie du B Toolkit, qui permet d'exporter la gestion de la variable *y* de *Minimal_r1* (la variable *x* de *Minimal*). Ainsi, nos données sont stockées dans la variable *MVar_V* de la machine importée, et elles sont maintenant manipulées via des opérations de cette machine.

```
IMPLEMENTATION Minimal_r2
    REFINES Minimal_r1
    IMPORTS MVar(NAT) /* Machine de lib */
    INVARIANT y = MVar_V /* variable MVar_V */
    INITIALISATION
    MVar_STO_VAR(0)
    OPERATIONS up(v) =
    BEGIN
    MVar_ADD_VAR(v)
    END
    END /* Implémentation Minimal_r2 */
```

Figure III.47. : exemple d'implémentation

3.4. Conclusion

Une spécification de la méthode B est basée sur une hiérarchie de :

- machines abstraites, qui s'incluent, s'utilisent ou se voient,
- raffinements, qui raffinent des machines abstraites ou d'autres raffinements, incluent d'autres machines abstraites, en utilisent ou en voient,
- implémentations, qui raffinent des machines abstraites ou des raffinements, importent ou voient d'autres machines abstraites implémentées.

Chacun des modules propose des opérations pour manipuler les données encapsulées. Néanmoins, elles ne servent que d'interface, et les opérations d'une même machine ne peuvent pas s'appeler entre elles (mais elles peuvent se servir des opérations promues).

Un avantage important de la méthode B est la possibilité de vérifier la cohérence du projet grâce aux obligations de preuve qui peuvent être déduites de la spécification, essentiellement à partir des invariants. Ces preuves permettent notamment de vérifier qu'un raffinement conserve bien le comportement de son abstraction.

Ce mécanisme d'obligations de preuve prend toute son importance grâce à un outil de développement comme l'Atelier B, qui permet d'automatiser une grande partie du travail de vérification des machines de la spécification, de générer les obligations de preuve et d'essayer d'en faire la preuve.

Ce travail de thèse ne cherche pas à proposer une nouvelle approche de développement en B. Il n'est pas non plus question de s'appuyer sur le mécanisme de raffinement ou d'obligations de preuve de la méthode B. En fait, nous nous contentons d'écrire une spécification formelle sous forme de machines abstraites, et de savoir que la suite du développement, jusqu'à la génération du code, peut être assuré.

Nous n'utilisons donc qu'une partie du langage de spécification, en conservant toutefois à l'esprit que les autres aspects de la méthode B pourront compléter notre approche en aval du cycle de vie.

4. La logique de réécriture

4.1. Introduction

La logique de réécriture ("rewriting logic" ou RWL), telle que présentée dans [Clavel et al. 1999], [Martí-Oliet et Meseguer 1993], [Martí-Oliet et Meseguer 1996] et [Meseguer 1998], est une logique de changement concurrent, qui peut traiter de façon naturelle l'état du système, et des calculs concurrents non déterministes de haut niveau. Elle a de bonnes propriétés comme cadre de travail sémantique flexible et général, qui apporte une sémantique à un large ensemble de langages et de modèles à concurrence. En particulier, elle supporte très bien des calculs concurrents orientés-objet. En outre, la logique de réécriture est réflexive.

Une spécification en logique de réécriture peut être interprétée en termes de calcul ou en termes de logique. Les mêmes raisons qui en font un bon cadre de travail sémantique au niveau calculatoire, en font aussi un bon cadre de travail logique, c'est-à-dire une métalogique dans laquelle bien d'autres logiques peuvent être représentées et implémentées de façon naturelle. Par conséquent, certaines des applications les plus intéressantes sont celles de métalangages, dans lesquelles sont créés des environnements exécutables pour des prouveurs de théorèmes dans différentes logiques, des langages ou des modèles de calcul.

Un outil basé sur la logique de réécriture peut constituer à la fois un langage et un système de haute performance pour des calculs de logique équationnelle comme de logique de réécriture, avec un éventail d'applications important.

La suite de cette partie est organisée en commençant par la présentation des différents éléments de base de la logique de réécriture. Nous nous attarderons ensuite sur les théories de la logique de réécriture. Finalement, nous évoquerons les diverses applications que nous pouvons en faire dans le cadre de notre travail.

Pour d'avantages de précisions sur la réécriture de termes (dans un cadre plus général que la seule logique de réécriture), le lecteur peut se référer à [Baader et Nipkow 1998] ou [Ohlebusch 2002].

4.2. Eléments de base logiques

La logique de réécriture permet d'écrire des spécifications orientées propriétés de systèmes logiciels, sous la forme de théories qui peuvent comprendre une partie décrivant les changements d'états (par réécriture de termes). Selon qu'elle comprend ou non ces informations de réécriture, une théorie de la logique de réécriture sera considérée comme un "module principal" représentant un système, ou bien plutôt comme une "bibliothèque".

Les **modules fonctionnels** sont des théories de la logique équationnelle d'appartenance ("membership equational logic"), une logique de Horn dont les phrases atomiques sont des égalités $t = t'$ et des assertions d'appartenance de la forme $t : s$, qui établit qu'un terme t est de sorte s. Une telle logique étend la logique équationnelle à sortes ordonnées ("order-sorted equational logic"), et supporte les sortes, les relations de sous-sortes, la surcharge polymorphe des opérateurs, sur la base de sous-sortes, et la définition de fonctions partielles avec des domaines définis par des équations. Les modules fonctionnels sont supposés avoir la propriété de Church-Rosser.

En fait, la logique équationnelle d'appartenance ("membership equational logic") est une sous-logique de la logique de réécriture : une **théorie de réécriture** est une paire (T, R) où T est une **théorie** équationnelle d'appartenance et R est une collection de **règles de réécriture**, avec des labels et éventuellement des conditions, portant sur des termes de la signature de T.

Les **modules systèmes** sont de telles théories de réécriture. Les règles de réécriture $r : t \rightarrow t'$ de R ne sont pas des équations. D'un point de vue calculatoire, elles sont interprétées comme des règles de transitions locales dans un système qui peut être concurrentiel. Au niveau logique, elles sont interprétées comme des règles d'inférence dans un système logique. Ainsi, comme il a été dit précédemment, la logique de réécriture constitue à la fois un **cadre de travail sémantique** général pour spécifier des systèmes et des langages concurrents, en même temps qu'un **cadre de travail logique** général pour représenter et exécuter différentes logiques.

La réécriture dans (T, R) a lieu modulo les axiomes équationnels de T. Par exemple, la réécriture peut se faire modulo la plupart des différentes combinaisons d'axiomes d'associativité, de commutativité, d'identité et d'idempotence. Les règles de R n'ont pas besoin d'avoir la propriété de Church-Rosser ou de terminer. Plusieurs chemins de réécriture différents sont alors possibles ; il en résulte que le choix de **stratégies** appropriées est crucial pour "exécuter" des théories de réécriture.

De telles stratégies ne forment pas pour autant une partie en dehors de la logique de ce langage. Au contraire, ce sont des stratégies internes définies par des théories de réécriture au méta-niveau. En effet, cela est rendu possible parce que la logique de réécriture est **réflexive** : il existe une théorie universelle U qui peut représenter n'importe quelle théorie de réécriture T présentée de façon finie (incluant U elle-même) et tous termes t et t' de T, comme des termes $\underline{T}$, $\underline{t}$ et $\underline{t'}$ dans U, de telle sorte à avoir l'équivalence suivante :

$T \vdash t \rightarrow t' \Leftrightarrow U \vdash \langle \underline{T}, \underline{t} \rangle \rightarrow \langle \underline{T}, \underline{t'} \rangle$.

Puisque U peut se représenter elle-même, une "tour réflexive" peut alors être obtenue, avec un nombre arbitraire de niveaux de réflexivité. Celle-ci rend possibles, non seulement la définition déclarative et l'exécution de stratégies de réécriture définies par l'utilisateur, mais aussi bien d'autres applications, dont par exemple une algèbre de modules extensible, d'opérations sur les modules paramétrées, qui serait définie et exécutée à l'intérieur de la logique.

Cette extension par réflexivité permet d'étendre les fonctionnalités de base du langage : la hiérarchie de modules constituée de modules fonctionnels et systèmes (non paramétrés) peuvent être étendu en une algèbre de modules, de modules paramétrés, de vues, d'expressions sur les modules ou encore de modules orientés-objet disposant d'une syntaxe adéquate pour des applications orientées-objet.

4.3. Théories de la logique de réécriture

Une théorie de la logique de réécriture peut être décrite comme un quadruplet : $R = (\Sigma, E, L, R)$, où :

- Σ est un alphabet rangé de symboles de fonctions,
- E est un ensemble de Σ-équations,
- L est un ensemble de labels,
- R est un ensemble de paires.

Chaque paire de R est composée d'un label (une étiquette) et d'un couple de classes d'E-équivalence de termes (c'est-à-dire les classes d'équivalence des termes modulo E) : $R \subseteq L \times T_{\Sigma, E}(X)^2$.

$X = \{x_1, \ldots, x_n, \ldots\}$ représente un ensemble de variables de cardinalité infinie.

Les éléments de R sont appelés des règles de réécriture : un couple labellisé $(r ([t], [t'])$ est noté $r : [t] \rightarrow [t']$.

La représentation d'une théorie de la logique de réécriture peut se faire, d'une part, sous la forme d'un module fonctionnel. Un module fonctionnel représente essentiellement des types de données, et les fonctions qui s' y appliquent, au moyen de théories équationnelles. Comme évoqué précédemment, leurs équations doivent avoir la propriété de Church-Rosser et doivent terminer. Dans ce cas, chaque étape de réécriture se fait par un remplacement d'égal à égal, jusqu'à ce qu'une valeur équivalente à celle de départ, et complètement évaluée, soit obtenue.

La figure III.48 présente un exemple de module fonctionnel ("*fmod … endfm*"), qui introduit trois types, ou sortes ("*sorts*"), simultanément : *Nat* pourrait représenter l'ensemble des entiers naturels, *NzNat* les entiers naturels non nuls et *Nat3* l'ensemble des entiers modulo 3. Pour cela, *NzNat* est déclaré comme une sous-sorte ("*subsort*") de *Nat*. Le constructeur *zero* permet d'obtenir l'élément de base des *Nat*, et l'opération ("*op*") *s* servira à construire les autres, tout en ne générant que des *NzNat* ; l'opération *p*, au contraire, est l'opération inverse et pourrait avoir comme résultat un *Nat* n'appartenant pas à *NzNat* (c'est-à-dire *zero*). L'addition + est définie sur l'ensemble *Nat* tout entier, mais il est utile de préciser que le résultat de l'addition de deux éléments de *NzNat* reste dans *NzNat*. Les trois premières équations ("*eq*") décrivent la façon dont interagissent les opérations, sur des variables ("*vars*") de sorte *Nat* ; nous retrouvons les propriétés usuelles des entiers, comme le prédécesseur du successeur d'un nombre est ce nombre, l'addition de zéro à un nombre ne le change pas, et l'ajout du successeur d'un nombre à un autre est équivalent à demander le successeur de l'addition de ces nombres. En outre, les trois constantes *0*, *1* et *2* sont les éléments de base de *Nat3* (la spécification de Nat3 aurait pu faire l'objet d'un autre module fonctionnel). La seule autre opération définie sur cet ensemble est à nouveau l'addition +, mais cette fois-ci elle est déclarée comme étant commutative ("*[comm]*") – d'autres propriétés auraient pu être ajoutées, comme l'associativité ou l'importance relative de l'opérateur par rapport aux autres. Enfin, quatre équations, requérant une unique variable *N3*, achève la spécification du type de données *Nat3*.

Chaque équation participe à la définition de classes d'équivalence de termes. Dans l'exemple de la figure III.48, chacune peut être caractérisée par la valeur de l'entier équivalent.

```
fmod NUMBERS is
    sorts Nat NzNat Nat3 .
    subsort NzNat < Nat .
    op zero : -> Nat .
    op s_ : Nat -> NzNat .
    op p_ : NzNat -> Nat .
    op _+_ : Nat Nat -> Nat .
    op _+_ : NzNat NzNat -> NzNat .
    ops 0 1 2 : -> Nat3 .
    op _+_ : Nat3 Nat3 -> Nat3 [comm] .
```

```
    vars N M : Nat .
    var N3 : Nat3 .
    eq p s N = N .
    eq N + zero = N .
    eq N + s M = s (N + M) .
    eq (N3 + 0) = N3 .
    eq 1 + 1 = 2 .
    eq 1 + 2 = 0 .
    eq 2 + 2 = 1 .
endfm
```

Figure III.48. : exemple de module fonctionnel

D'autre part, un ensemble de règles de réécriture n'a pas besoin de terminer : à la différence des équations, elles n'indiquent pas des équivalences entre termes connus, mais elles spécifient des règles de production de nouveaux termes, à partir de termes existants. Elles sont introduites dans les modules systèmes. Un module système peut contenir :

- une ou plusieurs théories en paramètres (par exemple avec le mot-clé "*protecting*") ;
- un ensemble d'équations, qui peut être divisé en :
  - un ensemble A d'axiomes,
  - un ensemble E d'équations, qui doivent avoir la propriété de Church-Rosser, terminer et permettre de faire diminuer les sortes, modulo les axiomes de A (cette partie équationnelle est équivalente à un module fonctionnel) ;
- un ensemble de règles, qui doivent être cohérentes avec les équations de E, modulo les axiomes de A.

Un exemple de module système ("*mod … endm*") est présenté dans la figure III.49. Ce module système *ND-INT* – pour entier non déterministe – réutilise (avec "*protecting*"), ou étend, la théorie prédéfinie *MACHINE-INT* en définissant une sorte *NdInt* (grâce à "*sort*") qui l'encapsule (avec "*subsort*"). Une seule opération *?* est ajoutée, qui sera spécifique aux nouveaux éléments ajoutés ; de plus, elle est définie comme étant commutative et associative (c'est le sens de "*[assoc comm]*"). La seule équation, portant sur le nouvel opérateur, indique que l'application d'une même valeur de *MachineInt*, pour les deux opérandes de *?*, a comme résultat cette même valeur (idempotence). Enfin, la règle de réécriture ("*rl*") suivante, appelée *choice*, permet de réécrire un terme de *NdInt*, avec *?*, en choisissant de ne conserver que la partie de gauche, s'il s'agit d'un terme de la sous-sorte *MachineInt*.

```
mod ND-INT is
    protecting MACHINE-INT.
    sort NdInt .
    subsort MachineInt < NdInt .
    op _?_ : NdInt NdInt -> NdInt [assoc comm] .
    var N : MachineInt .
    var ND : NdInt .
    eq N ? N = N .
    rl [choice] : N ? ND => N .
endm
```

Figure III.49. : exemple de module système

4.4. Applications

Toutes les applications typiques de la programmation équationnelle et de la spécification algébrique peuvent être supportées de façon adéquate et efficace par le sous-langage des modules fonctionnels. En fait, la logique équationnelle, c'est-à-dire la logique équationnelle d'appartenance ("membership equational logic"), a un pouvoir d'expression suffisant pour offrir d'aussi bons avantages qu'un cadre de travail logique, pour un très large éventail de langages de spécification algébrique, basés sur des formalismes de logique équationnelle à la fois totale et partielle – et elle peut être facilement implémentée.

Néanmoins, beaucoup d'autres applications portent au-delà de la logique équationnelle. En effet, les modules systèmes supportent les applications générales de la logique de réécriture. De plus, le domaine important de la spécification et du prototypage de systèmes concurrents et distribués basés sur les objets peut être supporté par des modules orientés-objet.

Par ailleurs, la réflexivité rend possibles de nombreuses nouvelles applications de métaprogrammation et de métalangage. En particulier, elle est très appréciable dans toutes celles qui utilisent la logique de réécriture comme cadre de travail logique et sémantique.

En outre, les langages de conception orientés-objet, les langages de description d'architecture et les langages pour composants distribués trouvent une sémantique naturelle dans la logique de réécriture.

4.5. Conclusion

La logique de réécriture peut être utilisée comme un cadre de travail logique, dans lequel serait décrite la façon dont chaque transformation élémentaire est effectuée, fournissant ainsi un support pour exprimer la syntaxe de notre approche de raffinement. De plus, elle peut servir de cadre de travail sémantique, afin de prouver que le comportement de notre architecture abstraite est respecté, offrant du même coup un support à l'expression de la sémantique de nos transformations.

Les chapitres suivants présenteront plus précisément l'utilisation que nous pouvons faire de la logique de réécriture dans le cadre de notre travail. D'un côté, elle nous fournit une syntaxe. En effet, nous pouvons décrire les grammaires des langages raffinés sous la forme de modules fonctionnels. En outre, les différentes transformations à leur appliquer, pour passer de l'un à l'autre, peuvent être représentées par des modules systèmes : les traductions de vocabulaire sont traitées via des règles de réécriture, et la propagation des modifications à l'intérieur des éléments architecturaux en utilisant des équations.

La figure III.50 propose un exemple de règle, appelée ici *rule1*, (cf. chapitre 5 pour une présentation plus détaillée). Les identificateurs en italique désignent des variables, servant à identifier soit des éléments bien précis que l'on veut réécrire, soit leur contexte. Ainsi, la logique de réécriture nous laisse une liberté assez importante dans l'écriture des règles de réécriture, pour lesquelles les contraintes résident dans la structure épurée (*rl […] :* … => … .) et dans l'obligation de n'utiliser, dans la deuxième partie d'une règle, que des variables introduites dans sa première partie.

rl [rule1**] :**

ARCHI

compose *COMPOSITION1* {
COMPONENT1 : *COMPONENTTYPE1* [*PORT1* : *PORTTYPE1* [*PORTTYPEPARAM*]] ||
CONNECTOR1 : *CONNECTORTYPE1* [*ROLE1* : *ROLETYPE1* [*ROLETYPEPARAM*] ,
LTPORTS1] || *AEDECL*
where attach *COMPONENT1* @ *PORT1* to *CONNECTOR1* @ *ROLE1* , *LADECL* }

=>

ARCHI [*COMPONENTTYPE1* @ *PORT1* : *PORTTYPE1*
[*PORTTYPEPARAM*] / *CONNECTORTYPE1* @ *ROLE1* : *ROLETYPE1* [
ROLETYPEPARAM]]

compose *COMPOSITION1* {
COMPONENT1 : *COMPONENTTYPE1* [*PORT1* : *PORTTYPE1* [*PORTTYPEPARAM*]] ||
CONNECTOR1 : *CONNECTORTYPE1* [*PORT1* : *PORTTYPE1* [*PORTTYPEPARAM*] ,
LTPORTS1] || *AEDECL*
where *LADECL* } **.**

Figure III.50. : exemple de règle de réécriture

D'un autre côté, la logique de réécriture est également capable de donner une sémantique au raffinement. Cela peut concerner aussi bien la description de l'interprétation des comportements d'une architecture en π-SPACE dans l'espace des processus, que la vérification de l'équivalence observationnelle pour la théorie spécifiant la transformation uniquement (plutôt que de le refaire pour chaque système raffiné à chaque fois).

5. Conclusion sur les formalismes utilisés

Les trois différents formalismes présentés dans ce chapitre ne sont pas les objets de notre travail, mais ils constituent les moyens formels pour la mise en œuvre de notre approche.

D'un côté, π-SPACE est un langage de description d'architecture logicielle, qui dispose d'un pouvoir expressif très important, notamment pour la spécification d'architectures dynamiques.

D'un autre côté, la méthode B nous offre la possibilité de poursuivre un développement formel, garanti par des obligations de preuves, jusqu'à la génération du code de l'application, grâce à un outil commercial.

Enfin, les chapitres suivants présenteront plus en détail la façon dont nous nous servons de la logique de réécriture pour spécifier et traiter le raffinement.

Chapitre 4 : Patrons de transformation par réécriture

Chapitre 4 : Patrons de transformation par réécriture

Les chapitres précédents nous ont permis d'introduire des méthodes du génie logiciel existantes afin de présenter, dans un premier temps, les différents types de relations de raffinement qui peuvent être rencontrés dans le domaine, puis les différents formalismes qui nous intéressent dans le cadre de notre travail.

Après une rapide introduction, nous entamerons cette partie par la présentation de la relation de raffinement que nous voulons couvrir. Nous poursuivrons en rappelant comment nous proposons d'utiliser les différents formalismes choisis. Dans la suite, nous nous intéresserons au langage de description de raffinement d'architecture dont nous avons besoin pour la formalisation du raffinement, qui se trouve au cœur de notre proposition et qui vient compléter le processus de développement. Nous montrerons notamment comment nous structurons notre approche β-SPACE sur la base de patrons mais aussi des formalismes supports, avec une présentation de la formalisation en logique de réécriture. Les différentes étapes de raffinement seront ensuite détaillées : le premier et le deuxième patrons nous amèneront ainsi à introduire la notion de forme architecturale canonique, qui servira de base de départ pour le troisième patron. Après cette présentation théorique de notre approche, nous détaillerons son fonctionnement sur une étude de cas. Nous conduirons le développement de cet exemple à partir de la description architecturale en π-SPACE, en lui appliquant successivement les trois patrons de transformation, en détaillant quelques règles de réécriture utilisées (cf. annexe B pour l'ensemble des patrons), ainsi qu'en donnant la spécification du système à l'issue de chaque étape de transformation.

1. Introduction

A l'image du domaine plus vaste du génie logiciel, les architectures logicielles sont l'objet de travaux très différents. Ainsi, certains langages de description d'architectures logicielles classiques permettent de représenter des comportements d'architectures, dynamiques ou non. Cependant, ils ne fournissent pas de mécanisme de transformation pour passer d'une spécification du système à une implémentation. D'autres se concentrent essentiellement sur la structure de l'architecture et son raffinement, mais sans gérer tous les aspects intéressants d'une description architecturale, notamment la sémantique du comportement [Bolusset et al. 2000b].

Par ailleurs, le développement complet d'un logiciel, tout au long de son cycle de vie, met en jeu des concepts et des niveaux d'abstraction très éloignés les uns des autres, qu'il n'est pas possible de représenter de manière assez fine avec un formalisme unique. En effet, certains langages de description d'architecture sont capables de spécifier convenablement le système et ses propriétés à un niveau d'abstraction donné, mais ils ne sont pas adaptés pour gérer les transitions entre les différents niveaux, ni la génération finale du code exécutable de l'application. D'autre part, il existe des méthodes formelles supportant le développement d'un système logiciel depuis sa spécification formelle jusqu'à son implémentation dans un langage de programmation.

De façon à pallier ces différentes limitations, nous proposons dans cette thèse d'utiliser un langage pour la description du raffinement. L'intérêt est multiple puisqu'il s'agit à la fois de compléter des formalismes existants sans avoir à les réinventer, et de permettre la vérification ou le suivi des transformations, de la spécification initiale jusqu'à l'obtention du résultat.

La solution retenue est la définition de patrons génériques (cf. chapitre 2, paragraphe 7) pour transformer une description architecturale, écrite en π-SPACE en amont d'un développement formel, en une spécification formelle en machines abstraites de la méthode B.

2. Différents types de raffinement

En génie logiciel, le terme "raffinement" est employé pour désigner des choses différentes selon les approches considérées, comme illustré dans le chapitre 2. Quoi qu'il en soit, toutes ces utilisations reposent sur la notion d'un ajout de détails à une description. De façon intuitive, une description "plus raffinée" est plus détaillée ou "meilleure" dans un certain sens, que son abstraction. Néanmoins, pour pouvoir parler de relation dans le raffinement, il faut conserver un lien entre ce qu'elle relie, notamment par le biais d'aspects communs, c'est-à-dire la préservation de propriétés – au sens large – lors du changement de niveau d'abstraction : quel que soit le contexte dans lequel la spécification abstraite est plongée, son raffinement ou implémentation doit pouvoir y être plongé sans que l'environnement ne s'en rende compte.

Le raffinement peut ainsi prendre des aspects très variés ; entre autres, nous nous intéressons à deux classes distinctes bien spécifiques, que nous appelons dans le cadre de notre travail les raffinements architecturaux horizontaux et verticaux [Bolusset et Oquendo 2002] (cf. chapitre 1, paragraphe 3 et figure IV.1).

Le *raffinement horizontal*, d'une part, concerne de façon générale la décomposition d'architecture : il induit l'ajout de détails à une spécification, tout en conservant le même langage de description d'architecture. Le vocabulaire de base ne change pas, mais l'interprétation sémantique du comportement de l'architecture change : elle est plus détaillée, du fait des différents points développés.

Le *raffinement vertical*, d'autre part, rapproche de l'implémentation en passant d'un premier langage de description d'architecture à un deuxième – qui peut éventuellement être un langage de programmation. Globalement, il permet d'introduire des détails nécessaires à l'obtention d'un système exécutable, sans que pour autant l'interprétation sémantique du comportement du résutat diffère de l'interprétation sémantique du comportement de la spécification de départ. Cela revient souvent à ajouter des détails qui n'auraient pas pu être spécifiés avec le langage de description d'architecture de plus haut niveau.

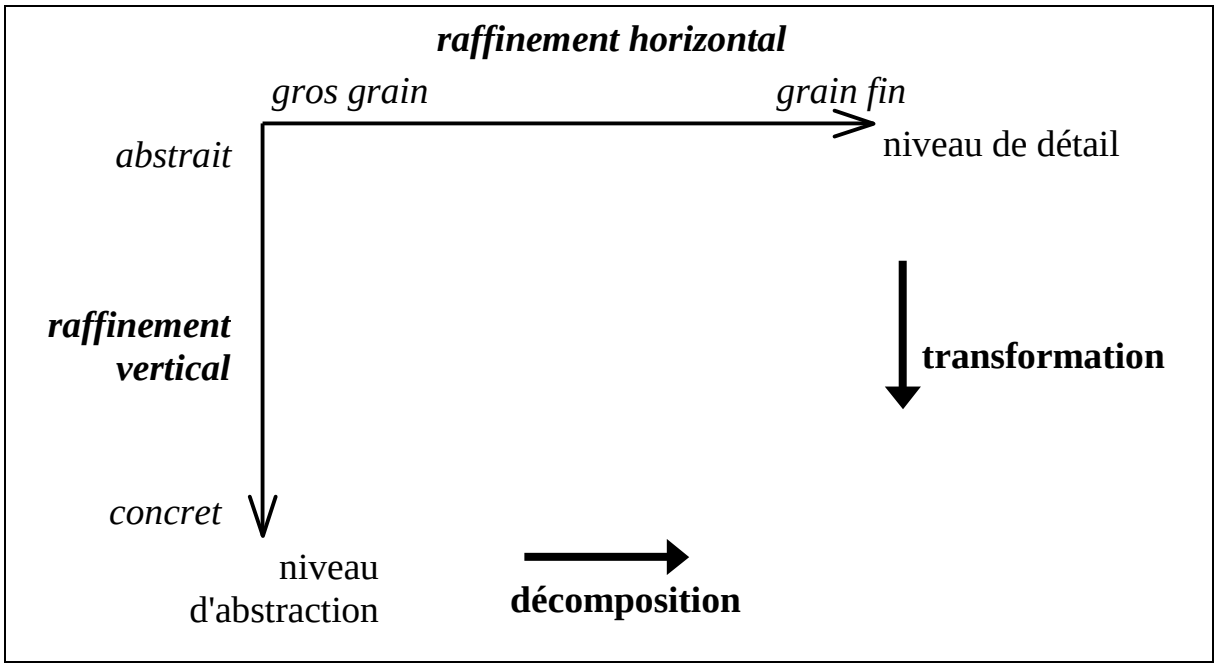

Figure IV.1. : illustration des raffinements architecturaux horizontaux et verticaux

Par ailleurs, des différences peuvent également être mises à jour dans la façon de raffiner elle-même (cf. chapitre 2). SADL, par exemple, remplace la totalité de l'architecture lorsqu'il effectue un de ses raffinements (mais il ne s'intéresse pas réellement aux comportements), alors que dans le

cas de la méthode B un lien est créé : la machine abstraite est conservée telle qu'elle, et elle est étendue à l'aide d'une spécification différentielle.

La décomposition, ou raffinement horizontal, induit en fait une modification de la spécification, sans qu'il n'y ait de changement de niveau d'abstraction. Etant donné qu'il n'est pas possible de vérifier quelque chose qui n'a pas encore été écrit, il ne peut pas y avoir de contrainte pour établir ou gérer un lien entre les deux spécifications, sur la base des parties raffinées. Néanmoins, le morceau de l'architecture abstraite qui est développé est clairement identifié. Par conséquent, le raffinement horizontal pourrait être vu comme une substitution, c'est-à-dire que l'architecture originale est remplacée explicitement par la nouvelle architecture plus détaillée.

Le raffinement vertical, quant à lui, ne devrait pas être traité de la même manière puisqu'il consiste en une transformation de la spécification : seules des décisions d'implémentation sont opérées, liées à la plate-forme cible. Une représentation plus appropriée serait alors une sorte de lien entre deux spécifications, comme dans le cas de la méthode B. La sémantique originale de l'architecture abstraite n'est pas remise en question, mais les détails ajoutés pour faciliter l'implémentation peuvent la rendre plus déterministe à un niveau plus concret. Les propriétés que nous attendons d'un raffinement vertical concernent la similitude des comportements des architectures abstraites et concrètes mises en relation : l'équivalence faible, ou équivalence observationnelle [Milner 1999], nous garantira la conservation, au niveau d'abstraction concret, d'un comportement compris dans celui du niveau abstrait.

Comme cadre pour l'interprétation des comportements des architectures, nous avons choisi de façon tout à fait naturelle l'algèbre de processus π-calcul, qui sert de base sémantique formelle à π-SPACE, le langage de description abstraite de notre approche (cf. chapitre précédent).

3. Formalismes utilisés

Certains langages de description d'architecture disposent d'un système de décomposition, et d'autres d'un système de raffinement vertical. Malheureusement, la distinction n'est généralement pas faite entre les constructions pour raffiner verticalement ou horizontalement (pour s'en persuader, il suffit d'examiner les nombreuses approches, comme par exemple [Sa et al. 1993], [Fahmy et Holt 2000], [Honsell et al. 2000], [Sannella 2000], …). Si ces constructions étaient distinguées, alors il serait plus facile de différencier ce qui relève de la décomposition de ce qui concerne la transformation. Dans pareil contexte, il deviendrait plus aisé de voir comment faire pour ne s'intéresser au raffinement que de tout ou partie de la spécification, et comment vérifier les propriétés du système.

Séparer les constructions de raffinement vertical et horizontal permet, en effet, de distinguer clairement celles qui prennent en charge une partie de la décomposition de celles qui induisent un changement de niveau d'abstraction. Les langages classiques de description d'architecture logicielle supportent habituellement la décomposition, mais fort peu le raffinement vertical. En fait, la décomposition prend souvent part au processus de développement d'une architecture dans ces langages, alors que le raffinement vertical, avec son changement de vocabulaire entre les niveaux d'abstraction, n'apparaît pas naturellement dans ces approches. SADL, par exemple, parle de raffinement, d'une façon distincte de la décomposition, en proposant de passer d'un style architectural à un autre (cf. chapitre 2) ; cependant, il ne s'agit pas là véritablement d'un raffinement vertical puisque ce langage ne permet pas de développer le comportement de l'architecture (et donc de vérifier sa conservation).

Le raffinement d'architecture, quant à lui, a pour objectif de changer une architecture abstraite en une autre concrète. De ce fait, les langages de description des architectures abstraites et concrètes peuvent être différents : de façon tout à fait analogue, dans SADL, le vocabulaire du style abstrait est remplacé par le vocabulaire du style concret. Dans ce cadre, les constructions du langage de description de raffinement servent à éliminer, petit à petit, le vocabulaire du langage de description d'architecture abstrait de départ, et à introduire celui du langage concret cible qui facilitera la

génération du code de l'application, tout en respectant le comportement de la spécification originale.

Or, décrire le raffinement vertical de façon spécifique implique l'utilisation d'un vocabulaire de constructions appropriées. En outre, cette séparation entre les différents raffinements peut être obtenue en utilisant d'un côté les mécanismes d'un langage de description d'architecture expressif pour les aspects concernant la décomposition, et de l'autre un langage dédié distinct (mais pas complètement indépendant pour autant) afin de gérer le raffinement vertical. La définition d'un tel langage dédié spécifiquement aux descriptions de raffinement vertical repose d'une part sur un principe fondamental de traduction entre les constructions et primitives d'assemblage respectives des langages de description d'architecture abstraits et concrets, et d'autre part sur la conservation des propriétés essentielles du système.

Le choix du formalisme abstrait (cf. chapitre 1, paragraphe 4) pourrait se porter sur un langage de description d'architecture avec un pouvoir d'expression assez élevé pour identifier les propriétés du système au cours des premières phases du développement. π-SPACE [Chaudet et Oquendo 2000], par exemple, est un langage formel de description d'architecture ; contrairement à la plupart des autres langages de spécification, il permet de représenter des architectures logicielles dynamiques à un niveau abstrait et il possède en même temps des mécanismes de vérification pour des propriétés critiques du système modélisé [Bolusset et al. 2000a].

Par ailleurs, le raffinement doit conduire à la génération du code. Ainsi, le langage concret pourrait être aussi bien un langage d'implémentation ou un langage de spécification qui puisse automatiser (au moins partiellement) la génération du code de l'application. Il apparaît alors comme préférable de choisir un formalisme permettant à la fois de générer facilement le code de l'application et de vérifier les propriétés de la représentation architecturale concrète. La méthode B semble particulièrement adaptée dans ce cadre.

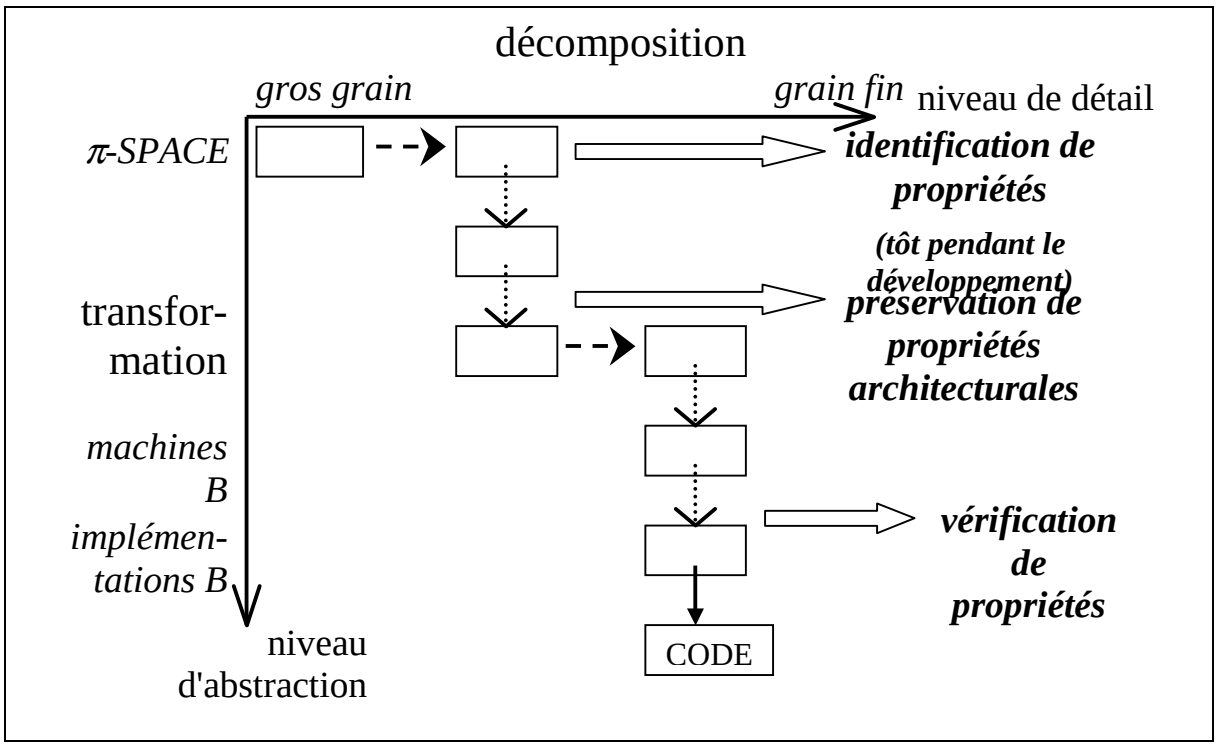

Figure IV.2. : processus de raffinement proposé

L'utilisation de patrons de transformation se présente tout naturellement comme un moyen efficace pour dérouler un processus incrémental et réutilisable depuis des descriptions abstraites d'architectures logicielles jusqu'à des spécifications concrètes (cf. chapitre 1, paragraphe 5). En effet, un tel mécanisme permet, entre autres choses, de vérifier uniquement que les patrons préservent les propriétés architecturales, plutôt que d'avoir à le faire à chaque fois qu'une étape de

raffinement est franchie. Il s'ensuit donc que la syntaxe de transformation doit avoir une base sémantique solide, c'est-à-dire que le formalisme doit permettre à la fois d'exprimer des modifications génériques de spécifications et d'étudier la conservation des propriétés sémantiques entre les descriptions abstraites et concrètes.

La logique de réécriture [Meseguer 1998] (présentée dans le chapitre précédent) répond tout à fait à ces besoins : elle peut être utilisée indifféremment comme cadre logique, pour exprimer la façon avec laquelle chaque transformation élémentaire est effectuée, ou comme cadre sémantique, pour prouver que le comportement de l'architecture abstraite est respecté durant tout le processus de raffinement.

La figure IV.2 complète la figure IV.1 en lui adjoignant ces nouvelles informations.

