THESE

présentée par

Radu MATEESCU

pour obtenir le grade de **DOCTEUR**

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 30 mars 1992)

(Spécialité : INFORMATIQUE)

Vérification des propriétés temporelles des programmes parallèles

Date de soutenance :	10 avril 1998		
Composition du jury :	MM.	JEAN-PIERRE VERJUS	Président
		ANDRÉ ARNOLD RANCE CLEAVELAND	Rapporteurs
		HUBERT GARAVEL JOSEPH SIFAKIS ROBERT DE SIMONE	Examinateurs

Thèse préparée au sein de l'Unité de Recherche INRIA Rhône-Alpes
(projet SPECTRE / laboratoire VERIMAG, puis action VASY)

Remerciements

Je tiens à remercier :

Jean-Pierre Verjus, *Professeur à l'INPG, Directeur de l'Unité de Recherche INRIA Rhône-Alpes, pour m'avoir fait l'honneur de présider le jury de cette thèse*

André Arnold, *Professeur à l'Université de Bordeaux, pour avoir accepté de juger ce travail et pour ses encouragements à poursuivre les idées exposées dans ce document*

Rance Cleaveland, *Professeur Associé à l'Université de Caroline du Nord (Etats-Unis), pour avoir eu la patience de lire soigneusement le manuscrit et pour les critiques constructives qu'il m'a fournies en retour*

Hubert Garavel, *Chargé de recherche à l'INRIA, qui a assumé la direction de cette thèse, pour les nombreuses heures de discussion et les relectures attentives qui m'ont permis d'améliorer aussi bien le contenu que la forme du manuscrit*

Joseph Sifakis, *Directeur de recherche au CNRS, pour m'avoir accueilli au sein du projet SPECTRE / laboratoire VERIMAG durant la première partie de ma thèse et pour l'attention qu'il a portée à la lecture de ce document*

Robert de Simone, *Directeur de recherche à l'INRIA, pour avoir examiné soigneusement ce travail et pour ses précieux commentaires et suggestions.*

Je suis reconnaissant envers tous les membres du projet SPECTRE / laboratoire VERIMAG, en particulier Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Peter Habermehl, Alain Kerbrat, Florence Maraninchi, Laurent Mounier, Xavier Nicollin, Pascal Raymond et Sergio Yovine, auprès desquels j'ai toujours trouvé des conseils pertinents et qui ont su créer une atmosphère de travail accueillante dont je garderai un excellent souvenir.

Mes remerciements vont également à Mihaela Sighireanu, Charles Pecheur, Mark Jorgensen et Bruno Vivien pour la relecture attentive des parties de cette thèse et pour leurs commentaires avisés qui m'ont permis d'améliorer le document final. Je n'oublierai pas l'amitié et le soutien qu'ils m'ont témoigné en permanence, contribuant ainsi à rendre mon séjour au sein de l'équipe VASY de l'INRIA Rhône-Alpes particulièrement agréable.

Table des matières

Introduction

Les applications réparties, telles que les protocoles de communication et les systèmes distribués, sont caractérisées par une grande complexité. En même temps, à cause de leur caractère critique, elles sont souvent soumises à de sévères exigences de fiabilité, visant la qualité *zéro défaut*. Il est maintenant largement reconnu que la conception de telles applications nécessite des méthodes et outils de vérification formelle, permettant d'assurer leurs propriétés de bon fonctionnement. De nombreux formalismes ont été définis pendant les dernières années, dédiés d'une part à la description des applications réparties et, d'autre part, à la spécification de leurs propriétés attendues.

Il existe essentiellement deux approches dans le domaine de la vérification formelle : celle basée sur la preuve de théorèmes (*theorem-proving*) et celle basée sur les modèles (*model-checking*). Ces deux approches ont été étudiées intensivement dans la littérature et de nombreux algorithmes et outils ont été développés. Les méthodes basées sur la preuve de théorèmes permettent de traiter des systèmes ayant un nombre infini d'états, mais elles ne peuvent pas être complètement automatisées. En revanche, les techniques basées sur les modèles, bien que restreintes à des systèmes ayant un nombre fini d'états, permettent une vérification simple et efficace, qui s'avère particulièrement utile dans les premières phases du processus de conception, quand les erreurs sont susceptibles d'être plus fréquentes.

Dans l'approche basée sur les modèles, l'application à vérifier est d'abord décrite dans un langage parallèle de haut niveau ayant une sémantique opérationnelle bien définie, comme LOTOS[1] [ISO88b], μCRL[2] [GP90], etc. Ensuite, cette description est traduite vers un modèle sous-jacent, qui souvent est un *système de transitions étiquetées* (STE), c'est-à-dire un graphe (ou automate) contenant, éventuellement avec certaines abstractions, tous les comportements possibles du programme. Finalement, les propriétés de bon fonctionnement de l'application, exprimées dans un formalisme approprié (logiques temporelles, μ-calcul, etc.), sont vérifiées sur le modèle STE à l'aide d'outils appelés *évaluateurs* (*model-checkers*).

Logiques temporelles et vérification de programmes parallèles

Les propriétés de bon fonctionnement des programmes parallèles portent sur leurs comportements, c'est-à-dire les séquences (éventuellement infinies) d'états ou d'actions générées pendant leur exécution. Les logiques temporelles sont bien adaptées pour spécifier ces propriétés, car elles permettent d'obtenir des spécifications *abstraites* et *modulaires* du programme [MP90]. L'abstraction signifie l'indépendance de la spécification par rapport à toute implémentation : les propriétés requises sont exprimées séparément, sans indiquer la manière dont elles sont implémentées. La modularité signifie qu'une spécification est facilement modifiable en rajoutant, enlevant ou modifiant une des propriétés ;

[1] Language Of Temporal Ordering Specification

[2] *micro* Common Representation Language

de plus, le processus de vérification sur un modèle peut aussi être effectué de façon modulaire, en vérifiant chaque propriété séparément.

De nombreuses logiques temporelles ont été définies et étudiées dans la littérature. Parmi les plus représentatives, nous pouvons mentionner :

les logiques modales comme HML [HM85], munies d'opérateurs de possibilité ("$\langle\ \rangle$") et de nécessité ("[ ]") exprimant l'exécution possible ou obligatoire d'actions individuelles du programme ;

les logiques temporelles linéaires comme LTL [Lam80] ou PTL [MP92], qui permettent de décrire des propriétés portant sur les séquences d'exécution individuelles (issues de l'état initial) du programme ;

les logiques temporelles avec expressions régulières comme PDL [FL79], PDL-Δ [Str82] ou $\overset{*}{\mathrm{R}}$ICO [Gar89a, chap. 9], autorisant une caractérisation concise et naturelle des séquences régulières d'actions du programme ;

les logiques temporelles arborescentes comme LTAC [QS83], CTL et CTL* [CES86, EH86], ACTL et ACTL* [NV90, NFGR91], qui permettent de décrire des propriétés portant sur les arbres d'exécution du programme ;

les logiques avec opérateurs de point fixe comme le μ-calcul modal [Koz83], le calcul de Dicky [Dic86], la logique STL [GS86] ou la logique HML avec récursion [Lar88], qui permettent de caractériser des arborescences (finies ou infinies) de l'exécution du programme au moyen d'un ensemble restreint d'opérateurs primitifs.

Tous ces formalismes sont définis sur un vocabulaire d'actions *atomiques* (noms de portes ou de canaux de communication), ce qui permet d'exprimer des propriétés temporelles sur le comportement des programmes décrits dans des algèbres de processus "pures" (comme *basic* LOTOS) ou des systèmes de processus communicants comme MEC [Arn89, ABC94] et FC2 [BRRd96]. Par contre, ceci ne permet pas de caractériser les propriétés des programmes décrits dans des langages avec valeurs (comme *full* LOTOS ou μCRL), dont les modèles STE associés contiennent des actions structurées (noms de portes et valeurs échangées par rendez-vous). Par exemple, pour exprimer la propriété *"après l'émission d'un message m, il est inévitable d'atteindre la réception du même message"* dans une logique temporelle classique, il faudrait écrire une formule différente pour chaque message m contenu dans les actions du STE, ce qui en pratique est prohibitif tant en taille de la spécification qu'en efficacité d'évaluation.

Il existe aussi de nombreux outils de vérification consacrés aux logiques temporelles : EMC [CES86], développé à l'Université de Carnegie-Mellon ; CWB [CPS89], développé à l'Université d'Edimbourg ; MEC [Arn89, ABC94], développé à l'Université de Bordeaux ; XESAR [GRRV89], développé à IMAG, Grenoble ; SPIN [Hol91], développé aux laboratoires AT&T ; TAV [Lar92], développé à l'Université de Aalborg ; MWB [VM94], développé à l'Université d'Uppsala ; JACK [BGL94], développé à l'Université de Pise ; Concurrency Factory [CLSS96], développé à l'Université de Caroline du Nord et à SUNY, Stony Brook.

Bien qu'ils mettent en œuvre des algorithmes efficaces, la plupart de ces évaluateurs souffrent de certaines limitations. D'une part, ils sont souvent liés à un langage de description des applications (ou au modèle sous-jacent) particulier, ce qui les rend difficilement adaptables à d'autres langages ou modèles. D'autre part, les outils dédiés à une logique temporelle particulière peuvent s'avérer incapables de vérifier certaines classes de propriétés. Par exemple, l'outil EMC est dédié à la logique CTL (qui ne permet d'exprimer que des propriétés sur les états), ce qui le rend inutilisable pour vérifier des propriétés sur les actions. Cet inconvénient est accentué par le fait que le développement d'un évaluateur pour une logique temporelle particulière est une opération complexe et coûteuse.

Contexte de la thèse

Notre travail de thèse s'inscrit dans le contexte de la boîte à outils CADP (CÆSAR/ALDÉBARAN) [FGK+96] pour la vérification de programmes LOTOS. L'approche que nous avons adoptée vise à remédier aux limitations des logiques temporelles classiques concernant l'expression et la vérification des propriétés qui portent aussi sur les valeurs manipulées dans les programmes à vérifier. Plus précisément, nous avons poursuivi un double objectif :

- Définir un formalisme de spécification capable d'exprimer des propriétés temporelles portant sur les valeurs, qui soit suffisamment puissant pour permettre une description concise des propriétés exprimables dans les logiques temporelles classiques ;

- Concevoir et développer un évaluateur capable d'interpréter sur des modèles STE les propriétés exprimées dans ce formalisme. Cet outil doit être, autant que possible, indépendant du langage des programmes à vérifier, tout en ayant des performances comparables aux meilleurs évaluateurs existants.

Pour exprimer les propriétés temporelles, nous avons conçu un langage de spécification spécial, appelé XTL (*eXecutable Temporal Language*), constitué principalement des éléments suivants.

Mécanismes de filtrage : ces constructions permettent d'extraire les valeurs contenues dans les états ou les actions du STE et de les affecter à des variables typées, qui peuvent être utilisées ensuite dans les formules temporelles.

Formules de μ-calcul étendues : les opérateurs modaux sont étendus avec des variables typées, permettant de récupérer les valeurs contenues dans les actions du STE ; ces variables peuvent être utilisées dans des prédicats et/ou passées en paramètre aux opérateurs de point fixe étendus. Ceci permet d'exprimer de façon naturelle des propriétés temporelles portant sur les valeurs (comme le fait que l'émission d'un message est suivie de la réception du même message). Des formalismes similaires ont été proposés en [Dam94b] pour les systèmes décrits en π-calcul polyadique et en [RH96] pour les systèmes de transitions symboliques.

Formules et expressions régulières sur actions : ces constructions, inspirées des logiques ACTL et PDL, sont étendues avec des variables typées permettant d'extraire et de propager les valeurs contenues dans les actions du STE. Utilisées dans les opérateurs modaux du μ-calcul, ces formules et expressions régulières autorisent une description concise et naturelle des propriétés portant sur des séquences d'actions du STE.

Opérateurs inspirés des langages fonctionnels : ces constructions usuelles ("let", "if-then-else", "case", "loop", etc.), permettent d'effectuer des calculs conditionnels ou répétitifs sur les valeurs extraites du STE. Des quantificateurs sur les variables à domaines finis sont aussi disponibles, facilitant l'écriture des prédicats du premier ordre.

Méta-opérateurs de manipulation des états et des actions : ces opérateurs augmentent la puissance du langage XTL, autorisant l'expression de propriétés difficiles, voire même impossibles à décrire dans une logique temporelle classique (un exemple de telle propriété est *"il existe un chemin d'exécution issu d'un état et menant au même état"*). Outre l'expressivité accrue, ces opérateurs permettent une définition naturelle des prédicats de base (formules ne contenant pas d'opérateurs temporels) portant sur les états et/ou les actions du STE.

Méta-opérateurs d'évaluation de formules temporelles : ces opérateurs constituent le "moteur" du langage XTL, car ils permettent de vérifier si un certain état (tous les états) satisfait (satisfont) une formule, ou de calculer l'ensemble d'états du STE satisfaisant une formule. Les résultats de l'évaluation des formules peuvent être utilisés dans d'autres calculs ou imprimés sur un fichier de sortie.

Méta-opérateurs d'exploration de la relation de transition : ces opérateurs, dont certains sont inspirés du calcul de Dicky, permettent d'accéder à l'état initial du STE, ainsi qu'aux successeurs et prédécesseurs des états et des transitions. Grâce à ces opérateurs, il est possible d'exprimer des propriétés temporelles non-standard, ainsi que de calculer différentes informations sur le STE (nombre d'états et de transitions, facteur de branchement, etc.).

Définitions de fonctions et de formules : ces mécanismes d'abstraction permettent de construire des bibliothèques réutilisables d'opérateurs de logique temporelle.

Après avoir défini formellement la syntaxe et la sémantique du langage XTL, nous avons abordé le problème de l'évaluation des formules temporelles XTL sur des modèles STE. Il existe, à l'heure actuelle, de nombreux algorithmes [EL86, AC88, Cle90, CS91b, SW91, Win91, Lar92, VL92, CKS92, Bra92, And92, VL94, LBC+94] pour l'évaluation des formules du μ-calcul standard. Du fait de la présence des valeurs, aucun de ces algorithmes n'est directement applicable aux formules XTL. C'est pourquoi nous avons conçu des algorithmes d'évaluation pour XTL qui généralisent les algorithmes dédiés au μ-calcul standard (voir la section suivante). Nos algorithmes permettent, dans le cas des formules sans valeurs, de retrouver la même complexité que celle des meilleurs algorithmes classiques.

Comme modèle de représentation des STEs, nous avons choisi le format BCG (*Binary Coded Graph*) [Gar94, Ruf94], qui est suffisamment général pour convenir à plusieurs langages de description (en particulier LOTOS et μCRL). Ce modèle permet l'exploration de la relation de transition du STE, ainsi que l'accès à toutes les informations intéressantes provenant du programme source à vérifier (notamment, les valeurs des variables, les types et les fonctions). Le fonctionnement de l'évaluateur XTL est illustré dans la figure 1. L'outil prend en entrée :

- un modèle STE généré par le compilateur CÆSAR [Gar89a, GS90] à partir d'un programme LOTOS (ou par un autre outil, à partir d'un autre langage)
- un programme XTL contenant les propriétés temporelles à vérifier

et produit en sortie les résultats de l'évaluation du programme XTL sur le modèle STE.

Plan du document

Le chapitre 1 contient une réflexion critique sur l'état de l'art dans le domaine de la vérification basée sur les modèles. Différents formalismes de spécification (logiques temporelles arborescentes et linéaires, μ-calcul modal) sont comparés suivant les classes de propriétés qu'ils peuvent exprimer (sûreté, vivacité, équité), leur adaptation aux divers types de programmes parallèles et l'efficacité des algorithmes de vérification qui leur sont associés. Cette discussion nous permet de choisir un modèle convenable de représentation des programmes et de dégager les principes de conception du langage XTL et de l'outil de vérification associé.

Le chapitre 2 constitue un tutoriel du langage XTL. Nous avons structuré la présentation de manière à faciliter l'apprentissage du langage : après une définition complète des éléments lexicaux et syntaxiques, les différentes constructions XTL sont décrites progressivement, en allant des plus simples (types, variables et expressions fonctionnelles) aux plus compliquées (formules et expressions régulières sur actions, formules sur états, méta-opérateurs d'évaluation). La sémantique statique et dynamique de chaque élément du langage est définie informellement, et son utilisation est illustrée à travers de nombreux exemples de propriétés temporelles.

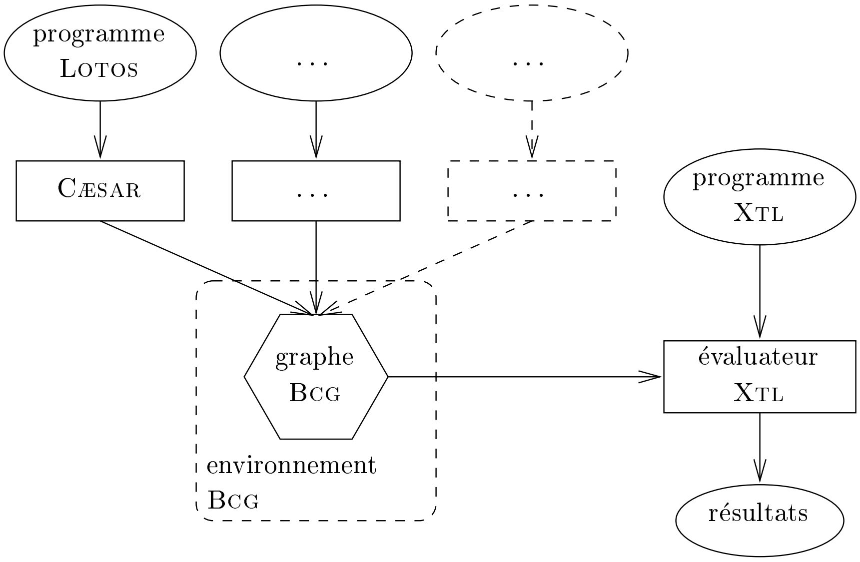

Figure 1: L'évaluateur Xtl

Le chapitre 3 définit la sémantique dénotationnelle des formules Xtl (sur actions et sur états), qui servira de base pour la présentation des algorithmes d'évaluation de ces formules sur un modèle Ste. La sémantique que nous proposons étend de manière naturelle, d'une part, la sémantique du μ-calcul standard et, d'autre part, la sémantique des fonctions récursives des langages de programmation (les opérateurs de point fixe étendus sont représentés comme des fonctions ayant des paramètres typés et renvoyant des ensembles d'états du Ste). Différentes transformations préliminaires (élimination des opérateurs dérivés, traduction des expressions régulières, transformation en forme normale positive) sont appliquées sur les formules afin de les traduire en une forme simplifiée, destinée à faciliter leur évaluation.

Le chapitre 4 est consacré à l'évaluation des formules Xtl sur un modèle Ste. Nous commençons par étudier le fragment des formules Xtl d'alternance 1 (c'est-à-dire, ne contenant pas d'opérateurs de plus petit et de plus grand point fixe mutuellement récursifs), l'alternance [EL86] étant une mesure du degré de récursion mutuelle des opérateurs de plus petit et de plus grand point fixe. Ce fragment est intéressant du point de vue pratique, car il est suffisamment expressif pour permettre la traduction de plusieurs logiques temporelles (comme Ctl, Actl ou Pdl-Δ), tout en étant susceptible de s'évaluer efficacement. Généralisant l'approche adoptée dans les meilleurs algorithmes dédiés au μ-calcul standard [AC88, CS91a, CS91b, VL92, VL94, And94], nous ramenons l'évaluation de formules Xtl d'alternance 1 à la résolution de systèmes d'équations booléennes paramétrées par des variables typées. Ensuite, nous proposons une méthode de résolution de ces systèmes qui, sous certaines conditions de terminaison, conduit à des algorithmes d'évaluation globale (sur un modèle Ste complètement construit) ou locale (en générant le Ste "à la volée", au fur et à mesure de l'évaluation).

Pour les formules Xtl d'alternance quelconque, nous proposons un algorithme itératif global qui généralise les algorithmes correspondants dédiés au μ-calcul standard [EL86, And92, LBC+94]. Notre algorithme permet, toujours sous certaines conditions de terminaison, de calculer itérativement l'ensemble des états du Ste satisfaisant une formule Xtl d'alternance quelconque.

Le chapitre 5 décrit la réalisation d'une première version de l'évaluateur XTL et illustre son utilité pour la validation d'applications réelles, notamment le protocole BRP de Philips [Mat96] et le protocole de la couche liaison du bus série P-1394 ("FireWire") normalisé par l'IEEE [SM97]. Ces expériences ont permis de confirmer le bon fonctionnement de notre outil et ont mis en évidence les avantages de notre approche : les propriétés temporelles comportant des valeurs ont une utilité pratique indiscutable, tout en étant évaluées avec des performances comparables aux autres évaluateurs existants.

L'annexe A définit formellement la syntaxe concrète et la sémantique statique du langage XTL. En dépit de son caractère technique, cette annexe nous semble intéressante, car elle illustre bien les difficultés soulevées par le caractère original du langage. En effet, la syntaxe doit offrir, d'une part, les notations mathématiques employées dans la littérature pour les formules de μ-calcul modal et, d'autre part, les constructions couramment utilisées dans les langages de programmation fonctionnels. En outre, un niveau de complexité sémantique supplémentaire est ajouté par la possibilité d'accéder aux objets définis dans le programme source à vérifier (donc externes aux programmes XTL). Les différentes phases d'analyse (liaison des types, des variables simples, des variables propositionnelles et des fonctions, typage des expressions et des formules) sont décrites à l'aide de grammaires attribuées.

L'annexe B définit la sémantique dénotationnelle des expressions XTL. Cette annexe, avec la sémantique des formules décrite au chapitre 3, constitue une définition complète de la sémantique de XTL, pouvant servir de base au développement de compilateurs, ainsi qu'aux extensions futures du langage.

L'annexe C contient les traductions en XTL de plusieurs logiques temporelles comme CTL, ACTL et (un fragment "purement" arborescent de) la logique modale de μCRL. Plusieurs opérateurs temporels dérivés de la logique LTAC et du calcul de Dicky, couramment employés dans les applications, sont également traduits en XTL.

Notations

Nous présentons ici les notations mathématiques utilisées dans les chapitres suivants du document. Nous employons les abréviations suivantes :

- "$\overset{\mathrm{d}}{=}$" signifie "égal par définition" ;
- "ssi" signifie "si et seulement si".

Ensembles

Soient les ensembles E, E_1 et E_2. Nous utilisons les notations ensemblistes classiques suivantes :

- $\emptyset$ dénote l'*ensemble vide* ;
- $E_1 \cup E_2$ dénote l'*union* de E_1 et E_2 ;
- $E_1 \cap E_2$ dénote l'*intersection* de E_1 et E_2 ;
- $E_1 \setminus E_2$ dénote la *différence* de E_1 et E_2 ;
- $E_1 \subseteq E_2$ dénote l'*inclusion* de E_1 dans E_2 ;
- $E_1 \times E_2$ dénote le *produit cartésien* de E_1 et E_2 ;
- $|E|$ dénote le *cardinal* de E ;
- 2^E dénote l'ensemble des parties de E ;
- E^* dénote l'ensemble des séquences ayant zéro ou plusieurs éléments de E ;
- E^+ dénote l'ensemble des séquences ayant un ou plusieurs éléments de E.

