Université de Montréal

MODELES ET ALGORITHMES POUR LA SIMULATION DES SYSTEMES A TEMPS REEL

par

Ana-Francisca Nicolae

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de

Maître ès Sciences (M. Sc.) en informatique

juin 1999

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé

MODELES ET ALGORITHMES POUR LA SIMULATION DES SYSTEMES A TEMPS REEL

présenté par

Ana-Francisca Nicolae

a été évalué par un jury composé des personnes suivantes:

[redacted]

(président-rapporteur)

[redacted]

(directeur de recherche)

[redacted] ...eur Marc Feeley

(membre du jury)

Mémoire accepté le : 99-08-01

Sommaire

Traditionnellement, les méthodes de spécification et de vérification des interfaces matérielles sont utilisées avec prépondérance dans le domaine de la conception et du développement des systèmes numériques à temps réel. Ces systèmes et leurs interfaces sont généralement spécifiés à l'aide des chronogrammes (appelés aussi "diagrammes temporels" ou "diagrammes d'actions").

Les diagrammes d'actions sont des outils de spécification permettant la description précise et adéquate de l'aspect temporel des interfaces matérielles, de l'évolution des événements (ou actions) sur les ports de l'interface. Les ports constituent les signaux physiques transportant les informations communiquées entre le système et son environnement.

La vérification des interfaces matérielles est nécessaire afin de déterminer si les composants interconnectés peuvent fonctionner ensemble et si les exigences temporelles sont satisfaites. Le résultat de notre recherche sont les algorithmes d'interprétation et d'implantation des diagrammes d'actions pour la vérification des interfaces matérielles dans le cadre de la simulation VHDL. Ces algorithmes sont gouvernés par l'algèbre de processus ACTC^P ("Algebra of Communicating Timing Charts - Port Oriented"), créée dans notre laboratoire de recherche. L'utilisation d'une algèbre de processus ayant une sémantique formelle facilite une implantation correcte des algorithmes de simulation. Aussi, le logiciel devient plus facilement adaptable aux besoins spécifiques des usagers et aux changements dans la définition de l'algèbre ACTC^P.

L'interface peut être modélisée de manière modulaire. Les diagrammes d'actions de base, appelés "diagrammes d'actions de type feuille", sont composés à l'aide d'opérateurs hiérarchiques (obtenant ainsi les "diagrammes d'actions hiérarchiques"). La sémantique opérationnelle des opérateurs de composition est exprimée par les règles de transition de l'algèbre ACTC^P. Les diagrammes d'actions peuvent être annotés de prédicats et de procédures afin de spécifier des comportements plus complexes. Cependant, la sémantique de ces annotations n'est pas incluse dans ACTC^P pour le moment. Les diagrammes d'actions sont exprimés à l'aide du langage de spécification HAAD de syntaxe LISP et sont utilisés comme entrées pour notre logiciel. Ensuite, un modèle VHDL est généré à partir de la spé-

cification en HAAD. Ce modèle décrit la fonctionnalité du système matériel et l'interaction avec son environnement. L'exécution de ce modèle dans le cadre d'une simulation VHDL est utilisée pour décider si le modèle est conforme aux spécifications.

La contribution originale de ce travail de recherche est représentée par:

- La participation à la définition des opérateurs hiérarchiques de composition des diagrammes d'actions (composition séquentielle ou concaténation, composition parallèle avec rendez-vous, le choix retardé, l'exception, la récursion),
- La conception des algorithmes qui interprètent la sémantique formelle de l'algèbre $ACTC^P$ et la présentation de l'équivalence entre la sémantique formelle et l'exécution du modèle VHDL,
- L'implantation de l'opérateur d'exception et la généralisation de l'opérateur de récursion,
- L'illustration de la méthode utilisée sur un exemple pratique.

Mots clés: simulation, interfaces, vérification, temps-réel, chronogrammes, algèbre de processus.

Remerciements

L'auteur remercie son directeur Eduard Cerny pour le soutien accordé, sa supervision et ses conseils pendant toute la période de recherche de ce travail.
L'auteur désire aussi à remercier Karim Khordoc pour l'acheminement dans la phase initiale de la recherche.
Merci à Andrei Tarnauceanu pour tout son aide et sa disponibilité.

Table des matières

Liste des figures

Liste des sigles et des abbréviations

$a@P(t) \in Act$	Action réelle a exécutée sur le port P au moment t	19
ACP	Algebra of Communicating Processes	19
ACTC	Algebra of Communicating Timing Charts	6
$ACTC^P$	Algebra of Communicating Timing Charts - Port Oriented	1
Activated	Primitive de l'algèbre $ACTC^P$	48
ATP	Algebra of Timed Processes	6
$a@P(t)$	Action réelle a exécutée sur le port P au moment t	19
begin $\beta \in \mathcal{B}$	Action spéciale	52
BDD	Binary Decision Diagrams	5
Concat	Opérateur de concaténation	12
$\mathbf{Concat_i}$	Règle de transition pour Concat	22
CONCAT_AD	Etiquette du type had_node utilisée pour une concaténation	34
CONDITION_AD	Etiquette du type had_node utilisée pour une condition d'exception	34
CONCUR_AD	Etiquette du type had_node utilisée pour une composition parallèle avec rendez-vous	34
$\mathbf{DC_i}$	Règle de transition pour DChoice	25
DChoice	Opérateur de choix retardé	14
D_CHOICE_AD	Etiquette du type had_node utilisée pour un choix retardé	34
Empty (*P, Terme*)	Primitive de l'algèbre $ACTC^P$	48
end $\varepsilon \in \mathcal{E}$	Action spéciale	53
Excep	Opérateur de l'exception	14
$\mathbf{Excep_i}$	Règle de transition pour Excep	28
EXCEP_AD	Etiquette du type had_node utilisée pour une exception	34
HAAD	Hierarchical Annotated Action Diagrams	16
HANDLER_AD	Etiquette du type had_node utilisée pour un comportement d'exception	34

LEAF_AD	Etiquette du type had_node utilisée pour un diagramme feuille	34
LOTOS	Language of Temporal Ordering Specifications	6
NORMAL_AD	Etiquette du type had_node utilisée pour un comportement normal	34
nothing	Primitive de l'algèbre ACTC^P	42
P_{In}	Port de direction d'entrée	19
P_{Out}	Port de direction de sortie	19
$P_C = P \mid P'$	Ports de communication	20
$\mathbf{RC}_i$	Règle de transition pour RComp$_P$	26
RComp$_P$	Opérateur de composition parallèle avec rendez-vous	14
recX	Opérateur de récursion	15
RECX	Etiquette du type had_node utilisée pour une récursion	34
Shift$_t$	Opérateur de décalage de temps	21
Shift$_t$ (*Terme*)	Primitive de l'algèbre ACTC^P	49
TCSP	Timed Communicating Sequential Processes	19
TCTL	Timed Computation Tree Logic	7
TeCCS	Temporal Calculus of Communicating Processes	6
Terme$_{cond}$	Terme de la condition d'exception	14
Terme$_{excep}$	Terme de l'exception	14
T-LOTOS	Timed Language of Temporal Ordering Specifications	7
VHDL	VHSIC Hardware Description Language	2
VHSIC	Very High Speed Integrated Circuit	3
Wait$_t$ (*Terme*)	Primitive de l'algèbre ACTC^P	49

Chapitre 1. Introduction

L'importance de la recherche dans le domaine de la vérification des spécifications et des implantations des interfaces des systèmes matériels ne cesse de s'accroître. Les systèmes matériels représentent des conglomérats de modules organisés hiérarchiquement et qui communiquent entre eux. Les *interfaces matérielles* définissent les protocoles de communication entre les composants du système ou entre le système et son environnement. A part l'avantage de la réutilisation des composants, la construction modulaire et hiérarchique des systèmes facilite la spécification, la conception et la vérification des systèmes et de leur composants. Etant donné que les modules des systèmes complexes sont conçus de manière quasi-indépendante les uns des autres du point de vue de leur fonctionnement, il est nécessaire de spécifier et définir le comportement des modules sur les ports de communication de leurs interfaces en fonction des diverses hypothèses sur leur environnement.

Les interfaces matérielles des modules ou des systèmes spécifient les suppositions sur le comportement de leur environnement et les réponses du système suite à des stimuli venus de l'extérieur. Souvent les spécifications initiales des systèmes et de leur interfaces manquent de rigueur, étant donné que lors des phases initiales de la conception des systèmes une quantité importante d'information est décrite de manière informelle (en langage naturel), complétant ainsi les spécifications plus formelles: *les chronogrammes* (ou *diagrammes temporels* ou *diagramme d'actions*).

Les interfaces complexes sont modélisées hiérarchiquement en composant des diagrammes d'actions du type de base "feuille" ou des diagrammes hiérarchiques à l'aide de divers opérateurs de composition tels que la composition séquentielle (ou la concaténation), la composition parallèle avec ou sans communication, le choix retardé, l'exception ou la récursion. La sémantique opérationnelle des opérateurs hiérarchiques est décrite par des ensembles de règles de transitions dans le cadre de l'algèbre $ACTC^P$("Algebra of Communicating Timing Charts - Port Oriented", voir [1]) créée dans notre laboratoire de recherche dans le but de définir formellement la sémantique opérationnelle des diagrammes d'actions.

Les diagrammes d'actions de type "feuille" décrivent les opérations de base de l'interface.

Un diagramme d'actions "feuille" est défini sur un ensemble de *ports* (signaux physiques transportant l'information nécessaire à la communication par l'intermédiaire des actions sur le port), un ensemble d'*actions* (des changements de valeur sur les ports) et un ensemble de *contraintes temporelles* (relations temporelles entre les moments d'occurrence des actions). Les *annotations* des diagrammes d'actions (des procédures et des prédicats) contiennent l'aspect fonctionnel de l'interface.

La vérification des interfaces comporte deux aspects, d'égale importance:

- L'aspect fonctionnel (le respect du protocole de communication et des valeurs échangées entre les modules ou entre le système et son environnement),
- L'aspect temporel (décider si au niveau de l'interface les relations temporelles entre les actions observées ou produites par le système sont satisfaites). L'aspect temporel est représenté sous forme de diagramme temporel.

Les méthodes de spécification et de vérification des interfaces matérielles sont utilisées dans le cas des systèmes numériques où le temps est une coordonnée importante (mémoires, processeurs, bus, etc.).

Le résultat de notre recherche est un ensemble d'algorithmes d'interprétation et implantation des diagrammes d'actions permettant l'exécution des spécifications des interfaces matérielles dans un cadre de simulation VHDL. Ces algorithmes suivent le formalisme mathématique de l'algèbre de processus $ACTC^P$ pour la définition formelle de la sémantique opérationnelle des diagrammes d'actions. Notre travail représente une nouvelle méthode de spécification et de vérification des interfaces matérielles. Une autre méthode, moins formelle toutefois, est implantée dans le logiciel [2], conçu et développé dans notre laboratoire. L'approche générale de modélisation des interfaces matérielles utilisée par les deux méthodes est celle décrite dans [3]. La nouvelle méthode est plus flexible quant à la modélisation des systèmes à *pipelines*. L'instant d'occurrence de la première action sur un port du diagramme d'actions peut être spécifié indépendamment de l'instant d'occurrence de la première action sur tout autre port du même diagramme. La même situation existe dans le cas de l'instant d'occurrence des dernières actions sur les ports d'un diagramme d'actions.

Un modèle exécutable est généré automatiquement en langage VHDL par notre logiciel à partir d'une spécification d'interface. Le langage VHDL (VHSIC Hardware Description

Language, où VHSIC signifie "Very High Speed Integrated Circuit", voir [4]) est un langage utilisé pour la conception, la modélisation et la simulation des systèmes numériques. La versatilité de ce langage permet la création de modèles VHDL à plusieurs niveaux d'abstraction. Ces modèles consistent un ensemble de processus concurrents qui peuvent communiquer entre eux par des signaux.

La structure générale d'un modèle VHDL (voir la figure 1) est composée de:

- une entité (*entity*) représentant l'interface d'un composant (la seule partie du modèle VHDL à laquelle l'usager a l'accès): elle contient les ports de l'entité, où les actions réelles sont observées. Lors de l'initialisation, l'usager doit fournir la valeur initiale des ports de l'entité.
- une ou plusieurs *architectures* décrivant la fonctionnalité du composant et consistant en un ou plusieurs *processus* communicant par l'intermède des signaux. Chaque processus est un programme séquentiel qui participe à la description du comportement du composant. Un processus peut être sensible à une liste de signaux (appelée *liste de sensibilité*). Aussitôt que l'un de ces signaux est modifié, le processus contenant ce signal dans sa liste de sensibilité est réveillé (énoncé "WAIT ON" en VHDL), certaines opérations sont exécutées et il est ensuite suspendu à nouveau. Un processus peut être réveillé aussi par l'écoulement d'un délai fixé (énoncé "WAIT FOR"). Dans un cycle de simulation, tout processus qui est réveillé est executé jusqu'à la rencontre d'un nouveau énoncé "WAIT".
- un ou plusieurs *packages* contenant des types de données, procédures et composants utilisés à l'intérieur de l'entité et des architectures.

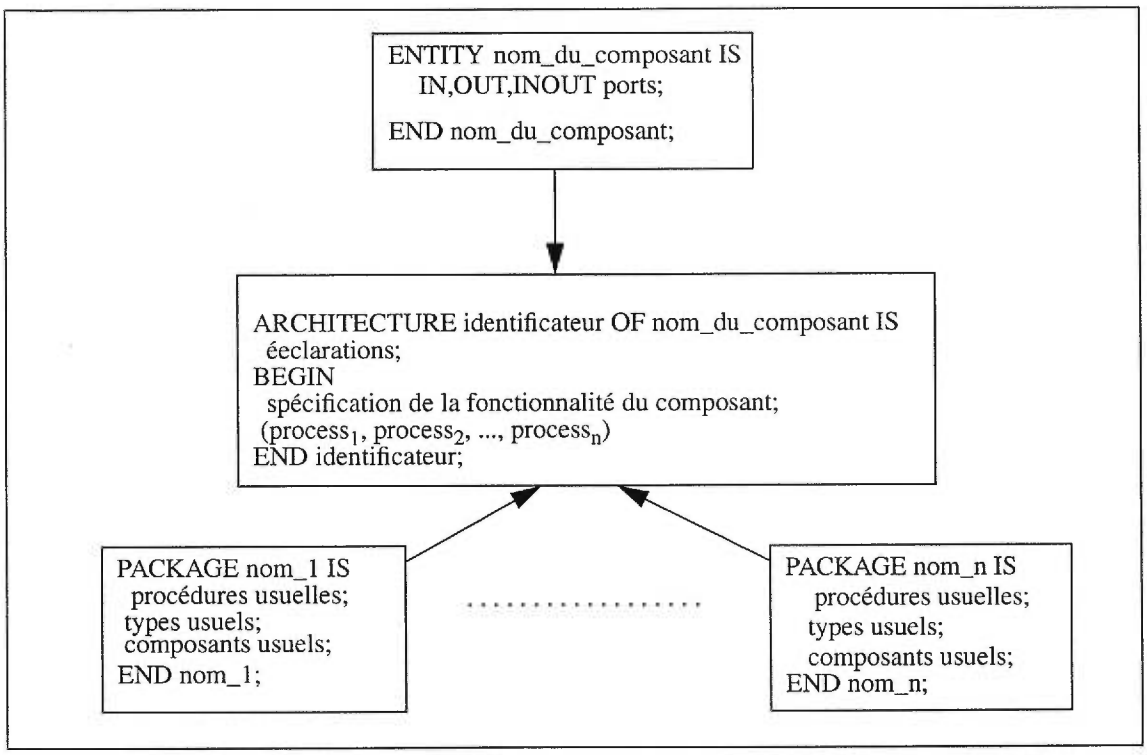


FIGURE 1: La structure générale d'un modèle VHDL

Le logiciel utilise en entrée la spécification de l'interface décrite en langage HAAD, dont la grammaire est contenue en l'annexe A, et génère le modèle VHDL en utilisant un générateur de code (voir [2] et [5]). L'exécution du modèle VHDL peut être effectuée avec des systèmes de simulation comme VANTAGE et SYNOPSYS (voir [6, 7]). Le modèle exécutable correspond au diagramme hiérarchique d'actions de plus haut niveau décrivant l'interface. L'exécution du modèle valide (ou invalide, selon le cas) les moments d'occurrence des actions produites par l'environnement et observées par le système et génère les réponses du système, en concordance avec la spécification de l'interface.

L'apport original de ce travail consiste en:

- La définition des opérateurs hiérarchiques de composition des diagrammes d'actions (composition séquentielle ou concaténation, composition parallèle avec rendez-vous, le choix retardé, l'exception, la récursion),

- La conception des algorithmes de simulation décrivant la sémantique opérationnelle de tous les opérateurs hiérarchiques mentionnés, ainsi que la correspondance entre les primitives de l'algèbre ACTC^P (les paquets d'informations à la base de la construction des règles de transitions de la sémantique opérationnelle), les règles de transitions et leur implantation,
- L'implantation de l'opérateur d'exception et la généralisation de l'opérateur de récursion (afin de modéliser des structures complexes),
- Un exemple pratique montrant l'utilisation de notre logiciel.

1.1 Revue de la littérature

De nombreux travaux traitent de la vérification d'interfaces des systèmes matériels en général et, en particulier, des systèmes à temps réel, dont la fiabilité du point de vue temporel est une condition sine qua non. Les chercheurs essayent continuellement d'améliorer le développement des applications en introduisant des méthodes de plus en plus rigoureuses afin d'accroître la fiabilité des systèmes.

Pour la spécification et la vérification des systèmes matériels, différentes méthodes ont été adoptées. La simulation est la méthode de vérification la plus utilisée dans la industrie et qui consiste en l'étude des réponses du système face aux stimuli auxquels il est soumis. La simulation, qui est une vérification partielle sur un modèle complet (une énumération exhaustive des stimuli serait trop complexe), peut être utilisée conjointement avec d'autres techniques:

- La vérification de modèle, "model-checking" [8], est une vérification complète sur un modèle incomplet, car elle énumère tous les états possibles de la machine à états associée au système.
- La vérification symbolique de modèle [9, 10, 11], où des formules booléennes représentent les états du modèle afin d'atténuer le problème d'explosion de l'espace d'états, dans le cas de la vérification de modèle. Les BDD ("Binary Decision Diagrams") sont utilisés pour l'évaluation des expressions booléennes. Les techniques de vérification de

modèle sont facilement automatisables et utilisables par les développeurs de systèmes.

- Les méthodes basés sur les preuves ("theorem provers") utilisent les logiques temporelles ou les algèbres de processus. Dans le cas des logiques temporelles (voir [12, 32]), des raisonnements logiques sont appliqués pour démontrer qu'une spécification satisfait une propriété donnée. Les algèbres de processus écrivent toutes les exécutions possibles d'un terme du langage de spécification de l'algèbre. Parmi ce type d'algèbres nous mentionnons ATP [13, 14], TeCCS [15], ACTC [16], ACTC^P [1], les extensions temporisées de Lotos [17, 18]. Une synthèse complète des algèbres de processus pour la spécification et l'analyse des systèmes temporisés (systèmes pour lesquels le temps est un paramètre global qui restreint l'occurrence des actions) se trouvent dans [14].

Du point de vue de la méthode de spécification, à part les notations textuelles telles que les algèbres de processus, il existe aussi des notations graphiques. Parmi les notations graphiques, nous pouvons mentionner les versions avec temps des réseaux de Petri [19, 20, 21], les "StateCharts" [22, 31] qui utilisent une combinaison entre la description formelle et informelle. Les "StateCharts" combinent la méthode des automates à états avec une vue modulaire et hiérarchique, proche de l'organisation des systèmes matériels.

La première utilisation des chronogrammes pour la spécification et la vérification des interfaces des systèmes numériques a été introduite par [23]. Les diagrammes temporels sont utilisés par [24] conjointement avec les diagrammes d'états pour la génération de modèles VHDL pour les cycles de lecture et écriture d'un processeur qui décide le choix du cycle.

Les auteurs de [25] proposent une méthode de spécification et d'analyse des systèmes à temps réel en utilisant un langage de spécification pour la description des systèmes de processus communicants. Les spécifications sont ensuite transformées dans des automates temporisés avec une exécution à activation contrôlée par les événements ("event-driven execution"). Les compteurs de temps sont des variables d'états remis à zéro suite à des transitions et dont la valeur représente le temps écoulé depuis la dernière remise à zéro. Les conditions associées aux transitions sont les contraintes temporelles (une transition est exécutée si la condition associée est vraie). A partir des automates temporisés, du code exécutable est généré. Les propriétés du système, exprimées dans la logique temporelle TCTL [26] peuvent être vérifiées par les méthodes usuelles: vérification de modèle, vérifi-

cation symbolique de modèle.

Les chercheurs participant au projet FORMAT [18] décrivent une approche de spécification et d'analyse des systèmes matériels utilisant un langage graphique de spécification conjointement avec le langage VHDL. Les composants du système sont modélisés avec des diagrammes temporels annotés et avec une variante de "StateCharts" hiérarchiques. Les spécifications de type chronogrammes sont transformées en processus du langage de spécification T-LOTOS (J. Quemada et A. Fernandez, 1987) et ensuite en code VHDL. Même s'ils affirment que leur méthode est très puissante et qu'elle a été utilisée dans l'industrie, les auteurs avouent qu'un effort considérable de la part des usagers est nécessaire pour maîtriser leur langage de spécification. La vérification symbolique de modèle et la méthode basée sur les preuves sont utilisées dans le projet FORMAT.

1.2 Plan du mémoire

Le mémoire est organisé de la manière suivante:

- Le chapitre 2 présente les notions de diagrammes d'actions feuilles et hiérarchiques qui modélisent le fonctionnement et l'aspect temporel des interfaces des systèmes numériques et le langage HAAD que nous utilisons pour la spécification des interfaces matérielles.
- Le chapitre 3 expose l'algèbre de processus $ACTC^P$ sur laquelle se base notre système de simulation pour la vérification des interfaces. Ce chapitre présente toutes les transitions valables entre deux états du système (c.à.d., de l'interface) et les opérations de base (que nous appelons "primitives") utilisées pour la construction des règles de transition. Nous présentons aussi la classification des règles de transition selon les actions exécutées par l'interface et selon l'influence de l'occurrence des actions sur la hiérarchie des diagrammes d'actions.
- Le chapitre 4 présente la représentation et le principe de simulation des diagrammes d'actions que notre logiciel utilise. Le chapitre suit par les algorithmes de simulation utilisés et la correspondance que nous préservons entre la sémantique opérationnelle de

utilisés et la correspondance que nous préservons entre la sémantique opérationnelle de l'algèbre ACTC^P et l'implantation.

- Le chapitre 5 contient un exemple pratique d'utilisation et du fonctionnement du logiciel.
- Le chapitre 6 présente les conclusions de notre travail.

Chapitre 2. Les diagrammes d'actions

Un chronogramme de type "feuille" (que nous appelons *diagramme feuille*) est une spécification des interfaces des modèles matériels représentée par un ensemble de ports et un ensemble ordonné d'actions (ou événements) qui se produisent sur les ports. L'ordonnancement des actions se réalise par des relations temporelles (contraintes temporelles) entre les actions spécifiées. Les *diagrammes hiérarchiques* décrivent des comportements d'interfaces plus complexes: ils sont représentés par divers opérateurs hiérarchiques gouvernés par des règles de la sémantique opérationnelle (énumérées dans le chapitre 3). Le chapitre continue par une description des diagrammes d'actions feuilles.

2.1 Les diagrammes feuilles

Un diagramme feuille est défini sur un ensemble de ports et d'actions. Graphiquement, les ports (voir la figure 2) sont disposés le long de l'axe vertical, tandis que les actions sont disposées le long de l'axe horizontal.

Les **ports** sont les ressources nécessaires pour la communication entre le modèle du matériel et son environnement. La direction des ports peut avoir une des valeurs suivantes:

- d'entrée ou "in" pour les ports transportant des actions produites par l'environnement et destinées au système
- de sortie ou "out" pour les ports contenant des actions produites par le système et envoyées à l'environnement
- bidirectionnelle ou "inout" (voir les ports P_1 et P_2 dans la figure 2) pour les ports qui peuvent transporter des actions d'entrée ou de sortie (mais une seule action à la fois).

Les **actions** ont la même direction que les ports auxquels elles appartiennent, sauf dans les cas suivants:

1. Sur les ports bidirectionnels, la direction de chaque action doit être spécifiée.
2. Sur chaque port il existe deux actions spéciales lui délimitant la portée temporelle:

l'action spéciale *begin* (de type *start_action* et de direction d'entrée), et l'action spéciale *end* (de type *end_action* et de direction de sortie). Par défaut, le moment d'occurrence de *end* coïncide avec celui de la dernière action réelle sur le port (e_1 dans la figure 2). Nous soulignons que les actions spéciales *begin* et *end* permettent la composition hiérarchique.

L'usager a la possibilité de spécifier l'intervalle d'occurrence de la première action réelle sur le port (a1 dans la figure 2), qui peut être retardée par un délai quelconque par rapport à l'action spéciale *begin*. De manière similaire, il est possible de retarder l'action spéciale *end* par rapport à la dernière action réelle sur le port (e_2 dans la figure 2).

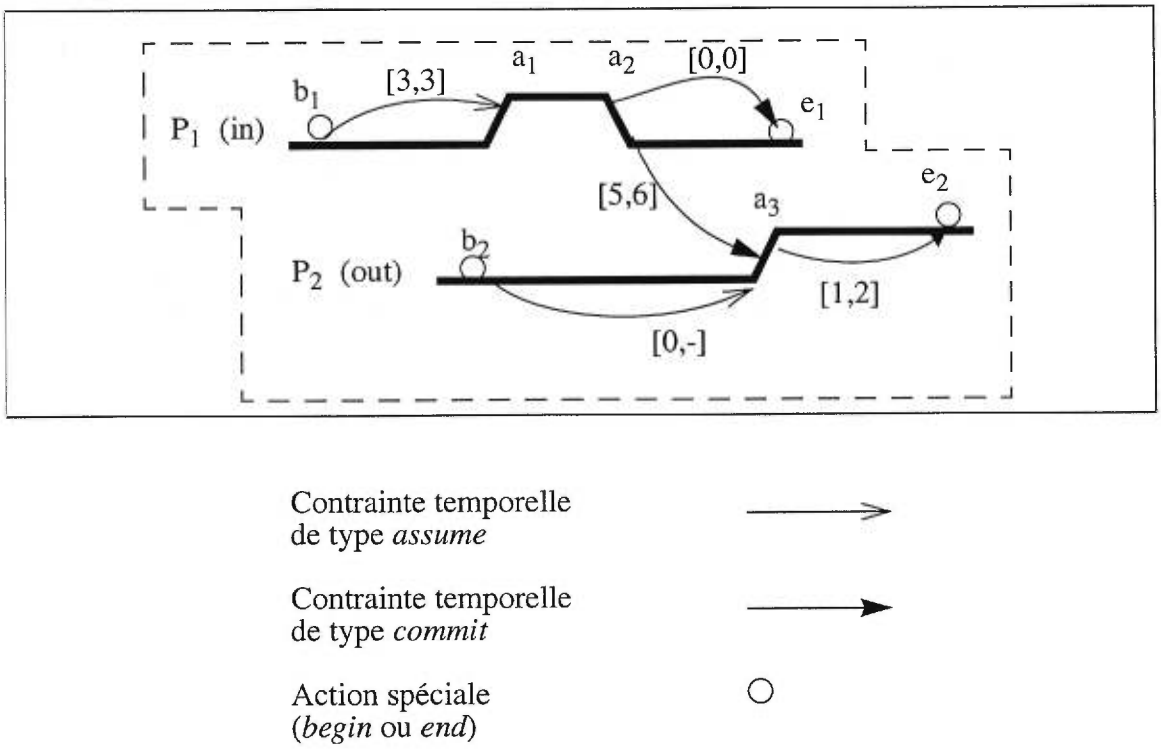


FIGURE 2: Exemple d'un diagramme feuille

Les actions virtuelles START et END délimitent l'intervalle de temps dans lequel un diagramme est *actif*. Elles sont associées par défaut à la première action spécifiée dans un diagramme (le premier *begin* qui a lieu sur un des ports du diagramme), respectivement la dernière action (le dernier *end* qui a lieu sur un port de l'ensemble des ports du diagramme).

Les relations temporelles entre les actions sont exprimées par des **contraintes temporelles**, spécifiées graphiquement par des flèches entre l'action source et l'action puits, auxquelles des intervalles temporels d'occurrence [borne_inférieure, borne_supérieure] sont associés. L'action puits se produit dans l'intervalle temporel [borne_inférieure, borne_supérieure] après l'instant d'occurrence de l'action source. Les intervalles peuvent être fermés ou ouverts. Les contraintes expriment une *précédence* entre les actions si la borne inférieure de l'intervalle d'occurrence est un nombre réel strictement positif ou une *concurrence*, si la borne inférieure est négative ou égale à zéro.

Les contraintes temporelles sont soit du type *commit* ou du type *assume*. Les contraintes *assume* expriment une supposition sur le moment d'occurrence d'une action puits de direction d'entrée. Les contraintes *commit* représentent l'engagement du modèle à générer l'action puits dans l'intervalle spécifié, après le moment d'occurrence de l'action source). Par exemple, sur un port, entre *begin* et la première action réelle il y a, par défaut, une contrainte temporelle de type *assume* [0,-], où "-" est notre notation pour l'infini (voir b_2 dans la figure 2). Pour une contrainte de type *commit*, les bornes de son intervalles doivent avoir des valeurs finies.

L'ordre de spécification des actions sur un port indique la succession des occurrences des actions pendant l'exécution du modèle. Les contraintes du même type et ayant une action puits commune sont composées à l'aide des opérateurs *conjunctive* (c.à.d., toutes les actions source influencent l'instant d'occurrence de l'action puits), *latest* (l'intervalle temporel d'occurrence de l'action puits est retardé au maximum par rapport aux actions sources) ou *earliest* (l'intervalle d'occurrence de l'action puits est devancé le plus possible par rapport aux actions sources).