4. Langage de description de raffinement d'architecture

L'approche β-SPACE proposée pour le traitement du raffinement vertical réside donc dans l'utilisation d'un langage dédié de description de raffinement, pour transformer les constructions de base d'un langage de description d'architecture expressif existant (π-SPACE), en éléments d'un formalisme "concret" basé sur des constructions de la méthode B. Cette dernière n'est pas familière avec les abstractions basées sur des composants, qui peuvent être rencontrées dans une description d'architecture logicielle. Aussi, opérer toutes les transformations en une seule fois serait une tâche difficile et la démarche, complètement dépendante des langages abstraits et concrets, s'avèrerait inadaptable à un cadre différent.

C'est pourquoi le raffinement vertical est opéré par une succession d'étapes plus simples. Chacun de ces pas de raffinement est défini par un patron de transformation spécifique (cf. chapitre 2, paragraphe 7). Les différents patrons sont alors composés pour obtenir le résultat escompté. Chaque patron est écrit de façon générique, de façon à pouvoir s'appliquer dans des contextes différents. Il est alors envisageable de modifier aisément certaines parties du raffinement, comme par exemple le langage de spécification concret, en changeant simplement certains patrons, sans toucher aux autres. Cette flexibilité autorise donc la réutilisation des différentes parties du processus de raffinement.

Ce découpage du raffinement vertical en patrons implique certaines contraintes sur le langage utilisé. En effet, il convient de se placer dans un cadre formel permettant d'opérer une succession d'opérations de transformation. D'ailleurs, les structures manipulées dépendent fortement du choix des langages de description d'architecture abstrait et concret.

La description architecturale du système en π-SPACE (cf. chapitre 3, paragraphe 2) tient lieu de spécification abstraite et possède une sémantique comportementale basée sur l'algèbre de processus du π-calcul [Milner et al. 1992]. Les différents processus stockés dans les composants et connecteurs indépendants doivent être transformés, tout en conservant le même comportement global pour un observateur, en machines abstraites B organisées de façon hiérarchique. Les bases sémantiques des deux langages sont très différentes, mais le raffinement vertical doit néanmoins garantir le passage de l'une à l'autre.

Chaque patron de transformation est conçu pour traiter des parties différentes de l'architecture originale. Le processus s'effectue ainsi pas à pas, par l'application successive de patrons de raffinement. Au fil de l'application de ces patrons, les constructions du formalisme "abstrait" seront, les unes après les autres, raffinées pour utiliser à leur place les constructions du formalisme concret. Il s'ensuit que les états intermédiaires de la transformation globale ne correspondront pas à des descriptions homogènes, que ce soit en π-SPACE ou en machines abstraites B.

Comme nous l'avons déjà constaté (notamment dans le chapitre 1, paragraphe 5), la logique de réécriture (présentée dans le chapitre 3 également) fournit un cadre approprié à l'expression de nos transformations. Elle nous permet d'écrire les différents patrons mais également de les mettre en

application. C'est cette logique que nous utilisons dans nos travaux comme langage de description de raffinement d'architecture.

4.1. Patrons de raffinement pour la transformation

Notre premier patron de transformation raffine la description de la composition de l'architecture et les connecteurs : il prend en compte les attachements entre composants et connecteurs décrits dans la spécification d'architecture originale, et substitue aux noms des ports de connecteurs concernés les noms de ports de composants correspondants.

Si la spécification en π-SPACE est supposée correcte (voir le chapitre précédent), alors les ports attachés (un port de composant avec un port de connecteur) sont cohérents et dans chaque connecteur, les ports sont cohérents avec le comportement. La première étape de transformation de β-SPACE concerne donc les informations d'attachement : les références aux ports de connecteurs sont remplacées par les références aux ports de composants correspondants, qui y étaient attachés.

En effet, les noms de ports sont uniques. Les éléments superflus, c'est-à-dire les attachements qui viennent d'être déroulés, sont ensuite supprimés. Les définitions des ports des connecteurs raffinés doivent cependant être conservés, au cas où ils soient utilisés dans la définition d'autres connecteurs ou composants.

Aucune modification de propriété n'est introduite par ce premier patron, qui ne fait que renommer des éléments et retirer les requêtes d'attachements devenues inutiles.

Cette première étape de raffinement, si elle peut paraître simple, utilise néanmoins 21 règles et 32 équations en logique de réécriture. Même si toutes ne sont pas nécessaires à la transformation d'une description d'architecture logicielle spécifique, elles constituent un patron qui pourra être appliqué à n'importe quelle architecture abstraite de système (un exemple sera développé ultérieurement dans ce chapitre). En outre, l'équivalence observationnelle peut être prouvée entre les spécifications mises en relation par ce patron de transformation : cette transformation n'affecte en effet ni la séquence ni la nature des actions à l'intérieur des comportements des composants et connecteurs.

Le patron de raffinement suivant effectue globalement des actions similaires, à l'intérieur des composants cette fois, en y éliminant les descriptions des ports, qui sont eux aussi supposés cohérents vis-à-vis des comportements. Ainsi, ces ports peuvent également être considérés comme superflus. Comme dans le cas précédent, les spécifications des ports des composants et les références aux types de ports ne sont plus nécessaires et sont donc supprimées de l'architecture. Néanmoins, certaines informations liées aux descriptions de types de ports méritent d'être conservées ; en effet, les différents canaux de communications, qui permettent la synchronisation et la transmission de messages entre les composants et les connecteurs, sont référencés dans les ports, mais sont aussi utilisés dans les comportements (les mécanismes de π-SPACE sont rappelés dans le chapitre 3).

62 règles et 28 équations sont nécessaires à la description de ce patron de transformation en logique de réécriture. Vu qu'il s'agit à nouveau de renommer pour pouvoir ôter un excès d'informations, tout en sauvegardant les informations sur les échanges de communications, les propriétés architecturales ne sont pas altérées.

Par ailleurs, les spécifications des comportements de composants et connecteurs en π-SPACE sont basées sur des déclarations de processus du π-calcul, qui peuvent faire appel les unes aux autres. De plus, les composants peuvent inclure des fonctionnalités internes. L'étape de raffinement suivante permet alors, entre autres choses, de transposer les déclarations des fonctionnalités internes et celles des comportements des composants et connecteurs dans le formalisme de la méthode B (présentée également dans le chapitre précédent).

Dans ce cadre, la représentation d'un composant architectural dans une spécification formelle B nécessite deux machines abstraites : la première décrit les différentes actions que le composant peut être amené à mettre en œuvre (opérations internes ou communication via les canaux), et la seconde établit les actions envisageables selon l'état du composant, c'est-à-dire leur contrôle.

Les composants communiquent entre eux via des connecteurs. Il s'ensuit qu'un même connecteur est requis par au moins deux composants à chaque fois, mais aussi qu'un composant peut solliciter le concours de plusieurs connecteurs. Or, en B, il n'est pas possible d'inclure plusieurs fois une même machine abstraite, représentant par exemple un composant, dans des machines de connecteurs différents. C'est pourquoi les descriptions des comportements des connecteurs sont toutes regroupées dans une unique machine abstraite qui contrôle et établit les liens entre les canaux des différents composants.

Le raffinement intégral des processus est terminé, via le même patron, en détaillant les spécifications des canaux de communications. En fait, chaque canal de communication peut être représenté par une paire d'opérations en B, qui accèdent une même variable : la première opération assure le changement de valeur et la seconde permet sa consultation. La portée d'un canal de communication en π-SPACE n'est pas limitée à une seule entité architecturale : il est partagé par au moins un composant et un connecteur. Leurs spécifications, tout comme celles des différents types de données utilisés, qui caractérisent les canaux de communications, auraient pu être stockées dans une partie séparée de la description architecturale, et utilisées à chaque fois qu'elles seront nécessaires dans la spécification B. Néanmoins, cette solution ne fut pas retenue car, en contraignant de la sorte la description du système, l'indépendance des composants vis-à-vis les uns des autres eût disparu. C'est pourquoi l'approche que nous proposons spécifie, au contraire, les types et les canaux (représentés chacun par une variable et deux opérations), dans la machine abstraite décrivant les actions du composant concerné. Ainsi, seule la machine abstraite des connecteurs est en charge du lien entre les composants, en s'occupant aussi de la correspondance entre les données échangées et leurs types.

Enfin, une machine abstraite supplémentaire est nécessaire pour rappeler que les composants et les connecteurs sont composés en parallèle dans la configuration de l'architecture de départ.

Par conséquent, nous structurons notre processus de raffinement autour de trois patrons de transformation pour le raffinement vertical. Les deux premiers servent respectivement à supprimer des descriptions d'attachements et des définitions de ports (séparées des composants et connecteurs qui doivent les utiliser). Le dernier patron est chargé de la traduction du contrôle architectural dans la méthode B.

4.2. Compléments sur les langages utilisés

L'objectif principal de notre travail est de faciliter le développement de logiciels, et plus particulièrement la phase de raffinement de l'architecture d'un système, pour mener d'une spécification abstraite à une spécification concrète, voire à une implémentation (cf. chapitre 1). Etant donné que ni la phase de spécification, ni celle de la programmation ne nous concernent au premier abord, il nous importe d'utiliser en amont et en aval de notre approche β-SPACE des formalismes les plus complets possibles, afin de nous préoccuper le moins possible à propos de ces tâches annexes.

L'approche de raffinement d'architecture logicielle proposée, β-SPACE, est ainsi contrainte par le choix de plusieurs langages (présentés dans le chapitre 1, paragraphe 4 et de façon plus exhaustive dans le chapitre 3). Tout d'abord, les descriptions d'architectures logicielles à traiter sont définies dans le formalisme abstrait choisi, π-SPACE. Ce langage permet de décrire des architectures logicielles dynamiques à un niveau abstrait. En effet, il permet de décrire des configurations architecturales composées de composants et de connecteurs. Chaque composant peut être défini comme le résultat de la combinaison d'une sous-architecture. D'autre part, la sémantique opérationnelle de ce langage est basée sur le π-calcul. Or, une des particularités essentielles de cette algèbre de processus est la mobilité : les processus en π-calcul peuvent communiquer entre eux en partageant la connaissance de canaux de communication par lesquels pourront passer des messages de tous ordres, y compris des références à des processus ou à des canaux. Cela permet à π-SPACE de modéliser des architectures dynamiques et évolutives, dans lesquelles les configurations ne sont pas figées, mais peuvent au contraire être sujettes à des ajouts et suppressions de composants et

connecteurs ; cela implique notamment qu'à l'intérieur d'un composant ou d'un connecteur, il soit possible d'adapter le nombre de ports pour certains types de communication.

De plus, la base sémantique formelle de π-SPACE permet l'utilisation de différents outils pour la vérification des architectures et de leurs propriétés.

En outre, le choix du langage de spécification formelle est au moins tout aussi important. Le système logiciel doit au final pouvoir être exécuté, tout en correspondant à sa spécification. La méthode formelle B permet à la fois la vérification de propriétés, et une génération de code. Elle a aussi le grand avantage de disposer d'outils commerciaux permettant d'automatiser à la fois la démonstration de propriétés et la génération de code.

Les patrons de raffinement décrits dans le paragraphe 4.1 permettent de mettre en œuvre un processus de raffinement d'architecture réutilisable qui préserve l'équivalence observationnelle. Ils sont définis en logique de réécriture (cf. chapitre 3), selon la syntaxe supportée par l'outil Maude [Clavel et al. 1999], un système de logique de réécriture. Cette forme de la logique de réécriture est essentiellement basée sur la définition de théories et de modules utilisant ces théories. Une théorie en logique de réécriture est composée d'un vocabulaire, d'équations définies sur ce vocabulaire, et de règles de réécriture étiquetées.

Sont distingués d'une part les modules fonctionnels, qui sont utilisés pour représenter des types de données et les fonctions qui s'appliqueront sur eux, au moyen de théories équationnelles ; chaque étape de réécriture consiste alors en une étape de remplacement d'égal à égal, jusqu'à ce qu'une valeur équivalente, complètement évaluée, soit obtenue.

D'autre part, les modules systèmes peuvent être paramétrés par plusieurs autres théories. Leurs équations regroupent en plus d'une partie équationnelle équivalente à un module fonctionnel, des axiomes. De plus, ces modules systèmes renferment également des règles de réécriture, qui doivent rester cohérentes avec les équations (fonctionnelles) modulo leurs axiomes.

Dans ce cadre, les grammaires de langages peuvent être représentées sous la forme de modules fonctionnels, et les patrons de transformation écrits comme des modules systèmes utilisant les précédents : les traductions du vocabulaire sont parfaitement exprimées par des règles de réécritures, et ces modifications sont propagées dans les éléments architecturaux grâce aux équations.

Mais, la logique de réécriture peut représenter plus que la syntaxe et servir de support à la sémantique. En effet, elle permet de décrire l'interprétation des comportements dans l'espace des processus, et ainsi de vérifier l'équivalence observationnelle pour la théorie de transformation seulement, plutôt que de le faire à chaque application (nous pouvons vérifier que chaque patron de transformation conserve les propriétés de la spécification qu'il raffine, au lieu de le faire à chacune de ses utilisations).

Notre approche β-SPACE est conçue pour permettre la vérification de modèle d'une architecture abstraite en π-SPACE et son raffinement progressif. Par exemple, si la grammaire de π-SPACE est définie en logique de réécriture sous la forme d'un module fonctionnel, il sera possible de vérifier que la description architecturale abstraite est bien reconnue comme un théorème de cette théorie. Les patrons de transformation, stockés dans des modules systèmes, serviront alors à effectuer des réécritures sur la spécification abstraite, jusqu'à obtenir la spécification concrète correspondante en machines abstraites B. Le processus de raffinement obtenu est alors réutilisable et assure l'équivalence observationnelle entre les différents niveaux d'abstraction.

4.3. Restrictions et hypothèses générales dues aux langages

Ces langages ont été choisis pour leurs propriétés respectives, appropriées à des phases bien distinctes de la conception. Pourtant, leurs interactions imposent de faire quelques hypothèses sur la façon d'utiliser chacun en fonction des autres.

En effet, la méthode B ne permet pas de spécifier simplement tout ce que π-SPACE peut exprimer. Son pouvoir d'expression nous amène à ne traiter que certaines constructions du langage de description d'architecture pour ne pas alourdir l'écriture des patrons de transformation. Tout d'abord, les machines abstraites B n'autorisent, à la base, que la description statique de systèmes. C'est pourquoi le dynamisme et l'évolution dans π-SPACE ne sont pas pris en compte. De plus, on suppose que tous les paramètres ont été choisis, et qu'il n'y a donc plus de paramètre formel en attente d'instanciation dans la description architecturale du système. Par ailleurs, la réplication (opérateur de "boucle infinie" du π-calcul) n'a pas d'équivalent dans la méthode B. Elle peut néanmoins être exprimée autrement en π-SPACE à l'aide de processus, aussi nous ne la raffinons pas.

Certains détails dans la gestion des composants et des connecteurs soulèvent d'autres problèmes. Ainsi, les processus basés sur le π-calcul – dans les comportements de composants ou de connecteurs, et dans les ports eux-mêmes – prennent en paramètres des listes de ports ; mais la gestion de l'ordre des ports dans ces listes, comme elle fait partie des aspects dynamiques de π-SPACE, n'a pas d'utilité pour nous. Par conséquent, une hypothèse est ajoutée sur le fait qu'à chaque fois qu'il y est fait appel, un processus est toujours appelé avec les mêmes ports jouant les mêmes rôles, c'est-à-dire que les ports ne constituent en aucun cas des paramètres formels des processus. En outre, les types de canaux (les types des données transmises par ces canaux) sont considérés comme des types de données simples (dans les parties concernant la composition et les types de canaux après transformation). Les types de comportements peuvent encore manipuler des canaux avec listes de paramètres ; seulement, ces listes seront utilisées sous une forme réduite à un paramètre unique (le type simple).

A la fois pour simplifier la transformation et pour compléter les hypothèses précédentes, il est supposé qu'une phase d'instanciation a déjà eu lieu, avant que la description architecturale abstraite en π-SPACE soit manipulée. Certains mots-clés et constructions du langage peuvent avoir ainsi disparu de la spécification après avoir été pris en compte. C'est par exemple le cas des "extends" et des types de composition ; en effet, ces derniers peuvent avoir été utilisés lors de l'instanciation, et donc n'apparaîtront plus, ni dans la partie sur la composition, ni dans celle de déclaration d'éléments architecturaux, ni dans celle des attachements (la partie "where"). Par ailleurs, les ports deviennent commutatifs : une fois le renommage effectué, il n'est plus nécessaire d'en conserver l'ordre. De même, les diverses déclarations d'éléments architecturaux sont rendues elles aussi commutatives, du fait que les attachements directs de canaux sont désormais pris en compte.

Pour un établissement correct des communications, chaque connecteur doit disposer d'au moins deux ports.

4.4. Base de travail en logique de réécriture

Le langage de description d'architecture abstrait est donc dérivé de π-SPACE. Il n'en est pas très éloigné mais ajoute cependant quelques restrictions syntaxiques (cf. paragraphe précédent).

Pour les besoins du raffinement, il est nécessaire de pouvoir distinguer les différentes parties d'une spécification architecturale. C'est pourquoi, le fait de pouvoir écrire en logique de réécriture la grammaire du langage initial est primordial. Nous référençons ce formalisme sous l'appellation π-SPACE$_0$. Le module fonctionnel présente la structure la plus adaptée (cf. chapitre 3, paragraphes 4.2 et 4.3). La figure IV.3 montre des extraits de cette grammaire.

"fmod PI-SPACE0 is" identifie le module fonctionnel. Les différentes constructions qu'il est possible de rencontrer sont les définitions de sortes, de sous-sortes, d'opérations, de variables, de contraintes ensemblistes et d'équations. Jusqu'au mot-clé "endfm", ces différentes instructions peuvent s'enchaîner, chacune terminée par un "." final. Les mots-clés "sort" ou "sorts" introduisent ici les différentes sortes, ou types de structures syntaxiques ; les identifiants ainsi déclarés peuvent alors servir dans le reste du module fonctionnel afin d'expliciter les liens entre eux pour construire les architectures. La ligne "protecting QID ." qui suit directement l'en-tête du module fonctionnel permet de réutiliser la théorie prédéfinie "QID", qui permet d'identifier certains éléments

syntaxiques comme des identificateurs, c'est-à-dire des noms entrés par l'utilisateur, ou noms terminaux. Les commentaires sont précédés dans la figure de "***".

Les sortes représentent des ensembles de valeurs, et le lien le plus facile à établir entre elles est l'inclusion de ces ensembles les uns dans les autres, par le mot-clé "subsorts". Le caractère "<" souligne le fait que la ou les sortes à sa droite sont plus grandes, ou contiennent, la ou les sortes à sa gauche :
"subsorts PortName ParameterName < ProcessParameter < ListOfProcessParameters ."
indique ainsi que les sortes "PortName" et "ParameterName" font toutes deux partie de la sorte "ProcessParameter", et qu'un élément de cette dernière peut également être vu comme un membre de la sorte plus grande "ListOfProcessParameters". Globalement, toutes les sortes introduites dans ce module fonctionnel correspondent aux noms terminaux et non-terminaux de la grammaire originale du langage π-SPACE (modulo les restrictions).

```
fmod PI-SPACE0 is
protecting QID .
***    SORTES
sorts Architecture ComponentTypeName PortName PortTypeName ListOfTypedChannels
    ConnectorTypeName .
sorts CompositeName ComponentName ConnectorName ListOfTypedPorts
    ArchitecturalElementDeclarations .
***
subsorts PortName ParameterName < ProcessParameter < ListOfProcessParameters .

*** PORTS TYPE
*** BEHAVIOUR COMPONENT TYPE
*** BEHAVIOUR CONNECTOR TYPE
*** COMPONENT TYPE
*** CONNECTOR TYPE

*** COMPOSITION DECLARATION
op attach_@_to_@_ : ComponentName PortName ConnectorName PortName ->
    AttachOperation .
op _`,_ : AttachOperation ListOfAttachmentDeclarations -> ListOfAttachmentDeclarations .

op _`:_`[_`] : ConnectorName ConnectorTypeName ListOfTypedPorts ->
    TypedArchitecturalElement .
op _`:_`[_`] : ComponentName ComponentTypeName ListOfTypedPorts ->
    TypedArchitecturalElement .

op _||_ : TypedArchitecturalElement ArchitecturalElementDeclarations ->
    ArchitecturalElementDeclarations .

op compose_`{decompose__where_`} : CompositeName CompositeName
    ArchitecturalElementDeclarations ListOfAttachmentDeclarations -> Composite .
op compose_`{_where_`} : CompositeName ArchitecturalElementDeclarations
    ListOfAttachmentDeclarations -> Composite .

op __ : Architecture Architecture -> Architecture .

*** FIN COMPOSITION DECLARATION
endfm
```

Figure IV.3. : extraits du module fonctionnel décrivant la grammaire du langage de description d'architecture abstrait π-SPACE$_0$

La structure et les mots-clés du langage sont introduits grâce aux opérations en logique de réécriture. L'exemple de la figure IV.3 montre les opérations liées à la composition des composants et connecteurs entre eux pour former les architectures, aussi bien dans l'expression de la composition (avec les mots-clés de π-SPACE "compose", "attach", …), qu'avec "op _ _ : Architecture Architecture -> Architecture .", pour former de façon syntaxique la description architecturale à partir de ses différents constituants (deux morceaux d'architecture juxtaposés constituent une architecture plus complexe).

Cet exemple, s'il ne présente qu'une petite partie de la grammaire complète, permet néanmoins de révéler toute la difficulté de sa formalisation en logique de réécriture. En effet, ce langage n'offre pas toutes les subtilités syntaxiques d'autres méthodes formelles spécifiquement étudiées pour la description grammaticale de langages, comme la BNF (notation formelle pour représenter les grammaires de langages proposée par John Backus et Peter Naur). D'une part, une même syntaxe peut servir à représenter des choses différentes, comme le montre la figure avec "_`:_`[_`]", qui peut servir à représenter soit une instance de composant, soit une instance de connecteur, dans la partie compositionnelle de l'architecture. Dans les deux cas, le nom donné à l'instance est suivi de ":", du nom du type de composant ou de connecteur et entre crochets de la liste des ports types utilisés. D'autre part, "compose_`{decompose__where_`}" et "compose_`{_where_`}" sont deux profils d'opérations, différant l'un de l'autre par une partie optionnelle "decompose_" ; chaque possibilité supplémentaire de construction nécessite d'expliciter sa syntaxe spécifique en utilisant une nouvelle opération, même si celle-ci présentera peu de modifications par rapport à ses voisines.

La grammaire de la figure IV.3 ne nécessite pas de déclaration de variable (introduites par "var" ou "vars"), de contrainte ensembliste ou d'équation (avec "cmb", "ceq"…), mais elle comprend en fait bien plus de sortes et des opérations pour présenter la syntaxe des types de ports, de composants, de connecteurs, de comportements de composants et de comportements de connecteurs.

Cependant, le premier patron de raffinement de β-SPACE manipule les constructions de π-SPACE_0, en efface certaines, en introduit de nouvelles. Il doit ainsi correspondre en même temps à ce module fonctionnel "PI-SPACE0" et au premier langage intermédiaire de description d'architecture. Au final, si nous considérons l'intégralité de la transformation, la grammaire présentée doit être étendue, afin de regrouper les syntaxes utilisées comme support par toutes les étapes du raffinement.

4.5. Premier patron de transformation

Nos différents patrons de raffinement seront décrits plus précisément dans la suite du chapitre, notamment leurs représentations en logique de réécriture. Globalement, ils servent à mettre en œuvre un processus de raffinement vertical complémentaire aux mécanismes de décomposition présents dans les langages de description d'architecture. Ils sont, pour cela, séparés en fonction des différents objectifs de l'étape de transformation qu'ils conduisent (cf. paragraphe 4.1).

Le premier patron s'intéresse à la prise en compte des correspondances entre les ports des instances de composants et de connecteurs. Au niveau de la définition grammaticale du langage de description d'architecture π-SPACE, cette première transformation se borne à supprimer la liste de déclaration d'attachements, avec son mot-clé "where" ; en logique de réécriture, nous pouvons en même temps laisser de côté les déclarations de sortes, sous-sortes et opérations de la grammaire qui y sont exclusivement associées.

Un opérateur booléen "absence" est introduit pour tester l'absence d'un port dans une liste de ports.

Les déclarations des types de ports et des composants en général ne sont pas affectées, mais les références aux différents noms et types de ports des connecteurs sont remplacées par celles correspondant aux ports de composants associés.

La figure IV.4 propose un schéma pour une configuration architecturale contenant 2 composants (en foncé, sur les extrémités) et un connecteur (en plus clair, au centre). Les éléments au centre des composants et du connecteur représentent les comportements, et les cylindres en périphérie, les

ports. Les cylindres au dessus correspondent aux types de ports, avec les canaux de communication utilisés ; chaque type de port correspond dans cet exemple au port en dessous, dans le même ordre de la gauche vers la droite.

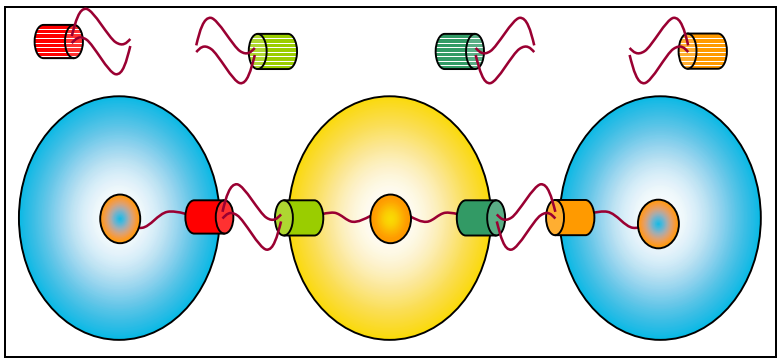

Figure IV.4. : schéma pour un exemple de configuration de 2 composants et 1 connecteur avant le premier patron

Après l'application du premier patron de transformation de β-SPACE (cf. figure IV.5), les attachements disparaissent du diagramme ainsi que les ports initiaux du connecteur.

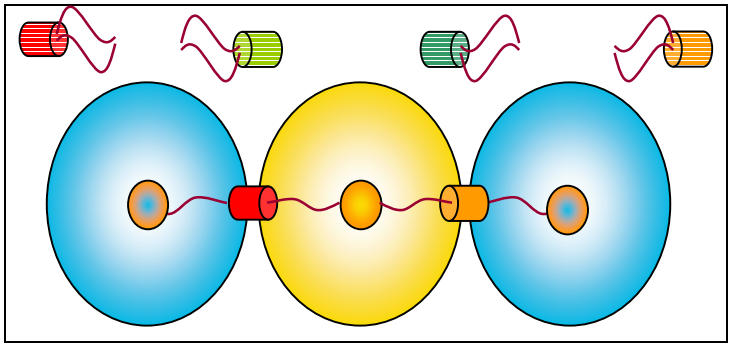

Figure IV.5. : schéma pour l'exemple de configuration après l'application du premier patron

4.6. Deuxième patron de transformation

L'objectif de l'étape suivante du raffinement est l'élimination des dernières informations devenues superflues dans une description en π-SPACE dès lors que les descriptions d'architecture doivent rester statiques.

La mise en œuvre du deuxième patron requiert des hypothèses supplémentaires. Tout d'abord, tous les ports sont supposés attachés : plus aucun port ne doit être resté libre à ce moment, après l'instanciation et la prise en compte des attachements. Pour simplifier le nombre de cas à traiter, un même nom de port ne doit pas apparaître plusieurs fois dans la liste de paramètres d'un processus ou dans la liste de noms d'un port. Pour la même raison, les appels d'opérations faisant référence à certains paramètres seulement (dont les appels sont construits à partir des noms d'opérations et de "@"), distinguant la requête de la réception du résultat, dans les définitions de comportements, ne nous intéressent pas.

D'autre part, après la première transformation, toutes les demandes de renommage des ports de connecteurs doivent avoir été effectuées, avant que la première suppression de port de composant n'intervienne dans la deuxième étape.

Le résultat du passage de l'exemple de la figure IV.5 par le deuxième patron de β-SPACE est représenté dans la figure IV.6. Les types de ports en ont quasiment disparu ; il n'en reste plus que les canaux de communication utilisés par les ports de composants.

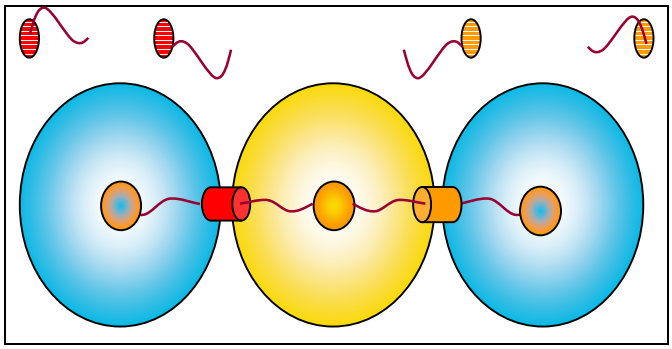

Figure IV.6. : schéma pour l'exemple de configuration après l'application du deuxième patron

4.7. Forme architecturale canonique

L'objectif de ces deux premières phases du raffinement d'architecture a essentiellement consisté à éliminer les structures du langage abstrait qui étaient devenues superflues, soit parce qu'il s'agissait à l'origine de sucre syntaxique, soit parce qu'après instanciation leurs informations devenaient redondantes (cf. paragraphes 4.5 et 4.6). L'architecture intermédiaire obtenue n'est toujours pas sous la forme voulue, mais la suite des transformations sera motivée uniquement par l'obtention du langage cible.

Le langage intermédiaire correspondant provient de π-SPACE, le langage de description d'architecture abstrait choisi. Par contre, n'ont été conservées que les constructions architecturales habituelles et indispensables, avec un vocabulaire qui aurait pu être tout autre, du moment que tous les concepts de base sont là, que rien ne manque ou ne surcharge la description. Néanmoins, à ce moment, il ne dépend pas encore de la méthode B, et arrive en fait juste avant que cela ne devienne le cas. Il est ainsi justifié de le considérer comme canonique.

Ce langage conviendrait en fait très bien pour des processus de raffinement vertical basés sur d'autres langages de descriptions d'architectures abstraits ou concrets. Il permet de modéliser des compositions de composants et de connecteurs qui communiquent via des canaux de communication selon des protocoles définis dans des ports cohérents avec les comportements. Les comportements des composants et connecteurs peuvent être explicités sous la forme de plusieurs processus et les composants peuvent disposer d'opérations internes spécifiques.

4.8. Troisième patron de transformation

Le traitement de la forme canonique par le troisième patron de transformation ajoute de nouvelles hypothèses à celles concernant déjà le langage de description d'architecture abstrait. Même si cet ensemble de règles de réécriture s'intéresse essentiellement au formalisme concret (cf. paragraphe précédent), il apporte pourtant de nouvelles restrictions sur la spécification de départ.

En effet, afin de simplifier la traduction du contrôle des actions menées dans les composants et connecteurs en machines abstraites B, il est préférable de clarifier, voire d'éliminer, certains indéterminismes de la syntaxe de π-SPACE. Ainsi, il est plus aisé de supposer que les définitions de processus sont commutatives, tout comme les déclarations d'opérations, les déclarations des types de canaux de communication, celles d'ensembles de typage ou encore les listes de types de paramètres et de paramètres typés. De plus, dans la définition d'un comportement, le membre de gauche d'une séquence ne devrait être un appel ni de processus ni de comportement.

En outre, les listes de paramètres (avec ou sans leurs types) des processus sont également considérées à ce niveau comme commutatives pour faciliter leur traitement, les noms ayant déjà été harmonisés lors des phases préliminaires : la description architecturale de départ est supposée correcte et elle a subi une phase d'instanciation pour traiter des renommages statiques qui ont éventuellement pu être déclarés avec les instances. Il est aussi supposé, de façon tout à fait naturelle, que toutes les variables utilisées dans les processus sont déclarées dans les

comportements correspondants. Dans le même ordre d'idée, un même paramètre pourrait être utilisé plusieurs fois dans un même appel : par exemple, pour décrire le double d'un nombre comme étant la somme de deux de ses occurrences. Par contre, une même variable ne peut pas servir deux fois en sortie d'une même opération (avec les mots-clés "out" et/ou inout"). D'autre part, les paramètres en entrée (indiqués par le mot-clé "in") et ceux en sortie (avec le mot-clé "out") sont considérés comme différents, à moins qu'ils ne soient déclarés avec le mot-clé "inout". D'ailleurs, il est utile de préciser qu'il ne faut aucune variable qui ait un nom se terminant par "_inout" : c'est une terminaison qui nous sert ici pour créer de nouveaux noms de variables dans ce patron de raffinement.

Enfin, il peut être bon de rappeler que les déclarations en B de variables, d'invariants ou d'initialisation sont commutatives. Cela allège également le travail sur les spécifications et n'entame en rien le pouvoir d'expression de la méthode B.

La figure IV.7 présente le résultat de l'application du troisième patron de transformation de β-SPACE à l'exemple de la figure IV.6. Les ellipses représentent chacune une machine abstraite B différente. Nous retrouvons pour chaque composant une machine pour le contrôle et une autre pour le comportement et les canaux de communication utilisés. Une machine abstraite encapsule le connecteur et une dernière conserve les informations de configuration.

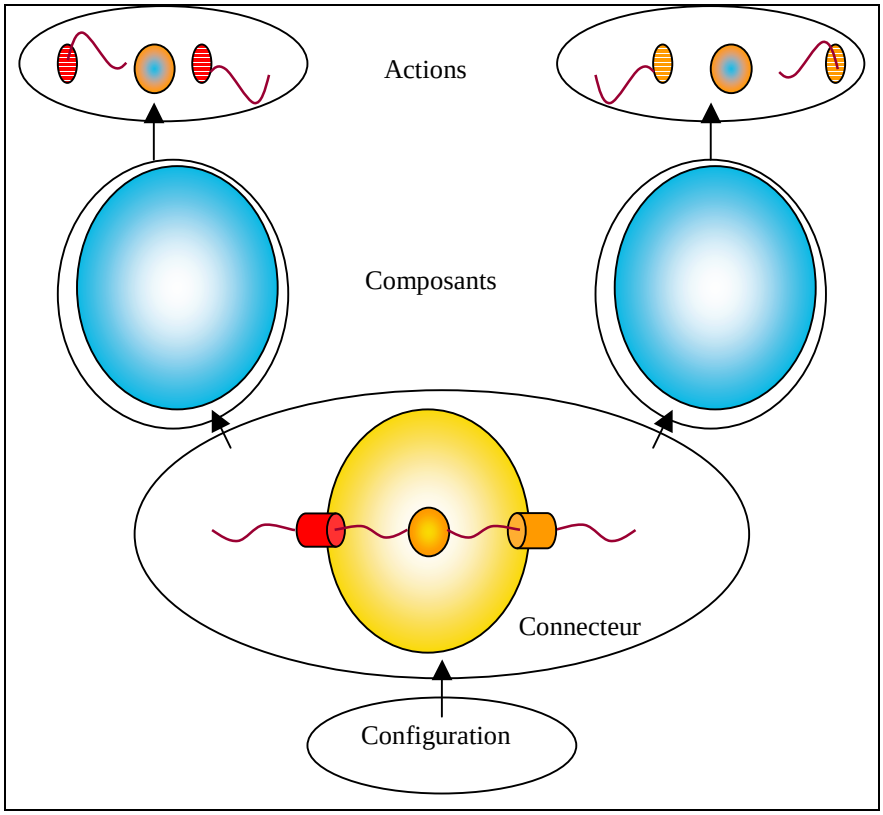

Figure IV.7. : schéma pour l'exemple de configuration après l'application du troisième patron

4.9. Conclusion

L'approche β-SPACE, que nous venons de présenter dans cette partie, a pour objectif principal le raffinement vertical d'une spécification de l'architecture d'un système en π-SPACE, un langage de description d'architectures logicielles dynamiques et évolutives, en machines abstraites de la méthode B, un formalisme orienté modèle. Ce dernier permet, par la suite, un développement sûr pour obtenir un exécutable de l'application. Cependant, les spécifications de la méthode B sont par essence statiques (cf. paragraphe 4.3). Dès lors, cette restriction importante doit également porter sur l'architecture initiale afin de ne pas augmenter, de façon exponentielle, la tâche d'écriture des règles de réécriture.

Ce processus de raffinement est donc contraint par les deux langages de spécification formelle choisis comme source et cible. Néanmoins, afin d'améliorer la réutilisation, les différentes transformations à opérer sont divisées en plusieurs patrons, constitués de règles de réécriture, et d'équations pour diminuer le nombre de cas à traiter par des règles (cf. paragraphe 4.1). Les deux premiers patrons, présentés dans les paragraphes 4.5 et 4.6, servent à éliminer de l'architecture abstraite les descriptions des types de port π-SPACE qui sont devenues inutiles du fait des hypothèses de départ.

La description ainsi obtenue est alors canonique, dans le sens où elle ne présente plus de constructions architecturales propres au langage de description d'architecture de départ, mais uniquement des constructions architecturales usuelles (cf. paragraphe 4.7).

Le patron de raffinement suivant sert à transformer l'architecture canonique en machines abstraites B (cf. paragraphe 4.8). Cela requiert de changer le mode de contrôle des actions à l'intérieur des descriptions du même système : il faut passer de comportements concurrents de composants et connecteurs, synchronisés par des communications, à des modules organisés hiérarchiquement, liés par des appels d'opérations.