Tuples et projections

Les éléments d'un produit cartésien $E_1 \times \ldots \times E_n$ sont appelés *tuples* et sont notés $(x_1, \ldots, x_n)$. La *projection* d'un tuple $(x_1, \ldots, x_n)$ sur un ensemble d'indices $I = \{i_1, \ldots, i_p\} \subseteq \{1, \ldots, n\}$, notée $(x_1, \ldots, x_n)_I$, est un tuple défini comme suit :

$$(x_1, \ldots, x_n)_I \overset{\mathrm{d}}{=} (x_{i_1}, \ldots, x_{i_p})$$

où $i_1 < \ldots < i_p$.

Relations

Une *relation* (n-aire) $\mathcal{R}$ sur des ensembles $E_1, \ldots, E_n$ est un sous-ensemble du produit cartésien de $E_1, \ldots, E_n$:

$$\mathcal{R} \subseteq E_1 \times \ldots \times E_n$$

La projection est étendue naturellement aux relations. Pour une relation $\mathcal{R} \subseteq E_1 \times \ldots \times E_n$ et un ensemble d'indices $I = \{i_1, \ldots, i_p\} \subseteq \{1, \ldots, n\}$, la projection de $\mathcal{R}$ sur I, notée $\mathcal{R}_I$, est la relation définie comme suit :

$$\mathcal{R}_I \stackrel{\mathrm{d}}{=} \{(x_1, \ldots, x_n)_I \mid (x_1, \ldots, x_n) \in \mathcal{R}\}$$

Soient $\mathcal{R}, \mathcal{R}_1, \mathcal{R}_2 \subseteq E \times E$ des relations binaires sur E. Pour tous éléments $x, y \in E$, le prédicat $(x, y) \in \mathcal{R}$ est noté aussi $x \ \mathcal{R} \ y$. Nous utilisons les opérations classiques suivantes :

- $\mathcal{R}_1 \cup \mathcal{R}_2$ dénote l'*union* de $\mathcal{R}_1$ et $\mathcal{R}_2$;
- $\mathcal{R}_1 \circ \mathcal{R}_2$ dénote la *composition* de $\mathcal{R}_1$ et $\mathcal{R}_2$, définie par :

$$\mathcal{R}_1 \circ \mathcal{R}_2 \stackrel{\mathrm{d}}{=} \{(x_1, x_2) \in E \times E \mid \exists x \in E. x_1 \ \mathcal{R}_1 \ x \ \wedge \ x \ \mathcal{R}_2 \ x_2\}$$

- $\mathcal{R}^k$ dénote la *puissance* de $\mathcal{R}$, définie par :

$$\mathcal{R}^{k+1} \stackrel{\mathrm{d}}{=} \mathcal{R} \circ \mathcal{R}^k, \ \mathcal{R}^0 \stackrel{\mathrm{d}}{=} Id_E$$

  où $Id_E \stackrel{\mathrm{d}}{=} \{(x, x) \mid x \in E\}$ est la *relation d'identité* sur E ;
- $\mathcal{R}^*$ dénote la fermeture transitive et réflexive de $\mathcal{R}$, définie par :

$$\mathcal{R}^* \stackrel{\mathrm{d}}{=} \bigcup_{k \geq 0} \mathcal{R}^k$$

- $\mathcal{R}^+$ dénote la fermeture transitive de $\mathcal{R}$, définie par :

$$\mathcal{R}^+ \stackrel{\mathrm{d}}{=} \bigcup_{k > 0} \mathcal{R}^k.$$

Une relation binaire $\mathcal{R} \subseteq E \times E$ est une *relation d'ordre* ssi elle satisfait les propriétés suivantes :

- $x \ \mathcal{R} \ x$ (réflexivité)
- $x \ \mathcal{R} \ y \ \wedge \ y \ \mathcal{R} \ x \Rightarrow x = y$ (antisymétrie)
- $x \ \mathcal{R} \ y \ \wedge \ y \ \mathcal{R} \ z \Rightarrow x \ \mathcal{R} \ z$ (transitivité)

pour tous $x, y, z \in E$.

Soit un ensemble E muni d'une relation d'ordre $\leq$ et soit $E' \subseteq E$.

- la *borne inférieure* de E' (si elle existe), notée $\sqcap E'$, est un élément $z \in E$ tel que :

$$(\forall x \in E'. z \leq x) \text{ et } (\forall y \in E. (\forall x \in E'. y \leq x) \Rightarrow y \leq z)$$

- la *borne supérieure* de E' (si elle existe), notée $\sqcup E'$, est un élément $z \in E$ tel que :

$$(\forall x \in E'. x \leq z) \text{ et } (\forall y \in E. (\forall x \in E'. x \leq y) \Rightarrow z \leq y)$$

Fonctions partielles

Soient E_1 et E_2 deux ensembles. Une *fonction partielle* $e : E_1 \to E_2$ est une application qui associe à chaque élément x d'un sous-ensemble $E \subseteq E_1$ un élément $y \in E_2$. Le sous-ensemble $E \subseteq E_1$ sur lequel e est définie est appelé *support* de e et est noté $supp(e)$.

Une fonction partielle $e : E_1 \to E_2$ ayant le support $supp(e) = \{x_1, \ldots, x_n\}$ et associant un élément $y_i \in E_2$ à chaque élément x_i (pour $0 \leq i \leq n$) est notée $[y_1/x_1, \ldots, y_n/x_n]$. Une fonction partielle ayant un support vide est notée $[\,]$.

La *restriction* d'une fonction partielle $e : E_1 \to E_2$ à un sous-ensemble $E \subseteq E_1$, notée $e|_E$, est une fonction partielle $e|_E : E_1 \to E_2$ définie comme suit :

$$(e|_E)\,(x) \stackrel{\mathrm{d}}{=} e(x)$$

pour tout $x \in supp(e) \cap E$. Pour toute fonction partielle $e : E_1 \to E_2$ et pour tous ensembles $E, E', E'' \subseteq E_1$, l'opérateur de restriction a les propriétés suivantes :

- $supp(e|_E) = supp(e) \cap E$
- $e|_E = e|_{supp(e) \cap E}$
- $(e|_{E'})|_{E''} = (e|_{E''})|_{E'} = e|_{E' \cap E''}$

Soient deux fonctions partielles $e_1 : E_1 \to E_2$ et $e_2 : E_1 \to E_2$ telles que $supp(e_1) \cap supp(e_2) = \emptyset$. La *somme* de e_1 et e_2, notée $e_1 \oplus e_2$, est une fonction partielle définie comme suit :

$$(e_1 \oplus e_2)(x) \stackrel{\mathrm{d}}{=} \begin{cases} e_1(x) \text{ si } x \in supp(e_1) \\ e_2(x) \text{ si } x \in supp(e_2) \end{cases}$$

Pour toutes fonctions partielles $e_1 : E_1 \to E_2$, $e_2 : E_1 \to E_2$ et $e_3 : E_1 \to E_2$ ayant des supports deux à deux disjoints, l'opérateur $\oplus$ a les propriétés suivantes :

- $supp(e_1 \oplus e_2) = supp(e_1) \cup supp(e_2)$
- $e_1 \oplus e_2 = e_2 \oplus e_1$ (commutativité)
- $e_1 \oplus (e_2 \oplus e_3) = (e_1 \oplus e_2) \oplus e_3$ (associativité)
- $(e_1 \oplus e_2)|_E = (e_1|_E) \oplus (e_2|_E)$.

L'*extension* de deux fonctions partielles $e_1 : E_1 \to E_2$ et $e_2 : E_1 \to E_2$, notée $e_1 \oslash e_2$, est une fonction partielle définie comme suit :

$$(e_1 \oslash e_2)(x) \stackrel{\mathrm{d}}{=} \begin{cases} e_1(x) \text{ si } x \in supp(e_1) \setminus supp(e_2) \\ e_2(x) \text{ si } x \in supp(e_2) \end{cases}$$

Pour toutes fonctions partielles $e_1 : E_1 \to E_2$, $e_2 : E_1 \to E_2$ et $e_3 : E_1 \to E_2$ et pour tout ensemble $E \subseteq E_1$, l'opérateur $\oslash$ a les propriétés suivantes :

- $supp(e_1 \oslash e_2) = supp(e_1) \cup supp(e_2)$
- $e_1 \oslash (e_2 \oslash e_3) = (e_1 \oslash e_2) \oslash e_3$ (associativité)
- $e_1 \oslash e_2 = e_1|_{supp(e_1) \setminus supp(e_2)} \oplus e_2$
- $(e_1 \oslash e_2)|_E = (e_1|_E) \oslash (e_2|_E)$.

Treillis

Soit un ensemble E muni d'une relation d'ordre $\sqsubseteq$. Le quadruplet $\langle E, \sqcup, \sqcap, \sqsubseteq \rangle$ est un *treillis complet* ssi tout sous-ensemble $E' \subseteq E$ admet une borne inférieure $\sqcap E'$ et une borne supérieure $\sqcup E'$. Tout treillis complet contient un plus petit élément, noté $\bot$, égal à $\sqcap E$ et un plus grand élément, noté $\top$, égal à $\sqcup E$.

Soient $\langle E_1, \sqcup_1, \sqcap_1, \sqsubseteq_1 \rangle$ et $\langle E_2, \sqcup_2, \sqcap_2, \sqsubseteq_2 \rangle$ deux treillis complets. Une fonction $f : E_1 \to E_2$ est dite *monotone* ssi, pour tous $x, y \in E_1$:

$$x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)$$

Soit un treillis complet $\langle E, \sqcup, \sqcap, \sqsubseteq \rangle$ et une fonction monotone $f : E \to E$. Alors [Tar55] :

- il existe un élément unique $\mu f \in E$, appelé le *plus petit point fixe* de f, égal à :

  $$\mu f \stackrel{\mathrm{d}}{=} \bigsqcap \{x \in E \mid f(x) \sqsubseteq x\}$$

  tel que :

  $$f(\mu f) = \mu f \text{ et } (\forall z \in E.f(z) = z \Rightarrow \mu f \sqsubseteq z)$$

- il existe un élément unique $\nu f \in E$, appelé le *plus grand point fixe* de f, égal à :

  $$\nu f \stackrel{\mathrm{d}}{=} \bigsqcup \{x \in E \mid x \sqsubseteq f(x)\}$$

  tel que :

  $$f(\nu f) = \nu f \text{ et } (\forall z \in E.f(z) = z \Rightarrow z \sqsubseteq \nu f)$$

Soit un treillis complet $\langle E, \sqcup, \sqcap, \sqsubseteq \rangle$ et une fonction $f : E \to E$.

- f est dite $\sqcap$-*continue* ssi, pour toute suite décroissante $(x_i)_{i \geq 0}$ d'éléments de E :

  $$f\left(\bigsqcap \{x_i \mid i \geq 0\}\right) = \bigsqcap \{f(x_i) \mid i \geq 0\}$$

- f est dite $\sqcup$-*continue* ssi, pour toute suite croissante $(x_i)_{i \geq 0}$ d'éléments de E :

  $$f\left(\bigsqcup \{x_i \mid i \geq 0\}\right) = \bigsqcup \{f(x_i) \mid i \geq 0\}$$

Toute fonction $\sqcap$-continue ou $\sqcup$-continue est monotone. De plus [Kle52] :

- si f est $\sqcup$-continue, le plus petit point fixe de f a la caractérisation itérative suivante :

  $$\mu f = \bigsqcup \{f^n(\bot) \mid n \geq 0\}$$

- si f est $\sqcap$-continue, le plus grand point fixe de f a la caractérisation itérative suivante :

  $$\nu f = \bigsqcap \{f^n(\top) \mid n \geq 0\}.$$

Chapitre 1

Vérification par évaluation de formules logiques

Les dernières années ont été très riches en résultats concernant la vérification des programmes parallèles. Différentes théories et méthodes ont été définies et étudiées, et de nombreux algorithmes et outils ont été développés.

Ce chapitre contient une synthèse des principaux résultats concernant l'expression des propriétés temporelles et leur vérification sur des modèles finis générés à partir de programmes parallèles. Nous commençons par examiner les différents modèles de programmes parallèles utilisés dans la littérature, ce qui nous permet de fixer un modèle suffisamment général convenant à notre étude. Ensuite, nous considérons l'expression des propriétés temporelles des programmes parallèles : plusieurs logiques temporelles représentatives (linéaires et arborescentes, interprétées sur états ou sur actions, etc.) sont présentées et comparées suivant leur capacité à décrire diverses propriétés intéressantes, ainsi que suivant l'efficacité de leur évaluation sur les modèles. Finalement, nous explorons les résultats concernant l'évaluation des formules de logique temporelle sur des modèles finis. Les différentes classes d'algorithmes d'évaluation sont décrites et comparées selon leurs performances et leur généralité.

Compte tenu de la quantité de résultats existants dans le domaine, cette synthèse n'est pas exhaustive ; néanmoins, les modèles, les formalismes et les algorithmes présentés sont représentatifs des grandes approches et illustrent bien les aspects essentiels de notre étude. Nous indiquons également les références d'articles de synthèse et d'ouvrages traitant plus en détail certains points particuliers.

1.1 Modèles des programmes parallèles

Nous examinons deux modèles classiques des programmes concurrents : les structures de Kripke et les systèmes de transitions étiquetées. Nous proposons ensuite un modèle étendu, adapté à plusieurs classes de langages de description, qui sera utilisé comme base dans la suite du document.

1.1.1 Sémantique de l'entrelacement

Les applications réparties (telles que les protocoles de communication et les systèmes distribués) peuvent être représentées comme des *systèmes asynchrones* composés de plusieurs entités ("boîtes noires") qui évoluent de manière concurrente et communiquent par échange de messages. Le comportement d'un tel système est décrit par les *actions* (ou *événements*) qu'il effectue au cours de son exécution.

Habituellement, l'exécution des actions dans un système asynchrone est modélisée (à un certain niveau d'abstraction) en faisant les hypothèses suivantes :

Atomicité : chaque action est *atomique*, au sens où elle ne peut pas être décomposée en actions plus élémentaires ;

Non-déterminisme : il n'est pas possible d'observer simultanément l'exécution de plusieurs actions différentes.

Ces hypothèses constituent la base de la sémantique d'entrelacement (*interleaving*) : le comportement d'un système asynchrone est représenté comme l'entrelacement des comportements de ses sous-systèmes. Par exemple, si deux actions a_1 et a_2 sont simultanément exécutables, le comportement du système contiendra les deux séquences a_1a_2 et a_2a_1 produites par l'entrelacement de a_1 et a_2 (voir figure 1.1). Ceci justifie le terme "sémantique du losange" utilisé aussi pour désigner la sémantique d'entrelacement.

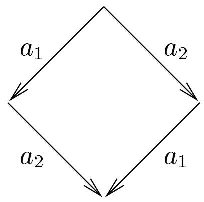

Figure 1.1: Entrelacement des actions a_1 et a_2

La sémantique d'entrelacement sert de base aux algèbres de processus comme Ccs [Mil89], Csp [BHR84] ou Acp [BK85], ainsi qu'aux langages de description qui en sont inspirés, comme Lotos [ISO88b] ou μCrl [GP90]. Elle sert également de base aux formalismes de description d'automates communicants, tels que le modèle d'Arnold-Nivat utilisé dans l'outil Mec [ABC94] ou le modèle Fc2 utilisé dans la boîte à outils Fc2Tools [BRRd96]. Au-delà des langages basés sur la communication par rendez-vous, cette sémantique est aussi utilisée pour définir des langages de programmation parallèle comme Sr [And91], qui contiennent des mécanismes de communication par variables partagées.

Il existe d'autres sémantiques du parallélisme, notamment celles utilisées dans les langages *synchrones*, où plusieurs actions peuvent se produire en même temps. Dans cette étude, nous ne considérons que des langages de description ayant une sémantique d'entrelacement.

1.1.2 Structures de Kripke et systèmes de transitions étiquetées

Conformément à la sémantique d'entrelacement, le comportement d'un programme parallèle peut être naturellement modélisé par un graphe (ou automate) dont les sommets représentent les états du programme et les arcs (ou transitions) dénotent les changements d'état du programme suite aux actions qu'il effectue durant son exécution.

Il existe généralement deux classes de modèles de programmes parallèles, employés suivant le fait que les informations pertinentes sont attachées aux états (*resp.* aux actions) du programme : les structures de Kripke (*resp.* les systèmes de transitions étiquetées, ou Stes). Les définitions formelles de ces modèles sont données ci-dessous.

Définition 1-1 (Structure de Kripke)
Une *structure de Kripke* est un quadruplet $\mathcal{K} = (S, P, L, T)$, où :

- S est un ensemble d'*états*, notés s_1, s_2, ... ;
- P est un ensemble de *propositions atomiques*, notées p_1, p_2, ... ;
- $L : S \to 2^P$ est un *étiquetage*, associant à chaque état s l'ensemble des propositions atomiques satisfaites par s ;
- $T \subseteq S \times S$ est la *relation de transition*. Les éléments $(s_1, s_2) \in T$ (notés aussi $s_1 \to s_2$) sont appelés *transitions*.

Un *chemin* π est une séquence (finie ou infinie) $s_0 \to s_1 \to \cdots$ d'états. Le k-ième état de π est noté $\pi(k)$ et le k-ième suffixe $(s_k \to s_{k+1} \to \cdots)$ de π est noté π^k. Un chemin π est dit *maximal* soit s'il est infini, soit s'il est fini et que son dernier état s_n n'a pas de successeur (c'est-à-dire qu'il n'existe pas d'état $s \in S$ tel que $s_n \to s$). L'ensemble des chemins maximaux d'une structure de Kripke est noté R. ■

Définition 1-2 (Système de transitions étiquetées)
Un *système de transitions étiquetées* est un triplet $\mathcal{A} = (S, A, T)$, où :

- S est un ensemble d'*états*, notés s_1, s_2, ... ;
- A est un ensemble d'*actions*, notées a_1, a_2, ... Il existe une action particulière, notée τ, appelée action *invisible* ou *interne* ;
- $T \subseteq S \times A \times S$ est la *relation de transition*. Les éléments $(s_1, a, s_2) \in T$ (notés aussi $s_1 \xrightarrow{a} s_2$) sont appelés *transitions*.

Un *chemin* π est une séquence (finie ou infinie) $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ d'états. Le k-ième état de π est noté $\pi(k)$, la k-ième action de π est notée $l(\pi, k)$ et le k-ième suffixe $(s_k \to s_{k+1} \to \cdots)$ de π est noté π^k. Un chemin π est dit *maximal* soit s'il est infini, soit s'il est fini et que son dernier état s_n n'a pas de successeur (c'est-à-dire qu'il n'existe pas d'état $s \in S$, ni d'action $a \in A$ tels que $s_n \xrightarrow{a} s$). L'ensemble des chemins maximaux d'un STE est noté R. ■

Les structures de Kripke n'attachent aucune information aux actions effectuées par le programme ; elles sont utilisées comme modèle d'interprétation pour des logiques temporelles comme PTL, CTL, ECTL, CTL*, etc., qui permettent d'exprimer des propriétés portant sur les états. Ces modèles conviennent aux langages de description comme ESTELLE [ISO88a], SDL [IT92] ou PROMELA [Hol91].

En revanche, les systèmes de transitions étiquetées n'attachent aucune information aux états du programme ; ils sont utilisés comme modèle d'interprétation pour des logiques modales et temporelles comme HML, ACTL, ACTL*, etc., qui permettent d'exprimer des propriétés portant sur les actions. Ces modèles conviennent aux algèbres de processus comme CCS, CSP ou ACP, ainsi qu'aux langages de description basés sur ces algèbres, comme LOTOS ou μCRL.

1.1.3 Systèmes de transitions étendus

Certaines logiques temporelles comme LTAC, PDL, la logique de Dicky ou le μ-calcul modal, autorisent l'expression des propriétés portant sur les états aussi bien que sur les actions des programmes. Ces logiques sont naturellement interprétées sur des modèles ayant des informations sur les états et sur les actions. Nous donnons ci-dessous la définition d'un tel modèle, que nous appellerons *système de transitions mixte* (STM). Ce modèle nous sera utile dans la suite de ce chapitre, notamment pour définir la sémantique des logiques mentionnées ci-dessus (voir la section 1.2.3).

Définition 1-3 (Système de transitions mixte)
Un système de transitions mixte est un tuple $\mathcal{L} = (S, P, L, A, T)$, où :

- S est un ensemble d'*états*, notés s_1, s_2, ... ;
- P est un ensemble de *propositions atomiques*, notées p_1, p_2, ... ;
- $L : S \to 2^P$ est un *étiquetage*, associant à chaque état s l'ensemble des propositions atomiques satisfaites par s ;
- A est un ensemble d'*actions*, notées a_1, a_2, ... (l'action *invisible* est notée τ) ;
- $T \subseteq S \times A \times S$ est la *relation de transition*. Les éléments $(s_1, a, s_2) \in T$ (notés aussi $s_1 \xrightarrow{a} s_2$) sont appelés *transitions*.

Les chemins maximaux d'un STM sont définis de la même manière que pour les STEs. ■

Il existe d'autres modèles contenant des informations dans les états et les actions, comme les systèmes de transitions paramétrés d'Arnold-Nivat [Arn92, ABC94] ou les structures multi-processus [EC82], que nous ne considérerons pas ici.

Les STMs sont des modèles théoriques abstraits permettant de représenter les aspects essentiels du comportement des programmes parallèles. Cependant, les modèles concrets générés par les compilateurs des langages de haut niveau contiennent des informations concernant les valeurs manipulées dans les programmes, qu'il convient de représenter explicitement.

Dans cette étude, nous adoptons un modèle de représentation des programmes parallèles qui est suffisamment général pour convenir à plusieurs langages de description (en particulier, à LOTOS). Par souci de compatibilité avec la terminologie utilisée pour LOTOS et les autres algèbres de processus, nous désignerons ce modèle par le terme *système de transitions étiquetées étendu* (STE étendu).

Quelques notions auxiliaires sont nécessaires. Considérons l'ensemble $SVar = \{x_1, \ldots, x_n\}$ des *variables d'état* du programme. Chaque variable x_i a un type T_i. L'ensemble des valeurs appartenant au type T_i est noté $\mathbf{Val}_i$ et l'ensemble $\mathbf{Val} = \bigcup_{i=1}^{n} \mathbf{Val}_i$ contient toutes les valeurs appartenant aux types T_i. Le modèle STE étendu est formellement défini comme suit.

Définition 1-4 (Système de transitions étiquetées étendu)
Un système de transitions étiquetées étendu est un tuple $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$, où :

- S est un ensemble d'*états*, notés s_1, s_2, ... ;
- $val_S : S \to (SVar \to \mathbf{Val})$ est un *étiquetage des états*. Pour chaque état $s \in S$ et chaque variable $x_i \in SVar$, $(val_S(s))(x_i)$ représente la valeur de x_i dans l'état s ;
- A est un ensemble d'*actions*, notées a_1, a_2, ... ;
- $val_A : A \to \mathbf{Val}^+$ est un *étiquetage des actions*. Pour chaque action $a \in A$, $val_A(a)$ représente la liste (non vide) des valeurs contenues dans a ;
- $T \subseteq S \times A \times S$ est la *relation de transition*. Les éléments $(s_1, a, s_2) \in T$ (notés aussi $s_1 \xrightarrow{a} s_2$) sont appelés *transitions* ;
- $s_{init} \in S$ est l'*état initial*.

Les chemins maximaux d'un STE étendu sont définis de la même manière que pour les STEs. ■

Intuitivement, un état s du programme est représenté par le tuple des valeurs courantes des variables $x_1, \ldots, x_n$ (aussi appelé *vecteur d'état*). Une action a est caractérisée par les valeurs $v_0, \ldots, v_p$ qu'elle contient. Dans les modèles STE étendus générés à partir des programmes LOTOS, v_0 dénote une *porte* (canal de communication) et $v_1, \ldots, v_p$ représentent les valeurs échangées par rendez-vous lors de l'exécution de a. L'état initial s_{init} du programme est précisé explicitement, afin de faciliter l'expression de certaines propriétés temporelles (en particulier, les propriétés portant sur le passé). Dans cette étude, nous ne considérons que des modèles STEs étendus finis, c'est-à-dire ayant un nombre fini d'états et d'actions.