Les diagrammes feuilles peuvent contenir des **annotations**:

- Des *paramètres*: si un paramètre est associé à une action d'entrée, la valeur du port est affectée au paramètre. Par contre, si un paramètre est associé à une action de sortie, la valeur du paramètre est affectée au port. La direction des paramètres est d'entrée, de sortie ou bidirectionnelle, selon l'association avec une action.
- Des *variables* d'état de tout type compatible avec VHDL: elles sont déclarées dans un diagramme et sont utilisées de la même manière que dans un programme classique.
- Des *procédures* ou des *prédicats*: ils décrivent le fonctionnement interne du système

matériel dont l'interface est modélisée et peuvent modifier la valeur des paramètres de sortie ou bidirectionnels.

2.2 Les diagrammes hiérarchiques

Pour décrire des comportements complexes infinis des interfaces, les diagrammes feuilles ne suffisent pas. Il est donc nécessaire d'utiliser des opérateurs de composition des diagrammes. Les opérateurs hiérarchiques suivent les règles de transition qui constituent la sémantique opérationnelle de l'algèbre de processus $ACTC^P$.

1. La concaténation

L'opérateur de concaténation Concat ($Terme_1$, $Terme_2$) possède un *ordre* intrinsèque: $Terme_2$ suit $Terme_1$ et l'enchaînement s'effectue *port par port*. Dans cette perspective, il n'est plus nécessaire, tel qu'il était le cas de la méthode utilisée dans [2], d'attendre la terminaison correcte de l'exécution de *tous* les ports du premier terme de la concaténation pour démarrer l'exécution du deuxième terme. Les ressources, dans notre cas les ports, aussitôt utilisées, sont relâchées pour qu'elles puissent être utilisées par d'autres consommateurs (l'aspect séquentiel de la concaténation). Donc, il est possible que, pour certaines périodes de temps, les deux (ou plusieurs) termes de la concaténation soient actifs en même temps, sur des ports distincts (c.à.d., en utilisant des ressources différentes).
La concaténation Concat ($Terme_1$, $Terme_2$) commence quand le premier terme (c.à. d., $Terme_1$) effectue la première action *begin* sur l'ensemble de ses ports et finit quand le dernier port P, sur l'ensemble des ports des deux termes, effectue son action *end*. Il revient à l'usager de spécifier correctement l'ordre des termes: $Terme_1$ est le premier terme où une action *begin* a lieu sur un de ses ports.
La figure 3 montre un exemple d'une concaténation entre deux diagrammes feuilles ($Terme_1$ et $Terme_2$), tous les deux définis sur les ports P_1 et P_2.

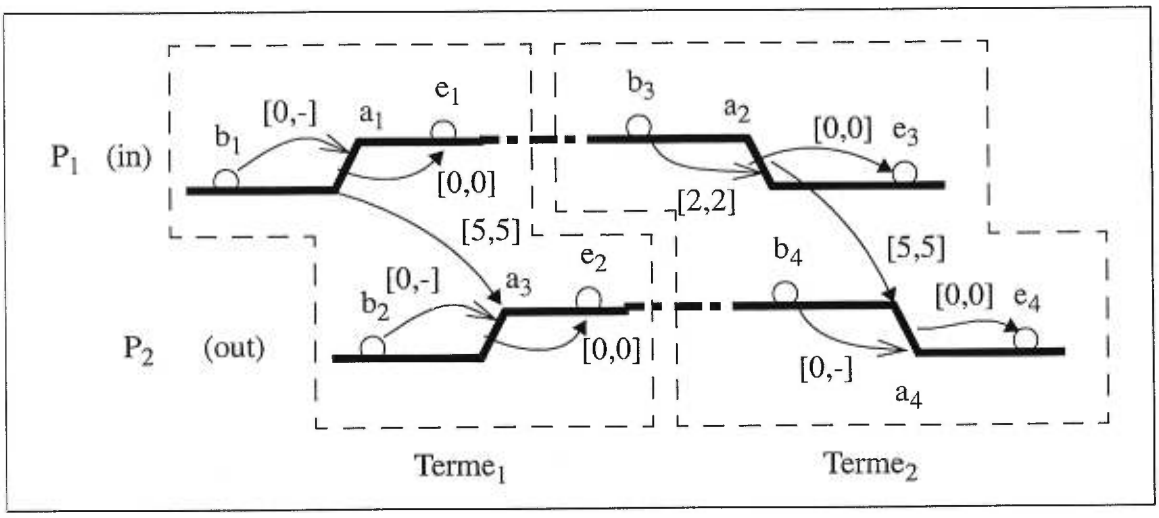


FIGURE 3: La concaténation Concat ($Terme_1$, $Terme_2$)

Dans la figure 3, après l'occurrence de l'action *end* e_1, dans $Terme_1$, le port P_1 devient disponible pour $Terme_2$ et alors pour 3 unités de temps (2 unités de temps après l'occurrence de e_1 dans $Terme_1$, jusqu'à l'occurrence de e_2 dans $Terme_1$ et celle de b_4 dans $Terme_2$, implicitement), les deux termes $Terme_1$ et $Terme_2$ sont actifs en même temps (voir aussi l'action a_3 produite 5 unités de temps après a_1).

Pour les ports appartenant à un seul terme de la concaténation, leur utilisation est librement permise: ils sont disponibles dès le début de l'exécution du diagramme hiérarchique (l'aspect "parallèle" de la concaténation).

Il est très important à noter que dans le cas d'une concaténation l'action *end* sur un port P donné dans le premier terme de la concaténation et l'action *begin* sur le même port dans le deuxième terme (si le port P existe dans le deuxième terme) ont lieu au *même instant* (ils ont le même temps d'occurrence). Dans la figure 3, les actions e_1 et b_3 ont lieu simultanément; de même pour les actions e_2 et b_4.

2. La composition parallèle avec rendez-vous

L'opérateur de composition parallèle utilisé dans $ACTC^P$ nécessite la synchronisation sur les ports appartenant à l'ensemble des ports communicants. Dans la simulation, si la synchronisation sur toute action (incluant les actions spéciales *begin* et *end*) sur les ports communicants n'est pas respectée, une erreur est signalée. La composition avec rendez-vous $RComp_P$ ($Terme_1$, $Terme_2$) commence (finit, respectivement) quand la première (la dernière) action *begin* (*end*) a lieu sur un port de l'ensemble des ports des deux termes.

3. Le choix retardé

Le choix retardé DChoice ($Terme_1$, $Terme_2$) commence (finit, respectivement) quand le premier (le dernier) port des deux termes effectue son action *begin* (*end*). Dans le cadre de cet opérateur, si un terme de la composition hiérarchique effectue une action que l'autre terme ne peut pas exécuter, le dernier est abandonné et le choix est ainsi décidé. Un port P ne devient pas disponible à l'extérieur du diagramme hiérarchique DChoice tant que tous les termes de la composition n'ont pas fini leur exécution sur P (l'action *end* doit avoir lieu sur P dans tous les termes du diagramme hiérarchique DChoice).

4. L'exception

Etant donné que l'algèbre $ACTC^P$ est orientée vers les ressources (ports) et que notre but est d'avoir une grande liberté d'utilisation des ports, nous avons été confrontés avec un problème dans le cadre de l'opérateur de l'exception Excep (Terme, $Terme_{cond}$, $Terme_{excep}$). Les ports communs aux comportement normal et à la condition d'exception qui ne sont pas utilisés par un diagramme ayant un ancêtre de type "concaténation" commun avec le diagramme de l'exception peuvent être activés lors de l'initialisation du modèle, que nous verrons plus en détail dans la section 4.4 et leurs actions *begin* ont lieu au même instant. Les ports appartenant à la condition d'exception et non au comportement normal, ne seront activés que lorsque le diagramme du comportement normal est activé (c.à.d., après l'occurrence d'un premier *begin* sur un port quelconque).

L'exception Excep (Terme, $Terme_{cond}$, $Terme_{excep}$) débute quand le comportement normal Terme et la condition d'exception $Terme_{cond}$ commencent. Si le comportement normal Terme finit, l'exception termine aussi. Si la condition d'exception $Terme_{cond}$ finit, alors le comportement normal Terme est arrêté et le comportement d'exception $Terme_{excep}$ commence. Ensuite, l'exception finit quand $Terme_{excep}$ finit.

Après la désactivation d'un port dans le terme du comportement normal, si le port existe dans le comportement d'exception, selon la règle qui gouverne généralement l'algèbre $ACTC^P$, il pourrait être rendu disponible au comportement d'exception. Compte tenu du fait que la condition d'exception n'a pas fini son exécution, l'activation du port mentionné dans le comportement d'exception ne devrait pas s'effectuer. Alors, étant donné que pour décider de l'activation des ports dans Terme ou dans $Terme_{cond}$ il est nécessaire d'attendre la terminaison d'un de ces deux termes, dans le cas de l'opérateur de l'exception nous imposons à l'usager les règles syntaxiques suivantes qui nous assurent de la disparition de ce problème:

- Le comportement normal devrait être une boucle d'itération infinie.
- L'ensemble des ports du comportement d'exception devrait être un sous-ensemble de celui du comportement normal (tant que la condition d'exception n'a pas fini son exécution, les actions réelles qui apparaissent sur les ports communs au comportement normal et au comportement d'exception sont jumelées aux actions spécifiées sur les ports du comportement normal). Ainsi nous évitons d'activer des ports dans le comportement d'exception au moment de l'activation du diagramme Excep.

Dans ce contexte, l'opérateur de l'exception est moins souple du point de vue de l'utilisation des ressources que les autres opérateurs.

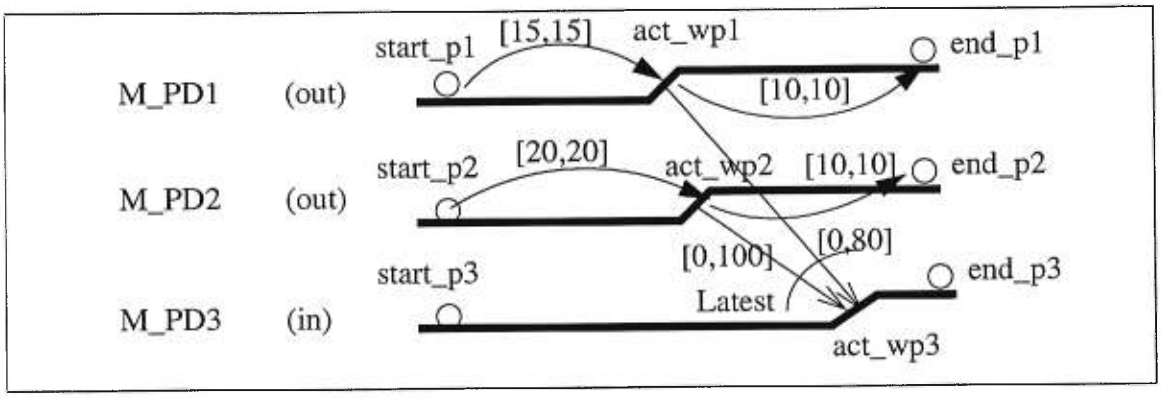
5. La récursion

L'opérateur recX permet, à part d'autres comportements généraux, la modélisation des boucles d'itération. L'occurrence de la première action dans le terme de la récursion entraîne l'ajout d'une nouvelle instance dans la hiérarchie du terme sur lequel la récursion est définie.

Nous reprendrons la présentation des diagrammes d'actions feuilles et hiérarchiques à la section 4.1. La section suivante présente notre manière de spécifier les modèles des interfaces, en utilisant le langage appelé HAAD qui permet de décrire la structure d'une hiérarchie de chronogrammes.

2.3 Le langage HAAD de spécification de l'interface du matériel

La spécification de l'interface du matériel est exprimée à l'aide du langage HAAD ("Hierarchical Annotated Action Diagrams") dont la grammaire se trouve dans l'annexe A.
Sa syntaxe, similaire au langage LISP, permet la description des deux aspects, temporel et fonctionnel, du comportement des systèmes vus de leur interfaces. La spécification en HAAD est utilisée comme entrée par un logiciel, qui génère automatiquement des modèles VHDL exécutables à l'aide d'un générateur de code (voir [5]).
Pour faciliter la compréhension de la syntaxe du langage HAAD, nous montrons sur deux exemples la spécification des comportements: le cas d'un diagramme feuille (voir la figure 4) et celui d'un diagramme hiérarchique (à la figure 5).
Explication de la syntaxe HAAD utilisée dans l'exemple de la figure 4:
Le mot-clé DEFBEHAVIOR indique que le comportement dénommé Write suit. L'interface dont le comportement est décrit possède trois ports: M_PD1 et M_PD2, de direction de sortie (OUT), et M_PD3 de direction d'entrée (IN). Le comportement est décrit par un diagramme feuille (LEAF). A l'intérieur de ce diagramme, les ports sont spécifiés par le mot clé CARRIER-SPEC. Pour chaque port (CARRIER-SPEC) la valeur initiale est indiquée (INITIAL-SPEC) ainsi que les actions spécifiées sur chacun des ports (ACTION-START pour les actions *begin*, ACTION-SPEC, et ACTION-END pour les actions *end*).
La deuxième partie de la spécification du diagramme consiste en une énumération de contraintes temporelles (PRECEDENCE) de type ASSUME ou COMMIT, pour chacune indiquant l'action source et l'action destination, les bornes inférieure et supérieure. Les deux contraintes de type ASSUME qui ont la même destination (l'action act_wp3) sont composées à l'aide de l'opérateur LATEST.




```
(DEFBEHAVIOR Write
  (PORTS
    (PORT M_PD1 OUT "std_logic" EVENT)
    (PORT M_PD2 OUT "std_logic" EVENT)
    (PORT M_PD3 IN “std_logic” EVENT))
  (LEAF
    (CARRIER-SPEC M_PD1
      (INITIAL-SPEC (CONSTANT '"0"))
      (ACTION-START 'start_p1)
      (ACTION-SPEC 'act_wp1 (CONSTANT '"1") (DIRECTION OUT))
      (ACTION-END 'end_p1))
    (CARRIER-SPEC M_PD2
      (INITIAL-SPEC (CONSTANT '"0"))
      (ACTION-START 'start_p2)
      (ACTION-SPEC 'act_wp2 (CONSTANT '"1") (DIRECTION OUT))
      (ACTION-END 'end_p2))
    (CARRIER-SPEC M_PD3
      (INITIAL-SPEC (CONSTANT '"0"))
      (ACTION-START 'start_p3)
      (ACTION-SPEC 'act_wp3 (CONSTANT '"1") (DIRECTION IN))
      (ACTION-END 'end_p3))

  (PRECEDENCE 'start_p1 'act_wp1 (CMIN 15) (CMAX 15) (INTENT COMMIT))
  (PRECEDENCE 'start_p2 'act_wp2 (CMIN 20) (CMAX 20) (INTENT COMMIT))
  (PRECEDENCE 'act_wp2 'end_p2 (CMIN 10) (CMAX 10) (INTENT COMMIT))
  (PRECEDENCE 'act_wp1 'end_p1 (CMIN 10) (CMAX 10) (INTENT COMMIT))
  (LATEST
      (PRECEDENCE 'act_wp1 'act_wp3 (CMIN 0) (CMAX 80) (INTENT ASSUME))
      (PRECEDENCE 'act_wp2 'act_wp3 (CMIN 0) (CMAX 100) (INTENT ASSUME))))
)
```

FIGURE 4: Formes graphique et textuelle d'un diagramme feuille dans le langage HAAD

Explication de la syntaxe HAAD utilisée dans l'exemple de la figure 5:
Le comportement DEFBEHAVIOR dénommé Read_After_Write possède trois ports, M_PD1, M_PD2 et M_PD3, bidirectionnels, constituant l'interface du comportement. Le diagramme hiérarchique est une CONCATENATION. Le mot clé BEHAVIOR instancie un comportement ayant déjà été déclaré en utilisant le mot clé DEFBEHAVIOR (a_Write est le nom de l'instance et Write est le nom de la définition du comportement). PORT-MAP fait l'association entre les ports du diagramme Read_After_Write et les ports des diagrammes Read et Write.

```
(DEFBEHAVIOR Read_After_Write
  (PORTS
    (PORT M_PD1 INOUT "std_logic" EVENT)
    (PORT M_PD2 INOUT "std_logic" EVENT)
    (PORT M_PD3 INOUT "std_logic" EVENT))

 (CONCATENATION
     (BEHAVIOR a_Write Write
       (PORT-MAP M_PD1 M_PD2 M_PD3))
     (BEHAVIOR a_Read Read
       (PORT-MAP M_PD1 M_PD2 M_PD3))
 )
)
```

FIGURE 5: Spécification d'un diagramme hiérarchique de type CONCATENATION dans le langage HAAD

Dans le présent chapitre nous avons introduit les notions de diagrammes d'actions feuilles et hiérarchiques et le langage HAAD utilisé par le logiciel pour la spécification des interfaces.
Le chapitre suivant présente des notions générales au sujet des algèbres de processus et les particularités de l'algèbre de processus $ACTC^P$, que nous avons utilisée pour valider l'implantation. Les diagrammes d'actions sont des termes de l'algèbre qui peuvent être composés à l'aide d'opérateurs hiérarchiques. L'exécution d'une action dans un terme (diagramme) et l'influence de l'action sur ce terme et sur la hiérarchie de diagrammes sont décrites par les règles de transitions, que nous présentons dans le chapitre suivant.

Chapitre 3. L'algèbre de processus ACTC^P

3.1 La sémantique opérationnelle de l'algèbre de processus ACTC^P

Les algèbres de processus sont des formalismes de spécification bien adaptés pour la spécification et l'analyse des systèmes concurrents qui sont mis en communication. Initialement utilisées pour la description des systèmes non-temporisés (temps arbitraire), certaines des algèbres de processus ont été adaptées pour inclure le temps dans leur analyse, telles que TeCCS [15], TCSP [28], les extensions à temps réel de ACP [29,30], ou bien conçues spécialement pour les systèmes temporisés: ATP [13,14], ACTC [16], ACTC^P [1].

Les constituants de toute algèbre de processus sont:

- un langage de spécification muni d'une sémantique exacte représentée par un ensemble de règles de transition qui décrivent les exécutions possibles des termes,
- une notion d'équivalence des termes qui représentent les mots du langage: deux termes (ou processus) sont équivalents s'ils ont le même comportement.

L'algèbre de processus ACTC^P ("Algebra of Communicating Timing Charts - Port Oriented") a été créée dans le but d'appliquer la vérification formelle aux spécifications d'interfaces des systèmes matériels de type diagramme d'actions. La méthode de spécification des systèmes matériels, développée par [3,33], se base sur la description modulaire des systèmes: la spécification d'un diagramme hiérarchique est obtenue en composant, à l'aide d'opérateurs hiérarchiques, des chronogrammes de base, appelés diagrammes feuilles.

Les diagrammes feuilles, qui ont été présentés au chapitre 2, sont définis sur un ensemble de ressources (ports) par l'intermède desquels se réalise la communication entre le système et son environnement. Les ports sont soit de direction d'entrée P_{In} (portant des actions d'entrée, exécutées par l'environnement et observées par le système) ou de sortie P_{Out} (transportant des actions de sortie, effectuées par le système et observées par l'environnement). Les actions représentent les changements de valeur sur les ports et sont notées par $a@P(t)$, c.à.d., l'action a est exécutée sur le port P au moment t.

La correspondance entre les termes mis en composition hiérarchique se réalise port par port. La première (la dernière) action sur un port est l'action spéciale *begin* $\beta \in \mathcal{B}$ (*end* $\varepsilon \in \mathcal{E}$). Sur un port P, les actions réelles $a \in Act$ sont exécutées après l'action *begin* et avant l'action *end* (qui signifie la terminaison correcte de l'exécution des actions sur le port). Les actions partagées par plusieurs ports (nommés "ports de communication" ou "ports communicants" $P_C = P \mid P'$) sont les "actions de communication" notées par $a \mid b$ (a en communication avec b). Les actions qui ne sont pas observées aux niveaux supérieurs de la hiérarchie sont appelées "actions silencieuses" τ.

$Terme \xrightarrow{a\ @\ P(t)} Terme'$ dénote l'exécution d'une action $a\ @\ P(t)$ par le terme *Terme* qui ensuite évolue dans *Terme'*

$Terme \xrightarrow{a\ @\ P(t)}\!\!\!\!\!\!\not\;\;$ dénote l'impossibilité de *Terme* d'exécuter l'action $a\ @\ P(t)$

La forme générale des règles de transition est donnée dans le format suivant, défini par [35], où la partie supérieure contient les hypothèses et la partie inférieure représente la conclusion:

$$\frac{Terme_1 \xrightarrow{a\ @\ P(t)} Terme'_1, \quad Terme_1 \xrightarrow{b\ @\ P(t)} Terme'_2}{\mathbf{RComp}_P\ (Terme_1, Terme_2) \xrightarrow{a \mid b\ @\ P(t)} \mathbf{RComp}_P\ (Terme'_1, Terme'_2)}$$

La syntaxe du langage hiérarchique utilisé pour décrire les termes est capturé sous forme BNF:

Terme := *Terme_de_base* (*ou diagramme feuille*) | **Shift**$_t$(*Terme*) |
Concat (*Terme*,*Terme*) | **DChoice** (*Terme*,*Terme*) |
RComp$_P$ (*Terme*,*Terme*) | **Excep** (*Terme*,*Terme*,*Terme*) | **recX.** (*Terme*)

où:

- Le décalage du temps **Shift**$_t$(*Terme*) définit un terme qui commence l'exécution de ses actions après ou au moment *t*.
- La concaténation (**Concat**) est une composition séquentielle port par port sur les ports communs de ses termes et une composition parallèle sur les ports non-communs.
- Le choix retardé (**DChoice**): les termes ont un comportement initial commun et se distinguent plus tard, quand l'un d'entre eux exécute une action non-commune ou une action commune à un instant temporel différent. Pour une exécution correcte, le choix doit se faire avant la fin de l'exécution d'un des termes.
- La composition parallèle avec rendez-vous (**RComp**$_P$): les actions des termes sont exécutées de manière concurrente, nécessitant, pour une exécution correcte, la synchronisation sur toutes les actions spécifiées sur les ports communicants.
- L'exception (**Excep** (*Terme_CompNormal*, *Terme_CondExc*, *Terme_CompExc*)) est un terme qui se comporte comme *Terme_CompNormal*; si *Terme_CondExc*, contenant seulement d'actions d'entrée, termine avant ou en même temps que *Terme_CompNormal*, le comportement d'exception *Terme_CompExc* est démarré.
- La récursion (**recX**) est utilisée pour créer des structures arborescentes telles que les boucles d'itération.

La sémantique opérationnelle du langage exprime les transitions que les termes ont le droit d'effectuer et les effets que l'exécution d'actions a sur les termes hiérarchiques ou feuille. Les transitions d'un état à un autre sont possibles seulement par l'exécution d'actions.

Pour exprimer la sémantique opérationnelle, l'introduction des prédicats suivants a été nécessaire:

- *Empty*(*P, Terme*): *Terme* n'est pas défini sur le port *P* ou il a fini correctement l'exécution de toutes les actions sur *P*.
- *Activated*(*Terme*): ce prédicat devient vrai lorsque *Terme* exécute la première action *begin* sur l'ensemble de ses ports.
- *Wait*$_t$(*Terme*): *Terme* peut attendre jusqu'au moment *t* avant exécuter ses actions.

Pour des détails et des exemples sur l'écriture d'un chronogramme sous la forme d'un terme de l'algèbre, voir le chapitre 2 de la référence [1].

Dans la section suivante nous expliquons de manière intuitive les règles de transition pour

les opérateurs hiérarchiques de l'algèbre de processus ACTC^P.

3.2 Les règles de transition

La concaténation (Concat (*Terme*,*Terme*))

Les règles que nous expliquons se trouvent à la figure 6.

- La règle de transition **Concat**$_1$ stipule que si *Terme*$_1$ effectue l'action *a@P(t)* réelle ou l'action spéciale *begin* et devient *Terme*$_1$' et que si le deuxième terme peut attendre jusqu'au temps *t*, alors le terme hiérarchique devient **Concat** (*Terme*$_1$', **Shift**$_t$(*Terme*$_2$)), où le deuxième terme a été décalé dans le temps.
- La partie "hypothèse" de la deuxième règle de transition décrit l'exécution simultanée des actions *end* et *begin* dans les deux termes, qui deviennent *Terme*$_1$' et *Terme*$_2$' suite à l'exécution de ces actions. La partie "conclusion" de cette règle exprime l'exécution de l'action silencieuse et l'évolution de la composition hiérarchique vers **Concat** (*Terme*$_1$',*Terme*$_2$').
- La règle **Concat**$_3$ diffère par rapport à la deuxième règle par les deux faits suivants: l'action effectuée est une action *end* et le prédicat *Empty*(*P*, *Terme*$_2$) est vrai.
- Par la règle **Concat**$_4$, le deuxième terme exécute une action sur un port qui soit n'est pas défini dans *Terme*$_1$ ou il existe dans ce terme, mais il a fini correctement son exécution (le prédicat *Empty*(*P*, *Terme*$_1$) est vrai) et évolue vers *Terme*$_1$'. *Terme*$_2$ peut attendre: *Wait*$_t$(*Terme*$_2$). Dans ces conditions, la concaténation effectue l'action *a@P(t)* et l'opérateur hiérarchique est préservé sur les termes *Terme*$_1$ décalé dans le temps *t* et *Terme*$_2$'.
- Suite à l'exécution d'une action *end*, *Terme*$_2$ termine et devient **nothing**, *Terme*$_1$ peut attendre et le prédicat *Empty*(*P*, *Terme*$_1$) est vrai (où le terme **nothing** est un terme spécial de l'algèbre désignant un processus qui a fini son exécution avec succès). Dans ce

cas, par la règle **Concat**$_5$, le terme de concaténation est remplacé par *Terme*$_2$, restreint à exécuter ses actions après ou au moment *t*.

- **Concat**$_6$: si le premier terme finit complètement son exécution sur un port qui n'apparaît pas dans le deuxième terme, la concaténation est remplacée par son deuxième terme, décalé après le temps *t*.
- **Concat**$_7$: si le premier terme finit correctement son exécution sur un port commun et le deuxième terme exécute l'action *begin* sur ce port et devient *Terme*$_2$', alors la concaténation est remplacée par *Terme*$_2$' et l'action effectuée par la concaténation est invisible aux niveaux supérieurs de la hiérarchie.

- **Concaténation**

$$\mathbf{Concat}_1)\ \frac{Terme_1 \overset{a@P(t)}{\rightarrow} Terme_1', Wait_t(Terme_2)}{\mathbf{Concat}(Terme_1, Terme_2) \overset{a@P(t)}{\rightarrow} \mathbf{Concat}(Terme_1', \mathbf{Shift}_t(Terme_2))}\ (\ a \in \mathcal{A}ct \cup \mathcal{B}\ )$$

$$\mathbf{Concat}_2)\ \frac{Terme_1 \overset{\varepsilon@P(t)}{\rightarrow} Terme_1', Terme_2 \overset{\beta@P(t)}{\rightarrow} Terme_2'}{\mathbf{Concat}(Terme_1, Terme_2) \overset{\tau@P(t)}{\rightarrow} \mathbf{Concat}(Terme_1', Terme_2')}$$

$$\mathbf{Concat}_3)\ \frac{Terme_1 \overset{\varepsilon@P(t)}{\rightarrow} Terme_1', Wait_t(Terme_2),\ Empty(P, Terme_2)}{\mathbf{Concat}(Terme_1, Terme_2) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{Concat}(Terme_1', \mathbf{Shift}_t(Terme_2))}$$

$$\mathbf{Concat}_4)\ \frac{Terme_2 \overset{a@P(t)}{\rightarrow} Terme_2', Wait_t(Terme_1)\ ,\ Empty(P, Terme_1)}{\mathbf{Concat}(Terme_1, Terme_2) \overset{a@P(t)}{\rightarrow} \mathbf{Concat}(\mathbf{Shift}_t(Terme_1), Terme_2')}\ (\ a \in \mathcal{A}ct \cup \mathcal{E} \cup \mathcal{B}\ )$$

$$\mathbf{Concat}_5)\ \frac{Terme_2 \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}, Wait_t(Terme_1)\ ,\ Empty(P, Terme_1)}{\mathbf{Concat}(Terme_1, Terme_2) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{Shift}_t(Terme_1)}$$

$$\mathbf{Concat}_6)\ \frac{Terme_1 \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}, Empty(P, Terme_2)}{\mathbf{Concat}(Terme_1, Terme_2) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{Shift}_t(Terme_2)}$$

$$\mathbf{Concat}_7)\ \frac{Terme_1 \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}, Terme_2 \overset{\beta@P(t)}{\rightarrow} Terme_2'}{\mathbf{Concat}(Terme_1, Terme_2) \overset{\tau@P(t)}{\rightarrow} Terme_2'}$$

FIGURE 6: Les règles de transition pour l'opérateur de concaténation

Le choix retardé (DChoice (*Terme,Terme*))

Les règles de transition que nous expliquons dans cette section se trouvent à la figure 7.

- **DC**$_1$: Tant que les deux termes effectuent des actions d'entrée ou d'actions spéciales *begin* sur les ports d'entrée simultanément, le choix entre les deux termes est retardé.
- **DC**$_2$: Si seulement un des deux termes peut exécuter une action réelle sur un port d'entrée au moment *t*, le choix est décidé: le terme avoir effectué l'action l'emporte. Le terme **DChoice** (*Terme$_1$*, *Terme$_2$*) est remplacé par *Terme$_i$*, qui avait exécuté l'action.
- **DC**$_3$: *Terme$_i$* effectue l'action ε@*P(t)* et le port *P* n'a pas encore exécuté l'action *end* dans *Terme$_j$*. L'exécution d'une action *end* ne décide pas le choix. L'opérateur est conservé sur *Terme$_i$'* et *Terme$_j$* qui exécute ses actions après ou au moment *t*.
- **DC**$_4$: Lorsque le port *P* est désactivé dans *Terme$_i$* et *P* n'existe pas dans *Terme$_j$* ou bien *P* est désactivé aussi dans *Terme$_j$*, le terme **DChoice** (*Terme$_1$*, *Terme$_2$*) exécute l'action ε@*P(t)* et l'opérateur hiérarchique est conservé.