Chaque patron est constitué de règles de réécriture, écrites de façon systématique à partir de la grammaire formelle de π-SPACE$_0$ (sous-ensemble statique de π-SPACE introduit dans le paragraphe 4.4). Cette démarche assure la complétude de notre approche par rapport à π-SPACE$_0$. De même, les transformations définies par ces patrons sont correctes par construction, chaque règle respectant la grammaire des machines abstraites B.

5. Présentation des patrons à l'aide d'une étude de cas

Il est plus aisé de se représenter le fonctionnement d'une méthodologie sur une étude de cas. En effet, l'illustration sur un exemple permet de visualiser de façon simple mais efficace les différents aspects traités. Le problème choisi, s'il est assez sommaire, permet néanmoins de développer des explications sur toutes les étapes de la transformation.

Le système de l'écrivain/vérificateur, développé dans [Kirby 1998], [Kirby et Greenwood 1998] et [Warboys et al. 1999], décrit un processus de développement d'un artefact (logiciel ou autre) impliquant une personne qui écrit un document, ou module, et une autre qui doit le vérifier (cf. figure IV.8). Toutes deux doivent interagir jusqu'à la validation du document : si le vérificateur envoie une demande de modification, le rédacteur devra lui fournir une nouvelle version du document ; en cas d'accord du vérificateur, les deux agents pourront s'arrêter sur un constat de réussite.

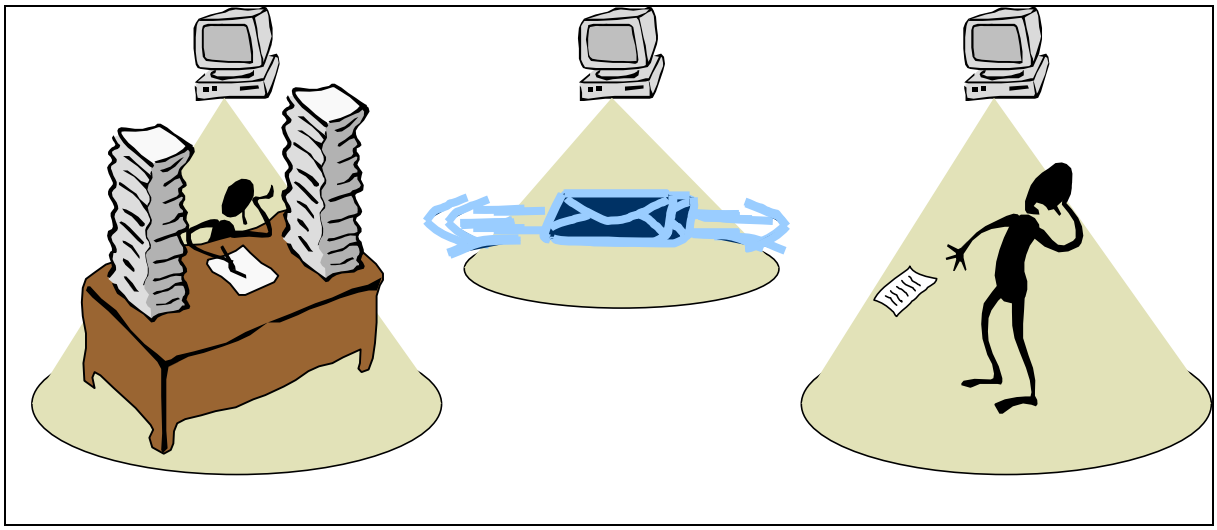

Figure IV.8. : schéma du système écrivain/vérificateur

5.1. Description architecturale abstraite

La description architecturale du système en π-SPACE tient lieu de spécification abstraite, et elle présente une sémantique pour son comportement basée sur l'algèbre de processus π-calcul [Milner et al. 1992]. Sa traduction en machines abstraites B n'est pas triviale et s'effectue plus facilement étape par étape. Chaque patron de transformation de β-SPACE est ainsi conçu pour agir spécifiquement sur des parties différentes de l'architecture originale.

L'architecture "E1A1V1" en π-SPACE [Chaudet et Oquendo 2000], illustrée par la figure IV.9, est constituée d'un composant écrivain, qui est une instance "E1" du type de composant "Ecrivain1" (définie par "E1: Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]]"), d'un composant vérificateur, représenté par l'instance "V1" du type de composant "Vérificateur1" (avec "A1: Aiguillage1[appelant: Réponse[envoyer: [Module], recevoir: [Module]], appelé: Requête[recevoir: [Module], envoyer: [Module]]]"), et d'un connecteur assurant la transmission du document entre les deux composants, défini par comme l'instance "A1" du type de connecteur "Aiguillage1" (par "V1: Vérificateur1[fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]"). Tous trois sont composés ; ils sont attachés deux à deux par leurs ports : le port "demande_vérification" de "E1" est attaché au port "appelant" de "A1" (par "attach E1 @ demande_vérification to A1 @ appelant"), et le port "fournit_vérification" de "V1" au port "appelé" de "A1" (comme décrit par "attach V1 @ fournit_vérification to A1 @ appelé").

Le composant "E1", par exemple, est une instance du type de composant "Ecrivain1" (défini par "define component type Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]] { port demande_vérification: Requête[envoyer: [Module], recevoir: [Module]] || behaviour écriture: Ecriture[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]] }"). La déclaration de l'instance précise le nom de port "demande_vérification" avec comme type de port "Requête" en utilisant les canaux "envoyer" et "recevoir" servant à véhiculer des messages de type "Module". Ces noms correspondent à ceux utilisés dans le type de composant et le type de port, mais ils auraient pu être différents.

C'est notamment pour cette raison que nous avons posé l'hypothèse d'une phase d'instanciation préliminaire. Une conséquence essentielle, dans notre exemple en tout cas, est le dédoublement des définitions des types de ports, comme "define port type Requête[recevoir: [Module], envoyer: [Module]] { par1: Module, par2: Module, Requête[recevoir, envoyer] = ((recevoir<par1> ¤ envoyer (par2) ¤ Requête[recevoir, envoyer]) + $) }" et "define port type Requête[envoyer: [Module], recevoir: [Module]] { par1: Module, par2: Module, Requête[envoyer, recevoir] = ((envoyer<par1> ¤ recevoir (par2) ¤ Requête[envoyer, recevoir]) + $) }" qui correspondent au même type de port dans l'architecture abstraite initiale, mais à des utilisations avec des noms de canaux différents. Par conséquent, au lieu de ne caractériser un type de port que par le seul nom utilisé dans l'architecture abstraite originale, il est représenté plus avantageusement par l'association de ce nom avec la liste exhaustive des noms et types des canaux utilisés. L'avantage principal est que les patrons de raffinement n'auront pas, par la suite, à devoir s'acquitter d'éventuels renommages à effectuer pendant le traitement des appels liés aux ports des composants et connecteurs, alors que l'utilisation d'une dénomination plus complexe ne complique pas le traitement en logique de réécriture.

De ce fait, chaque déclaration de type de composant, de connecteur, de comportement de composant ou de comportement de connecteur, va faire référence à des types de ports pleinement définis, qui n'auront pas à être remaniés dans la suite des opérations.

```
define port type Requête[recevoir: [Module], envoyer: [Module]] {
par1: Module, par2: Module,
Requête[recevoir, envoyer] = ((recevoir<par1> ¤ envoyer (par2) ¤ Requête[recevoir, envoyer])
    + $)
}
define port type Réponse[envoyer: [Module], recevoir: [Module]] {
par1: Module, par2: Module,
Réponse[envoyer, recevoir] =
    ((envoyer(par1) ¤ recevoir<par2> ¤ Réponse[envoyer, recevoir]) + $)
}
define port type Requête[envoyer: [Module], recevoir: [Module]] {
par1: Module, par2: Module,
Requête[envoyer, recevoir] = ((envoyer<par1> ¤ recevoir (par2) ¤ Requête[envoyer, recevoir])
    + $)
}
define port type Réponse[recevoir: [Module], envoyer: [Module]] {
par1: Module, par2: Module,
Réponse[recevoir, envoyer] =
    ((recevoir(par1) ¤ envoyer<par2> ¤ Réponse[recevoir, envoyer]) + $)
}

define behaviour component type Ecriture[demande_vérification: Requête[envoyer:
    [Module], recevoir: [Module]]] {
module: Module, retour: Module,
faire_initialisation[out[Module]] { ... },
faire_écriture1[inout[Module]] { ... },
Ecriture[demande_vérification] =
    ( (faire_initialisation[module] ¤ demande_vérification @ envoyer <module>
    ¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[demande_vérification,
    retour]) + $),
processus_écriture[demande_vérification: Requête, module: Module] =
    ( (faire_écriture1[module] ¤ demande_vérification @ envoyer <module>
    ¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[demande_vérification,
    retour]) + $)
}
define behaviour component type Vérification[fournit_vérification: Réponse[recevoir:
    [Module], envoyer: [Module]]] {
module: Module, retour: Module,
faire_vérification1[in[Module], out[Module]] { ... },
Vérification[fournit_vérification] =
    ( (fournit_vérification @ recevoir (module) ¤ faire_vérification1[module, retour]
    ¤ fournit_vérification @ envoyer <retour> ¤ Vérification[fournit_vérification]) + $)
}
define behaviour connector type Aiguillage[appelant: Réponse[envoyer: [Module], recevoir:
    [Module]], appelé: Requête[recevoir: [Module], envoyer: [Module]]] {
module: Module, retour: Module,
Aiguillage[appelant, appelé] =
    ( (appelant @ envoyer (module) ¤ appelé @ recevoir<module>
    ¤ appelé @ envoyer (retour) ¤ appelant @ recevoir <retour> ¤ Aiguillage[appelant, appelé])
    + $)
}
```

```
define component type Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]]] {
port demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]                    ||
behaviour écriture: Ecriture[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]]]
}
define component type Vérificateur1[fournit_vérification: Réponse[recevoir: [Module],
    envoyer: [Module]]] {
port fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]                    ||
behaviour vérification: Vérification[fournit_vérification: Réponse[recevoir: [Module], envoyer:
    [Module]]]
}
define connector type Aiguillage1[appelant: Réponse[envoyer: [Module], recevoir: [Module]],
    appelé: Requête[recevoir: [Module], envoyer: [Module]]] {
port appelant: Réponse[envoyer: [Module], recevoir: [Module]]                                ||
port appelé: Requête[recevoir: [Module], envoyer: [Module]]                                  ||
behaviour aiguillage: Aiguillage[appelant: Réponse[envoyer: [Module], recevoir: [Module]],
    appelé: Requête[recevoir: [Module], envoyer: [Module]]]
}

compose E1A1V1 {
    E1: Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]]      ||
    A1: Aiguillage1[appelant: Réponse[envoyer: [Module], recevoir: [Module]],
                appelé: Requête[recevoir: [Module], envoyer: [Module]]]                      ||
    V1: Vérificateur1[fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]
where
    attach E1 @ demande_vérification      to      A1 @ appelant,
    attach V1 @ fournit_vérification      to      A1 @ appelé
}
```

Figure IV.9. : spécification architecturale abstraite de l'étude de cas en π-SPACE

5.2. Premier patron de transformation

Dès lors que la spécification originale en π-SPACE est correcte, il est possible de supposer que les ports attachés entre eux sont cohérents et que, à l'intérieur d'un même connecteur, les ports sont cohérents avec le comportement, c'est-à-dire que dans le type du connecteur, le type de comportement utilisé est en cohérence avec les types de ports qui lui sont liés. C'est pourquoi le premier patron de transformation de β-SPACE, présenté dans les paragraphes 4.1 et 4.5, concerne les informations d'attachement : toutes les références aux ports de connecteurs sont remplacées par les références aux ports de composants correspondants.

La figure IV.10 présente les différentes composantes architecturales de la spécification de départ que la première étape de raffinement affectera. Les éléments mis en relief, en gras et en italique, auront soit disparu, soit été remplacés pour obtenir la première spécification architecturale intermédiaire.

Cette première étape de raffinement est décrite en logique de réécriture à l'aide de 21 règles, préfigurant les différents types de configuration qui peuvent être rencontrés. En outre, 33 équations viennent compléter le patron afin de propager les changements à opérer dans l'architecture (notamment les changements de noms de ports) et diminuer le nombre de cas à traiter par des règles. Même si toutes ces règles et équations ne sont pas nécessaires pour la transformation de la description de l'architecture logicielle de l'étude de cas, elles peuvent néanmoins être appliquées à toute architecture de système.

```
define port type Requête[recevoir: [Module], envoyer: [Module]] {
par1: Module, par2: Module,
Requête[recevoir, envoyer] = ((recevoir<par1> ¤ envoyer (par2) ¤ Requête[recevoir, envoyer])
    + $)
}

define port type Réponse[envoyer: [Module], recevoir: [Module]] {
par1: Module, par2: Module,
Réponse[envoyer, recevoir] =
    ((envoyer(par1) ¤ recevoir<par2> ¤ Réponse[envoyer, recevoir]) + $)
}

define behaviour connector type Aiguillage[appelant: Réponse[envoyer: [Module], recevoir:
    [Module]], appelé: Requête[recevoir: [Module], envoyer: [Module]]] {
module: Module, retour: Module,
Aiguillage[appelant, appelé] =
    ( (appelant @ envoyer (module) ¤ appelé @ recevoir<module>
    ¤ appelé @ envoyer (retour) ¤ appelant @ recevoir <retour> ¤ Aiguillage[appelant, appelé])
    + $)
}

define connector type Aiguillage1[appelant: Réponse[envoyer: [Module], recevoir: [Module]],
    appelé: Requête[recevoir: [Module], envoyer: [Module]]] {
port appelant: Réponse[envoyer: [Module], recevoir: [Module]]                    ||
port appelé: Requête[recevoir: [Module], envoyer: [Module]]                      ||
behaviour aiguillage: Aiguillage[appelant: Réponse[envoyer: [Module], recevoir: [Module]],
    appelé: Requête[recevoir: [Module], envoyer: [Module]]]
}

compose E1A1V1 {
    E1: Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]]    ||
    A1: Aiguillage1[appelant: Réponse[envoyer: [Module], recevoir: [Module]],
                appelé: Requête[recevoir: [Module], envoyer: [Module]]]                   ||
    V1: Vérificateur1[fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]
where
    attach E1 @ demande_vérification      to      A1 @ appelant,
    attach V1 @ fournit_vérification      to      A1 @ appelé
}
```

Figure IV.10. : identification de la partie de la spécification architecturale abstraite ciblée par le premier patron de transformation

Les règles de réécriture du premier patron de transformation peuvent être séparées en 6 groupes, correspondant chacun à un élément architectural différent. Le premier groupe de règles traite ainsi la partie composition de l'architecture abstraite. C'est en effet celle-ci qui contient l'information sur les attachements. Huit règles sont nécessaires pour recouvrir les différents cas qui peuvent se présenter : selon que cette composition s'appuie elle-même sur une décomposition ou pas, que le type de composant concerné contient plusieurs types de ports ou un seul (celui que nous traitons ici), et s'il s'agit du dernier attachement à prendre en compte ou non.

La figure IV.11 présente la première règle de transformation utilisée pour la transformation de l'architecture abstraite E1A1V1 de l'exemple de la figure IV.9 ; le caractère "`" précède certains caractères pour indiquer qu'ils doivent être pris en considération non pas comme des éléments de la syntaxe de la règle de réécriture, mais comme objets de cette réécriture. L'intérêt de cette règle est essentiellement focalisé sur la partie "compose" de la spécification, le reste de l'architecture restant présent sous la forme de la variable "ARCHI", de sorte "Architecture" ; cette dernière est définie

dans le module fonctionnel décrivant le langage de description architectural abstrait comme une collection de blocs architecturaux (compositions, types de composants, types de connecteurs, types de comportements de composants, types de comportements de connecteurs ou types de ports). "ARCHI" représente donc les quatre types de ports, les trois types de comportement, les deux types de composants et le type de connecteur de l'architecture abstraite. "COMPOSITION1" est une variable qui prend pour valeur "E1A1V1", le nom de la composition. Cette règle s'intéresse à la première déclaration d'attachement, dans le cas où il en reste plusieurs dans la liste (après le "where"). "COMPOSANT1" est donc identifié à "E1", "PORT1" à "demande_vérification", "CONNECTEUR1" à "A1" et "ROLE1" à "appelant" ; "LADECL" représente le reste de la liste d'attachements qui n'ont pas encore été traités. Les autres déclarations, situées avant le "where", concernent les éléments architecturaux ; les deux seuls qui nous intéressent dans cette règle correspondent aux "COMPOSANT1" et "CONNECTEUR1" déjà identifiés, tandis que les autres instances de composants et connecteurs sont prises en compte grâce à "AEDECL". Cette règle suppose que le composant concerné ne dispose que d'un seul port. "COMPONENTTYPE1" est alors associé à "Ecrivain1", "PORTTYPE1" à "Requête" et "PORTTYPEPARAM" à "envoyer: [Module], recevoir: [Module]]" ; de façon similaire, "CONNECTORTYPE1" prend pour valeur "Aiguillage1", "ROLETYPE1" devient "Réponse" avec "ROLETYPEPARAM" qui vaut "envoyer: [Module], recevoir: [Module]]", et "LTPORTS1" représente les autres ports du connecteur désigné par "CONNECTEUR1", c'est-à-dire "appelé: Requête[recevoir: [Module], envoyer: [Module]]". A ce stade, toute l'architecture a été reconnue par le membre de gauche de la règle, et va se retrouver modifiée de la façon indiquée par le membre de droite, en conservant aux variables les mêmes valeurs que dans le membre de gauche. La partie concernant le port "appelant" du connecteur dans la déclaration de l'instance "A1" est donc remplacée par la déclaration du port "demande_vérification" déjà présente pour l'instance "E1", et la ligne correspondante dans la déclaration des attachements (après le "where") a disparu. Par ailleurs, "ARCHI `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `]" va servir à propager ce remplacement d'une dénomination de port par une autre dans le reste de l'architecture, grâce aux autres groupes de règles de réécriture et aux équations.

```
rl [transfo1.1.111] :

ARCHI

compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[
    ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL where
    attach COMPOSANT1 @ PORT1 to CONNECTEUR1 @ ROLE1 `, LADECL `}

=>

ARCHI `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] /
    CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]

compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[
    PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL where
    LADECL `} .
```

Figure IV.11. : première règle de réécriture du premier patron de transformation appliquée à l'exemple

A cet instant de l'application du premier patron de transformation, le système de réécriture a le choix, soit de continuer à utiliser le premier groupe de règles pour prendre en compte le second attachement de l'exemple, soit de commencer à traiter le premier attachement dans le reste de l'architecture.

L'équation présentée dans la figure IV.12 reprend un changement de noms à opérer sur des éléments architecturaux, comme celui qui résulte de la mise en œuvre de la règle de la figure IV.11,

et permet ensuite de le propager à tous les éléments architecturaux concernés. En effet, cette équation exprime le fait que vouloir effectuer le renommage sur deux ensembles d'éléments consécutifs est équivalent à le faire sur les deux ensembles d'éléments séparément.

```
eq ( ARCHI1 ARCHI2 ) `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[
    ROLETYPEPARAM `] `] =

    ARCHI1 `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] /
    CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]

    ARCHI2 `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] /
    CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `] .
```

Figure IV.12. : équation du premier patron de raffinement assurant la propagation du changement de dénomination des ports des connecteurs dans le reste de l'architecture

Le second attachement de l'exemple est résolu en se servant de la règle de la figure IV.13. Le fonctionnement est en tout point semblable à celui de la règle précédente : "COMPOSANT1" prend ici la valeur "V1" ; "COMPONENTTYPE1" aura "Vérificateur1" ; "PORT1" devient "fournit_vérification" ; "PORTTYPE1" correspond à "Réponse" ; "ROLE1" change de valeur pour "appelé", "ROLETYPE1" pour "Requête", "LTPORTS1" pour "appelant: Réponse[envoyer: [Module], recevoir: [Module]]" ; enfin "PORTTYPEPARAM" et "ROLETYPEPARAM" prennent pour valeur "recevoir: [Module], envoyer: [Module]]". Les seules différences résident dans l'étiquette de la règle de réécriture et le fait qu'il ne reste plus d'attachement à traiter ("LADECL" dans la règle précédente). L'architecture résultant de l'application de cette deuxième règle ne comportera donc plus d'attachement, d'où la disparition en même temps du mot-clé "where". De façon similaire toujours à l'explication pour la première réécriture, l'ancienne référence au port du connecteur est remplacée par celle du port associé, dans la partie "compose" dans un premier temps, et est transmise au reste de l'architecture représenté par "ARCHI" pour une prise en compte dans les autres éléments architecturaux.

```
rl [transfo1.1.112] :

ARCHI

compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[
    ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL where
    attach COMPOSANT1 @ PORT1 to CONNECTEUR1 @ ROLE1 `}

=>

ARCHI `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] /
    CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]

compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[
    PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL `} .
```

Figure IV.13. : règle de réécriture du premier patron de transformation appliquée à l'exemple pour traiter le dernier attachement

Le deuxième groupe de règles de réécriture du premier patron de raffinement de β-SPACE comporte également huit règles. L'élément architectural considéré est la définition d'un type de comportement de connecteur. Les différents cas correspondent à la présence ou non de définitions de variables locales à cette définition, celle de processus complétant la description du comportement, et bien entendu si ce type de comportement de connecteur est concerné ou non par le renommage de port de connecteur qui est à appliquer.

Dans l'exemple de "E1A1V1", le seul type de comportement de connecteur défini est "Aiguillage". La règle correspondante dans le deuxième groupe de règles de réécriture est donnée dans la figure IV.14. "CONNECTORBEHAVIOURTYPE1" représente donc "Aiguillage", "LTPARAM1" a pour valeur "module: Module, retour: Module". "CONNBEHA1" vaut "((appelant @ envoyer (module) ¤ appelé @ recevoir<module> ¤ appelé @ envoyer (retour) ¤ appelant @ recevoir <retour> ¤ Aiguillage[appelant, appelé]) + $)" avant le premier des deux renommages. La signification des autres variables dépend finalement du renommage considéré (celui du port "appelant" ou celui du port "appelé"). Deux applications de cette règle permettent d'assurer la plupart des modifications requises. Les dernières à effectuer sont dues à "CONNBEHA1 `[ PORT1 / ROLE1 `]" dans le résultat de la règle de réécriture ; elles seront en fait prises en charge par d'autres équations chargées de laisser les changements de noms traverser les processus basés sur le π-calcul et remplacer les "ROLE1" par des "PORT1" partout où ils apparaissent.

```
rl [transfo1.2.111] :

define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ ROLE1`:
    ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] `{ LTPARAM1 `,
    CONNECTORBEHAVIOURTYPE1 `[ ROLE1 `, LPN1 `] `= CONNBEHA1 `} `[
    COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] /
    CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]

=>

define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `{ LTPARAM1 `,
    CONNECTORBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= CONNBEHA1 `[ PORT1 /
    ROLE1 `] `} .
```

Figure IV.14. : règle de réécriture du premier patron de transformation appliquée à l'exemple pour traiter le type de comportement de connecteur

Le troisième groupe de deux règles de réécriture traite les types de connecteurs, suivant qu'il y a lieu d'effectuer ou nom un changement de nom de ports. La règle présentée dans la figure IV.15 est donc appliquée 2 fois sur l'exemple pour le type de connecteur "Aiguillage1", correspondant dans la règle de logique de réécriture à la variable "CONNECTORTYPE1". Cette fois, tous les renommages sont effectués par le biais de la règle de réécriture.

```
rl [transfo1.3.1] :

define connector type CONNECTORTYPE1 `[ ROLE1 `: ROLETYPE1 `[
    ROLETYPEPARAM `] `, LTPORTS1 `] `{ port ROLE1`: ROLETYPE1 `[
    ROLETYPEPARAM `] || PORTDECL1 || behaviour CONNECTORBEHAVIOUR1 `:
    CONNECTORBEHAVIOURTYPE1 `[ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `]
    `, LTPORTS1 `] `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[
    ROLETYPEPARAM `] `]

=>

define connector type CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] `, LTPORTS1 `] `{ port PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] || PORTDECL1 || behaviour CONNECTORBEHAVIOUR1 `:
    CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `]
    `, LTPORTS1 `] `} .
```

Figure IV.15. : règle de réécriture du premier patron de transformation appliquée à l'exemple pour traiter le type de connecteur

Les trois groupes de règles de réécriture suivants sont réduits à une seule règle chacun, comme il est possible de la voir dans la figure IV.16. En effet, ils ne traduisent que le fait que l'action sur les ports de connecteurs de ce premier patron de transformation n'affecte en rien les définitions de types de port, de type de comportement de composant ou de type de composant.

```
rl [transfo1.4] : define port type PORTTYPE1 `[ LTCHANNELS1 `] `{ PORTSPEC1 `} `[
    COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] /
    CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]
=>
define port type PORTTYPE1 `[ LTCHANNELS1 `] `{ PORTSPEC1 `} .

rl [transfo1.5] : define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PTP1
    `] `{ COMPBEHAVSPEC1 `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[
    ROLETYPEPARAM `] `]
=>
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PTP1 `] `{
    COMPBEHAVSPEC1 `} .

rl [transfo1.6] : define component type COMPONENTTYPE2 `[ PTP1 `] `{ PORTDECL1 ||
    COMPBEHAVDECL `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[
    ROLETYPEPARAM `] `]
=>
define component type COMPONENTTYPE2 `[ PTP1 `] `{ PORTDECL1 ||
    COMPBEHAVDECL `} .
```

Figure IV.16. : règles de réécriture du premier patron de transformation traitant des éléments architecturaux non affectés

La spécification obtenue après l'application du premier patron de transformation de β-SPACE sur la description d'architecture de la figure IV.9 est reproduite dans la figure IV.17.

Elle résulte de l'application, sur l'architecture abstraite de la figure IV.9, des règles des figures IV.11, IV.13, IV.14, IV.15 et IV.16, ainsi que des équations (par exemple celle de la figure IV.12). Les parties modifiées sont celles qui furent mises en relief dans la figure IV.10.

Cette architecture intermédiaire conserve tous les éléments de base de la version abstraite. En effet, même si certaines définitions de types de ports sont devenues inutiles dans cet exemple, cela pourrait ne pas être le cas pour un système différent, où un même type de port servirait plusieurs fois pour divers composants ou connecteurs.

D'autre part, ces deux architectures permettent de vérifier expérimentalement que le premier patron de transformation répond à la propriété de l'équivalence observationnelle. Effectivement, cette étape de raffinement n'affecte pas les comportements, en se contentant d'effectuer quelques changements de noms ; ni l'ordre, ni la nature des actions des différents types de comportements et types de ports ne sont modifiés.

```
define port type Requête[recevoir: [Module], envoyer: [Module]] {
par1: Module, par2: Module,
Requête[recevoir, envoyer] = ((recevoir<par1> ¤ envoyer (par2) ¤ Requête[recevoir, envoyer])
    + $)
}
define port type Réponse[envoyer: [Module], recevoir: [Module]] {
par1: Module, par2: Module,
Réponse[envoyer, recevoir] =
    ((envoyer(par1) ¤ recevoir<par2> ¤ Réponse[envoyer, recevoir]) + $)
}
define port type Requête[envoyer: [Module], recevoir: [Module]] {
par1: Module, par2: Module,
Requête[envoyer, recevoir] = ((envoyer<par1> ¤ recevoir (par2) ¤ Requête[envoyer, recevoir])
    + $)
}
define port type Réponse[recevoir: [Module], envoyer: [Module]] {
par1: Module, par2: Module,
Réponse[recevoir, envoyer] =
    ((recevoir(par1) ¤ envoyer<par2> ¤ Réponse[recevoir, envoyer]) + $)
}

define behaviour component type Ecriture[demande_vérification: Requête[envoyer:
    [Module], recevoir: [Module]]] {
module: Module, retour: Module,
faire_initialisation[out[Module]] { ... },
faire_écriture1[inout[Module]] { ... },
Ecriture[demande_vérification] =
    ( (faire_initialisation[module] ¤ demande_vérification @ envoyer <module>
    ¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[demande_vérification,
    retour]) + $),
processus_écriture[demande_vérification: Requête, module: Module] =
    ( (faire_écriture1[module] ¤ demande_vérification @ envoyer <module>
    ¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[demande_vérification,
    retour]) + $)
}
define behaviour component type Vérification[fournit_vérification: Réponse[recevoir:
    [Module], envoyer: [Module]]] {
module: Module, retour: Module,
faire_vérification1[in[Module], out[Module]] { ... },
Vérification[fournit_vérification] =
    ( (fournit_vérification @ recevoir (module) ¤ faire_vérification1[module, retour]
    ¤ fournit_vérification @ envoyer <retour> ¤ Vérification[fournit_vérification]) + $)
}
define behaviour connector type Aiguillage[demande_vérification: Requête[envoyer:
    [Module], recevoir: [Module]], fournit_vérification: Réponse[recevoir: [Module], envoyer:
    [Module]]] {
module: Module, retour: Module,
Aiguillage[demande_vérification, fournit_vérification] =
    ( (demande_vérification @ envoyer (module) ¤ fournit_vérification @ recevoir<module>
    ¤ fournit_vérification @ envoyer (retour) ¤ demande_vérification @ recevoir <retour> ¤
    Aiguillage[demande_vérification, fournit_vérification])
    + $)
}
```

```
define component type Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]]] {
port demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]                    ||
behaviour écriture: Ecriture[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]]]
}
define component type Vérificateur1[fournit_vérification: Réponse[recevoir: [Module],
    envoyer: [Module]]] {
port fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]                    ||
behaviour vérification: Vérification[fournit_vérification: Réponse[recevoir: [Module], envoyer:
    [Module]]]
}
define connector type Aiguillage1[demande_vérification: Requête[envoyer: [Module],
    recevoir: [Module]], fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]
    {
port demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]                    ||
port fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]                    ||
behaviour aiguillage: Aiguillage[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]], fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]
}

compose E1A1V1 {
    E1: Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]]      ||
    A1: Aiguillage1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]],
                fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]        ||
    V1: Vérificateur1[fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]
}
```

Figure IV.17. : spécification architecturale de l'étude de cas après application du premier patron de raffinement

5.3. Deuxième patron de transformation

La transformation de l'architecture ne s'arrête pas à un simple renommage des ports des connecteurs pour supprimer une déclaration des liens entre composants et connecteurs. La deuxième étape cherche en effet à compléter le travail déjà entamé sur les ports. Ce patron de raffinement est fondé sur la cohérence à l'intérieur des composants, entre les définitions des types des comportements et celles des ports qui leur sont associés (cf. paragraphes 4.1 et 4.6). Etant donné que l'architecture abstraite de départ est censée être correcte, il n'est plus besoin de s'inquiéter de l'adéquation de ces définitions entre elles. Pour la suite du développement, les spécifications des types de ports deviennent donc superflues.

Néanmoins, il convient de ne pas supprimer toutes les références aux anciens ports de façon aveugle : ce sont les ports qui permettent la communication entre les différents éléments de l'architecture. C'est pourquoi il importe de se souvenir des canaux de communication utilisés dans la description architecturale en π-SPACE pour synchroniser les comportements et transmettre des messages entre les composants et les connecteurs. Ces media de transmission jouent un peu le rôle de variables globales, ou de tubes, et ont donc une existence propre, indépendamment de leur utilisation dans tel composant ou connecteur. C'est d'autant plus vrai en π-SPACE que ce langage de description d'architecture logicielle fut conçu pour pouvoir spécifier des architectures dynamiques et évolutives, dans lesquelles les attachements de ports ne seraient pas pérennes et pourraient même varier en nombre.

Par conséquent, les transformations opérées par le deuxième patron de raffinement de β-SPACE vont épurer la spécification architecturale en ôtant les descriptions des types de ports et les références à ces types. De plus, les déclarations de ports des composants et connecteurs font partie à part entière

de la structure de l'architecture et ne doivent pas disparaître : ils servent toujours à relier les composants et connecteurs entre eux. Enfin, les appels de comportements et de processus (dans les définitions de types de comportements) sont simplifiés, avec le souci toujours présent de rester dans une perspective statique, correspondant à celle de la méthode B. Une hypothèse est ajoutée sur le fonctionnement de ces appels, en supposant qu'ils utiliseront toujours les mêmes ports ; ces derniers peuvent ainsi être retirés de la syntaxe des appels. En d'autres termes, les comportements des composants et connecteurs ne doivent pas être définis en π-SPACE (c'est-à-dire en π-calcul) en utilisant des processus récursifs ou mutuellement récursifs destinés à être interprétés tantôt avec un port et tantôt avec un autre : cela peut d'ailleurs être évité en écrivant des processus différents pour chaque port à prendre en considération (en effet, si la définition de l'architecture initiale est statique, le nombre de ports dans chaque composant ou connecteur est fixe et connu).

La figure IV.18 met en évidence les parties de l'architecture obtenue après application du premier patron de raffinement sur l'exemple, qui sont concernées par cette deuxième vague de transformation. Les portions de spécifications directement affectées apparaissent en gras et en italique. Contrairement à l'application du premier patron de transformation, cette fois l'intégralité de la description architecturale est modifiée, des définitions de types de ports à la composition, en passant par les définitions de types de composants, connecteurs et comportements.

define port type Requête[recevoir: [Module], envoyer: [Module]] {
par1: Module, par2: Module,
Requête[recevoir, envoyer] =
((recevoir<par1> ¤ envoyer (par2) ¤ Requête[recevoir, envoyer]) + $)
}
define port type Réponse[envoyer: [Module], recevoir: [Module]] {
par1: Module, par2: Module,
Réponse[envoyer, recevoir] =
((envoyer(par1) ¤ recevoir<par2> ¤ Réponse[envoyer, recevoir]) + $)
}
define port type Requête[envoyer: [Module], recevoir: [Module]] {
par1: Module, par2: Module,
Requête[envoyer, recevoir] = ((envoyer<par1> ¤ recevoir (par2) ¤ Requête[envoyer, recevoir]) + $)
}
define port type Réponse[recevoir: [Module], envoyer: [Module]] {
par1: Module, par2: Module,
Réponse[recevoir, envoyer] =
((recevoir(par1) ¤ envoyer<par2> ¤ Réponse[recevoir, envoyer]) + $)
}

define behaviour component type Ecriture***[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]]*** {
module: Module, retour: Module,
faire_initialisation[out[Module]] { ... },
faire_écriture1[inout[Module]] { ... },
Ecriture***[demande_vérification]*** =
((faire_initialisation[module] ¤ demande_vérification @ envoyer <module>
¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[***demande_vérification,*** retour]) + $),
processus_écriture[***demande_vérification: Requête,*** module: Module] =
((faire_écriture1[module] ¤ demande_vérification @ envoyer <module>
¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[***demande_vérification,*** retour]) + $)
}

```
define behaviour component type Vérification[fournit_vérification: Réponse[recevoir:
    [Module], envoyer: [Module]]] {
module: Module, retour: Module,
faire_vérification1[in[Module], out[Module]] { ... },
Vérification[fournit_vérification] =
    ( (fournit_vérification @ recevoir (module) ¤ faire_vérification1[module, retour]
    ¤ fournit_vérification @ envoyer <retour> ¤ Vérification[fournit_vérification]) + $)
}
define behaviour connector type Aiguillage[demande_vérification: Requête[envoyer:
    [Module], recevoir: [Module]], fournit_vérification: Réponse[recevoir: [Module], envoyer:
    [Module]]] {
module: Module, retour: Module,
Aiguillage[demande_vérification, fournit_vérification] =
    ( (demande_vérification @ envoyer (module) ¤ fournit_vérification @ recevoir<module>
    ¤ fournit_vérification @ envoyer (retour) ¤ demande_vérification @ recevoir <retour> ¤
    Aiguillage[demande_vérification, fournit_vérification]) + $)
}

define component type Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]]] {
port demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]                ||
behaviour écriture: Ecriture[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]]]
}
define component type Vérificateur1[fournit_vérification: Réponse[recevoir: [Module],
    envoyer: [Module]]] {
port fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]                ||
behaviour vérification: Vérification[fournit_vérification: Réponse[recevoir: [Module],
    envoyer: [Module]]]
}
define connector type Aiguillage1[demande_vérification: Requête[envoyer: [Module],
    recevoir: [Module]], fournit_vérification: Réponse[recevoir: [Module], envoyer:
    [Module]]] {
port demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]                ||
port fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]                ||
behaviour aiguillage: Aiguillage[demande_vérification: Requête[envoyer: [Module], recevoir:
    [Module]], fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]
}

compose E1A1V1 {
    E1: Ecrivain1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]]]  ||
    A1: Aiguillage1[demande_vérification: Requête[envoyer: [Module], recevoir: [Module]],
            fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]        ||
    V1: Vérificateur1[fournit_vérification: Réponse[recevoir: [Module], envoyer: [Module]]]
}
```

Figure IV.18. : identification de la partie de l'architecture ciblée par le deuxième patron de transformation

Le deuxième patron de transformation de β-SPACE comporte 62 règles de réécriture, réparties dans 6 groupes, ainsi que 88 équations, pour faciliter l'application du raffinement à l'intérieur des définitions de comportements et de leurs processus associés.

Le premier groupe de règles de réécriture s'intéresse, comme pour le patron précédent, à la partie compositionnelle de la spécification, c'est-à-dire le bloc de déclaration de la composition, qui débute par le mot-clé "compose". Cependant, il est cette fois composé de 32 règles, ce qui

témoigne d'un nombre plus important qu'auparavant de détails à prendre en compte pour le traitement.