Les STMs sont des cas particuliers de STEs étendus, dans lesquels les actions ne sont pas structurées et les propositions atomiques sont des représentations symboliques de prédicats portant sur les variables d'état.

Un évaluateur capable d'interpréter des propriétés temporelles sur un modèle STE étendu généré à partir d'un programme parallèle doit naturellement reposer sur une implémentation de ce modèle. La boîte à outils CADP implémente deux représentations différentes du modèle STE étendu :

Représentation explicite : il s'agit du format BCG (*Binary Coded Graph*) [Gar94] qui permet un stockage compact des STEs. Un fichier en format BCG contient les états, les actions et la relation de transition du STE, ainsi que diverses autres informations provenant du programme source à partir duquel il a été généré (types, fonctions, etc.). Des primitives spéciales permettent d'accéder aux valeurs contenues dans les états et dans les étiquettes des transitions, ainsi que d'explorer la relation de transition du STE (accès à l'état initial, aux successeurs et prédécesseurs des états et des transitions).

Les fichiers BCG peuvent être traduits vers d'autres formats de STEs (en particulier MEC et FC2) à l'aide de l'outil `bcg_io`, ainsi que visualisés en PostScript au moyen de l'outil `bcg_draw`[3] (voir la figure 1.2).

Représentation implicite : il s'agit de l'outil OPEN/CÆSAR [Gar98] qui permet la génération "à la volée" (*on-the-fly*) des STEs à partir de programmes parallèles. Cet outil fournit des primitives d'accès à l'état initial et aux transitions successeurs d'un état donné, permettant l'exploration efficace des STEs. OPEN/CÆSAR est un environnement extensible, offrant des bibliothèques avec interfaces standard, qui peuvent être utilisées pour développer des outils spécialisés de simulation, exécution, recherche de séquences, détection de blocages, etc.

Dans la version actuelle d'OPEN/CÆSAR, les contenus des états et des actions du STE sont représentés comme des chaînes de caractères ASCII, ce qui limite les possibilités de manipulation des valeurs.

1.2 Expression des propriétés temporelles

Dans l'approche de vérification sur les modèles que nous considérons ici, les propriétés attendues du programme sont exprimées comme formules de logique temporelle : il s'agit de *spécifications logiques* du programme à vérifier. Comme nous avons précisé dans l'introduction du document, les spécifications logiques bénéficient des avantages de la modularité et de l'abstraction [MP90].

Cependant, il existe aussi d'autres approches pour exprimer les propriétés à vérifier, notamment les *spécifications comportementales*, qui consistent à caractériser le comportement attendu d'un programme P_1 sous forme d'un autre programme P_2, plus "abstrait" (dans le domaine des protocoles de communication, P_1 et P_2 sont appelés souvent *protocole* et *service*, respectivement). Le fait que P_1 ait le même comportement que P_2 est vérifié en comparant les modèles STEs M_1 et M_2 des deux programmes selon des relations d'équivalence ou de préordre, comme la bisimulation forte [Par81], l'équivalence observationnelle [Mil89], l'équivalence de branchement [vGW89], etc.

[3] Les algorithmes heuristiques de placement des états et des étiquettes ont été conçus et implémentés par l'auteur.

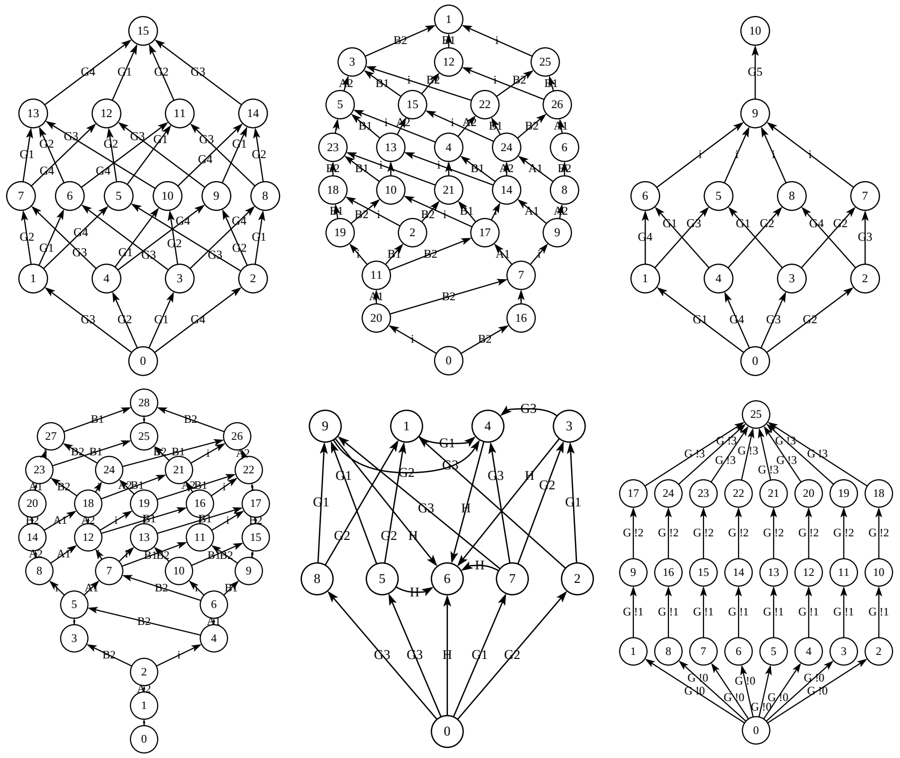

Figure 1.2: Exemples de STEs produits à partir de programmes LOTOS

Néanmoins, les spécifications logiques permettent une meilleure caractérisation des propriétés globales des programmes (par exemple, l'exclusion mutuelle entre plusieurs processus). En outre, il est souvent plus aisé de traduire les propriétés attendues (exprimées informellement en langage naturel) sous forme de formules de logique temporelle que sous forme de programmes.

Nous pouvons mentionner aussi une troisième approche, qui combine la spécification comportementale et la vérification des propriétés logiques : pour vérifier qu'un programme P_1 est (fortement) équivalent à un programme P_2, il suffit de générer à partir de P_1 une *formule caractéristique* φ_1 [IS94], c'est-à-dire une formule qui est satisfaite par tout programme (fortement) équivalent à P_1, et de la vérifier sur le modèle STE M_2 généré à partir de P_2.

Il existe une multitude de logiques temporelles qui ont été définies et étudiées pour spécifier et vérifier les propriétés des programmes parallèles. Nous présentons ici une synthèse des principaux résultats existants, en mettant l'accent sur l'expressivité comparée des diverses logiques temporelles, sur leur adaptation aux différents types de programmes parallèles, ainsi que sur la complexité de leur évaluation sur des modèles finis. La sémantique des logiques temporelles présentées est définie sous forme dénotationnelle, ce qui permet d'avoir une description homogène et unifiée de ces formalismes tout en soulignant leurs différences respectives.

1.2.1 Logiques temporelles basées sur états

Les logiques temporelles permettant d'exprimer des propriétés sur les états des programmes ont été intensivement étudiées dans la littérature. Ces logiques, interprétées sur des structures de Kripke

$\mathcal{K} = (S, P, L, T)$ (voir la définition 1-1), sont généralement partagées en deux classes :

logiques du temps linéaire comme LTL [Lam80] et PTL [MP92], permettant d'exprimer des propriétés portant sur les chemins individuels (issus de l'état initial) du programme ;

logiques du temps arborescent comme CTL et CTL* [CES86, EH86], permettant d'exprimer des propriétés portant sur les arbres d'exécution (issus de l'état initial) du programme.

Dans les paragraphes suivants, nous donnons les définitions des logiques temporelles linéaires et arborescentes représentatives, accompagnées d'exemples de propriétés qu'elles permettent d'exprimer. Ensuite, nous présentons une comparaison entre les deux classes de logiques, du point de vue de l'expressivité, de l'efficacité d'évaluation et de l'adaptation pour la spécification des programmes parallèles. Finalement, nous indiquons diverses extensions de ces logiques qui ont été proposées dans la littérature.

Logiques linéaires et arborescentes La logique temporelle CTL* permet d'exprimer des propriétés sur les chemins aussi bien que sur les arbres d'exécution des programmes. Suivant la présentation synthétique donnée en [ES89], nous donnons la définition de CTL* et nous précisons ensuite ses fragments "purement" linéaires et arborescents.

Définition 1-5 (Syntaxe et sémantique de CTL*)
La logique temporelle CTL* contient des formules sur états $\varphi \in \mathit{SForm}$ et des formules sur chemins $\psi \in \mathit{PForm}$, ayant la syntaxe suivante :

$$\begin{array}{rcl} \varphi & ::= & p \mid \mathit{true} \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{E}\psi \\ \psi & ::= & \varphi \mid \neg\psi_1 \mid \psi_1 \wedge \psi_2 \mid \mathbf{X}\psi_1 \mid \psi_1 \ \mathbf{U}\ \psi_2 \end{array}$$

où $p \in P$ est une proposition atomique, $\mathbf{E}$ dénote le quantificateur existentiel sur les chemins, $\mathbf{X}$ est l'opérateur "neXt" et $\mathbf{U}$ est l'opérateur "Until". Les connecteurs booléens dérivés sont définis de façon habituelle : $\mathit{false} \overset{\mathrm{d}}{=} \neg \mathit{true}$, $\varphi_1 \vee \varphi_2 \overset{\mathrm{d}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \overset{\mathrm{d}}{=} \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \Leftrightarrow \varphi_2 \overset{\mathrm{d}}{=} (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$ (les mêmes identités sont valables pour les formules ψ). Les modalités temporelles dérivées sont définies comme suit : $\mathbf{A}\psi \overset{\mathrm{d}}{=} \neg\mathbf{E}\neg\psi$, $\mathbf{F}\psi \overset{\mathrm{d}}{=} \mathit{true}\ \mathbf{U}\ \psi$, $\mathbf{G}\psi \overset{\mathrm{d}}{=} \neg\mathbf{F}\neg\psi$, $\overset{\infty}{\mathbf{F}}\psi \overset{\mathrm{d}}{=} \mathbf{GF}\psi$, $\overset{\infty}{\mathbf{G}}\psi \overset{\mathrm{d}}{=} \mathbf{FG}\psi$.

La sémantique des formules sur états (*resp.* sur chemins) est définie au moyen d'une fonction d'interprétation $[\![.]\!] : \mathit{SForm} \to 2^S$ (*resp.* $\|.\| : \mathit{PForm} \to 2^R$). Pour une formule φ (*resp.* ψ), la dénotation $[\![\varphi]\!]$ (*resp.* $\|\psi\|$) renvoie l'ensemble d'états (*resp.* de chemins maximaux) du modèle qui satisfont φ (*resp.* ψ). Les fonctions sémantiques sont définies inductivement comme suit :

$$\begin{array}{rcl} [\![p]\!] & \overset{\mathrm{d}}{=} & \{s \in S \mid p \in L(s)\} \\ [\![\mathit{true}]\!] & \overset{\mathrm{d}}{=} & S \\ [\![\neg\varphi_1]\!] & \overset{\mathrm{d}}{=} & S \setminus [\![\varphi_1]\!] \\ [\![\varphi_1 \wedge \varphi_2]\!] & \overset{\mathrm{d}}{=} & [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\ [\![\mathbf{E}\psi]\!] & \overset{\mathrm{d}}{=} & \{s \in S \mid \exists \pi \in R. \pi(0) = s \wedge \pi \in \|\psi\|\} \end{array}$$

et

$$\begin{array}{rcl} \|\varphi\| & \overset{\mathrm{d}}{=} & \{\pi \in R \mid \pi(0) \in [\![\varphi]\!]\} \\ \|\neg\psi_1\| & \overset{\mathrm{d}}{=} & R \setminus \|\psi_1\| \\ \|\psi_1 \wedge \psi_2\| & \overset{\mathrm{d}}{=} & \|\psi_1\| \cap \|\psi_2\| \end{array}$$

$$\begin{aligned}
\|\mathbf{X}\psi_1\| &\stackrel{\mathrm{d}}{=} \{\pi \in R \mid \pi^1 \in \|\psi_1\|\} \\
\|\psi_1 \ \mathbf{U} \ \psi_2\| &\stackrel{\mathrm{d}}{=} \{\pi \in R \mid \exists k \geq 0.\pi^k \in \|\psi_2\| \ \wedge \ \forall 0 \leq i < k.\pi^i \in \|\psi_1\|\}
\end{aligned}$$

■

Un état s satisfait φ (ce que l'on note $s \models \varphi$) ssi $s \in [\![\varphi]\!]$. Un chemin π satisfait ψ (ce que l'on note $\pi \models \psi$) ssi $\pi \in \|\psi\|$. Un état s satisfait $\mathbf{E}\psi$ (*resp.* $\mathbf{A}\psi$) ssi ψ est satisfaite par un certain chemin (*resp.* tous les chemins) issu(s) de s. Un chemin π satisfait $\mathbf{X}\psi$ ssi ψ est satisfaite par le suffixe issu du successeur du premier état de π. Un chemin π satisfait ψ_1 **U** ψ_2 ssi ψ_2 est satisfaite par un certain suffixe de π et ψ_1 est continuellement satisfaite jusque-là. Un chemin π satisfait $\mathbf{F}\psi$ (*resp.* $\mathbf{G}\psi$) ssi ψ est satisfaite par un suffixe (*resp.* tous les suffixes) de π. La figure 1.3 donne une représentation graphique des modalités **EF**, **AF**, **EG** et **AG** sur des arbres d'exécution des programmes.

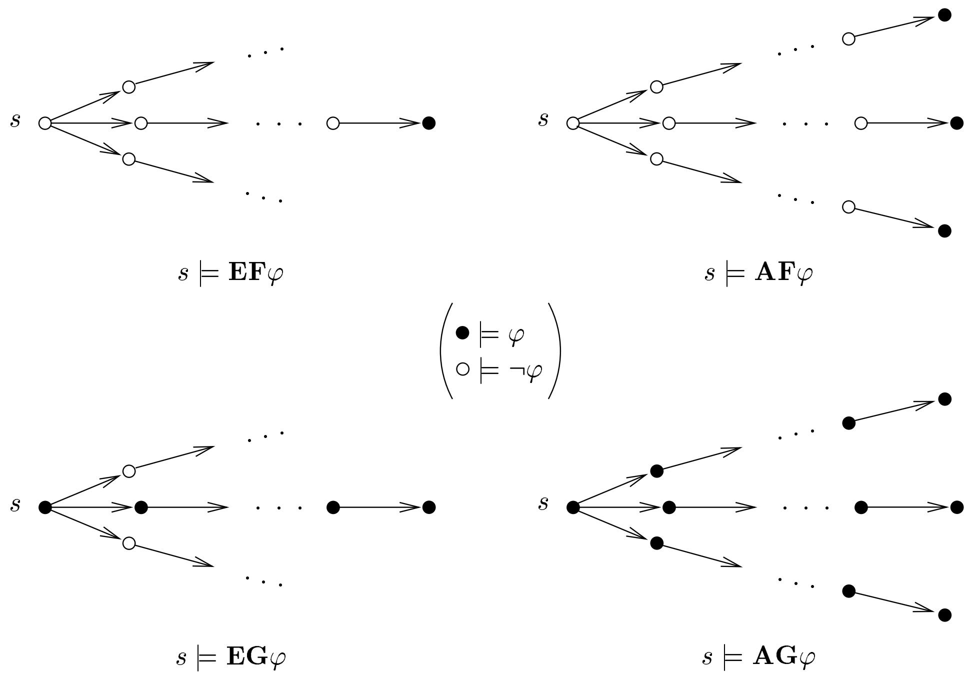

Figure 1.3: Les modalités **EF**, **AF**, **EG** et **AG**

Les différentes combinaisons de ces opérateurs permettent d'exprimer diverses classes de propriétés.

les propriétés de sûreté (*safety properties*) [Lam83], exprimant le fait que le programme ne sera jamais dans un état indésirable, peuvent être décrites comme des assertions *invariantes* (satisfaites par tous les états du modèle).

Exemple 1-1

La formule CTL* suivante caractérise l'exclusion mutuelle entre deux processus P_i et P_j, signifiant qu'il est interdit que les deux processus soient en même temps en section critique :

$$\mathbf{AG}\neg(cs_i \ \wedge \ cs_j)$$

où la proposition atomique cs_i dénote le fait que le processus P_i est en section critique. ■

les propriétés de vivacité (*liveness properties*) [AS85], exprimant le fait que le programme atteindra un certain état désirable, peuvent être décrites comme des assertions de *potentialité* (atteignabilité sur un chemin) et d'*inévitabilité* (atteignabilité sur tous les chemins).

Exemple 1-2
La formule CTL* suivante spécifie le fait qu'après avoir demandé l'accès à la section critique, un processus P_i l'obtiendra au bout d'un temps fini :

$$try_i \Rightarrow \mathbf{AF}cs_i$$

où les propositions atomiques try_i et cs_i dénotent respectivement la demande et l'accès du processus P_i à la section critique. ■

les propriétés d'équité (*fairness properties*) [GPSS80, AFK87], portant sur la répétition infinie de certains états du programme, peuvent être décrites au moyen des modalités $\overset{\infty}{\mathbf{F}}$ ("infiniment souvent") et $\overset{\infty}{\mathbf{G}}$ ("presque toujours").

L'*équité faible* (*weak fairness*), appelée aussi *réponse à l'insistance*, exprime le fait que, si une propriété est continuellement vérifiée, alors une autre propriété le sera infiniment souvent.

Exemple 1-3
La formule CTL* suivante spécifie qu'un processus P_i qui, à partir d'un certain moment, est continuellement prêt à être exécuté, le sera infiniment souvent :

$$\overset{\infty}{\mathbf{G}}ready_i \Rightarrow \overset{\infty}{\mathbf{F}}running_i$$

où les propositions atomiques $ready_i$ (*resp.* $running_i$) dénotent le fait que le processus P_i est prêt pour l'exécution (*resp.* est exécuté). ■

L'équité forte (*strong fairness*), appelée aussi *réponse à la persistance*, exprime le fait que si une propriété est vérifiée infiniment souvent, alors une autre propriété le sera également.

Exemple 1-4
La formule CTL* suivante spécifie qu'un processus P_i qui demande infiniment souvent l'accès à une ressource, l'obtiendra infiniment souvent aussi :

$$\overset{\infty}{\mathbf{F}}request_i \Rightarrow \overset{\infty}{\mathbf{F}}grant_i$$

où les propositions atomiques $request_i$ et $grant_i$ dénotent respectivement la demande et l'accès du processus P_i à la ressource. ■

Différentes sous-logiques particulières de CTL* ont été étudiées dans la littérature.

- PTL (*Propositional Temporal Logic*) [GPSS80] est une logique temporelle qui, bien qu'antérieure à CTL*, peut être vue comme un fragment de CTL* restreint aux formules de chemins. Cette logique, souvent considérée comme le représentant canonique des logiques temporelles "purement linéaires", est définie comme suit :

  Définition 1-6 (Syntaxe et sémantique de PTL)
  Les formules $\psi \in PForm$ de la logique temporelle PTL ont la syntaxe suivante :

  $$\psi \quad ::= \quad p \mid true \mid \neg\psi_1 \mid \psi_1 \ \wedge \ \psi_2 \mid \mathbf{X}\psi_1 \mid \psi_1 \ \mathbf{U} \ \psi_2$$

  Les connecteurs booléens et les modalités temporelles dérivées sont définis de la même manière que pour CTL*. La sémantique des formules de PTL est identique à la sémantique des formules ψ de CTL* dans lesquelles les formules sur états sont réduites à des propositions atomiques. ■

  Intuitivement, les formules de PTL sont obtenues en éliminant les quantificateurs de chemins **E** et **A** des formules sur états. La satisfaction des formules PTL est définie en termes de chemins d'exécution : une structure de Kripke $\mathcal{K}$ satisfait une formule ψ ssi tous les chemins maximaux de $\mathcal{K}$ satisfont ψ.

- CTL (*Computation Tree Logic*) [CES86] est un fragment de CTL* restreint aux formules d'états. Cette logique, souvent considérée comme le représentant canonique des logiques temporelles "purement arborescentes", est définie comme suit :

  Définition 1-7 (Syntaxe et sémantique de CTL)
  Les formules $\varphi \in \mathit{SForm}$ de la logique temporelle CTL ont la syntaxe suivante :

  $$\varphi \quad ::= \quad p \mid \mathit{true} \mid \neg\varphi_1 \mid \varphi_1 \ \wedge \ \varphi_2 \mid \mathbf{EX}\varphi_1 \mid \mathbf{AX}\varphi_1 \mid \mathbf{E}[\varphi_1 \ \mathbf{U} \ \varphi_2] \mid \mathbf{A}[\varphi_1 \ \mathbf{U} \ \varphi_2]$$

  Les connecteurs booléens et les modalités temporelles dérivées sont définis de la même manière que pour CTL*. La sémantique des formules de CTL est identique à la sémantique des formules φ de CTL* dans lesquelles les arguments des formules sur chemins sont réduits à des formules sur états. ■

  Intuitivement, les formules de CTL sont obtenues en préfixant chaque modalité linéaire **X** et **U** par un quantificateur de chemin **E** ou **A**. La satisfaction des formules de CTL est définie en termes d'états : une structure de Kripke $\mathcal{K}$ satisfait une formule φ ssi tous les états de $\mathcal{K}$ satisfont φ.

- Différentes autres logiques arborescentes (qui, bien qu'antérieures à CTL, peuvent être vues comme des fragments de cette logique) ont été définies et étudiées, notamment UB (*Unified system of Branching time*) [BAPM83]. Des études comparatives de ces logiques sont exposées de manière synthétique dans [ES89, Lar94].

Comparaison entre les logiques linéaires et arborescentes Il existe plusieurs aspects à prendre en compte pour le choix d'une logique temporelle particulière. Nous présentons ci-dessous une comparaison entre les logiques temporelles linéaires et arborescentes selon différents critères.

Expressivité. Un aspect important est le pouvoir expressif d'une logique temporelle, c'est-à-dire sa capacité de décrire différentes classes de propriétés. De ce point de vue, les logiques PTL et CTL sont incomparables, chacune d'entre elles permettant de décrire des propriétés qui ne sont pas exprimables dans l'autre. Par exemple, du fait de l'absence des quantificateurs sur chemins, PTL ne permet pas de discerner l'atteignabilité potentielle de l'atteignabilité inévitable, qui s'expriment respectivement en CTL par les formules $\mathbf{EF}\varphi$ et $\mathbf{AF}\varphi$. Inversement, du fait que les quantificateurs sur chemins sont attachés aux modalités linéaires, CTL ne permet pas d'exprimer les propriétés nécessitant l'imbrication des modalités de chemins (notamment, certaines propriétés d'équité) : par exemple, le fait qu'une propriété φ est vérifiée continuellement sur chaque chemin du modèle (sauf éventuellement sur un préfixe fini de ce chemin), exprimée en PTL par la formule $\mathbf{FG}\varphi$, n'a pas de formulation équivalente en CTL.

De manière générale, grâce à la possibilité d'imbriquer les modalités de chemins, les logiques purement linéaires sont mieux adaptées pour exprimer les propriétés d'équité, tandis que les logiques purement arborescentes, grâce à la présence explicite des quantificateurs, permettent de caractériser des chemins individuels du modèle. La logique CTL*, pour sa part, permet d'exprimer les deux types de propriétés.