- **Choix retardé**

$$\mathbf{DC}_1)\ \frac{Terme_1 \overset{a@I(t)}{\rightarrow} Terme_1', Terme_2 \overset{a@I(t)}{\rightarrow} Terme_2'}{\mathbf{DChoice}(Terme_1, Terme_2) \overset{a@I(t)}{\rightarrow} \mathbf{DChoice}(Terme_1', Terme_2')}\ (\ a \in In \cup \mathcal{B}\ )$$

$$\mathbf{DC}_2)\ \frac{Terme_i \overset{a@I(t)}{\rightarrow} Terme_i', Terme_j \overset{a@I(t)}{\not\rightarrow}}{\mathbf{DChoice}(Terme_1, Terme_2) \overset{a@I(t)}{\rightarrow} Terme_i'}\ (\ a \in In\ , i, j \in \{1,2\}, i \neq j)$$

$$\mathbf{DC}_3)\ \frac{Terme_i \overset{\varepsilon@P(t)}{\rightarrow} Terme_i', Wait_t(Terme_j)\ ,\ \neg Empty(P, Terme_j)}{\mathbf{DChoice}(Terme_1, Terme_2) \overset{\tau@P(t)}{\rightarrow} \mathbf{DChoice}(Terme_i', \mathbf{Shift}_t(Terme_j))}$$

$$\mathbf{DC}_4)\ \frac{Terme_i \overset{\varepsilon@P(t)}{\rightarrow} Terme_i', Wait_t(Terme_j)\ ,\ Empty(P, Terme_j)}{\mathbf{DChoice}(Terme_1, Terme_2) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{DChoice}(Terme_i', \mathbf{Shift}_t(Terme_j))}$$

FIGURE 7: Les règles de transition pour l'opérateur de choix retardé

La composition parallèle avec rendez-vous (RComp$_P$ (*Terme*,*Terme*))

La présente section explique les règles de transition pour l'opérateur de composition avec rendez-vous, présentées à la figure 8.

- Si un terme peut exécuter une action sur un port non-communicant et l'autre terme peut attendre, la composition parallèle exécute, elle aussi, cette action et l'opérateur est préservé sur le terme évolué suite à l'exécution de l'action et sur le terme qui est seulement décalé dans le temps (**RC**$_1$ et **RC**$_2$).
- Si les termes se synchronisent sur les actions de communication *a* et *b*, la composition se maintient (**RC**$_3$). Si les actions sont des actions *end* (**RC**$_4$), nous notons aussi que le port de communication de la composition est désactivé.
- **RC**$_5$: Lorsqu'un terme finit son exécution suite à une action ε@*P(t)* sur un port non-communicant, la composition exécute, elle aussi, cette action et est remplacée par le terme qui attend, décalé après le temps *t*.
- **RC**$_6$: Si les deux termes finissent leur exécution par la synchronisation sur ε@*P(t)*, la composition aussi termine son exécution après avoir exécuté ε@*P(t)*.

- **Composition parallèle avec rendez-vous**

$$\mathcal{RC}_1)\ \frac{Terme_1 \overset{a@P(t)}{\rightarrow} Terme'_1, Wait_t(Terme_2)}{\mathbf{RComp}_{\mathcal{P}}(Terme_1, Terme_2) \overset{a@P(t)}{\rightarrow} \mathbf{RComp}_{\mathcal{P}}(Terme'_1, \mathbf{Shift}_t(Terme_2))}\ (\ P \notin \mathcal{P}, a \in \mathcal{A}ct \cup \mathcal{B} \cup \mathcal{E}\ )$$

$$\mathcal{RC}_2)\ \frac{Terme_2 \overset{a@P(t)}{\rightarrow} Terme'_2, Wait_t(Terme_1)}{\mathbf{RComp}_{\mathcal{P}}(Terme_1, Terme_2) \overset{a@P(t)}{\rightarrow} \mathbf{RComp}_{\mathcal{P}}(\mathbf{Shift}_t(Terme_1), Terme'_2)}\ (\ P \notin \mathcal{P}, a \in \mathcal{A}ct \cup \mathcal{B} \cup \mathcal{E}\ )$$

$$\mathcal{RC}_3)\ \frac{Terme_1 \overset{a@P(t)}{\rightarrow} Terme'_1, Terme_2 \overset{b@P(t)}{\rightarrow} Terme'_2}{\mathbf{RComp}_{\mathcal{P}}(Terme_1, Terme_2) \overset{a|b@P(t)}{\rightarrow} \mathbf{RComp}_{\mathcal{P}}(Terme'_1, Terme'_2)}\ (\ P \in \mathcal{P}, a \in \mathcal{A}ct \cup \mathcal{B}\ )$$

$$\mathcal{RC}_4)\ \frac{Terme_1 \overset{\varepsilon@P(t)}{\rightarrow} Terme'_1, Terme_2 \overset{\varepsilon@P(t)}{\rightarrow} Terme'_2}{\mathbf{RComp}_{\mathcal{P}}(Terme_1, Terme_2) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{RComp}_{\mathcal{P}-\{\mathcal{P}\}}(Terme'_1, Terme'_2)}\ (\ P \in \mathcal{P}\ )$$

$$\mathcal{RC}_5)\ \frac{Terme_i \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}, Wait_t(Terme_j)}{\mathbf{RComp}_{\mathcal{P}}(Terme_1, Terme_2) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{Shift}_t(Terme_j)}\ (\ P \notin \mathcal{P}\ , i, j \in \{1, 2\}, i \neq j)$$

$$\mathcal{RC}_6)\ \frac{Terme_1 \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}, Terme_2 \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}}{\mathbf{RComp}_{\mathcal{P}}(Terme_1, Terme_2) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}}\ (\ P \in \mathcal{P}\ )$$

FIGURE 8: Les règles de transition pour l'opérateur de composition parallèle avec rendez-vous

L'exception (Excep (T,T_{cond}, T_{excep})**)**

Nous passons en revue maintenant les règles de transition pour l'opérateur d'exception, présentées à la figure 9.

- **Excep**$_1$: Le comportement normal T et la condition d'exception T_{cond} exécutent simultanément une action réelle ou une action *begin* sur un port d'entrée. L'opérateur d'exception est conservé après avoir exécuté la même action que son terme.
- **Excep**$_2$: Le port d'entrée I appartenant uniquement au comportement normal est activé, étant donné que le comportement normal a déjà été activé (car le prédicat *Activated*(T) est vrai). Dans ces conditions, l'opérateur d'exception est préservé, après l'exécution de $\beta@P(t)$ par le terme hiérarchique d'exception.
- **Excep**$_3$: La situation est similaire à celle décrite par la règle de transition **Excep**$_2$, à l'exception du prédicat *Activated*, qui est absent de **Excep**$_3$.
- **Excep**$_4$ et **Excep**$_5$: Le comportement normal T (la condition d'exception T_{cond}, respectivement) exécute un action *end* sur un port et la condition T_{cond} (le comportement normal T, respectivement) peut attendre. Cette action ne coïncide pas avec la terminaison correcte de T (T_{cond}, respectivement). Dans ce cas, l'opérateur hiérarchique d'exception est préservé et aucune action n'est visible aux niveaux supérieurs de la hiérarchie.
- **Excep**$_6$ et **Excep**$_7$: Le comportement normal T (la condition T_{cond}, respectivement) exécute l'action a, impossible à être effectuée au moment t par la condition (le comportement, respectivement), qui n'a aucune action urgente à exécuter. La construction hiérarchique est conservée après avoir exécuté l'action a.
- **Excep**$_8$ et **Excep**$_9$: Ces règles de transition sont similaires à **Excep**$_6$ et **Excep**$_7$, mais, de plus, l'action $\varepsilon@P(t)$ effectuée par un terme (comportement normal ou condition d'exception) coïncide avec la fin de l'exécution du terme. Dans ces conditions, le terme hiérarchique effectue $\varepsilon@P(t)$. Ensuite, il est remplacé par le comportement d'exception exécutant ses actions après le temps t, pour la règle **Excep**$_8$, ou il est abandonné, dans le cas de la règle **Excep**$_9$.

- **Exception**

$$\mathbf{Excep}_1)\ \frac{T \overset{a@I(t)}{\rightarrow} T', T_{cond} \overset{a@I(t)}{\rightarrow} T'_{cond}}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{a@I(t)}{\rightarrow} \mathbf{Excep}(T', T'_{cond}, T_{excep})}\ (\ a \in In \cup \mathcal{B}\ )$$

$$\mathbf{Excep}_2)\ \frac{T_{cond} \overset{\beta@I(t)}{\rightarrow} T'_{cond}, Wait_t(T), Activated(T), Empty(P,T)}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{\beta@I(t)}{\rightarrow} \mathbf{Excep}(\mathbf{Shift}_t(T), T'_{cond}, T_{excep})}$$

$$\mathbf{Excep}_3)\ \frac{T \overset{\beta@P(t)}{\rightarrow} T', Wait_t(T_{cond}), Empty(P, T_{cond})}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{\beta@P(t)}{\rightarrow} \mathbf{Excep}(T', \mathbf{Shift}_t(T_{cond}), T_{excep})}$$

$$\mathbf{Excep}_4)\ \frac{T \overset{\varepsilon@P(t)}{\rightarrow} T', Wait_t(T_{cond})}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{\tau@P(t)}{\rightarrow} \mathbf{Excep}(T', \mathbf{Shift}_t(T_{cond}), T_{excep})}$$

$$\mathbf{Excep}_5)\ \frac{T_{cond} \overset{\varepsilon@I(t)}{\rightarrow} T'_{cond}, Wait_t(T)}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{\tau@I(t)}{\rightarrow} \mathbf{Excep}(\mathbf{Shift}_t(T), T'_{cond}, T_{excep})}$$

$$\mathbf{Excep}_6)\ \frac{T \overset{a@P(t)}{\rightarrow} T', T_{cond} \overset{a@P(t)}{\not\rightarrow}, Wait_t(T_{cond})}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{a@P(t)}{\rightarrow} \mathbf{Excep}(T', \mathbf{Shift}_t(T_{cond}), T_{excep})}\ (\ a \in \mathcal{A}ct)$$

$$\mathbf{Excep}_7)\ \frac{T \overset{a@I(t)}{\not\rightarrow}, T_{cond} \overset{a@I(t)}{\rightarrow} T'_{cond}, Wait_t(T)}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{a@I(t)}{\rightarrow} \mathbf{Excep}(\mathbf{Shift}_t(T), T'_{cond}, T_{excep})}\ (\ a \in In)$$

$$\mathbf{Excep}_8)\ \frac{T_{cond} \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}, Wait_t(T)}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{Shift}_t(T_{excep})}$$

$$\mathbf{Excep}_9)\ \frac{T \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}, Wait_t(T_{cond})}{\mathbf{Excep}(T, T_{cond}, T_{excep}) \overset{\varepsilon@P(t)}{\rightarrow} \mathbf{nothing}}$$

FIGURE 9: Les règles de transition pour l'opérateur d'exception

La récursion recX. (*Terme*)

La règle de transition de la récursion dans la figure 10 exprime le principe suivant: aussitôt que *Terme* exécute une action, la récursion exécute cette action et la variable X dans *Terme* est remplacée par *Terme*.

- **Récursion**

$$\textbf{rec)}\ \frac{Chart \overset{a@P(t)}{\rightarrow} Chart'}{\mathbf{rec}X.(Chart) \overset{a@P(t)}{\rightarrow} Chart'[X/\mathbf{rec}X.(Chart)]}\ (a \in Act \cup \mathcal{B} \cup \mathcal{E})$$

FIGURE 10: La règle de transition pour l'opérateur de récursion

Dans la section suivante nous identifions les opérations de base (ou les "primitives") participant à la construction des règles de transition. Une classification des règles de transition, aidant à l'implantation, sera présentée plus tard, à la section 3.4.

3.3 Les primitives de l'algèbre de processus ACTC^P

En analysant la structure des règles de la sémantique opérationnelle, nous pouvons extraire des "primitives", des groupements d'informations facilement identifiables, apparaissant dans plusieurs règles de transition (soit dans la partie "hypothèse" ou dans la partie "conclusion" des règles) et que nous utilisons dans notre logiciel pour la construction et la vérification de l'applicabilité des règles de transition, lors de la simulation, comme les briques dans la maçonnerie: individuellement, chaque unité.
Les primitives utilisées sont les suivantes:

- l'occurrence d'une action $a \in Act \cup \mathcal{B} \cup \mathcal{E}$ (c.à.d., action d'entrée / sortie / action spé-

ciale *begin* ou *end*) sur un port donné,

- l'évolution du terme (de *Terme* vers *Terme'*) exprimant le changement de l'état de *Terme* suite à l'occurrence d'une action,
- le prédicat *Empty*(*P, Terme*) qui est vrai lorsque *Terme* a fini correctement l'exécution de toutes les actions sur le port *P* ou si *Terme* n'est pas défini sur *P,*
- l'information sur la communication des ports: le port P appartient à l'ensemble *P* des ports de communication (P $\in$ *P*) s'il est partagé par plusieurs termes,
- le prédicat $Wait_t$(*Terme*) qui est vrai si *Terme* n'a aucune action urgente à exécuter au moment *t*,
- l'opérateur $\mathbf{Shift}_t$(*Terme*): *Terme* est restreint à exécuter ses actions après ou au moment *t*,
- le terme **nothing** signifiant la terminaison correcte de l'exécution d'un terme,
- le prédicat *Activated*(*Terme*) qui est vrai si le diagramme *Terme* est activé (après l'exécution du premier begin sur l'ensemble de ses ports).

3.4 La classification des règles de transition de l'algèbre de processus $ACTC^P$

Dans le but de structurer le traitement des actions à travers le corps du processus d'exécution d'un modèle donné et de faciliter la tâche d'un futur usager de modifier la sémantique opérationnelle pour des besoins spécifiques, nous classifions les situations décrites par les règles de transition dans 6 catégories, suivant les actions exécutées, leur influence sur la structure de la hiérarchie (création de nouvelles instances de diagrammes) et sur l'état actif ou inactif des diagrammes et / ou des ports:

1. Les règles de transition menant à l'activation de diagrammes (et de ports) au moment de l'initialisation de la hiérarchie (début de l'exécution).
   - L'action exécutée est un *begin*,
   - Les règles suivantes font partie à cette catégorie:

     $\mathbf{Concat}_1$, $\mathbf{Concat}_4$

RC_1, RC_2, RC_3

$Excep_1$, $Excep_2$, $Excep_3$

DC_1

2. Les règles de transition menant à la désactivation de ports (et, éventuellement, à l'activation de ports ou même de diagrammes dans la hiérarchie).
   - L'action (ou les actions) spécifique dans ce cas est *end* (et *begin* pour **$Concat_2$**, **$Concat_7$**), ce qui est une condition commune à toutes les hypothèses des règles de transition appartenant à cette catégorie.
   - Les règles de transition faisant partie de la catégorie 2 sont:

     $Concat_2$, $Concat_3$, $Concat_4$, $Concat_5$, $Concat_6$, $Concat_7$

     RC_1, RC_2, RC_4, RC_5, RC_6

     $Excep_4$, $Excep_5$, $Excep_8$, $Excep_9$

     DC_3, DC_4

3. Les règles de transition qui engendrent la désactivation de diagrammes (la désactivation de ports dans une feuille coïncide avec la désactivation de diagrammes-père de la feuille).
   - L'action (ou les actions) caractéristique est *end* $\in \mathfrak{E}$.
   - Les règles de transition décrivant cette situation sont:

     $Concat_5$, $Concat_6$, $Concat_7$

     RC_5, RC_6

     $Excep_8$, $Excep_9$

4. Les règles de transition qui ne modifient aucunement la hiérarchie (de nouvelles instances de diagrammes ne sont pas créées et il n'y a pas d'activation ou désactivation de diagrammes). Elles sont appliquées à l'intérieur des diagrammes feuilles et leur traitement ne dépend pas du type d'opérateur père de la feuille.
   - Les actions exécutées spécifiques à cette catégorie sont les actions $a \in Act$.
   - Les règles sémantiques dans ce cas sont:

     $Concat_1$, $Concat_4$

     RC_1, RC_2, RC_3

Excep$_1$, **Excep**$_6$, **Excep**$_7$

DC$_1$

5. La règle de la récursion (l'opérateur **recX**) modifie la hiérarchie par la création de nouvelles instances de diagrammes soit au moment de l'initialisation de la hiérarchie (tel que les situations appartenant à la catégorie 1) ou suite à des désactivations de ports décrites par les situations de type 2 (la feuille ayant désactivé un port et le diagramme-noeud **recX** ont un ancêtre commun de type concaténation). L'action gâchette de la récursion est la première action *begin* exécutée sur un port du terme.
6. La règle de transition **DC**$_2$ traite des actions d'entrée $a \in In$ (qui, généralement, n'ont pas d'influence sur l'état des diagrammes, tel que les situations de type 4): les termes pour lesquels il est impossible d'effectuer l'action a sont désactivés.

Ce chapitre a présenté la sémantique opérationnelle de l'algèbre ACTC^P et les diverses notions nécessaires au lecteur pour la compréhension de l'implantation des règles de transition.

Le chapitre suivant décrit représentation des diagrammes d'actions, telle qu'elle est utilisée par le système de simulation, ainsi que le principe de simulation des diagrammes d'actions. Par la suite, nous présentons la manière dans laquelle nous avons implanté les primitives de l'algèbre ACTC^P ainsi que la correspondance préservée entre les règles de transition et l'exécution du modèle. La description de cette correspondance suit les catégories vues lors de la classification des règles de transition (la section 3.4).

Chapitre 4. Algorithmes de simulation et implantation

Ayant présenté la sémantique opérationnelle et les règles de transition de l'algèbre de processus $ACTC^P$, nous pouvons analyser maintenant de quelle manière notre implantation est réalisée et comment elle respecte les transitions d'état permises au système.

Il est important à noter que, malgré le caractère binaire de l'algèbre $ACTC^P$ (tous les opérateurs hiérarchiques, à l'exception de la récursion, sont des opérateurs binaires), notre implantation est plus générale, elle acceptant ainsi des opérateurs n-aires ($n \geq 2$). Toutefois, pour faciliter la tâche au lecteur, nous présentons les algorithmes de simulation en préservant le caractère binaire de la hiérarchie.

La section suivante présente les structures de données utilisées pour la représentation des diagrammes d'actions feuilles et hiérarchiques, telles qu'elles sont implantées dans notre système de simulation.

4.1 La représentation des diagrammes d'actions à l'intérieur du processus de simulation

La structure de données représentant des diagrammes feuilles ou hiérarchiques dans le processus se ressemble à leur spécification dans le langage HAAD (voir la section 2.3), à laquelle nous avons ajouté des informations supplémentaires nécessaires à l'implantation de l'algorithme de simulation.

Un diagramme hiérarchique est représenté par un arbre où chaque noeud interne est un diagramme hiérarchique (correspondant à un opérateur hiérarchique dans l'algèbre $ACTC^P$) et les feuilles de l'arbre correspondent aux diagrammes feuilles. Dans le code VHDL, les noeuds de l'arbre sont définis par le type **had_node**, chacun ayant une des étiquettes suivantes: feuille ("LEAF_AD"), concaténation ("CONCAT_AD"), composition parallèle avec rendez-vous ("CONCUR_AD"), choix retardé ("D_CHOICE_AD"), récursion ("RECX"), exception ("EXCEPTION_AD") avec le comportement normal ("NORMAL_AD"), la condition d'exception ("CONDITION_AD") et le comportement

d'exception ("HANDLER_AD") (voir la figure 11).

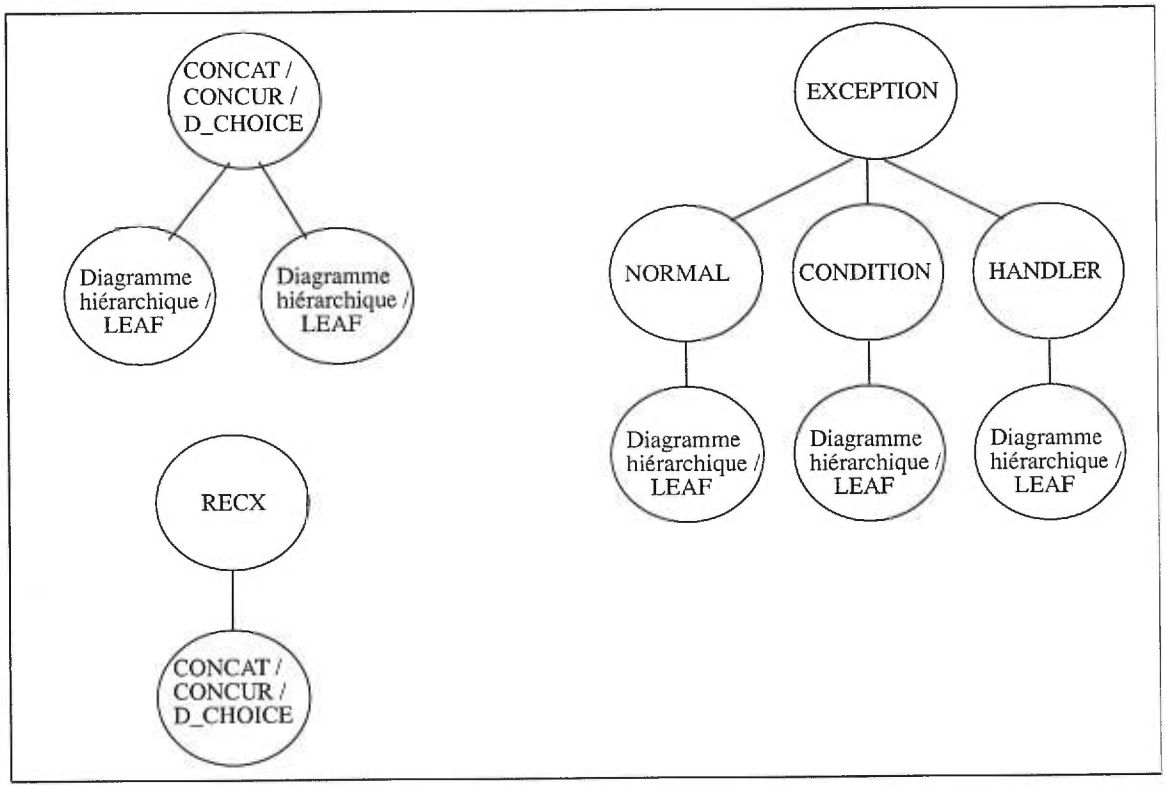


FIGURE 11: La représentation de divers diagrammes hiérarchiques

La recherche dans la hiérarchie, effectuée lors de la simulation, sert à coupler une action réelle observée sur les ports de l'entité avec une action spécifiée et à identifier l'évolution des termes selon les règles sémantiques. Ce parcours visite tous les noeuds actifs (l'information nécessaire pour identifier les noeuds actifs se trouve dans l'attribut *active* de type booléen du noeud) et descend vers les feuilles actives. Les diagrammes sont actifs si leur action START a eu lieu, mais leur action END n'a pas encore été effectuée (voir la section 2.1).

Pour toute recherche dans la hiérarchie, le parcours de l'arbre se fait en post-ordre, ce qui garantit le respect de la sémantique de l'opérateur de concaténation. Par conséquent, les fils d'un noeud de type "concaténation" seront toujours visités dans le "bon" ordre, c.à.d.,

de gauche à droite, dans la manière que l'opérateur a été défini.

Chaque diagramme feuille de l'arbre possède une liste de ports, où un port est une structure définie par **h_port** dans le code.

Les plus importants attributs d'un port sont:

- la valeur initiale sur le port (*init_value*),
- un pointeur vers la liste d'actions spécifiées sur le port (*first_action*),
- la dernière action spécifiée sur le port (*end_action*: l'action spéciale *end*),
- un pointeur vers l'action courante spécifiée sur le port; elle est la prochaine action à faire correspondre avec une action réelle,
- une correspondance (l'attribut *mapping*) vers un port de l'entité sur lequel nous observons l'activité (pour tout port d'une feuille, il existe un seul port de l'entité qui lui correspond selon la correspondance transitive de "PORT-MAP" dans le modèle HAAD, voir la figure 5),
- le diagramme auquel le port appartient (*had*)

Le diagramme feuille contient aussi l'information sur l'ensemble des contraintes temporelles, organisées dans un graphe orienté (voir [5] pour des détails) dont les arcs, étiquetés par des intervalles temporels, représentent les contraintes temporelles correspondantes et les sommets sont les actions spécifiées, de type **spec_action**.

Parmi les caractéristiques les plus importantes des actions spec_action nous notons:

- le type de l'action: CONSTANT (une seule action réelle correspond à l'action spécifiée), VALID, DONT_CARE (zéro ou plusieurs actions, appelées événements indistinguables, correspondent à l'action spécifiée - voir [2, 27] pour plus de détails), START_ACTION (dans le cas de l'action *begin* sur un port quelconque et de l'action virtuelle START indiquant le début de l'exécution d'un diagramme) et END_ACTION (pour l'action *end* sur un port et pour l'action virtuelle END de fin de diagramme),
- la direction de l'action (INPUT_D, OUTPUT_D, IO_D) et sa valeur (*value*),
- le port (*h_port*) et le diagramme feuille (*had*) auxquels l'action appartient,
- le temps d'occurrence de l'observation de l'action (indique le moment quand la correspondance entre l'action réelle et l'action spécifiée a été faite),
- l'attribut *occurred* indiquant que l'action spec_action a été exécutée,
- un pointeur vers la prochaine action se trouvant sur le même port (*next_action*),

- les contraintes temporelles dont l'action est la source (*tc_out*),
- les contraintes temporelles *assume* et *commit* dont l'action est le puits,
- l'instant de génération des actions de sortie (*trigger_time*) réalisées par le modèle, appartenant à l'intervalle temporel d'occurrence établi par les contraintes *commit* dont l'action est le puits.

Les contraintes temporelles sont définies à l'aide du type **tc** ayant des informations concernant:

- le type d'opérateur de composition des contraintes (*latest*, *earliest*, *conjunctive*),
- le type de contrainte (*assume* ou *commit*),
- les actions source et puits (*source*, *sink*),
- les bornes de l'intervalle constituant la relation temporelle (*min*, *max*)

Dans la prochaine section nous présentons le principe de simulation des diagrammes d'actions utilisé par notre logiciel pour la vérification des interfaces.

4.2 Le principe de simulation des diagrammes d'actions

La sémantique opérationnelle d'un diagramme d'actions est exprimée sous la forme de son exécution dans un environnement qui fournit les actions sur les ports d'entrée et bidirectionnels et qui observe les ports de sortie et bidirectionnels.

L'exécution du diagramme débute par l'action virtuelle START et continue avec les actions spécifiées dans un ordre dépendant de leur spécification sur les ports et des relations temporelles (contraintes temporelles) entre elles.

Les *actions spécifiées* composent la spécification de l'interface du système, tandis que les changements de valeurs perçus sur les ports de l'interface du système lors de la simulation (de l'exécution du modèle VHDL) représentent les *actions réelles*.

Une erreur d'exécution apparaît dans un diagramme feuille dans un des cas suivants:

- la valeur initiale spécifiée sur un port n'est pas respectée (la valeur de l'action réelle générée par l'environnement ne correspond pas à la valeur de l'action spécifiée)
- la séquence des actions d'entrée observée ne correspond pas à la séquence spécifiée (les

contraintes *assume* qui lient les moments d'occurrence des actions ne sont pas respectées)

- une action inattendue apparaît sur un port

Si une erreur d'exécution apparaît pendant la simulation, le diagramme est abandonné et la suite dépend du contexte hiérarchique du diagramme.

Pendant la simulation, le système exerce deux fonctions:

- il valide les actions d'entrée et de sortie observées sur le port (en comparant le moment d'occurrence et la valeur sur le port des actions réelles par rapport aux contraintes temporelles et aux actions spécifiées) et
- il produit les actions réelles de sortie en respectant la valeur sur le port et l'instant d'occurrence spécifiés (en fonction des contraintes *commit* dont l'action de sortie est la destination).

Nous examinons maintenant l'idée générale du processus général de l'exécution du modèle.

Le temps de simulation avance jusqu'au moment où une ou les deux conditions d'arrêt suivantes soient satisfaites:

1. Une actions réelle est perçue sur un port ou
2. Le temps calculé sur le système de contraintes indique que l'instant courant de simulation correspond au moment limite d'occurrence d'une action d'entrée ou le temps d'occurrence projeté d'une action de sortie (nous sommes dans le cas d'un délai expiré, appelé *timeout*) et alors toute la hiérarchie de diagrammes d'actions est parcourue afin de:

- coupler l'action réelle avec l'action ou les actions spécifiées correspondantes (dans la situation de la condition 1) ou
- décider quelle est l'action dont le délai expiré *timeout* de son occurrence est arrivé

Si la paire (action spécifiée, action réelle) est trouvée, l'action est exécutée. L'exécution d'une action consiste en une série d'opérations effectuées par le logiciel, comme par exemple:

- enregistrer le moment d'occurrence de l'action courante,
- mettre à "vrai" le booléen *occurred* de l'action courante,
- calculer l'intervalle temporel d'occurrence des successeurs de l'action courante et des

contraintes temporelles dont l'action courante est la source,

- déplacer le pointeur *next_action* sur la prochaine action spécifiée sur le même port.

Pendant l'exécution de toutes les actions à l'instant courant de la simulation, l'écoulement du temps est interrompu (l'exécution des actions prends un temps zéro du point de vue de la simulation).

Dans le cas où l'action réelle n'a pu être jumelée avec une action spécifiée ou dans la situation où une action d'entrée n'a pas été observée au moment de son délai expiré *timeout*, une erreur est signalée au niveau de la feuille. L'erreur est propagée vers les diagrammes hiérarchiques auxquels le diagramme feuille appartient dans le but de conclure dans chaque cas, selon la sémantique de chaque diagramme hiérarchique visité, si l'erreur peut être remontée ou non vers la racine. Par exemple, une erreur dans un des diagrammes feuilles d'un diagramme hiérarchique de type concaténation se traduit dans une erreur dans la concaténation, tandis qu'une erreur dans un diagramme feuille participant à un choix retardé ne mène pas nécessairement vers une erreur du diagramme hiérarchique).