Le travail sur cette partie consiste à repérer un port de composant et le port de connecteur correspondant, pour effacer pas à pas un de leurs canaux de communication, mais en en conservant une trace dans une nouvelle partie spécifique "channels" ; au fur et à mesure de l'application de règles, les canaux sont donc sortis de la partie composition pour être déclarés tous ensemble à l'écart. Lorsqu'une déclaration de port se retrouve sans canal, elle est retirée à la fois de la déclaration du composant et de celle du connecteur ; quand un composant ou un connecteur se retrouve avec une liste vide de ports, celle-ci est effacée à son tour.

Ainsi, les différents cas à prendre en considération, en plus de savoir si l'architecture est basée sur une décomposition ou pas, concernent la présence ou pas, au moment de choisir laquelle de ces règles doit être appliquée, de la partie "channels" (après l'exécution d'au moins une de ces règles), d'autres canaux de communication restant dans la définition du port considéré, et de ports supplémentaires à traiter dans la déclaration du composant ou du connecteur. Les autres groupes de règles, conçus pour être appliqués uniquement après le premier, supposent que cette partie de déclaration des canaux de communication a déjà été formée.

La figure IV.19 regroupe les différentes règles de réécriture de ce premier groupe qui sont appliquées dans l'étude de cas, sur l'architecture de la figure IV.17.

```
rl [transfo2.1.11121] : *** sans decompose, sans channels au départ, avec plusieurs paramètres
    et 1 port
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] ||
    CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `:
    CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[
    PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `} .

rl [transfo2.1.12121] : *** sans decompose, channels au départ, avec plusieurs paramètres et 1
    port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] ||
    CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `:
    CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[
    PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `} .
```

```
rl [transfo2.1.12221] : *** sans decompose, channels au départ, avec 1 paramètre et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `:
    CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `,
    LTPORTS2 `] || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `:
    CONNECTORTYPE1 `[ LTPORTS2 `] || AEDECL `} .

rl [transfo2.1.12122] : *** sans decompose, channels au départ, avec plusieurs paramètres et 1
    port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] ||
    CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `:
    CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[
    PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || AEDECL `} .

rl [transfo2.1.12222] : *** sans decompose, channels au départ, avec 1 paramètre et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `:
    CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `]
    || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `:
    CONNECTORTYPE1 || AEDECL `} .
```

Figure IV.19. : règles de réécriture du deuxième patron de transformation traitant la partie composition dans l'étude de cas

Le deuxième groupe de règles de transformation traite plus spécifiquement les définitions de types de connecteurs. La première règle est appliquée pour enlever la plupart des déclarations de types de ports, ainsi que les déclarations de port de l'entête. La seconde achève le travail en traitant le dernier port.

Le troisième groupe agit de même sur les définitions de types de composants. Il y a néanmoins une règle de plus pour traiter le cas où le composant, comme c'est le cas dans l'exemple, ne comporte en tout et pour tout qu'un unique port.

La gestion des définitions des types de comportements de connecteurs nécessite 8 règles ; celles-ci constituent le quatrième ensemble et diffèrent sur le fait qu'il reste des déclarations de ports à ôter, que le type de comportement utilise des processus ou des variables locales. Néanmoins, des

équations sont utilisées pour achever la suppression des références aux types de ports dans les corps des définitions des comportements et leurs processus.

Le cinquième groupe de règles de réécriture comporte deux fois plus d'éléments, du fait qu'il manipule les définitions de types de comportements de composants. En effet, ceux-ci peuvent avoir en plus des définitions d'opérations internes ; ces opérations, qui ne peuvent pas être rencontrées dans les comportements de connecteurs, représentent des fonctionnalités ou des unités de calcul sans besoin de communication avec l'environnement extérieur du composant, et elles sont donc appelées au cours de l'exécution du comportement du composant intéressé. La mise en œuvre d'équations vient, comme précédemment, assurer l'application de la transformation à l'intérieur même des processus comportementaux.

Enfin, le sixième et dernier groupe se réduit à une règle de réécriture unique qui vient supprimer les définitions de types de ports devenues superflues.

Les règles de réécriture, parmi ces cinq groupes, qui sont utilisées sur l'exemple, sont récapitulées dans la figure IV.20 (sans oublier qu'elles sont complétées par des équations pour supprimer les déclarations de types de ports à l'intérieur des comportements).

```
rl [transfo2.2.1] :
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] `, LTPORTS1 `] `{ port PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] || PORTDECL1 || behaviour CONNECTORBEHAVIOUR1 `:
    CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `]
    `, LTPORTS1 `] `}
=>
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `[ LTPORTS1 `] `{ port PORT1 || PORTDECL1 ||
    behaviour CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[
    LTPORTS1 `] `} .

rl [transfo2.2.2] :
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] `] `{ port PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] ||
    PORTDECL1 || behaviour CONNECTORBEHAVIOUR1 `:
    CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `]
    `] `}
=>
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `{ port PORT1 || PORTDECL1 || behaviour
    CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `} .

rl [transfo2.3.3] :
channels `{ CLIST `}
ARCHI
define component type COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[
    PORTTYPEPARAM `] `] `{ port PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] ||
    behaviour COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `[ PORT1
    `: PORTTYPE1 `[ PORTTYPEPARAM `] `] `}
=>
```

```
channels `{ CLIST `}
ARCHI
define component type COMPONENTTYPE1 `{ port PORT1 || behaviour
    COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `} .

rl [transfo2.4.121] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `{ LTPARAM1 `,
    CONNECTORBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= CONNBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{
    LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 `[`-
    PORT1 `-`] `} .

rl [transfo2.4.221] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] `{ LTPARAM1 `,
    CONNECTORBEHAVIOURTYPE1 `[ PORT1 `] `= CONNBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `{ LTPARAM1 `,
    CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2211] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] `{ LTPARAM1 `, LOPSPEC1 `,
    COMPONENTBEHAVIOURTYPE1 `[ PORT1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `,
    LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `,
    LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2221] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `:
    PORTTYPE1 `[ PORTTYPEPARAM `] `] `{ LTPARAM1 `, LOPSPEC1 `,
    COMPONENTBEHAVIOURTYPE1 `[ PORT1 `] `= COMPBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `,
    LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `} .
```

```
rl [transfo2.6] :
channels `{ CLIST `}
ARCHI
define port type PORTTYPE1 `[ LTCHANNELS1 `] `{ PORTSPEC1 `}
=>
channels `{ CLIST `}
ARCHI .
```

Figure IV.20. : règles de réécriture du deuxième patron de transformation, du deuxième au sixième groupe, appliquées à l'étude de cas

Une fois toutes ces règles appliquées, et les équations de ce deuxième patron de raffinement de β-SPACE prises en considération, l'architecture intermédiaire de la figure IV.17 est transformée pour donner la spécification de la figure IV.21.

```
channels { demande_vérification @ envoyer: [Module], demande_vérification @ recevoir:
    [Module], fournit_vérification @ recevoir: [Module], fournit_vérification @ envoyer:
    [Module]
}

define behaviour component type Ecriture {
module: Module, retour: Module,
faire_initialisation[out[Module]] { ... },
faire_écriture1[inout[Module]] { ... },
Ecriture =
    ( (faire_initialisation[module] ¤ demande_vérification @ envoyer <module>
    ¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[retour]) + $),
processus_écriture[module: Module] =
    ( (faire_écriture1[module] ¤ demande_vérification @ envoyer <module>
    ¤ demande_vérification @ recevoir (retour) ¤ processus_écriture[retour]) + $)
}

define behaviour component type Vérification {
module: Module, retour: Module,
faire_vérification1[in[Module], out[Module]] { ... },
Vérification =
    ( (fournit_vérification @ recevoir (module) ¤ faire_vérification1[module, retour]
    ¤ fournit_vérification @ envoyer <retour> ¤ Vérification) + $)
}

define behaviour connector type Aiguillage {
module: Module, retour: Module,
Aiguillage =
    ( (demande_vérification @ envoyer (module) ¤ fournit_vérification @ recevoir<module>
    ¤ fournit_vérification @ envoyer (retour) ¤ demande_vérification @ recevoir <retour>
    ¤ Aiguillage) + $)
}

define component type Ecrivain1 {
port demande_vérification                                                    ||
behaviour écriture: Ecriture
}
```

```
define component type Vérificateur1 {
port fournit_vérification                                    ||
behaviour vérification: Vérification
}

define connector type Aiguillage1 {
port demande_vérification                                    ||
port fournit_vérification                                    ||
behaviour aiguillage: Aiguillage
}

compose E1A1V1 {
    E1: Ecrivain1                                            ||
    A1: Aiguillage1                                          ||
    V1: Vérificateur1
}
```

Figure IV.21. : architecture canonique de l'étude de cas après application du deuxième patron de raffinement

L'architecture obtenue à l'issue de l'application des deux premiers patrons de transformation de β-SPACE n'est plus tributaire des détails propres à une spécification en π-SPACE. En effet, le raffinement opéré jusqu'à présent ne s'est intéressé qu'aux ports des composants et connecteurs, dont la gestion était spécifique à ce langage de description d'architecture logicielle. Les définitions de types de ports, établies séparément des autres éléments architecturaux pour permettre davantage de réutilisation, ne sont plus intéressantes dès lors que seuls les aspects statiques sont pris en compte. Dans ce cas de figure, les attachements entre ports n'évolueront plus, pas plus que le nombre de communications que nécessiteront les calculs des différents composants, via des connecteurs.

La spécification élaborée à partir de l'architecture abstraite originale après le raffinement mené par les deux patrons précédents, ne comporte plus que les éléments constitutifs de n'importe quel autre langage de description d'architecture : c'est une composition de composants et de connecteurs, les communications étant établies entre un composant et un connecteur par le partage d'un port. Les seules constructions moins standard toujours présentes concernent la base sémantique du langage de description d'architecture de départ pour la description des comportements, à savoir des processus basés sur le π-calcul, des déclarations d'opérations internes et des canaux de communications pour l'interaction des processus entre eux. Il faut donc s'attendre à rencontrer les mêmes structures architecturales pour tout autre langage de description d'architecture abstraite, qui bénéficierait d'un pouvoir expressif équivalent (pour des systèmes statiques tout au moins).

Puisque l'architecture à ce niveau d'abstraction contient exactement les structures architecturales de base désirées pour toute description architecturale abstraite originale, cette spécification architecturale résultant des deux premiers patrons de transformation de notre approche β-SPACE peut être appelée architecture canonique (cf. paragraphe 4.7).

Les opérations de transformations qui lui seront appliquées par la suite ne concerneront donc plus le langage abstrait de description architecturale de départ, mais les aspects spécifiques au langage concret cible.

5.4. Troisième patron de transformation

L'objet du troisième patron de transformation de β-SPACE est le raffinement de l'architecture canonique pour obtenir la spécification concrète en machines abstraites de la méthode B (cf. paragraphes 4.1 et 4.8). Cette étape s'avère bien plus complexe que les précédentes, du fait qu'il ne s'agit plus seulement de modifier l'agencement des différentes parties de la spécification, mais aussi de passer des concepts de composants et connecteurs avec des processus expliquant les interactions

entre eux, à une hiérarchies de machines abstraites où les communications sont bien plus limitées et strictement régentées.

Le contrôle des événements à l'intérieur du système n'est pas du tout géré de la même façon par les deux méthodologies (décrites dans le chapitre 3). En effet, dans une architecture logicielle, une action élémentaire est typiquement un passage de message entre un composant et un connecteur (une synchronisation) ou bien un appel à une opération interne. Le contrôle est assuré au niveau sémantique par le formalisme sous-jacent, l'algèbre de processus π-calcul. D'autre part, une machine abstraite B contient des opérations agissant sur des variables locales ; une opération ne peut pas faire appel à une opération (que ce soit la même ou une autre) de la même machine abstraite, et si elle fait appel à une opération d'une machine extérieure, alors il ne pourra pas y avoir d'appel dans l'autre sens. De plus, les opérations d'une machine abstraite donnée ne peuvent pas être manipulées depuis plusieurs autres machines abstraites. Par conséquent, la structure en termes de composants et connecteurs, conservée jusqu'à présent, doit être abandonnée et modifiée pour répondre aux astreintes du langage de spécification concret.

Ce troisième patron de raffinement est également structuré en différents groupes de règles de réécriture, chacun s'attachant à une transformation bien spécifique. Le premier groupe de règles, par exemple, contient 8 règles pour traiter un type de composant et son type de comportement correspondant. Les différents cas pris en compte correspondent à la présence, dans la définition du type de comportement de composant, de déclarations d'opérations internes, de processus supplémentaires et de variables locales. Chacune de ces règles isole la définition d'un type de composant avec la définition de son type de comportement pour commencer à les transformer en 2 machines abstraites : une première machine abstraite est amenée à regrouper tout ce qui concerne les différents états des processus définissant le comportement du composant ainsi que les actions permettant de passer d'un de ces états à un autre ; la seconde machine abstraite correspondra plus précisément à la gestion du contrôle de ces états et de ces actions pour obtenir le comportement attendu. Toutes les transformations nécessaires seront structurées en plusieurs passes de réécriture ; aussi, une partie du travail, celle qui concerne plus précisément le raffinement des processus en π-calcul, est-elle laissée en attente, en prévision de l'application ultérieure d'autres règles de réécriture et équations.

Les règles de ce premier groupe utilisées dans le cadre de l'étude de cas, pour raffiner l'architecture canonique de la figure IV.21, sont présentées dans la figure IV.22. La première est appliquée au composant "Ecrivain1" tandis que la seconde est employée pour traiter le composant "Vérificateur1".

```
rl [transfo3.1.211] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `,
    LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 `}
define component type COMPONENTTYPE1 `{ PORTDECL1 || behaviour
    COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `}
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ( 'Actions_ , COMPONENTBEHAVIOURTYPE1 )
SETS conc ( 'ETATS_ , COMPONENTBEHAVIOURTYPE1 ) `= `{ index ( conc ( conc ( 'etat_
    , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1
    0 +`> `} `;
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES conc ( 'var_etats_ , COMPONENTBEHAVIOURTYPE1 )
INVARIANT conc ( 'var_etats_ , COMPONENTBEHAVIOURTYPE1 ) `: conc ( 'ETATS_ ,
    COMPONENTBEHAVIOURTYPE1 )
INITIALISATION conc ( 'var_etats_ , COMPONENTBEHAVIOURTYPE1 ) `:`= index ( conc
    ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 0 )
```

```
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
    LCOMPPROC1 **`]
END
MACHINE conc ( 'Comportement_ , COMPONENTTYPE1 )
EXTENDS conc ( 'Actions_ , COMPONENTBEHAVIOURTYPE1 )
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.1.221] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `,
    LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `}
define component type COMPONENTTYPE1 `{ PORTDECL1 || behaviour
    COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `}
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ( 'Actions_ , COMPONENTBEHAVIOURTYPE1 )
SETS conc ( 'ETATS_ , COMPONENTBEHAVIOURTYPE1 ) `= `{ index ( conc ( conc ( 'etat_
    , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1
    0 +`> `} `;
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES conc ( 'var_etats_ , COMPONENTBEHAVIOURTYPE1 )
INVARIANT conc ( 'var_etats_ , COMPONENTBEHAVIOURTYPE1 ) `: conc ( 'ETATS_ ,
    COMPONENTBEHAVIOURTYPE1 )
INITIALISATION conc ( 'var_etats_ , COMPONENTBEHAVIOURTYPE1 ) `:`= index ( conc
    ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 0 )
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE conc ( 'Comportement_ , COMPONENTTYPE1 )
EXTENDS conc ( 'Actions_ , COMPONENTBEHAVIOURTYPE1 )
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .
```

Figure IV.22. : règles du premier groupe de règles de réécriture du troisième patron de transformation appliquées à l'étude de cas

Le deuxième groupe de règles de réécriture assure une transformation similaire pour les connecteurs. Il n'y a donc plus que 4 règles (deux fois moins que dans le cas précédent étant donné qu'un comportement de connecteur n'utilise pas d'opération interne).

```
rl [transfo3.2.21] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `{ LTPARAM1 `,
    CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 `}
define connector type CONNECTORTYPE1 `{ PORTDECL1 || behaviour
    CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `}
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ( 'Comportement_ , CONNECTORTYPE1 )
SETS CONNECTEURS `= `{ CONNECTORBEHAVIOURTYPE1 `} `;
ETATS_CONNECTEURS `= `{ index ( conc ( conc ( 'etat_ ,
    CONNECTORBEHAVIOURTYPE1 ) , '_ ) , 0 ) `<+ CONNECTORBEHAVIOURTYPE1 0
    +`> `} `;
`[+`- LTPARAM1 `-+`]
EXTENDS
VARIABLES var_etats_connecteurs
INVARIANT var_etats_connecteurs `: CONNECTEURS --> ETATS_CONNECTEURS
INITIALISATION var_etats_connecteurs `:`= `{ CONNECTORBEHAVIOURTYPE1 |`->`
    index ( conc ( conc ( 'etat_ , CONNECTORBEHAVIOURTYPE1 ) , '_ ) , 0 ) `}
OPERATIONS
CONNECTORTYPE1 `= `[`[+ CONNBEHA1 +`]`]
END .
```

Figure IV.23. : règle du deuxième groupe de règles de réécriture du troisième patron de transformation appliquée à l'étude de cas

Cependant, une seule machine abstraite est générée à partir de la définition du type de connecteur et de son type de comportement. En effet, les connecteurs n'ont pas d'opération interne et les communications sont toutes liées à des composants. C'est pourquoi cette étape de raffinement crée une seule machine abstraite par connecteur ; elle sert à regrouper les informations sur les différents états possibles du connecteur, en attendant que les liens avec les composants soient traités par d'autres groupes de règles. La règle utilisée pour l'exemple, pour un connecteur dont le comportement est défini avec des variables locales mais sans processus supplémentaire, est montrée dans la figure IV.23.

24 règles de réécriture composent le troisième groupe. Celui-ci transforme les anciennes déclarations de variables locales d'un composant pour déclarer les types de données adéquates dans la machine abstraite stockant les actions et états du composant. Tout d'abord, il est possible de différencier le cas où aucun autre type n'a encore été ajouté ; si au moins un autre type est déjà présent, alors il se peut que celui qui est à traiter ne soit pas encore là, sinon le traitement ne changera pas le nombre de types. En plus de ces trois possibilités, il faut envisager le cas où d'autres types restent à traiter et, de la même façon que pour le premier groupe de règles, le type de comportement de composant – et désormais la machine abstraite des actions du composant - peut comporter des opérations ou des processus. Par ailleurs, un prédicat supplémentaire "nouveau" vient compléter ces règles pour déterminer si, parmi les types de la machine abstraite des actions du composant, se trouve le type traité.

Les règles de ce groupe utilisées pour l'exemple illustrent la figure IV.24. Les deux premières sont appliquées à "Ecrivain1" et les deux autres à "Vérificateur1".

```
rl [transfo3.3.1121] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
    LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
    LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

crl [transfo3.3.2221] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
```

```
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
    LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
    LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
    `) .

rl [transfo3.3.1122] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
```

```
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

crl [transfo3.3.2222] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
```

```
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
    `) .
```

Figure IV.24. : règles du troisième groupe de règles de réécriture du troisième patron de transformation appliquées à l'étude de cas

Le quatrième groupe de règles de réécriture continue à traiter les machines abstraites représentant les composants. Ces règles s'intéressent plus précisément à la transformation de la première construction à interpréter dans la description du comportement d'un composant. Les différentes possibilités dépendent de la syntaxe de ces comportements en π-SPACE, basée sur le π-calcul. En tout, il y a jusqu'à vingt constructions qui peuvent être envisagées. Toutes ne peuvent pas être rencontrées, car elles dépendent des autres éléments se trouvant dans la description du comportement de composant : opérations internes, processus supplémentaires ou variables locales. Au final, 192 règles de transformation sont nécessaires pour couvrir tous les différents cas.

La figure IV.25 montre un exemple de règle de ce groupe. Il s'agit en fait de la règle simple appliquée dans le cas du composant "Ecrivain1", pour traiter les parenthèses encadrant la définition principale du comportement "Ecriture". Une règle similaire, où n'apparaîtrait pas de déclaration de processus supplémentaire, est utilisée pour le comportement "Vérification" du composant "Vérificateur1".

Cette règle se contente de supprimer la paire de parenthèses qui viennent entourer le processus à traiter, car elles n'ont d'autre but que de faciliter la compréhension à la lecture du comportement. Par conséquent, une autre règle de ce groupe sera amenée à être appliquée sur l'exemple par la suite.

Les différents autres cas à envisager, en tête de la définition d'un comportement, sont respectivement : une mise en séquence de deux comportements, un appel d'opération interne (avec ou sans paramètres), une communication (en émission ou réception) avec traitement des canaux de communication déclarés dans la partie "channels", un appel de processus (avec ou sans paramètre, et éventuellement un seul processus supplémentaire), un choix indéterministe, un choix gardé (par un test basé soit sur une différence, soit un inférieur large ou strict, soit un supérieur large ou strict, soit une égalité), une mise en parallèle de deux comportements, un appel récursif au comportement, un caractère spécial de terminaison avec succès.

```
rl [transfo3.4.1.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
```

```
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `)
    `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `( COMPBEHA1 `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
    LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .
```

Figure IV.25. : exemple de règle du quatrième groupe de règles de réécriture du troisième patron de transformation, appliquée à l'étude de cas

Le cinquième groupe de règles de réécriture de ce patron de transformation de β-SPACE se situe dans le cas où le comportement traité par une règle du groupe précédent se résumait à un appel de processus avec paramètres (éventuellement après avoir rencontré des parenthèses, des mises en séquence, en parallèle ou un choix).

L'objectif de ces règles est de déclarer les paramètres utilisés dans les appels des processus, à l'intérieur de la machine abstraite correspondant au comportement étudié. Pour cela, des variables sont déclarées, si elles n'existaient pas encore, pour la variable utilisée dans le comportement et celle qui lui correspond dans le processus appelé. De ce fait, des informations peuvent ainsi être ajoutées conjointement dans les variables de la machine abstraite, ainsi que des invariants pour leur affecter leur type (défini dans la machine abstraite des actions et états du composant) et des initialisations dans le domaine de ces types de données ; une affectation est également mise en parallèle de l'appel du processus, dans l'opération décrivant le contrôle des actions du processus, afin de décrire que la variable correspondant au paramètre formel de la machine abstraite prend pour valeur le contenu de la variable utilisée dans l'appel du processus.

Ce groupe ne traite donc qu'un seul genre de donnée. Cependant, 144 règles sont nécessaires pour couvrir tous les cas possibles. En effet, il faut d'abord distinguer le cas où le nom du paramètre formel du processus diffère du nom de la variable utilisée dans le processus, du celui où ils sont identiques. Ensuite, le paramètre considéré peut être le dernier à traiter ou non. De plus, il pourrait

n'y avoir qu'un seul type défini pour tout le composant (et son comportement) ; dans ce cas, il est possible qu'il n'y ait qu'une seule variable pour le composant également. Par ailleurs, il convient à nouveau de vérifier si le composant contient des définitions d'opérations internes et d'autres processus supplémentaires. Enfin, des variables ont déjà pu être définies dans la machine abstraite contrôlant le composant, soit par ce jeu de règles, auquel cas il y a aussi des affectations présentes, soit sans affectations.

Un sixième groupe de règles de réécriture s'intéresse sensiblement à la même chose, mais pour les opérations internes, c'est-à-dire qu'un appel à une opération interne a été traité comme premier élément du comportement de composant. Comme pour le groupe précédent, les paramètres des opérations sont l'objet de la transformation. Le profil de l'opération B correspondant à l'opération interne du composant est mis à jour dans la machine abstraite des actions et états du composant, pendant que la variable du comportement, correspondant au paramètre considéré à ce moment, est déclarée, avec son invariant et son initialisation, dans la machine abstraite du contrôle de ce composant.

Cette fois, pas moins de 664 règles sont nécessaires pour décrire toutes les applications différentes qu'il se peut trouver pour cette seule tâche. En premier lieu, les paramètres peuvent avoir définis en entrée seulement (avec le mot-clé "in"), en sortie d'opération seulement (avec le mot-clé "ou") ou à la fois en entrée et sortie (avec le mot-clé "inout"). Il peut aussi rester d'autres paramètres de l'opération à traiter, ou bien aucun. En outre, s'il n'y a qu'un seul type défini dans la machine abstraite, il se peut que cela soit dû au fait qu'il n'y ait eu qu'une seule variable dans la description du comportement du composant. De même, le comportement ne possédait peut être d'autres opérations internes ou non, ou encore des processus supplémentaires.

Le nombre plus élevé de règles dans ce groupe provient du fait qu'à partir du deuxième paramètre d'opération interne à traduire, il faut envisager, tout en considérant les mêmes cas, d'avoir eu auparavant uniquement des paramètres en entrée seulement, que des paramètres en sortie seulement, que des paramètres en entrée et sortie simultanées, ou bien encore des mélanges de deux ou trois de ces catégories. Certains cas sont cependant à proscrire : par exemple, un paramètre déjà défini en entrée seulement ne pourra pas être retrouvé en sortie seulement pour la même opération interne, alors qu'on pourrait le retrouver plusieurs fois en entrée. Un prédicat "nouvelle_var" est nécessaire pour tester l'existence de la variable du comportement parmi les variables déjà déclarées dans la machine abstraite du contrôle du composant.

D'autres règles de réécriture, nombreuses, doivent encore compléter ce patron de transformation. Il faut en effet traiter des actions suivantes des comportements de composants, mais aussi des actions de leurs processus.

De plus, le traitement des comportements de connecteurs est plus simple. D'une part, il n'y a pas d'opération interne. D'autre part, tous se retrouvent finalement dans une unique machine abstraite regroupant la gestion des états de tous les connecteurs du système, en fonction des communications avec les machines abstraites de composants.

Enfin, la partie composition de l'architecture est facile à transformer puisqu'il ne reste plus qu'à mettre en parallèle les opérations des machines abstraites assurant les contrôles de tous les connecteurs simultanément, et de chaque composant individuellement.

La figure IV.26 présente le résultat de la transformation de l'architecture canonique de l'étude de cas après application du troisième patron de transformation de β-SPACE, c'est-à-dire le résultat du raffinement de l'architecture abstraite de départ.

```
MACHINE Actions_Ecriture
SETS
    ETATS_Ecriture = { etat_Ecriture_0 , etat_Ecriture_1 , etat_Ecriture_2 ,
    etat_processus_écriture_0 , etat_processus_écriture_1 , etat_processus_écriture_2 } ;
    Module_Ecriture
VARIABLES
    var_etats_Ecriture , demande_vérification_envoyer , demande_vérification_recevoir
```

```
INVARIANT
    var_etats_Ecriture : ETATS_Ecriture &
    demande_vérification_envoyer : Module_Ecriture &
    demande_vérification_recevoir : Module_Ecriture
INITIALISATION
    var_etats_Ecriture := etat_Ecriture_0 ||
    demande_vérification_envoyer :: Module_Ecriture ||
    demande_vérification_recevoir :: Module_Ecriture
OPERATIONS
    outModule <-- faire_initialisation =
    PRE         var_etats_Ecriture : { etat_Ecriture_0 }
    THEN
        IF var_etats_Ecriture = etat_Ecriture_0
                THEN var_etats_Ecriture := etat_Ecriture_1
        END ||
        outModule :: Module_Ecriture
    END ;

    demande_vérification_envoyer_in ( module ) =
    PRE         var_etats_Ecriture : { etat_Ecriture_1 , etat_processus_écriture_1 } &
        module : Module_Ecriture
    THEN
        IF var_etats_Ecriture = etat_Ecriture_1
                THEN var_etats_Ecriture := etat_Ecriture_2
        ELSIF var_etats_Ecriture = etat_processus_écriture_1
                THEN var_etats_Ecriture := etat_processus_écriture_2
        END ||
        demande_vérification_envoyer := module
    END ;

    retour <-- demande_vérification_recevoir_out =
    PRE         var_etats_Ecriture : { etat_Ecriture_2 , etat_processus_écriture_2 }
    THEN
        IF var_etats_Ecriture = etat_Ecriture_2
                THEN var_etats_Ecriture := etat_processus_écriture_0
        ELSIF var_etats_Ecriture = etat_processus_écriture_2
                THEN var_etats_Ecriture := etat_processus_écriture_0
        END ||
        retour := demande_vérification_recevoir
    END ;

    outModule <-- faire_écriture1 ( inModule ) =
    PRE         var_etats_Ecriture : { etat_processus_écriture_0 } &
        inModule : Module_Ecriture
    THEN
        IF var_etats_Ecriture = etat_processus_écriture_0
                THEN var_etats_Ecriture := etat_processus_écriture_1
        END ||
        outModule :: Module_Ecriture
    END ;

    module <-- demande_vérification_envoyer_out =
    BEGIN
        module := demande_vérification_envoyer
    END ;
```

```
    demande_vérification_recevoir_in ( retour ) =
    PRE         retour : Module_Ecriture
    THEN
        demande_vérification_recevoir := retour
    END
END


MACHINE Comportement_Ecrivain1
EXTENDS Actions_Ecriture
VARIABLES
    Ecriture_module , Ecriture_retour , Ecriture_module_inout
INVARIANT
    Ecriture_module : Module_Ecriture &
    Ecriture_retour : Module_Ecriture &
    Ecriture_module_inout : Module_Ecriture
INITIALISATION
    Ecriture_module :: Module_Ecriture ||
    Ecriture_retour :: Module_Ecriture ||
    Ecriture_module_inout :: Module_Ecriture
OPERATIONS
    Ecrivain1 =
    IF var_etats_Ecriture = etat_Ecriture_0
        THEN  CHOICE Ecriture_module <-- faire_initialisation
                OR skip                                                    /* + $ */
                END
    ELSIF var_etats_Ecriture = etat_Ecriture_1
        THEN demande_vérification_envoyer_in ( Ecriture_module )
    ELSIF var_etats_Ecriture = etat_Ecriture_2
        THEN Ecriture_retour <-- demande_vérification_recevoir_out
    ELSIF var_etats_Ecriture = etat_processus_écriture_0
        THEN  CHOICE Ecriture_module_inout <-- faire_écriture1 ( Ecriture_module ) ||
                        Ecriture_module := Ecriture_module_inout
                OR skip
                END
    ELSIF var_etats_Ecriture = etat_processus_écriture_1
        THEN demande_vérification_envoyer_in ( Ecriture_module )
    ELSIF var_etats_Ecriture = etat_processus_écriture_2
        THEN Ecriture_retour <-- demande_vérification_recevoir_out
    /* ; Ecrivain1 */
    END
END


MACHINE Actions_Vérification
SETS
    ETATS_Vérification = { etat_Vérification_0 , etat_Vérification_1 , etat_Vérification_2 } ;
    Module_Vérification
VARIABLES
    var_etats_Vérification , fournit_vérification_recevoir , fournit_vérification_envoyer
INVARIANT
    var_etats_Vérification : ETATS_Vérification &
    fournit_vérification_recevoir : Module_Vérification &
    fournit_vérification_envoyer : Module_Vérification
```

```
INITIALISATION
    var_etats_Vérification := etat_Vérification_0 ||
    fournit_vérification_recevoir :: Module_Vérification ||
    fournit_vérification_envoyer :: Module_Vérification
OPERATIONS
    module <-- fournit_vérification_recevoir_out =
    PRE         var_etats_Vérification : { etat_Vérification_0 }
    THEN
        IF var_etats_Vérification = etat_Vérification_0
                THEN var_etats_Vérification := etat_Vérification_1
        END ||
        module := fournit_vérification_recevoir
    END ;

    outModule <-- faire_vérification1 ( inModule ) =
    PRE         var_etats_Vérification : { etat_Vérification_1 } &
        inModule : Module_Vérification
    THEN
        IF var_etats_Vérification = etat_Vérification_1
                THEN var_etats_Vérification := etat_Vérification_2
        END ||
        outModule :: Module_Vérification
    END ;

    fournit_vérification_envoyer_in ( retour ) =
    PRE         var_etats_Vérification : { etat_Vérification_2 } &
        retour : Module_Vérification
    THEN
        IF var_etats_Vérification = etat_Vérification_2
                THEN var_etats_Vérification := etat_Vérification_0
        END ||
        fournit_vérification_envoyer := retour
    END ;

    fournit_vérification_recevoir_in ( module ) =
    PRE         module : Module_Vérification
    THEN
        fournit_vérification_recevoir := module
    END ;

    retour <-- fournit_vérification_envoyer_out =
    BEGIN
        retour := fournit_vérification_envoyer
    END
END


MACHINE Comportement_Vérificateur1
EXTENDS Actions_Vérification
VARIABLES
    Vérification_module , Vérification_retour
INVARIANT
    Vérification_module : Module_Vérification &
    Vérification_retour : Module_Vérification
```

```
INITIALISATION
    Vérification_module :: Module_Vérification ||
    Vérification_retour :: Module_Vérification
OPERATIONS
    Vérificateur1 =
    IF var_etats_Vérification = etat_Vérification_0
        THEN  CHOICE Vérification_module <-- fournit_vérification_recevoir_out
              OR skip                                          /* + $ */
              END
    ELSIF var_etats_Vérification = etat_Vérification_1
        THEN Vérification_retour <-- faire_vérification1 (Vérification_module )
    ELSIF var_etats_Vérification = etat_Vérification_2
        THEN fournit_vérification_envoyer_in ( Vérification_retour )
    /* ; Vérificateur1 */
    END
END


MACHINE Connecteurs
SETS
    CONNECTEURS = { Aiguillage } ;
    ETATS_CONNECTEURS = { etat_Aiguillage_0 , etat_Aiguillage_1 , etat_Aiguillage_2 ,
    etat_Aiguillage_3 } ;
    Module
EXTENDS Comportement_Ecrivain1 , Comportement_Vérificateur1
VARIABLES
    var_etats_connecteurs , mod_Ecriture , mod_Vérification , Aiguillage_module ,
    Aiguillage_retour , Aiguillage_module_Ecriture , Aiguillage_retour_Ecriture ,
    Aiguillage_module_Vérification , Aiguillage_retour_Vérification
INVARIANT
    var_etats_connecteurs : CONNECTEURS --> ETATS_CONNECTEURS &
    mod_Ecriture : Module >->> Module_Ecriture &
    mod_Vérification : Module >->> Module_Vérification &
    Aiguillage_module : Module &
    Aiguillage_retour : Module &
    Aiguillage_module_Ecriture : Module_Ecriture &
    Aiguillage_retour_Ecriture : Module_Ecriture &
    Aiguillage_module_Vérification : Module_Vérification &
    Aiguillage_retour_Vérification : Module_Vérification
INITIALISATION
    var_etats_connecteurs := { Aiguillage |-> etat_Aiguillage_0 } ||
    mod_Ecriture :: Module >->> Module_Ecriture ||
    mod_Vérification :: Module >->> Module_Vérification ||
    Aiguillage_module :: Module ||
    Aiguillage_retour :: Module ||
    Aiguillage_module_Ecriture :: Module_Ecriture ||
    Aiguillage_retour_Ecriture :: Module_Ecriture ||
    Aiguillage_module_Vérification :: Module_Vérification ||
    Aiguillage_retour_Vérification :: Module_Vérification
OPERATIONS
    Aiguillage1 =
    IF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_0
```

```
        THEN  CHOICE        Aiguillage_module_Ecriture <--
    demande_vérification_envoyer_out ||
                            Aiguillage_module := mod_Ecriture~
    (Aiguillage_module_Ecriture ) ||
                            var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_1
                OR skip                                          /* + $ */
                END
    ELSIF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_1
        THEN  Aiguillage_module_Vérification := mod_Vérification ( Aiguillage_module ) ||
                fournit_vérification_recevoir_in ( Aiguillage_module_Vérification ) ||
                var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_2
    ELSIF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_2
        THEN  Aiguillage_retour_Vérification <-- fournit_vérification_envoyer_out ||
                Aiguillage_retour := mod_Vérification~ ( Aiguillage_retour_Vérification ) ||
                var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_3
    ELSIF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_3
        THEN  Aiguillage_retour_Ecriture := mod_Ecriture ( Aiguillage_retour ) ||
                demande_vérification_recevoir_in ( Aiguillage_retour_Ecriture ) ||
                var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_0
    END
    /* ; Aiguillage1 */
END


MACHINE Configuration
INCLUDES Connecteurs
OPERATIONS
    E1A1V1 =
    BEGIN
        skip /* Ecrivain1 || Vérificateur1 || Aiguillage1 */
    END
END
```

Figure IV.26. : spécification concrète en machines abstraites B résultant de l'application du troisième patron de raffinement

5.5. Conclusion

Les figures IV.9, IV.17, IV.21 et IV.26 présentent les principales étapes de l'étude de cas, en partant de l'architecture abstraite en π-SPACE (figure IV.9), jusqu'à l'obtention de la spécification concrète en machines abstraites de la méthode B (figure IV.26).

Les deux premiers patrons de transformation de notre approche β-SPACE opèrent simplement une simplification de la description originale, correspondant à une diminution du pouvoir expressif pour revenir aux éléments architecturaux standards (configurations statiques composées de composants et de connecteurs, basées sur la correspondance de ports). Le résultat intermédiaire de la figure IV.17 montre la prise en compte des attachements entre composants et connecteurs à l'intérieur de ces derniers, avant d'arriver à une architecture canonique (figure IV.21) lorsque les attachements et la correction de l'architecture abstraite sont considérés aussi pour alléger les composants. Si l'architecture abstraite est supposée statique dès le départ, le comportement n'est en rien altéré par ces deux premiers patrons de transformation : il y a équivalence observationnelle entre l'architecture abstraite et l'architecture canonique. De plus, s'agissant toujours d'une description architecturale, des opérations de raffinement horizontal sont encore envisageables (cela induit cependant un changement de comportement).