Efficacité d'évaluation. En pratique, il est important de disposer d'algorithmes efficaces pour évaluer les formules temporelles sur des modèles. De ce point de vue, les logiques purement arborescentes sont avantagées par rapport aux logiques purement linéaires. En effet, le problème d'évaluation des formules de la logique PTL sur un modèle est PSPACE-complet [SC85] (ce qui entraîne la même complexité pour l'évaluation de CTL*). Les algorithmes classiques d'évaluation de PTL [LP85], basés sur la traduction des formules vers des automates de Büchi, ont une complexité linéaire en taille du modèle et exponentielle en taille de la formule. En revanche, l'évaluation des formules CTL est (beaucoup) plus efficace : les algorithmes classiques [CES86] sont linéaires en taille du modèle et aussi en taille de la formule.

La complexité accrue de l'évaluation des logiques linéaires provient essentiellement de leur capacité à imbriquer les modalités de chemins (ce qui n'est pas autorisé en CTL) ; à l'opposé, les logiques arborescentes s'évaluent efficacement, mais ne permettent pas d'exprimer les propriétés d'équité. Plusieurs approches pour étendre CTL avec des opérateurs d'équité (tout en maintenant une complexité d'évaluation raisonnable) ont été proposées dans la littérature ; nous en présentons quelques-unes dans le paragraphe suivant.

La comparaison entre l'expressivité et l'efficacité d'évaluation des logiques linéaires et arborescentes a été l'objet de recherches [Lam80, EH83, Lam83, EL85] qui ont abouti à la définition de CTL*. Une discussion détaillée de ces aspects peut être trouvée dans [Eme83, ES89]. Il est cependant important de souligner qu'à partir d'une logique temporelle L purement linéaire, on peut toujours construire une logique temporelle arborescente contenant L, qui soit strictement plus expressive que L, tout en ayant la même complexité d'évaluation [EL85].

Adaptation aux programmes parallèles. Pour capturer de manière suffisamment fine les propriétés "intéressantes" des programmes parallèles, une logique temporelle doit avoir une interprétation adaptée à la sémantique de ces programmes. Les logiques purement linéaires comme PTL, bien qu'elles soient interprétées sur des structures de Kripke (graphes orientés quelconques), expriment uniquement les propriétés sur les séquences d'exécution des programmes. Autrement dit, ces logiques considèrent le modèle d'un programme comme l'ensemble de toutes ses séquences d'exécution possibles ("traces"). Cette interprétation simplifiée n'est guère compatible avec la sémantique d'entrelacement souvent utilisée pour les programmes parallèles. Ainsi, la théorie des algèbres de processus (comme CCS, CSP et ACP) repose sur l'équivalence de comportements, habituellement définie au moyen de relations comme la bisimulation forte [Par81], la bisimulation de branchement [vGW89], etc. Ces relations prennent en compte les branchements (choix entre plusieurs possibilités) dans les modèles. De ce point de vue, une logique temporelle purement linéaire peut ne pas être satisfaisante pour ces algèbres de processus, puisqu'elle ne permet pas de distinguer entre les deux comportements $a(b+c)$ et $ab+ac$ qui, bien qu'ayant les mêmes séquences d'exécution, ne sont pas fortement équivalents [MP89]. Ceci explique, en un sens, pourquoi les logiques temporelles utilisées avec prédilection dans le domaine des algèbres de processus sont les logiques arborescentes qui, elles, permettent de distinguer plus finement la structure de branchement.

Nous avons précisé ici uniquement les critères de comparaison des logiques temporelles qui nous ont semblé essentiels pour cette étude. D'autres aspects, plus fins, tels que la capacité d'une logique temporelle à exprimer de manière *concise* et *naturelle* les propriétés, ont été étudiés dans la littérature (voir les synthèses [ES89, Lar94]). Nous reviendrons sur ces aspects par la suite.

Extensions des logiques linéaires et arborescentes Les logiques temporelles linéaires et arborescentes classiques peuvent être enrichies avec différents types d'opérateurs, afin d'accroître leur pouvoir expressif et/ou la concision de description des propriétés. Nous indiquons ci-dessous quelques-unes des ces extensions, groupées selon le type d'opérateurs rajoutés aux logiques respectives.

Opérateurs d'équité. Ces extensions concernent principalement la logique CTL : elles ont pour but de permettre l'expression des propriétés d'équité, sans (trop) diminuer la performance des algorithmes d'évaluation.

Une première approche est illustrée par les logiques ECTL (*Extended CTL*) et ECTL$^+$ [EC80], qui permettent aux quantificateurs **E** et **A** de préfixer des formules d'états contenant des (combinaisons booléennes de) modalités de chemins **F**, **G**, $\overset{\infty}{\mathbf{F}}$ et $\overset{\infty}{\mathbf{G}}$. Ces logiques, bien que moins expressives que CTL*, permettent d'exprimer certaines propriétés d'équité (en particulier, la réponse à l'insistance et à la persistance), tout en autorisant des algorithmes d'évaluation dont la complexité est quadratique en taille du graphe et de la formule.

Une deuxième approche, utilisée dans la logique FCTL (*Fair CTL*) [EL85], consiste à remplacer les quantificateurs de chemins par deux nouveaux quantificateurs $\mathbf{E}_\Phi$ et $\mathbf{A}_\Phi$, où Φ représente une *contrainte d'équité*, c'est-à-dire une formule linéaire contenant une combinaison booléenne de modalités $\overset{\infty}{\mathbf{F}}$ et $\overset{\infty}{\mathbf{G}}$. Contrairement à ECTL et ECTL$^+$, qui rajoutent des modalités d'équité dans les formules, FCTL restreint l'interprétation des formules aux ensembles de chemins *équitables*, c'est-à-dire satisfaisant des contraintes Φ. Dans [EL85] il est montré que le fragment de FCTL où les contraintes d'équité ont la "forme canonique" $\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} (\overset{\infty}{\mathbf{G}}\psi_{ij} \ \vee \ \overset{\infty}{\mathbf{F}}\psi'_{ij})$ possède un algorithme d'évaluation linéaire en taille du modèle et polynômiale en taille de la formule. Cependant, pour des contraintes d'équité Φ arbitraires, le problème de l'évaluation de FCTL devient NP-complet.

Des approches similaires pour étendre les logiques arborescentes avec des opérateurs d'équité sont proposées en [QS83, CES86, Arn89]. Nous présenterons certains de ces opérateurs plus loin, à l'annexe C.4.

Opérateurs réguliers. Les logiques temporelles linéaires comme PTL permettent de décrire l'ordonnancement des événements sur les séquences d'exécution des programmes. Cependant, il existe des propriétés sur chemins qui sont facilement exprimables en termes d'expressions régulières, mais qui ne peuvent pas être décrites en PTL : un exemple de telle propriété est le prédicat $\mathit{Even}(\varphi)$, qui est satisfait par un chemin π ssi la formule φ est vérifiée par chaque état $\pi(2k)$ (avec $k \geq 0$). La logique temporelle ETL (*Extended Temporal Logic*) [Wol83] constitue une extension de PTL avec des opérateurs temporels définis au moyen de grammaires régulières et capables de caractériser des propriétés régulières sur les chemins. ETL est strictement plus expressive que PTL, tout en ayant la même complexité d'évaluation.

La même approche a été utilisée pour les logiques temporelles arborescentes. Ainsi, la logique BRTL (*Branching time Regular Temporal Logic*) [HHY90] est une extension de CTL avec des opérateurs définis au moyen d'automates de Büchi déterministes, permettant d'exprimer des propriétés régulières sur les arbres d'exécution des programmes. Cette logique est strictement plus expressive que CTL, tout en ayant la même complexité d'évaluation (linéaire en taille du modèle et de la formule). ECTL* (*Extended CTL**) [Tho89] est une extension similaire de CTL*.

Opérateurs de point fixe. Un autre moyen d'augmenter la puissance d'expression des logiques temporelles linéaires est d'introduire des opérateurs de point fixe. Ces opérateurs permettent d'exprimer des propriétés temporelles comme plus petits points fixes $\mu X.\psi(X)$ et plus grands points fixes $\nu X.\psi(X)$ de fonctionnelles monotones (la monotonie est obtenue en exigeant que la variable propositionnelle X apparaisse dans la formule de chemins ψ sous un nombre pair de négations).

Dans le cas des logiques temporelles linéaires, nous pouvons citer deux extensions de PTL avec des opérateurs de point fixe : νTL [BB86] et μTL [Var88] (cette dernière logique ayant aussi des opérateurs sur le passé). Ces logiques ont la même puissance d'expression (et la même complexité d'évaluation) qu'ETL, mais présentent l'avantage d'avoir un ensemble restreint d'opérateurs primitifs (en l'occurrence, l'opérateur $\mathbf{X}$ et les opérateurs μ et ν, de plus petit et de plus grand point fixe).

Une extension similaire des logiques temporelles arborescentes est le *μ-calcul modal* [Koz83]. Ce formalisme permet d'exprimer des propriétés sur les états aussi bien que sur les actions des programmes (et donc il est naturellement interprété sur des modèles STEs étendus). Nous présenterons le μ-calcul modal à la section 1.2.3.

Opérateurs sur le passé. La logique CTL* est une logique "du futur", au sens où elle ne permet d'exprimer que des propriétés sur le futur, c'est-à-dire portant sur les états qui seront atteints *après* l'état courant du programme. Cependant, certaines propriétés temporelles s'expriment plus naturellement en fonction des états parcourus par le programme *avant* d'atteindre l'état courant. Par exemple, le fait qu'un dysfonctionnement φ_2 soit toujours précédé par une cause φ_1, s'exprime naturellement comme $\varphi_2 \Rightarrow \mathbf{F}^{-1}\varphi_1$ (où $\mathbf{F}^{-1}$, l'opérateur symétrique de $\mathbf{F}$ sur le passé, signifie qu'une formule a été vérifiée avant l'état courant). Ceci justifie l'introduction d'opérateurs sur le passé dans les logiques temporelles. Ainsi, PPTL (*Past PTL*) [LPZ85, MP92] représente l'extension de PTL avec les opérateurs du passé $\mathbf{X}^{-1}$ ("Previous") et $\mathbf{S}$ ("Since") qui sont symétriques aux opérateurs du futur $\mathbf{X}$ et $\mathbf{U}$. Des extensions similaires ont été proposées pour les logiques arborescentes, notamment PCTL et PCTL$^+$ [Lar94] ou PCTL* [HT87, Lar94].

Cependant, dans le contexte des modèles ayant un état initial sans passé (comme c'est naturellement le cas des programmes parallèles) [MP89], la plupart des logiques avec passé peuvent se traduire en logiques du futur. Une étude comparative détaillée des différentes logiques avec passé, ainsi que des traductions efficaces de ces logiques vers des logiques du futur, peuvent être trouvées en [Lar94].

D'autres extensions apportées aux logiques temporelles linéaires et arborescentes, notamment avec des valeurs typées, seront présentées à la section 1.2.5.

1.2.2 Logiques temporelles basées sur actions

Les logiques temporelles décrites à la section 1.2.1 permettent d'exprimer des propriétés portant sur les états des programmes. Cependant, certaines classes de langages parallèles, comme les algèbres de processus CCS, CSP et ACP ou les langages de description LOTOS et μCRL, ont une sémantique opérationnelle définie en termes d'*actions* effectuées par le programme. Les propriétés des programmes décrits dans ces langages doivent donc naturellement porter sur l'ordonnancement des actions dans le temps. Ceci a motivé le développement de logiques temporelles basées sur actions, qui ne sont pas interprétées sur des structures de Kripke, mais sur des systèmes de transitions étiquetées $\mathcal{A} = (S, A, T)$ (voir la définition 1-2).

Comme nous l'avons précisé dans la section 1.2.1, les logiques temporelles arborescentes sont mieux adaptées que les logiques linéaires pour la spécification des programmes décrits dans des algèbres de processus. C'est pourquoi nous nous intéressons de préférence aux logiques temporelles arborescentes interprétées sur actions.

Logiques arborescentes sur actions Nous présentons ci-dessous plusieurs logiques arborescentes sur actions, parmi les plus représentatives, accompagnées d'exemples de propriétés qu'elles permettent d'exprimer.

La logique temporelle ACTL* (*Action CTL**) [NV90] constitue l'équivalent de CTL* en termes d'actions. Nous définissons d'abord ACTL*, puis son fragment purement arborescent ACTL. Afin de souligner l'analogie avec CTL*, au lieu de la définition originelle [NV90], nous avons suivi la formalisation donnée dans [NFGR91], en séparant les formules sur états et les formules sur chemins. Une autre différence par rapport à la définition [NV90] est l'usage explicite de l'action τ dans les modalités temporelles.

Définition 1-8 (Syntaxe et sémantique d'ACTL*)
La logique temporelle ACTL* contient des formules sur actions $\alpha \in AForm$, des formules sur états $\varphi \in SForm$ et des formules sur chemins $\psi \in PForm$, ayant la syntaxe suivante :

$$
\begin{array}{lcl}
\alpha & ::= & a \mid true \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \\
\varphi & ::= & true \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{E}\psi \\
\psi & ::= & \varphi \mid \neg\psi_1 \mid \psi_1 \wedge \psi_2 \mid \mathbf{X}_\alpha\psi_1 \mid \psi_1{}_\alpha\mathbf{U}\ \psi_2 \mid \psi_1{}_{\alpha_1}\mathbf{U}_{\alpha_2}\psi_2
\end{array}
$$

où $a \in A$ est une action, $\mathbf{E}$ dénote le quantificateur existentiel sur les chemins et $\mathbf{X}$ (*resp.* $\mathbf{U}$) sont les opérateurs "neXt" (*resp.* "Until") paramétrés par des formules sur actions. Les connecteurs booléens dérivés sont définis de façon habituelle : $false \stackrel{\mathrm{d}}{=} \neg true$, $\alpha_1 \vee \alpha_2 \stackrel{\mathrm{d}}{=} \neg(\neg\alpha_1 \wedge \neg\alpha_2)$, $\alpha_1 \Rightarrow \alpha_2 \stackrel{\mathrm{d}}{=} \neg\alpha_1 \vee \alpha_2$, $\alpha_1 \Leftrightarrow \alpha_2 \stackrel{\mathrm{d}}{=} (\alpha_1 \Rightarrow \alpha_2) \wedge (\alpha_2 \Rightarrow \alpha_1)$ (les mêmes identités sont valables pour les formules φ et ψ). Les modalités temporelles dérivées sont définies de manière similaire à celles de CTL* : $\mathbf{A}\psi \stackrel{\mathrm{d}}{=} \neg\mathbf{E}\neg\psi$, $\mathbf{F}\psi \stackrel{\mathrm{d}}{=} true_{true}\mathbf{U}\ \psi$, $\mathbf{G}\psi \stackrel{\mathrm{d}}{=} \neg\mathbf{F}\neg\psi$.

La sémantique des formules sur actions est définie par la fonction d'interprétation $[\![.]\!] : AForm \to 2^A$. Pour une formule α, la dénotation $[\![\alpha]\!]$ renvoie l'ensemble d'actions du modèle qui satisfont α. La fonction sémantique est définie inductivement comme suit :

$$
\begin{array}{rcl}
[\![a]\!] & \stackrel{\mathrm{d}}{=} & \{a\} \\
[\![true]\!] & \stackrel{\mathrm{d}}{=} & A \\
[\![\neg\alpha_1]\!] & \stackrel{\mathrm{d}}{=} & A \setminus [\![\alpha_1]\!] \\
[\![\alpha_1 \wedge \alpha_2]\!] & \stackrel{\mathrm{d}}{=} & [\![\alpha_1]\!] \cap [\![\alpha_2]\!]
\end{array}
$$

La sémantique des formules sur états (*resp.* sur chemins) est définie au moyen d'une fonction d'interprétation $[\![.]\!] : SForm \to 2^S$ (*resp.* $\|.\| : PForm \to 2^R$). Pour une formule φ (*resp.* ψ), la dénotation $[\![\varphi]\!]$ (*resp.* $\|\psi\|$) renvoie l'ensemble d'états (*resp.* de chemins maximaux) du modèle qui satisfont φ (*resp.* ψ). Les fonctions sémantiques sont définies inductivement comme suit :

$$
\begin{array}{rcl}
[\![true]\!] & \stackrel{\mathrm{d}}{=} & S \\
[\![\neg\varphi_1]\!] & \stackrel{\mathrm{d}}{=} & S \setminus [\![\varphi_1]\!] \\
[\![\varphi_1 \wedge \varphi_2]\!] & \stackrel{\mathrm{d}}{=} & [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![\mathbf{E}\psi]\!] & \stackrel{\mathrm{d}}{=} & \{s \in S \mid \exists\pi \in R.\pi(0) = s \wedge \pi \in \|\psi\|\}
\end{array}
$$

et

$$
\begin{array}{rcl}
\|\varphi\| & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid \pi(0) \in [\![\varphi]\!]\} \\
\|\neg\psi_1\| & \stackrel{\mathrm{d}}{=} & R \setminus \|\psi_1\| \\
\|\psi_1 \wedge \psi_2\| & \stackrel{\mathrm{d}}{=} & \|\psi_1\| \cap \|\psi_2\| \\
\|\mathbf{X}_\alpha\psi_1\| & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid l(\pi, 0) \in [\![\alpha]\!] \wedge \pi^1 \in \|\psi_1\|\} \\
\|\psi_1{}_\alpha\mathbf{U}\ \psi_2\| & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid \exists k \geq 0.\pi^k \in \|\psi_2\| \wedge \forall 0 \leq i < k.\pi^i \in \|\psi_1\| \wedge l(\pi, i) \in [\![\alpha]\!]\} \\
\|\psi_1{}_{\alpha_1}\mathbf{U}_{\alpha_2}\psi_2\| & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid \exists k \geq 0.\pi^k \in \|\psi_1\| \wedge l(\pi, k) \in [\![\alpha_2]\!] \wedge \pi^{k+1} \in \|\psi_2\| \wedge \\
& & \quad \forall 0 \leq i < k.\pi^i \in \|\psi_1\| \wedge l(\pi, i) \in [\![\alpha_1]\!]\}
\end{array}
$$

■

Une action a satisfait α (ce que l'on note $a \models \alpha$) ssi $a \in [\![\alpha]\!]$. Un état s satisfait φ (ce que l'on note $s \models \varphi$) ssi $s \in [\![\varphi]\!]$. Un chemin π satisfait ψ (ce que l'on note $\pi \models \psi$) ssi $\pi \in \|\psi\|$. Une action qui satisfait une formule α est appelée *α-action* ; une transition étiquetée par une α-action est appelée *α-transition*.

La sémantique des modalités **A**, **F** et **G** est similaire aux opérateurs correspondants de CTL*. D'autres modalités dérivées utiles en pratique peuvent être définies comme suit [dNV90] : $\psi_1 \langle\alpha\rangle \psi_2 = \mathbf{E}(\psi_1{}_\tau \mathbf{U}_\alpha \psi_2)$, $\psi_1 \langle\varepsilon\rangle \psi_2 = \mathbf{E}(\psi_1{}_\tau \mathbf{U}\ \psi_2)$, $\langle\alpha\rangle \psi = \mathit{true} \langle\alpha\rangle \psi$, $[\alpha] \psi = \neg \langle\alpha\rangle \neg\psi$. Un état s satisfait $\psi_1 \langle\alpha\rangle \psi_2$ s'il existe un chemin issu de s conduisant (après zéro ou plusieurs τ-transitions) à une α-transition, après laquelle ψ_2 est satisfaite et avant laquelle ψ_1 est continuellement satisfaite. Un état s satisfait $\psi_1 \langle\varepsilon\rangle \psi_2$ s'il existe un chemin issu de s ayant un préfixe de (zéro ou plusieurs) τ-transitions, après lequel ψ_2 est satisfaite, ψ_1 étant continuellement satisfaite jusque-là. Enfin, un état s satisfait $\langle\alpha\rangle \psi$ (*resp.* $[\alpha] \psi$) si un chemin (*resp.* tous les chemins) issu(s) de s mène(nt), après un nombre quelconque de τ-transitions, à une α-transition après laquelle ψ est satisfaite.

Les diverses combinaisons de ces opérateurs permettent d'exprimer des propriétés de sûreté, de vivacité et d'équité portant sur les actions des programmes.

Exemple 1-5
L'exclusion mutuelle entre deux processus P_i et P_j qui accèdent à une ressource partagée signifie que chaque fois que P_i gagne l'accès à la ressource, il n'est pas possible que P_j le gagne aussi tant que P_i n'ait pas libéré la ressource. Ceci peut être exprimé en ACTL* par la formule suivante :

$$[\mathit{OPEN}_i]\neg\mathbf{E}(\mathit{true}_{\neg \mathit{CLOSE}_i} \mathbf{U}_{\mathit{OPEN}_j} \mathit{true})$$

où les actions OPEN_i et CLOSE_i dénotent respectivement l'accès et la libération de la ressource par le processus P_i. ■

Exemple 1-6
Le fait que chaque émission de message soit inévitablement suivie par une réception de message peut être exprimé en ACTL* par la formule suivante :

$$[\mathit{SEND}]\mathbf{A}(\mathit{true}_{\neg \mathit{SEND}} \mathbf{U}_{\mathit{RECV}} \mathit{true})$$

où les actions *SEND* et *RECV* dénotent respectivement l'émission et la réception d'un message. La clause $\neg$*SEND* présente en partie gauche de l'opérateur **U** signifie qu'aucune autre émission de message n'est autorisée avant la réception. ■

Exemple 1-7
La propriété d'équité forte spécifiant qu'un processus qui demande infiniment souvent l'accès à une ressource partagée l'obtienne infiniment souvent peut être exprimée par la formule ACTL* ci-dessous :

$$\mathbf{A}(\mathbf{G}(\mathit{true}_{\mathit{true}} \mathbf{U}_{\mathit{REQUEST}_i} \mathit{true}) \Rightarrow \mathbf{G}(\mathit{true}_{\mathit{true}} \mathbf{U}_{\mathit{GRANT}_i} \mathit{true}))$$

où les actions $\mathit{REQUEST}_i$ et GRANT_i dénotent respectivement la demande et l'accès à la ressource partagée par le processus P_i. ■

ACTL* peut être traduite vers CTL* (et vice-versa), moyennant une traduction entre les STEs et les structures de Kripke [NV90, NFGR91]. Ceci signifie qu'ACTL* permet de décrire, en termes d'actions, toutes les propriétés exprimables en CTL* en termes d'états.

De la même manière que pour CTL*, des sous-logiques particulières d'ACTL* ont été étudiées. Ainsi, ACTL (*Action CTL*) [NV90, NFGR91] est un fragment d'ACTL* restreint aux formules d'états. Cette logique, qui peut être considérée comme le représentant standard des logiques temporelles "purement arborescentes" basées sur actions, est définie ci-dessous.