Si aucune erreur n'est détectée avant l'exécution de l'action virtuelle END du diagramme, alors le diagramme complète (avec succès) son exécution et il est désactivé.

L'exécution des diagrammes d'actions hiérarchiques suit la sémantique opérationnelle de l'algèbre de processus ACTC^P telle qu'elle sera décrite au chapitre 3.

En ce qui concerne les diagrammes feuilles et les diagrammes hiérarchiques d'actions, tout contexte qui ne correspond pas à une règle de transition de l'algèbre ACTC^P entraîne la signalisation d'une erreur. Du point de vue des opérateurs hiérarchiques, seule l'exécution d'une action engendre une prise de décision sur la règle de transition s'appliquant. Nous pouvons ainsi dire que les actions sont des gâchettes déclenchant une recherche dans la hiérarchie dans le but d'appliquer les règles de transition correspondantes.

Le processus du modèle VHDL (voir la figure 12) consiste, en lignes générales, en deux parties:

- La création de la hiérarchie de diagrammes,
- Une boucle réalisant l'observation des actions spécifiées, la génération de l'activité sur les ports et la mise-à-jour de la hiérarchie suite à l'exécution des actions.

Les opérations réalisées par le processus seront détaillées dans le chapitre 4.

```
VARIABLE root, finished_hads: ptr_had_node;
VARIABLE candidats: ptr_h_port := null;
VARIABLE activated: boolean := false;
VARIABLE progress: boolean := true;
VARIABLE time_out: time := time'HIGH;
VARIABLE real_actions, out_actions, end_actions, ptr_end_action: ptr_spec_action;
Begin -- PROCESS
        load1(root); -- création de la hiérarchie de diagrammes à simuler, où "root" est la racine de l'arbre
        root.finish := false;
        root.error := false;
                        -- détecter les actions spéciales begin à être exécutées au moment courant et
                        -- les insérer dans la liste "candidats":
        activate_possible_ports (root, candidats);
                        -- activer les diagrammes feuilles qui contiennent les ports candidats et les
                        -- diagrammes hiérarchiques ancêtres de ces diagrammes feuilles:
        activate_possible_diagrams (candidats);
                        -- exécuter les actions begin sur les ports candidats:
        init_actions_on_port (candidats, root);
                        -- exécuter la boucle tant que la racine n'a pas fini (NOT root.finish) et qu'il
                        -- n'y a pas d'erreur propagée à la racine (NOT root.error):
        WHILE NOT root.finish AND NOT root.error LOOP
                        -- tant qu'il y a des actions end dans la liste end_actions au moment courant:
                WHILE ptr_end_action /= null LOOP
                        ptr_end_action := end_actions;
                                -- détecter les diagrammes qui ont fini leur exécution et les insérer
                                -- dans la liste finished_hads:
                        find_finished_hads (ptr_end_action, finished_hads);
                                -- désactiver les diagrammes se trouvant dans la liste finished_hads:
                        terminate_hads (finished_hads);
                                -- pour une action end sur port_to_check, déterminer les ports
                                -- correspondants qui peuvent exécuter leur action begin et les
                                -- insérer dans la liste candidats:
                        start_ports (ptr_end_action.had, ptr_end_action.h_port, activated, root,
                                                                                candidats);
                        finished_hads := null;
                        end_action := null;
                        activate_possible_diagrams (candidats);
                        init_actions_on_port (candidats, root);
                        ptr_end_action := ptr_end_action.link;
                END LOOP;
                                -- calculer l'instant d'occurrence de la prochaine action à observer ou
                                -- à générer en fonction du graphe de contraintes temporelles:
                        compute_time_out (root, time_out);
                                -- attente jusqu'à l'observation d'une action réelle sur les ports de la
                                -- racine ou l'arrivée du délais expiré time-out:
                        WAIT ON ports_de_l'entité FOR time_out;
                                        -- détecter toutes les actions d'entrée et de sortie à exécuter au
                                        -- moment courant et les insérer dans la liste real_actions:
                                catch_possible_actions (root, real_actions);
                                        -- récolter les actions de sortie dans la liste out_actions et
                                        -- les actions spéciales end dans la liste end_actions:
                                catch_generated_actions (root, out_actions, end_actions);
                                        -- effectuer les actions de sortie et les actions end:
```

```
                process_output (out_actions, end_actions, root);
                        -- effectuer les actions d'entrée (les actions de sortie ne sont pas
                        -- prises en compte):
                process_reals (real_actions, root);
                        -- vérifier que l'instant d'occurrence des actions dans la liste
                        -- real_actions respecte l'intervalle temporel d'occurrence et
                        -- que leur valeur est égale à la valeur courante du port:
                validate_ports_activities (real_actions);
                        -- pour toutes les actions ayant une variable attachée, affecter à
                        -- la variable la valeur du port contenant l'action:
                auto_var_side_effects (real_actions);
                        -- appeler les prédicats et les procédures attachées aux actions
                        -- qui sont exécutées au moment courant:
                call_action_pred_and_proc (real_actions, out_actions);
                        -- affecter au port correspondant la valeur de l'action de sortie
                        -- ou de la variable attachée à cette action:
                generate_port_activity (out_actions);
                real_actions := null;
                out_actions := null;
        END LOOP;
END PROCESS;
```

FIGURE 12: La structure du processus VHDL d'une spécification HAAD

La représentation des diagrammes d'actions à l'intérieur du processus de simulation et le principe de leur simulation étant vus lors des deux sections précédentes, nous présentons par la suite la manière dans laquelle notre système de simulation interprète la sémantique opérationnelle de l'algèbre ACTC^P.

4.3 La réalisation des primitives de l'algèbre de processus ACTC^P

Les opérations de base (ou "primitives") de l'algèbre ACTC^P, sont reflétées par divers services mis à notre disposition par le simulateur ou par des constructions que nous avons implantées dans le processus de simulation. Nous présentons dans cette section la manière dans laquelle les primitives de l'algèbre sont réalisées à l'intérieur de notre système de simulation. Ces notions seront utilisées dans la section 4.4, lors de l'explication de la correspondance entre la sémantique opérationnelle de l'algèbre ACTC^P et l'exécution du

modèle.

1) "**nothing**"

Après la terminaison sans erreur d'un diagramme feuille ou hiérarchique (quand tous les actions ont été exécutées), le diagramme est inséré dans la liste *finished_hads* par la procédure *find_finished_hads* (*ptr_end_action, finished_hads*), appelée dans le processus - voir les figures 13 et 14.

Dans la procédure *terminate_hads* (*finished_hads*), l'action virtuelle END de chaque diagramme dans la liste *finished_hads* est exécutée et le diagramme est désactivé (voir les figures 15 à 19). Un diagramme feuille ou hiérarchique désactivé n'est pas inclus dans le parcours de la hiérarchie lors de la simulation.

```
PROCEDURE find_finished_hads
                        (ptr_end_action : INOUT ptr_spec_action;
                        finished_hads : INOUT ptr_had_node) IS

BEGIN
        -- vérifier si le diagramme feuille a fini son exécution
        check_if_leaf_end (ptr_end_action, finished_hads);
        -- et le désactiver si c'est le cas
        IF ptr_end_action.had.finish THEN
                end_diag (ptr_end_action.had, ptr_end_action.h_port, finished_hads);
        END IF;
END find_finished__hads;
```

FIGURE 13: La procédure *find_finished_hads*

```
PROCEDURE check_if_leaf_end
                              (leaf_had : INOUT ptr_had_node;
                              finished_hads : INOUT ptr_had_node) IS
 VARIABLE ptr_port : ptr_h_port := leaf_had.ports_list;
 VARIABLE finish : boolean := true;

-- si tous les ports du diagramme feuille ont fini alors la feuille finit et est insérée dans la liste finished_hads;

BEGIN
 WHILE ptr_port /= null LOOP
  finish := finish AND ptr_port.current_action = null
  ptr_port := ptr_port.brother;
 END LOOP;
 leaf_had.finish := finish;
 IF finish THEN
   insérer leaf_had dans la liste finished_hads;
 END IF;
END check_if_leaf_end;
```

FIGURE 14: La procédure *check_if_leaf_end*

```
PROCEDURE terminate_hads
                              (finished_hads : INOUT ptr_had_node) IS
 VARIABLE ptr_had : ptr_had_node := finished_hads;

-- désactiver les diagrammes dans la liste finished_hads

BEGIN
 WHILE ptr_had /= null LOOP
  ptr_had.active := false;
  ptr_had := ptr_had.link;
 END LOOP;
END terminate_hads;
```

FIGURE 15: La procédure *terminate_hads*

```
PROCEDURE end_diag
                        (had_to_end : INOUT ptr_had_node;
                        port_to_check : INOUT ptr_h_port;
                        finished_hads : INOUT ptr_had_node) IS
 VARIABLE still_active : boolean := false;
 VARIABLE ptr_had : ptr_had_node := had_to_end;

-- vérifier quels sont les diagrammes hiérarchiques qui ont fini leur exécution et
-- les insèrer dans finished_hads

BEGIN
 WHILE ptr_had /= null AND NOT still_active LOOP
  IF ptr_had.father /= null THEN
    CASE ptr_had.father.kind IS
     WHEN CONCAT_AD => end_concur_concat (ptr_had, port_to_check);
     WHEN CONCUR_AD => end_concur_concat (ptr_had, port_to_check);
     WHEN D_CHOICE_AD => end_choice (ptr_had, port_to_check);
     WHEN HANDLER_AD => end_exception (ptr_had, port_to_check);
     WHEN NORMAL_AD => end_exception (ptr_had, port_to_check);
     WHEN CONDITION_AD => end_exception (ptr_had, port_to_check);
     WHEN EXCEPTION_AD => end_exception (ptr_had, port_to_check);
     WHEN RECX => still_active := true; -- la récursion ne finit pas
      WHEN OTHERS => null;
    END CASE;
  END IF;
  IF ptr_had.father /= null AND ptr_had.father.finish AND NOT ptr_had.father.error THEN
    insérer ptr_had.father dans la liste finished_hads;
    ptr_had := ptr_had.father;
  ELSE
    still_active := true;
  END IF;
 END LOOP;
END end_diag;
```

FIGURE 16: La procédure *end_diag*

```
PROCEDURE end_choice
                          (had_to_end : INOUT ptr_had_node;
                          port_to_check : INOUT ptr_h_port) IS
 VARIABLE ptr_brother : ptr_had_node := had_to_end.brother;
 VARIABLE only_one : boolean := true;

BEGIN
 IF NOT (ptr_brother.error OR ptr_brother.disabled) THEN
   had_to_end.father.error := true;
   propager l'erreur vers la racine
 ELSE
   had_to_end.father.finish := true;
 END IF;
END end_choice;
```

FIGURE 17: La procédure *end_choice*

```
PROCEDURE end_concur_concat
                          (had_to_end : INOUT ptr_had_node;
                          port_to_check : INOUT ptr_h_port) IS
 VARIABLE ptr_brother : ptr_had_node := had_to_end.brother;
 VARIABLE finish : boolean := true;

BEGIN
 IF NOT had_to_end.father.error THEN
   IF ptr_brother.error THEN
     had_to_end.father.error := true;
     propager l'erreur vers la racine
   END IF;
   finish := finish AND ptr_brother.finish;
 END IF;
 had_to_end.father.finish := finish;
END end_concur_concat;
```

FIGURE 18: La procédure *end_concur_concat*

```
PROCEDURE end_exception
                        (had_to_end : INOUT ptr_had_node;
                        port_to_check : INOUT ptr_h_port) IS

BEGIN
 IF had_to_end.error = true THEN
   ok := false;
   had_to_end.father.error := true;
   propager l'erreur vers la racine de l'arbre;
 END IF;
 CASE had_to_end.father.kind IS
                -- si had_to_end appartient au comportement normal
   WHEN NORMAL_AD =>
        had_to_end.father.finish := true;          -- finir le comportement normal
                -- Activated(T) devient faux:
        had_to_end.father.min_one_activated_port := false;
        finir la condition d'exception et toute sa hiérarchie de diagrammes;
                -- si had_to_end appartient à la condition d'exception:
   WHEN CONDITION_AD =>
        had_to_end.father.finish := true;          -- finir la condition d'exception
        finir le comportement normal et toute sa hiérarchie de diagrammes;
                -- si had_to_end appartient au comportement d'exception
   WHEN HANDLER_AD =>
        had_to_end.father.finish := true;          -- finir le comportement d'exception
                -- si had_to_end appartient à l'exception:
   WHEN EXCEPTION_AD =>
        IF had_to_end.kind = NORMAL_AD or had_to_end.kind = HANDLER_AD THEN
                had_to_end.father.finish := true;          -- finir l'exception
        END IF;
 END CASE;
END end_exception;
```

FIGURE 19: La procédure *end_exception*

Nous pouvons remarquer que les règles de transition traitant de la désactivation d'un diagramme (évolution vers "nothing" d'un terme) découlent des règles où la terminaison d'un port ne coïncide pas avec la terminaison de l'exécution du diagramme. Dans l'exemple de la figure 20, dans le cas de l'occurrence d'une action *end* sur un port, nous pouvons observer que la différence entre la règle $\mathbf{Concat}_4$ et la règle $\mathbf{Concat}_5$ provient de la terminaison du deuxième terme de la concaténation, qui devient **nothing**. La disparition de ce terme engendre aussi la disparition de l'opérateur de concaténation, du point de vue de la sémantique opérationnelle, ce qui équivaut à la désactivation du diagramme hiérarchique Concaténation dans notre système de simulation.

Nous pouvons ainsi affirmer que, dans les deux types de règles de transition, l'exécution des diagrammes est identique jusqu'à un certain point. La seule différence entre les deux types de règles consiste, dans le cas des règles où un terme finit son exécution, dans l'ajout des opérations de localisation et de désactivation du terme qui a fini son exécution. Ces opérations sont reflétées dans les procédures décrites, *find_finished_hads* et *terminate_hads*.

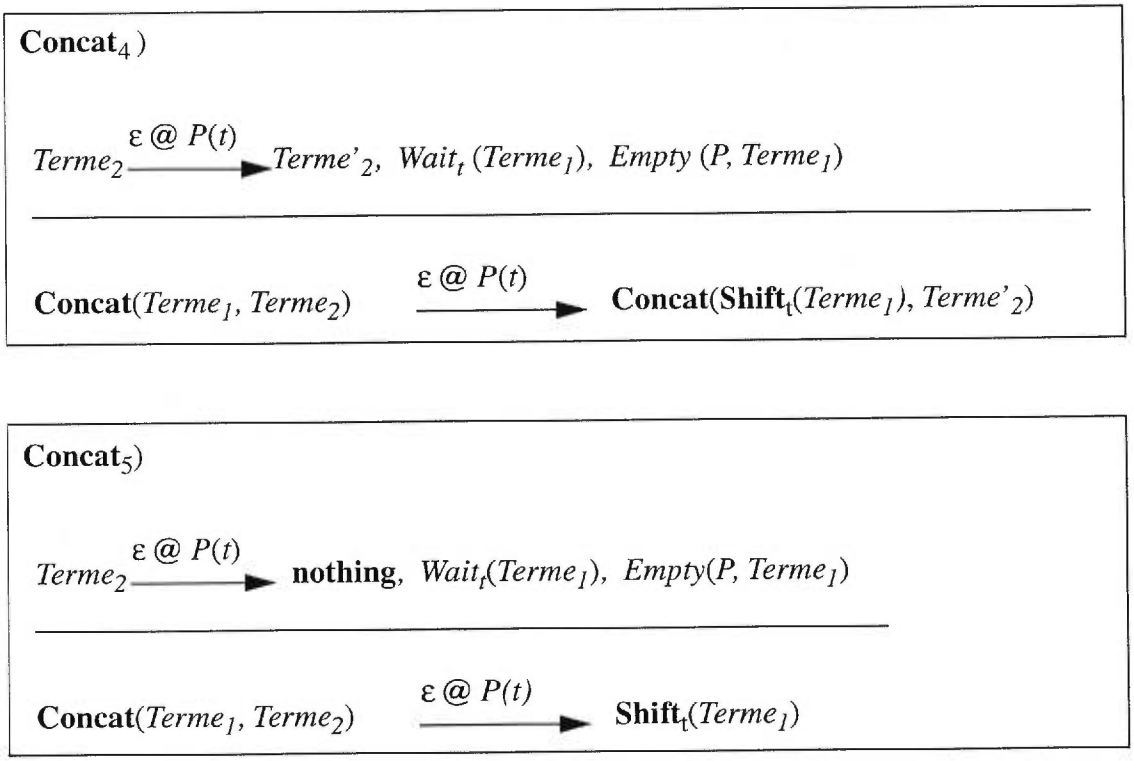


FIGURE 20: Règles de transition semblables du point de vue de l'exécution des diagrammes

En conclusion, les règles **Concat**$_5$, **Concat**$_6$, **Concat**$_7$, **RC**$_5$, **RC**$_6$ découlent des règles **Concat**$_4$, **Concat**$_3$, **Concat**$_2$, **RC**$_{1\text{-}2}$, **RC**$_4$, respectivement, et c'est ainsi que nous pouvons faire la différence entre: **Concat**$_4$ et **Concat**$_5$, **Concat**$_3$ et **Concat**$_6$, **Concat**$_2$ et **Concat**$_7$, **RC**$_{1\text{-}2}$ et **RC**$_5$, **RC**$_4$ et **RC**$_6$. Les règles de transition concernant l'opérateur d'exception **Excep**$_8$, **Excep**$_9$ dérivent de **Excep**$_5$, **Excep**$_4$, mais, dans ce cas particulier, d'autres termes composant le diagramme de l'exception sont désactivés en même temps que le terme qui a

effectué la dernière action *end* sur ses ports.

2) "**Activated**"

Le prédicat **Activated** de la sémantique opérationnelle prend, dans notre système de simulation, la forme d'un attribut booléen (*min_one_activ_port*) d'un diagramme feuille ou hiérarchique. Le booléen en question a la valeur "vrai" quand le diagramme est activé. Sa valeur, nécessaire si le diagramme hiérarchique est de type exception, est vérifiée dans la procédure *activate_possible_ports* (*root, candidats*), décrite à la section 4.4.

3) **Ports communicants "P $\in$ *P*"**

Un port P est nommé "communicant" si, à l'intérieur du diagramme feuille ou hiérarchique qui nous intéresse, il existe au moins un autre port P' pour lequel la relation P'.mapping = P.mapping est vraie (c.à.d., les deux ports ont la même correspondance parmi les ports de la racine). Donc, les ports communicants observent ou commandent l'activité de la même ressource physique (éventuellement un signal de VHDL).

4) "**Empty** (*P, Terme*)"

Si le diagramme désigné par *Terme* est un diagramme feuille, alors le prédicat **Empty** (*P, Terme*) est vrai dans un des deux cas suivants:

- soit le diagramme ne contient pas le port correspondant, c.à.d., un port dont l'attribut *mapping* est identique à P.mapping,
- soit ce port correspondant existe, mais il a déjà exécuté son action *end.*

Pour tout diagramme feuille la procédure *check_prec_sons_for_port* (*diagramme, port_to_check*) (voir la figure 21), appelée par *activate_possible_ports* (*root, candidats*) dans le processus, établit la valeur de ce prédicat représenté par l'attribut *active* de *port_to_check.* Si le diagramme représenté par *Terme* est un diagramme hiérarchique, le prédicat **Empty** (*P, Terme*) est vrai si en descendant vers tous les diagrammes feuilles appartenant au sous-arbre de *Terme*, pour tout diagramme feuille visité, le prédicat **Empty**

(*P, Terme*) est vrai.

```
PROCEDURE check_prec_sons_for_port
                              (had : INOUT ptr_had_node;
                              port_to_check : INOUT ptr_h_port) IS
-- had est un diagramme feuille
 VARIABLE ptr_port : ptr_h_port := had.ports_list;

BEGIN
 WHILE ptr_port /= null AND port_to_check.active LOOP
  IF ptr_port.mapping = port_to_check.mapping THEN
   port_to_check.active := false;
  END IF;
  ptr_port := ptr_port.brother;
 END LOOP;
END check_prec_sons_for_port;
```

FIGURE 21: La procédure *check_prec_sons_for_port*

5) "***Wait***$_t$ (*Terme*)", "**Shift**$_t$ (*Terme*)"

La simulation en VHDL est divisée en cycles de simulation (représentés par les délais ou cycles *delta* en VHDL). Une itération du processus du modèle VHDL se passe de la manière suivante:

- Le simulateur réalise l'avancement du temps jusqu'à ce qu'il y ait un événement sur un signal dans la liste de sensibilité du processus et alors le processus est réveillé. Ainsi, l'opérateur **Shift** $_t$ (*Terme*) de l'algèbre est relié à l'énoncé "Wait" du langage VHDL. Quand le programme rencontre l'énoncé "Wait", le processus est suspendu et il recommence son exécution quand la condition (ou la "clause") spécifiée devient vraie. Ainsi, si aucun changement de valeur ne s'est produit sur les signaux dans la liste de sensibilité du processus, mais le temps t s'est écoulé (**Shift** $_t$ (*Terme*)), ce qui se réalise en utilisant la clause "FOR x unités_de_temps" de l'énoncé VHDL "Wait", et si le délai fixé est expiré, le processus est réveillé. L'algorithme de simulation détermine ce délai qui représente le moment d'occurrence de la prochaine action à observer ou à générer, en fonction du graphe de contraintes temporelles de type *assume* pour les actions d'entrée

et de type *commit* pour les actions de sortie. Par conséquent, le simulateur ordonne les actions dans une *relation d'ordre total*, accomplissant de cette manière la sémantique du prédicat $Wait_t$ (*Terme*) de l'algèbre: pendant l'exécution d'une action dans un terme $Terme_1$, le terme $Terme_2$ se trouvant dans une composition hiérarchique avec $Terme_1$ est inactif ("attend") un temps $t \geq 0$.

6) **L'exécution des actions**

L'ordonnancement des actions à observer et / ou exécuter est réalisé par le simulateur et suit un ordre total dans le temps. Pour un moment de temps donné, lorsque le processus est réveillé, les actions sont exécutées suivant deux critères: leur type (d'abords les actions de sortie, par la suite les actions d'entrée) et, pour un type donné, selon l'ordre de parcours de la hiérarchie (toujours en post-ordre). L'appel des procédures qui concernent l'exécution des action est contenu dans le processus VHDL de la figure 12.

- Les **actions d'entrée** $a \in Act$ (en effet $a \in In$) sont exécutées au même cycle de simulation que leur observation.

  La procédure *catch_possible_actions* (*root, real_actions*) détecte toutes les actions d'entrée et de sortie auxquelles il est possible de faire correspondre une action réelle au moment courant de simulation. En ce moment, l'action est insérée dans la liste *real_actions*. De plus, certaines erreurs au niveau du diagramme feuille sont capturées. Prenons l'exemple d'une contrainte temporelle [5,5] de type *commit* entre la dernière action réelle sur un port et l'action *end* sur le même port. Supposons que toutes les actions spécifiées sur le port, à l'exception de l'action *end*, ont été exécutées. Si, dans l'intervalle temporel [0,5) après l'exécution de la dernière action, une action réelle apparaît sur le port, alors une erreur est signalée.

  L'exécution des actions d'entrée est effectuée dans la procédure *process_reals* (*real_actions, root*) montrée à la figure 22; leur moment d'occurrence y est enregistré. Si, par exemple, une erreur est détectée lors de l'exécution d'une action dans un diagramme participant à un choix retardé, le diagramme qui ne peut pas exécuter l'action est abandonné et le choix est ainsi décidé (le cas spécial de la règle de transition $\mathbf{DC}_2$

de la catégorie 6 dans la classification des règles de transition est détecté dans la procédure *update_had_node* (*had*), qui vérifie si une action ne peut pas être exécutée par un diagramme, voir la figure 22).

```
PROCEDURE process_reals
                        (real_actions : INOUT flat_port_array;
                        had : INOUT ptr_had_node) IS
   VARIABLE ptr : ptr_spec_action := real_actions;

BEGIN
   WHILE ptr /= null AND NOT ptr.had.error LOOP
---les actions de sortie ne sont pas prises en compte, car elles ont été traitées
---au cycle delta antérieur
     IF (NOT (ptr.direction = OUTPUT_D OR ptr.genre = END_ACTION) THEN
        ptr.normal_occured := true;
        ptr.occ_time := time_to_integer(now);
     ELSE
        ptr.had.error := true;
        update_had_node (had); -- propager l'erreur vers la racine
     END IF;
     ptr := ptr.link;
   END LOOP;
 END process_reals;
```

FIGURE 22: La procédure *process_reals*

- La procédure *catch_generated_actions* (*root, out_actions, end_actions*) détecte les **actions de sortie** $a \in Act$ (en effet, $a \in Out$) et les actions spéciales *end* dont l'instant de génération *trigger_time* (défini dans la section 4.1) est égal à l'instant courant de simulation. Leur moment d'occurrence est pris en compte dans le calcul du délai expiré *timeout*. Les actions de sortie sont récoltées dans la liste *out_actions*, tandis que les actions *end* sont insérées dans la liste *end_actions*. Ensuite, dans la procédure *process_output* (*out_actions, end_actions, root*), le moment d'occurrence des actions de sortie est enregistré, l'instant de génération (*trigger_time*) pour les successeurs de direction de sortie est calculé et l'intervalle d'occurrence des successeurs de direction d'entrée est mis à jour. Les actions réelles sur le port sont générées suite à l'appel à la procédure *generate_port_activity* (*out_actions*). Un cycle *delta* plus tard, l'action réelle est observée et jumelée à l'action correspondante spécifiée (dans la procédure

catch_possible_actions (*root, real_actions*)). Ainsi, l'action de sortie est exécutée. Nous sommes assurés que l'action ne sera pas traitée à nouveau car dans la procédure *process_reals* les actions de sortie ne sont pas prises en compte.

- Les actions spéciales **begin** $\in \mathcal{B}$ (de type *start_action* et de direction d'entrée) à exécuter sur un port donné au moment courant sont détectées dans la procédure *activate_possible_ports* (*root, candidats)* lors de l'initialisation et dans la procédure *start_ports* (exposée dans la figure 23) dans la boucle WHILE extérieure du processus pour tout autre moment. La description de la procédure *activate_possible_ports* (*root, candidats*) sera vue dans la section 4.4. Pour un port qui effectue son action *end*, la procédure *start_ports* détermine les ports correspondants qui exécutent leur action *begin* au même instant. Tous les ports ainsi trouvés sont réunis dans la liste *candidats*. L'exécution de toute action *begin* est effectuée dans la procédure *init_actions_on_port* (*candidats, root*). Cette procédure met à jour les divers attributs des ports *candidats* indiquant l'occurrence de l'action *begin* sur chaque *candidat*.

```
PROCEDURE start_ports (leaf_had : INOUT ptr_had_node;
                       port_to_check : INOUT ptr_h_port;
                       activated : INOUT boolean;
                       root : INOUT ptr_had_node;
                       candidats : INOUT ptr_h_port) IS
-- le booléen "activated" devient "vrai" quand un port de même "mapping" que port_to_check a été activé
-- et exécute son action "begin" dans un terme ayant un ancêtre "CONCAT_AD" commun avec leaf_had
-- établit les ports à activer si port_to_check a fini son exécution dans le diagramme feuille leaf_had
BEGIN
 port_to_check.active := false;
 port_to_check.current_action := null;
---parcourt le reste de la hiérarchie, appelle la procédure correspondante,
---en fonction du type du diagramme-père
 IF leaf_had.father /= null THEN
   CASE leaf_had.father.kind IS
    WHEN CONCAT_AD => check_concatenation (leaf_had, port_to_check, activated, root, candidats);
    WHEN CONCUR_AD => check_concur (leaf_had, port_to_check, activated, root, candidats);
    WHEN D_CHOICE_AD => check_choice (leaf_had, port_to_check, activated, root, candidats);
    WHEN HANDLER_AD => check_exception (leaf_had, port_to_check, activated, root, candidats);
    WHEN CONDITION_AD => check_exception (leaf_had, port_to_check, activated, root, candidats);
    WHEN NORMAL_AD => check_exception (leaf_had, port_to_check, activated, root, candidats);
    -- le cas "EXCEPTION_AD" et "RECX" n'ont pas de fils de type feuille,
    -- donc ils n'apparaissent pas dans cette discussion
    WHEN OTHERS => null;
   END CASE;
 END IF;
END start_ports;
```

FIGURE 23: La procédure *start_ports*

- Les actions spéciales **end** $\in \mathfrak{E}$ (de type *end_action* et de direction de sortie) correspondent à la terminaison correcte de l'exécution de toutes les actions spécifiées sur un port. Pour un instant donné, calculé par le simulateur, toutes les actions *end* sont récoltées dans la liste *end_actions* par la procédure *catch_generated_actions* (*root, out_actions, end_actions*), quand l'instant de génération *trigger_time* (le moment qui déclenche l'exécution de l'action *end* et des actions de sortie) est atteint. L'instant de génération de l'action *end,* calculé en fonction des contraintes *commit* dont l'action est le puits, est pris en compte dans le calcul du délai *time_out* (dans la procédure compute_time_out (root, time_out)). Donc, les actions *end* sur un port sont insérées dans une liste différente de la liste des actions de sortie pour éviter une deuxième suspension du simulateur comme dans le cas des actions de sortie, étant donné qu'une

nouvelle valeur sur le port n'est pas générée. L'exécution des actions *end* se réalise dans la procédure *process_output* (*out_actions, end_actions, root*), où le moment de leur occurrence est enregistré.

Prenons l'exemple d'une action *o* de sortie suivie sur le même port de l'action spéciale *end*. Entre les deux actions existe une contrainte *commit* [0,0], pour laquelle l'action *o* est la source et l'action *end* est la destination. Quand le moment *trigger_time* de l'action *o* est atteint, l'action est générée et le moment *trigger_time* de l'action *end* est calculé. Un cycle *delta* plus tard, l'action *o* est observée comme un changement de valeur sur le port. Le pointeur de l'action courante est déplacé de l'action *o* à l'action *end*. L'attribut *trigger_time* de l'action *end* est déterminé et pris en compte dans le calcul du délai *time_out*. Etant donné que l'action *end* est prévue pour le même instant de simulation que l'action *o* (à cause de la contrainte *commit* [0,0]), elle sera générée un cycle *delta* plus tard par un "WAIT FOR 0 ns".