Par contre, l'étape de raffinement suivante s'intéresse au passage d'un contrôle architectural à un autre très différent à base de machines abstraites de la méthode B. C'est la différence radicale entre les fondements sémantiques des formalismes abstraits et concrets qui induit les problèmes, nécessitant ainsi un grand nombre de règles de réécriture. Néanmoins, le troisième patron de transformation de β-SPACE est conçu pour respecter également le comportement architectural, même s'il devient plus difficile de l'interpréter dans le même espace que pour les spécifications des niveaux d'abstraction précédents.

L'étude de cas présentée dans ce chapitre ne contraint pas la solution proposée pour les patrons de raffinement. Ceux-ci sont indépendants de l'architecture initiale ; ils restent cependant étroitement liés au choix des langages de descriptions abstraites et concrètes, dans le cas présent π-SPACE et la méthode B. Pourtant, l'application montre qu'une étape intermédiaire, correspondant à une forme canonique de l'architecture, peut être dégagée : en effet, les patrons de transformation présentés ont tout d'abord dû prendre en compte les spécificités du langage de description d'architecture de départ pour éliminer les constructions superflues ; arrivés à ce niveau d'abstraction, ils se sont ensuite concentrés sur les aspects du formalisme concret à faire ressortir. Ce niveau intermédiaire permet de mettre en avant un langage intermédiaire de description d'architecture, canonique, un langage pivot qui pourrait servir de base pour des travaux ultérieurs visant à rendre l'approche plus générique (cf. paragraphe 4.7).

Chapitre 5 : Implémentation et validation

Chapitre 5 : Implémentation et validation

1. Introduction

Dans le chapitre précédent, nous avons introduit notre approche β-SPACE pour un raffinement vertical d'architecture logicielle (cf. chapitre 1, paragraphe 3), distincte de la décomposition usuelle, dans le but d'obtenir des machines abstraites B. Elle s'appuie principalement sur trois patrons de raffinement, composés de règles et équations de la logique de réécriture.

La figure IV.2 présentait le processus de raffinement tel que nous le proposons, séparant le raffinement horizontal d'architecture logicielle (sa décomposition) et le raffinement vertical (qui change véritablement le niveau d'abstraction sans modifier le comportement de la description du système). La figure V.1, qui en est extraite, illustre ce que nos patrons de raffinement apportent au langage de description d'architecture logicielle π-SPACE.

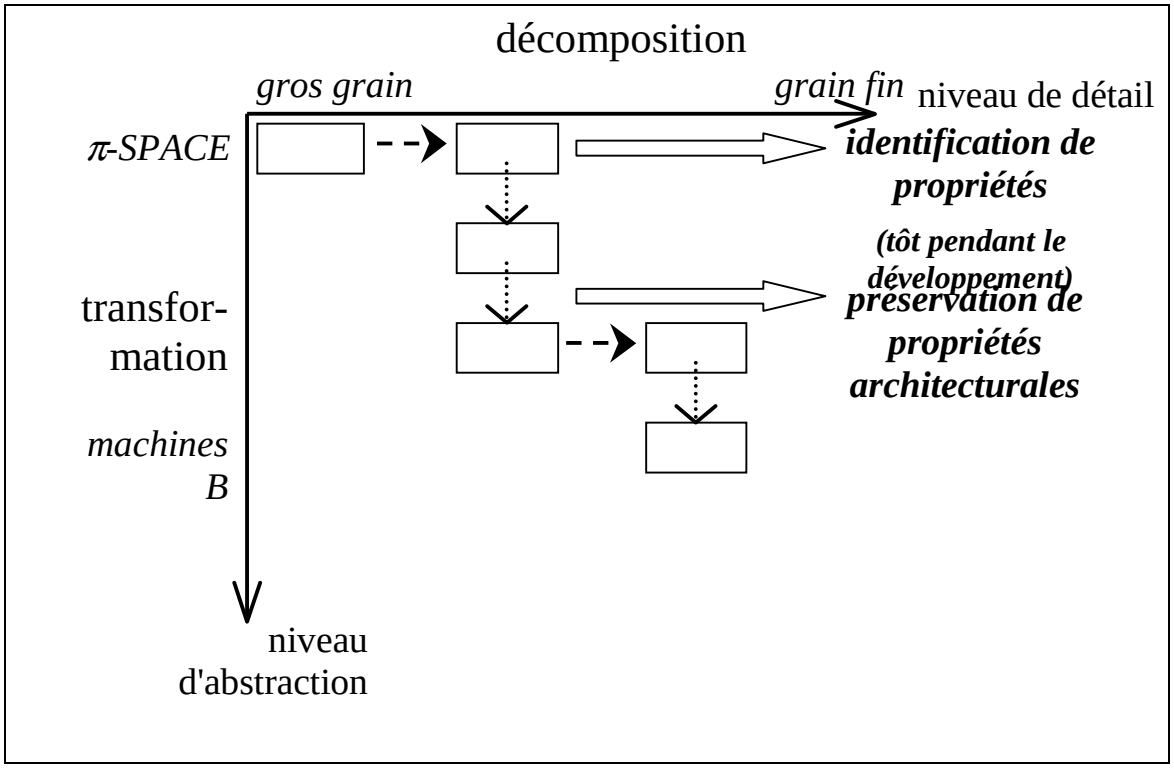

Figure V.1. : décomposition en π-SPACE, et transformation par nos patrons de raffinement

En effet, la phase de spécification formelle débute par la description de l'architecture du système logiciel en π-SPACE. Cela peut demander un travail plus ou moins important selon la nature du problème à traiter. Après d'éventuelles décompositions supportées par ce formalisme, nous pouvons nous engager dans une étape de raffinement vertical, en soumettant la description architecturale abstraite en π-SPACE à nos patrons de transformation. A l'issue de l'application des deux premiers, la description architecturale du système se trouve sous forme "canonique" : nous pourrions donc envisager de continuer à opérer des décompositions de l'architecture jusqu'à ce niveau d'abstraction. Tel ne pourra plus être le cas après le passage par le troisième et dernier patron de raffinement, car le résultat n'est plus une description architecturale du système, même si nous en avons préservé les propriétés : la spécification obtenue est un ensemble de machines abstraites B correctes.

La figure V.2 complète la précédente en y ajoutant le raffinement qui peut y faire suite dans le cadre de la méthode B (encadré en pointillé). Cette dernière permet de développer des modules plus concrets, dénotés par "refinement" et "implementation", tout en pouvant garantir la conservation des propriétés des machines abstraites grâce aux obligations de preuve. Par ailleurs, les outils commerciaux disponibles pour supporter la méthode B sont capables de générer de façon automatique le code correspondant aux implémentations B.

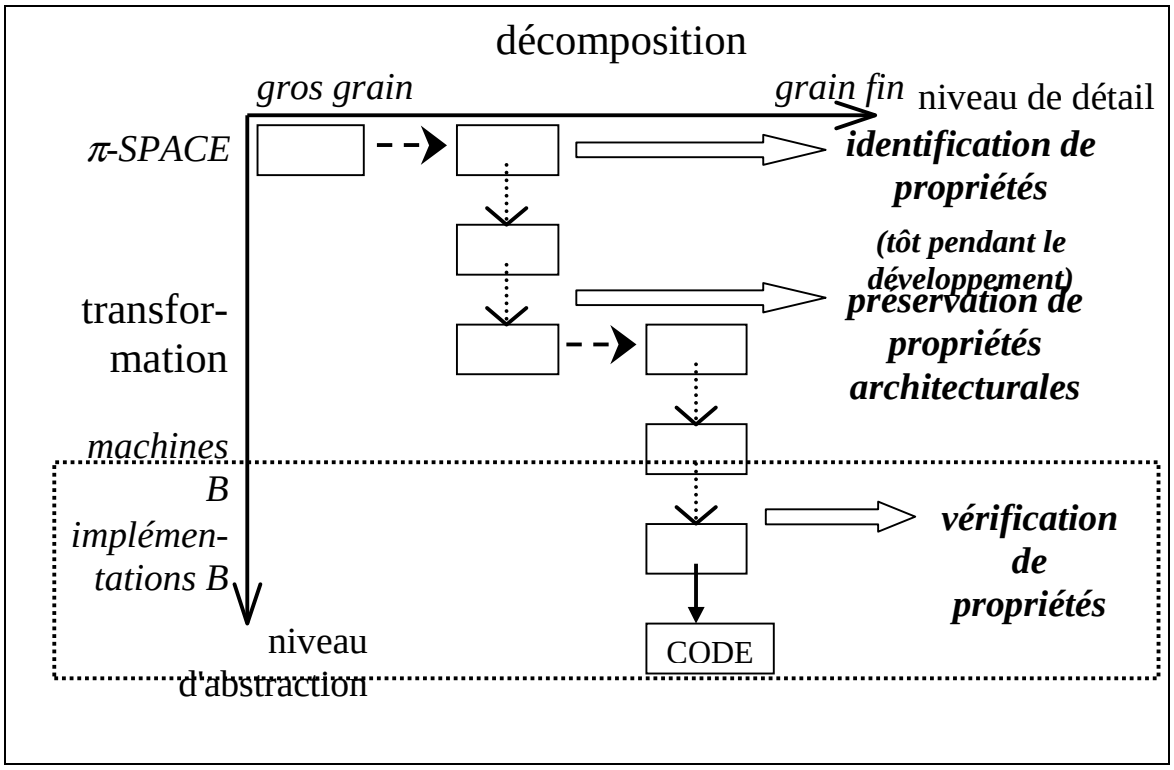

Figure V.2. : processus de raffinement architectural et de raffinement B

Dans le chapitre précédent, nous avons illustré l'utilisation de nos patrons de transformation sur une étude cas. Ces patrons de transformations, basés sur la logique de réécriture, sont corrects par construction : nous les avons définis dans le respect de la syntaxe de la logique de réécriture, en couvrant tous les cas à traiter. La validation des machines abstraites B obtenues peut néanmoins être vérifiée dans la méthode B.

Nous présentons, dans la suite de ce chapitre, les différents aspects de la mise en œuvre de notre méthode β-SPACE. Tout d'abord, nous examinerons plus précisément l'utilisation des patrons de transformation décrits dans la logique de réécriture. Nous montrerons par la suite comment les machines abstraites issues de notre processus de raffinement vertical peuvent être validées dans la méthode B.

2. Utilisation des patrons de transformation

La syntaxe utilisée est celle de la version de la logique de réécriture de l'outil Maude, un système de logique de réécriture (évoqué chapitre 4, paragraphe 4.2 et décrit dans [Clavel et al. 1999] et [Clavel et al. 2000]). Il en résulte que les spécifications de nos patrons de raffinement sont exécutables dans ce système. Cet outil est le seul, à notre connaissance, à supporter la logique de réécriture.

Il nous suffit donc d'importer nos patrons de transformation, sous la forme de modules systèmes (cf. chapitre précédent), pour pouvoir ensuite demander à l'outil de réécrire une description en

fonction de nos règles et équations. De cette façon, nous obtenons directement un prototype, gratuit par ailleurs : l'outil est disponible gratuitement, comme un prototype de recherche[3].

L'exemple de la figure V.3 montre un appel à la version 1.0.5 de l'outil Maude pour l'environnement Linux. La première ligne indique que nous nous trouvions dans un contexte d'exécution "bash", et que le répertoire courant se situe, dans la hiérarchie du système de fichiers, juste un niveau au dessus du programme exécutable "maude.linux". Celui-ci prend la main dans la fenêtre de commande et commence par rappeler quelques informations sur la version de Maude utilisée. Maude sert alors d'environnement pour interpréter des modules de la logique de réécriture. Ainsi, la commande "in" nous permet de charger le contenu du fichier "patron1.maude", par exemple, qui contiendrait, au format texte, le module système avec ses règles de réécriture et ses équations.

```
bash$ ./bin/maude.linux
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
        Maude version 1.0.5 built: Apr  5 2000 15:56:52
             Copyright 1997-2000 SRI International
                    Tue Jul 24 16:44:27 2001
Maude> in patron1.maude
```

Figure V.3. : exemple de chargement du premier patron de raffinement

Pour transformer de façon automatique la description architecturale abstraite, il faut introduire celle-ci dans l'environnement de logique de réécriture avec la commande "rew" suivie de la spécification du système. Celle-ci doit cependant être formatée spécialement à cet effet, en faisant précéder chaque identificateur qui ne fait pas partie du langage de spécification d'une apostrophe : de tels éléments n'apparaissent pas dans les patrons de raffinement et ne doivent pas être traités comme des mots-clés par Maude.

Le résultat de la réécriture est affiché directement dans la fenêtre de commande. Il est possible de le récupérer dans un fichier texte en redirigeant la sortie standard. Pour plus de facilité, il serait alors préférable de placer cette redirection dans un fichier de commande qui lance également l'environnement Maude sur les fichiers contenant les patrons de transformation et la spécification architecturale abstraite.

Si les trois patrons de transformation avaient été chargés avec "in", avant d'utiliser "rew" sur notre exemple de la figure IV.9 (avec les apostrophes en plus), nous aurions obtenu une spécification concrète correspondant au contenu de la figure IV.26, avec les apostrophes (sur les mêmes éléments qu'avant la réécriture). Les machines abstraites B peuvent apparaître dans un ordre quelconque : les liens hiérarchiques entre elles sont établis explicitement dans la spécification, et non pas dans l'enchaînement des modules B.

3. Validation des machines abstraites B obtenues

Les machines abstraites de la méthode B issues de notre processus de transformation d'architecture logicielle sont correctes, mais n'ont pas vocation à être le niveau final du développement. En réalité, notre intention est d'utiliser un outil tel que l'Atelier B[4] pour se servir du raffinement en B, de sorte à générer automatiquement le code du système.

Afin qu'elle puisse être utilisée par ce logiciel, notre description concrète doit être séparée en autant de fichiers que de machines abstraites. Chacun de ces fichiers doit avoir l'extension ".mch" (pour

[3] http://maude.csl.sri.com ou http://maude.cs.uiuc.edu

[4] http://www.atelierb.societe.com

machine abstraite), sans contenir d'apostrophe ni de caractère accentué. Par exemple, la figure V.4 présente la machine abstraite "Connecteurs", extraite de la figure IV.26 (sans les accents).

L'utilisation de l'Atelier B pour valider des machines nécessite de créer un nouveau projet, contenant ces machines abstraites. L'outil nous permet alors d'opérer de façon automatique la vérification de la syntaxe et des types, mais aussi de générer toutes les obligations de preuve. L'Atelier B dispose d'un prouveur automatique qui permet d'automatiser ces preuves. Ainsi, il n'est pas besoin de calculer toutes les preuves mathématiques à la main. Ce prouveur automatique dispose de plusieurs niveaux de complexité, ou de compétence, numérotés de 0 à 3, du plus facile au plus difficile – plus le niveau est élevé, plus l'algorithme correspondant est complexe, et donc son temps de traitement est plus élevé.

```
MACHINE Connecteurs
SETS
   CONNECTEURS = { Aiguillage } ;
   ETATS_CONNECTEURS = { etat_Aiguillage_0 , etat_Aiguillage_1 , etat_Aiguillage_2 ,
   etat_Aiguillage_3 } ;
   Module
EXTENDS Comportement_Ecrivain1 , Comportement_Verificateur1
VARIABLES
   var_etats_connecteurs , mod_Ecriture , mod_Verification , Aiguillage_module ,
   Aiguillage_retour , Aiguillage_module_Ecriture , Aiguillage_retour_Ecriture ,
   Aiguillage_module_Verification , Aiguillage_retour_Verification
INVARIANT
   var_etats_connecteurs : CONNECTEURS --> ETATS_CONNECTEURS &
   mod_Ecriture : Module >->> Module_Ecriture &
   mod_Verification : Module >->> Module_Verification &
   Aiguillage_module : Module &
   Aiguillage_retour : Module &
   Aiguillage_module_Ecriture : Module_Ecriture &
   Aiguillage_retour_Ecriture : Module_Ecriture &
   Aiguillage_module_Verification : Module_Verification &
   Aiguillage_retour_Verification : Module_Verification
INITIALISATION
   var_etats_connecteurs := { Aiguillage |-> etat_Aiguillage_0 } ||
   mod_Ecriture :: Module >->> Module_Ecriture ||
   mod_Verification :: Module >->> Module_Verification ||
   Aiguillage_module :: Module ||
   Aiguillage_retour :: Module ||
   Aiguillage_module_Ecriture :: Module_Ecriture ||
   Aiguillage_retour_Ecriture :: Module_Ecriture ||
   Aiguillage_module_Verification :: Module_Verification ||
   Aiguillage_retour_Verification :: Module_Verification
OPERATIONS
   Aiguillage1 =
   IF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_0
      THEN  CHOICE  Aiguillage_module_Ecriture <-- demande_verification_envoyer_out ||
                  Aiguillage_module := mod_Ecriture~ (Aiguillage_module_Ecriture ) ||
                  var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_1
        OR skip                                                    /* + $ */
        END
   ELSIF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_1
      THEN  Aiguillage_module_Verification := mod_Verification ( Aiguillage_module ) ||
        fournit_verification_recevoir_in ( Aiguillage_module_Verification ) ||
        var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_2
   ELSIF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_2
      THEN  Aiguillage_retour_Verification <-- fournit_verification_envoyer_out ||
        Aiguillage_retour := mod_Verification~ ( Aiguillage_retour_Verification ) ||
        var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_3
   ELSIF var_etats_connecteurs ( Aiguillage ) = etat_Aiguillage_3
      THEN  Aiguillage_retour_Ecriture := mod_Ecriture ( Aiguillage_retour ) ||
        demande_verification_recevoir_in ( Aiguillage_retour_Ecriture ) ||
        var_etats_connecteurs ( Aiguillage ) := etat_Aiguillage_0
   END
   /* ; Aiguillage1 */
END
```

Figure V.4. : exemple de machine abstraite validée par l'Atelier B, Connecteurs.mch

Les machines abstraites B générées par β-SPACE sont correctes par construction, mais l'Atelier B en demandera la vérification avant de pouvoir poursuivre le développement formel en B. La figure V.5 présente le résultat du calcul des obligations de preuve, à la "force 0", pour la machine abstraite

"Connecteurs.mch" de la figure V.4. Cet exemple, plus que les autres machines abstraites de l'exemple, est intéressant du fait que toutes les obligations de preuve ne sont pas établies. En effet, quatre obligations de preuve correspondant à l'extension de la machine abstraite "Comportement_Ecrivain1", et notamment son opération "Ecrivain1", ne sont pas encore démontrées.

```
Proving Connecteurs

  Proof pass 0, still 18 unproved PO

    clause Initialisation
   ++
    clause Aiguillage1
   -++-++-++-++
    clause Ecrivain1
   ++++

End of Proof
  InstanciatedConstraintsLemmas       Proved 0     Unproved 0
  Initialisation   Proved 2     Unproved 0
  Aiguillage1      Proved 8     Unproved 4
  Ecrivain1        Proved 4     Unproved 0
  faire_initialisation   Proved 0     Unproved 0
  demande_verification_envoyer_in     Proved 0     Unproved 0
  demande_verification_recevoir_out   Proved 0     Unproved 0
  faire_ecriture1  Proved 0     Unproved 0
  demande_verification_envoyer_out    Proved 0     Unproved 0
  demande_verification_recevoir_in    Proved 0     Unproved 0
  Verificateur1    Proved 0     Unproved 0
  fournit_verification_recevoir_out   Proved 0     Unproved 0
  faire_verification1    Proved 0     Unproved 0
  fournit_verification_envoyer_in     Proved 0     Unproved 0
  fournit_verification_recevoir_in    Proved 0     Unproved 0
  fournit_verification_envoyer_out    Proved 0     Unproved 0
TOTAL for Connecteurs    Proved 14    Unproved 4
```

Figure V.5. : exemple de machine abstraite validée par l'Atelier B, Connecteurs.mch

Bien que les quatre obligations de preuves qui nous posent problème sur cet exemple correspondent à l'inclusion d'une autre machine abstraite, nous pouvons vérifier qu'elles ne sont pas dues uniquement à celle-ci. En effet, l'Atelier B nous permet d'afficher le "statut du projet" (cf. figure V.6). Nous nous apercevons alors que les seules quatre obligations de preuves non prouvées (colonne "nUn") ont bien été générées pour la machine abstraite "Connecteurs".

```
Project status
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| COMPONENT                 | TC | POG | Obv | nPO | nUn | %Pr | B0C |  C  | Ada | C++ |
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Actions_Ecriture          | OK | OK  |  41 |   0 |   0 | 100 |  -  |     |     |     |
| Actions_Verification      | OK | OK  |  26 |   0 |   0 | 100 |  -  |     |     |     |
| Comportement_Ecrivain1    | OK | OK  |  76 |   4 |   0 | 100 |  -  |     |     |     |
| Comportement_Verificateur1 | OK | OK  |  37 |   0 |   0 | 100 |  -  |     |     |     |
| Configuration             | OK | OK  |   3 |   0 |   0 | 100 |  -  |     |     |     |
| Connecteurs               | OK | OK  | 501 |  18 |   4 |  77 |  -  |     |     |     |
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| TOTAL                     | OK | OK  | 684 |  22 |   4 |  81 |  -  | OK  | OK  | OK  |
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure V.6. : statut du projet de l'exemple après le passage par le prouveur automatique en force 0

La figure V.7 montre le déroulement de la preuve automatique de ces obligations de preuve en "force 1". Le prouveur automatique détecte que la preuve a déjà été tentée en "force 0" et il passe alors directement à la force demandée. Celle-ci s'avère suffisante pour résoudre les obligations de preuve restantes.

```
Proving Connecteurs

  Proof pass 0, still 4 unproved PO

  Proof pass 1, still 4 unproved PO

    clause Aiguillage1
   ++++

End of Proof
  InstanciatedConstraintsLemmas       Proved 0     Unproved 0
  Initialisation   Proved 2     Unproved 0
  Aiguillage1      Proved 12    Unproved 0
  Ecrivain1        Proved 4     Unproved 0
  faire_initialisation   Proved 0     Unproved 0
  demande_verification_envoyer_in     Proved 0     Unproved 0
  demande_verification_recevoir_out   Proved 0     Unproved 0
  faire_ecriture1  Proved 0     Unproved 0
  demande_verification_envoyer_out    Proved 0     Unproved 0
  demande_verification_recevoir_in    Proved 0     Unproved 0
  Verificateur1    Proved 0     Unproved 0
  fournit_verification_recevoir_out   Proved 0     Unproved 0
  faire_verification1    Proved 0     Unproved 0
  fournit_verification_envoyer_in     Proved 0     Unproved 0
  fournit_verification_recevoir_in    Proved 0     Unproved 0
  fournit_verification_envoyer_out    Proved 0     Unproved 0
TOTAL for Connecteurs    Proved 18    Unproved 0
```

Figure V.7. : preuve automatique en force 1 des dernières obligations de preuve de l'exemple

Enfin, un nouvel affichage du statut du projet (cf. figure V.8) nous assure que toutes les obligations de preuve de toutes nos machines abstraites ont bien été vérifiées. Par conséquent, la spécification concrète que nous obtenons à l'issue de notre processus de transformation de description d'architecture en π-SPACE est bien un ensemble de machines abstraites valides de la méthode B.

```
Project status
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| COMPONENT                 | TC | POG | Obv | nPO | nUn | %Pr | B0C |  C  | Ada | C++ |
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Actions_Ecriture          | OK | OK  |  41 |   0 |   0 | 100 | OK  |     |     |     |
| Actions_Verification      | OK | OK  |  26 |   0 |   0 | 100 | OK  |     |     |     |
| Comportement_Ecrivain1    | OK | OK  |  76 |   4 |   0 | 100 | OK  |     |     |     |
| Comportement_Verificateur1 | OK | OK  |  37 |   0 |   0 | 100 | OK  |     |     |     |
| Configuration             | OK | OK  |   3 |   0 |   0 | 100 | OK  |     |     |     |
| Connecteurs               | OK | OK  | 501 |  18 |   0 | 100 | OK  |     |     |     |
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| TOTAL                     | OK | OK  | 684 |  22 |   0 | 100 | OK  | OK  | OK  | OK  |
+---------------------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure V.8. : statut du projet de l'exemple après le passage par le prouveur automatique en force 1

4. Conclusion

La conception de notre démarche de raffinement s'appuyait sur la volonté d'avoir un processus réutilisable de construction du raffinement, d'une description d'architecture logicielle jusqu'au code de l'application (cf. chapitre 1, paragraphe 2). Au bout du compte, le choix du langage de description d'architecture de départ ne nous apparaît pas comme spécialement important – à partir du moment où nous sommes capables d'écrire un patron de transformation pour aboutir à la forme architecturale canonique (présentée dans le chapitre précédent). Par contre, il n'en va pas de même pour les deux autres formalismes que nous utilisons.

D'une part, la méthode B nous permet de conserver un cadre formel, qui nous assure un développement sûr de l'implémentation du système dans un langage de programmation. De plus, sa mise en œuvre nous est facilitée par des outils commerciaux.

D'autre part, la logique de réécriture peut jouer le rôle à la fois de cadre syntaxique (comme langage d'expression des transformations) et de cadre sémantique (pour vérifier, au besoin, des propriétés des modèles). Elle aussi bénéficie d'un environnement disponible, et elle nous autorise à écrire nos mécanismes de raffinement vertical sous la forme de modules indépendants et réutilisables.

Nous obtenons ainsi un prototypage très rapide de notre approche β-SPACE, et la possibilité de vérifier la validité de chaque étape du raffinement.

Chapitre 6 : Conclusions et perspectives

Chapitre 6 : Conclusions et perspectives

1. Synthèse de notre approche

Nous avons traité, dans cette thèse, la transformation d'une description d'architecture logicielle en une description formelle dans la notation d'une méthode formelle classique. Le but poursuivi était essentiellement de faciliter le développement formel d'un système logiciel en renforçant les liens entre les étapes de description architecturale et de spécification classique (cf. chapitre 1).

Plus précisément, nous constatons que les méthodes formelles, comme la méthode B que nous avons utilisée (présentée dans le chapitre 3), constituent déjà un outil important pour les spécifications formelles, leur raffinement sûr, et même leur implémentation dans des langages de programmation. Le fondement sémantique formel, mathématique, permet de garantir la conservation des propriétés du système. Pourtant, elles ne sont pas faciles à prendre en main et à utiliser, notamment pour la modélisation initiale du système.

C'est pourquoi nous avons souhaité conjuguer la puissance du développement formel dans la méthode B à la convivialité et la flexibilité d'une spécification architecturale, en termes de composants communiquant via des connecteurs, dans un langage de description d'architecture logicielle. En effet, cette spécification préliminaire permet un découpage plus naturel, selon les calculs à effectuer et les interactions à modéliser. Afin de conserver une base sémantique formelle, avec un pouvoir d'expression important, nous avons choisi d'employer π-SPACE, un langage de description d'architecture logicielle fondé sur l'algèbre de processus du π-calcul (également décrit dans le chapitre 3).

Notre travail a alors consisté à fournir des mécanismes pour transformer une description architecturale écrite en π-SPACE, en une spécification faite de machines abstraites de la méthode B. De plus, cette tâche n'étant pas aisée, nous préférons qu'elle soit automatisée, et réutilisable.

C'est la raison pour laquelle nous avons fait appel à la logique de réécriture (cf. chapitre 3), afin de spécifier nos transformations par des règles de réécriture, génériques par nature. Elles sont regroupées sous la forme de patrons de transformation, développés dans le chapitre 4, et qui peuvent alors être appliqués systématiquement à n'importe quelle description architecturale en π-SPACE. Toutefois, la méthode B n'a pas été conçue pour supporter la description des aspects dynamiques ou évolutifs d'un système. Nous devons donc nous contenter de descriptions d'architectures statiques en entrée de nos patrons de raffinement.

Notre approche de raffinement β-SPACE est appliquée pendant la phase de conception architecturale (cf. tableau VI.1). Elle assure la traduction des constructions d'un langage dans un autre qui ne manipule pas les mêmes concepts. Pour ce faire, chaque "transformation élémentaire" est représentée par une ou plusieurs règles de la logique de réécriture.

Place du raffinement	**Type de processus de raffinement**	**Relation de raffinement**
conception architecturale	transformation par patrons génériques utilisant des théories de la logique de réécriture	traduction des concepts de contrôle architecturaux par réécriture des éléments architecturaux de π-SPACE en machines abstraites B

Tableau VI.1. : critères principaux pour l'approche de raffinement β-SPACE

Nous effectuons, par ailleurs, une distinction entre le raffinement supporté par π-SPACE – la décomposition architecturale, un raffinement horizontal – et celui que nous effectuons pour transformer une architecture en machines abstraites de la méthode formelle B – un raffinement vertical (cf. tableau VI.2). Cette transformation n'est pas destinée à modifier les données de la spécification, mais à assurer le raffinement de son comportement, de ses "fonctions" de calcul. Le découpage de notre approche en plusieurs patrons nous autoriserait à nous attarder sur une

description architecturale d'un niveau d'abstraction intermédiaire (par exemple, sur l'architecture canonique issue de l'application du deuxième patron de transformation) ; à cet instant, nous pourrions opérer de nouvelles décompositions, et composer de la sorte différentes étapes de raffinements horizontaux et verticaux. Nous ne gérons pas le raffinement de l'architecture jusqu'à la génération de code, mais celle-ci peut être effectuée dans le cadre d'un développement formel à partir de la spécification en machines abstraites B générée.

Comme la logique de réécriture nous sert à la fois de cadre syntaxique et de cadre sémantique, nous pouvons garantir la conservation des propriétés des descriptions raffinées. En outre, elle nous permettrait également de vérifier de nouvelles propriétés, qui seraient, par exemple, utiles au développement formel en B.

Distinction RH / RV	**Raffinement Horizontal (RH)**	**Raffinement Vertical (RV)**	**Raffinement de données**	**Raffinement fonctionnel**	**Raffinement comportemental**	**Raffinement compositionnel**	**Génération de code**	**Multiples niveaux d'abstraction**	**Réutilisation**	**Préservation de propriétés "inhérentes"**	**Préservation de propriétés "définies par l'utilisateur"**	**Multilangages**
OUI	NON (π-SPACE)	OUI	NON	OUI	OUI	OUI	NON (méthode B)	OUI	OUI	OUI	OUI	OUI

Tableau VI.2. : critères auxiliaires pour l'approche de raffinement β-SPACE

2. β-SPACE et l'état de l'art

Contrairement aux autres méthodologies du génie logiciel, nous différencions les relations de raffinement possibles suivant deux axes. D'un côté, la décomposition architecturale est indubitablement au cœur des intentions du raffinement horizontal : la spécification, une fois décomposée, contient de nouveaux détails. De l'autre côté, le raffinement vertical est supposé lui être orthogonal : il ne devrait pas être vu comme menant à une description plus précise, mais plutôt à une description plus "implémentable" (cf. chapitre 1, paragraphe 3).

Cette séparation nous permet de ne pas réinventer un mécanisme déjà existant dans les autres langages. De façon plus générale, plutôt que de chercher à obtenir en fin de raffinement une autre description d'architecture, nous ciblons (dès le chapitre 1, paragraphe 2) une spécification plus concrète dans une méthode formelle qui permettra de continuer avec un développement plus classique afin d'obtenir une implémentation sûre.

Peu de langages de description d'architecture logicielle supportent le raffinement vertical. Outre ceux qui ne permettent qu'une phase de compilation en code, SADL ou Rapide (présentés dans le chapitre 2) permettent de développer un processus de raffinement en plusieurs étapes. SADL utilise des patrons de raffinement réutilisables pour remplacer les constructions d'un style architectural dans une architecture par celles d'un autre style. Cependant, SADL ne permet de s'intéresser qu'à la structure de l'architecture, en termes de configurations de composants et connecteurs, mais pas à leur comportement. Rapide, d'autre part, décrit l'architecture d'un système de façon plus détaillée, en séparant les déclarations des interfaces des modules auxquelles elles correspondent. Néanmoins, le raffinement en Rapide ne sert pas à construire une nouvelle description, mais il est simplement vérifié a posteriori, en comparant les traces d'événements générées par des simulations des spécifications considérées.

Notre approche est différente de SADL, en cela qu'elle repose au départ sur une description d'architecture dont la sémantique plus précise est exprimée dans une algèbre de processus, et que nous cherchons à en conserver les propriétés. Contrairement à Rapide, nous utilisons des patrons de raffinement réutilisables, qui nous permettent, en plus, de construire le raffinement plutôt que de le constater après coup.

Notre approche du raffinement, β-SPACE, vient donc compléter les descriptions architecturales en les menant vers un développement formel pour obtenir leurs implémentations, et les méthodes formelles de développement en leur adjoignant une phase de conception architecturale préliminaire. Nous ne nous intéressons pas seulement qu'à l'ajout de détails à la spécification, mais aussi à la transformation de sa structure de contrôle. C'est une approche originale à la fois sur sa portée architecturale (structure mais aussi comportement) ainsi que sur sa formalisation et son articulation avec les méthodes formelles classiques.

Enfin, même si β-SPACE est une approche contrainte à la fois par un langage de description d'architecture, π-SPACE, et par le choix de la méthode formelle B, son découpage sous la forme de patrons génériques de règles de réécriture lui confère une structure modulaire autour d'une forme architecturale canonique (après application des deux premiers patrons, et avant celle du troisième, comme évoqué dans le chapitre 4, paragraphe 4).

3. Perspectives

Les perspectives ouvertes par nos travaux peuvent se grouper selon trois axes (cf. figure VI.1) : la continuation du travail proprement dit, des nouvelles applications et des nouveaux thèmes de recherche ouverts.

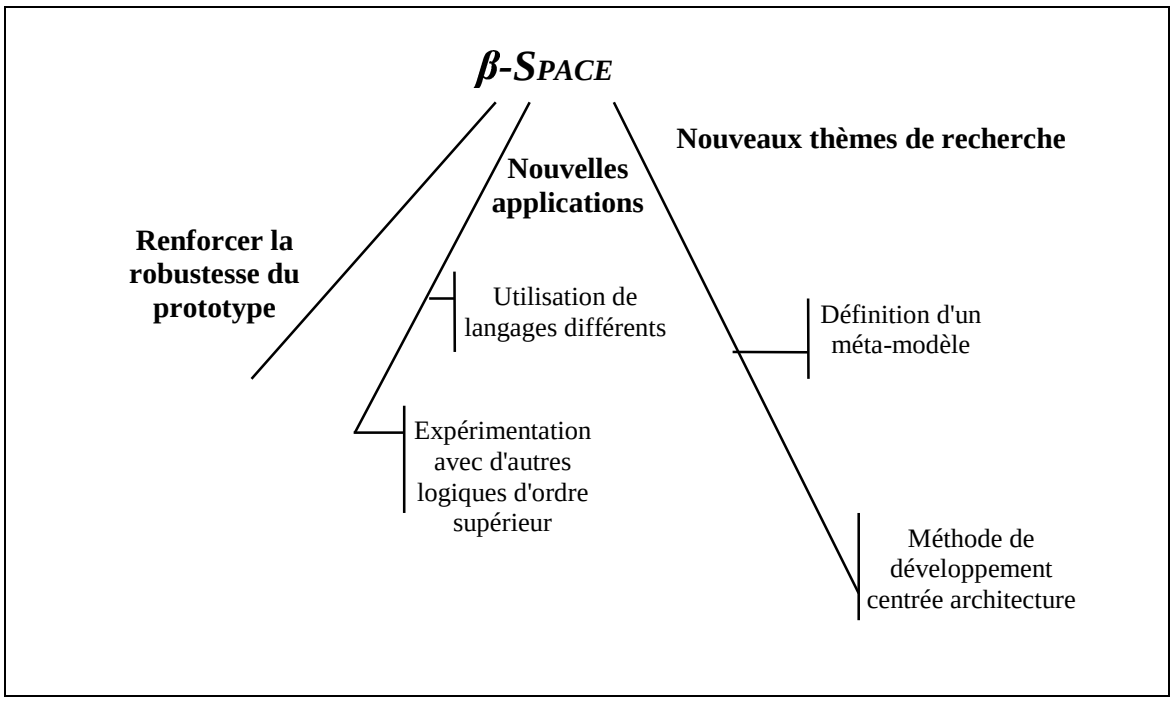

Figure VI.1. : perspectives de notre travail

Nous allons par la suite développer les perspectives développées selon les trois axes.

3.1. Renforcer la robustesse du prototype

Le premier pas dans la continuation de notre travail concerne le développement de l'environnement permettant à l'utilisateur de raffiner une description en π-SPACE.

En effet, nous n'avons pas encore éprouvé notre approche sur un nombre significatif de systèmes. Appliquer β-SPACE au raffinement de descriptions d'architectures différentes, de plus grande taille, pourrait ainsi en faire ressortir d'autres avantages et inconvénients, par exemple pour des aspects non fonctionnels comme l'ergonomie de l'interface, mais aussi pour confirmer la complétude de nos règles de réécriture relativement aux descriptions d'architectures logicielles en π-SPACE.

Nous pourrions également travailler à l'optimisation de nos patrons de transformation, notamment en ajoutant au noyau de Maude de nouvelles stratégies de réécriture : cette possibilité est issue de la réflexivité de la logique de réécriture, exploitée dans la conception même du "méta-niveau" de l'outil.

Par ailleurs, nous nous intéressons à la prise en compte des aspects dynamiques et évolutifs de π-SPACE, auxquels nous avons renoncé au cours de notre travail pour ne pas ajouter à la difficulté de la formalisation des transformations en logique de réécriture celle de la formalisation en machines abstraites B.