Définition 1-9 (Syntaxe et sémantique d'ACTL)
Les formules $\alpha \in AForm$ et $\varphi \in SForm$ de la logique temporelle ACTL ont la syntaxe suivante :

$$\begin{array}{rcl} \alpha & ::= & a \mid true \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \\ \varphi & ::= & true \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{EX}_\alpha\varphi_1 \mid \mathbf{AX}_\alpha\varphi_1 \\ & \mid & \mathbf{E}[\varphi_{1\,\alpha}\mathbf{U}\ \varphi_2] \mid \mathbf{A}[\varphi_{1\,\alpha}\mathbf{U}\ \varphi_2] \mid \mathbf{E}[\varphi_{1\,\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2] \mid \mathbf{A}[\varphi_{1\,\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2] \end{array}$$

Les connecteurs booléens et les modalités temporelles dérivées sont définis de la même manière que pour ACTL*. Les formules sur actions ont la même sémantique que celles d'ACTL*. La sémantique des formules sur états est identique à la sémantique des formules φ d'ACTL* dans lesquelles les arguments des formules sur chemins sont réduits à des formules sur états. ■

Intuitivement, les formules d'ACTL sont obtenues en préfixant chaque modalité linéaire **X** et **U** par un quantificateur de chemin **E** ou **A**. La satisfaction des formules ACTL est définie en termes d'états : un STE $\mathcal{A}$ satisfait une formule φ ssi tous les états de $\mathcal{A}$ satisfont φ.

La logique ACTL est suffisamment puissante pour exprimer des propriétés de sûreté et de vivacité (les formules des exemples 1-5 et 1-6 sont en fait des formules ACTL). En outre, tout comme CTL, ACTL bénéficie d'un algorithme d'évaluation efficace, linéaire en taille du modèle et de la formule [BGL94], ce qui la rend attractive du point de vue pratique.

Logiques modales sur actions D'autres logiques sur actions (qui, bien qu'antérieures à ACTL* et ACTL, peuvent être vues aujourd'hui comme des fragments de ces logiques) ont été définies afin de caractériser les propriétés des programmes décrits dans les algèbres de processus : il s'agit de logiques *modales*, munies d'opérateurs permettant d'exprimer la *possibilité* et la *nécessité*. La logique HML (Hennessy-Milner) [HM85] peut être considérée comme le représentant standard de cette classe de logiques. La définition de HML que nous donnons ci-dessous est une légère extension de la définition originelle [HM85], dont les modalités contiennent uniquement des actions du STE (et non des formules sur actions).

Définition 1-10 (Syntaxe et sémantique de HML)
La logique HML contient des formules sur actions $\alpha \in AForm$ et des formules sur états $\varphi \in SForm$, ayant la syntaxe suivante :

$$\begin{array}{rcl} \alpha & ::= & a \mid true \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \\ \varphi & ::= & true \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \langle\alpha\rangle\,\varphi_1 \end{array}$$

Les connecteurs booléens dérivés (sur les formules d'actions et d'états) sont définis de la même manière que pour ACTL. L'opérateur de nécessité est le dual de l'opérateur de possibilité : $[\alpha]\,\varphi \stackrel{\mathrm{d}}{=} \neg\,\langle\alpha\rangle\,\neg\varphi$.

Les formules sur actions ont la même sémantique que celles d'ACTL. La sémantique des formules sur états est définie par la fonction d'interprétation $[\![.]\!] : SForm \to 2^S$. Pour une formule φ, la dénotation $[\![\varphi]\!]$ renvoie l'ensemble d'états du STE satisfaisant φ. La fonction sémantique est définie inductivement comme suit :

$$\begin{array}{rcl} [\![true]\!] & \stackrel{\mathrm{d}}{=} & S \\ [\![\neg\varphi_1]\!] & \stackrel{\mathrm{d}}{=} & S \setminus [\![\varphi_1]\!] \\ [\![\varphi_1 \wedge \varphi_2]\!] & \stackrel{\mathrm{d}}{=} & [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\ [\![\langle\alpha\rangle\,\varphi_1]\!] & \stackrel{\mathrm{d}}{=} & \{s \in S \mid \exists s' \in S.\exists a \in A.s \xrightarrow{a} s' \wedge a \in [\![\alpha]\!] \wedge s' \in [\![\varphi_1]\!]\} \end{array}$$

■

La relation de satisfaction des formules α et φ par les actions et les états du STE est définie de manière habituelle. Un état s satisfait $\langle\alpha\rangle\,\varphi$ (*resp.* $[\alpha]\,\varphi$) si une α-transition (toutes les α-transitions) issue(s) de s mène(nt) à un état (des états) satisfaisant φ.

HML est une sous-logique d'ACTL, puisque $\langle\alpha\rangle\,\varphi = \mathbf{EX}_\alpha\varphi$. Elle permet de caractériser des propriétés simples portant sur les actions du STE.

Exemple 1-8
Les états de blocage (*deadlock*), c'est-à-dire les états n'ayant aucun successeur, peuvent être caractérisés en HML par la formule suivante :

$$[true]\,false$$

Il convient de préciser que, dans la logique HML originelle [HM85], cette propriété est exprimée par la formule $\bigwedge_{a\in A}[a]\,false$. Les formules sur actions α utilisées dans les modalités permettent donc une description plus concise des propriétés. ■

Bien qu'elle présente un intérêt théorique (par son adéquation pour l'équivalence observationnelle [HM85]), la logique HML n'est pas suffisamment puissante pour exprimer des propriétés sur des chemins de longueur non bornée. C'est pourquoi différentes extensions de HML ont été proposées, comme la logique HML avec "Until" [dNV90]. Cette logique (qui est adéquate pour l'équivalence de branchement) contient les modalités $\varphi_1\,\langle\varepsilon\rangle\,\varphi_2$ (*resp.* $\varphi_1\,\langle\alpha\rangle\,\varphi_2$) qui expriment la possibilité d'atteindre un état satisfaisant φ_2 après un nombre quelconque de τ-transitions (*resp.* suivies d'une α-transition), tout en passant par des états qui satisfont φ_1. Ces deux opérateurs ont été définis dans le paragraphe précédent comme opérateurs dérivés d'ACTL*. En fait, ils sont aussi exprimables en ACTL ; par conséquent, la logique HML avec "Until" est, elle aussi, une sous-logique d'ACTL.

Des présentations synthétiques des logiques modales et temporelles basées sur actions, ainsi que de leurs applications pour la vérification dans le domaine des algèbres de processus, peuvent être trouvées dans [Sti87, Sti92].

1.2.3 Logiques temporelles basées sur états et actions

Les logiques présentées en section 1.2.1 (*resp.* 1.2.2 sont interprétées sur les états (*resp.* les actions). Nous examinons à présent différentes logiques permettant de combiner ces deux types de propriétés. Ces logiques sont naturellement interprétées sur des STMS $\mathcal{L} = (S, P, L, A, T)$ (voir la définition 1-3).

Extensions des logiques basées sur actions Bien que les logiques HML, ACTL et ACTL* soient interprétées sur des STEs, elles contiennent toutes des formules φ sur états. Il est donc facile d'étendre ces logiques au cas des modèles STM, en rajoutant des propositions atomiques p (comme en CTL*) portant sur l'étiquetage L du STM (voir la définition 1-3). Hormis cette modification, les sémantiques de ces logiques restent quasi-identiques à celles définies à la section 1.2.2.

Exemple 1-9
La formule suivante, écrite dans une extension d'ACTL avec des propositions atomiques, exprime le fait qu'après avoir émis une requête d'accès à une ressource partagée, un processus P atteindra inévitablement un état où il utilise la ressource :

$$[REQUEST]\,\mathbf{A}[true_{true}\mathbf{U}\ use]$$

où l'action *REQUEST* et la proposition atomique *use* dénotent respectivement la demande d'accès et l'utilisation de la ressource. ■

Extensions des logiques basées sur états Une autre méthode pour spécifier des propriétés sur états et sur actions est d'étendre les logiques basées sur états, comme CTL ou CTL*, avec des formules sur actions. Cette approche a été utilisée dans la logique LTAC [QS83] qui contient, outre des opérateurs similaires à ceux de CTL, des prédicats sur actions, notés $\mathbf{enable}(a)$ et $\mathbf{after}(a)$. Un état s satisfait $\mathbf{enable}(a)$ s'il existe une a-transition issue de s. Un état s satisfait $\mathbf{after}(a)$ si toutes les transitions menant à s sont étiquetées par a.

Exemple 1-10
La propriété décrite dans l'exemple 1-9 peut être exprimée en LTAC par la formule suivante :

$$\mathbf{after}(\mathit{REQUEST}) \Rightarrow \mathbf{inev}(\mathit{use})$$

où l'opérateur $\mathbf{inev}(\varphi)$ de LTAC est équivalent à la modalité $\mathbf{AF}\varphi$ de CTL. ■

Logiques avec expressions régulières Il existe d'autres classes de logiques temporelles permettant d'exprimer des propriétés sur états et sur actions. Les logiques *dynamiques* (introduites pour l'étude des programmes à commandes gardées) sont des extensions des logiques modales, permettant d'exprimer les propriétés sous forme d'expressions régulières[4] construites sur un vocabulaire d'actions élémentaires[5].

PDL (*Propositional Dynamic Logic*) [FL79] est représentative de cette classe de logiques. La définition sémantique originelle interprète PDL sur des structures de Kripke $\mathcal{K} = (S, P, L, T)$, chaque action élémentaire a induisant une relation binaire sur S qui relie les états marquant le début et la fin de l'exécution de a. Néanmoins, nous pouvons tout aussi bien interpréter PDL sur un STM $\mathcal{L} = (S, P, L, A, T)$, en cherchant à vérifier si les séquences d'exécution de $\mathcal{L}$ satisfont certaines propriétés exprimées par des expressions régulières portant sur les actions de $\mathcal{L}$. Cette interprétation, qui nous semble plus proche des logiques temporelles classiques, est utilisée dans la définition ci-dessous.

Définition 1-11 (Syntaxe et sémantique de PDL)
La logique PDL contient des expressions régulières $\beta \in \mathit{RegExp}$ et des formules sur états $\varphi \in \mathit{SForm}$, ayant la syntaxe suivante :

$$\begin{array}{rcl} \beta & ::= & a \mid \beta_1 ; \beta_2 \mid \beta_1 \cup \beta_2 \mid \beta_1^* \\ \varphi & ::= & p \mid \mathit{true} \mid \neg\varphi_1 \mid \varphi_1 \ \wedge \ \varphi_2 \mid \langle\beta\rangle\, \varphi_1 \end{array}$$

où $a \in A$ est une action du modèle, ";" dénote la composition séquentielle, "$\cup$" dénote le choix non-déterministe, "$*$" dénote l'itération, $p \in P$ est une proposition atomique et $\langle\ \rangle$ dénote la modalité de possibilité. Les connecteurs booléens dérivés (false, $\vee$, $\Rightarrow$, $\Leftrightarrow$) sur les formules sont définis de manière habituelle. La modalité de nécessité est la duale de la modalité de possibilité : $[\beta]\,\varphi \stackrel{\mathrm{d}}{=} \neg\,\langle\beta\rangle\,\neg\varphi$.

La sémantique des expressions régulières est définie au moyen d'une fonction d'interprétation $\|.\|$: $\mathit{RegExp} \to 2^{S\times S}$ (nous avons utilisé la notation $\|.\|$ par analogie avec l'interprétation des formules de chemins en logique temporelle linéaire). Pour une expression régulière β, la dénotation $\|\beta\|$ renvoie la relation reliant les états qui sont source et but des séquences de transitions satisfaisant β. La fonction sémantique est définie inductivement comme suit ($\circ$, $\cup$ et $*$ dénotant respectivement la composition, l'union et la fermeture transitive et réflexive de relations) :

$$\begin{array}{rcl} \|a\| & \stackrel{\mathrm{d}}{=} & \left\{(s_1, s_2) \in S \times S \mid s_1 \xrightarrow{a} s_2\right\} \\ \|\beta_1 ; \beta_2\| & \stackrel{\mathrm{d}}{=} & \|\beta_1\| \circ \|\beta_2\| \end{array}$$

[4] appelées *programmes* en [FL79]
[5] appelées *programmes atomiques* ou *instructions* en [FL79]

$$\begin{aligned}\|\beta_1 \cup \beta_2\| &\stackrel{\text{d}}{=} \|\beta_1\| \cup \|\beta_2\| \\ \|\beta_1^*\| &\stackrel{\text{d}}{=} (\|\beta_1\|)^*\end{aligned}$$

La sémantique des formules sur états est définie par la fonction d'interprétation $[\![.]\!] : \mathit{SForm} \to 2^S$. Pour une formule φ, la dénotation $[\![\varphi]\!]$ renvoie l'ensemble des états du modèle qui satisfont φ. La fonction sémantique est définie inductivement comme suit :

$$\begin{aligned}[\![p]\!] &\stackrel{\text{d}}{=} \big\{s \in S \mid p \in L(s)\big\} \\ [\![\mathit{true}]\!] &\stackrel{\text{d}}{=} S \\ [\![\neg\varphi_1]\!] &\stackrel{\text{d}}{=} S \setminus [\![\varphi_1]\!] \\ [\![\varphi_1 \ \wedge \ \varphi_2]\!] &\stackrel{\text{d}}{=} [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\ [\![\langle\beta\rangle\, \varphi_1]\!] &\stackrel{\text{d}}{=} \big\{s \in S \mid \exists s' \in S.(s,s') \in \|\beta\| \ \wedge \ s' \in [\![\varphi_1]\!]\, \big\}\end{aligned}$$

■

La sémantique des opérateurs sur les expressions régulières β est celle habituellement attribuée aux langages réguliers. Un état s satisfait $\langle\beta\rangle\, \varphi$ (*resp.* $[\beta]\, \varphi$) si un chemin (*resp.* tous les chemins) issu(s) de s dont les actions forment un mot appartenant au langage défini par β mène(nt) à un état (*resp.* des états) satisfaisant φ. Dans l'interprétation originelle de PDL, ceci signifierait qu'une (*resp.* toutes les) exécution(s) du "programme" β mène(nt) à un état (*resp.* des états) satisfaisant la post-condition φ.

Les opérateurs de PDL permettent d'exprimer naturellement des propriétés portant sur des séquences régulières d'actions.

Exemple 1-11

La formule PDL suivante exprime l'existence d'un chemin contenant une émission de message, suivie d'une réception de message, puis de la réception d'un acquittement, ces trois événements étant éventuellement séparés par des séquences d'actions invisibles :

$$\langle \mathit{SEND}; \tau^*; \mathit{RECV}; \tau^*; \mathit{ACK}\rangle\ \mathit{true}$$

où les actions *SEND*, *RECV* et *ACK* dénotent respectivement une émission, une réception et un acquittement. ■

Comme les autres logiques modales et temporelles mentionnées jusqu'ici, PDL a été étendue avec différents opérateurs. Ainsi, PDL avec tests (qui est en fait la logique traitée en [FL79]) étend les expressions régulières PDL avec l'opérateur de test "?", ayant la sémantique suivante :

$$\|\varphi?\| \stackrel{\text{d}}{=} \big\{(s,s) \in S \times S \mid s \in [\![\varphi]\!]\,\big\}$$

qui permet d'obtenir les constructions "if-then-else" des langages de programmation :

$$\mathit{if}\ \varphi\ \mathit{then}\ \beta_1\ \mathit{else}\ \beta_2 \stackrel{\text{d}}{=} \varphi?; \beta_1 \cup \neg\varphi?; \beta_2$$

La logique PDL-Δ [Str82] étend les formules PDL avec un opérateur de bouclage (*looping*) "Δ", défini comme suit :

$$[\![\Delta\beta]\!] \stackrel{\text{d}}{=} \bigcup \big\{S' \subseteq S \mid \forall s \in s'.\exists s' \in S'.(s,s') \in [\![\beta]\!]\,\big\}$$

Cet opérateur permet d'exprimer la répétition infinie d'une séquence régulière d'actions.

En pratique, pour pouvoir exprimer certaines propriétés utiles, il nous semble nécessaire d'enrichir le langage des expressions régulières PDL avec des opérateurs booléens ($\neg$, $\wedge$, $\vee$) sur les actions élémentaires, ce qui revient à construire les expressions régulières de PDL non pas sur des actions élémentaires a mais sur des formules α comme en ACTL.

Exemple 1-12
L'atteignabilité inévitable d'une action a (qui est décrite en ACTL par la formule $\mathbf{A}[true_{true}\mathbf{U}_a true]$) peut être exprimée en PDL-Δ par la formule suivante :

$$[(\neg a)^*] \langle true \rangle \, true \;\wedge\; \neg\Delta(\neg a)$$

Cette formule spécifie qu'à partir d'un état s, il n'est pas possible d'atteindre un état de blocage (c'est-à-dire, un état sans successeurs), ni de boucler indéfiniment avant d'avoir effectué une action a. La conjonction de ces deux propriétés (dont chacune nécessite la négation $\neg$) assure que tous les chemins issus de s aboutiront, au bout d'un temps fini, à une action a. ■

Une autre logique basée sur des expressions régulières est R$\star$ICO (*Regular Information Chronological Ordering*) [Gar89a, chap. 9], proposée pour spécifier les propriétés des programmes LOTOS. R$\star$ICO permet d'exprimer des propriétés temporelles sous forme de séquences régulières de transitions du STE. Bien qu'elle est définie en termes d'actions, il serait possible de l'étendre (de manière similaire à ACTL) pour exprimer aussi des propriétés sur les états.

1.2.4 Logiques avec opérateurs de point fixe

Il est bien connu que certaines modalités temporelles peuvent être caractérisées comme plus petits et plus grands points fixes de fonctionnelles monotones [EC80]. Ceci a motivé le développement de logiques temporelles contenant des opérateurs de point fixe (nous en avons mentionné quelques-unes à la fin de la section 1.2.1), capables d'exprimer une large gamme de modalités en utilisant un ensemble restreint d'opérateurs primitifs. Nous présentons dans les paragraphes suivants deux logiques représentatives basées sur des opérateurs de point fixe.

Le μ-calcul modal Le μ-calcul modal [Koz83] est une logique très expressive, étudiée intensivement dans la littérature. Ce formalisme peut être vu comme une extension de la logique modale HML avec des opérateurs de point fixe. Dans sa version originelle [Koz83], le μ-calcul est interprété sur des structures de Kripke $\mathcal{K} = (S, P, L, T)$. Les formules modales sont définies sur un vocabulaire d'actions atomiques, chaque action a induisant une relation binaire sur S qui relie les états de début et de fin de l'exécution de a. Néanmoins, de façon similaire à PDL, nous pouvons définir la sémantique des formules du μ-calcul modal sur des STMs $\mathcal{L} = (S, P, L, A, T)$, en interprétant les modalités sur les actions contenues dans A. En outre, comme nous avons procédé pour HML et ACTL, la définition ci-dessous étend celle de [Koz83], en autorisant les opérateurs modaux à contenir, au lieu d'actions individuelles a, les formules sur actions d'ACTL. Ceci permet une description concise de différentes propriétés utiles [CPS89, Bra92].

Quelques notions auxiliaires sont nécessaires. Soit $PVar$ un ensemble de *variables propositionnelles*, notées $Y_1, Y_2, \ldots$. Intuitivement, les variables propositionnelles sont des noms qui seront associés aux formules de point fixe. Nous définissons le domaine $\mathbf{PEnv} \stackrel{\mathrm{d}}{=} PVar \to 2^S$ des *environnements propositionnels*. Un environnement propositionnel $\rho \in \mathbf{PEnv}$ est une fonction partielle[6] associant à chaque variable $Y \in supp(\rho)$ l'ensemble $\rho(Y)$ des états du STM qui satisfont Y.

[6] voir le chapitre de notations pour la terminologie sur les fonctions partielles utilisée dans ce document.

Définition 1-12 (Syntaxe et sémantique du μ-calcul modal)
Le μ-calcul modal contient des formules sur actions $\alpha \in \mathit{AForm}$ et des formules sur états $\varphi \in \mathit{SForm}$, ayant la syntaxe suivante :

$$
\begin{array}{lcl}
\alpha & ::= & a \mid \mathit{true} \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \\
\varphi & ::= & p \mid \mathit{true} \mid Y \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \langle\alpha\rangle\,\varphi_1 \mid \mu Y.\varphi_1
\end{array}
$$

où $a \in A$ est une action du modèle, $p \in P$ est une proposition atomique, Y est une variable propositionnelle, $\langle\ \rangle$ est la modalité de possibilité et μ est l'opérateur de plus petit point fixe. Les connecteurs booléens et les modalités dérivées sont définis de manière habituelle. L'opérateur de plus grand point fixe ν est le dual de μ : $\nu Y.\varphi \stackrel{\mathrm{d}}{=} \neg\mu Y.\neg\varphi[\neg Y/Y]$, où $\varphi[\neg Y/Y]$ dénote la substitution syntaxique de Y par $\neg Y$ dans φ.

Une occurrence de variable propositionnelle Y dans une formule φ est dite *liée* si elle est contenue dans une sous-formule $\sigma Y.\varphi'$ de φ. Toutes les autres occurrences de variables propositionnelles dans φ sont dites *libres*. Les formules sur états doivent vérifier la condition de *monotonie syntaxique* [Koz83] : pour chaque formule de point fixe $\sigma Y.\varphi$ (où $\sigma \in \{\mu, \nu\}$), toutes les occurrences de Y dans φ doivent être placées sous un nombre pair de négations.

La sémantique des formules sur actions est identique à celle des formules α d'ACTL. La sémantique des formules sur états est définie par la fonction d'interprétation $\varphi : \mathit{SForm} \to \mathbf{PEnv} \to 2^S$. Pour une formule φ et un environnement ρ (qui doit initialiser toutes les variables propositionnelles libres dans φ), la dénotation $[\![\varphi]\!]\,\rho$ renvoie l'ensemble d'états du modèle qui satisfont φ dans le contexte de ρ. La fonction sémantique est définie inductivement comme suit :

$$
\begin{array}{rcl}
[\![p]\!]\,\rho & \stackrel{\mathrm{d}}{=} & \{s \in S \mid p \in L(s)\} \\
[\![\mathit{true}]\!]\,\rho & \stackrel{\mathrm{d}}{=} & S \\
[\![Y]\!]\,\rho & \stackrel{\mathrm{d}}{=} & \rho(Y) \\
[\![\neg\varphi_1]\!]\,\rho & \stackrel{\mathrm{d}}{=} & S \setminus [\![\varphi_1]\!]\,\rho \\
[\![\varphi_1 \wedge \varphi_2]\!]\,\rho & \stackrel{\mathrm{d}}{=} & [\![\varphi_1]\!]\,\rho \cap [\![\varphi_2]\!]\,\rho \\
[\![\langle\alpha\rangle\,\varphi_1]\!]\,\rho & \stackrel{\mathrm{d}}{=} & \{s \in S \mid \exists s' \in S.\exists a \in A.s \xrightarrow{a} s' \wedge a \in [\![\alpha]\!] \wedge s' \in [\![\varphi_1]\!]\,\rho\} \\
[\![\mu Y.\varphi_1]\!]\,\rho & \stackrel{\mathrm{d}}{=} & \bigcap\{S' \subseteq S \mid \Phi_{1\rho}(S') \subseteq S'\}
\end{array}
$$

où les fonctionnelles $\Phi_{1\rho} : 2^S \to 2^S$, associées aux formules de point fixe, sont définies comme suit :

$$\Phi_{1\rho}(S') \stackrel{\mathrm{d}}{=} [\![\varphi_1]\!]\,(\rho \oslash [S'/Y]).$$

pour tout $S' \subseteq S$. ■

Une formule qui ne contient pas de variables libres est dite *fermée*. Les formules Y, *true* et p (ou p est une proposition atomique) sont dites *atomiques*. Une variable propositionnelle définie par un opérateur μ (*resp.* ν) est appelée μ-variable (*resp.* ν-variable).