7) **L'évolution du terme**

Suite à l'exécution de chaque action, l'état du diagramme ayant effectué l'action change (spécifié par " *Terme -> Terme'* " dans la sémantique opérationnelle). Le diagramme feuille qui a exécuté une action subit une succession d'opérations pour mettre à jour son état:

- L'instant d'occurrence de l'action est vérifié (il doit respecter les bornes de l'intervalle d'occurrence).
- La valeur spécifiée sur le port est comparée à la valeur actuelle et, dans le cas d'une inadvertance, une erreur est signalée et, éventuellement, selon la sémantique des opérateurs hiérarchiques ancêtres du diagramme feuille, elle est propagée vers la racine. Une erreur dans le terme feuille peut entraîner la désactivation du diagramme feuille et / ou celle de certains de ces ancêtres.
- Les bornes des intervalles d'occurrence de tous les successeurs de direction d'entrée de l'action exécutée sont mises à jour. Ces successeurs sont les actions d'entrée qui représentent la destination des contraintes *assume* qui émergent de l'action courante.
- Le moment de génération des successeurs de direction de sortie (destinations des con-

traintes *commit* émergeant de l'action courante) est calculé.

- Les prédicats et les procédures attachés aux actions sont évalués, respectivement appelées (les procédures *call_action_pred* (*real_actions, out_actions*) et *call_action_proc* (*real_actions, out_actions*)) et la valeur spécifiée est affectée au port dans le cas des actions de sortie (*generate_port_activity* (*out_actions*)).
- Le pointeur indiquant la prochaine action à exécuter sur le port avance vers l'action suivante spécifiée sur le port (la procédure *validate_port_activity* (*real_actions*)).

Les diagrammes feuilles étant modifiés suite à l'exécution d'actions, il va de soi que la hiérarchie au complet change d'état.

La section suivante présente la manière dans laquelle nous avons réalisé la correspondance entre les deux approches: la sémantique opérationnelle et le modèle VHDL avec l'environnement de simulation.

4.4 La correspondance entre les règles de transition de l'algèbre ACTC^P et l'exécution du modèle

Munis d'explications au sujet des diagrammes d'actions, du principe de leur simulation, de la sémantique opérationnelle de l'algèbre de processus ACTC^P et de la réalisation des opérations de base (primitives), nous sommes maintenant en mesure d'analyser la correspondance entre les règles de transition et leur implantation à l'intérieur de notre système.
Nous passerons en revue les étapes (les procédures) les plus significatives du processus représentant l'exécution du modèle VHDL (voir la figure 12).

4.4.1 L'initialisation de la hiérarchie selon les règles de transition de l'algèbre de processus ACTC^P

Tel qu'il a été mentionné lors de la description du processus de simulation des diagrammes

d'actions et de leur représentation, la hiérarchie est parcourue en post-ordre à partir de la racine, pour respecter l'ordre intrinsèque existant entre les termes d'une concaténation. Le but de la descente dans la hiérarchie est de jumeler des actions réelles observées sur les ports et les actions spécifiées sur les ports actifs des diagrammes feuilles.

Il est important à noter que le parcours de la hiérarchie occupe une grande partie dans la plupart des procédures, car, pour toute action effectuée, la sémantique opérationnelle demande d'analyser le contexte hiérarchique du diagramme feuille où l'action s'est produite.

Le parcours de l'arbre déclenché par la procédure *activate_possible_ports (root, candidats)* au début de l'exécution sert à activer et à effectuer des actions spéciales *begin* sur des ports n'ayant pas de contraintes sur leur utilisation. Dans ce sens, lors de l'initialisation, tous les ports des diagrammes feuilles exécutent leur action *begin* au même cycle de simulation, à l'exception de quelques situations bien précises.

La première situation qui interdit l'exécution de l'action *begin* sur un port est détectée de la manière suivante (voir la figure 24):

- **si** le port P sur lequel l'action *begin* doit être effectuée se trouve dans un diagramme feuille F_2 dont le père est une **Concat**, ce qui nous restreint à l'application des règles de transitions **Concat**$_1$ ou **Concat**$_4$, **et**
- la feuille F_2 n'est pas le premier fils de la concaténation **et**
- il y a un port P' avec l'attribut *mapping* identique à P.mapping, c.à.d., P et P' correspondent au même port de la racine. Cette situation, décrite dans la sémantique opérationnelle par le prédicat $\neg$ *Empty*(*P, Terme*$_1$), exclut l'application de la règle de transition **Concat**$_4$.
- **alors** il n'est pas possible d'appliquer **Concat**$_1$ ou **Concat**$_4$ et, donc, sur le port P l'action *begin* ne sera pas exécutée.

La condition $\neg$ *Empty*(*P, Terme*$_1$) qui interdit l'application de la règle de transition **Concat**$_4$ est vérifiée par la procédure *check_prec_sons_for_port* (*diagramme, port*) appelée par *activate_possible_ports* (*root, candidats*) (voir aussi la description du prédicat *Empty* dans la section 4.3).

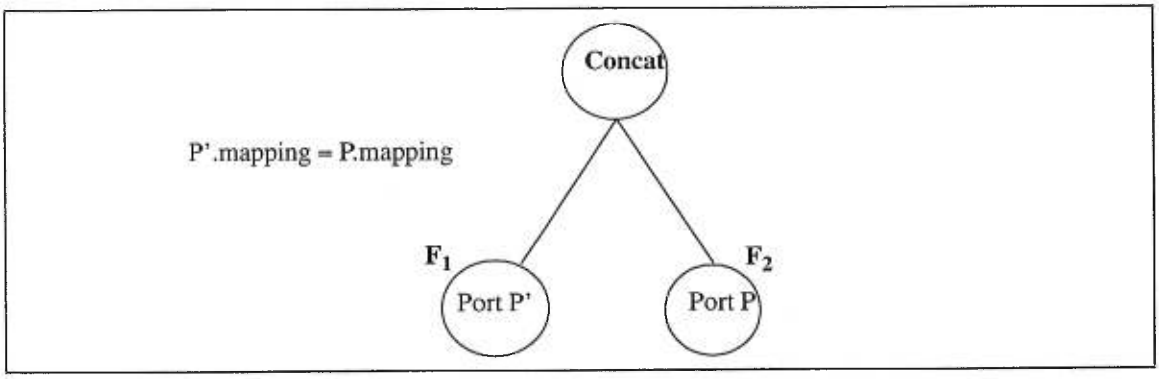


FIGURE 24: Situation empêchant l'exécution de l'action *begin* sur le port P de F2

De plus, il est nécessaire de vérifier **tout ancêtre** de la feuille F contenant le port P, car il est possible qu'une concaténation contraignante (de la manière décrite) existe seulement à un niveau strictement supérieur au père du diagramme feuille F (voir la figure 25).
Dans cet exemple, en considérant que P_1.mapping = P_2.mapping = P_3.mapping = P_4.mapping, il n'est pas possible d'exécuter l'action *begin* sur P_3 et P_4, étant donné que P_1 et P_2 n'ont pas fini leur exécution et que $\mathbf{RComp}_P$ ("en-dessous" duquel F_1 et F_2 se trouvent) est le premier terme de l'opérateur **Concat**.

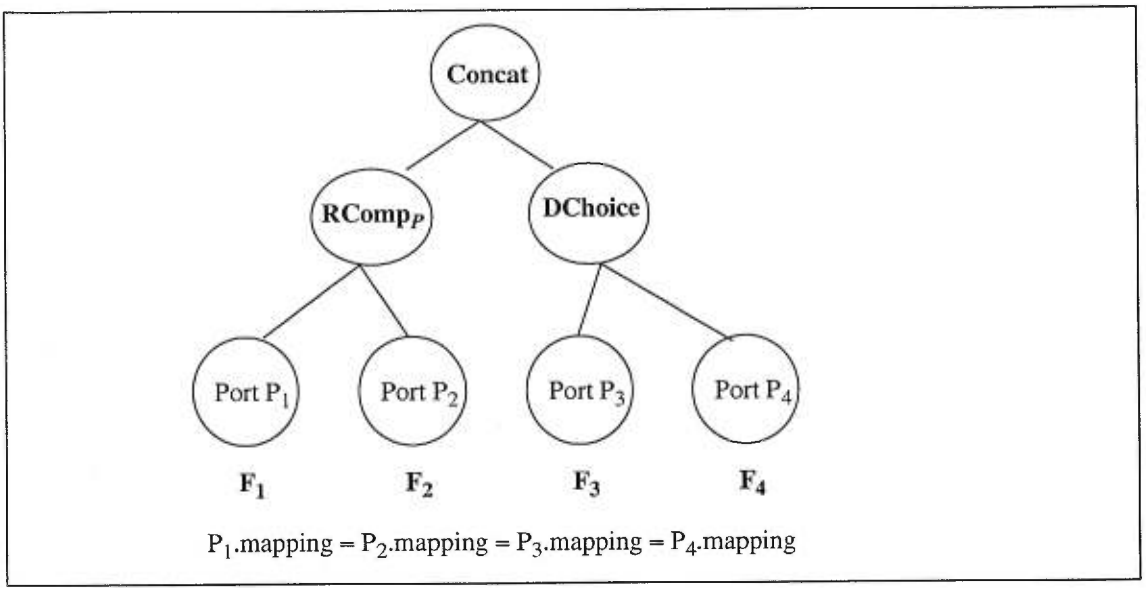


FIGURE 25: Situation empêchant l'exécution de l'action *begin* sur les ports P_3 et P_4

La deuxième situation qui interdit l'exécution d'une action *begin* sur un port a comme cause l'opérateur **Exception**: il n'est pas possible d'exécuter *begin* sur un port P appartenant à la condition d'exception T_{cond} tant que le comportement normal T n'ait pas effectué au minimum une action *begin* sur un de ses ports (voir *Activated*(T) dans la section 4.3). Si le prédicat *Activated*(T) est faux, la règle **Excep**$_2$ de la sémantique opérationnelle interdit l'exécution de l'action *begin* sur le port P. Lors de l'initialisation, tous les ports de la condition d'exception sont activés (et exécutent leurs actions *begin*) si le comportement normal est activé. L'activation de la condition d'exception se fait au même instant de simulation que l'activation du comportement normal.

La dernière situation rencontrée qui interdit l'exécution de l'action *begin* sur un port ressort aussi de l'utilisation de l'opérateur d'exception: il n'est pas possible d'exécuter *begin* si le port P se trouve dans un diagramme feuille appartenant au comportement d'exception T_{excep} (cette situation n'est décrite par aucune des règles de transition de l'algèbre ACTC^P et, donc, elle n'est pas une transition acceptée).

Nous soulignons que les deux dernières situations s'appliquent quand le père ou un ancêtre

du diagramme feuille est une condition d'exception T_{cond} ou un comportement d'exception T_{excep}.

La vérification remonte vers la racine de la hiérarchie: si une des situations décrites est rencontrée, le port P qui avait engendré la vérification est désactivé et l'action *begin* ne s'exécute pas. Ainsi, tous les ports pour lesquels le contexte hiérarchique le permet exécutent leur action *begin*, leur diagramme feuille sera mis à jour (voir la section 4.3) et la structure de la hiérarchie est inchangée (aucun diagramme n'est désactivé).

4.4.2 Les règles de transition menant à la désactivation de ports

Une fois l'initialisation de la hiérarchie et de ses ports effectuée et les premières actions *begin* exécutées, nous passons à l'examination de la boucle WHILE extérieure représentant, en effet, le corps du processus (voir la figure 12). A l'intérieur de cette boucle, des actions sont exécutées, des ports et des diagrammes sont désactivés et / ou activés, chaque contexte d'action étant reflété par une règle de transition dans l'algèbre de processus $ACTC^P$.

Une importante partie du processus est représentée par l'exécution des actions spéciales *end*, étant donné que l'enchaînement des diagrammes se fait port par port. Les règles de transition qui traitent de l'exécution des actions *end* sont les suivantes (voir la section 3.4, la catégorie 2):

$Concat_2$, $Concat_3$, $Concat_4$, $Concat_5$, $Concat_6$, $Concat_7$

RC_1, RC_2, RC_4, RC_5, RC_6

$Excep_4$, $Excep_5$, $Excep_8$, $Excep_9$

DC_3, DC_4

Les situations qui correspondent à la désactivation de diagrammes (catégorie 3 dans la classification: **$Concat_5$**, **$Concat_6$**, **$Concat_7$**, **RC_5**, **RC_6**, **$Excep_8$**, **$Excep_9$**) sont détectées par la procédure *find_finished_hads* (*ptr_end_action, finished_hads*), les diagrammes à désactiver sont réunis dans la liste *finished_hads* et désactivés dans la procédure *terminate_hads* (*finished_hads*) (voir "**nothing**" dans la section 4.3).

Lors de la première itération de la boucle WHILE extérieure, aucun port n'a pas encore exécuté son action *end*, alors la condition de la boucle WHILE intérieure n'est pas respectée (la liste *end_actions* est vide); le délai expiré *timeout* est déterminé et des actions sont exécutées dans les procédures *process_output*, *process_reals*. Pour chaque action *end*, la procédure *start_ports* détermine les ports qui peuvent être activés (peuvent exécuter leur action *begin*), en fonction de l'ensemble de règles de transition applicables. Les ports ainsi trouvés sont insérés dans la liste *candidats*. La procédure *find_finished_hads* (*ptr_end_action, finished_hads*) détermine les diagrammes qui finissent leur exécution au moment courant de simulation et les insère dans la liste *finished_hads*. L'exécution des actions *end* tient du contexte du diagramme feuille, mais tout *end* est suivi d'une vérification: si une action *end* est exécutée dans une feuille ayant un ancêtre de type concaténation, alors la hiérarchie doit être parcourue dans le but d'identifier les ports qui peuvent exécuter les actions *begin* au même instant et, ensuite, les exécuter.

Donc, selon le père du diagramme feuille contenant le port qui vient de finir son exécution, un des sous-ensembles suivants est choisi: {**Concat**$_2$, **Concat**$_3$, **Concat**$_4$, **Concat**$_5$, **Concat**$_6$, **Concat**$_7$}, {**RC**$_1$, **RC**$_2$, **RC**$_4$, **RC**$_5$, **RC**$_6$}, {**Excep**$_4$, **Excep**$_5$, **Excep**$_8$, **Excep**$_9$}, {**DC**$_3$, **DC**$_4$}.

Nous avons remarqué que toutes les règles applicables dans le cas où une action *end* sur un port est exécutée, à l'exception des règles traitant l'opérateur Exception, demandent l'examination de *l'état du diagramme-frère* de la feuille où *end* a été exécuté, c.à.d., les primitives **Ports communicants**, **Empty** (*P, Terme*), l'exécution des actions spéciales *end* et *begin*, l'évolution du terme *Terme -> Terme'* (voir la section 4.3).

L'occurrence d'une action *end* dans un diagramme feuille est la condition commune à toutes les hypothèses des règles de la catégorie 2 de la classification. Les autres conditions, spécifiques à chaque règle de transition, sont examinées par les procédures suivantes, selon le type de l'opérateur père du diagramme feuille où l'action *end* s'est produite:

- *check_choice* pour les règles {**DC**$_3$, **DC**$_4$} (voir la figure 27)
- *check_concur* pour les règles de transition {**RC**$_1$, **RC**$_2$, **RC**$_4$, **RC**$_5$, **RC**$_6$} (voir la figure 28)
- *check_concatenation* pour les règles {**Concat**$_2$, **Concat**$_3$, **Concat**$_4$, **Concat**$_5$, **Concat**$_6$,

Concat$_7$**}** (à la figure 34*)*

- *check_exception* pour **{Excep**$_4$**, Excep**$_5$**, Excep**$_8$**, Excep**$_9$**}** (à la figure 35*)*

Notons que le traitement de la récursion (l'opérateur **recX**) n'apparaît pas parmi les procédures énumérées, car le fils d'une récursion n'est **jamais** un diagramme feuille, mais un diagramme hiérarchique (et, inversement, le père d'un diagramme feuille ayant exécuté une action *end* sur un port ne peut jamais être une récursion).

Nous pouvons observer qu'il existe une similarité entre les procédures:

- *check_concur* (voir les règles de transition **RC**$_1$, **RC**$_2$, **RC**$_4$, **RC**$_5$, **RC**$_6$*)*
- *check_choice* (voir **DC**$_3$, **DC**$_4$*)*

qui analysent la situation suivant l'exécution d'une action *end* sur un *port* dans une *feuille.* En analysant les règles mentionnées, nous pouvons extraire une forme générale, montrée à la figure 26.

$$\frac{Terme_1 \xrightarrow{\varepsilon\ @\ P(t)} Terme'_1,\ Wait_t(Terme_2),\ Condition\_spécifique\_à\_l'opérateur}{Opérateur\ (Terme_1, Terme_2) \xrightarrow{action\_spécifique} Opérateur\ (Terme'_1, Shift_t(Terme_2))}$$

FIGURE 26: Exécution d'une action *end* dans un diagramme de type choix retardé ou composition avec rendez-vous

Dû à cette similarité, nous pouvons traiter de la même manière la situation où l'action *end* est exécutée dans un terme participant à une composition parallèle avec rendez-vous que la situation où l'action *end* est exécutée dans un terme participant à un choix retardé.

La *Condition_spécifique_à_l'opérateur* est l'appartenance ou la non-appartenance aux *ports communicants* dans le cas de l'opérateur de composition parallèle avec rendez-vous (**RComp**$_P$) et la valeur du prédicat *Empty* (*P, Terme*) pour l'opérateur de choix retardé (**DChoice**) (voir la section 4.3). Les deux conditions sont vérifiées de la même manière, en

utilisant la procédure *check_if_port_ended_in_leaf*, dû à une similarité entre les conditions mentionnées. Ainsi, la décision sur l'*action_spécifique* à exécuter par l'opérateur hiérarchique dépend de la *Condition_spécifique_à_l'opérateur* de la manière suivante:

- Il existe une *synchronisation* sur les actions *end* exécutées sur les *ports communicants* appartenant aux diagrammes-fils du $\mathbf{RComp}_P$. Dans ce cas, l'opérateur $\mathbf{RComp}_P$ exécute, du point de vue de l'algèbre, une action *end* sur le port communicant. Dans la simulation, la situation ci-décrite équivaut à la permission de chercher dans la hiérarchie des ports qui exécutent leurs actions *begin* au même instant (les règles de transition $\mathbf{RC}_4$ et $\mathbf{RC}_6$). Cette permission est exprimée par le booléen *ok* dans les procédures *check_concur*, *check_if_port_ended_in_leaf*.
- Si un port P termine dans un fils (T_i) du diagramme hiérarchique **DChoice** et s'il existe un port P' dans T_j (le frère de T_i), où P' a le même attribut *mapping* que P, alors, dans la simulation, la permission de chercher dans la hiérarchie des ports exécutant leur action *begin* (exprimée par le booléen *ok* dans la procédure *check_choice*) est décidée par la valeur du prédicat *Empty* (P, T_j): *ok* devient "vrai" si *Empty*(P, T_j) est vrai. En conclusion, la permission est donnée si tous les ports ayant un attribut *mapping* commun dans le sous-arbre du noeud **DChoice** ont fini leur exécution. Par contre, l'algèbre distingue les deux situations (c.à.d., permission accordée ou non) par l'*action_spécifique*: l'action ε est utilisée pour une permission accordée (règle $\mathbf{DC}_4$), par rapport à l'action silencieuse τ (invisible au niveaux supérieurs de la hiérarchie) dans la règle $\mathbf{DC}_3$ où la permission n'est pas accordée.

Dans ce contexte nous notons que:

- Les procédures *check_choice* ou *check_concur* sont appelées par la procédure *start_ports*, suite à l'exécution de l'action *end* sur le port *port_to_check*. Par l'intermède de ces procédures, nous obtenons les ports (*candidats*) qui exécutent leur action *begin* au même instant de la simulation.
- Les procédures *check_brother_choice_concur_concat*, *check_brother_exception*, *check_brother_recX* (les figures 29, 30, 31) descendent récursivement dans la hiérarchie, à partir du frère du diagramme où l'action *end* a été exécutée, vers les feuilles, où la procédure *check_if_port_ended_in_leaf* (la figure 33) ou

check_if_port_ended_in_leaf_exception (la figure 32) est appelée. Elles n'effectuent pas de traitement pour les noeuds de l'arbre (les diagrammes hiérarchiques), car les ports se trouvent seulement dans les diagrammes feuilles.

- Les procédures *check_if_port_ended_in_leaf* et *check_if_port_ended_in_leaf_exception* vérifient les conditions spécifiques et établissent la décision à prendre: activer ou non le port correspondant ailleurs dans la hiérarchie, dans le parcours habituel.

```
PROCEDURE check_choice (choice_son_had : INOUT ptr_had_node;
                        port_to_check : INOUT ptr_h_port;
                        activated : INOUT boolean;
                        root : INOUT ptr_had_node;
                        candidats : INOUT ptr_h_port) IS
 VARIABLE ok, rest_disabled : boolean := true;
 VARIABLE ptr_brother : ptr_had_node := choice_son_had.brother;

BEGIN
 IF choice_son_had.father.error THEN
  ok := false; -- nous ne poursuivons pas la recherche des ports à activer
 ELSE -- examiner le frère de choice_son_had et descendre vers les feuilles si nécessaire
  IF NOT ptr_brother.disabled THEN
   CASE ptr_brother.kind IS
     -- pour un diagramme hiérarchique descendre vers les feuilles
    WHEN CONCAT_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
    WHEN CONCUR_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
    WHEN D_CHOICE_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
    WHEN EXCEPTION_AD => check_brother_exception (ptr_brother, port_to_check, ok);
    WHEN RECX => check_brother_recX (ptr_brother, port_to_check, ok);
    WHEN LEAF_AD => check_if_port_ended_in_leaf (ptr_brother, port_to_check, ok);
    WHEN OTHERS => null;
   END CASE;
  END IF;
 END IF;
 IF ok THEN
  go_up_in_concat (choice_son_had, port_to_check, activated, candidats);
 END IF;
END check_choice;
```

FIGURE 27: La procédure *check_choice*

```
PROCEDURE check_concur (concur_son_had : INOUT ptr_had_node;
                        port_to_check : INOUT ptr_h_port;
                        activated : INOUT boolean;
                        root : INOUT ptr_had_node;
                        candidats : INOUT ptr_h_port) IS
 VARIABLE ok : boolean := true;
 VARIABLE ptr_brother : ptr_had_node := concur_son_had.brother;
BEGIN
   IF NOT ptr_brother.error AND NOT ptr_brother.disabled THEN
    CASE ptr_son.kind IS          -- pour un diagramme hiérarchique descendre vers les feuilles
     WHEN CONCAT_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
     WHEN CONCUR_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
     WHEN D_CHOICE_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
     WHEN EXCEPTION_AD => check_brother_exception (ptr_brother, port_to_check, ok);
     WHEN RECX => check_brother_recX (ptr_brother, port_to_check, ok);
     WHEN LEAF_AD => check_if_port_ended_in_leaf (ptr_brother, port_to_check, ok);
     WHEN OTHERS => null;
    END CASE;
   ELSE
     ok := false;
   END IF;
   IF ok THEN
     go_up_in_concat (concur_son_had, port_to_check, activated, candidats);
   END IF;
END check_concur;
```

FIGURE 28: La procédure *check_concur*

```
PROCEDURE check_brother_recX (rec_father : INOUT ptr_had_node;
                              port_to_check : INOUT ptr_h_port;
                              ok : INOUT boolean) IS
 VARIABLE rec_son : ptr_had_node := rec_father.son;
BEGIN
 IF NOT rec_son.error AND NOT ptr_brother.disabled THEN
  CASE rec_son.kind IS
    WHEN CONCAT_AD => check_brother_choice_concur_concat (rec_son, port_to_check, ok);
    WHEN CONCUR_AD => check_brother_choice_concur_concat (rec_son, port_to_check, ok);
    WHEN D_CHOICE_AD => check_brother_choice_concur_concat (rec_son, port_to_check, ok);
    WHEN OTHERS => null;      -- le fils d'une récursion ne peut pas être autre type que concaténation,
                              -- composition parallèle avec rendez-vous ou choix retardé
  END CASE;
 END IF;
END check_brother_recX;
```

FIGURE 29: La procédure *check_brother_recX*

```
PROCEDURE check_brother_choice_concur_concat (ptr_brother : INOUT ptr_had_node;
                                              port_to_check : INOUT ptr_h_port;
                                              ok : INOUT boolean) IS
        -- descendre récursivement vers les diagrammes feuilles
BEGIN
 IF NOT ptr_brother.error AND NOT ptr_brother.disabled THEN
  CASE ptr_brother.kind IS
    WHEN CONCAT_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
    WHEN CONCUR_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
    WHEN D_CHOICE_AD => check_brother_choice_concur_concat (ptr_brother, port_to_check, ok);
    WHEN EXCEPTION => check_brother_exception (ptr_brother, port_to_check, ok);
    WHEN RECX => check_brother_recX (ptr_brother, port_to_check, ok);
    WHEN LEAF => check_if_port_ended_in_leaf (ptr_brother, port_to_check, ok);
    WHEN OTHERS => null;
  END CASE;
 END IF;
END check_brother_choice_concur_concat;
```

FIGURE 30: La procédure *check_brother_choice_concur_concat*

```
PROCEDURE check_brother_exception (exception_father : INOUT ptr_had_node;
                                   port_to_check : INOUT ptr_h_port;
                                   ok : INOUT boolean) IS
BEGIN
 IF la condition d'exception et le comportement normal sont actifs THEN
   IF le fils de la condition d'exception est un diagramme feuille THEN
     check_if_port_ended_in_leaf_exception (fils de la condition d'exception, port_to_check, ok);
   ELSE          --descendre ves les feuilles
     IF le fils de la condition est une récursion THEN
       check_brother_recX (fils de la condition d'exception, port_to_check, ok);
     ELSE
       check_brother_choice_concur_concat (fils de la condition d'exception, port_to_check, ok);
     END IF;
   END IF;
   IF le fils du comportement normal est un diagramme feuille THEN
     check_if_port_ended_in_leaf_exception (fils du comportement normal, port_to_check, ok);
   ELSE          --descendre ves les feuilles
     IF le fils du comportement normal est une récursion THEN
       check_brother_recX (fils du comportement normal, port_to_check, ok);
     ELSE
       check_brother_choice_concur_concat (fils du comportement normal, port_to_check, ok);
     END IF;
   END IF;
 ELSE
   IF le comportement d'exception est actif THEN
     IF le fils du comportement d'exception est un diagramme feuille THEN
       check_if_port_ended_in_leaf_exception (fils du comportement d'exception, port_to_check, ok);
     ELSE       --descendre ves les feuilles
       ELSE
         IF  le fils du comportement d'exception est une récursion THEN
           check_brother_recX (fils du comportement d'exception, port_to_check, ok);
         ELSE
           check_brother_choice_concur_concat (fils du comportement d'exception, port_to_check, ok);
         END IF;
     END IF;
   END IF;
 END IF;
END check_brother_exception;
```

FIGURE 31: La procédure *check_brother_exception*

```
PROCEDURE check_if_port_ended_in_leaf_exception
                              (had : INOUT ptr_had_node;
                              port_to_check : INOUT ptr_h_port;
                              ok : INOUT boolean) IS
-- il suffit de trouver le port dans le diagramme, étant donné que dans la procédure appellante
-- check_brother_exception nous avons vérifié que l'exception est encore active
-- cette procédure correspond aux règles Excep4 et Excep5
 VARIABLE ptr_port : ptr_h_port := had.ports_list;
BEGIN
 WHILE ptr_port /= null AND ok LOOP
  IF ptr_port.mapping = port_to_check.mapping THEN
    ok := false;
  END IF;
  ptr_port := ptr_port.brother;
 END LOOP;
END check_if_port_ended_in_leaf_exception;
```

FIGURE 32: La procédure *check_if_port_ended_in_leaf_exception*

```
PROCEDURE check_if_port_ended_in_leaf
                        (had : INOUT ptr_had_node;
                        port_to_check : INOUT ptr_h_port;
                        ok : INOUT boolean) IS
 VARIABLE ptr_port : ptr_h_port := had.ports_list;
-- vérifier si le port port_to_check appartient au diagramme feuille had et si l'action "end" a été exécutée
BEGIN
 WHILE ptr_port /= null AND ok LOOP
  IF ptr_port.mapping = port_to_check.mapping AND
     ptr_port.current_action /= null THEN
                        -- ptr_port, de même attribut mapping que port_to_check n'a pas encore exécuté
                        -- son action "end"
      ok := false;
                        -- Dans le cas d'un choix retardé, dans cette condition nous sommes dans la
                        -- situation de la règle de transition DC3 et l'action silencieuse τ de l'algèbre
                        -- empêche la recherche d'un autre port à activer;
                        -- si la condition de l'énoncé "IF" est fausse (donc, dans l'algèbre, le prédicat
                        -- Empty est vrai), alors la règle DC4 s'applique (action ε de l'opérateur)
                        -- Pour la composition parallèle avec rendez-vous RCompP, pour permettre
                        -- l'activation d'autres ports, il est nécessaire que le booléen "ok" reste vrai, ce
                        -- qu'il arrive si soit "NOT ptr_port.mapping = port_to_check.mapping" (le cas
                        -- des règles RC1, RC2, RC5) ou "ptr_port.current_action /= null" est vrai
                        -- (les règles RC4, RC6)
  END IF;
  ptr_port := ptr_port.brother;
 END LOOP;
END check_if_port_ended_in_leaf;
```

FIGURE 33: La procédure *check_if_port_ended_in_leaf*

En ce qui concerne l'opérateur de concaténation, *check_concatenation* (voir la figure 34) est la procédure analogue à *check_choice* et *check_concur*, en utilisant l'ensemble de règles de transitions propres à cet opérateur, soit **Concat**$_2$, **Concat**$_3$, **Concat**$_4$, **Concat**$_5$, **Concat**$_6$, **Concat**$_7$.