3.2. Nouvelles applications

3.2.1. Utilisation de langages différents

L'approche β-SPACE présentée est dépendante à la fois de la méthode B et de π-SPACE. Néanmoins, nous aurions pu choisir un autre langage de description d'architecture logicielle, comme par exemple WRIGHT [Allen 1997], fondé sur l'agèbre de processus CSP (définie dans [Hoare 1985] et présentée dans [Hinchey et Jarvis 1995]), un langage basé sur l'algèbre de processus CCS [Verdejo et Martí-Oliet 2000], sur LOTOS [Azcorra et al. 1993] …, ou une autre méthode formelle comme VDM [Plat et Larsen 1992] ou Z [Spivey 1992].

En effet, nos patrons de transformations s'articulent autour d'un format architectural canonique, basé sur les constituants élémentaires de toute architecture logicielle. Les deux langages, source et cible, se trouvent fortement découplés et en changer un ne posera d'autre problème que celui d'écrire un nouveau patron de règles de réécriture pour assurer la transformation reliant le nouveau langage utilisé à cette forme canonique.

3.2.2. Expérimentation avec d'autres logiques d'ordre supérieur

Si nous pouvons remettre en question deux des formalismes sur lesquels nous appuyons notre travail, il peut également être intéressant de réfléchir à la pertinence du troisième.

La logique de réécriture a l'avantage certain de fournir un cadre adapté à la réécriture, dans laquelle nos transformations se formalisent assez naturellement. Quoi qu'il en soit, il pourrait être envisagé de s'intéresser à ce que d'autres logiques d'ordre supérieur seraient capables d'apporter, en termes de pouvoir expressif ou de capacité de vérification.

3.3. Nouveaux thèmes de recherche

3.3.1. Définition d'un méta-modèle

Comme il a été dit précédemment, l'approche que nous proposons dans cette thèse est contrainte par plusieurs éléments, et fournit dans le même temps des structures génériques.

Aussi, notre travail ouvre de nouvelles perspectives, que ce soit dans la dissociation des raffinements horizontaux et verticaux, ou dans la transformation d'un formalisme vers un autre.

L'application de notre approche à d'autres langages peut être envisagée à court terme (cf. paragraphe 3.2.1), mais l'écriture "manuelle" de nouveaux patrons nécessite un dégré d'expertise certain de la logique de réécriture et du nouveau langage source ou cible considéré. C'est pourquoi nous devons, à moyen terme, après avoir pris un peu plus de recul et d'expérience, considérer la définition d'un méta-modèle pour le processus de transformation d'une spécification dans un langage de description d'architecture quelconque, voire un langage formel, en une spécification dans un autre langage formel. C'est une tâche qui s'annonce difficile, mais qui s'inscrit en droite ligne des travaux menés actuellement au laboratoire dans le domaine du raffinement d'architectures logicielles [Megzari 2004].

3.3.2. Méthode de développement centrée architecture

Dans la continuité de ce raisonnement, bien que certainement à plus long terme, nous pouvons également nous donner un objectif plus ambitieux comme la définition d'une méthode de développement entièrement centrée sur les architectures logicielles. En effet, l'expérience et la compétence de l'équipe dans ce domaine s'accroissent régulièrement ([Chaudet 2002], [Leymonerie 2004], …).

4. Conclusion

Cette thèse montre la faisabilité et l'intérêt de coupler en amont d'une méthode formelle classique une phase de description architecturale formelle, avec un support automatisé au raffinement. En effet, une description architecturale selon un point de vue "composant-connecteur" fournit des abstractions plus naturelles que les constructions des méthodes formelles classiques. Les patrons de raffinement permettent d'automatiser la traduction entre ces deux descriptions formelles. Toutefois, cette thèse montre également la limite de cette approche, notamment l'impédance entre les contructions des langages de description architecturale et de la méthode formelle ainsi que le problème d'expressivité des concepts dynamiques.

Comme le montre de manière assez significative la figure VI.1, les travaux présentés dans le cadre de cette thèse ouvrent des nouvelles perspectives de recherche. Le raffinement de spécifications dans des méthodes manipulant des concepts différents est suffisamment complexe, adresse suffisamment de problèmes qui font l'objet d'études par ailleurs, pour qu'il soit le cœur d'autres investigations.

Enfin, un des objectifs, à terme, est de fournir un environnement complet permettant non seulement la modélisation et le développement complet d'un système logiciel, mais également d'offrir un support à son évolution (des travaux en cours sont présentés dans [Oquendo et al. 2004], [Oquendo 2004a] et [Oquendo 2004b]).

REFERENCES BIBLIOGRAPHIQUES

[Abrial 1996] J.-R. Abrial, *The B-Book: Assigning programs to meanings*, Cambridge University Press, 1996.

[Abrial 2000] J.-R. Abrial, *Event Driven Sequential Program Construction*, rapport technique, mars 2000.

[Ait-Ameur et al. 1998] Y. Ait-Ameur, P. Girard et F. Jambon, "A Uniform Approach for Specification and Design of Interactive Systems: the B Method", *Proceedings of the 5th International Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'98)*, Cosener's House, Abingdon (Royaume Uni), 3-5 juin 1998.

[Allen 1997] R. J. Allen, *A formal approach to software architecture*, Mémoire de thèse, School of Computer Science, Carnegie Mellon University, mai 1997.

[Allen et al. 1998] R. Allen, D. Garlan et J. Ivers, "Formal Modeling and Analysis of the HLA Component Integration Standard", *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, novembre 1998.

[Allen et Garlan 1997] R. Allen et D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, juillet 1997.

[Azcorra et al. 1993] A. Azcorra Saloña, J. Quemada Vives et S. Pavón Gómez, *An introduction to LOTOS*, Dpto. Ingeniería de Sistemas Telemáticos, Univesidad Politécnica de Madrid, mai 1993.

[Baader et Nipkow 1998] Franz Baader et Tobias Nipkow, *Term Rewriting and All That*, Cambridge University Press, février 1998.

[Back et von Wright 1989] R.-J. R. Back et J. von Wright, *Refinement Calculus, Part I: Sequential Nondeterministic Programs*, 2 novembre 1989.

[Banach et Poppleton 1998] R. Banach et M. Poppleton, "Retrenchment: An Engineering Variation on Refinement", *Proceedings of B'98*, LNCS 1393: pp. 129-147, D. Bert (ed.), 1998.

[Banach et Poppleton 1999] R. Banach et M. Poppleton, "Sharp Retrenchment, Modulated Refinement and Simulation", *Form. Asp. Comp.*, 11, pp. 498-540, 1999.

[Bert et al. 1996] D. Bert, M.-L. Potet et Y. Rouzaud, "A Study on Components and Assembly Primitives in B", *Proceedings of First B Conference*, IRIN, Nantes, novembre 1996.

[Bolusset et al. 1999a] T. Bolusset, F. Oquendo et H. Verjus, "Software component-based federation architectures are software architectures too", *Proceedings of IPTW'99 International Process Technology Workshop*, Villard de Lans, 1-3 septembre 1999.

[Bolusset et al. 1999b] T. Bolusset, C. Chaudet et F. Oquendo, *π-SPACE: A Formal Architecture Description Language based on Process-Algebra for Component-Driven Engineering*, rapport technique, novembre 1999.

[Bolusset et al. 2000a] T. Bolusset, C. Chaudet et F. Oquendo, "Description d'architectures logicielles dynamiques : langage formel et applications industrielles", *Génie Logiciel*, 53 : 44-50, juin 2000.

[Bolusset et al. 2000b] T. Bolusset, L. Da Silva et F. Oquendo, "Développement à base de composants : une approche centrée architecture pour le raffinement et l'implémentation de logiciels en Java Beans", *Proceedings of the 13th International Conference on Software ans Systems Engineering and their Applications (ICSSEA'2000)*, CNAM, Paris, décembre 2000.

[Bolusset et Oquendo 2002] T. Bolusset et F. Oquendo, "Formal Refinement of Software Architectures Based on Rewriting Logic", *RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, IMAG, Grenoble, France, 22 janvier 2002.

[Bon et al. 2000] P. Bon, E. M. El Koursi et P. Yim, "Du cahier des charges aux spécifications formelles", *Proceedings de AFADL'2000*, janvier 2000.

[Butler 1999] Michael Butler, *csp2B: A Practical Approach To Combining CSP and B*, Declarative Systems and Software Engineering Group, Rapport Technique DSSE-TR-99-2, février 1999.

[Chaudet 2002] C. Chaudet, *π-SPACE : langages et outils pour la description d'architectures évolutives à composants dynamiques*, Mémoire de thèse, Annecy, 12 décembre 2002.

[Chaudet et Oquendo 2000] C. Chaudet et F. Oquendo, "π-SPACE: A Formal Architecture Description Language Based on Process Algebra For Evolving Software Systems", *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE' 00)*, Grenoble, septembre 2000.

[Clarke 1998] Lori A. Clarke, "Improving Architectural Description Languages to Support Analysis Better", *Proceedings of International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, pp. 78-80, Marsala, 30 juin – 3 juillet 1998.

[Clavel et al. 1999] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer et J. F. Quesada, *Maude: Specification and Programming in Rewriting Logic*, Maude system documentation, Computer Science Laboratory, SRI International, 8 mars 1999.

[Clavel et al. 2000] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer et J. F. Quesada, *A Maude Tutorial*, rapport technique, mars 2000.

[Clements 1996] Paul C. Clements, "A Survey of Architecture Description Languages", Eighth International Workshop on Software Specification and Design, Allemagne, mars 1996.

[Compare et al. 1999] D. Compare, P. Inverardi et A. L. Wolf, "Uncovering architectural mismatch in component behavior", *Science of Computer Programming 3*, Elsevier Science, février 1999.

[D'Souza et Wills 1998] Desmond Francis D'Souza et Alan Cameron Wills, *Objects, Components, and Frameworks with UML: the Catalysis Approach*, Addison-Wesley, 1998.

[Egyed et al. 1999] A. Egyed, N. Mehta et N. Medvidovic, *Software Connectors and Refinement in Family Architectures*, rapport technique, USC-CSE-99-527, 2 novembre 1999.

[Egyed et al. 2000] A. Egyed, N. Mehta et N. Medvidovic, "Software Connectors and Refinement in Family Architectures", *Proceedings of the 3rd International Workshop on Software Architectures*, Las Palmas de Gran Canaria, pp. 95-105, 2000.

[Egyed et Medvidovic 2000] A. Egyed et N. Medvidovic, "A Formal Approach to Heterogeneous Software Modeling", *Proceedings of the 3rd International Conference on Foundational Aspects of Software Engineering (FASE)*, Berlin, 2000.

[Egyed et Medvidovic 2001] A. Egyed et N. Medvidovic, "Consistent Architectural Refinement and Evolution using the Unified Modeling Language", *Proceedings of ICSE UML 2001*, 2001.

[Fahmy et Holt 2000] H. Fahmy et R. C. Holt, "Software Architecture Transformations", *Proceedings of the International Conference on Software Maintenance*, San Jose, octobre 2000.

[Fidge 1997] C. J. Fidge, *Modelling program compilation in the refinement calculus*, Rapport Technique No. 97-22, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1997.

[Garlan 1996] D. Garlan, "Style-Based Refinement for Software Architecture", *Proceedings of the 2nd International Software Architecture Workshop (ISAW-2)*, A. L. Wolf éditeur, pp. 72-75, San Francisco, octobre 1996.

[Garlan et al. 1995] D. Garlan, R. Allen et J. Ockerbloom, "Architectural Mismatch or, Why it's hard to build systems out of existing parts", *Proceedings of the 17th International Conference on Software Engineering (ICSE-17)*, avril 1995.

[Garlan et Perry 1995] D. Garlan et D. Perry, "Introduction to the Special Issue on Software Architecture", *IEEE Transactions on Software Engineering*, avril 1995.

[Garlan et Wang 1998] D. Garlan et Z. Wang, "A Case Study in Software Architecture Interchange", rapport technique, Composable Systems Group, Carnegie Mellon University, mars 1998.

[Gorrieri et Rensink 1999] R. Gorrieri et A. Rensink, *Action Refinement*, rapport technique UBLCS-99-09, Department of Computer Science, University of Bologna, avril 1999.

[Hinchey et Jarvis 1995] Michael G. Hinchey et Stephen A. Jarvis, *Concurrent Systems: Formal Development in CSP*, McGraw-Hill International Series in Software Engineering, 1995.

[Hoare 1985] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science, 1985.

[Honsell et al. 2000] F. Honsell, J. Longley, D. Sannella et A. Tarlecki, "Constructive data refinement in typed lambda calculus", *Proceedings of the 3rd International Conference on Foundations of Software Science and Computation Structures, European Joint Conferences on Theory and Practice of Software (ETAPS'2000)*, vol. 1784 of Lecture Notes in Computer Science, pp. 149-164, Springer, 2000.

[Kerschbaumer 2002] Andreas Kerschbaumer, "Non-refinement Transformations of Software Architectures", *RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, IMAG, Grenoble, France, 22 janvier 2002.

[Kirby 1998] G. Kirby, Notes on Java Implementation of Writer / Checker Process, rapport technique, University of Manchester, 30 novembre 1998.

[Kirby et Greenwood 1998] G. Kirby et M. Greenwood, *Writer / Checker Process Specification*, rapport technique, University of Manchester, 16 novembre 1998.

[Klenov et Pierre 2002] L. Klenov et L. Pierre, "Refining Iterative Programs in the B Method", *RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, IMAG, Grenoble, France, 22 janvier 2002.

[Lecomte 2002] Thierry Lecomte, "Event Driven B : methodology, language, tool support and experiments", *RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, Grenoble – IMAG, 22 janvier 2002.

[Lermer et Fidge 1997] Karl Lermer et Colin J. Fidge, *Compilation as refinement*, technical report No. 97-29, Software Verification Research Centre, Department of Computer Science, University of Queensland, mai 1997.

[Leymonerie 2004] Fabien Leymonerie, Définition *et utilisation des styles architecturaux pour les systèmes dynamiques*, Mémoire de thèse, Université de Savoie, Annecy, décembre 2004.

[Lowe et Zedan 1995] Gavin Lowe et Hussein Zedan, "Refinement of Complex Systems: A Case Study", *The Computer Journal*, 15 décembre 1995.

[Luckham et al. 1995] D. C. Luckham et al., "Specification and analysis of system architecture using Rapide", *IEEE Transactions on Software Engineering*, 21 (4) : 336-355, avril 1995.

[Marcano et al. 2000] R. Marcano, E. Meyer, N. Levy et J. Souquières, "Utilisation de patterns dans la construction de spécifications en UML et B", *Proceedings de AFADL'2000*, janvier 2000.

[Martí-Oliet et Meseguer 1993] N. Martí-Oliet et J. Meseguer, *Rewriting logic as a Logical and Semantic Framework*, rapport technique, SRI International, Computer Science Laboratory, Menlo Park, août 1993.

[Martí-Oliet et Meseguer 1996] N. Martí-Oliet et J. Meseguer, Rewriting logic as a Logical and Semantic Framework, *Electronic Notes in Theoretical Computer Science*, 4, 1996.

[Medvidovic et Taylor 2000] N. Medvidovic et R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, janvier 2000.

[Megzari 2004] Karim Megzari, Refiner :Environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture, Mémoire de thèse, Université de Savoie, Annecy, novembre 2004.

[Melton et Garlan 1997] Ralph Melton et David Garlan, "Architectural Unification", *Proceedings of CASCON'97*, novembre 1997.

[Meseguer 1998] José Meseguer, "Research Directions in Rewriting Logic", *Computational Logic*, NATO Advanced Study Institute, U. Berger et H. Schwichtenberg éditeurs, Springer-Verlag, 1998.

[Milner 1989] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[Milner 1999] R. Milner, *Communicating and mobile systems: the π-calculus*, Cambridge University Press, 1999.

[Milner et al. 1992] R. Milner, J. Parrow and D. Walker, "A calculus of mobile processes", *Journal of Information and Computation*, 100 : 1-77, 1992.

[Moriconi et Qian 1994] M. Moriconi et X. Qian, "Correctness and Composition of Software Architectures", *in Proceedings of ACM SIGSOFT'94: Symposium on the Foundations of Software Engineering*, pp. 164-174, New Orleans, Louisiana, décembre 1994.

[Moriconi et al. 1995] M. Moriconi, X. Qian et R. A. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, 21 (4) : 356-372, avril 1995.

[Moriconi et Riemenschneider 1997] M. Moriconi et R. A. Riemenschneider, *Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies*, Rapport Technique SRI-CSL-97-01, Computer Science Laboratory, SRI International, mars 1997.

[Ohlebusch 2002] Enno Ohlebusch, *Advanced Topics in Term Rewriting*, Springer-Verlag, XV, 2002.

[Oquendo 2004a] Flavio Oquendo, "π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures", J. ACM Software Engineering Notes, Vol. 28, No. 8, USA, Mai 2004.

[Oquendo 2004b] Flavio Oquendo, "π-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures", J. ACM Software Engineering Notes, Vol. 28, No. 9, USA, Septembre 2004.

[Oquendo et al. 2004] Flavio Oquendo et al., "ArchWare: Architecting Evolvable Software", Proceedings of the First European Workshop on Software Architecture (EWSA'04), European Projects in Software Architecture – Invited Papers, Lecture Notes in Computer Science, Springer Verlag, St Andrews, UK, Mai 2004.

[Oreizy et al. 1999] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum et A. L. Wolf, "An architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, 1094-7167 / 99, pp. 54-62, mai 1999.

[Osterweil 1998] Leon Osterweil, "Architecting Processes are Key to Software Quality", *Proceedings of International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, pp. 18-20, Marsala, 30 juin – 3 juillet 1998.

[Perry 1997a] D. E. Perry, "An Overview of the State of the Art in Software Architecture", *1997 International Software Engineering Conference (ICSE97)*, Boston, mai 1997.

[Perry 1997b] D. E. Perry, "Software Architecture and its Relevance to Software Engineering", *Coordination 1997*, Berlin, septembre 1997.

[Perry 1998] D. E. Perry, "Generic Architecture Descriptions for Product Lines", *ARES II Product Line Architecture Workshop*, Las Palmas, février 1998.

[Perry et Wolf 1992] D. E. Perry et A. L. Wolf, "Foundations for the study of Software Architecture", *Software Engineering Notes*, ACM SIGSOFT, vol. 17, n°4, pp. 40-52, octobre 1992.

[Plat et Larsen 1992] N. Plat et P. G. Larsen, "An Overview of the ISO/VDM-SL Standard", *ACM SIGPLAN Notices*, 1992.

[Potet et Rouzaud 1998] M.-L. Potet et Y. Rouzaud, "Composition and Refinement in the B-Method", *Proceedings of the Second International B Conference*, LNCS 1393, Springer Verlag, pp. 46-65, avril 1998.

[Riemenschneider 1998] R. A. Riemenschneider, *Correct Transformation Rules for Incremental Development of Architecture Hierarchies*, Working Paper DSA-98-01, SRI CSL Dependable System Architecture Group, février 1998.

[Sa et al. 1993] J. Sa, B. C. Warboys et J. A. Keane, "OBM: A Specification Method for Modelling Organisational Process", *Proceedings of the Workshop on Constraint Processing at CSAM'93*, Saint Petersbourg, 1993.

[Sanlaville 1997] Rémy Sanlaville, *Description d'architectures logicielles : Utilisation du formalisme WRIGHT pour l'interconnexion de machines abstraites B*, Mémoire de DEA, Grenoble, 1997.

[Sannella 2000] D. Sannella, "Algebraic specification and program development by stepwise refinement", *Proceedings of the 9th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR'99*, vol. 1817 of Lecture Notes in Computer Science, pp. 1-9, Springer, 2000.

[Shaw et Garlan 1995] M. Shaw et D. Garlan, "Formulations and Formalisms in Software Arcitecture", *Computer Science Today: Recent Trends and Developments*, Jan Van Leeuwen éditeur, Springer Verlag, Lecture Notes in Computer Science, vol. 1000, 1995.

[Spivey 1992] J. M. Spivey, *The Z Notation: A Reference Manual, Second Edition*, by Prentice Hall International (UK) Ltd, première publication en 1992.

[Steria 1998] Steria, *Manuel de référence du langage B*, version 1.8.1, STERIA support Atelier B, Aix-en-Provence, 27 novembre 1998.

[Treharne et Schneider 1999] H. Treharne et S. Schneider, "Using a Process Algebra to Control B OPERATIONS", *Proceedings of IFM'99*, Springer, York, 1999.

[Treharne et Schneider 2000] H. Treharne et S. Schneider, "How to Drive a B Machine", *Proceedings of ZB'2000: Formal Specification and Developmnet in Z and B*, LNCS 1878, 2000.

[Verdejo et Martí-Oliet 2000] Alberto Verdejo et Narciso Martí-Oliet, *Executing and Verifying CCS in Maude*, rapport technique TR-SIP-99-00, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, février 2000.

[Vickers et Morgan 1994] Trevor Vickers et Carroll Morgan, *Procedures and invariants in the refinement calculus*, Rapport Technique TRCS9404, Department of Computer Science, Australian National University, mai 1994.

[Vos et al. 2002] T. E. J. Vos, S. D. Swierstra et W. Prasetya, "Yet Another Program Refinement Relation", *RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, IMAG, Grenoble, France, 22 janvier 2002.

[Warboys et al. 1999] B. C. Warboys, D. Balasubramaniam, R. M. Greenwood, G. N. C. Kirby, K. Mayes, R. Morrison et D. R. Munro, "Collaboration and Composition: Issues for a Second Generation Process Language", *Proceedings of the European Software Engineering Conference (ESEC'99)*, septembre 1999.

[Ziane 2001] Mikal Ziane, "Towards tool support for design patterns using program transformations", *Conférence Langage et Modèles à Objets (LMO 2001)*, 2001.

GLOSSAIRE

Architecture d'un logiciel : voir "Architecture logicielle".

Architecture logicielle : modélisation du système comme une composition de composants et de connecteurs.

Atteignabilité : propriété indiquant que tout état pourra toujours être atteint.

Calcul : traitement informatique à opérer pour résoudre tout ou partie du problème.

Code : description du système logiciel écrite dans un langage de programmation en vue de son exécution sur une plate-forme matérielle donnée ; programme exécutable obtenu par compilation pour une plate-forme matérielle donnée.

Code de l'application : voir "Code".

Code exécutable : voir "Code" (deuxième définition).

Communication : dans une architecture logicielle, événement mettant en relation deux composants, ou un composant avec un connecteur.

Compilation : étape de génération de code exécutable à partir d'une implémentation.

Comportement : ensemble partiellement ordonné d'actions entreprises par un composant ou un connecteur pour achever sa part de calcul dans l'architecture globale du système ; le comportement d'un composant ou d'un connecteur indique comment sont combinés tous ses ports.

Composant : unité de calcul localisée et indépendante ; il possède un comportement et communique avec l'extérieur via des ports.

Composant composite : configuration envisagée comme un composant dans le reste de l'architecture globale du système.

Configuration : assemblage de composants communiquant via des connecteurs ; sous-architecture.

Connecteur : interaction ou lien de communication entre plusieurs composants, dont le comportement représente le protocole de communication ; il possède un comportement et communique avec l'extérieur via des ports.

Décomposition : action de détailler le comportement d'un composant sous la forme d'une sous-architecture ; voir "Raffinement horizontal".

Description architecturale : représentation de l'architecture logicielle du système dans un langage de description d'architecture, sous la forme d'une spécification informelle, semi-formelle ou formelle.

Description d'architecture : voir "Description architecturale".

Description d'architecture logicielle : voir "Description architecturale".

Etat : interprétation valide d'une représentation d'un système, avec valuation de l'ensemble des variables.

Etat final : état à partir duquel aucune action ne peut être effectuée.

Génération de code : étape, automatisée ou non, de création d'une implémentation du système à partir d'une spécification concrète.

Implémentation : spécification concrète du système, au niveau d'indéterminisme le plus bas ; description du système logiciel écrite dans un langage de programmation en vue de son exécution sur une plate-forme matérielle donnée.

Implantation : voir "Implémentation".

Interaction : voir "Communication".

Langage de description d'architecture : langage dont la syntaxe permet la représentation d'une architecture logicielle sous la forme d'une spécification formelle ou semi-formelle.

Langage de description d'architecture logicielle : voir "Langage de description d'architecture".

Patron : structure générique et réutilisable, appliquée seule ou en combinaison pour obtenir une spécification du système.

Port : interface de communication d'un composant ou d'un connecteur, représentant un protocole de communication possible.

Propriété : théorème vérifié par une représentation d'un système.

Raffinement : action de raffiner ; spécialisation d'une représentation d'un système, pour passer d'une spécification abstraite à une spécification plus concrète ou à une implémentation.

Raffinement horizontal : décomposition d'une partie d'une spécification pour détailler une partie du comportement.

Raffinement vertical : transformation de tout ou partie d'une spécification pour préciser des choix liés à la plate-forme d'implémentation ultérieure, en diminuant le niveau d'abstraction et d'indéterminisme.

Raffiner : offrir une nouvelle formulation de la spécification abstraite qui ne doit pas contredire ses propriétés ; diminuer le niveau d'abstraction et d'indéterminisme.

Relation de raffinement : lien logique entre une spécification abstraite raffinée et un de ses raffinements possibles.

Spécification : représentation du système à un haut niveau d'abstraction.

Spécification abstraite : spécification qui doit être raffinée avant d'obtenir une implémentation.

Spécification architecturale : voir "Description architecturale".

Spécification concrète : spécification obtenue après raffinement.

Spécification d'architecture : voir "Description architecturale".

Spécification formelle : spécification dont la sémantique est complètement et précisément définie sur des bases mathématiques.

Spécification informelle : spécification en langage naturel ou dont la sémantique est ambiguë, voire non précisée.

Spécification semi-formelle : spécification dont la sémantique est partiellement définie.

Sûreté : propriété indiquant que toute action pouvant être effectuée à partir d'un état donné mènera à un autre état.

Système : logiciel à développer ; solution au problème posé.

Terminaison : propriété indiquant qu'à partir de tout état donné, une suite finie, éventuellement vide, d'actions mènera à un état final.

Transformation : remplacement d'une partie du vocabulaire et de la syntaxe ; voir "Raffinement vertical".

Vivacité : propriété indiquant qu'à partir de tout état donné, une action pourra toujours être effectuée.

ANNEXE A

Noms	Place du raffinement	Type de processus de raffinement	Relation de raffinement	Distinction RH / RV	Raffinement Horizontal (RH)	Raffinement Vertical (RV)	Raffinement de données	Raffinement fonctionnel	Raffinement comportemental	Raffinement compositionnel	Génération de code	Multiples niveaux d'abstraction	Réutilisation	Préservation de propriétés "inhérentes"	Préservation de propriétés "définies par l'utilisateur"	Multi-langages
Darwin, MetaH, UniCon, Weaves	implémentation	compilateur (excepté Weaves)	compilation	NON	-	OUI	-	-	-	NON	OUI	NON	NON	OUI	NON	-
Rapide	conception	comparaison de traces a posteriori	inclusion de la trace abstraite dans la trace concrète	-	-	-	NON	NON	OUI	NON	NON	-	NON	OUI mais juste 1 niveau	NON	plusieurs sous-langages
Catalysis	conception	documentation a posteriori	documentation	NON	OUI	OUI	OUI	OUI	limité	NON	NON	OUI	NON	NON	NON	-
VDM	conception	comparaison a posteriori des modèles possibles	l'implémentation doit avoir une fonctionnalité vue comme implémentant les constructions spécifiées de façon moins précise	-	-	NON pas réellement	NON	OUI	OUI (modèles possibles)	OUI	NON	pas très distincts	NON	OUI (modèles possibles)	NON	NON
Z	conception	vérification a posteriori de conditions mathématiques (logiques)	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme), lorsque l'abstraction se termine	-	-	OUI	OUI	OUI	limité	OUI	NON	pas très distincts	NON	OUI (obligations de preuves)	NON	NON
Méthode B	conception	différentiel	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme) : comportement possible de l'abstraction	-	-	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	limité	OUI (excepté les implémentations)	OUI	OUI	NON	OUI (obligations de preuve)	NON (pas automatisée)	mots-clés réservés en fonction du niveau d'abstraction
B événementiel	conception	différentiel (extension du B classique)	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme)	NON	OUI (mais pas sous forme de composant et connecteur)	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	limité	OUI (excepté les implémentations) : possibilité d'enchaîner raffinements et décompositions	OUI	OUI	NON	OUI (obligations de preuve sur des états et des événements)	NON (pas automatisée)	mots-clés réservés en fonction du niveau d'abstraction

B et CSP	conception	différentiel (raffinement classique en B : la partie CSP est conservée dans les différents niveaux d'abstraction)	affaiblissement des préconditions et renforcement des postconditions (plus de déterminisme) : comportement possible de l'abstraction	-	-	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	limité	OUI (excepté les implémentations)	NON	OUI	NON	OUI (obligations de preuve)	NON	OUI : CSP et un sous-ensemble de B
CSP2B	conception	différentiel (raffinement classique en B : les machines B et CSP sont raffinées indépendamment)	affaiblissement des préconditions et renforcement des postconditions	-	-	OUI (mais limité surtout aux corps d'opérations)	limité	OUI modification des corps d'opérations	OUI	OUI (excepté les implémentations)	OUI (conversion des machines CSP en B)	OUI	NON	OUI (obligations de preuve)	NON (pas automatisée)	OUI (CSP et B)
Réduction pour la méthode B	conception	différentiel mais le résultat est la spécification d'un nouveau problème	renforcement des préconditions et affaiblissement des postconditions	-	-	NON pas vraiment (ajout d'opération et changement de paramètres)	limité	OUI	limité	NON	NON	NON	NON	OUI mais limitée aux relations de recouvrement	NON	NON
Calcul de raffinement	conception / implémentation	transformation et raffinement de comportement ou justification de programmation	lois de raffinement (renforcement de postconditions, …) et invariants	NON	possible	possible	OUI	OUI (utilisation d'invariants locaux)	OUI	NON	OUI (lois de raffinement)	OUI	OUI (lois de raffinement)	OUI (invariants)	NON	NON (mais différents vocabulaires)
SADL	conception	mise en correspondance de styles	traduction selon la mise en correspondance : exactement la même chose	distinction entre décomposition et raffinement de style	composants composites statiques	changement de styles architecturaux entiers	pour ce qui peut être défini dans les styles	pour ce qui peut être défini dans les styles	NON	OUI	NON	si les styles sont considérés ainsi	OUI patrons	OUI	NON	NON
β-SPACE	conception architecturale	transformation par patrons génériques utilisant des théories de la logique de réécriture	traduction des concepts de contrôle architecturaux par réécriture des éléments architecturaux de π-SPACE en machines abstraites B	OUI	NON (π-SPACE)	OUI	NON	OUI	OUI	OUI	NON (méthode B)	OUI	OUI	OUI	OUI	OUI

ANNEXE B

*** **PREMIERE TRANSFORMATION**
*** *prise en compte des attachements et substitution des rôles du connecteur*

*** ***hypothèses*** : pas d'**extends** (disparus durant la phase d'instanciation), au moins 2 ports par connecteur,
*** ports commutatifs (renommage effectué donc plus besoin de garder l'ordre),
*** déclarations d'éléments architecturaux commutatives, attachements directs de canaux déjà pris en compte,
*** pas de type de composition (peuvent avoir été utilisés à l'instanciation) donc n'apparaissent pas
*** dans la composition, ni dans la partie déclaration d'éléments architecturaux,
*** ni celle des attachements (**where**)
*** + 1 opérateur booléen de test d'absence d'un port dans une liste de port : **absence `(** ROLE1 **`,** LPN1 **`)**

```
vars ARCHI1 ARCHI2 ARCHI : Architecture .
vars COMPONENTTYPE1 COMPONENTTYPE2 : ComponentTypeName .
vars PORT1 ROLE1 PORT2 ROLE2 : PortName .
vars PORTTYPE1 ROLETYPE1 PORTTYPE2 ROLETYPE2 : PortTypeName .
vars PORTTYPEPARAM ROLETYPEPARAM LTCHANNELS1 PORTTYPEPARAM2 ROLETYPEPARAM2 : ListOfTypedChannels .
vars CONNECTORTYPE1 CONNECTORTYPE2 : ConnectorTypeName .
vars COMPOSITION1 COMPOSITION2 : CompositeName .
var COMPOSANT1 : ComponentName .
var CONNECTEUR1 : ConnectorName .
vars LTPORTS1 PTP1 : ListOfTypedPorts .
var AEDECL : ArchitecturalElementDeclarations .
var LADECL : ListOfAttachmentDeclarations .
var CONNECTORBEHAVIOURTYPE1 : ConnectorBehaviourTypeName .
var LTPARAM1 : ListOfTypedParameters .
var LPN1 : ListOfPortNames .
var CONNBEHA1 : ConnectorBehaviour .
var LCONNPROC1 : ListOfConnectorProcessus .
var PORTDECL1 : PortDeclaration .
var CONNECTORBEHAVIOUR1 : BehaviourName .
var PORTSPEC1 : PortSpecification .
var COMPONENTBEHAVIOURTYPE1 : ComponentBehaviourTypeName .
var COMPBEHAVSPEC1 : ComponentsBehaviourSpecification .
var COMPBEHAVDECL : ComponentBehaviourDeclaration .

*** équations complémentaires pour la première transformation
***
*** 32 équations

*** ARCHI `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] /
*** CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]
eq ( ARCHI1 ARCHI2 ) `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `] =
        ARCHI1 `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `] .

eq ARCHI1 `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `] `[ COMPONENTTYPE2 @ PORT2 `: PORTTYPE2 `[ PORTTYPEPARAM2 `] / CONNECTORTYPE2 @ ROLE2 `: ROLETYPE2 `[ ROLETYPEPARAM2 `] `] =
        ARCHI1 `[ COMPONENTTYPE2 @ PORT2 `: PORTTYPE2 `[ PORTTYPEPARAM2 `] / CONNECTORTYPE2 @ ROLE2 `: ROLETYPE2 `[ ROLETYPEPARAM2 `] `] `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `] .


*** CONNBEHA1 `[ PORT1 / ROLE1 `]
vars X Y Z T : PortName .
vars M N : ConnectorBehaviour .
vars CM CN : ChoiceConnectorBehavior .
var PN : ParameterName .
var EWWC : ElementWithWhichCompared .
var CBTN : ConnectorBehaviourTypeName .
var LPN : ListOfPortNames .
var C : Channel .

eq `( M `) `[ X / Y `] = `( M `[ X / Y `] `) .
eq ( M ¤ N ) `[ X / Y `] = M `[ X / Y `] ¤ N `[ X / Y `] .
eq ( CM + CN ) `[ X / Y `] = CM `[ X / Y `] + CN `[ X / Y `] .
eq ( `[ PN `= EWWC `] M ) `[ X / Y `] = `[ PN `= EWWC `] M `[ X / Y `] .
eq ( `[ PN `> EWWC `] M ) `[ X / Y `] = `[ PN `> EWWC `] M `[ X / Y `] .
eq ( `[ PN `< EWWC `] M ) `[ X / Y `] = `[ PN `< EWWC `] M `[ X / Y `] .
eq ( `[ PN `>`= EWWC `] M ) `[ X / Y `] = `[ PN `>`= EWWC `] M `[ X / Y `] .
eq ( `[ PN `<`= EWWC `] M ) `[ X / Y `] = `[ PN `<`= EWWC `] M `[ X / Y `] .
eq ( `[ PN `<`> EWWC `] M ) `[ X / Y `] = `[ PN `<`> EWWC `] M `[ X / Y `] .
eq ( M | N ) `[ X / Y `] = M `[ X / Y `] | N `[ X / Y `] .
eq ( CBTN `[ LPN `] ) `[ X / Y `] = CBTN `[ LPN `[ X / Y `] `] .
eq ( X `, LPN ) `[ Z / X `] = Z `, LPN .
ceq ( Y `, LPN ) `[ Z / X `] = Y `, LPN `[ Z / X `] if X =/= Y .
eq X `[ Z / X `] = Z .
ceq Y `[ Z / X `] = Y if X =/= Y .
eq $ `[ X / Y `] = $ .
eq ( Y @ C ) `[ Z / X `] = Y `[ Z / X `] @ C .

ceq M `[ X / Y `] `[ Z / T `] = M `[ Z / T `] `[ X / Y `] if T =/= Y .
ceq CM `[ X / Y `] `[ Z / T `] = CM `[ Z / T `] `[ X / Y `] if T =/= Y .
ceq LPN `[ X / Y `] `[ Z / T `] = LPN `[ Z / T `] `[ X / Y `] if T =/= Y .

*** LCONNPROC1 `[ PORT1 / ROLE1 `]
vars LCP LCP1 : ListOfConnectorProcessus .
var PROCESS : ProcessName .
var LTPP : ListOfTypedProcessParameters .
```

*** subsorts TypedPort TypedParameter < ListOfTypedProcessParameters .
var POTYNA : PortTypeName .
var PATYNA : ParameterTypeName .

eq (LCP `, LCP1 ) `[X / Y `] = LCP `[X / Y `] `, LCP1 `[ X / Y `] .
eq (PROCESS `[ LTPP `] `= M ) `[X / Y `] = PROCESS `[LTPP `[ X / Y `] `] `= M `[ X / Y `] .
eq (Y `: POTYNA `, LTPP) `[ X / Y `] = X `: POTYNA `, LTPP .
ceq (Y `: POTYNA `, LTPP) `[ X / Z `] = Y `: POTYNA `, LTPP `[ X / Z `] if Y =/= Z .
eq (PN `: PATYNA `, LTPP) `[ X / Y `] = PN `: PATYNA `, LTPP `[ X / Y `] .
eq (Y `: POTYNA ) `[X / Y `] = X `: POTYNA .
ceq (Y `: POTYNA ) `[X / Z `] = Y `: POTYNA if Y =/= Z .
eq (PN `: PATYNA ) `[X / Y `] = PN `: PATYNA .

ceq LCP `[ X / Y `] `[ Z / T `] = LCP `[ Z / T `] `[ X / Y `] if T =/= Y .
ceq LTPP `[ X / Y `] `[ Z / T `] = LTPP `[ Z / T `] `[ X / Y `] if T =/= Y .