Les formules modales ont une sémantique identique à celles de HML (qui est donc une sous-logique du μ-calcul). La formule de point fixe $\mu Y.\varphi$ (*resp.* $\nu Y.\varphi$) est interprétée, dans le contexte d'un environnement ρ, comme le plus petit (*resp.* plus grand) point fixe de la fonctionnelle Φ_ρ. La condition de monotonie syntaxique assure la monotonie des fonctionnelles Φ_ρ, c'est-à-dire que pour tous $S_1 \subseteq S$ et $S_2 \subseteq S$:

$$S_1 \subseteq S_2 \Rightarrow \Phi_\rho(S_1) \subseteq \Phi_\rho(S_2).$$

La sémantique des formules étant définie sur le treillis complet $\langle 2^S, \cup, \cap, \subseteq \rangle$, le théorème de Tarski [Tar55] assure l'existence et l'unicité des plus petits points fixes $\mu\Phi_\rho$ et des plus grands points fixes $\nu\Phi_\rho$ des fonctionnelles Φ_ρ, qui sont respectivement égaux aux interprétations des formules μ et ν données dans la définition 1-12.

De plus, la hauteur du treillis $\langle 2^S, \cup, \cap, \subseteq \rangle$ étant finie (égale à $|S|$, le nombre d'états du modèle), les fonctionnelles Φ_ρ sont aussi $\cup$- et $\cap$-continues (c'est-à-dire qu'elles préservent les limites des suites croissantes et décroissantes dans 2^S). Par conséquent, leurs points fixes ont aussi la caractérisation itérative suivante [Kle52] :

$$[\![\mu Y.\varphi]\!]\,\rho \stackrel{\mathrm{d}}{=} \mu\Phi_\rho = \bigcup_{k\geq 0} \Phi_\rho^k(\emptyset) \tag{1.1}$$

$$[\![\nu Y.\varphi]\!]\,\rho \stackrel{\mathrm{d}}{=} \nu\Phi_\rho = \bigcap_{k\geq 0} \Phi_\rho^k(S) \tag{1.2}$$

Le μ-calcul modal est un formalisme puissant, permettant une traduction concise de différentes autres logiques temporelles : il peut être vu comme un "langage assembleur" pour l'expression des propriétés temporelles. A titre d'exemple, les traductions en μ-calcul des modalités de Ctl, d'Actl et de Pdl-Δ sont données dans la table 1.1.

Opérateur		Traduction
Ctl	$\mathbf{EX}\varphi$	$\langle true \rangle\, \varphi$
	$\mathbf{AX}\varphi$	$[true]\, \varphi \wedge \langle true \rangle\, true$
	$\mathbf{E}[\varphi_1 \;\mathbf{U}\; \varphi_2]$	$\mu Y.(\varphi_2 \vee (\varphi_1 \wedge \langle true \rangle\, Y))$
	$\mathbf{A}[\varphi_1 \;\mathbf{U}\; \varphi_2]$	$\mu Y.(\varphi_2 \vee (\varphi_1 \wedge \langle true \rangle\, true \wedge [true]\, Y))$
Actl	$\mathbf{EX}_\alpha\varphi$	$\langle \alpha \rangle\, \varphi$
	$\mathbf{AX}_\alpha\varphi$	$[\alpha]\, \varphi \wedge [\neg\alpha]\, false \wedge \langle true \rangle\, true$
	$\mathbf{E}[\varphi_{1\,\alpha}\mathbf{U}\; \varphi_2]$	$\mu Y.(\varphi_2 \vee (\varphi_1 \wedge \langle \alpha \rangle\, Y))$
	$\mathbf{A}[\varphi_{1\,\alpha}\mathbf{U}\; \varphi_2]$	$\mu Y.(\varphi_2 \vee (\varphi_1 \wedge [\neg\alpha]\, false \wedge \langle true \rangle\, true \wedge [\alpha]\, Y))$
	$\mathbf{E}[\varphi_{1\,\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$	$\mu Y.(\varphi_1 \wedge (\langle \alpha_2 \rangle\, \varphi_2 \vee \langle \alpha_1 \rangle\, Y))$
	$\mathbf{A}[\varphi_{1\,\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$	$\mu Y.(\varphi_1 \wedge \langle true \rangle\, true \wedge [\neg(\alpha_1 \vee \alpha_2)]\, false \wedge$ $[\neg\alpha_1 \wedge \alpha_2]\, \varphi_2 \wedge [\neg\alpha_2]\, Y \wedge [\alpha_1 \wedge \alpha_2]\, (\varphi_2 \vee Y))$
Pdl-Δ	$\langle \beta_1 ; \beta_2 \rangle\, \varphi$	$\langle \beta_1 \rangle\, \langle \beta_2 \rangle\, \varphi$
	$\langle \beta_1 \cup \beta_2 \rangle\, \varphi$	$\langle \beta_1 \rangle\, \varphi \vee \langle \beta_2 \rangle\, \varphi$
	$\langle \beta^* \rangle\, \varphi$	$\mu Y.(\varphi \vee \langle \beta \rangle\, Y)$
	$\Delta\beta$	$\nu Y.\, \langle \beta \rangle\, Y$

Table 1.1: Traduction en μ-calcul des modalités de Ctl, Actl et Pdl-Δ

Bien entendu, les formules de μ-calcul qui ne contiennent pas de formules sur actions (comme les traductions des modalités de Ctl) peuvent être interprétées sur des structures de Kripke. De manière similaire, les formules de μ-calcul qui ne contiennent pas de propositions atomiques (comme les traductions des modalités d'Actl) peuvent être interprétées sur des Stes.

Les traductions données dans la table 1.1 sont basées sur les caractérisations de point fixe des modalités temporelles respectives. Elles peuvent être démontrées par induction structurelle sur les formules φ [EC80, EL86, Bra92] en utilisant les caractérisations itératives des formules de point fixe. Des exemples de preuves pour des opérateurs similaires aux modalités de Pdl-Δ seront donnés à la section 3.7.2.

Plusieurs remarques sur les traductions données dans la table 1.1 s'imposent.

- Les sous-formules $\langle true \rangle\, true$ qui apparaissent dans les traductions des modalités d'inévitabilité $\mathbf{AX}\varphi$ (*resp.* $\mathbf{A}[\varphi_1 \ \mathbf{U} \ \varphi_2]$ et $\mathbf{A}[\varphi_1{}_{\alpha_1}\mathbf{U}_{\alpha_2} \ \varphi_2]$) assurent l'existence des chemins respectifs dans le modèle ; elles sont nécessaires puisque ces opérateurs exigent d'atteindre finalement un état satisfaisant φ (*resp.* φ_2), alors que la relation de transition T n'est pas supposée totale (il peut exister des états de blocage). Aussi, des formules sur actions (en particulier, contenant des négations) sont nécessaires afin de pouvoir traduire les modalités d'ACTL (ceci n'aurait pas été possible dans le μ-calcul défini en [Koz83]).

- Les formules de μ-calcul obtenues par traduction ont une structure assez simple : elles ne contiennent qu'un seul opérateur de point fixe et la variable propositionnelle Y qu'il définit n'est libre dans aucun des arguments (φ, φ_1 ou φ_2) de la modalité respective. La notion d'*alternance* (*alternation depth*) [EL86] des formules du μ-calcul permet de construire une hiérarchie des fragments de cette logique, ayant une puissance d'expression (et aussi une complexité d'évaluation) croissante. Intuitivement, une formule φ est d'alternance n si le nombre maximal d'alternances des sous-formules non fermées μ et ν sur les chemins syntaxiques allant de la racine aux sous-formules atomiques de φ est égal à n. En particulier, les formules d'alternance 1 (*alternation-free*) ne contiennent pas d'opérateurs de plus petit et de plus grand point fixe mutuellement récursifs. Toutes les formules de μ-calcul données dans la table 1.1 appartiennent à cette classe ; par conséquent, les logiques temporelles CTL, ACTL et PDL-Δ sont traduisibles vers le fragment du μ-calcul d'alternance 1.

- En général, les formules temporelles contenant des modalités de chemins ($\mathbf{F}$, $\mathbf{G}$, $\mathbf{U}$) imbriquées se traduisent vers des formules de μ-calcul ayant des alternances supérieures. Ainsi, la modalité $\mathbf{E}\overset{\infty}{\mathbf{F}}$ et sa duale $\mathbf{A}\overset{\infty}{\mathbf{G}}$ d'ECTL, qui permettent d'exprimer des propriétés d'équité, se traduisent vers des formules de μ-calcul d'alternance 2 :

$$\begin{aligned} \mathbf{E}\overset{\infty}{\mathbf{F}}\varphi \overset{\mathrm{d}}{=} \mathbf{EGF}\varphi \ &= \ \nu Y_1.\mu Y_2.((\varphi \ \wedge \ \langle true \rangle\, Y_1) \ \vee \ \langle true \rangle\, Y_2) \\ \mathbf{A}\overset{\infty}{\mathbf{G}}\varphi \overset{\mathrm{d}}{=} \mathbf{AFG}\varphi \ &= \ \mu Y_1.\nu Y_2.((\varphi \ \vee \ [true]\, Y_1) \ \wedge \ [true]\, Y_2) \end{aligned}$$

La traduction de $\mathbf{E}\overset{\infty}{\mathbf{F}}\varphi$ est facile à vérifier en utilisant la caractérisation itérative 1.2 des formules de plus grand point fixe. La fonctionnelle Φ_1 associée à la formule en partie droite de l'égalité est définie comme suit (la formule étant fermée, l'environnement ρ peut être omis) :

$$\Phi_1(S') \overset{\mathrm{d}}{=} [\![\mu Y_2.((\varphi \ \wedge \ \langle true \rangle\, Y_1) \ \vee \ \langle true \rangle\, Y_2)]\!]\, [S'/Y_1] = [\![\mathbf{EF}(\varphi \ \wedge \ \langle true \rangle\, Y_1)]\!]\, [S'/Y_1]$$

La sémantique de la traduction de $\mathbf{E}\overset{\infty}{\mathbf{F}}\varphi$ est égale à la limite de la suite décroissante $\Phi_1^k(S)$:

$$\begin{aligned} &\Phi_1^1(S) = [\![\mathbf{EF}(\varphi \ \wedge \ \langle true \rangle\, true)]\!] \\ &\Phi_1^2(S) = [\![\mathbf{EF}(\varphi \ \wedge \ \langle true \rangle\, \mathbf{EF}(\varphi \ \wedge \ \langle true \rangle\, true))]\!] \\ &\ldots \end{aligned}$$

$\Phi_1^k(S)$ dénote donc les états à partir desquels il est possible d'atteindre k fois de suite un état satisfaisant φ ; par conséquent, la limite de la suite signifie "il est possible d'atteindre infiniment souvent un état satisfaisant φ".

A la différence de $\mathbf{E}\overset{\infty}{\mathbf{F}}$ et $\mathbf{A}\overset{\infty}{\mathbf{G}}$, les autres modalités d'équité d'ECTL se traduisent directement en CTL (donc en μ-calcul d'alternance 1) : $\mathbf{E}\overset{\infty}{\mathbf{G}}\varphi = \mathbf{EFEG}\varphi$ et $\mathbf{A}\overset{\infty}{\mathbf{F}}\varphi = \mathbf{AGAF}\varphi$.

Outre les traductions succinctes des logiques temporelles arborescentes mentionnées, il existe aussi des traductions en μ-calcul, plus élaborées, des logiques temporelles CTL* et ECTL* (et, par conséquent, de PTL) [Dam94a, BC96].

La logique de Dicky Une autre logique temporelle contenant des opérateurs de point fixe et permettant d'exprimer des propriétés sur états et sur actions, est la logique de Dicky [Dic86]. Cette logique est interprétée sur des systèmes de transitions paramétrés d'Arnold-Nivat [Arn92, ABC94] ; néanmoins, nous pouvons aussi lui donner une interprétation sur des modèles STM. La logique de Dicky comporte les opérateurs $\mathbf{in}, \mathbf{out} : 2^S \to 2^T$ et $\mathbf{src}, \mathbf{tgt} : 2^T \to 2^S$, permettant d'accéder aux successeurs et prédécesseurs des états et des transitions. Ces opérateurs ont la sémantique suivante :

$$\begin{aligned}
\mathbf{in}(S') &\stackrel{\mathrm{d}}{=} \left\{ s \xrightarrow{a} s' \in T \mid s' \in S' \right\} \\
\mathbf{out}(S') &\stackrel{\mathrm{d}}{=} \left\{ s \xrightarrow{a} s' \in T \mid s \in S' \right\} \\
\mathbf{src}(T') &\stackrel{\mathrm{d}}{=} \left\{ s \in S \mid \exists s' \in S. \exists a \in A. s \xrightarrow{a} s' \in T' \right\} \\
\mathbf{tgt}(T') &\stackrel{\mathrm{d}}{=} \left\{ s \in S \mid \exists s' \in S. \exists a \in A. s' \xrightarrow{a} s \in T' \right\}
\end{aligned}$$

où $S' \subseteq S$ et $T' \subseteq T$. Intuitivement, $\mathbf{in}(S')$ (*resp.* $\mathbf{out}(S')$) représente l'ensemble de toutes les transitions dont l'état d'arrivée (*resp.* l'état de départ) appartient à S' ; $\mathbf{src}(T')$ (*resp.* $\mathbf{tgt}(T')$) dénote l'ensemble de tous les états de départ (*resp.* d'arrivée) d'une transition appartenant à T'.

Les opérateurs temporels sont définis comme plus petites et plus grandes solutions de systèmes d'équations. A la différence du μ-calcul modal, la logique de Dicky permet, grâce aux opérateurs primitifs ci-dessus, d'exprimer de manière symétrique les propriétés portant sur le futur et sur le passé.

Exemple 1-13
L'opérateur $\mathbf{E}[\varphi_1 \mathbf{U} \varphi_2]$ de CTL peut être exprimé dans la logique de Dicky par l'équation de point fixe suivante :

$$X^+ = \varphi_2 \ \vee \ (\varphi_1 \ \wedge \ \mathbf{src}(\mathbf{in}(X)))$$

La variable X, similaire aux variables propositionnelles du μ-calcul, dénote un ensemble d'états (le signe + indique le fait qu'il s'agit d'une variable de plus petit point fixe). La plus petite solution de l'équation ci-dessus est précisément égale à l'ensemble d'états satisfaisant $\mathbf{E}[\varphi_1 \mathbf{U} \varphi_2]$. ■

Exemple 1-14
L'équation ci-dessous définit l'opérateur $\mathbf{reach}(\varphi)$ [Arn89], qui caractérise l'ensemble des états atteignables à partir des états qui satisfont φ :

$$X^+ = \varphi \ \vee \ \mathbf{tgt}(\mathbf{out}(X))$$

Cet opérateur porte sur le passé. ■

Des descriptions détaillées de la logique de Dicky, ainsi que différents exemples de propriétés exprimables dans cette logique, peuvent être trouvées en [Dic86, Arn92, ABC94].

Enfin, nous pouvons mentionner d'autres logiques avec des opérateurs de point fixe qui ont été proposées dans la littérature, comme la logique STL (*Synchronization Tree Logic*) [GS86] ou la logique HML avec récursion [Lar88].

1.2.5 Logiques temporelles étendues avec des valeurs

Les logiques modales et temporelles mentionnées jusqu'ici sont interprétées sur des modèles contenant des actions atomiques et/ou des constantes propositionnelles associées aux états. En particulier, les logiques interprétées sur actions sont adaptées pour spécifier les propriétés des programmes décrits dans des algèbres de processus “pures” (*pure* CCS, *basic* LOTOS, etc.), c'est-à-dire ne contenant pas de communication avec passage de valeurs. En revanche, ces logiques ne sont pas adaptées pour les langages de description basés sur valeurs (*full* CCS, *full* LOTOS, μCRL, etc.). Ceci a motivé l'introduction de nouvelles logiques temporelles, capables d'exprimer des propriétés sur les valeurs manipulées dans les programmes à vérifier. Nous présentons ci-dessous quelques-unes de ces logiques.

La logique modale de μCRL La logique modale de μCRL [GvV94] est une logique arborescente basée sur actions, dédiée à la description des propriétés temporelles des programmes μCRL. Cette logique peut être vue comme une extension d'ACTL* avec des opérateurs sur le passé. Les propriétés sur valeurs sont exprimées au moyen de variables typées, de prédicats d'égalité sur les termes et de quantificateurs du premier ordre.

Cette logique est interprétée sur les modèles des programmes μCRL, qui sont des systèmes de transitions ayant des actions structurées (noms de canaux de communication et listes de valeurs échangées) mais ne contenant pas d'informations sur les états. Nous donnons ci-dessous la définition d'un fragment arborescent pur-futur de cette logique, interprété sur des modèles STE étendus (voir la définition 1-4) qui n'ont pas d'informations attachées aux états.

Quelques notions auxiliaires sont nécessaires. Soit $DVar$ un ensemble de variables typées, notées $y_1, y_2, \ldots$. Chaque variable y_i a un type T_i, et l'ensemble des valeurs appartenant aux types T_i est noté **Val**. Nous définissons le domaine $\mathbf{DEnv} \stackrel{\mathrm{d}}{=} DVar \to \mathbf{Val}$ des *environnements*. Un environnement $\varepsilon \in \mathbf{DEnv}$ est une fonction partielle associant à chaque variable $y_i \in supp(\varepsilon)$ une valeur $\varepsilon(y_i) \in T_i$. Nous supposons aussi l'existence d'un ensemble de fonctions $Func$, notées $f_1, f_2, \ldots$. Chaque fonction dénote une application $f_i : T_i^1 \times \ldots \times T_i^{n_i} \to T_i$, où n_i est l'arité de f_i, $T_i^1, \ldots, T_i^{n_i}$ sont les types des arguments et T_i est le type du résultat de f_i.

Définition 1-13 (Syntaxe et sémantique d'un fragment de la logique modale de μCRL)

La logique modale de μCRL contient des termes $t \in Term$, des formules sur actions $\alpha \in AForm$, des formules sur états $\varphi \in SForm$ et des formules sur chemins $\psi \in PForm$, ayant la syntaxe suivante :

$$\begin{array}{lcl} t & ::= & y \mid f(t_1, \ldots, t_n) \\ \alpha & ::= & c(t_1, \ldots, t_n) \\ \varphi & ::= & true \mid t_1 = t_2 \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \exists\psi \\ \psi & ::= & \varphi \mid \neg\psi_1 \mid \psi_1 \wedge \psi_2 \mid @\,\psi_1 \mid \psi_1 \ \mathbf{U}\ \psi_2 \mid \exists y_1{:}T_1, \ldots, y_2{:}T_2 . \psi_1 \end{array}$$

où y est une variable typée, f est une fonction, c est un nom de canal de communication et $=$ dénote l'égalité des valeurs. Les opérateurs booléens dérivés sont définis de manière habituelle. Les modalités temporelles dérivées sont définies comme suit : $\forall\psi \stackrel{\mathrm{d}}{=} \neg\exists\neg\psi$, $\Diamond\psi \stackrel{\mathrm{d}}{=} true\ \mathbf{U}\ \psi$, $\Box\psi \stackrel{\mathrm{d}}{=} \neg\Diamond\neg\psi$.

Une occurrence de variable y dans une formule ψ est dite *liée* si elle est contenue dans une sous-formule $\exists y{:}T \ldots .\psi'$ ou $\forall y{:}T \ldots .\psi'$ de ψ. Toutes les autres occurrences de variables y sont dites *libres*.

La sémantique des termes est donnée par la fonction d'interprétation $[\![.]\!] : Term \to \mathbf{DEnv} \to \mathbf{Val}$ définie inductivement ci-dessous. Pour un terme t et un environnement ε (qui doit initialiser toutes les variables contenues dans t), la dénotation $[\![t]\!]\,\varepsilon$ renvoie la valeur de t dans le contexte de ε :

$$\begin{array}{rcl} [\![y]\!]\,\varepsilon & \stackrel{\mathrm{d}}{=} & \varepsilon(y) \\ [\![f(t_1, \ldots, t_n)]\!]\,\varepsilon & \stackrel{\mathrm{d}}{=} & f([\![t_1]\!]\,\varepsilon, \ldots, [\![t_n]\!]\,\varepsilon) \end{array}$$

La sémantique des formules sur actions est donnée par la fonction d'interprétation $[\![.]\!] : AForm \to \mathbf{DEnv} \to 2^A$ définie ci-dessous. Pour une action α et un environnement ε (qui doit initialiser toutes les variables contenues dans α), la dénotation $[\![\alpha]\!]\,\varepsilon$ renvoie l'ensemble d'actions du modèle qui satisfont α dans le contexte de ε :

$$[\![c(t_1,\ldots,t_n)]\!]\,\varepsilon \stackrel{\mathrm{d}}{=} \{a \in A \mid a = c\,[\![t_1]\!]\,\varepsilon \ldots\ [\![t_n]\!]\,\varepsilon\}$$

La sémantique des formules sur états (*resp.* sur chemins) est définie au moyen d'une fonction d'interprétation $[\![.]\!]$: *SForm* $\rightarrow$ **DEnv** $\rightarrow 2^S$ (*resp.* $\|.\|$: *PForm* $\rightarrow$ **DEnv** $\rightarrow 2^R$). Pour une formule φ (*resp.* ψ) et un environnement ε — qui doit initialiser toutes les variables libres dans φ (*resp.* ψ) — la dénotation $[\![\varphi]\!]\,\varepsilon$ (*resp.* $\|\psi\|\varepsilon$) renvoie l'ensemble d'états (*resp.* de chemins maximaux) du modèle qui satisfont φ (*resp.* ψ) dans le contexte de ε. Les fonctions sémantiques sont définies inductivement comme suit :

$$\begin{array}{rcl}
[\![\mathit{true}]\!]\,\varepsilon & \stackrel{\mathrm{d}}{=} & S \\
[\![t_1 = t_2]\!]\,\varepsilon & \stackrel{\mathrm{d}}{=} & \mathit{if}\ \ [\![t_1]\!]\,\varepsilon = [\![t_2]\!]\,\varepsilon\ \mathit{then}\ S\ \mathit{else}\ \emptyset \\
[\![\neg\varphi_1]\!]\,\varepsilon & \stackrel{\mathrm{d}}{=} & S \setminus [\![\varphi_1]\!]\,\varepsilon \\
[\![\varphi_1 \ \wedge\ \varphi_2]\!]\,\varepsilon & \stackrel{\mathrm{d}}{=} & [\![\varphi_1]\!]\,\varepsilon \cap [\![\varphi_2]\!]\,\varepsilon \\
[\![\exists\psi]\!]\,\varepsilon & \stackrel{\mathrm{d}}{=} & \{s \in S \mid \exists\pi \in R.\pi(0) = s\ \wedge\ \pi \in \|\psi\|\varepsilon\}
\end{array}$$

et

$$\begin{array}{rcl}
\|\varphi\|\,\varepsilon & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid \pi(0) \in [\![\varphi]\!]\,\varepsilon\} \\
\|\neg\psi_1\|\,\varepsilon & \stackrel{\mathrm{d}}{=} & R \setminus [\![\psi_1]\!]\,\varepsilon \\
\|\psi_1\ \wedge\ \psi_2\|\,\varepsilon & \stackrel{\mathrm{d}}{=} & [\![\psi_1]\!]\,\varepsilon \cap [\![\psi_2]\!]\,\varepsilon \\
\|@\,\psi_1\|\,\varepsilon & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid \exists k \geq 0.\exists k' > k.l(\pi,k) \in \|\alpha\|\varepsilon\ \wedge\ \pi^{k'} \in \|\psi_1\|\varepsilon\ \wedge \\
 & & \qquad \forall 0 \leq i < k.l(\pi,i) = \tau\ \wedge\ \forall k < j < k'.l(\pi,j) = \tau\} \\
\|\psi_1 \mathbf{U} \psi_2\|\,\varepsilon & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid \exists k \geq 0.\pi^k \in \|\psi_2\|\varepsilon\ \wedge\ \forall 0 \leq i < k.\pi^i \in \|\psi_1\|\varepsilon\} \\
\|\exists y_1{:}T_1,\ldots,y_2{:}T_2.\psi_1\|\,\varepsilon & \stackrel{\mathrm{d}}{=} & \{\pi \in R \mid \exists v_1{:}T_1.\ldots\ \exists v_n{:}T_n.\pi \in \|\psi_1\|(\varepsilon \oslash [v_1/y_1,\ldots,v_n/y_n])\}
\end{array}$$

■

La sémantique des modalités **U**, $\Diamond$ et $\Box$ est similaire aux opérateurs correspondants **U**, **F** et **G** d'ACTL*. La modalité $@\,\psi$ a une sémantique similaire à la formule $\mathit{true}\,\langle\alpha\rangle\,\psi$ d'ACTL* : elle signifie qu'il est possible d'atteindre, après un nombre quelconque de τ-transitions, une α-transition conduisant à son tour (toujours après une séquence de τ-transitions) à un état satisfaisant ψ.