La procédure *check_concatenation* prend en compte si l'action *end* s'est produite dans le premier ou dans le deuxième fils de l'opérateur, étant donné que l'ordre des termes appartenant à la concaténation joue un rôle important:

- si l'action *end* sur un port P est exécutée dans le premier fils de la concaténation, alors il faut chercher le port correspondant *candidat*, de même attribut *mapping* que P, qui exécute son action *begin* dans le deuxième fils de la concaténation.

- si l'action *end* s'est produite dans le deuxième fils de l'opérateur, il faut remonter dans la hiérarchie, au père du diagramme concaténation et reprendre le raisonnement selon le type d'opérateur rencontré.

Les procédures *go_up_in_concat* (*had, port_to_check, activated, candidats*) et *go_down_in_concat* (*had, port_to_check, activated, candidats*) (les figures 37 et 38) parcourent la hiérarchie en montant ou en descendant pour un diagramme *had* de type concaténation; dans les diagrammes feuilles visités, si le port correspondant existe (ce qui est vérifié par la procédure *check_if_port_in_leaf* de la figure 36), alors il exécute son action *begin*.

Les procédures *call_proc_for_choice* (*had, port_to_check, activated, candidats*), *call_proc_for_concur* (*had, port_to_check, activated, candidats*) et *call_proc_for_exception* (*had, port_to_check, activated, candidats*) descendent dans la hiérarchie vers les feuilles pour les opérateurs de choix retardé, composition parallèle avec rendez-vous et exception, respectivement. Nous présentons la procédure *call_proc_for_exception* (*had, port_to_check, activated, candidats*) à la figure 39, car dans le cas de l'exception il faut parcourir d'abord le comportement normal et ensuite la condition d'exception. Le comportement d'exception n'est visité que lorsque la condition d'exception a finit son exécution.

```
PROCEDURE check_concatenation (concat_son_had : INOUT ptr_had_node;
                               port_to_check : INOUT ptr_h_port; activated : INOUT boolean;
                               root : INOUT ptr_had_node; candidats : INOUT ptr_h_port) IS
 VARIABLE ptr_brother : ptr_had_node := concat_son_had.brother;
BEGIN
 activated := false;
 IF concat_son_had est le premier fils de la concaténation THEN
       -- le cas de règles Concat2, Concat3, Concat6, Concat7; pour les règles Concat6 et Concat7 la
       -- primitive "nothing" est traitée séparément dans la procédure end_diag
  IF NOT ptr_brother.error AND NOT ptr_brother.disabled and NOT activated THEN
   IF ptr_brother.kind = LEAF_AD THEN
                -- vérifier la valeur du prédicat Empty(P, Terme2) tel que demandé par les règles Concat3
                -- et Concat6 pour faire la différence par rapport aux règles Concat2 et Concat7
      check_if_port_in_leaf (ptr_brother, port_to_check, activated, candidats);
      IF NOT activated THEN    -- remonter vers le père et chercher un port à activer
         go_up_in_concat (ptr_brother, port_to_check, activated, candidats);
      END IF;
    ELSE
     CASE ptr_brother.kind IS    -- pour un frère de type hiérarchique, descendre vers ses feuilles
      WHEN CONCAT_AD => go_down_in_concat (ptr_brother, port_to_check, activated, candidats);
      WHEN CONCUR_AD => call_proc_for_concur (ptr_brother, port_to_check, activated, candidats);
      WHEN D_CHOICE_AD => call_proc_for_choice (ptr_brother, port_to_check, activated, candidats);
      WHEN EXCEPTION_AD => call_proc_for_exception (ptr_brother, port_to_check, activated,
                                                    candidats);
      WHEN RECX =>
         IF ptr_brother.son.kind = CONCAT_AD THEN
               go_down_in_concat (ptr_brother.son, port_to_check, activated, candidats);
         ELSE  IF ptr_brother.son.kind = D_CHOICE_AD THEN
               call_proc_for_choice (ptr_brother.son, port_to_check, activated, candidats)
               ELSE IF ptr_brother.son.kind = CONCUR_AD THEN
                        call_proc_for_concur (ptr_brother, port_to_check, activated, candidats);
                    END IF;
               END IF;
         END IF;
      WHEN OTHERS => null;
     END CASE;
     IF NOT activated THEN go_up_in_concat (ptr_brother, port_to_check, activated, candidats);
     END IF;
    END IF;
   END IF;
  ELSE   -- concat_son_had est le deuxième fils de la concaténation (le cas des règles Concat4, Concat5)
   IF NOT concat_son_had.error AND NOT concat_son_had.disabled THEN
     go_up_in_concat (concat_son_had, port_to_check, activated, candidats);
   END IF;
  END IF;
END check_concatenation;
```

FIGURE 34: La procédure *check_concatenation*

```
PROCEDURE check_exception (exception_son_had : INOUT ptr_had_node;
                    port_to_check : INOUT ptr_h_port; activated : INOUT boolean;
                    root : INOUT ptr_had_node; candidats : INOUT ptr_h_port) IS
-- exception_son_had est un diagramme feulle dont le père est un comportement normal ou une condition
-- d'exception ou un comportement d'exception
-- port_to_check a fini son exécution dans la feuille exception_son_had
BEGIN
 IF exception_son_had n'a pas d'erreur THEN
   IF le père de exception_son_had est un comportement normal THEN
     IF le comportement normal a fini complètement son exécution THEN
       go_up_in_concat (exception_son_had.father, port_to_check, activated, candidats);
   ELSE
     IF le père de exception_son_had est une condition d'exception THEN
       IF la condition d'exception a fini son exécution THEN
                descendre dans le comportement d'exception et vérifier si port_to_check se trouve là et,
                dans ce cas, l'activer et exécuter l'action begin;
       END IF;
     ELSE   -- le père de exception_son_had est un comportement d'exception
       go_up_in_concat (exception_son_had.father, port_to_check, activated, candidats);
     END IF;
   END IF;
 END IF;
END check_exception;
```

FIGURE 35: La procédure *check_exception*

```
PROCEDURE check_if_port_in_leaf
                        (had : INOUT ptr_had_node; port_to_check : INOUT ptr_h_port;
                        activated : INOUT boolean; candidats : INOUT ptr_h_port) IS
 VARIABLE ptr_port : ptr_h_port := had.ports_list;
BEGIN
 IF NOT had.error AND NOT had.disabled THEN
   WHILE ptr_port /= null LOOP
     IF ptr_port.mapping = port_to_check.mapping AND NOT ptr_port.active THEN
       activated := true;
       ptr_port.active := true;
       insérer ptr_port dans la liste candidats;
     END IF;
     ptr_port := ptr_port.brother;
   END LOOP;
 END IF;
END check_if_port_in_leaf;
```

FIGURE 36: La procédure *check_if_port_in_leaf*

```
PROCEDURE go_up_in_concat
                        (had : INOUT ptr_had_node; port_to_check : INOUT ptr_h_port;
                        activated : INOUT boolean; candidats : INOUT ptr_h_port) IS
 VARIABLE father : ptr_had_node := had.father;
 VARIABLE grandfather : ptr_had_node := had.father.father;
-- appellée quand aucun port dans la hiérarchie de had n'a pas été activé; la procédure remonte les niveaux
-- dans la hiérarchie un par un et cherche le(s) port(s) à activer et à exécuter son (leur) action begin
BEGIN
 IF grandfather /= null THEN
   CASE grandfather.kind IS
     WHEN CONCAT_AD => check_concatenation (father, port_to_check, activated, root, candidats);
     WHEN CONCUR_AD => check_concur (father, port_to_check, activated, root, candidats);
     WHEN D_CHOICE_AD => check_choice (father, port_to_check, activated, root, candidats);
     WHEN NORMAL_AD OR HANDLER_AD OR CONDITION_AD => check_exception
                                                (father, port_to_check, activated, root, candidats);
     WHEN OTHERS => null;
     -- le cas de la récursion n'apparaît pas car nous ne remontons jamais d'un terme récursif (nous
     -- descendons toujours vers les instances les plus récemment créées)
   END CASE;
 END IF;
END go_up_in_concat;
```

FIGURE 37: La procédure *go_up_in_concat*

```
PROCEDURE go_down_in_concat (had : INOUT ptr_had_node; port_to_check : INOUT ptr_h_port;
                             activated : INOUT boolean; candidats : INOUT ptr_h_port) IS
-- en partant de had, qui est une concaténation, descendre pour chercher dans le sous-arbre
 -- de had le port à activer, qui exécute son action begin au même moment de simulation
 VARIABLE current_son : ptr_had_node := had.son;
BEGIN
 WHILE current_son /= null AND NOT activated LOOP
  IF NOT current_son.error AND NOT current_son.disabled THEN
    IF current_son.kind = LEAF_AD THEN
      check_if_port_in_leaf (current_son, port_to_check, activated, candidats);
      IF NOT activated THEN      --si pas de port activé dans le premier fils de la concaténation,
                                 -- vérifions dans le deuxième fils
        current_son := current_son.brother;
      END IF;
    ELSE   -- descendre vers les diagrammes feuilles
     CASE current_son.kind IS
        WHEN CONCAT_AD => go_down_in_concat (current_son,port_to_check,activated,candidats);
        WHEN CONCUR_AD => call_proc_for_concur (current_son,port_to_check,activated,
                                                                        candidats);
        WHEN D_CHOICE_AD => call_proc_for_choice (current_son, port_to_check,activated,
                                                                        candidats);
        WHEN EXCEPTION_AD => call_proc_for_exception (current_son, port_to_check, activated,
                                                                        candidats);
        WHEN RECX => IF current_son.son.kind = CONCAT_AD THEN
                       go_down_in_concat (current_son.son, port_to_check, activated, candidats);
                     ELSE
                       IF current_son.son.kind = D_CHOICE_AD THEN
                        call_proc_for_choice (current_son.son, port_to_check, activated, candidats)
                       ELSE
                        IF current_son.son.kind = CONCUR_AD THEN
                          call_proc_for_concur (current_son, port_to_check, activated, candidats);
                        END IF;
                       END IF;
                     END IF;
        WHEN OTHERS => null;
     END CASE;
     IF NOT activated THEN
        current_son := current_son.brother;
     END IF;
    END IF;
  ELSE
    IF current_son.error THEN
       had.error := true;
       propager l'erreur vers la racine;
    END IF;
    current_son := null;
  END IF;
 END LOOP;
END go_down_in_concat;
```

FIGURE 38: La procédure *go_down_in_concat*

```
PROCEDURE call_proc_for_exception
                        (had : INOUT ptr_had_node;
                        port_to_check : INOUT ptr_h_port;
                        activated : INOUT boolean;
                        candidats : INOUT ptr_h_port) IS
 VARIABLE ok : boolean := false;
-- had est toujours un diagramme d'exception

BEGIN

 IF le fils du comportement n'a pas d'erreur et il n'est pas désactivé THEN
  IF le fils du comportement normal de had est un diagramme feuille THEN
       check_if_port_in_leaf (fils du comportement normal de had, port_to_check, activated, candidats);
                -- vérifier la condition d'exception de had:
       go_down_for_condition (had, port_to_check, activated, candidats);
  ELSE -- le fils du comportement normal de had est un diagramme hiérarchique
    descendre dans la hiérarchie vers les feuilles à partir du fils du comportement normal;
                -- vérifier la condition d'exception de had:
    go_down_for_condition (had, port_to_check, activated, candidats);
  END IF;
 END IF;

 IF le fils de la condition d'exception de had a fini son exécution THEN
   IF le fils du comportement d'exception de had n'a pas d'erreur et il n'est pas désactivé THEN
    IF le fils du comportement d'exception de had est un diagramme feuille THEN
      check_if_port_in_leaf (fils du comportement d'exception de had, port_to_check,
                                                                  activated, candidats);
    ELSE -- le fils du comportement d'exception de had est un diagramme hiérarchique
      descendre dans la hiérarchie vers les feuilles à partir du fils du comportement d'exception;
    END IF;
   END IF;
 END IF;

END call_proc_for_exception;
```

FIGURE 39: La procédure *call_proc_for_exception*

```
PROCEDURE go_down_for_condition
                        (had : INOUT ptr_had_node;
                        port_to_check : INOUT ptr_h_port;
                        activated : INOUT boolean;
                        candidats : INOUT ptr_h_port) IS
-- had est toujours un opérateur Excep
BEGIN
   IF activated THEN
      IF le fils de la condition d'exception de had est un diagramme feuille THEN
         IF le fils de la condition de had n'a pas d'erreur et n'est pas désactivé THEN
            check_if_port_in_leaf (fils de la condition d'exception de had, port_to_check,
                                                                    activated, candidats);
         END IF;
      ELSE              -- le fils de la condition de had est un diagramme hiérarchique
         descendre dans la hiérarchie vers les feuilles par l'intermède des procédures
         go_down_in_concat / call_proc_for_choice / call_proc_for_concur /
         call_proc_for_exception, en fonction du type d'opérateur que le fils de la condition de had
         représente
      END IF;
   ELSE              -- NOT activated
      IF le comportement normal de had a effectué au moins une action begin sur l'ensemble
         de ses ports THEN
         IF le fils de la condition d'exception de had est un diagramme feuille THEN
            IF le fils de la condition de had n'a pas d'erreur et n'est pas désactivé THEN
               check_if_port_in_leaf (fils de la condition d'exception de had, port_to_check,
                                                                    activated, candidats);
            END IF;
         ELSE              -- le fils de la condition de had est un diagramme hiérarchique
            descendre dans la hiérarchie vers les feuilles;
         END IF;
      END IF;
   END IF;
END go_down_for_condition;
```

FIGURE 40: La procédure *go_down_for_condition*

4.4.3 Les règles de transition menant à la désactivation de diagrammes

Dans le cas où une action end sur un port coïncide avec la fin de l'exécution d'un diagramme feuille et / ou hiérarchique (comme dans le cas des règles **Concat**$_5$, **Concat**$_6$, **Concat**$_7$, **RC**$_5$, **RC**$_6$, **Excep**$_8$, **Excep**$_9$ de la troisième catégorie de la classification), les diagrammes représentant les termes **nothing** sont désactivés.
La primitive **nothing** et sa réalisation (son implantation) ont été présentées dans la section 4.3.

4.4.4 Les règles de transition qui ne modifient pas la hiérarchie

L'exécution des actions d'entrée ou de sortie n'a pas d'effet sur la hiérarchie dans le cas des règles de transition **Concat**$_1$, **Concat**$_4$, **RC**$_1$, **RC**$_2$, **RC**$_3$, **Excep**$_1$, **Excep**$_6$, **Excep**$_7$, **DC**$_1$ (la catégorie 4 dans la classification des règles de transition).
Les primitives formant chaque règle mentionnée ont été détaillées à la section 4.3.

4.4.5 La récursion

La récursion se ressemble aussi bien à la première catégorie de la classification des règles de transition qu'à la deuxième catégorie étant donné que son effet sur la hiérarchie peut avoir lieu:

- suite à l'exécution de la première action *begin* sur l'ensemble des ports du terme récursif au premier moment de la simulation (lors de l'initialisation de la hiérarchie) ou
- plus tard, suite à des désactivations de ports décrites par les situations de type 2 de la classification

La récursion est l'opérateur hiérarchique nous aidant à modéliser des comportements infinis, tels que les boucles d'itération et les choix retardés à structure complexe.
Ainsi, les boucles (les concaténations infinies d'un terme avec lui-même) peuvent être

exprimées par **recX.** (**Concat** (*Terme, X*)) (voir la figure 41). Les choix retardés où il est difficile à faire le choix avant qu'une des alternatives termine complètement sont réalisés par **recX.** (**DChoice** (**Concat** ($Terme_1$, *X*), **Concat** ($Terme_2$, *X*)), où tous les ordres possibles d'exécution sont contenus dans cet arbre (voir la figure 41).

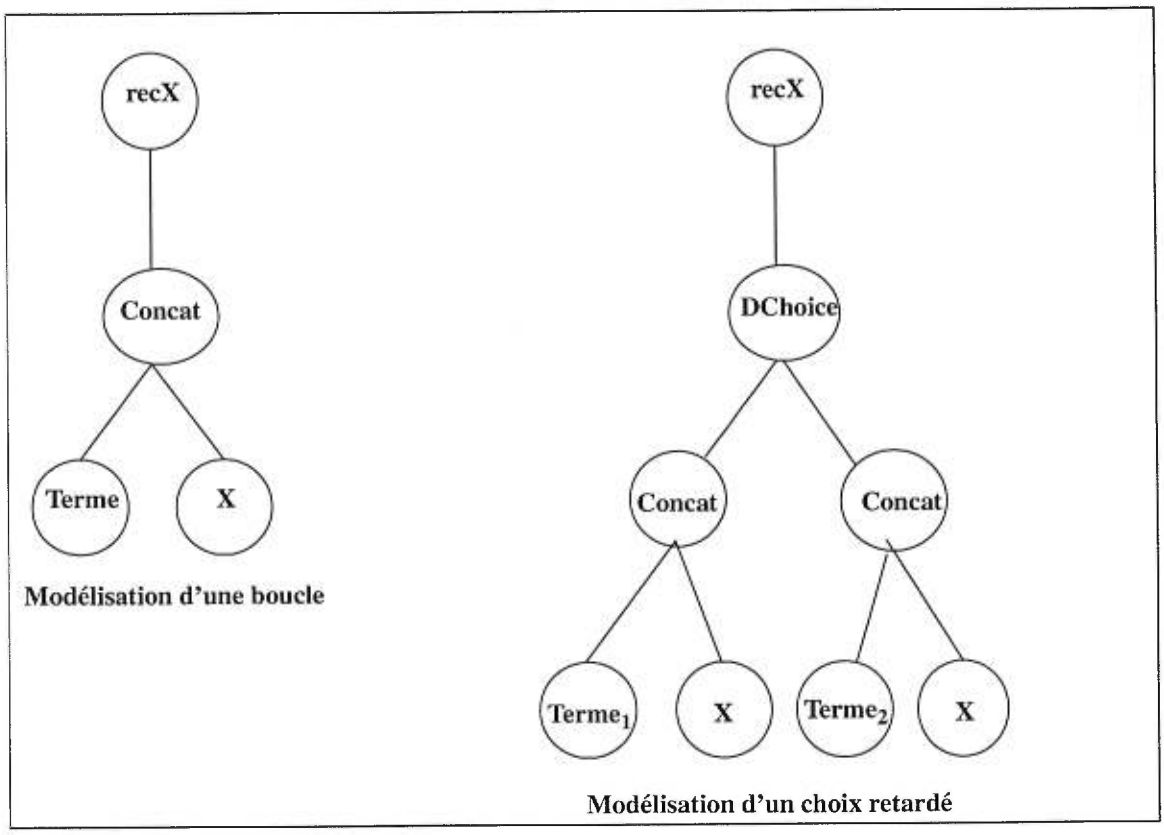


FIGURE 41: Modèle d'une boucle et d'un choix retardé à l'aide de l'opérateur **recX**

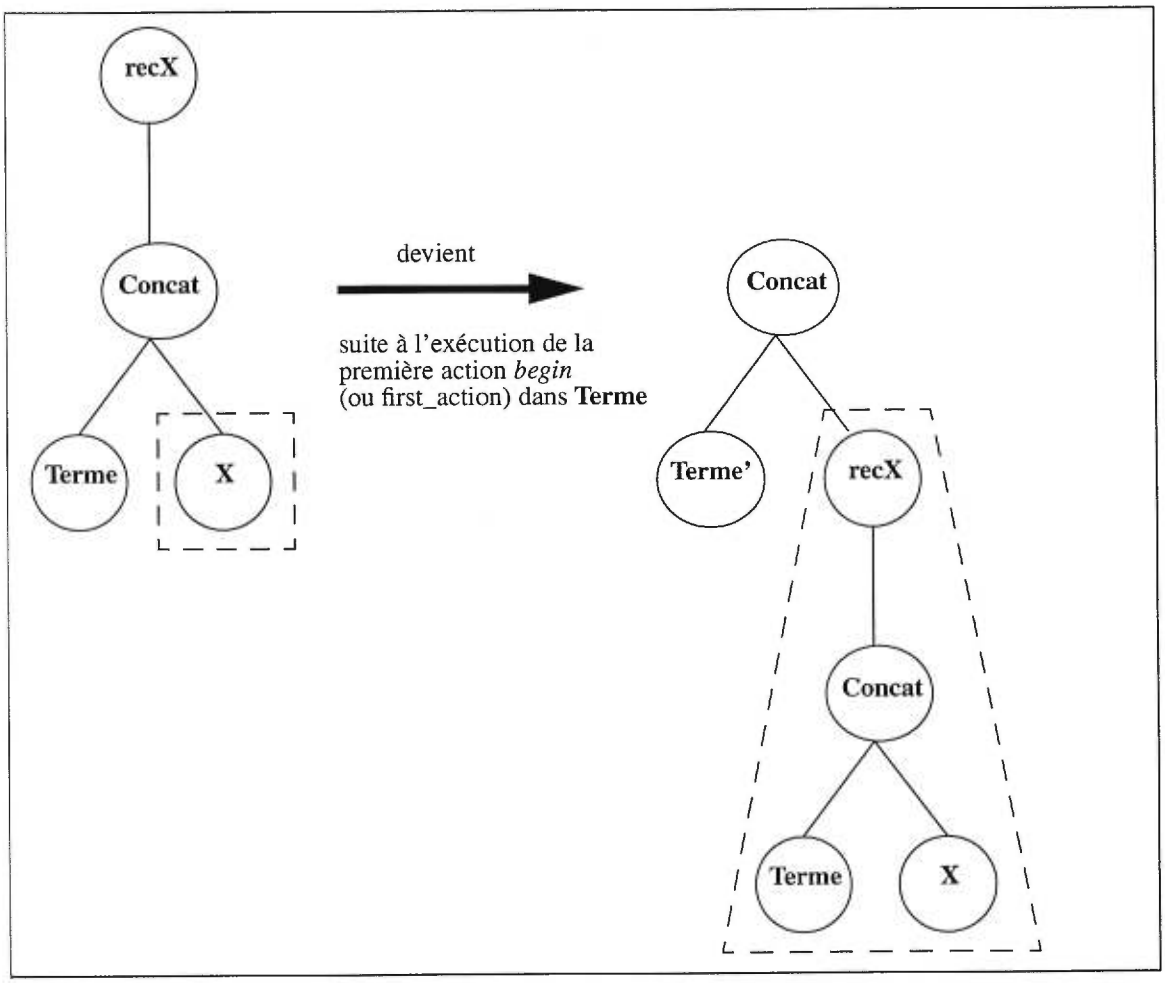


FIGURE 42: L'ajout d'une nouvelle instance du terme récursif

Prenons l'exemple de la boucle **recX**. (**Concat** (*Terme, X*)), que nous représentons à la figure 42. Aussitôt que **recX**. (**Concat** (*Terme, X*)) exécute la première action sur l'ensemble des ces ports (appelée *first_action*), la récursion, selon sa règle de transition, se transforme dans une concaténation entre le terme devenu *Terme*' et le résultat du remplacement de la variable libre *X* par **Concat** (*Terme, X*). Nous obtenons donc **Concat** (*Terme*', **recX**. (**Concat** *(Terme, X)*). Toutes les actions exécutées par le diagramme hiérarchique ainsi obtenu sont gouvernées par les règles de transition de la concaténation, jusqu'au premier *end* sur un port de *Terme*'. Etant donné que *Terme*' et **recX**. (**Concat** (*Terme, X*)) sont définis sur le même ensemble de ports, nous sommes assurés que la correspondance entre une action réelle et l'action spécifiée se fait correctement: l'action réelle est toujours jumelée à

l'action correspondante sur le port actif qui a exécuté son action *begin*, mais qui n'a pas encore exécuté son action *end*. Dans notre exemple, la correspondance se fait sur le port *P* dans *Terme'*, si *P* n'a pas encore effectué l'action *end* dans *Terme*.

Si plusieurs itérations sont intervenues déjà, la recherche du port descend dans l'arbre, à partir de la racine, pour chercher le port actif qui nous intéresse.

En revenant à *Terme'*, aussitôt qu'il exécute la première action *end* sur un de ses ports, la règle de transition de la récursion entre en jeu à nouveau, en remplaçant *X* par **Concat** (*Terme, X*).

Notre logiciel crée, en suivant la sémantique opérationnelle, une nouvelle instance du terme récursif avec lequel la variable libre *X* est remplacée à chaque exécution de *first_action* du terme **recX**. (**Concat** (*Terme, X*)).

A l'intérieur de toute récursion, il est nécessaire d'avoir une concaténation et, de plus, la variable libre *X* doit faire partie du deuxième terme de la concaténation. Ainsi nous n'acceptons pas:

- **recX**. (**Concat** (*X, Terme*)),
- **recX**. (**RComp**$_P$ (*Terme, X*)),
- **recX**. (**DChoice** (*Terme, X*))

Cette restriction s'impose car, dans les trois derniers cas nous ne pouvons pas démarrer des ports dans *X*: au début de la simulation (avant même la première action dans le terme de la récursion), *X* n'est pas un diagramme régulier, il ne possède pas de ports.

De manière similaire, dans le cas de la modélisation d'un choix retardé de la figure 41, aussitôt que **recX**. (**DChoice** (**Concat** (*Terme*$_1$, *X*), **Concat** (*Terme*$_2$, *X*)) exécute la première action sur l'ensemble de tous ses ports (donc soit dans *Terme*$_1$ ou dans *Terme*$_2$), les deux termes-feuilles X sont remplacés par toute la hiérarchie **recX**. (**DChoice** (**Concat** (*Terme*$_1$, *X*), **Concat** (*Terme*$_2$, *X*)).

Du point de vue de l'implantation, la procédure *activate_possible_diagrams* (*candidats*), appelée à partir du processus exposé à la figure 12, active les diagrammes feuilles qui contiennent les ports *candidats* et les diagrammes hiérarchiques ancêtres de ces diagrammes feuilles. Lors de l'activation d'un diagramme de type récursion **recX**, l'appel à la procédure *recX* (*had_to_start, port_to_check*) présentée à la figure 43 est effectué, en accord

avec la règle de transition de la récursion. Ainsi, les variables libres *X* sont remplacées par les termes récursifs.

```
PROCEDURE recX (had_to_start : INOUT ptr_had_node;
                port_to_check : INOUT ptr_h_port) IS
VARIABLE ptr_had : ptr_had_node := had_to_start;
-- la procédure est appelée lors de l'activation du terme de le récursion
-- had_to_start est le diagramme qui vient d'être activé

BEGIN
 WHILE NOT ptr_had.kind = recX AND ptr_had.father /= null LOOP
   ptr_had := ptr_had.father;
 END LOOP;
 IF ptr_had.kind = recX THEN
       -- remplacer la variable libre X par le terme sur lequel la récursion est définie:
   replaceX (ptr_had, port_to_check);
   remonter le fils de recX à la place de recX, en accord avec la règle de transition de la récursion;
 END IF;
END recX;
```

FIGURE 43: La procédure *recX*

La procédure *replaceX (had, port_to_check)* cherche les variables libres *X* à être remplacées par une nouvelle instance de la récursion, créée dynamiquement; *had* est un diagramme de type **recX.** Son sous-arbre représente la forme générale du terme récursif (voir figure 42). La procédure parcourt la hiérarchie (le sous-arbre de *had*) et trouve les diagrammes feuilles représentant les variables libres *X*. Chaque feuille ainsi trouvée est remplacée par l'arbre de la récursion.

Ce chapitre a présenté les structures de données et le principe de simulation des diagrammes d'actions feuilles et hiérarchiques. Toutefois, nous avons illustré la structure du processus VHDL d'une spécification HAAD. Nous avons également décrit, catégorie par catégorie, la manière dans laquelle nous avons implanté les règles de transition de l'algèbre $ACTC^P$ dans notre logiciel de génération de modèles exécutables pour la vérification d'interfaces matérielles par simulation. Les algorithmes d'interprétation et d'implantation des opérateurs hiérarchiques (concaténation, choix retardé, composition parallèle avec ren-

dez-vous, exception, récursion) ont été présentés. Nous avons détaillé les opérations effectuées lors de l'exécution d'une action dans un diagramme feuille et l'influence de l'occurrence de certains types d'actions sur la hiérarchie de diagrammes.

Nous montrons dans le chapitre suivant un exemple pratique pour décrire la méthodologie utilisée et le fonctionnement du système.

Chapitre 5. Exemple d'un modèle de *pipeline*

Nous présentons dans ce chapitre l'application de notre méthode et l'utilisation du logiciel pour la vérification du modèle d'un *pipeline* [34]. La composition séquentielle basée sur les ports (ou "concaténation"), telle que présentée à la section 2.2, permet de modéliser de façon simple et élégante un *pipeline* vu de ses interfaces. Un *pipeline* englobe deux aspects dans son comportement:

- Un aspect séquentiel: les résultats intermédiaires entre les étages du *pipeline* sont passés d'un étage au suivant. Chaque étage peut être vu comme une ressource effectuant une sous-opération quelconque. Une sous-opération doit être complétée afin de fournir le résultat correct à l'étage suivant du *pipeline* et, ainsi, obtenir le bon résultat final.
- Un aspect parallèle: plusieurs étages du *pipeline* peuvent fonctionner simultanément.

La manière la plus utilisée pour la représentation des *pipelines* est un système de coordonnées (temps, étage) (voir la figure 44 pour le cas d'un *pipeline* à 3 étages).

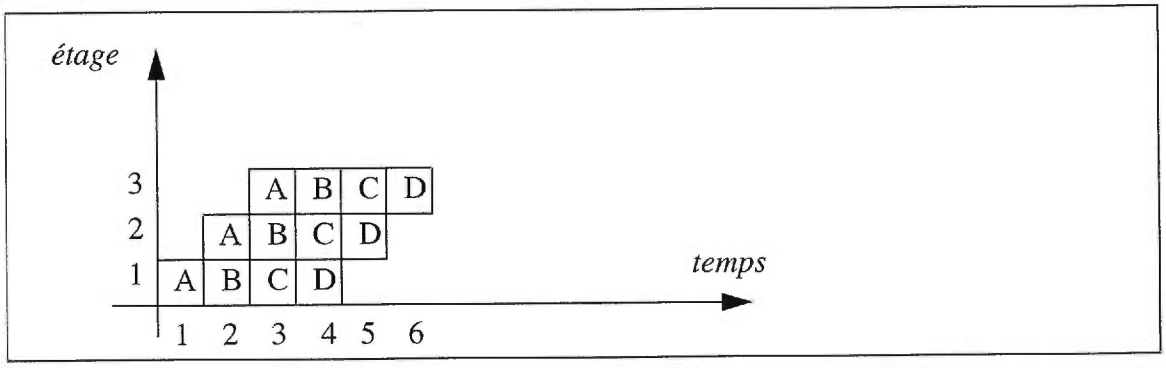


FIGURE 44: Diagramme d'occupation des étages d'un *pipeline*

La figure 44 illustre la représentation d'un *pipeline* sous la forme d'un diagramme d'occupation des étages. Les lettres dans les cases indiquent la tâche sur laquelle l'étage travaille à un moment donné.