*** première règle de transformation
***
*** **decompose** ou pas
*** PTP1 ou rien (liste de ports avec types de paramètres)
*** LADECL ou pas (liste de déclaration d'attachements)
*** 2x2x2=8 règles

rl [transfo1.1.111] : *** pas de **decompose**, pas d'élément supplémentaire, plusieurs attachements
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 `, LADECL **`}**
=>
ARCHI **`[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] **`]**
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL **where** LADECL **`} .**

rl [transfo1.1.211] : *** **decompose**, pas d'élément supplémentaire, plusieurs attachements
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 `, LADECL **`}**
=>
ARCHI **`[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] **`]**
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL **where** LADECL **`} .**

*** PTP1

rl [transfo1.1.121] : *** pas de **decompose**, pas d'élément supplémentaire, PTP1, plusieurs attachements
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 `, LADECL **`}**
=>
ARCHI **`[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] **`]**
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL **where** LADECL **`} .**

rl [transfo1.1.221] : *** **decompose**, pas d'élément supplémentaire, PTP1, plusieurs attachements
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 `, LADECL **`}**
=>
ARCHI **`[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] **`]**
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL **where** LADECL **`} .**

*** un seul attachement

rl [transfo1.1.112] : *** pas de **decompose**, pas d'élément supplémentaire, un seul attachement
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 **`}**
=>
ARCHI **`[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] **`]**
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL **`} .**

rl [transfo1.1.212] : *** **decompose**, pas d'élément supplémentaire, un seul attachement
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 `**}**
=>
ARCHI `**[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `**]**
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL `**}** .

*** PTP1

rl [transfo1.1.122] : *** pas de **decompose**, pas d'élément supplémentaire, PTP1, un seul attachement
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 `**}**
=>
ARCHI `**[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `**]**
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL `**}** .

rl [transfo1.1.222] : *** **decompose**, pas d'élément supplémentaire, PTP1, un seul attachement
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `, LTPORTS1 `] || AEDECL **where attach** COMPOSANT1 @ PORT1 **to** CONNECTEUR1 @ ROLE1 `**}**
=>
ARCHI `**[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `**]**
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, PTP1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS1 `] || AEDECL `**}** .

*** groupe de deuxième règle de transformation
***
*** LTPARAM1 ou pas (liste de paramètres)
*** LCONNPROC1 ou pas (liste de processus)

*** type de comportement de connecteur concerné ou pas
*** 8 règles
*** + 1 opérateur booléen de test d'absence d'un port dans une liste de port

rl [transfo1.2.111] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ ROLE1`: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] `**{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ ROLE1 `, LPN1 `] `= CONNBEHA1 `**}** `**[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `**]**
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `**{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= CONNBEHA1 `**[** PORT1 / ROLE1 `**]** `**}** .

rl [transfo1.2.121] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ ROLE1`: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] `**{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ ROLE1 `, LPN1 `] `= CONNBEHA1 `, LCONNPROC1 `**}** `**[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `**]**
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `**{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= CONNBEHA1 `**[** PORT1 / ROLE1 `**]** `, LCONNPROC1 `**[** PORT1 / ROLE1 `**]** `**}** .

rl [transfo1.2.211] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ ROLE1`: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] `**{** CONNECTORBEHAVIOURTYPE1 `[ROLE1 `, LPN1 `] `= CONNBEHA1 `**}** `**[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `**]**
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `**{** CONNECTORBEHAVIOURTYPE1 `[PORT1 `, LPN1 `] `= CONNBEHA1 `**[** PORT1 / ROLE1 `**]** `**}** .

rl [transfo1.2.221] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ ROLE1`: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] `**{** CONNECTORBEHAVIOURTYPE1 `[ROLE1 `, LPN1 `] `= CONNBEHA1 `, LCONNPROC1 `**}** `**[** COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `**]**
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `**{** CONNECTORBEHAVIOURTYPE1 `[PORT1 `, LPN1 `] `= CONNBEHA1 `**[** PORT1 / ROLE1 `**]** `, LCONNPROC1 `**[** PORT1 / ROLE1 `**]** `**}** .

***

crl [transfo1.2.112] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `]
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 `}
if **absence** `( ROLE1 `, LPN1 `) .

crl [transfo1.2.122] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 `, LCONNPROC1 `} `[COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 `, LCONNPROC1 `}
if **absence** `( ROLE1 `, LPN1 `) .

crl [transfo1.2.212] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ CONNECTORBEHAVIOURTYPE1 `[LPN1 `] `= CONNBEHA1 `} `[COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ CONNECTORBEHAVIOURTYPE1 `[LPN1 `] `= CONNBEHA1 `}
if **absence** `( ROLE1 `, LPN1 `) .

crl [transfo1.2.222] :
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ CONNECTORBEHAVIOURTYPE1 `[LPN1 `] `= CONNBEHA1 `, LCONNPROC1 `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `]
=>
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ CONNECTORBEHAVIOURTYPE1 `[LPN1 `] `= CONNBEHA1 `, LCONNPROC1 `}
if **absence** `( ROLE1 `, LPN1 `) .

*** groupe de troisième règle de transformation
***
*** type de connecteur concerné ou pas
*** 2 règles

rl [transfo1.3.1] :
define connector type CONNECTORTYPE1 `[ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `, LTPORTS1 `] `{ **port** ROLE1`: ROLETYPE1 `[ ROLETYPEPARAM `] || PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `, LTPORTS1 `] `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `]
=>
define connector type CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `{ **port** PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] || PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS1 `] `} .

crl [transfo1.3.2] :
define connector type CONNECTORTYPE2 `[ LTPORTS1 `] `{ PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `} `[COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ ROLETYPEPARAM `] `]
=>
define connector type CONNECTORTYPE2 `[ LTPORTS1 `] `{ PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `}
if CONNECTORTYPE1 =/= CONNECTORTYPE2 .

*** autres règles pour la première transformation
***
*** 3 règles

rl [transfo1.4] : **define port type** PORTTYPE1 `[ LTCHANNELS1 `] `{ PORTSPEC1 `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `]
=>
define port type PORTTYPE1 `[ LTCHANNELS1 `] `{ PORTSPEC1 `} .

rl [transfo1.5] : **define behaviour component type** COMPONENTBEHAVIOURTYPE1 `[ PTP1 `] `{ COMPBEHAVSPEC1 `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `]
=>
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PTP1 `] `{ COMPBEHAVSPEC1 `} .

rl [transfo1.6] : **define component type** COMPONENTTYPE2 `[ PTP1 `] `{ PORTDECL1 || COMPBEHAVDECL `} `[ COMPONENTTYPE1 @ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] / CONNECTORTYPE1 @ ROLE1 `: ROLETYPE1 `[ROLETYPEPARAM `] `]
=>
define component type COMPONENTTYPE2 `[ PTP1 `] `{ PORTDECL1 || COMPBEHAVDECL `}
.
***

*** **DEUXIEME TRANSFORMATION**
*** *on élimine les descriptions des ports supposés consistants avec les comportements*

*** ***hypothèses*** : on suppose que tous les ports sont attachés (pas de port libre) ;
*** on n'a pas 2 fois le même port dans une liste de paramètres de processus (ou une liste de noms de ports) ;
*** la première transformation s'effectuant avant la deuxième, toutes les demandes de renommage des ports de
*** connecteurs sont effectuées avant la première demande de suppression de port de composant ;
*** pas d' <operation name> @ <operation parameter>
***
```
var ARCHI : Architecture .
vars COMPOSITION1 COMPOSITION2 : CompositeName .
var COMPOSANT1 : ComponentName .
vars COMPONENTTYPE1 COMPONENTTYPE2 : ComponentTypeName .
var PORT1 : PortName .
var PORTTYPE1 : PortTypeName .
var CHANNEL1 : ChannelName .
var CHANNELTYPE1 : ChannelTypeImbrique .
vars PORTTYPEPARAM LTCHANNELS1 : ListOfTypedChannels .
vars LTPORTS1 LTPORTS2 : ListOfTypedPorts .
var AEDECL : ArchitecturalElementDeclarations .

var CLIST : ChannelsList .

var CONNECTEUR1 : ConnectorName .
vars CONNECTORTYPE1 CONNECTORTYPE2 : ConnectorTypeName .
var PORTDECL1 : PortDeclaration .
vars CONNECTORBEHAVIOUR1 COMPONENTBEHAVIOUR1 : BehaviourName .
var CONNECTORBEHAVIOURTYPE1 : ConnectorBehaviourTypeName .
var COMPONENTBEHAVIOURTYPE1 : ComponentBehaviourTypeName .

var LTPARAM1 : ListOfTypedParameters .
vars LPN1 : ListOfPortNames .
var CONNBEHA1 : ConnectorBehaviour .
var LCONNPROC1 : ListOfConnectorProcessus .
var COMPBEHA1 : ComponentBehaviour .
var LOPSPEC1 : ListOfOperationSpecifications .
var LCOMPPROC1 : ListOfComponentProcessus .
var PORTSPEC1 : PortSpecification .


*** équations complémentaires pour la deuxième transformation
***
*** 88 équations

*** CONNBEHA1 `[`- PORT1 `-`]
vars X Y Z : PortName .
vars M N : ConnectorBehaviour .
vars CM CN : ChoiceConnectorBehaviour .
vars PN PN1 : ParameterName .
var EWWC : ElementWithWhichCompared .
var CBTN : ConnectorBehaviourTypeName .
var LPN : ListOfPortNames .
var C : Channel .
var PROCESS : ProcessName .
var LPP : ListOfProcessParameters .

eq `( M `) `[`- X `-`] = `( M `[`- X `-`] `) .
eq ( M ¤ N ) `[`- X `-`] = M `[`- X `-`] ¤ N `[`- X `-`] .
eq ( CM + CN ) `[`- X `-`] = CM `[`- X `-`] + CN `[`- X `-`] .
eq ( `[ PN `= EWWC `] M ) `[`- X `-`] = `[ PN `= EWWC `] M `[`- X `-`] .
eq ( `[ PN `> EWWC `] M ) `[`- X `-`] = `[ PN `> EWWC `] M `[`- X `-`] .
eq ( `[ PN `< EWWC `] M ) `[`- X `-`] = `[ PN `< EWWC `] M `[`- X `-`] .
eq ( `[ PN `>`= EWWC `] M ) `[`- X `-`] = `[ PN `>`= EWWC `] M `[`- X `-`] .
eq ( `[ PN `<`= EWWC `] M ) `[`- X `-`] = `[ PN `<`= EWWC `] M `[`- X `-`] .
eq ( `[ PN `<`> EWWC `] M ) `[`- X `-`] = `[ PN `<`> EWWC `] M `[`- X `-`] .
eq ( M | N ) `[`- X `-`] = M `[`- X `-`] | N `[`- X `-`] .
eq ( CBTN `[ X `, LPN `] ) `[`- X `-`] = CBTN `[ LPN `] .
ceq ( CBTN `[ Y `, LPN `] ) `[`- X `-`] = CBTN `[ ( Y `, LPN ) `[`- X `-`] `] if X =/= Y .
eq ( CBTN `[ X `] ) `[`- X `-`] = CBTN .
ceq ( CBTN `[ Y `] ) `[`- X `-`] = CBTN `[ Y `] if X =/= Y .
ceq ( Y `, X `, LPN ) `[`- X `-`] = Y `, LPN if Y =/= X .
ceq ( Y `, Z `, LPN ) `[`- X `-`] = Y `, ( Z `, LPN ) `[`- X `-`] if Y =/= X and Z =/= X .
ceq ( Y `, X ) `[`- X `-`] = Y if Y =/= X .
ceq ( Y `, Z ) `[`- X `-`] = Y `, Z if Y =/= X and Z =/= X .
eq $ `[`- X `-`] = $ .
eq ( Y @ C ) `[`- X `-`] = Y @ C .
eq ( PROCESS `[`] ) `[`- X `-`] = PROCESS .
eq ( PROCESS `[ X `, LPP `] ) `[`- X `-`] = PROCESS `[ LPP `] .
ceq ( PROCESS `[ Y `, LPP `] ) `[`- X `-`] = PROCESS `[ ( Y `, LPP ) `[`- X `-`] `] if X =/= Y .
eq ( PROCESS `[ PN `, LPP `] ) `[`- X `-`] = PROCESS `[ ( PN `, LPP ) `[`- X `-`] `] .
eq ( PROCESS `[ X `] ) `[`- X `-`] = PROCESS .
ceq ( PROCESS `[ Y `] ) `[`- X `-`] = PROCESS `[ Y `] if X =/= Y .
eq ( PROCESS `[ PN `] ) `[`- X `-`] = PROCESS `[ PN `] .
ceq ( Y `, X `, LPP ) `[`- X `-`] = Y `, LPP if Y =/= X .
ceq ( Y `, Z `, LPP ) `[`- X `-`] = Y `, ( Z `, LPP ) `[`- X `-`] if Y =/= X and Z =/= X .
ceq ( Y `, PN `, LPP ) `[`- X `-`] = Y `, ( PN `, LPP ) `[`- X `-`] if Y =/= X .
eq ( PN `, X `, LPP ) `[`- X `-`] = PN `, LPP .
ceq ( PN `, Y `, LPP ) `[`- X `-`] = PN `, ( Y `, LPP ) `[`- X `-`] if Y =/= X .
eq ( PN `, PN1 `, LPP ) `[`- X `-`] = PN `, ( PN1 `, LPP ) `[`- X `-`] .
*** ceq ( Y `, X ) `[`- X `-`] = Y if Y =/= X .
*** ceq ( Y `, Z ) `[`- X `-`] = Y `, Z if Y =/= X and Z =/= X .
ceq ( Y `, PN ) `[`- X `-`] = Y `, PN if Y =/= X .
eq ( PN `, X ) `[`- X `-`] = PN .
ceq ( PN `, Y ) `[`- X `-`] = PN `, Y if Y =/= X .
eq ( PN `, PN1 ) `[`- X `-`] = PN `, PN1 .
```

```
eq M `[- X `-`] `[- Y `-`] = M `[- Y `-`] `[- X `-`] .
eq CM `[- X `-`] `[- Y `-`] = CM `[- Y `-`] `[- X `-`] .
eq LPN `[- X `-`] `[- Y `-`] = LPN `[- Y `-`] `[- X `-`] .
eq LPP `[- X `-`] `[- Y `-`] = LPP `[- Y `-`] `[- X `-`] .

*** LCONNPROC1 `[- PORT1 `-`]
vars LCP LCP1 : ListOfConnectorProcessus .
var LTPP : ListOfTypedProcessParameters .
vars POTYNA POTYNA1 : PortTypeName .
vars PATYNA PATYNA1 : ParameterTypeName .

eq ( LCP `, LCP1 ) `[- X `-`] = LCP `[- X `-`] `, LCP1 `[- X `-`] .
eq ( PROCESS `[ X `: POTYNA `] `= M ) `[- X `-`] = PROCESS `= M `[- X `-`] .
ceq ( PROCESS `[ Y `: POTYNA `] `= M ) `[- X `-`] = PROCESS `[ Y `: POTYNA `] `= M `[- X `-`]
if Y =/= X .
eq ( PROCESS `[ PN `: PATYNA `] `= M ) `[- X `-`] = PROCESS `[ PN `: PATYNA `] `= M `[- X `-
`] .
eq ( PROCESS `[ X `: POTYNA `, LTPP `] `= M ) `[- X `-`] = PROCESS `[ LTPP `] `= M `[- X `-`] .
ceq ( PROCESS `[ Y `: POTYNA `, LTPP `] `= M ) `[- X `-`] = PROCESS `[ ( Y `: POTYNA `, LTPP
) `[- X `-`] `] `= M `[- X `-`] if Y =/= X .
eq ( PROCESS `[ PN `: PATYNA `, LTPP `] `= M ) `[- X `-`] = PROCESS `[ ( PN `: PATYNA `,
LTPP ) `[- X `-`] `] `= M `[- X `-`] .
ceq ( Y `: POTYNA `, X `: POTYNA1 `, LTPP ) `[- X `-`] = Y `: POTYNA `, LTPP if Y =/= X .
ceq ( Y `: POTYNA `, Z `: POTYNA1 `, LTPP ) `[- X `-`] = Y `: POTYNA `, ( Z `: POTYNA1 `,
LTPP ) `[- X `-`] if Y =/= X and Z =/= X .
ceq ( Y `: POTYNA `, PN `: PATYNA `, LTPP ) `[- X `-`] = Y `: POTYNA `, ( PN `: PATYNA `,
LTPP ) `[- X `-`] if Y =/= X .
eq ( PN `: PATYNA `, X `: POTYNA `, LTPP ) `[- X `-`] = PN `: PATYNA `, LTPP .
ceq ( PN `: PATYNA `, Y `: POTYNA `, LTPP ) `[- X `-`] = PN `: PATYNA `, ( Y `: POTYNA `,
LTPP ) `[- X `-`] if Y =/= X .
eq ( PN `: PATYNA `, PN1 `: PATYNA1 `, LTPP ) `[- X `-`] = PN `: PATYNA `, ( PN1 `: PATYNA1
`, LTPP ) `[- X `-`] .
ceq ( Y `: POTYNA `, X `: POTYNA1 ) `[- X `-`] = Y `: POTYNA if Y =/= X .
ceq ( Y `: POTYNA `, Z `: POTYNA1 ) `[- X `-`] = Y `: POTYNA `, Z `: POTYNA1 if Y =/= X and Z
=/= X .
ceq ( Y `: POTYNA `, PN `: PATYNA ) `[- X `-`] = Y `: POTYNA `, PN `: PATYNA if Y =/= X .
eq ( PN `: PATYNA `, X `: POTYNA ) `[- X `-`] = PN `: PATYNA .
eq ( PN `: PATYNA `, Y `: POTYNA ) `[- X `-`] = PN `: PATYNA `, Y `: POTYNA if Y =/= X .
eq ( PN `: PATYNA `, PN1 `: PATYNA1 ) `[- X `-`] = PN `: PATYNA `, PN1 `: PATYNA1 .

eq LCP `[- X `-`] `[- Y `-`] = LCP `[- Y `-`] `[- X `-`] .
eq LTPP `[- X `-`] `[- Y `-`] = LTPP `[- Y `-`] `[- X `-`] .

*** COMPBEHA1 `[- PORT1 `-`]
vars A B : ComponentBehaviour .
vars MA MB : MatchingComponentBehaviour .
var COBTN : ComponentBehaviourTypeName .
var O : OperationName .
var LPAR : ListOfParameterNames .

eq `( A `) `[- X `-`] = `( A `[- X `-`] `) .
eq ( A ¤ B ) `[- X `-`] = A `[- X `-`] ¤ B `[- X `-`] .
eq ( O `[ LPAR `] ) `[- X `-`] = O `[ LPAR `] .
eq ( O `[ `] ) `[- X `-`] = O `[ `] .
eq ( MA + MB ) `[- X `-`] = MA `[- X `-`] + MB `[- X `-`] .
eq ( `[ PN `= EWWC `] A ) `[- X `-`] = `[ PN `= EWWC `] A `[- X `-`] .
eq ( `[ PN `> EWWC `] A ) `[- X `-`] = `[ PN `> EWWC `] A `[- X `-`] .
eq ( `[ PN `< EWWC `] A ) `[- X `-`] = `[ PN `< EWWC `] A `[- X `-`] .
eq ( `[ PN `>`= EWWC `] A ) `[- X `-`] = `[ PN `>`= EWWC `] A `[- X `-`] .
eq ( `[ PN `<`= EWWC `] A ) `[- X `-`] = `[ PN `<`= EWWC `] A `[- X `-`] .
eq ( `[ PN `<`> EWWC `] A ) `[- X `-`] = `[ PN `<`> EWWC `] A `[- X `-`] .
eq ( A | B ) `[- X `-`] = A `[- X `-`] | B `[- X `-`] .
eq ( COBTN `[ X `, LPN `] ) `[- X `-`] = COBTN `[ LPN `] .
ceq ( COBTN `[ Y `, LPN `] ) `[- X `-`] = COBTN `[ ( Y `, LPN ) `[- X `-`] `] if X =/= Y .
eq ( COBTN `[ X `] ) `[- X `-`] = COBTN .
ceq ( COBTN `[ Y `] ) `[- X `-`] = COBTN `[ Y `] if X =/= Y .

eq A `[- X `-`] `[- Y `-`] = A `[- Y `-`] `[- X `-`] .
eq MA `[- X `-`] `[- Y `-`] = MA `[- Y `-`] `[- X `-`] .

*** LCOMPPROC1 `[- PORT1 `-`]
vars LCO LCO1 : ListOfComponentProcessus .

eq ( LCO `, LCO1 ) `[- X `-`] = LCO `[- X `-`] `, LCO1 `[- X `-`] .
eq ( PROCESS `[ X `: POTYNA `] `= A ) `[- X `-`] = PROCESS `= A `[- X `-`] .
ceq ( PROCESS `[ Y `: POTYNA `] `= A ) `[- X `-`] = PROCESS `[ Y `: POTYNA `] `= A `[- X `-`]
if Y =/= X .
eq ( PROCESS `[ PN `: PATYNA `] `= A ) `[- X `-`] = PROCESS `[ PN `: PATYNA `] `= A `[- X `-
`] .
eq ( PROCESS `[ X `: POTYNA `, LTPP `] `= A ) `[- X `-`] = PROCESS `[ LTPP `] `= A `[- X `-`] .
ceq ( PROCESS `[ Y `: POTYNA `, LTPP `] `= A ) `[- X `-`] = PROCESS `[ ( Y `: POTYNA `, LTPP
) `[- X `-`] `] `= A `[- X `-`] if Y =/= X .
eq ( PROCESS `[ PN `: PATYNA `, LTPP `] `= A ) `[- X `-`] = PROCESS `[ ( PN `: PATYNA `,
LTPP ) `[- X `-`] `] `= A `[- X `-`] .

eq LCO `[- X `-`] `[- Y `-`] = LCO `[- Y `-`] `[- X `-`] .


*** première règle de transformation
***
*** decompose ou pas
*** channels au début ou pas
*** PORTTYPEPARAM ou pas (liste de paramètres de ports)
*** LTPORTS1 ou pas (liste de ports du composant)
*** LTPORTS2 ou pas (liste de ports du connecteur)
*** 2x2x2x2x2=32 règles
```

rl [transfo2.1.11111] : *** sans **decompose**, ni **channels** au départ, avec plusieurs paramètres et plusieurs ports
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}** .

rl [transfo2.1.21111] : *** **decompose**, sans **channels** au départ, avec plusieurs paramètres et plusieurs ports
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}** .

rl [transfo2.1.12111] : *** sans **decompose**, **channels** au départ, avec plusieurs paramètres et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}** .

rl [transfo2.1.22111] : *** **decompose**, **channels** au départ, avec plusieurs paramètres et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL `**}** .

rl [transfo2.1.11211] : *** sans **decompose**, sans **channels** au départ, avec 1 paramètre et plusieurs ports
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL `**}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL `**}** .

rl [transfo2.1.21211] : *** **decompose**, sans **channels** au départ, avec 1 paramètre et plusieurs ports
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL `**}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 `**{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL `**}** .

rl [transfo2.1.12211] : *** sans **decompose**, **channels** au départ, avec 1 paramètre et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL `**}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `**{** COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL `**}** .

rl [transfo2.1.22211] : *** **decompose**, **channels** au départ, avec 1 paramètre et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.11121] : *** sans **decompose**, sans **channels** au départ, avec plusieurs paramètres et 1 port
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.21121] : *** **decompose**, sans **channels** au départ, avec plusieurs paramètres et 1 port
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.12121] : *** sans **decompose**, **channels** au départ, avec plusieurs paramètres et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.22121] : *** **decompose**, **channels** au départ, avec plusieurs paramètres et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.11221] : *** sans **decompose**, sans **channels** au départ, avec 1 paramètre et 1 port
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.21221] : *** **decompose**, sans **channels** au départ, avec 1 paramètre et 1 port
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.12221] : *** sans **decompose**, **channels** au départ, avec 1 paramètre et 1 port
channels `{ CLIST `}

ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL **`}** .

rl [transfo2.1.22221] : *** **decompose**, **channels** au départ, avec 1 paramètre et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS2 `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `: CONNECTORTYPE1 `[LTPORTS2 `] || AEDECL **`}** .

*** plus de LPORTS2 !

rl [transfo2.1.11112] : *** sans **decompose**, ni **channels** au départ, avec plusieurs paramètres et plusieurs ports
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.21112] : *** **decompose**, sans **channels** au départ, avec plusieurs paramètres et plusieurs ports
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.12112] : *** sans **decompose**, **channels** au départ, avec plusieurs paramètres et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.22112] : *** **decompose**, **channels** au départ, avec plusieurs paramètres et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** **decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.11212] : *** sans **decompose**, sans **channels** au départ, avec 1 paramètre et plusieurs ports
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 || AEDECL **`}** .

rl [transfo2.1.21212] : *** **decompose**, sans **channels** au départ, avec 1 paramètre et plusieurs ports

ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 || AEDECL **`}** .

rl [transfo2.1.12212] : *** sans **decompose**, **channels** au départ, avec 1 paramètre et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 || AEDECL **`}** .

rl [transfo2.1.22212] : *** **decompose**, **channels** au départ, avec 1 paramètre et plusieurs ports
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `, LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ LTPORTS1 `] || CONNECTEUR1 `: CONNECTORTYPE1 || AEDECL **`}** .

rl [transfo2.1.11122] : *** sans **decompose**, sans **channels** au départ, avec plusieurs paramètres et 1 port
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.21122] : *** **decompose**, sans **channels** au départ, avec plusieurs paramètres et 1 port
ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.12122] : *** sans **decompose**, **channels** au départ, avec plusieurs paramètres et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.22122] : *** **decompose**, **channels** au départ, avec plusieurs paramètres et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `, PORTTYPEPARAM `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 **`{ decompose** COMPOSITION2 COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] || AEDECL **`}** .

rl [transfo2.1.11222] : *** sans **decompose**, sans **channels** au départ, avec 1 paramètre et 1 port
ARCHI
compose COMPOSITION1 **`{** COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[PORT1 `: PORTTYPE1 `[CHANNEL1 `: CHANNELTYPE1 `] `] || AEDECL **`}**
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}

```
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `:
CONNECTORTYPE1 || AEDECL `} .

rl [transfo2.1.21222] : *** decompose, sans channels au départ, avec 1 paramètre et 1 port
ARCHI
compose COMPOSITION1 `{ decompose COMPOSITION2 COMPOSANT1 `:
COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] ||
CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `:
CHANNELTYPE1 `] `] || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `}
ARCHI
compose COMPOSITION1 `{ decompose COMPOSITION2 COMPOSANT1 `:
COMPONENTTYPE1 || CONNECTEUR1 `: CONNECTORTYPE1 || AEDECL `} .

rl [transfo2.1.12222] : *** sans decompose, channels au départ, avec 1 paramètre et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[
CHANNEL1 `: CHANNELTYPE1 `] `] || CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `:
PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `{ COMPOSANT1 `: COMPONENTTYPE1 || CONNECTEUR1 `:
CONNECTORTYPE1 || AEDECL `} .

rl [transfo2.1.22222] : *** decompose, channels au départ, avec 1 paramètre et 1 port
channels `{ CLIST `}
ARCHI
compose COMPOSITION1 `{ decompose COMPOSITION2 COMPOSANT1 `:
COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `: CHANNELTYPE1 `] `] ||
CONNECTEUR1 `: CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ CHANNEL1 `:
CHANNELTYPE1 `] `] || AEDECL `}
=>
channels `{ PORT1 @ CHANNEL1 `: CHANNELTYPE1 `, CLIST `}
ARCHI
compose COMPOSITION1 `{ decompose COMPOSITION2 COMPOSANT1 `:
COMPONENTTYPE1 || CONNECTEUR1 `: CONNECTORTYPE1 || AEDECL `} .


*** groupe de deuxième règle de transformation
***
*** LTPORTS1 ou pas (liste de ports)
*** 2 règles

rl [transfo2.2.1] :
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `,
LTPORTS1 `] `{ port PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] || PORTDECL1 || behaviour
CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `, LTPORTS1 `] `}
=>
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `[ LTPORTS1 `] `{ port PORT1 || PORTDECL1 ||
behaviour CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `}
.

rl [transfo2.2.2] :
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `]
`{ port PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] || PORTDECL1 || behaviour
CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `] `}
=>
channels `{ CLIST `}
ARCHI
define connector type CONNECTORTYPE1 `{ port PORT1 || PORTDECL1 || behaviour
CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 `} .


*** groupe de troisième règle de transformation
***
*** LTPORTS1 ou pas (liste de ports)
*** avec PORTDECL1 ou pas (liste de déclaration de ports)
*** 3 règles

rl [transfo2.3.1] :
channels `{ CLIST `}
ARCHI
define component type COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `]
`, LTPORTS1 `] `{ port PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] || PORTDECL1 ||
behaviour COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `[ PORT1 `:
PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `}
=>
channels `{ CLIST `}
ARCHI
define component type COMPONENTTYPE1 `[ LTPORTS1 `] `{ port PORT1 || PORTDECL1 ||
behaviour COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] `}
.

rl [transfo2.3.2] :
channels `{ CLIST `}
ARCHI
```

define component type COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`{ port** PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] || PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`}**
=>
channels `{ CLIST `}
ARCHI
define component type COMPONENTTYPE1 **`{ port** PORT1 || PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}** .

rl [transfo2.3.3] :
channels `{ CLIST `}
ARCHI
define component type COMPONENTTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`{ port** PORT1 `: PORTTYPE1 `[PORTTYPEPARAM `] || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`}**
=>
channels `{ CLIST `}
ARCHI
define component type COMPONENTTYPE1 **`{ port** PORT1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}** .

*** groupe de quatrième règle de transformation
***
*** LTPORTS1 et LPN1 ou pas (listes de ports)
*** LCONNPROC1 ou pas (liste de processus)
*** LTPARAM1 ou pas (liste de paramètres)
*** 8 règles

rl [transfo2.4.111] : *** avec tout
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= CONNBEHA1 `, LCONNPROC1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 **`[`-** PORT1 **`-`]** `, LCONNPROC1 **`[`-** PORT1 **`-`] `}** .

rl [transfo2.4.211] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[PORT1 `] `= CONNBEHA1 `, LCONNPROC1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 **`[`-** PORT1 **`-`]** `, LCONNPROC1 **`[`-** PORT1 **`-`] `}** .

rl [transfo2.4.121] : *** sans processus de connecteur
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= CONNBEHA1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 **`[`-** PORT1 **`-`] `}** .

rl [transfo2.4.221] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `[PORT1 `] `= CONNBEHA1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 **`[`-** PORT1 **`-`] `}** .

rl [transfo2.4.112] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] **`{** CONNECTORBEHAVIOURTYPE1 `[PORT1 `, LPN1 `] `= CONNBEHA1 `, LCONNPROC1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] **`{** CONNECTORBEHAVIOURTYPE1 `[LPN1 `] `= CONNBEHA1 **`[`-** PORT1 **`-`]** `, LCONNPROC1 **`[`-** PORT1 **`-`] `}** .

rl [transfo2.4.212] :
channels `{ CLIST `}

```
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `] `{ CONNECTORBEHAVIOURTYPE1 `[ PORT1 `] `= CONNBEHA1 `,
LCONNPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `{
CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 `[`- PORT1 `-`] `, LCONNPROC1 `[`- PORT1
`-`] `} .

rl [transfo2.4.122] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `, LTPORTS1 `] `{ CONNECTORBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `]
`= CONNBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ LTPORTS1 `] `{
CONNECTORBEHAVIOURTYPE1 `[ LPN1 `] `= CONNBEHA1 `[`- PORT1 `-`] `} .

rl [transfo2.4.222] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `] `{ CONNECTORBEHAVIOURTYPE1 `[ PORT1 `] `= CONNBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 `{
CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 `[`- PORT1 `-`] `} .


*** groupe de cinquième règle de transformation
***
*** LTPORTS1 et LPN1 ou pas (liste de ports)
*** LOPSPEC1 ou pas (liste d'opérations internes)
*** LCOMPPROC1 ou pas (liste de processus)
*** LTPARAM1 ou pas (liste de paramètres)
*** 16 règles

rl [transfo2.5.1111] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `, LTPORTS1 `] `{ LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `[
PORT1 `, LPN1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] `{
LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `[ LPN1 `] `= COMPBEHA1 `[`- PORT1 `-`] `,
LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2111] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `] `{ LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `[ PORT1 `] `=
COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `,
COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `, LCOMPPROC1 `[`-
PORT1 `-`] `} .

rl [transfo2.5.1211] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `, LTPORTS1 `] `{ LTPARAM1 `, LOPSPEC1 `,
COMPONENTBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] `{
LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[ LPN1 `] `= COMPBEHA1 `[`-
PORT1 `-`] `, LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2211] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[
PORTTYPEPARAM `] `] `{ LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[
PORT1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `,
LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `,
LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.1121] :
channels `{ CLIST `}
ARCHI
```

define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `{ LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= COMPBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `[ LPN1 `] `= COMPBEHA1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2121] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] `{ LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `[PORT1 `] `= COMPBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `} .

rl [transfo2.5.1221] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `{ LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[PORT1 `, LPN1 `] `= COMPBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[LPN1 `] `= COMPBEHA1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2221] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] `{ LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[ PORT1 `] `= COMPBEHA1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `} .

rl [transfo2.5.1112] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `{ COMPONENTBEHAVIOURTYPE1 `[PORT1 `, LPN1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ COMPONENTBEHAVIOURTYPE1 `[LPN1 `] `= COMPBEHA1 `[`- PORT1 `-`] `, LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2112] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] `{ COMPONENTBEHAVIOURTYPE1 `[ PORT1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `, LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.1212] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] `{ LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] `{ LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[ LPN1 `] `= COMPBEHA1 `[`- PORT1 `-`] `, LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.2212] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] `{ LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[PORT1 `] `= COMPBEHA1 `, LCOMPPROC1 `}
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `{ LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `[`- PORT1 `-`] `, LCOMPPROC1 `[`- PORT1 `-`] `} .

rl [transfo2.5.1122] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] **`{** COMPONENTBEHAVIOURTYPE1 `[PORT1 `, LPN1 `] `= COMPBEHA1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] **`{** COMPONENTBEHAVIOURTYPE1 `[LPN1 `] `= COMPBEHA1 **`[`- PORT1 `-`] `}** .

rl [transfo2.5.2122] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`{** COMPONENTBEHAVIOURTYPE1 `[ PORT1 `] `= COMPBEHA1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`[`- PORT1 `-`] `}** .

rl [transfo2.5.1222] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `, LTPORTS1 `] **`{** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[ PORT1 `, LPN1 `] `= COMPBEHA1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ LTPORTS1 `] **`{** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[ LPN1 `] `= COMPBEHA1 **`[`- PORT1 `-`] `}** .

rl [transfo2.5.2222] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 `[ PORT1 `: PORTTYPE1 `[ PORTTYPEPARAM `] `] **`{** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `[PORT1 `] `= COMPBEHA1 **`}**
=>
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`[`- PORT1 `-`] `}** .

*** sixième règle de transformation
***
*** 1 règle

rl [transfo2.6] :
channels `{ CLIST `}
ARCHI
define port type PORTTYPE1 `[ LTCHANNELS1 `] **`{** PORTSPEC1 **`}**
=>
channels `{ CLIST `}
ARCHI .

***

*** **TROISIEME TRANSFORMATION**

*** ***hypothèses*** :
*** il n'y a pas d'appel de processus ou de comportement dans le membre de gauche d'une séquentialisation ;
*** les définitions de processus sont commutatives ;
*** les déclarations de types de canaux de communication sont commutatives ;
*** rappel : types de canaux monadiques ;
*** les déclarations d'opérations sont commutatives ;
*** le type imbriqué d'un canal est utilisé comme type de paramètre dans le composant et le connecteur
*** correspondants ;
*** les déclarations des ensembles de typage sont commutatives ;
*** les listes de types de paramètres et de paramètres typés sont commutatives
***
var .