Les quantificateurs sur les variables typées sont le mécanisme essentiel pour exprimer des propriétés sur les valeurs.

Exemple 1-15
La formule μCRL suivante spécifie le fait que chaque émission d'un message est inévitablement suivie par la réception du même message :

$$\forall\Box\forall m{:}M.\big(s(m)\ \mathit{true} \Rightarrow \Diamond\ r(m)\ \mathit{true}\big)$$

où la variable m de type M sert à représenter la valeur du message et les canaux s et r correspondent respectivement à l'émission et à la réception. Pour décrire cette propriété dans une logique qui ne comporte pas de valeurs (comme ACTL*), il aurait fallu écrire une formule différente pour chaque message m contenu dans les actions s du modèle. ■

Autres logiques avec valeurs Plusieurs autres logiques avec valeurs (qui peuvent être vues comme des extensions de logiques classiques) ont été proposées, mais dans des contextes quelque peu différents. Ainsi, une logique modale similaire à Hml, étendue avec des variables quantifiées, a été définie en [HL93]. Ultérieurement, cette logique a été étendue avec des opérateurs de point fixe, similaires à ceux du μ-calcul modal, paramétrés par des variables typées [RH96]. Ces deux logiques sont interprétées sur des systèmes de transitions *symboliques*, dont chaque état correspond à un terme non nécessairement clos (c'est-à-dire, pouvant contenir des variables libres) d'une algèbre de processus avec valeurs. La génération de ces systèmes est basée sur la notion de sémantique opérationnelle symbolique [HL92] qui représente une généralisation de la sémantique opérationnelle standard des algèbres de processus avec valeurs. Cependant, même si la génération du système de transitions symbolique peut être automatisée dans certains cas (le système symbolique fini obtenu représentant les comportements infinis du programme), il n'en est pas de même pour la vérification des propriétés temporelles. En effet, la présence des prédicats du premier ordre sur les valeurs nécessite un démonstrateur de théorèmes auxiliaire permettant de vérifier ces prédicats sur les domaines des valeurs respectives.

Une autre logique modale étendue avec des variables quantifiées et des opérateurs de point fixe paramétrés a été définie en [Dam94b], dans le contexte des programmes décrits en π-calcul polyadique. Les valeurs utilisées sont des noms de canaux de communication (valeurs d'un type énuméré) échangés entre les processus mobiles, et les prédicats sont des égalités entre les noms de canaux.

Enfin, nous pouvons mentionner — dans la classe des logiques temporelles interprétées sur états — la logique définie en [MP92]. Il s'agit d'une logique temporelle linéaire, avec des opérateurs sur le passé, étendue avec des variables quantifiées permettant d'exprimer des propriétés sur les variables d'état des programmes. Le modèle d'interprétation peut être vu comme un Ste étendu ne contenant pas d'informations sur les actions.

1.3 Evaluation des propriétés temporelles sur un modèle

La spécification logique d'un programme parallèle consiste habituellement en une liste de formules de logique temporelle, chacune exprimant une propriété de bon fonctionnement du programme. Le problème de la vérification basée sur les modèles est de déterminer si ces formules sont satisfaites par le modèle du programme, ce qui revient à *évaluer* la valeur de vérité de chaque formule sur le modèle. On distingue généralement deux classes de méthodes d'évaluation des formules temporelles sur un modèle :

les méthodes globales, qui utilisent une représentation explicite du modèle (le modèle devant être généré complètement *avant* de commencer l'évaluation de la formule) ;

les méthodes locales, aussi appelées "à la volée" (*on-the-fly*), qui utilisent une représentation implicite du modèle (le modèle étant généré *pendant* l'évaluation de la formule).

Chacune de ces approches a été étudiée dans le contexte de différentes logiques temporelles ; de nombreux algorithmes et outils ont été développés. Cette section contient une présentation et une comparaison des principaux résultats existants.

1.3.1 Evaluation globale

Les méthodes d'évaluation globale ont été employées pour beaucoup de logiques temporelles. Ces méthodes fonctionnent selon le schéma indiqué dans la figure 1.4. D'abord, le modèle du programme parallèle est généré à l'aide d'un compilateur. Ensuite, les formules de logique temporelle constituant la spécification du programme sont vérifiées sur le modèle grâce à un évaluateur, qui fournit en sortie leur valeur de vérité, éventuellement accompagnée d'un diagnostic.

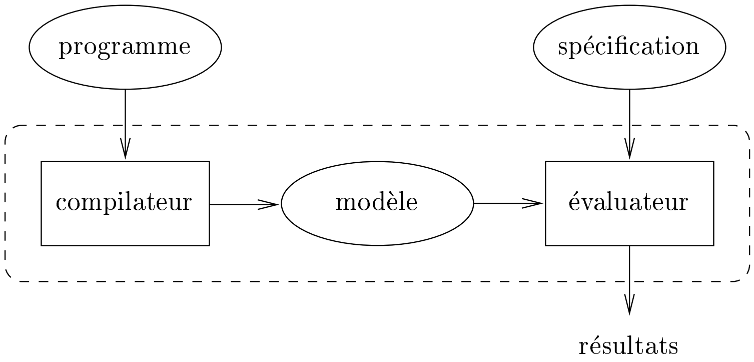

Figure 1.4: Principe de l'évaluation globale

Les méthodes globales présentent l'avantage de factoriser l'effort de construction du modèle, qui n'est généré qu'une seule fois même si plusieurs formules de logique temporelle doivent être évaluées (ce qui en pratique est souvent le cas). En outre, du fait que le modèle est entièrement connu (nombre d'états, de transitions, etc.), les méthodes globales conduisent à des algorithmes ayant une bonne complexité dans le pire des cas.

Les paragraphes suivants présentent brièvement les méthodes d'évaluation globale utilisées pour différentes logiques temporelles linéaires et arborescentes.

Evaluation des logiques temporelles linéaires L'interprétation des logiques temporelles linéaires est définie en termes de séquences (infinies) d'exécution : un modèle $\mathcal{K}$ satisfait une formule ψ ssi toutes les séquences issues de l'état initial qu'il contient satisfont ψ. Cette sémantique permet de ré-formuler le problème de l'évaluation de ψ sur $\mathcal{K}$ comme un problème de la théorie des automates sur mots infinis [Cho74] : le modèle $\mathcal{K}$ (représenté comme un automate de Büchi $\mathcal{B}_1$) vérifie la formule ψ (préalablement traduite vers un automate de Büchi $\mathcal{B}_2$) ssi toutes les séquences infinies de $\mathcal{B}_1$ sont acceptées par $\mathcal{B}_2$. Les algorithmes d'évaluation dédiés aux logiques temporelles linéaires [LP85, SVW87, JJ89] fonctionnent sur ce principe.

Comme nous l'avons mentionné en section 1.2.1, le problème de l'évaluation des logiques temporelles linéaires (comme PTL) est PSPACE-complet [SC85]. Cependant, les algorithmes existants ont généralement une complexité exponentielle en taille de la formule et linéaire en taille du modèle, ce qui en pratique peut donner des performances acceptables, la taille de la formule (nombre d'opérateurs) étant beaucoup plus petite que la taille (nombre d'états) du modèle.

Une autre méthode d'évaluation des logiques temporelles linéaires consiste à traduire ces logiques en (un fragment du) μ-calcul modal et à évaluer ensuite les formules obtenues en utilisant des algorithmes d'évaluation dédiés au (fragment respectif du) μ-calcul. Ceci permet d'obtenir des algorithmes ayant une complexité comparable aux algorithmes spécialisés mentionnés plus haut (voir le paragraphe suivant).

Evaluation des logiques temporelles arborescentes Les logiques temporelles purement arborescentes (comme CTL ou ACTL) sont interprétées en termes d'états : un modèle satisfait une formule φ ssi tous ses états satisfont φ. Les algorithmes d'évaluation globale dédiés aux logiques arborescentes doivent donc déterminer, d'une manière ou d'une autre, si tous les états du modèle satisfont la formule. Suivant le codage de la relation entre la formule et les états du modèle qui la satisfont, différents types d'algorithmes ont été développés.

Une première approche consiste à calculer, pour chaque formule, l'ensemble d'états du modèle qui la satisfont. Ceci constitue une implémentation directe de l'interprétation des formules (voir la définition 1-7 de Ctl), où la sémantique d'une formule est définie de manière inductive en fonction de la sémantique de ses sous-formules. Les ensembles d'états associés aux opérateurs booléens ($\wedge$, $\vee$, $\neg$) sont calculés à l'aide des opérations ensemblistes correspondantes ; les ensembles d'états associés aux opérateurs modaux (**EX**, **AX**, $\langle . \rangle$, $[.]$) sont calculés par exploration des prédécesseurs des états dans la relation de transition ; enfin, les ensembles d'états associés aux opérateurs temporels (**EU**, **AU**, μ, ν) sont calculés par itérations, en utilisant la caractérisation de point fixe de la modalité respective. Ce genre d'algorithmes ont été développés pour Ltac et Stl [Que82, Sch83, Rod88], ainsi que pour le μ-calcul modal [EL86, LBC+94, And94].

Une deuxième approche consiste à calculer, pour chaque état, l'ensemble des formules (sous-formules contenues dans la formule à vérifier) qu'il satisfait. Généralement, cet ensemble de formules est codé, pour chaque état, comme un tableau de booléens (mémorisés comme des bits), ce qui justifie le nom d'algorithmes basés sur des vecteurs de bits (*bit-vector-based*) employé pour cette classe d'algorithmes. Le calcul de l'ensemble de formules est fait par propagation, en explorant simultanément les dépendances induites par la relation de transition du modèle et par la relation d'inclusion syntaxique des formules. Ce genre d'algorithmes ont été développés pour la logique de Dicky [AC88] et le μ-calcul modal d'alternance 1 [CS91b, CS93]. Des algorithmes similaires sont utilisés pour Ctl [CES86] et Actl [BGL94]. Ces algorithmes ont une complexité linéaire en taille du modèle et de la formule, étant généralement meilleurs que ceux basés sur le calcul des ensembles d'états.

Une troisième approche, similaire avec celle basée sur les vecteurs de bits, consiste à traduire le modèle et la formule à vérifier vers un système d'équations booléennes ayant une variable associée à chaque couple état-formule. Intuitivement, la variable associée à (s, φ) est égale à vrai ssi s satisfait φ. Le modèle satisfait la formule φ ssi, pour tous les états $s \in S$, la solution du système booléen positionne à vrai les variables associées aux couples (s, φ). De tels algorithmes ont été développés pour le μ-calcul modal d'alternance 1 [VL92, And94]. Des algorithmes similaires, basés sur la traduction vers des systèmes d'équations modales, ont été développés pour le μ-calcul d'alternance n [CKS92].

Il convient de remarquer qu'il est possible d'obtenir des algorithmes d'évaluation pour des logiques temporelles particulières en traduisant ces logiques vers d'autres logiques plus expressives et en appliquant ensuite les algorithmes d'évaluation sous-jacents. Ainsi, une attention particulière a été portée au fragment du μ-calcul modal d'alternance 1, qui est suffisamment expressif pour permettre la traduction de plusieurs logiques temporelles comme Ctl, Actl ou Pdl-Δ (voir la section 1.2.3). Les algorithmes d'évaluation du μ-calcul d'alternance 1 développés en [CS91b, VL92, And94] ont une complexité linéaire en taille du modèle et de la formule, ce qui permet d'obtenir, par traduction, des algorithmes ayant une complexité comparable aux algorithmes spécialisés dédiés aux logiques temporelles particulières mentionnées ci-dessus. D'autres fragments du μ-calcul modal ont été étudiés, notamment ceux qui correspondent aux traductions des logiques Ctl* et Ectl*, pour lesquels des algorithmes spécialisés ont été développés [EJS93].

Cette approche d'évaluation par traduction nécessite en pratique la présence d'un mécanisme d'abstraction permettant de définir et d'utiliser des opérateurs de logique temporelle paramétrés par des formules. Ce type de mécanisme est employé dans plusieurs outils de vérification, comme Mec [Arn89, ABC94], Cwb [CPS89] ou Concurrency Factory [CLSS96].

1.3.2 Evaluation locale

Les méthodes d'évaluation globale nécessitent la construction préalable du modèle avant de commencer l'évaluation des formules. Toutefois, dans certains cas — notamment, lorsqu'une propriété de vivacité (*resp.* de sûreté) est (*resp.* n'est pas) satisfaite par le modèle — il n'est pas nécessaire de générer entièrement le modèle afin de déterminer la valeur de vérité d'une formule. Cette observation est à la base du développement des méthodes d'évaluation locales, qui fonctionnent selon le schéma indiqué dans la figure 1.5. A la différence des méthodes globales, l'évaluation des formules a lieu simultanément avec la génération du modèle : l'environnement de vérification contient un compilateur pour le langage des programmes à vérifier et un évaluateur pour les formules temporelles, ces deux outils fonctionnant conjointement.

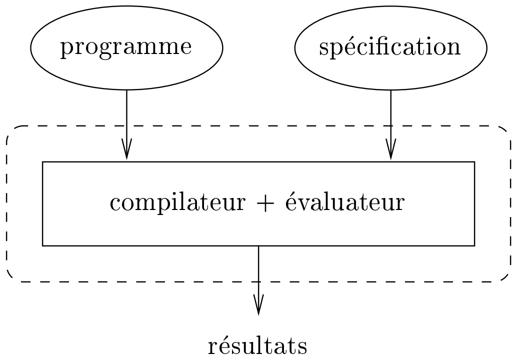

Figure 1.5: Principe de l'évaluation locale

Les méthodes locales présentent l'avantage de ne pas générer complètement le modèle lorsqu'une propriété de sûreté à vérifier est fausse (de ce fait, elles sont utiles dans les premières phases du processus de conception, quand les erreurs sont susceptibles d'être les plus fréquentes). En revanche, comme le modèle n'est pas entièrement connu au moment de l'évaluation des formules, les méthodes locales conduisent à des algorithmes ayant une complexité légèrement supérieure (d'un facteur logarithmique en taille du modèle) par rapport aux algorithmes globaux correspondants.

Les paragraphes suivants présentent brièvement les méthodes d'évaluation locale utilisées pour différentes logiques temporelles linéaires et arborescentes.

Evaluation des logiques temporelles linéaires Une première approche pour l'évaluation locale des formules de logique temporelle linéaire est basée sur la théorie des automates de mots infinis. A la différence avec les méthodes globales pour les logiques temporelles linéaires mentionnées dans la section 1.3.1, le produit des automates de Büchi correspondant à la formule et au modèle est calculé à la volée, par exploration en profondeur de l'espace des états-produit. Cette méthode, optimisée avec diverses stratégies de stockage des états visités, est employée dans plusieurs algorithmes d'évaluation [CVWY90, JJ91].

Une deuxième approche est basée sur les systèmes de preuve associés aux logiques temporelles linéaires. Elle consiste à construire, en partant du modèle et de la formule à vérifier, la preuve que le modèle (c'est-à-dire tous les chemins issus de l'état initial) satisfait la formule. Une preuve est modélisée comme un graphe orienté (*proof structure*) dont les sommets sont étiquetés par des assertions $s \vdash \varphi$ et les arcs correspondent aux implications. La construction de la preuve est effectuée par une exploration en profondeur simultanée du modèle et de la formule. Cette méthode est utilisée pour l'évaluation locale des formules de CTL* (et, par conséquent, de PTL) [BCG95].

Une troisième approche consiste à traduire les logiques temporelles linéaires vers d'autres formalismes plus expressifs, comme le μ-calcul modal, et à appliquer ensuite les algorithmes locaux sous-jacents (voir le paragraphe suivant).

Evaluation des logiques temporelles arborescentes Une première classe d'algorithmes d'évaluation locale des logiques arborescentes est basée sur la traduction du modèle et de la formule à vérifier vers des systèmes d'équations booléennes, ayant une variable associée à chaque couple état-formule. Cette traduction peut être effectuée par une exploration simultanée "en avant" (en profondeur ou en largeur) du modèle et de la formule. La résolution du système booléen est faite pendant sa construction, à l'aide d'algorithmes spécialisés qui calculent uniquement une solution partielle du système, suffisante pour établir la valeur de vérité de la formule. Plusieurs algorithmes d'évaluation dédiés à CTL [VL93] et au μ-calcul modal d'alternance 1 [And94, VWL94, VL94] et d'alternance 2 [VL94] ont été ainsi développés. La complexité de ces algorithmes dans le pire des cas (quand l'évaluation de la formule nécessite la génération de tout le modèle) est augmentée d'un facteur logarithmique, en taille du modèle, par rapport aux algorithmes globaux correspondants. Néanmoins, lorsque les formules ne sont pas satisfaites par le modèle, les algorithmes locaux peuvent s'arrêter de manière anticipée, ce qui, en pratique, donne de bonnes performances.

Une deuxième classe d'algorithmes locaux, utilisée essentiellement pour le μ-calcul modal, est basée sur la méthode des tableaux (*tableau-based*). Ces algorithmes construisent, au moyen de règles d'inférence, une preuve qu'un état du modèle satisfait une (sous-)formule. Le calcul des tableaux est fait selon diverses stratégies d'application des règles d'inférence ; ces stratégies ont en commun le fait que la relation de transition est explorée "en avant", permettant ainsi la génération du modèle à la volée. Les méthodes basées sur les tableaux ont été utilisées dans de nombreux algorithmes d'évaluation locale du μ-calcul modal [Lar88, Cle90, SW91, Win91, Lar92]. Ces algorithmes ont généralement une complexité exponentielle, même pour les formules d'alternance 1 (excepté l'algorithme proposé en [Lar92] qui, pour cette classe de formules, a une complexité quadratique), étant donc moins efficaces que les algorithmes globaux correspondants. Cependant, les algorithmes basés sur les tableaux présentent l'avantage de permettre plus facilement la génération de diagnostics (sous forme de formules modales) expliquant la valeur de vérité d'une formule sur un modèle [Lar92]. Les méthodes basées sur tableaux ont été généralisées pour la vérification de systèmes infinis [Bra92, Dam94b, RH96].

1.4 Discussion

Notre travail de thèse concerne la spécification et la vérification, par évaluation de propriétés temporelles, des programmes décrits dans des langages parallèles basés sur valeurs (en particulier LOTOS). Ceci nécessite, d'une part, la conception d'un langage de spécification approprié et, d'autre part, le développement d'algorithmes d'évaluation associés. La synthèse présentée dans ce chapitre, qui résume les principaux résultats existants dans le domaine, nous a permis de dégager les aspects essentiels à notre étude, qui ont conduit aux choix de conception suivants :

- Le modèle de représentation des programmes doit être adapté aux algèbres de processus avec communication de valeurs (comme *full* CCS, LOTOS ou μCRL), ainsi qu'aux langages de description avec variables d'état (comme ESTELLE ou SDL). A la section 1.1.3, nous avons proposé un tel modèle, appelé système de transitions étiquetées étendu (voir la définition 1-4), contenant des informations sur les états aussi bien que sur les actions. Ce modèle, qui généralise à la fois les structures de Kripke et les systèmes de transitions étiquetées, peut être utilisé comme base pour interpréter toutes les logiques temporelles présentées dans ce chapitre.

- La logique temporelle utilisée pour spécifier les propriétés doit être appropriée pour les langages de description ayant une sémantique d'entrelacement (en particulier, les algèbres de processus). Comme nous l'avons précisé à la section 1.2.1, les logiques linéaires ne sont pas adaptées aux algèbres de processus, car elles ne permettent pas de prendre en compte les branchements de l'exécution. Il est donc naturel de choisir, comme base du langage de spécification, une logique temporelle arborescente. En outre, cette logique doit pouvoir exprimer des propriétés portant sur les états aussi bien que sur les actions, comme les logiques présentées à la section 1.2.3.

- Un aspect important est la possibilité de décrire les propriétés temporelles de manière concise et naturelle. Les formules sur actions (voir la section 1.2.2), construites avec les opérateurs booléens $\wedge$, $\vee$ et $\neg$, permettent d'exprimer de façon concise les propriétés portant sur des actions individuelles du modèle. Des propriétés plus complexes, caractérisant des séquences d'actions de longueur non bornée du modèle, peuvent être décrites naturellement au moyen d'expressions régulières (voir la section 1.2.3) construites avec des formules sur actions.

- Le formalisme de spécification doit être suffisamment expressif pour autoriser la description de toutes les classes de propriétés intéressantes des programmes (propriétés de sûreté, de vivacité et d'équité). En particulier, ce formalisme doit permettre une traduction aisée des opérateurs temporels appartenant aux diverses logiques temporelles couramment utilisées. Ceci peut être réalisé au moyen d'opérateurs de plus petit et de plus grand point fixe, qui servent de base à des logiques très expressives, comme le μ-calcul modal et la logique de Dicky (voir la section 1.2.4). Des mécanismes d'abstraction sont également nécessaires, permettant de définir et d'utiliser des opérateurs temporels paramétrés par des formules.

- Afin de pouvoir exprimer des propriétés concernant les valeurs manipulées par le programme à vérifier, il est nécessaire d'enrichir le formalisme de spécification avec des variables typées et des prédicats du premier ordre, comme ceux présents dans la logique modale de μCRL (voir la section 1.2.5). Une autre extension utile, employée dans plusieurs logiques basées sur les valeurs mentionnées à la section 1.2.5, est le paramétrage des opérateurs de point fixe par des variables typées. Ces opérateurs doivent être complétés par des mécanismes d'extraction des valeurs contenues dans les états et les actions du modèle, afin de pouvoir exploiter toutes les informations provenant du programme source à vérifier.

- Enfin, le choix d'une logique temporelle particulière, si expressive soit-elle, peut ne pas faire face à toutes les situations rencontrées en pratique, qui exigent parfois la spécification de propriétés temporelles difficiles (voire même impossibles) à exprimer dans les logiques classiques. Il est donc nécessaire de disposer, en dehors des opérateurs temporels expressifs (comme les opérateurs de point fixe paramétrés et les expressions régulières), de mécanismes permettant la définition d'opérateurs temporels non-standard. Ceci peut être réalisé, par exemple, au moyen de constructions spéciales autorisant l'accès aux états du STE et l'exploration de la relation de transition (comme les opérateurs du calcul de Dicky), grâce auxquelles de nouveaux opérateurs temporels complexes peuvent être définis par calcul itératif d'ensembles d'états.

Les chapitres suivants présentent le langage de spécification de propriétés temporelles que nous avons conçu suivant les critères mentionnés ci-dessus, les algorithmes d'évaluation associés que nous avons développé, ainsi que des exemples d'applications du langage et de l'outil de vérification.