Un exemple d'utilisation du *pipeline* est rencontré dans la conception du matériel pour l'additionneur en virgule flottante:

- Le premier étage effectue la soustraction des exposants et l'alignement des fractions.
- Le deuxième étage additionne les fractions.
- Le troisième étage normalise le résultat.
- Le quatrième étage effectue l'arrondissement du résultat.

Nous continuons la présentation par les descriptions graphique et en langage HAAD des diagrammes d'actions qui constituent le modèle de comportement d'un *pipeline*.

5.1 Les diagrammes feuille et hiérarchique du *pipeline*

Notre exemple consiste en un *pipeline* qui calcule la sortie y = F(x) comme une fonction F de l'entrée x. Notre méthodologie modélise de manière concise le comportement du *pipeline* sans devoir décrire la fonction de chaque étage.

La période d'insertion du *pipeline* (le temps écoulé entre deux entrées consécutives) est **T** unités de temps. Le *pipeline* a **n** étages et sa latence est **n*T** (le temps entre l'arrivée de l'entrée et la production du résultat final). Le modèle du comportement du *pipeline*, en considérant une seule entrée, peut être décrit par un diagramme feuille tel que montré à la figure 45. L'arrivée des données est signalée sur le port d'entrée X_PD1 et la production du résultat est observée sur le port de sortie Y_PD2. Le module VHDL appelé "DATA_PROVIDER" (voir l'annexe C) fournit les valeurs apparaissant successivement sur le port X_PD1. La première action sur ce port est actX, qui coïncide avec l'action *begin*. Entre l'action *begin* sur le port X_PD1 et l'action actX il existe une contrainte [0, 0] de type *assume*, indiquée à la figure 46. L'action actX est annotée par la variable *varX* et la procédure *load_input (varX, first, last, data_buffer)* (voir l'annexe D), qui affecte à *varX* la valeur du port X_PD1. Idéalement, les variables (telles que *varX*) devraient être créées dynamiquement pour chaque instance d'un diagramme feuille. Pourtant, pour simplifier la génération du modèle VHDL, nous générons un seul ensemble de variables, utilisées par toute instance créée dynamiquement. La gestion des variables se réalise par l'entremise d'un tableau circulaire *data_buffer* (voir l'annexe C) modélisant une queue FIFO. Le tableau possède deux pointeurs, "last" et "first", utilisés pour la gestion du tableau (par exemple, pour empiler et dépiler des valeurs du tableau). La longueur du tableau, choisie

par l'usager, doit être suffisante pour que toute donnée soit traitée.. Au moment de l'occurrence de actX, la valeur de la variable *varX*, attachée à l'action d'entrée, est insérée à la fin ("*last*") du tableau. Le port X_PD1 exécute son action *end* **T** unités de temps après l'occurrence de l'action actX (il existe une contrainte commit [**T**, **T**] entre actX et end_p1). Le port de sortie Y_PD2 possède trois actions: begin_p2, actY et end_p2 coïncidant avec actY. L'action actY est annotée par une variable *varY* et une procédure *compute_output (first, last, data_buffer, varY)* (voir l'annexe D). La valeur de *varY* est calculée par la procédure *compute_output* **n*T** unités de temps après l'occurrence de actX.

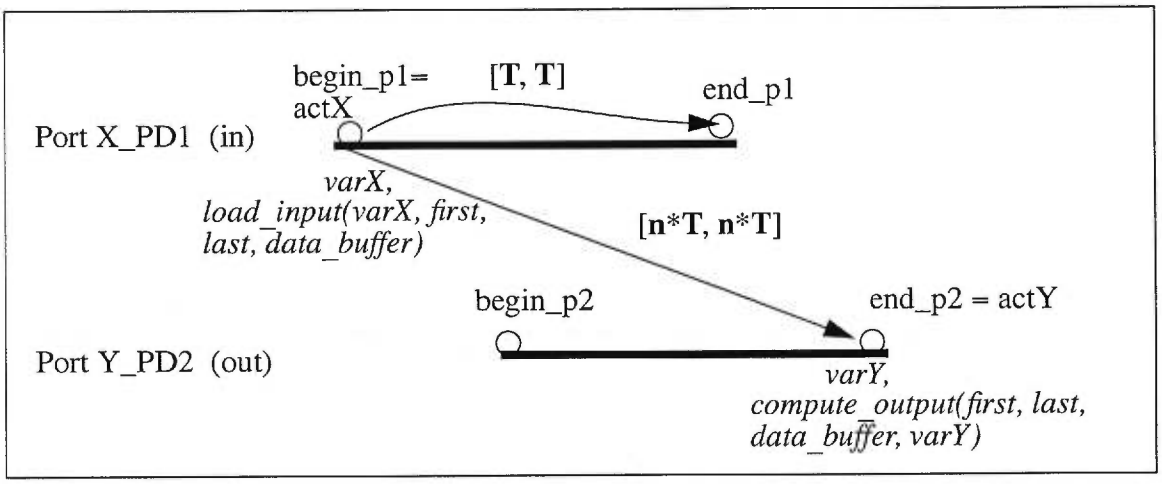


FIGURE 45: Le diagramme feuille du *pipeline*

La représentation en langage HAAD du diagramme feuille (où, pour les fins de l'illustration nous avons considéré la période d'insertion **T** = 5 et le nombre d'étages **n** = 3) se trouve à la figure 46 (voir la grammaire HAAD à l'annexe A). Sur les ports X_PD1 et Y_PD2 il n'y a pas nécessairement de changement de valeur, seulement des messages annonçant l'arrivée des entrées et l'obtention du résultat final (ainsi, à la figure 46, les ports sont déclarés comme "MESSAGE", voir [2]).

Pour obtenir une description complète du modèle du *pipeline* nous utilisons la récursion sur une concaténation entre le diagramme feuille Pipe_F et le terme récursif représenté par le diagramme feuille Empty_F (dont la représentation HAAD est donnée à la figure 47).

La représentation de la hiérarchie du modèle est montrée à la figure 48. Suite à l'exécution de la première action dans le sous-arbre de recX, le diagramme feuille Empty_F est remplacé par le terme récursif recX. Le fonctionnement de la récursion est tel que présenté à la section 4.2.5. A la figure 49 nous trouvons la représentation HAAD du diagramme hiérarchique du *pipeline*.

Ainsi, notre méthode permet la modélisation concise des comportements de type *pipeline* sans décrire la fonction de chaque étage.

```
(DEFBEHAVIOR Pipe_F
  (PORTS
    (PORT X_PD1 IN "inout_bus" MESSAGE)
    (PORT Y_PD2 OUT "inout_bus" MESSAGE))

  (VAR varX "inout_bus" "\"00000000\"")
  (VAR varY "inout_bus" "\"00000000\"")
  (VAR last "integer" "0")
  (VAR first "integer" "0")
  (VAR data_buffer "data_table")


  (LEAF
    (CARRIER-SPEC X_PD1
      (INITIAL-SPEC (VALID))
      (ACTION-START 'begin_p1)
      (ACTION-SPEC 'actX (VALID varX) (PROCEDURE-CALL "load_input" varX first last data_buffer)
)
      (ACTION-END 'end_p1))
    (CARRIER-SPEC Y_PD2
      (INITIAL-SPEC (VALID))
      (ACTION-START 'begin_p2)
       (ACTION-SPEC 'actY (VALID varY) (PROCEDURE-CALL "compute_output" first last data_buffer
varY))
      (ACTION-END 'end_p2))

  (PRECEDENCE 'begin_p1 'actX (CMIN 0) (CMAX 0) (INTENT ASSUME))
  (PRECEDENCE 'actX 'end_p1 (CMIN 5) (CMAX 5) (INTENT COMMIT))
  (PRECEDENCE 'actX 'actY (CMIN 15) (CMAX 15) (INTENT COMMIT))

  )
)
```

FIGURE 46: La représentation HAAD du diagramme feuille du *pipeline*

```
(DEFBEHAVIOR Empty_F
  (PORTS
    (PORT EMPTY_PORT IN "std_logic" MESSAGE))
  (LEAF
    (CARRIER-SPEC EMPTY_PORT
      (INITIAL-SPEC (CONSTANT '"0"') (DIRECTION IN))
      (ACTION-START 'begin_empty_port)
      (ACTION-END 'end_empty_port))
  )
)
```

FIGURE 47: La représentation HAAD du diagramme feuille Empty_F

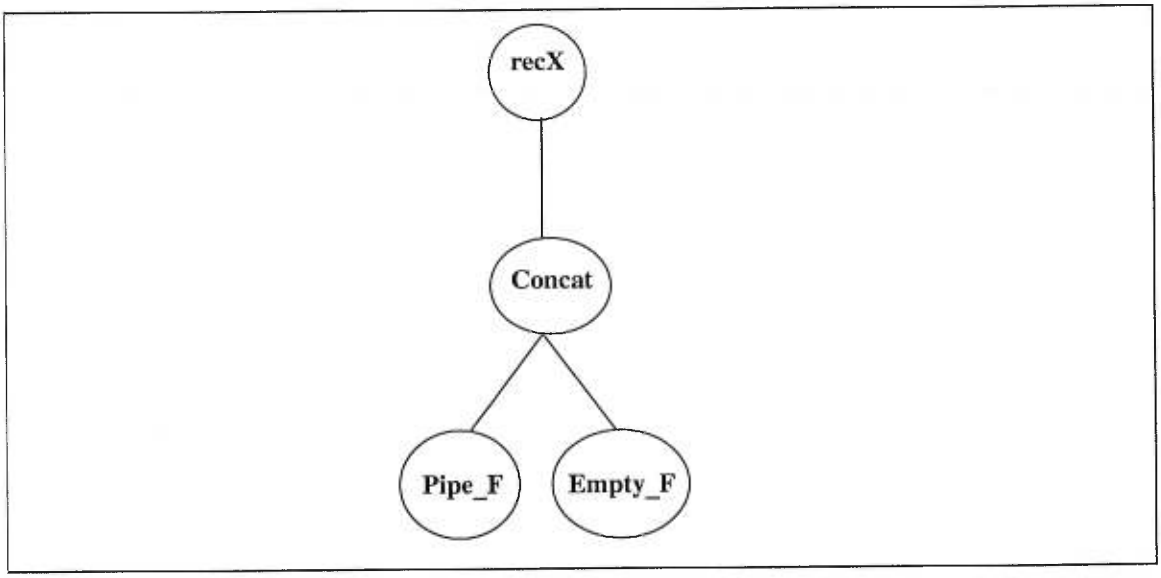


FIGURE 48: La représentation hiérarchique du *pipeline*

```
(DEFBEHAVIOR PIPE_REC
 (PORTS
   (PORT  PIPE_PD1 IN "inout_bus" MESSAGE)
   (PORT  PIPE_PD2 INOUT "inout_bus" MESSAGE)
   (PORT  EMPTY_PD3 IN "std_logic" MESSAGE))
 (RECX
  (CONCATENATION
    (BEHAVIOR a_Pipe_F Pipe_F
      (PORT-MAP PIPE_PD1 PIPE_PD2))
    (BEHAVIOR a_Empty_F Empty_F
      (PORT-MAP EMPTY_PD3))
  )
 )
)
```

FIGURE 49: La représentation HAAD du diagramme hiérarchique du *pipeline*

5.2 L'exécution du modèle du *pipeline*

Le modèle exécutable, généré en VHDL par le logiciel (voir [5]), peut être maintenant utilisé pour simuler le comportement du *pipeline*. Nous montrons à la figure 50 un schéma de l'occurrence des actions dans l'enchaînement des diagrammes feuilles du terme hiérarchique du *pipeline* dans l'intervalle temporel [0, 20]. L'annexe E présente un fragment de la trace d'exécution du modèle du pipeline, telle qu'elle a été générée par le logiciel. Les figures 51-54 montrent l'évolution de la hiérarchie (le dépliage de la récursion) suite à l'occurrence des actions sur les ports du modèle dans l'intervalle [0, 15]. Nous pouvons identifier sur les figures 50-54 et les actions qui sont exécutées sur les ports du modèle du *pipeline*. A partir de la figure 51 nous remarquons l'augmentation de la hiérarchie par l'ajout de nouvelles instances, étant donné que le noeud Empty_F est remplacé par la structure hiérarchique **recX** (**Concat** (Pipe_F, Empty_F)), situation illustrée par les sous-arbres entourés d'une ligne interrompue dans les figures mentionnées. Les changements de la hiérarchie remarquées dans les figures 51-54 sont générés suite à l'occurrence de la première action dans l'ensemble des ports (**begin_p$_1$** dans notre cas), de la même manière

qu'il a été décrite au chapitre 4 (la figure 41).

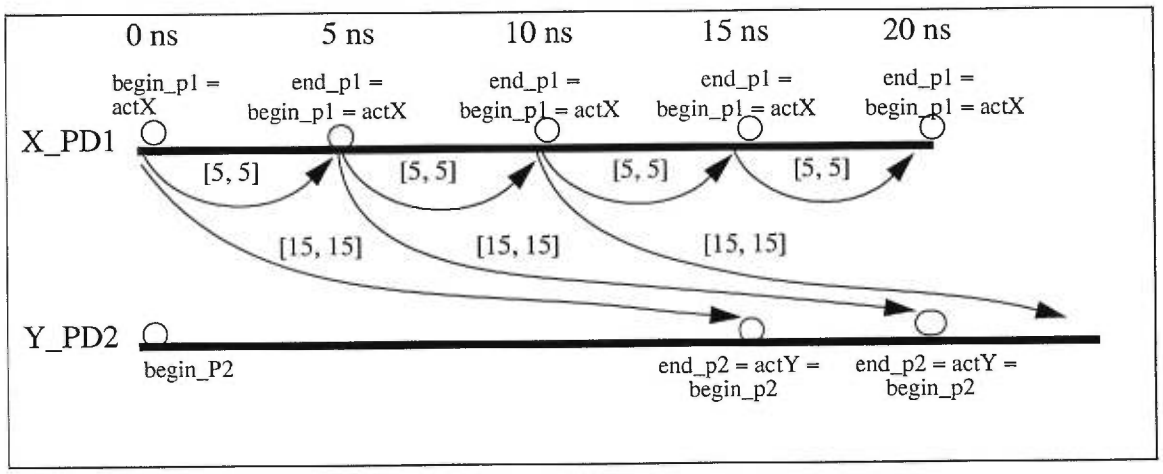


FIGURE 50: L'occurrence des actions dans l'intervalle [0, 20]

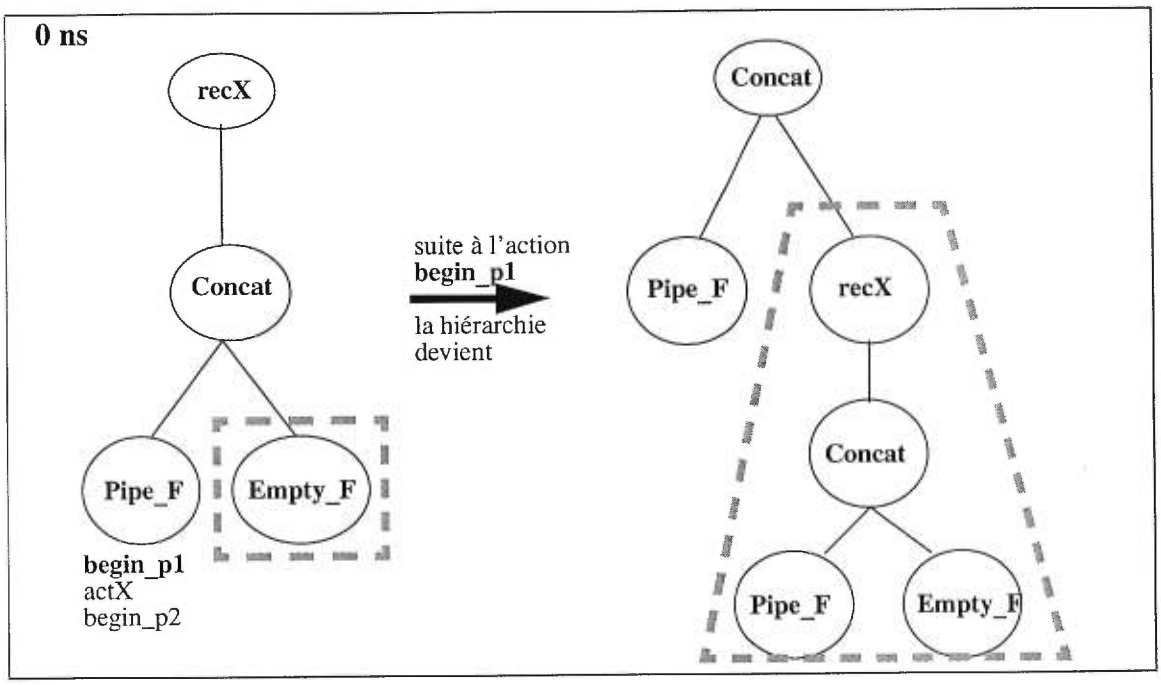


FIGURE 51: Structure de la hiérarchie au moment de simulation 0 ns

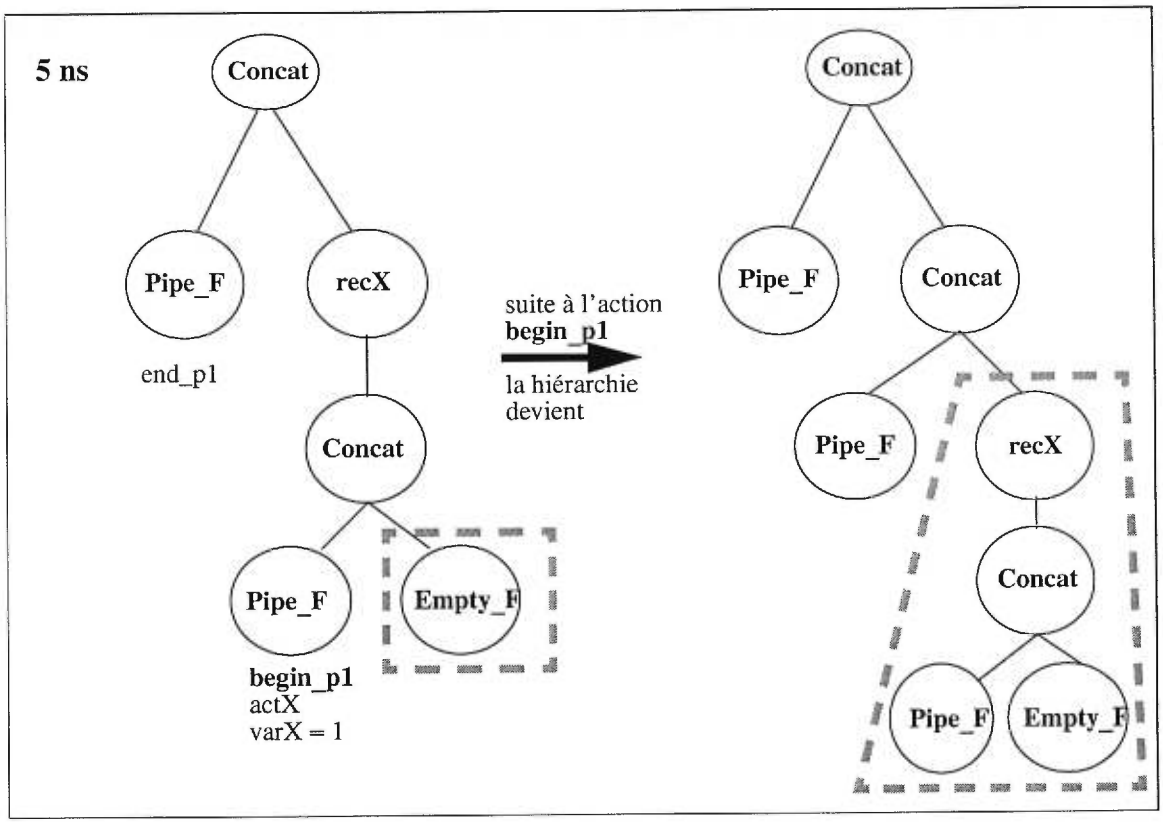


FIGURE 52: Structure de la hiérarchie au moment de simulation 5 ns

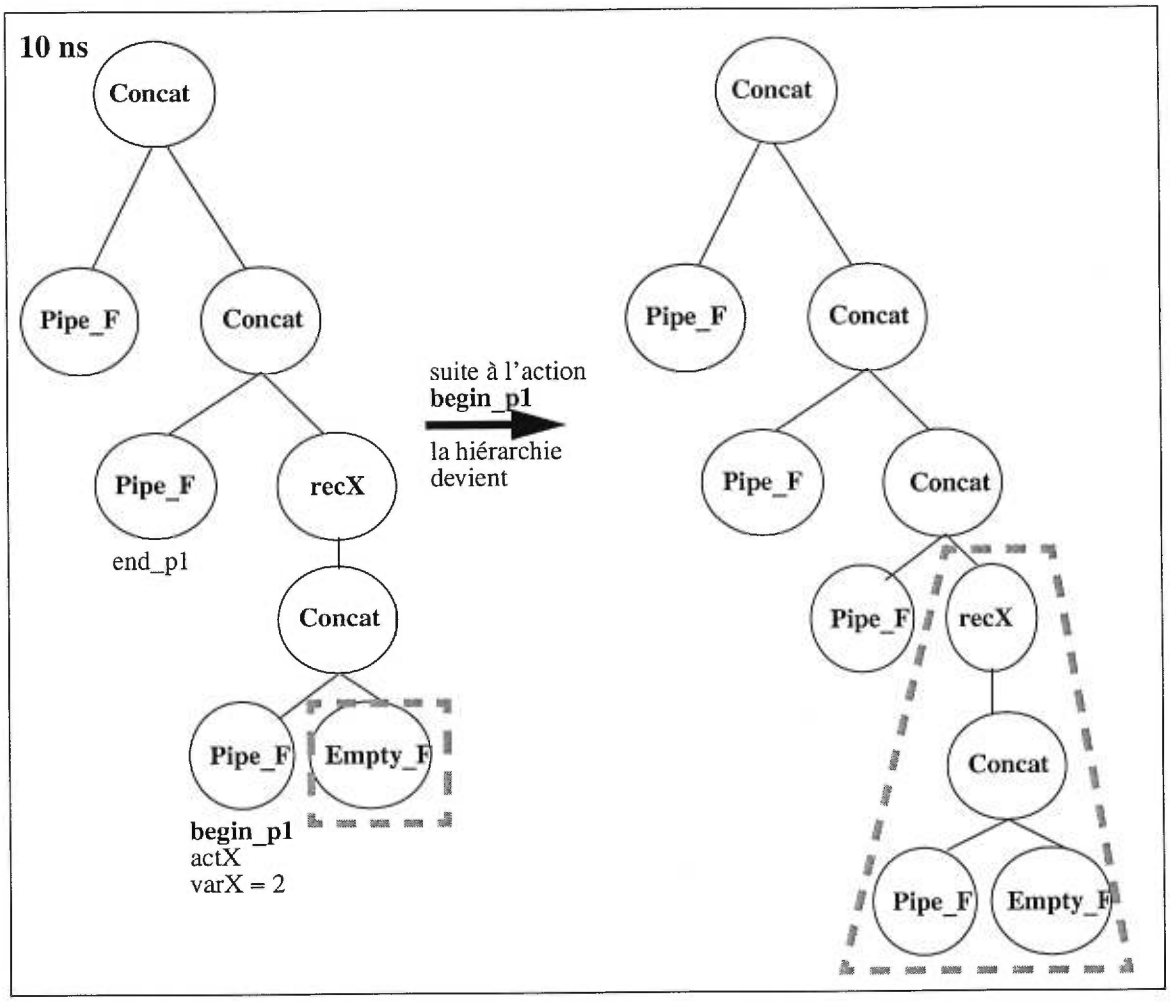


FIGURE 53: Structure de la hiérarchie au moment de simulation 10 ns

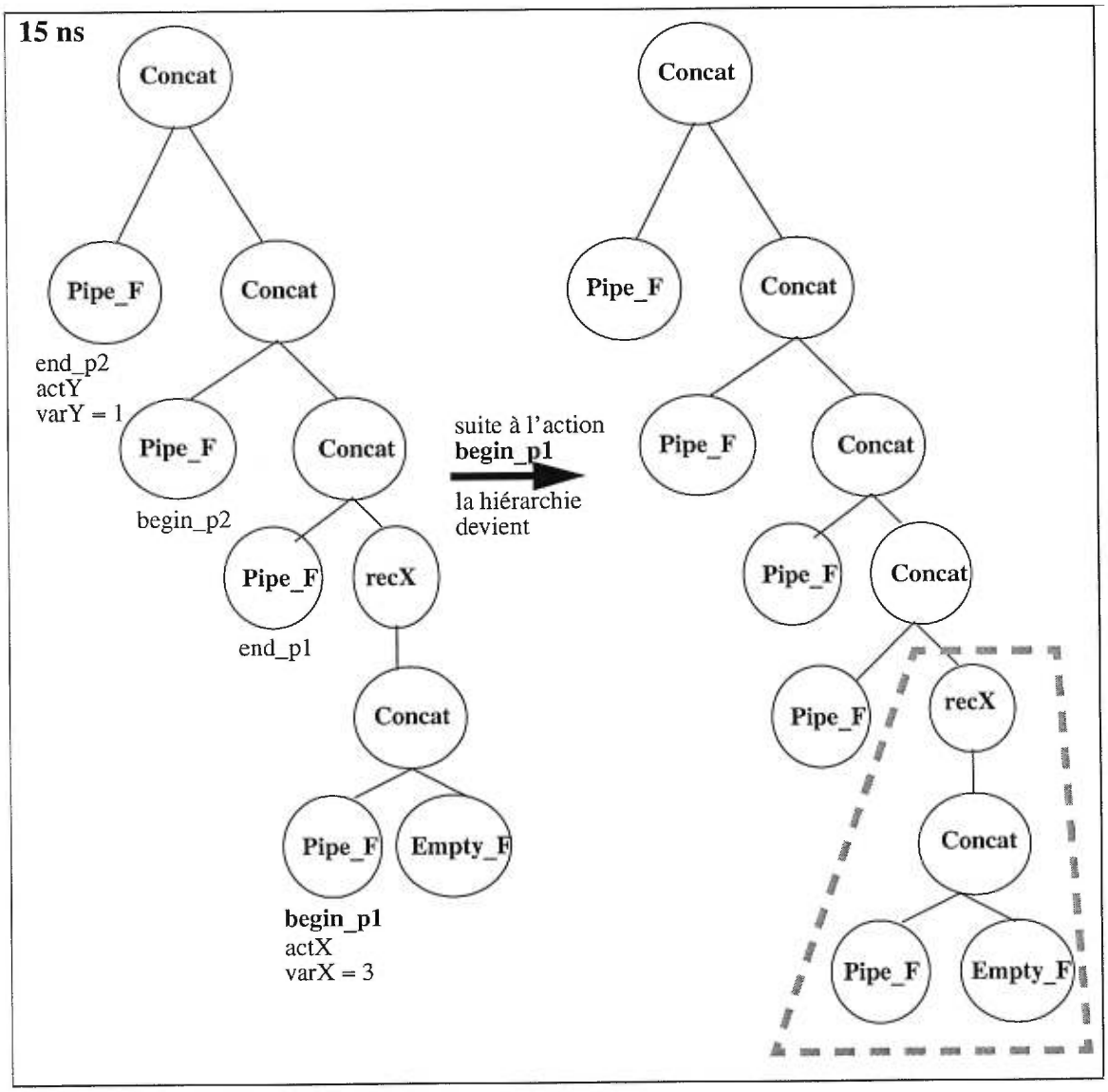


FIGURE 54: Structure de la hiérarchie au moment de simulation 15 ns

Chapitre 6. Conclusions

Le domaine de la vérification des interfaces des systèmes à temps réel est en développement continu. La vie quotidienne exige un échange d'informations de plus en plus rapide. Le comportement des systèmes à temps réel doit se soumettre à des contraintes temporelles strictes. Ainsi, la vérification d'interfaces entre les systèmes et leur environnement ou entre les divers composants des systèmes s'impose dans la conception des systèmes à temps réel.

Nous avons présenté des algorithmes de simulation des interfaces matérielles et l'outil de vérification se basant sur ces algorithmes. Une approche moins flexible quant aux spécifications des diagrammes d'actions feuilles et hiérarchiques était déjà construite dans notre laboratoire de recherche. Pourtant, dans certains cas, son utilisation demandait une quantité supplémentaire de travail de la part de l'usager pour rendre les spécifications utilisables. Par exemple, dans le cas des systèmes à *pipeline*, la gestion des résultats intermédiaires est plus ardue en comparaison avec la nouvelle approche que nous avons réalisée.

Ainsi, en spécifiant des actions spéciales *begin* et *end* comme le début, respectivement la fin de l'exécution des ports des diagrammes d'actions feuilles, nous permettons une composition de concaténation ayant deux aspects simultanés: un aspect parallèle et un aspect séquentiel. Les ports des diagrammes feuille sont vus comme des ressources du système auxquelles plusieurs diagrammes d'actions peuvent avoir accès en même temps. La composition parallèle avec rendez-vous permet la communication entre les ports partagés par plusieurs diagrammes. L'opérateur de récursion permet la création de structures (diagrammes) hiérarchiques complexes. D'autres opérateurs (tels que le choix retardé et l'exception) sont aussi disponibles à l'usager.