*** équations complémentaires pour la troisième transformation
*** équations

*** première règle de transformation
***
*** LOPSPEC1 ou pas
*** LCOMPPROC1 ou pas
*** LTPARAM1 ou pas
*** 8 règles

rl [transfo3.1.111] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)
INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.1.211] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)
INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.1.121] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>

channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)
INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.1.221] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)
INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.1.112] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)
INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.1.212] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)

INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.1.122] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)
INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.1.222] :
channels `{ CLIST `}
ARCHI
define behaviour component type COMPONENTBEHAVIOURTYPE1 **`{** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`}**
define component type COMPONENTTYPE1 **`{** PORTDECL1 || **behaviour** COMPONENTBEHAVIOUR1 `: COMPONENTBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
SETS conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1) `= `{ index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0) `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1)
INVARIANT conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `: conc ('ETATS_ , COMPONENTBEHAVIOURTYPE1)
INITIALISATION conc ('var_etats_ , COMPONENTBEHAVIOURTYPE1) `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 0)
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE conc ('Comportement_ , COMPONENTTYPE1)
EXTENDS conc ('Actions_ , COMPONENTBEHAVIOURTYPE1)
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

*** groupe de deuxième règle de transformation
***
*** LCONNPROC1 ou pas
*** LTPARAM1 ou pas
*** 4 règles

rl [transfo3.2.11] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 `, LCONNPROC1 **`}**
define connector type CONNECTORTYPE1 **`{** PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Comportement_ , CONNECTORTYPE1)
SETS CONNECTEURS `= `{ CONNECTORBEHAVIOURTYPE1 `} `;
ETATS_CONNECTEURS `= `{ index (conc (conc ('etat_ , CONNECTORBEHAVIOURTYPE1) , '_) , 0) `<+ CONNECTORBEHAVIOURTYPE1 0 +`> `} `;
`[+`- LTPARAM1 `-+`]

EXTENDS
VARIABLES var_etats_connecteurs
INVARIANT var_etats_connecteurs `: CONNECTEURS --> ETATS_CONNECTEURS
INITIALISATION var_etats_connecteurs `:`= `{ CONNECTORBEHAVIOURTYPE1 |`->` index ( conc ( conc ( 'etat_ , CONNECTORBEHAVIOURTYPE1 ) , '_ ) , 0 ) `}
OPERATIONS
CONNECTORTYPE1 `= `[`[+ CONNBEHA1 `, LCONNPROC1 +`]`]
END .

rl [transfo3.2.21] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 **`{** LTPARAM1 `, CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 **`}**
define connector type CONNECTORTYPE1 **`{** PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Comportement_ , CONNECTORTYPE1)
SETS CONNECTEURS `= `{ CONNECTORBEHAVIOURTYPE1 `} `;
ETATS_CONNECTEURS `= `{ index (conc (conc ('etat_ , CONNECTORBEHAVIOURTYPE1) , '_) , 0) `<+ CONNECTORBEHAVIOURTYPE1 0 +`> `} `;
`[+`- LTPARAM1 `-+`]
EXTENDS
VARIABLES var_etats_connecteurs
INVARIANT var_etats_connecteurs `: CONNECTEURS --> ETATS_CONNECTEURS
INITIALISATION var_etats_connecteurs `:`= `{ CONNECTORBEHAVIOURTYPE1 |`->` index ( conc ( conc ( 'etat_ , CONNECTORBEHAVIOURTYPE1 ) , '_ ) , 0 ) `}
OPERATIONS
CONNECTORTYPE1 `= `[`[+ CONNBEHA1 +`]`]
END .

rl [transfo3.2.12] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 **`{** CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 `, LCONNPROC1 **`}**
define connector type CONNECTORTYPE1 **`{** PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Comportement_ , CONNECTORTYPE1)
SETS CONNECTEURS `= `{ CONNECTORBEHAVIOURTYPE1 `} `;
ETATS_CONNECTEURS `= `{ index (conc (conc ('etat_ , CONNECTORBEHAVIOURTYPE1) , '_) , 0) `<+ CONNECTORBEHAVIOURTYPE1 0 +`> `}
EXTENDS
VARIABLES var_etats_connecteurs
INVARIANT var_etats_connecteurs `: CONNECTEURS --> ETATS_CONNECTEURS
INITIALISATION var_etats_connecteurs `:`= `{ CONNECTORBEHAVIOURTYPE1 |`->` index ( conc ( conc ( 'etat_ , CONNECTORBEHAVIOURTYPE1 ) , '_ ) , 0 ) `}
OPERATIONS
CONNECTORTYPE1 `= `[`[+ CONNBEHA1 `, LCONNPROC1 +`]`]
END .

rl [transfo3.2.22] :
channels `{ CLIST `}
ARCHI
define behaviour connector type CONNECTORBEHAVIOURTYPE1 **`{** CONNECTORBEHAVIOURTYPE1 `= CONNBEHA1 **`}**
define connector type CONNECTORTYPE1 **`{** PORTDECL1 || **behaviour** CONNECTORBEHAVIOUR1 `: CONNECTORBEHAVIOURTYPE1 **`}**
=>
channels `{ CLIST `}
ARCHI
MACHINE conc ('Comportement_ , CONNECTORTYPE1)
SETS CONNECTEURS `= `{ CONNECTORBEHAVIOURTYPE1 `} `;
ETATS_CONNECTEURS `= `{ index (conc (conc ('etat_ , CONNECTORBEHAVIOURTYPE1) , '_) , 0) `<+ CONNECTORBEHAVIOURTYPE1 0 +`> `}
EXTENDS
VARIABLES var_etats_connecteurs
INVARIANT var_etats_connecteurs `: CONNECTEURS --> ETATS_CONNECTEURS
INITIALISATION var_etats_connecteurs `:`= `{ CONNECTORBEHAVIOURTYPE1 |`->` index ( conc ( conc ( 'etat_ , CONNECTORBEHAVIOURTYPE1 ) , '_ ) , 0 ) `}
OPERATIONS
CONNECTORTYPE1 `= `[`[+ CONNBEHA1 +`]`]
END .

*** groupe de troisième règle de transformation
***
*** TYPES1 ou pas
*** + 1 opérateur booléen de test d'absence d'un ensemble de typage dans une liste d'ensembles
*** LTPARAM1 ou pas
*** LOPSPEC1 ou pas
*** LCOMPPROC1 ou pas
*** 24 règles

rl [transfo3.3.1111] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0

```
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

crl [transfo3.3.2111] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `) .

crl [transfo3.3.3111] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
```

```
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if not ( nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `)
) .

rl [transfo3.3.1211] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

crl [transfo3.3.2211] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
```

```
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `) .

crl [transfo3.3.3211] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if not ( nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `)
) .

rl [transfo3.3.1121] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

crl [transfo3.3.2121] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
```

VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc (conc (PATYNA , '_) , COMPONENTBEHAVIOURTYPE1)
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if nouveau `( TYPES1 `, conc (conc (PATYNA , '_) , COMPONENTBEHAVIOURTYPE1) `) .

crl [transfo3.3.3121] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if not (nouveau `( TYPES1 `, conc (conc (PATYNA , '_) , COMPONENTBEHAVIOURTYPE1) `)) .

rl [transfo3.3.1221] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1

```
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

crl [transfo3.3.2221] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `) .

crl [transfo3.3.3221] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
```

```
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END
if not ( nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `)
) .

rl [transfo3.3.1112] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

crl [transfo3.3.2112] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
```

```
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `) .

crl [transfo3.3.3112] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if not ( nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `)
) .

rl [transfo3.3.1212] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

crl [transfo3.3.2212] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `) .

crl [transfo3.3.3212] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if not ( nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `)
) .

rl [transfo3.3.1122] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
```

```
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

crl [transfo3.3.2122] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `) .

crl [transfo3.3.3122] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA `, LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- LTPARAM1 COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
```

```
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if not ( nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `)
) .

rl [transfo3.3.1222] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

crl [transfo3.3.2222] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 )
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `) .

crl [transfo3.3.3222] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
`[*`- PN `: PATYNA COMPONENTBEHAVIOURTYPE1 `-*`]
VARIABLES VAR_ETATS1
```

```
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM2 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END
if not ( nouveau `( TYPES1 `, conc ( conc ( PATYNA , '_ ) , COMPONENTBEHAVIOURTYPE1 ) `)
) .


*** groupe de quatrième règle de transformation
***
*** 20 cas = 8
*** ETAT0 (ou plus)
*** pas LOPSPEC1 (ou pas)
*** LCOMPPROC1 (ou pas)
*** pas LTPARAM1 (ou pas)
***
*** 192 règles

rl [transfo3.4.1.1] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `( COMPBEHA1 `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.4.2.1] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
```

VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 `, LCOMPPROC1 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

*** rl [transfo3.4.3.1] et rl [transfo3.4.4.1] pas de nom d'opération dans le cas pas de LOPSPEC1
*** rl [transfo3.4.5.1] et rl [transfo3.4.6.1] pas de communication seule avec LCOMPPROC1 , sans paramètre
*** rl [transfo3.4.7.1] et rl [transfo3.4.8.1] pas de sens d'entamer avec des paramètres non initialisés

rl [transfo3.4.9.1] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 + MATCOMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 `, LCOMPPROC1 *`]`]
END .

*** rl [transfo3.4.10.1] à rl [transfo3.4.15.1] pas de test cause pas de paramètre

rl [transfo3.4.16.1] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 | COMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}

```
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .
```

rl [transfo3.4.17.1] : *** à composer

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= $ `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
`[`[* LCOMPPROC1 *`]`]
END .
```

rl [transfo3.4.18.1] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPONENTBEHAVIOURTYPE1 `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
`[`[* LCOMPPROC1 *`]`]
END .
```

rl [transfo3.4.19.1] : *** on doit garder une trace du passage par le processus principal du comportement
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , PROCESS1) , '_) , 0) END END
`[** PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
END .

rl [transfo3.4.20.1] : *** même chose sans processus supplémentaire
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , PROCESS1) , '_) , 0) END END
`[** PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 *`]`]
END
END .

*** groupe de quatrième règle (suite)
*** 20 cas = 6
*** ETAT0 (ou plus)
*** pas LOPSPEC1 (ou pas)
*** pas LCOMPPROC1 (ou pas)
*** pas LTPARAM1 (ou pas)

```
rl [transfo3.4.1.2] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `( COMPBEHA1 `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.2.2] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

*** rl [transfo3.4.3.2] et rl [transfo3.4.4.2] pas de nom d'opération dans le cas pas de LOPSPEC1
*** rl [transfo3.4.5.2] et rl [transfo3.4.6.2] pas de communication sans paramètre
*** rl [transfo3.4.7.2] et rl [transfo3.4.8.2] pas de sens d'entamer avec des paramètres non initialisés

rl [transfo3.4.9.2] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 + MATCOMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
```

INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 *`]`]
END .

*** rl [transfo3.4.10.2] à rl [transfo3.4.15.2] pas de test cause pas de paramètre

rl [transfo3.4.16.2] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 | COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.17.2] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= $ **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
END
MACHINE COMPORTEMENT1

```
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
END .

rl [transfo3.4.18.2] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** COMPONENTBEHAVIOURTYPE1 `= COMPONENTBEHAVIOURTYPE1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
END .

*** rl [transfo3.4.19.2] et rl [transfo3.4.20.2] pas de processus sans LCOMPPROC1

*** groupe de quatrième règle (suite)

*** 20 cas (+ 1) = 10
*** ETAT0 (ou plus)
*** LOPSPEC1 (ou pas)
*** LCOMPPROC1 (ou pas)
*** pas LTPARAM1 (ou pas)

rl [transfo3.4.1.3] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `( COMPBEHA1 `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.4.2.3] :
channels `{ CLIST `}
ARCHI
```

MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 `, LCOMPPROC1 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

*** rl [transfo3.4.3.3] pas sans paramètre

rl [transfo3.4.4.3] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** OPERATION1 `[`]`{`.`.`.`} `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[`] `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || skip END
`[** LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
`[`[* LCOMPPROC1 *`]`]
END
END .

rl [transfo3.4.4b.3] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** OPERATION1 `[`]`{`.`.`.`} `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[`] `, LCOMPPROC1 **`]
END

```
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END
|| skip END
`[** LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
`[`[* LCOMPPROC1 *`]`]
END
END .
```

*** rl [transfo3.4.5.3] et rl [transfo3.4.6.3] pas de communication seule avec LCOMPPROC1 , sans paramètre
*** rl [transfo3.4.7.3] et rl [transfo3.4.8.3] pas de sens d'entamer avec des paramètres non initialisés

rl [transfo3.4.9.3] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 + MATCOMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 `, LCOMPPROC1 *`]`]
END .
```

*** rl [transfo3.4.10.3] à rl [transfo3.4.15.3] pas de test cause pas de paramètre

rl [transfo3.4.16.3] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 | COMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
```

```
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.4.17.3] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= $ `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
`[`[* LCOMPPROC1 *`]`]
END .

rl [transfo3.4.18.3] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPONENTBEHAVIOURTYPE1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
```

EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
`[`[* LCOMPPROC1 *`]`]
END .

rl [transfo3.4.19.3] : *** on doit garder une trace du passage par le processus principal du comportement
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , PROCESS1) , '_) , 0) END END
`[** LOPSPEC1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
END .

rl [transfo3.4.20.3] : *** même chose sans processus supplémentaire
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , PROCESS1) , '_) , 0) END END
`[** LOPSPEC1 `, PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 *`]`]
END

```
END .

*** groupe de quatrième règle (suite)
*** 20 cas (+ 1) = 8
*** ETAT0 (ou plus)
*** LOPSPEC1 (ou pas)
*** pas LCOMPPROC1 (ou pas)
*** pas LTPARAM1 (ou pas)

rl [transfo3.4.1.4] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `( COMPBEHA1 `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.2.4] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

*** rl [transfo3.4.3.4] pas sans paramètre

rl [transfo3.4.4.4] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
```

```
OPERATIONS
`[** OPERATION1 `[`]`{`.`.`.`} `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[`] **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END
|| skip END
`[** LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
END
END .
```

rl [transfo3.4.4b.4] : *** nouvel état qui doit servir comme le précédent en cas de composition

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** OPERATION1 `[`]`{`.`.`.`} `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[`] **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END
|| skip END
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
END
END .
```

*** rl [transfo3.4.5.4] et rl [transfo3.4.6.4] pas de communication sans paramètre
*** rl [transfo3.4.7.4] et rl [transfo3.4.8.4] pas de sens d'entamer avec des paramètres non initialisés

rl [transfo3.4.9.4] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 + MATCOMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
```

```
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `=
MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 *`]`]
END .
```

*** rl [transfo3.4.10.4] à rl [transfo3.4.15.4] pas de test cause pas de paramètre

```
rl [transfo3.4.16.4] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 | COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.17.4] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= $ **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 **`]
```

```
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
END .
```

rl [transfo3.4.18.4] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPONENTBEHAVIOURTYPE1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `}
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
END .
```

*** rl [transfo3.4.19.4] et rl [transfo3.4.20.4] pas de processus sans LCOMPPROC1

*** groupe de quatrième règle (suite)
*** 20 cas (+ 22) = 40
*** ETAT0 (ou plus)
*** pas LOPSPEC1 (ou pas)
*** LCOMPPROC1 (ou pas)
*** LTPARAM1 (ou pas)

rl [transfo3.4.1.5] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `( COMPBEHA1 `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .
```

rl [transfo3.4.2.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 `, LCOMPPROC1 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

*** rl [transfo3.4.3.5] et rl [transfo3.4.4.5] pas de nom d'opération dans le cas pas de LOPSPEC1

crl [transfo3.4.5.5] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 ‖ conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END ‖ conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.5t.5] : *** sans autre type
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}

```
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5pt.5] : *** sans autre paramètre (et donc type)

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
```

```
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5c.5] : *** sans autre canal (dernier composant)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5ct.5] : *** sans autre type ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
```

INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.5cpt.5] : *** sans autre paramètre (et donc type) ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END
`[** PN `: PATYNA `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6.5] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `(PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END

```
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6t.5] : *** sans autre type
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6pt.5] : *** sans autre paramètre (et donc type)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.6c.5] : *** sans autre canal (dernier composant)

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.6ct.5] : *** sans autre type ou canal

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
```

```
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{
ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_
, COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc
( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc (
COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6cpt.5] : *** sans autre paramètre (et donc type) ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{
ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_
, COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc
( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc (
COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

rl [transfo3.4.7.5] : *** (rappel) on suppose la description oiginale correcte
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `[ LPP1 `] `, PROCESS1 `[
LTPP1 `] `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
```

```
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `[ LPP1 `] `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 `,
LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+
COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN
VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) END END
`[** LTPARAM1 `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1 || `[`= PROCESS1 LTPP1
`= COMPONENTBEHAVIOURTYPE1 LPP1 `=`]
`[`[* PROCESS1 `[ LTPP1 `] `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
END .
```

rl [transfo3.4.8.5] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `[ LPP1 `] `, PROCESS1 `[
LTPP1 `] `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `[ LPP1 `] `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1
*`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+
COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN
VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) END END
`[** LTPARAM1 `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1 || `[`= PROCESS1 LTPP1
`= COMPONENTBEHAVIOURTYPE1 LPP1 `=`]
`[`[* PROCESS1 `[ LTPP1 `] `= COMPBEHA1 *`]`]
END
END .
```

rl [transfo3.4.9.5] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 +
MATCOMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
```

```
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.4.10.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.10t.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
```

OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.10pt.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.11.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS

```
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.11t.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.11pt.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `] COMPBEHA1
`, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.12.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
```

INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.12t.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.12pt.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION

```
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.13.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.13t.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.13pt.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `] COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.14.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
```

```
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .
```

```
rl [transfo3.4.14t.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .
```

```
rl [transfo3.4.14pt.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `] COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .
```

```
rl [transfo3.4.15.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
```

```
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.15t.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.15pt.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `] COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
```

```
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .
```

```
rl [transfo3.4.16.5] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 | COMPBEHA2 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .
```

```
rl [transfo3.4.17.5] : *** à composer
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= $ `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LCOMPPROC1 **`]
```

```
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
`[`[* LCOMPPROC1 *`]`]
END .
```

rl [transfo3.4.18.5] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPONENTBEHAVIOURTYPE1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
`[`[* LCOMPPROC1 *`]`]
END .
```

rl [transfo3.4.19.5] : *** on doit garder une trace du passage par le processus principal du comportement

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+
COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN
VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) END END
`[** LTPARAM1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
END .
```

rl [transfo3.4.20.5] : *** même chose sans processus supplémentaire

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) END END
`[** LTPARAM1 `, PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 *`]`]
END
END .
```

*** groupe de quatrième règle (suite)
*** 20 cas (+ 22) = 36
*** ETAT0 (ou plus)
*** pas LOPSPEC1 (ou pas)
*** pas LCOMPPROC1 (ou pas)
*** LTPARAM1 (ou pas)

rl [transfo3.4.1.6] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `( COMPBEHA1 `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .
```

rl [transfo3.4.2.6] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .
```

*** rl [transfo3.4.3.6] et rl [transfo3.4.4.6] pas de nom d'opération dans le cas pas de LOPSPEC1

crl [transfo3.4.5.6] : *** nouvel état qui doit servir comme le précédent en cas de composition

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 ‖ conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END ‖ conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5t.6] : *** sans autre type

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
```

```
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5pt.6] : *** sans autre paramètre (et donc type)

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5c.6] : *** sans autre canal (dernier composant)

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5ct.6] : *** sans autre type ou canal

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5cpt.6] : *** sans autre paramètre (et donc type) ou canal

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
```

VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END
`[** PN `: PATYNA **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6.6] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `(PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `<`-`- conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_out)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

```
crl [transfo3.4.6t.6] : *** sans autre type
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6pt.6] : *** sans autre paramètre (et donc type)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
```

COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `<`-`- conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_out)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6c.6] : *** sans autre canal (dernier composant)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `(PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `<`-`- conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_out)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6ct.6] : *** sans autre type ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `(PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 **`]
END
MACHINE COMPORTEMENT1

```
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.6cpt.6] : *** sans autre paramètre (et donc type) ou canal

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA **`]
END
```

```
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

*** rl [transfo3.4.7.6] et rl [transfo3.4.8.6] pas sans LCOMPPROC1

rl [transfo3.4.9.6] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 + MATCOMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
```

```
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 *`]`]
END .

rl [transfo3.4.10.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.10t.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.10pt.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
```

```
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `] COMPBEHA1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.11.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.11t.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
```

```
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.11pt.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `] COMPBEHA1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>`= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.12.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>`= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.12t.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>`= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.12pt.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>`= EWWC `] COMPBEHA1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .
```

```
rl [transfo3.4.13.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.13t.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.13pt.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `] COMPBEHA1
**`]
```

```
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.14.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.14t.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
```

```
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.14pt.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `] COMPBEHA1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.15.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
```

```
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .
```

```
rl [transfo3.4.15t.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .
```

```
rl [transfo3.4.15pt.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `] COMPBEHA1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .
```

```
rl [transfo3.4.16.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 | COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.17.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= $ **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
END .

rl [transfo3.4.18.6] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, COMPONENTBEHAVIOURTYPE1 `= COMPONENTBEHAVIOURTYPE1
**`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
```

INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
END .

*** rl [transfo3.4.19.6] et rl [transfo3.4.20.6] pas de processus sans LCOMPPROC1

*** groupe de quatrième règle (suite)
*** 20 cas (+ 24) = 44
*** ETAT0 (ou plus)
*** LOPSPEC1 (ou pas)
*** LCOMPPROC1 (ou pas)
*** LTPARAM1 (ou pas)

rl [transfo3.4.1.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `(COMPBEHA1 `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.4.2.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 `, LCOMPPROC1 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1

VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.3.7] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, OPERATION1 `[ LOPPARAM1 `]`{`.`.`.`} `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[ LPARAM1 `] `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[LPARAM1 `] `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `[O`[LOPPARAM1 COMPONENTBEHAVIOURTYPE1 `]O`] `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || skip END
`[** LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1 `[P`[LPARAM1 COMPONENTBEHAVIOURTYPE1 LTPARAM1 `]P`]
`[`[* LCOMPPROC1 *`]`]
END
END .

rl [transfo3.4.3b.7] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, OPERATION1 `[ LOPPARAM1 `]`{`.`.`.`} `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[LPARAM1 `] `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[LPARAM1 `] `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `[O`[LOPPARAM1 COMPONENTBEHAVIOURTYPE1 `]O`] `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || skip END
`[** LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1 `[P`[LPARAM1 COMPONENTBEHAVIOURTYPE1 LTPARAM1 `]P`]

```
`[`[* LCOMPPROC1 *`]`]
END
END .
```

rl [transfo3.4.4.7] : *** nouvel état qui doit servir comme le précédent en cas de composition

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, OPERATION1 `[`]`{`.`.`.`} `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1
`= OPERATION1 `[`] `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN
VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END
|| skip END
`[** LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
`[`[* LCOMPPROC1 *`]`]
END
END .
```

rl [transfo3.4.4b.7] : *** nouvel état qui doit servir comme le précédent en cas de composition

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, OPERATION1 `[`]`{`.`.`.`} `, COMPONENTBEHAVIOURTYPE1 `=
OPERATION1 `[`] `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN
VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END
|| skip END
`[** LTPARAM1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
`[`[* LCOMPPROC1 *`]`]
END
END .
```

crl [transfo3.4.5.7] : *** nouvel état qui doit servir comme le précédent en cas de composition

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
```

```
ARCHI
MACHINE ACTIONS1
SETS  ETATS1  `=  `{  ETAT0  `<+  COMPONENTBEHAVIOURTYPE1  0  +`>  `}  `;  TYPES1  `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION  VAR_ETATS1  `:`=  ETAT0  ||  conc  (  conc  (  PORT1  ,  '_  )  ,  CHANNEL1  )  `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
```

if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.5t.7] : *** sans autre type

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION  VAR_ETATS1  `:`=  ETAT0  ||  conc  (  conc  (  PORT1  ,  '_  )  ,  CHANNEL1  )  `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
```

```
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5pt.7] : *** sans autre paramètre (et donc type)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5c.7] : *** sans autre canal (dernier composant)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
```

```
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5ct.7] : *** sans autre type ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END

MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5cpt.7] : *** sans autre paramètre (et donc type) ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @
CHANNEL1 `< PN `> `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
```

```
`[** PN `: PATYNA `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6.7] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{
ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_
, COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc
( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6t.7] : *** sans autre type
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
```

```
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.6pt.7] : *** sans autre paramètre (et donc type)

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.6c.7] : *** sans autre canal (dernier composant)

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
```

```
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{
ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_
, COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc
( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc (
COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6ct.7] : *** sans autre type ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{
ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_
, COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc
( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc (
COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6cpt.7] : *** sans autre paramètre (et donc type) ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @
CHANNEL1 `( PN `) `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
```

```
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) `, LCOMPPROC1 *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
`[`[* LCOMPPROC1 *`]`]
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

rl [transfo3.4.7.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `[ LPP1 `] `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `[ LPP1 `] `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) END END
`[** LTPARAM1 `, LOPSPEC1 `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1 || `[`= PROCESS1 LTPP1 `= COMPONENTBEHAVIOURTYPE1 LPP1 `=`]
`[`[* PROCESS1 `[ LTPP1 `] `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
END .

rl [transfo3.4.8.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `[ LPP1 `] `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 **`]
```

```
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `[ LPP1 `] `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1
*`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+
COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN
VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) END END
`[** LTPARAM1 `, LOPSPEC1 `, PROCESS1 `[ LTPP1 `] `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1 || `[`= PROCESS1 LTPP1
`= COMPONENTBEHAVIOURTYPE1 LPP1 `=`]
`[`[* PROCESS1 `[ LTPP1 `] `= COMPBEHA1 *`]`]
END
END .

rl [transfo3.4.9.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 +
MATCOMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `=
MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.4.10.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
```

```
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.10t.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.10pt.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
```

`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.11.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.11t.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN

```
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.11pt.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.12.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.12t.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
```

```
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.12pt.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>`= EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.13.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<
EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
```

OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.13t.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.13pt.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `< EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0

```
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.14.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>
EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.14t.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>
EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
```

```
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.14pt.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.15.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `=
EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.15t.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
```

```
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `=
EWWC `] COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.15pt.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `]
COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `,
LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END END
END .

rl [transfo3.4.16.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 |
COMPBEHA2 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
```

OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 `, LCOMPPROC1 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 `, LCOMPPROC1 *`]`]
END .

rl [transfo3.4.17.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= $ `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
`[`[* LCOMPPROC1 *`]`]
END .

rl [transfo3.4.18.7] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPONENTBEHAVIOURTYPE1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1

VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
`[`[* LCOMPPROC1 *`]`]
END .

rl [transfo3.4.19.7] : *** on doit garder une trace du passage par le processus principal du comportement
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , PROCESS1) , '_) , 0) END END
`[** LTPARAM1 `, LOPSPEC1 `, PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 `, LCOMPPROC1 *`]`]
END
END .

rl [transfo3.4.20.7] : *** même chose sans processus supplémentaire
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PROCESS1 `, PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PROCESS1 `, PROCESS1 `= COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `, index ( conc ( conc ( 'etat_ , PROCESS1 ) , '_ ) , 0 ) `<+ COMPONENTBEHAVIOURTYPE1 0 + PROCESS1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
PROCESS1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , PROCESS1) , '_) , 0) END END
`[** LTPARAM1 `, LOPSPEC1 `, PROCESS1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN PROCESS1
`[`[* PROCESS1 `= COMPBEHA1 *`]`]
END
END .

*** groupe de quatrième règle (suite)
*** 20 cas (+ 24) = 40
*** ETAT0 (ou plus)
*** LOPSPEC1 (ou pas)
*** pas LCOMPPROC1 (ou pas)
*** LTPARAM1 (ou pas)

rl [transfo3.4.1.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `( COMPBEHA1 `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `(COMPBEHA1 `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.2.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 ¤ COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 ¤ COMPBEHA2 *`]`]
END
=>
`[¤ COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 ¤`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.3.8] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS

`[** LTPARAM1 `, OPERATION1 `[ LOPPARAM1 `]`{`.`.`.`} `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[ LPARAM1 `] **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[LPARAM1 `] *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `[O`[LOPPARAM1 COMPONENTBEHAVIOURTYPE1 `]O`] `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || skip END
`[** LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1 `[P`[LPARAM1 COMPONENTBEHAVIOURTYPE1 LTPARAM1 `]P`]
END
END .

rl [transfo3.4.3b.8] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, OPERATION1 `[ LOPPARAM1 `]`{`.`.`.`} `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[LPARAM1 `] **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[LPARAM1 `] *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `[O`[LOPPARAM1 COMPONENTBEHAVIOURTYPE1 `]O`] `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || skip END
`[** LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1 `[P`[LPARAM1 COMPONENTBEHAVIOURTYPE1 LTPARAM1 `]P`]
END
END .

rl [transfo3.4.4.8] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, OPERATION1 `[`]`{`.`.`.`} `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[`] **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES

INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || skip END
`[** LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
END
END .

rl [transfo3.4.4b.8] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, OPERATION1 `[`]`{`.`.`.`} `, COMPONENTBEHAVIOURTYPE1 `= OPERATION1 `[`] **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* OPERATION1 `[`] *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
OPERATION1 `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || skip END
`[** LTPARAM1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN OPERATION1
END
END .

crl [transfo3.4.5.8] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
channels `{ CLIST `}

```
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5t.8] : *** sans autre type

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .
```

crl [transfo3.4.5pt.8] : *** sans autre paramètre (et donc type)

```
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @
CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
```

```
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5c.8] : *** sans autre canal (dernier composant)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ )
, 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`:
PATYNA
OPERATIONS
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0
`} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc (
conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc ( conc (
PORT1 , '_ ) , CHANNEL1 ) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( conc ( PORT1 , '_ ) ,
CHANNEL1 ) , '_in ) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.5ct.8] : *** sans autre type ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1
@ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
```

VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.5cpt.8] : *** sans autre paramètre (et donc type) ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `< PN `> **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `< PN `> *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE VAR_ETATS1 `: `{ ETAT0 `} & PN `: PATYNA THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END ;
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= BEGIN PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END
`[** PN `: PATYNA `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6.8] : *** nouvel état qui doit servir comme le précédent en cas de composition
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) **`]

```
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6t.8] : *** sans autre type
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `( PN `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc ( conc ( PORT1 , '_ ) , CHANNEL1 )
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index ( conc ( conc ( 'etat_ , COMPONENTBEHAVIOURTYPE1 ) , '_ ) , 1 ) END || PN `:`= conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) END ;
conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_in ) `( PN `) `= PRE PN `: PATYNA THEN conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN )
INVARIANT conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `: PATYNA
INITIALISATION conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc ( conc ( COMPONENTBEHAVIOURTYPE1 , '_ ) , PN ) `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out )
END
END
if conc ( conc ( CHANNELTYPE1 , '_ ) , COMPONENTBEHAVIOURTYPE1 ) == PATYNA .

crl [transfo3.4.6pt.8] : *** sans autre paramètre (et donc type)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `, CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
```

VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `(PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END
`[** PN `: PATYNA `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `<`-`- conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_out)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6c.8] : *** sans autre canal (dernier composant)
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `<`-`- conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_out)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6ct.8] : *** sans autre type ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `( PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `<`-`- conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_out)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

crl [transfo3.4.6cpt.8] : *** sans autre paramètre (et donc type) ou canal
channels `{ PORT1 @ CHANNEL1 `: `[ CHANNELTYPE1 `] `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= PORT1 @ CHANNEL1 `(PN `) **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* PORT1 @ CHANNEL1 `(PN `) *`]`]
END
=>
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 , index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) `<+ COMPONENTBEHAVIOURTYPE1 1 +`> `} `; PATYNA
VARIABLES VAR_ETATS1 `, conc (conc (PORT1 , '_) , CHANNEL1)
INVARIANT VAR_ETATS1 `: ETATS1 & conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) `: PATYNA
INITIALISATION VAR_ETATS1 `:`= ETAT0 || conc (conc (PORT1 , '_) , CHANNEL1) `:`: PATYNA
OPERATIONS
PN `<`-`- conc ( conc ( conc ( PORT1 , '_ ) , CHANNEL1 ) , '_out ) `= PRE VAR_ETATS1 `: `{ ETAT0 `} THEN IF VAR_ETATS1 `= ETAT0 THEN VAR_ETATS1 `:`= index (conc (conc ('etat_ , COMPONENTBEHAVIOURTYPE1) , '_) , 1) END || PN `:`= conc (conc (PORT1 , '_) , CHANNEL1) END ;
conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_in) `( PN `) `= PRE PN `: PATYNA THEN conc (conc (PORT1 , '_) , CHANNEL1) `:`= PN END
`[** PN `: PATYNA `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN)
INVARIANT conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `: PATYNA
INITIALISATION conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `:`: PATYNA
OPERATIONS

COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN conc (conc (COMPONENTBEHAVIOURTYPE1 , '_) , PN) `<`-`- conc (conc (conc (PORT1 , '_) , CHANNEL1) , '_out)
END
END
if conc (conc (CHANNELTYPE1 , '_) , COMPONENTBEHAVIOURTYPE1) == PATYNA .

*** rl [transfo3.4.7.8] et rl [transfo3.4.8.8] pas sans LCOMPPROC1

rl [transfo3.4.9.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 + MATCOMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 + MATCOMPBEHA2 *`]`]
END
=>
`[+ COMPONENTBEHAVIOURTYPE1 0 + COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA2 +`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= MATCOMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* MATCOMPBEHA1 *`]`]
END .

rl [transfo3.4.10.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.10t.8] :
channels `{ CLIST `}

```
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`<`> EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.10pt.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`> EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `<`> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN /`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.11.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`<`= EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
```

VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.11t.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<`= EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.11pt.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[PN `<`= EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[PN `<`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA

```
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `<`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.12.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`>`= EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.12t.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN
`>`= EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
```

```
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.12pt.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>`= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `>`= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `>`= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.13.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<
EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.13t.8] :
channels `{ CLIST `}
ARCHI
```

```
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `<
EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.13pt.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `< EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `< EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `< EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.14.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `>
EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
```

```
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.14t.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.14pt.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `> EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `> EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
```

```
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `> EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.15.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `=
EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1 `;
PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.15t.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `=
EWWC `] COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
```

```
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.15pt.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= `[ PN `= EWWC `]
COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* `[ PN `= EWWC `] COMPBEHA1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; PATYNA
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** PN `: PATYNA `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= IF VAR_ETATS1 `= ETAT0 THEN IF PN `= EWWC THEN
`[`[* COMPBEHA1 *`]`]
END END
END .

rl [transfo3.4.16.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 |
COMPBEHA2 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 | COMPBEHA2 *`]`]
END
=>
`[| COMPONENTBEHAVIOURTYPE1 0 | COMPONENTBEHAVIOURTYPE1 `= COMPBEHA2 |`]
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= COMPBEHA1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPBEHA1 *`]`]
END .

rl [transfo3.4.17.8] :
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
```

```
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `= $ **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* $ *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip
END .
```

rl [transfo3.4.18.8] :

```
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 `, COMPONENTBEHAVIOURTYPE1 `=
COMPONENTBEHAVIOURTYPE1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= `[`[* COMPONENTBEHAVIOURTYPE1 *`]`]
END
=>
channels `{ CLIST `}
ARCHI
MACHINE ACTIONS1
SETS ETATS1 `= `{ ETAT0 `<+ COMPONENTBEHAVIOURTYPE1 0 +`> `} `; TYPES1
VARIABLES VAR_ETATS1
INVARIANT VAR_ETATS1 `: ETATS1
INITIALISATION VAR_ETATS1 `:`= ETAT0
OPERATIONS
`[** LTPARAM1 `, LOPSPEC1 **`]
END
MACHINE COMPORTEMENT1
EXTENDS ACTIONS1
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
COMPONENTTYPE1 `= skip /* COMPONENTBEHAVIOURTYPE1 */
END .
```

*** rl [transfo3.4.19.8] et rl [transfo3.4.20.8] pas de processus sans LCOMPPROC1

***

(…)

RESUME

L’architecture d’un logiciel décrit sa structure et son comportement en termes de composants et de connecteurs. Néanmoins, les langages de description d'architecture n’offrent pas un support suffisant pour le développement complet de systèmes logiciels complexes, de la conception architecturale au code exécutable. D'autre part, certaines méthodes de développement formel permettent de raffiner une spécification d'un logiciel pour en obtenir une autre plus proche de l'implémentation, voire de générer le code, mais sans prendre en compte la description architecturale du système.

Nous proposons, dans le cadre de cette thèse, d'utiliser un mécanisme de raffinement pour transformer la description architecturale en une spécification formelle "classique", pour laquelle nous avons d'ores et déjà des outils permettant d'achever le développement.

Le problème de recherche que nous traitons est le raffinement d’une description d’architecture logicielle en π-SPACE, langage de description d’architecture logicielle fondé sur une algèbre de processus, vers une spécification abstraite constituée de machines abstraites de la méthode B, qui dispose d’outils pour aider au développement formel et à la génération du code.

Nous développons un système formel – nommé β-SPACE – pour la mise en oeuvre de raffinements successifs, menant de la description architecturale de départ (en π-SPACE) à une spécification formelle (un ensemble de machines abstraites de la méthode B) telle qu’un développement formel de l’application soit possible dans le cadre de la méthode B, tout en garantissant que chaque étape de raffinement conserve les propriétés de la description architecturale initiale.

La définition formelle du raffinement d’architectures logicielles est basée sur la logique de réécriture, dans laquelle sont représentés les éléments architecturaux abstraits, mais aussi les constructions du langage cible de spécification. Cette logique dispose, elle aussi, d'un outil de développement et d'exécution qui permet d'automatiser les transformations.

Notre approche du raffinement architectural diffère des autres méthodes existantes, en ne s’intéressant pas seulement qu’à l’ajout de détails à la description formelle, mais aussi à la transformation de sa structure de contrôle : la composition de composants et de connecteurs de l’architecture est transformée pour obtenir une hiérarchie de machines abstraites B. Cela requiert de changer le mode de contrôle des actions à l’intérieur de ces différentes descriptions d'un même système : il faut passer de comportements concurrents de composants et connecteurs, synchronisés par des communications, à des machines abstraites organisées hiérarchiquement, liées par des appels d’opérations. Cependant, nous assurons que les propriétés architecturales qui nous intéressaient sont bien conservées.

C’est une approche originale à la fois sur sa portée architecturale (structure mais aussi comportement), comme sur sa formalisation et son articulation avec les méthodes formelles classiques.

Mots clés : Génie Logiciel, Raffinement Formel, Architecture Logicielle, Méthode Formelle.