La sémantique opérationnelle de tous les opérateurs, telle que décrite par les règles de transition de l'algèbre de processus $ACTC^P$, se reflète dans l'implantation de notre système de simulation. Nous avons décrit la correspondance entre les primitives (les opérations de base) de l'algèbre $ACTC^P$, les règles de transitions et l'exécution du modèle. Nous avons montré la méthodologie de modélisation sur un exemple qui, tout en étant assez

compréhensible, peut bien illustrer notre approche et ses traits caractéristiques.

Dans le cadre de ce projet de recherche, notre apport se définit par:

- La participation à la définition des opérateurs hiérarchiques pour la composition des diagrammes d'actions (composition séquentielle ou concaténation, composition parallèle avec rendez-vous, le choix retardé, l'exception, la récursion),
- La conception des algorithmes qui interprètent et qui implantent les primitives et la sémantique formelle de l'algèbre $ACTC^P$ et la mise en évidence de l'équivalence entre la sémantique formelle et l'exécution du modèle VHDL,
- L'implantation de l'opérateur d'exception et la généralisation de l'opérateur de récursion afin de modéliser des comportements complexes,
- Un exemple pratique illustrant la méthode de modélisation.

En ce qui concerne les futurs développements de notre système de modélisation et de génération de modèles VHDL, il nous semble approprié de suggérer la création d'un générateur automatique des règles de transition de l'algèbre de processus utilisée. Cette extension rendrait le logiciel encore plus flexible et plus facilement adaptable aux divers changements de la sémantique opérationnelle des opérateurs de composition. Toutefois, des applications futures sur des exemples plus complexes renforceront notre conviction que l'outil crée est bien adapté pour la vérification des interfaces matérielles des systèmes à temps réel.

Bibliographie

[1] A. M. Andreescu-Hilohi, *Axiomatisation d'un langage de description des chrono-grammes*, Thèse de maîtrise, Université de Montréal, 1998. Rédaction en cours.

[2] A. Tarnauceanu, *Logiciel pour la vérification par simulation de la spécification de haut niveau de systèmes matériels*, Thèse de maîtrise, Université de Montréal, 1997.

[3] K. Khordoc, E. Cerny, *Modelling Cell Processing Hardware with Action Diagrams,* IEEE ISCAS '94, # pages 245-248, London, England, 1994.

[4] R. Lipsett, C. Schaefer, C. Ussery, *VHDL : Hardware Description and Design,* Kluwer Academic Publisher, 1989.

[5] P. A. Babkine, *Un outil pour la spécification de matériel et la génération de modèles exécutables*, Thèse de maîtrise, Université de Montréal, 1996.

[6] VANTAGE, *VantageSpreadsheet for Electronic Design Verification and Analysis,* User Guide (vol. 1 et 2).

[7] SYNOPSYS, *VHDL System Simulator Core Programs Manual,* version 3.0, 1992.

[8] R. Alur, C. Courcoubetis, D. Dill, *Model-checking for real-time systems*, Proc. 5th Symp. on Logics in Computer Science, IEEE Computer Society Press, 1990.

[9] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, *Symbolic model checking: 10 to the power of 20 and beyond*, Proc. 5th Symp. on Logics in Computer Science, IEEE Computer Society Press, 1990.

[10] K. L. McMillan, *Symbolic model checking - an approach to state explosion problem*, PhD thesis, Carnegie Mellon Univ., 1992.

[11] J. Yang, A. K. Mok, F. Wang, *Symbolic model-checking for event-driven real-time systems*, Proc. of the 14th IEEE Real-Time Syst. Symp., Raleigh-Durham, December 1993.

[12] J. Ostroff, *Formal methods for the specification and the design of real-time safety-critical systems*, Journal of Systems and Software, vol. 33, # pages 890-904, 1992.

[13] X. Nicollin, J. Sifakis, *The algebra of timed processes ATP: Theory and application*, Information and Computation, 114, # pages 131-178, 1994.

[14] X. Nicollin, J. Sifakis, *An overview and synthesis on timed process algebras*, Proc. of the Computer Aided Verification Conference, vol. 575 de Lecture Notes in Computer Science, 1991.

[15] F. Moller, C. Tofts, *A temporal calculus of communicating processes*, (éditeurs J. C. M. Baeten et J. W. Klop), Proc. of CONCUR'90, Theories of concurrency: Unification and Extension, vol. 458 de Lecture Notes in Computer Science, # pages 401-415, Amsterdam, 1990, Springer-Verlag.

[16] B. Berkane, S. Gandrabur, E. Cerny, *Algebra for communicating timing charts for describing and verifying hardware interfaces*, Proc. of the Conference on Hardware Description Languages CHDL'97, Toledo, Spain, 1997.

[17] T. Bolognese, E. Brinksma, *Introduction to the ISO specification language LOTOS*, Computer Networks and ISDN Systems, vol. 14, # pages 25-59, 1987

[18] *Practical Formal Methods for Hardware Design*, Project 6128, FORMAT, vol. 1, C. Delgado Kloos et W. Damm (Editeurs), Springer, 1997.

[19] C. A. Petri, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood, 1981.

[20] G. Bruno, *Model-based Software Engineering*, Chapman-Hall, London, 1995.

[21] T. Yoneda, A. Shibayama, H. Schlingloff, E. M. Clarke, *Efficient verification of parallel real-time systems*, Lecture Notes in Computer Science, vol. 697, # pages 321-332, 1993.

[22] D. Drusinsky, D. Harel, *Using StateCharts for Hardware Description and Synthesis,* IEEE Transactions on Computer-Aided Design, vol. 8, no. 7, # pages 798-806, 1989.

[23] G. Borriello, *A New Interface Specification Methodology and its Application to Transducter Synthesis,* PhD Thesis, University of California, Berkeley, 1988.

[24] Y. Leong, William P. Birmingham, *The Automatic Generation of Bus-Interface Models,* ACM/IEEE Proc. of the 29th DAC, # pages 634-637, 1992.

[25] X. Nicollin, J. Sifakis, S. Yovine, *Computing Real-Time Specifications into Extended Automata*, IEEE Transactions on Software Engineering, vol. 18, no. 9, #

pages 794-804, September 1992.

[26] R. Alur, *Techniques for automatic verification of real-time systems*, PhD thesis, Department of Computer Science, Stanford University, August 1991.

[27] A. Tarnauceanu, *HAAD System Description*, Listage, Laboratoire LASSO, DIRO, Université de Montréal, 1997.

[28] G.M. Reed, A.W. Roscoe, *A timed model for Communicating Sequential Processes*, Theoretical Computer Science, vol. 58, # pages 249-261, 1988.

[29] J. Baeten, J. Bergstra, *Real-time process algebra*, Formal Aspects of Computing, vol. 3, no. 2, # pages 142-188, 1991.

[30] A. Klusener, *Models and axioms for a fragment of real time process algebra*, PhD thesis, CWI, 1993.

[31] D. Harel, *Statecharts: A visual formalism for complex systems,* Science of Computer Programming, vol. 8, # pages 231-274, 1987.

[32] A. Pnueli, *Applications of temporal logic to the specification and verification of reactive systems*, Current Trends in Concurrency, # pages 365-377, Springer Verlag, 1986.

[33] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine, A. Silburt, *Integrating Behavior and Timing in Executable Specifications,* IFIP Conference on Hardware Description Languages and their Applications, # pages 399-416, 1993.

[34] G. A. Gibson, *Computer Systems. Concepts and Design*, Prentice Hall, 1991.

[35] G. D. Plotkin, *A structural approach to operational semantics*, Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[36] E. Cerny, B. Berkane, P. Girodias, K. Kordoc, *Hierarchical Annotated Action Diagrams: An Interface-Oriented Specification and Verification Method*, Kluwer Academic Publishers, 1998.

ANNEXE A

La grammaire du langage de spécification HAAD

- La grammaire du langage de spécification HAAD est donnée dans la forme normale Backus-Naur.
- Les minuscules sont des non-terminaux ou des expressions LISP ou des nombres / symboles / chaînes de caractères.
- Un choix est indiqué par "|"
- Une liste d'articles entre accolades "{}" et séparés par des barres verticales indique un nombre d'occurrences positif ou zéro des articles spécifiés à l'intérieur des accolades. A l'intérieur de cette liste, les articles entre chevrons "<>" peuvent apparaître au plus une fois.

defbehavior ::= (DEFBEHAVIOR had-type-nameDef
{ <ports> | <parameters> | <generics> | time-units |
<corr-groups> | local-corr-group | signal | var |
<had-body> })

corr-groups ::= (CORR-GROUPS { corr-group })

corr-group ::= (CORR-GROUP group-nameDef { <correlation-degree> |
<correlation-point>)

local-corr-group ::= (LOCAL-CORR-GROUP local-group-nameDef correlation-degree
<correlation-point>)

correlation-degree ::= (CORRELATION-DEGREE degree)

degree ::= number

correlation-point ::= (CORRELATION-POINT point)

degree ::= number

point ::= number

time-units ::= (TIME-UNITS unit)

unit ::= unit-nameDef

ports ::= (PORTS { port })

port ::= (PORT port-nameDef direction v-type-nameRef interpretation)

direction ::= INOUT | IN | OUT

interpretation ::= EVENT | MESSAGE

parameters ::= (PARAMS { parameter })

parameter ::= (PARAM param-nameDef direction v-type-nameRef)

generics ::= (GENERICS { generic-nameDef })

signal ::= (SIGNAL signal-nameDef v-type-nameRef interpretation
{ <v-value> })

var ::= (VAR var-nameDef v-type-nameRef { <v-value> })

had-body ::= leaf { carrier-group | partition | <generation-instants> } | recX |
concatenation | parallel | d-choice | nd-choice | exception

generation-instants ::= (GENERATION-INSTANTS value-generation-instants)

value-generation-instants ::= number

carrier-group ::= (CARRIER-GROUP local-group-or-group-nameRef { tc-nameRef })

partition ::= (BLOCK-PARTITION block-number { action-nameRef })

block-number ::= number

leaf ::= (LEAF { carrier-spec | constraint })

carrier-spec ::= (CARRIER-SPEC signal-or-port-nameRef { <initial-spec> }
{ <action-start> } { action-spec } { <action-end> })

initial-spec ::= (INITIAL-SPEC state { <action-direction> })

action-direction ::= IN | OUT

action-start ::= (ACTION-START action-nameDef {<predicate-call> |
<procedure-call> })

action-end ::= (ACTION-END action-nameDef {<predicate-call> |
<procedure-call> }))

action-spec ::= (ACTION-SPEC action-nameDef state {
<action-direction-spec> |
<predicate-call> |
<procedure-call> })

action-direction-spec :: (DIRECTION action-direction)

state ::= dont-care | constant | valid

dont-care ::= (DONT-CARE)

constant ::= (CONSTANT v-value)

valid ::= (VALID { <var-nameRef> })

procedure-call ::= (PROCEDURE-CALL v-prog-nameRef
{ var-or-param-or-signal-or-port-nameRef })

predicate-call ::= (PREDICATE-CALL v-prog-nameRef
{ var-or-param-or-signal-or-port-nameRef })

constraint ::= conjunctive | earliest | latest | precedence | concurrency

conjunctive::= (CONJUNCTIVE { <tc-name-spec> | constraint })

earliest ::= (EARLIEST { <tc-name-spec> | constraint })

latest ::= (LATEST { <tc-name-spec> | constraint })

precedence ::= (PRECEDENCE source-action-nameRef sink-action-nameRef
{ <tc-name-spec> | <intent-spec> | <min-spec> | <max-spec> })

concurrency ::= (CONCURRENCY source-action-nameRef sink-action-nameRef
{ <tc-name-spec> | <intent-spec> | <min-spec> | <max-spec> })

tc-name-spec ::= (CNAME tc-nameDef)

intent-spec ::= (INTENT intent)

intent ::= ASSUME | COMMIT | REQUIREMENT

min-spec ::= (CMIN min)

max-spec ::= (CMAX max)

min ::= number

max ::= number

recX ::= (RECX had
{ <start-action> | <predicate-call> | <end-action> | <recurrence> })

concatenation ::= (CONCATENATION
{ <start-action> | had | <end-action> })

parallel ::= (PARALLEL
{ <start-action> | had | <end-action> })

d-choice ::= (D-CHOICE
{ <start-action> | choice-branch | <end-action> })

nd-choice ::= (ND-CHOICE
{ <start-action> | choice-branch | <end-action> })

choice-branch ::= (BRANCH had { <predicate-call> })

start-action ::= (START-ACTION { <predicate-call> | <procedure-call> })

end-action ::= (END-ACTION { <predicate-call> | <procedure-call> })

exception ::= (EXCEPTION {<condition> | <normal> | <handler> |
<start-action> | <end-action>})

condition ::= (CONDITION had)

normal ::= (NORMAL had)

handler ::= (HANDLER had)

recurrence ::= (RECURRENCE bool)

bool ::= TRUE | FALSE

had ::= had-instance-spec | had-body

had-instance-spec ::= (BEHAVIOR had-instance-nameDef had-type-nameRef
{ <corr-group-map> | <port-map> | <param-map> |
<generic-map> })

corr-group-map ::= (CORR-GROUP-MAP { group-nameRef })

port-map ::= (PORT-MAP { signal-or-port-nameRef })

param-map ::= (PARAM-MAP { var-or-param-nameRef })

generic-map ::= (GENERIC-MAP { lisp-expression })

ANNEXE B

Les structures de données du modèle VHDL

Le fichier PIPE_generated_data_structures.vhd contient (pour l'ensemble complet des structures de données du modèle, voir [4] et [27]):

```
TYPE tc;
TYPE ptr_tc IS ACCESS tc;
TYPE partition_block;
TYPE ptr_partition_block IS ACCESS partition_block;
TYPE array_of_blocks_trigger IS ARRAY (0 TO 4 ) OF ptr_partition_block;

TYPE partition_block IS RECORD
        block_actions          : block_actions_array;
        block_triggers         : block_triggers_array;
        matrix_0               : block_matrix;
        matrix_k_1             : block_matrix;
        matrix_k               : block_matrix;
        number_of_actions      : integer;
        number_of_triggers     : integer;
        brother                : ptr_partition_block;
        active                 : boolean;
        index                  : integer;
        only_latest_earliest   : boolean;
END RECORD;

TYPE tc IS RECORD
        kind                   : constraint_op;
        intent                 : intend_type;
        min, max               : utime;
```

```
        source, sink            : ptr_spec_action;
        name                    : string (1 to 10);
        mapping                 : group_space;
        son, brother            : ptr_tc;
        spread, corr_point      : integer;
        current_value           : utime;
        number                  : integer;
        change                  : boolean;
        link                    : ptr_tc;
        corr_next               : ptr_tc;
        corr_brother            : ptr_tc;
        its_next_instance       : ptr_tc;
        initial_min             : utime;
        initial_max             : utime;
END RECORD;

TYPE tc_link;
TYPE ptr_tc_link IS ACCESS tc_link;

TYPE tc_link IS RECORD
        it                      : ptr_tc;
        brother                 : ptr_tc_link;
END RECORD;

TYPE h_port;
TYPE ptr_h_port IS ACCESS h_port;

TYPE had_node;
TYPE ptr_had_node IS ACCESS had_node;
```

```
TYPE h_port IS RECORD
        mapping                 : port_space;
        const                   : boolean;
        init_value              : ptr_uptv;
        first_action            : ptr_spec_action;
        brother                 : ptr_h_port;
        had                     : ptr_had_node;
        current_action          : ptr_spec_action;
        auto_var_id             : auto_var_assignation_range;
        auto_init_var_id        : auto_init_var_assignation_range;
        active                  : boolean;
        not_yet_active          : boolean;
        same_father_mappings    : integer;
        last_action             : ptr_spec_action;
        link                    : ptr_h_port;
END RECORD;

TYPE spec_action IS RECORD
        set_vector_instant      : boolean;
        partion_block_as_action : ptr_partition_block;
        partion_block_as_trigger : array_of_blocks_trigger;
        number_of_blocks_trigger : integer;
        index_in_block_matrix   : integer;
        name                    : string (1 to 20);
        gendre                  : action_type;
        direction               : direction_type;
        next_action             : ptr_spec_action;
        procedure_id            : procedure_call_range;
        predicate_id            : action_predicate_range;
        auto_var_id             : auto_var_assignation_range;
        optional, MMR           : boolean;
```

```
        h_port                                              : ptr_h_port;
        had                                                 : ptr_had_node;
        assume, commit, requirement                         : ptr_tc;
        tc_out                                              : ptr_tc_link;
        value                                               : ptr_uptv;
        insert, pred_call, latest                           : boolean;
        conjunctive, earliest, set                          : boolean;
        virtual_occured, normal_occured                     : boolean;
        occ_utime, trigger_utime                            : utime;
        a_boundaries, c_boundaries, r_boundaries            : utime_interval;
        last_dont_care, last_valid, fixed_valid, end_valid  : boolean;
        now, now_dc, delete, executed_as_next, normal       : boolean;
        link                                                : ptr_spec_action;
        its_next_instance                                   : ptr_spec_action;
END RECORD;

TYPE flat_port IS RECORD
        last_observed_activity, last_generated_activity     : utime;
        interpretation                                      : interpretation_type;
        current_value                                       : uptv;
        active                                              : boolean;
        signal_port                                         : boolean;
END RECORD;

TYPE flat_port_array IS ARRAY (port_space) OF flat_port;

TYPE flat_group IS RECORD
        corr_degree                                         : integer;
END RECORD;

TYPE flat_group_array IS ARRAY (group_space) OF flat_group;
```

```
TYPE had_node IS RECORD
        partition_blocks_list                                   : ptr_partition_block;
        ports_list                                              : ptr_h_port;
        kind                                                    : had_kind;
        recX                                                    : boolean;
        son, brother                                            : ptr_had_node;
        loop_pred, choice_pred                                  : had_predicate_call_range;
        start_action, end_action                                : ptr_spec_action;
        father                                                  : ptr_had_node;
        array_of_instants                                       : ptr_array_of_instants;
        array_of_group_instants : ptr_array_of_instants_for_groups;
        commit_tcs, correlated_commit_tcs                       : ptr_tc;
        active, disabeled, choice_flag, exception_cond          : boolean;
        exception_normal, exception_beh, print                  : boolean;
        finish, error, loop_pred_res, choice_pred_res           : boolean;
        end_exhaustive_enumeration, end_exhaustive_enumeration_global : boolean;
        one_more_local_enum, have_global_enum                   : boolean;
        normal_execution, first_time, no_enumeration_global : boolean;
        no_enumeration, no_father_loop, no_constraints          : boolean;
        not_executing, first_execution, one_more, down          : boolean;
        one_more_execution, forced                              : boolean;
        number_of_constraints, number_of_groups                 : integer;
        number_of_generation_instants                           : integer;
        number_of_the_had                                       : integer;
        number_max_of_iterations                                : integer;
        number_of_iterations, current_iteration                 : integer;
        its_next_instance                                       : ptr_had_node;
        its_prev_instance                                       : ptr_had_node;
        last_activated                                          : ptr_had_node;
        link                                                    : ptr_had_node;
        next_recX                                               : ptr_had_node;
```

```
        initial_recX                    : boolean;
        new_node                        : boolean;
        lost_brother                    : ptr_had_node;
        min_one_activ_port              : boolean;
END RECORD;
```

ANNEXE C

Le fichier DATA_PROVIDER.vhd contient:

```
LIBRARY WORK, ADEL, IEEE;


USE std.textio.all, ieee.std_logic_1164.all;
USE WORK.TYPES.ALL, WORK.PROC.ALL;


entity DATA_PROVIDER is
  port(data_port: OUT inout_bus);
end DATA_PROVIDER;


-- l'entité DATA_PROVIDER fournit les données (des entiers consécutifs partant de la valeur 1)
-- sur le port d'entrées X_PD1 (voir la figure 45) après chaque intervalle de 5 ns


architecture DATA_PROVIDER_COMPONENT of DATA_PROVIDER is
begin
 PROCESS
   VARIABLE local_value : inout_bus := "00000000";
   VARIABLE i : integer := 1;
   VARIABLE carry, temp : std_logic := '0';
BEGIN
   carry := '0';
   i := 1;

   IF (local_value(0) = '1') THEN
      local_value(0) := '0';
      carry := '1';
   ELSE
      local_value(0) := '1';
   END IF;
   WHILE (i <= 7) LOOP
      temp := local_value(i);
      local_value(i) := local_value(i) XOR carry;
      carry := temp AND carry;
      i := i + 1;
   END LOOP;
   data_port <= local_value;
   WAIT FOR 5  ns;
END PROCESS;

end DATA_PROVIDER_COMPONENT;
```

Le fichier types.vhd contient:

```
PACKAGE types IS
  TYPE inout_bus IS ARRAY (0 TO 7) OF std_logic;
  TYPE data_table IS ARRAY ( 0 TO 4 ) OF inout_bus;
END types;
```

Le fichier master_instance.vhd contient:

```
LIBRARY
  WORK,
  IEEE;


USE
  WORK.types.all,
  WORK.DATA_PROVIDER,
  WORK.PIPE,
  ieee.std_l
ogic_1164.all;

 ARCHITECTURE proc_mem OF instance IS
  SIGNAL PIPE_PD1_TOP, PIPE_PD2_TOP : inout_bus;
  SIGNAL EMPTY_PD3_TOP : std_logic;


 COMPONENT DATA_PROVIDER_INSTANCE
        PORT (data_port : OUT inout_bus);
 END COMPONENT;


 COMPONENT PIPE_INSTANCE
        PORT (PIPE_PD1 : IN  inout_bus;
            PIPE_PD2 : INOUT inout_bus;
            EMPTY_PD3 : IN std_logic); -- ce port appartient au diagramme hiérarchique du pipeline
                                       -- sous forme de récursion (voir le diagramme feuille Empty_F
                                       -- de la figure 47 etle diagramme hiérarchique du pipeline
                                       -- de la figure 49)
        END COMPONENT;

   FOR ALL : DATA_PROVIDER_INSTANCE USE ENTITY WORK.DATA_PROVIDER
(DATA_PROVIDER_COMPONENT) ;
   FOR ALL : PIPE_INSTANCE USE ENTITY WORK.PIPE (ADELe) ;

 BEGIN
   g1 : DATA_PROVIDER_INSTANCE
             PORT MAP (PIPE_PD1_TOP) ;
   g2 : PIPE_INSTANCE PORT MAP (PIPE_PD1_TOP, PIPE_PD2_TOP, EMPTY_PD3_TOP) ;
 END proc_mem ;
```

ANNEXE D

Le fichier proc.vhd contient:

```
LIBRARY
      WORK,
     IEEE;


 USE
   WORK.types.all,
   ieee.std_logic_1164.all,
   std.textio.all;


PACKAGE proc IS

  PROCEDURE  load_input          (varX : IN inout_bus;
                                  first : IN integer;
                                  last : INOUT integer;
                                  data_buffer : INOUT data_table);

  PROCEDURE compute_output       (first : INOUT integer;
                                  last : INOUT integer;
                                  data_buffer : INOUT data_table;
                                  varY : INOUT inout_bus);

  FUNCTION array_is_empty        (first : IN integer;
                                  last  : IN integer) RETURN boolean;

  FUNCTION array_is_full         (first : IN integer;
                                  last  : IN integer) RETURN boolean;

END proc;
```

```
PACKAGE BODY proc IS

PROCEDURE load_input          (varX  : IN inout_bus; first : IN integer;
                               last  : INOUT integer; data_buffer : INOUT data_table) IS
  VARIABLE ligne : LINE;
  VARIABLE is_full : boolean;
BEGIN

  IF (array_is_full(first, last)) THEN
                WRITE (ligne, STRING'("The array is full"));
  ELSE
                data_buffer(last) := varX;
                last := (last + 1) mod (5);
  END IF;
  WRITE (ligne, STRING'("varX = "));
  WRITE (ligne, varX);
  WRITELINE (OUTPUT, ligne);

END load_input;


PROCEDURE compute_output      (first : INOUT integer; last  : INOUT integer;
                               data_buffer : INOUT data_table; varY : INOUT inout_bus)  IS
  VARIABLE ligne : LINE;
  VARIABLE is_empty : boolean;
  VARIABLE temp, temp1, carry : std_logic := '0';
  VARIABLE i : integer := 1;
BEGIN

  IF (array_is_empty(first, last)) THEN
                WRITE (ligne, STRING'("The array is empty"));
  ELSE
                varY := data_buffer(first);
                IF (varY(0) = '1') THEN
                                varY(0) := '0';
                                carry   := '1';
                ELSE
                                varY(0) := '1';
                END IF;
                WHILE (i <= 7) LOOP
                                temp := varY(i);
                                varY(i) := varY(i) XOR carry;
                                carry := temp AND carry;
                                i := i + 1;
                END LOOP;
                temp1 := varY(0);
                first := (first + 1) mod (5);
  END IF;
  WRITE (ligne, STRING'("Result varY = "));
  WRITE (ligne, temp1);
  WRITELINE (OUTPUT, ligne);
END compute_output;

 END proc;
```

ANNEXE E

Fragment de la trace d'exécution du *pipeline* (le signe "~" est notre notation pour l'infini):

had-loop-1/concatenation-1/leaf-1/ is started at utime : 0
concatenation-1/ is started at utime : 0
concatenation-1/leaf-1/BEGIN_P1 @ occurence time : 0 - is the START action of a port and is NORMAL executed
concatenation-1/leaf-1/BEGIN_P2 @ occurence time : 0 - is the START action of a port and is NORMAL executed
concatenation-1/leaf-1/ACTX @ occurence time : 0 - is NORMAL executed as the result of a mapping with a real event on the port
varX = 1
concatenation-1/leaf-1/END_P1 @ occurence time : 5 - is the END action of a port and is NORMAL executed
concatenation-1/recX-2/concatenation-1/leaf-1/ is started at utime : 5
concatenation-1/concatenation-2/ is started at utime : 5
concatenation-1/concatenation-2/leaf-1/BEGIN_P1 @ occurence time : 5 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/leaf-1/ACTX @ occurence time : 5 - is NORMAL executed as the result of a mapping with a real event on the port
varX = 2
concatenation-1/concatenation-2/leaf-1/END_P1 @ occurence time : 10 - is the END action of a port and is NORMAL executed
concatenation-1/concatenation-2/recX-2/concatenation-1/leaf-1/ is started at utime : 10
concatenation-1/concatenation-2/concatenation-2/ is started at utime : 10
concatenation-1/concatenation-2/concatenation-2/leaf-1/BEGIN_P1 @ occurence time : 10 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/leaf-1/ACTX @ occurence time : 10 - is NORMAL executed as the result of a mapping with a real event on the port
varX = 3
concatenation-1/concatenation-2/concatenation-2/leaf-1/END_P1 @ occurence time : 15 - is the END action of a port and is NORMAL executed
Result varY = 2
concatenation-1/concatenation-2/concatenation-2/recX-2/concatenation-1/leaf-1/ is started at utime : 15
concatenation-1/concatenation-2/concatenation-2/concatenation-2/ is started at utime : 15
concatenation-1/concatenation-2/concatenation-2/concatenation-2/leaf-1/BEGIN_P1 @ occurence time : 15 - is the START action of a port and is NORMAL executed
concatenation-1/leaf-1/ACTY @ occurence time : 15 - is an OUT action and is NORMAL executed because it reached its trigger time
concatenation-1/concatenation-2/concatenation-2/concatenation-2/leaf-1/ACTX @ occurence time : 15 - is NORMAL executed as the result of a mapping with a real event on the port
varX = 4
concatenation-1/leaf-1/END_P2 @ occurence time : 15 - is the END action of a port and is NORMAL executed
concatenation-1/leaf-1/ is ended at utime : 15
concatenation-1/concatenation-2/leaf-1/BEGIN_P2 @ occurence time : 15 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/concatenation-2/leaf-1/END_P1 @ occurence time : 20 - is the END action of a port and is NORMAL executed
Result varY = 3
concatenation-1/concatenation-2/concatenation-2/concatenation-2/recX-2/concatenation-1/leaf-1/ is started

at utime : 20
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/ is started at utime : 20
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/BEGIN_P1 @ occurence time : 20 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/leaf-1/ACTY @ occurence time : 20 - is an OUT action and is NORMAL executed because it reached its trigger time
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/ACTX @ occurence time : 20 - is NORMAL executed as the result of a mapping with a real event on the port
varX = 5
concatenation-1/concatenation-2/leaf-1/END_P2 @ occurence time : 20 - is the END action of a port and is NORMAL executed
concatenation-1/concatenation-2/leaf-1/ is ended at utime : 20
concatenation-1/concatenation-2/concatenation-2/leaf-1/BEGIN_P2 @ occurence time : 20 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/END_P1 @ occurence time : 25 - is the END action of a port and is NORMAL executed
Result varY = 4
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/recX-2/concatenation-1/leaf-1/ is started at utime : 25
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/ is started at utime : 25
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/BEGIN_P1 @ occurence time : 25 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/leaf-1/ACTY @ occurence time : 25 - is an OUT action and is NORMAL executed because it reached its trigger time
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/ACTX @ occurence time : 25 - is NORMAL executed as the result of a mapping with a real event on the port
varX = 6
concatenation-1/concatenation-2/concatenation-2/leaf-1/END_P2 @ occurence time : 25 - is the END action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/leaf-1/ is ended at utime : 25
concatenation-1/concatenation-2/concatenation-2/concatenation-2/leaf-1/BEGIN_P2 @ occurence time : 25 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/END_P1 @ occurence time : 30 - is the END action of a port and is NORMAL executed
Result varY = 5
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/recX-2/concatenation-1/leaf-1/ is started at utime : 30
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/ is started at utime : 30
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/BEGIN_P1 @ occurence time : 30 - is the START action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/concatenation-2/leaf-1/ACTY @ occurence time : 30 - is an OUT action and is NORMAL executed because it reached its trigger time
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/ACTX @ occurence time : 30 - is NORMAL executed as the result of a mapping with a real event on the port
varX = 7
concatenation-1/concatenation-2/concatenation-2/concatenation-2/leaf-1/END_P2 @ occurence time : 30 - is the END action of a port and is NORMAL executed
concatenation-1/concatenation-2/concatenation-2/concatenation-2/leaf-1/ is ended at utime : 30
concatenation-1/concatenation-2/concatenation-2/concatenation-2/concatenation-2/leaf-1/BEGIN_P2 @ occurence time : 30 - is the START action of a port and is NORMAL executed