Chapitre 2

Présentation du langage XTL

Afin de remédier aux limitations des formalismes de spécification "classiques" (logiques temporelles, μ-calcul modal), nous proposons un langage de spécification des propriétés temporelles appelé Xtl (*eXecutable Temporal Language*), conçu selon les principes dégagés au chapitre 1. Xtl présente l'avantage d'intégrer, d'une part, des opérateurs temporels expressifs (comme les formules d'actions de la logique Actl, les expressions régulières de la logique Pdl et les opérateurs de point fixe du μ-calcul modal) et, d'autre part, des constructions apparentées aux langages de programmation fonctionnels (comme les expressions "**let**", "**if**", "**case**", "**loop**", etc.). Etendues avec des mécanismes de filtrage capables d'extraire l'information contenue dans les états et les actions du modèle Ste, ces constructions permettent d'exprimer de manière concise et naturelle les propriétés temporelles portant sur les valeurs, en utilisant directement les notations du programme à vérifier.

Un autre aspect original du langage Xtl est la présence de *méta-opérateurs* de manipulation des éléments du modèle Ste (états, étiquettes, transitions). Utilisés dans les formules temporelles, ces méta-opérateurs offrent un moyen sémantique propre pour exprimer des prédicats sur les états ou sur les actions du Ste. En outre, Xtl offre des méta-opérateurs (similaires à ceux du calcul de Dicky) d'exploration de la relation de transition du Ste, ce qui permet la description de propriétés temporelles non-standard, ainsi que le calcul de diverses informations sur le Ste (nombre d'états, facteur de branchement, etc.). L'évaluation des formules est effectuée au moyen de méta-opérateurs d'évaluation, permettant soit de calculer l'ensemble d'états (ou d'actions) satisfaisant une formule, soit de vérifier si un état particulier (ou une action particulière) satisfait une formule. Des mécanismes d'abstraction permettent la définition de fonctions et de formules paramétrées, ainsi que la construction de bibliothèques d'opérateurs réutilisables implémentant des logiques temporelles particulières.

Ce chapitre constitue un manuel du langage Xtl. Après une description de la grammaire abstraite et des aspects lexicographiques, les différentes constructions du langage sont présentées par ordre croissant de complexité, en commençant par les types de données, les variables, les expressions et les fonctions, en continuant avec les formules sur actions, les expressions régulières et les formules sur états, et en terminant avec les méta-opérateurs d'évaluation des formules. La sémantique (statique et dynamique) de chaque construction du langage est décrite informellement et de nombreux exemples d'applications sont fournis, qui illustrent les avantages de Xtl par rapport aux formalismes classiques.

Tout au long de ce chapitre, nous considérons implicitement un modèle Ste étendu $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$ (voir la définition 1-4), généré à partir d'un programme parallèle et représenté sous forme d'un fichier en format Bcg [Gar94]. Les différentes constructions du langage Xtl seront interprétées sur ce modèle Ste.

2.1 Grammaire abstraite

Nous commençons par présenter la grammaire abstraite du langage XTL. Dans la suite du document, cette grammaire servira de base pour la définition de la sémantique dénotationnelle des formules (chapitre 3) et des expressions (annexe B).

2.1.1 Notations

Les symboles terminaux du langage XTL sont donnés dans la table 2.1. La structure lexicale des constantes littérales K sera décrite informellement à la section 2.3. Les autres symboles terminaux dénotent des identificateurs, qui seront définis à la section 2.2.

Terminal	Signification
K	constante littérale
x	variable simple
T	type
G	porte
C	constructeur
F	fonction
Y	variable propositionnelle
AF	macro-formule sur actions
SF	macro-formule sur états
PAR	paramètre de macro-formule
FN	nom de fichier

Table 2.1: Les symboles terminaux

Les symboles non-terminaux du langage sont donnés dans la table 2.2. Les constructions dénotées par ces symboles seront présentées aux sections suivantes.

Non-terminal	Signification
P	filtre (*pattern*)
RT	type résultat
O	offre
α	formule sur actions
R	expression régulière
φ	formule sur états
E	expression
D	définition de fonction
M	définition de formule
L	inclusion de bibliothèques
PG	programme

Table 2.2: Les symboles non-terminaux

Afin de faciliter la lecture de ce chapitre, les dépendances entre les symboles non-terminaux (accompagnés du numéro des sections où ils sont respectivement présentés) sont données dans la figure 2.1. Une flèche "$N_1 \longrightarrow N_2$" signifie que le symbole non-terminal N_2 apparaît dans la partie droite d'une règle syntaxique associée au symbole non-terminal N_1.

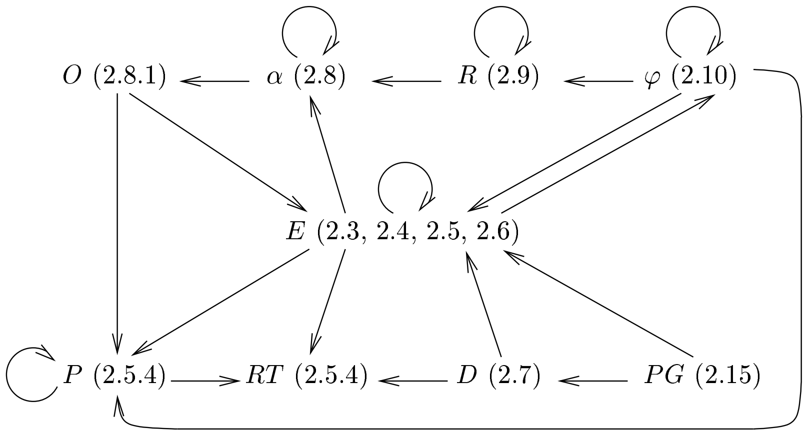

Figure 2.1: Les dépendances entre les symboles non-terminaux

2.1.2 Règles syntaxiques

Cette section contient les règles syntaxiques définissant la grammaire abstraite de XTL. Etant donné que les règles associées aux symboles non-terminaux seront reprises dans la suite de ce chapitre (aux sections indiquées sur la figure 2.1), le reste de cette section peut être évité lors d'une première lecture. En revanche, la définition de la grammaire abstraite XTL sera nécessaire pour la lecture des chapitres 3 et 4, ainsi que des annexes A et B.

Types résultat

$$
\begin{array}{rcl}
RT & ::= & T \\
 & | & (T_0, \ldots, T_n)
\end{array}
$$

Filtres

$$
\begin{array}{rcl}
P & ::= & x\text{:}T \\
 & | & (x_0\text{:}T_0, \ldots, x_n\text{:}T_n) \\
 & | & \textbf{any } T \\
 & | & P_1 \textbf{ of } RT \\
 & | & C\ (P_1, \ldots, P_n)
\end{array}
$$

Offres

$$
\begin{array}{rcl}
O & ::= & \textbf{any} \\
 & | & \textbf{!}\ E \\
 & | & \textbf{?}\ P_0 \mid \ldots \mid P_n
\end{array}
$$

Formules sur actions

$$
\begin{array}{lll}
\alpha & ::= & (G_0|O_0)\ O_1 \ldots O_m\ [\ldots]\ O_{m+1} \ldots O_{m+n}\ [\textbf{where}\ E] \\
& | & \textbf{true} \\
& | & \textbf{false} \\
& | & \textbf{not}\ \alpha_1 \\
& | & \alpha_1\ \textbf{or}\ \alpha_2 \\
& | & \alpha_1\ \textbf{and}\ \alpha_2 \\
& | & \alpha_1\ \textbf{implies}\ \alpha_2 \\
& | & \alpha_1\ \textbf{iff}\ \alpha_2 \\
& | & \alpha_1\ \textbf{xor}\ \alpha_2
\end{array}
$$

Expressions régulières

$$
\begin{array}{lll}
R & ::= & \alpha \\
& | & R_1\ .\ R_2 \\
& | & R_1\ |\ R_2 \\
& | & R_1^{*} \\
& | & R_1^{+}
\end{array}
$$

Formules sur états

$$
\begin{array}{lll}
\varphi & ::= & E \\
& | & Y\ (E_1, \ldots, E_n) \\
& | & \textbf{not}\ \varphi_1 \\
& | & \varphi_1\ \textbf{or}\ \varphi_2 \\
& | & \varphi_1\ \textbf{and}\ \varphi_2 \\
& | & \varphi_1\ \textbf{implies}\ \varphi_2 \\
& | & \varphi_1\ \textbf{iff}\ \varphi_2 \\
& | & \varphi_1\ \textbf{xor}\ \varphi_2 \\
& | & \langle R \rangle\ \varphi_1 \\
& | & [R]\ \varphi_1 \\
& | & @\ (R) \\
& | & \textbf{mu}\ Y\ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi_1 \\
& | & \textbf{nu}\ Y\ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi_1 \\
& | & \textbf{exists}\ x_0{:}T_0\ [\textbf{among}\ E_0], \ldots, x_n{:}T_n\ [\textbf{among}\ E_n]\ \textbf{in}\ \varphi_1 \\
& | & \textbf{forall}\ x_0{:}T_0\ [\textbf{among}\ E_0], \ldots, x_n{:}T_n\ [\textbf{among}\ E_n]\ \textbf{in}\ \varphi_1 \\
& | & \textbf{let}\ x_0{:}T_0{:=}E_0, \ldots, x_n{:}T_n{:=}E_n\ \textbf{in} \\
& & \quad \varphi_1 \\
& & \textbf{endlet} \\
& | & \textbf{let}\ (x_0^0{:}T_0^0, \ldots, x_0^{n_0}{:}T_0^{n_0}){:=}E_0, \ldots, (x_m^0{:}T_m^0, \ldots, x_m^{n_m}{:}T_m^{n_m}){:=}E_m\ \textbf{in} \\
& & \quad \varphi_1 \\
& & \textbf{endlet}
\end{array}
$$

$$
\begin{array}{lll}
 & | & \textbf{if } E_0 \textbf{ then } \varphi_0 \\
 & & \quad \textbf{elsif } E_1 \textbf{ then } \varphi_1 \\
 & & \quad \ldots \\
 & & \quad \textbf{elsif } E_n \textbf{ then } \varphi_n \\
 & & \ [\textbf{else } \varphi_{n+1}] \\
 & & \textbf{endif} \\
 & | & \textbf{case } E_0 \textbf{ in} \\
 & & \qquad P_1^0 \ | \ldots | \ P_1^{n_1} \ \ [\textbf{where } E_1] \ \ \texttt{->} \ \varphi_1 \\
 & & \qquad \ldots \\
 & & \quad | \ P_m^0 \ | \ldots | \ P_m^{n_m} \ [\textbf{where } E_m] \ \texttt{->} \ \varphi_m \\
 & & \ [| \ \textbf{otherwise} \ \texttt{->} \ \varphi_{m+1}] \\
 & & \textbf{endcase} \\
 & | & \textbf{case action } E_0 \textbf{ in} \\
 & & \qquad \alpha_1 \ \ [\textbf{where } E_1] \ \ \texttt{->} \ \varphi_1 \\
 & & \qquad \ldots \\
 & & \quad | \ \alpha_m \ [\textbf{where } E_m] \ \texttt{->} \ \varphi_m \\
 & & \ [| \ \textbf{otherwise} \ \texttt{->} \ \varphi_{m+1}] \\
 & & \textbf{endcase}
\end{array}
$$

Expressions

$$
\begin{array}{lll}
E & ::= & K \\
 & | & \textbf{true} \\
 & | & \textbf{false} \\
 & | & \textbf{not } E_1 \\
 & | & E_1 \textbf{ or } E_2 \\
 & | & E_1 \textbf{ and } E_2 \\
 & | & E_1 \textbf{ implies } E_2 \\
 & | & E_1 \textbf{ iff } E_2 \\
 & | & E_1 \textbf{ xor } E_2 \\
 & | & \textbf{nop} \\
 & | & E_1 \textbf{ ; } E_2 \\
 & | & \textbf{print } \textbf{(}E_1\textbf{)} \\
 & | & \textbf{(}E_0, \ldots, E_n\textbf{)} \\
 & | & \boldsymbol{\{}E_1, \ldots, E_n\boldsymbol{\}} \\
 & | & \boldsymbol{\{}E_1 \ \textbf{...} \ E_2\boldsymbol{\}} \\
 & | & E_1 \ F \ E_2 \\
 & | & F \ \textbf{(}E_1, \ldots, E_n\textbf{)} \\
 & | & E_1 \textbf{ of } RT \\
 & | & x \\
 & | & E_1 \ \textbf{.} \ x
\end{array}
$$

| **current**
| **let** x_0**:**T_0**:=**$E_0, \ldots, x_n$**:**T_n**:=**E_n **in**
  E'
endlet
| **let** **(**x_0^0**:**$T_0^0, \ldots, x_0^{n_0}$**:**$T_0^{n_0}$**)**:=$E_0, \ldots,$**(**x_m^0**:**$T_m^0, \ldots, x_m^{n_m}$**:**$T_m^{n_m}$**)**:=E_m **in**
  E'
endlet
| **if** E_0 **then** E'_0
  elsif E_1 **then** E'_1
  ...
  elsif E_n **then** E'_n
  [**else** E'_{n+1}]
endif
| **assert** $E_0, \ldots, E_n$ **in**
  E'
endassert
| **case** E_0 **in**
  P_1^0 |...| $P_1^{n_1}$ [**where** E_1] **->** E'_1
  ...
  | P_m^0 |...| $P_m^{n_m}$ [**where** E_m] **->** E'_m
  [| **otherwise** **->** E'_{m+1}]
endcase
| **case action** E_0 **in**
  α_1 [**where** E_1] **->** E'_1
  ...
  | α_m [**where** E_m] **->** E'_m
  [| **otherwise** **->** E'_{m+1}]
endcase
| **loop** **(**x_0**:**T_0**:=**$E_0, \ldots, x_n$**:**T_n**:=**E_n**)** **:** RT **in**
  E'
endloop
| **continue** **(**$E_0, \ldots, E_n$**)**
| **for** x'_0**:**T'_0[**among** E'_0]$, \ldots, x'_m$**:**T'_m[**among** E'_m]
  [**var** x_0**:**T_0**:=**$E_0, \ldots, x_n$**:**T_n**:=**E_n]
  [**where** E''_1]
  [**while** E''_2]
  in E''_3
  [**result** E''_4]
endfor

$$
\begin{array}{ll}
| & \{\ F\ \textbf{on}\ x_0{:}T_0\ [\textbf{among}\ E_0],\ldots,x_n{:}T_n\ [\textbf{among}\ E_n]\ [\textbf{where}\ E_1']\ \}\ E_2' \\
| & \{\ x{:}T\ [\textbf{among}\ E_1]\ \textbf{where}\ E_2\ \} \\
| & \textbf{exists}\ x_0{:}T_0\ [\textbf{among}\ E_0],\ldots,x_n{:}T_n\ [\textbf{among}\ E_n]\ \textbf{in}\ E' \\
| & \textbf{forall}\ x_0{:}T_0\ [\textbf{among}\ E_0],\ldots,x_n{:}T_n\ [\textbf{among}\ E_n]\ \textbf{in}\ E' \\
| & [E_1] \models \varphi \\
| & [E_1] \models \textbf{action}\ \alpha \\
| & [[\varphi]] \\
| & [[\textbf{action}\ \alpha]]
\end{array}
$$

Définitions de fonctions

$$
\begin{array}{rcl}
D & ::= & [\textbf{local}]\ \textbf{function}\ F\ (x_1{:}T_1,\ldots,x_n{:}T_n)\ \textbf{:}\ RT\ \textbf{is} \\
 & & \quad E \\
 & & \textbf{endfunc} \\
 & | & [\textbf{local}]\ \textbf{function}\ \_\ F\ \_\ (x_1{:}T_1,x_2{:}T_2)\ \textbf{:}\ RT\ \textbf{is} \\
 & & \quad E \\
 & & \textbf{endfunc}
\end{array}
$$

Définitions de formules

$$
\begin{array}{rcl}
M & ::= & \textbf{formula}\ AF\ (PAR_1,\ldots,PAR_n)\ \textbf{is} \\
 & & \quad \alpha \\
 & & \textbf{endform} \\
 & | & \textbf{formula}\ SF\ (PAR_1,\ldots,PAR_n)\ \textbf{is} \\
 & & \quad \varphi \\
 & & \textbf{endform}
\end{array}
$$

Inclusions de bibliothèques

$$
\begin{array}{rcl}
L & ::= & \textbf{library} \\
 & & \quad FN_0,\ldots,FN_n \\
 & & \textbf{endlib}
\end{array}
$$

Programme

$$
\begin{array}{rcl}
PG & ::= & [L_0 \ldots L_m]\ [M_0 \ldots M_n]\ [D_0 \ldots D_p] \\
 & & \quad E \\
 & & [\textbf{where}\ [L_{m+1} \ldots L_{m+q}]\ [M_{n+1} \ldots M_{n+r}]\ D_{p+1} \ldots D_{p+s}]
\end{array}
$$

2.2 Eléments lexicaux

Les unités lexicales du langage XTL sont groupées en trois classes : *mots-clés*, *identificateurs* et *séparateurs*. La liste des mots-clés est donnée dans la table 2.3. Outre ces mots-clés "alphabétiques", les symboles spéciaux suivants font également partie du langage : ( ) { } < > [ ] . , : ; := | -> ... ! ? * + @ |= [[ ]].

action	**elsif**	**endloop**	**implies**	**not**	**true**
among	**endassert**	**exists**	**in**	**nu**	**var**
and	**endcase**	**false**	**is**	**of**	**where**
any	**endfor**	**for**	**let**	**on**	**while**
assert	**endform**	**forall**	**library**	**or**	**xor**
case	**endfunc**	**formula**	**local**	**otherwise**	
continue	**endif**	**function**	**loop**	**print**	
current	**endlet**	**if**	**mu**	**result**	
else	**endlib**	**iff**	**nop**	**then**	

Table 2.3: Les mots-clés du langage XTL

Les identificateurs sont partagés en deux classes :

Identificateurs internes, correspondant aux objets définis dans le programme XTL ou prédéfinis dans le langage. Ces identificateurs peuvent être de deux sortes :

Identificateurs normaux, composés de lettres, chiffres et/ou du caractère '_' (souligné) et commençant obligatoirement par une lettre ou un '_'. Dans ces identificateurs, aucune différence n'est faite entre lettres majuscules et minuscules.

Exemple 2-1
Les identificateurs normaux suivants sont valides : SEND, Packet, msg_2, _data. ■

Identificateurs spéciaux, construits à partir des caractères '+', '-', '*', '/', '%', '&', '<', '=', '>', '@', '\', '^', '~', '#', '{' et '}'. Ces identificateurs permettent une notation plus intuitive pour les différents opérateurs mathématiques.

Exemple 2-2
Les identificateurs spéciaux suivants dénotent des opérateurs booléens, arithmétiques et relationnels couramment utilisés : /\ \/ ~ ==> + - * / <= >= . ■

Identificateurs externes, correspondant aux objets définis dans le programme parallèle à vérifier. Ces identificateurs sont stockés dans la zone des noms (*name area*) du fichier BCG [Gar94]. Si leur syntaxe n'entre pas en conflit avec celle des identificateurs internes ou des mots-clés, les identificateurs externes peuvent être écrits tels quels dans le programme XTL ; dans le cas contraire, ils doivent être placés entre deux caractères '`' (accent grave).

Exemple 2-3
Les identificateurs externes suivants sont valides : NatList, concat, `result`, `2nd_message`. Il faut écrire `result` puisque **result** est un mot-clé de XTL et `2nd_message` puisque 2nd_message n'est pas un identificateur XTL valide. ■

Les séparateurs sont des unités lexicales n'ayant pas de signification pour le compilateur ; ils sont utilisés dans les programmes XTL uniquement pour délimiter les occurrences des mots-clés et des identificateurs. Les séparateurs sont des séquences non vides composées de caractères espace, tabulation et/ou fin de ligne, ainsi que de *commentaires*, écrits entre (* et *).

2.3 Types

XTL est un langage fortement typé : l'évaluation de chaque expression XTL produit une valeur dont le type est déterminable statiquement. Il existe trois classes de types pouvant être utilisés dans un programme XTL : les types XTL, les types BCG et les types tuples.

2.3.1 Types XTL

Les types XTL, accompagnés des opérations correspondantes, sont prédéfinis dans le langage. Ces types et opérations sont considérés comme objets internes au langage XTL et, par conséquent, sont désignés au moyen d'identificateurs internes (voir la section 2.2). Les types XTL peuvent être groupés en plusieurs catégories, décrites dans les paragraphes suivants.

Types de base XTL fournit les types de base couramment rencontrés dans les langages de programmation : `boolean`, `integer`, `real`, `character` et `string`, qui représentent respectivement les valeurs booléennes, les nombres entiers, les nombres réels, les caractères et les chaînes de caractères. Deux autres types sont également prédéfinis : `intset` et `charset`, dénotant respectivement des ensembles d'entiers et de caractères. Ces types de base sont munis des opérations usuelles données dans la table 2.4 (les opérations associées aux types `intset` et `charset` seront définies plus loin, au paragraphe concernant les types ensembles).

OPÉRATEUR	SIGNIFICATION

`false, true : -> boolean`
`not : boolean -> boolean`
`or, and, implies, iff, xor :`
`    boolean, boolean -> boolean`
	opérateurs booléens

`+, -, *, /, mod : integer, integer -> integer`
`+, -, *, / : real, real -> real`
`- : integer -> integer`
`- : real -> real`
	opérateurs arithmétiques

`<, >, <=, >=, =, <> : integer, integer -> boolean`
`<, >, <=, >=, =, <> : real, real -> boolean`
`<, >, <=, >=, =, <> : character, character -> boolean`
`<, >, <=, >=, =, <> : string, string -> boolean`
	opérateurs relationnels

`min, max : integer, integer -> integer`
`min, max : real, real -> real`
`min, max : character, character -> character`
`min, max : string, string -> string`
	minimum et maximum

`+ : string, string -> string`	concaténation
`char : string, integer -> character`	sélection
`length : string -> integer`	longueur

`integer : character -> integer`
`character : integer -> character`
`string : character -> string`
	opérateurs de conversion

Table 2.4: Opérations prédéfinies sur les types de base

Remarque 2-1
Dans la suite du document, nous adoptons la convention de notation suivante : les noms des types et des opérateurs prédéfinis seront écrits en caractères de dactylographie (par exemple, `integer`), sauf ceux qui sont des mots-clés du langage, qui seront écrits en caractères gras (par exemple, **true**). ■

Les constantes littérales (symbole terminal K) dénotant des valeurs de ces types sont écrites selon la syntaxe habituelle : les nombres entiers sont notés en décimal (par exemple, `1997`) ; les nombres réels sont notés en virgule flottante (par exemple, `2.71`) ; les nombres négatifs s'obtiennent au moyen de l'opérateur "`-`" unaire (par exemple, `-(1)`, `-(3.14)`) ; les caractères sont notés entre apostrophes (par exemple, `'a'`), les caractères spéciaux ainsi que le caractère apostrophe étant précédés par un caractère '\' (par exemple, `'\n'` – fin de ligne, `'\t'` – tabulation, `'\''` – apostrophe) ; les chaînes de caractères sont notées entre guillemets, les caractères spéciaux étant précédés par un '\' (par exemple, `"Resultats :\n"`).

Afin de faciliter la compilation, nous avons choisi une syntaxe des constantes littérales proche de celle du langage C, qui est le langage cible du compilateur XTL. Ce choix se retrouve partiellement dans l'implémentation des types prédéfinis : par exemple, les caractères composant les chaînes `str` sont numérotés de `0` à `length(str) - 1`, comme dans le langage C.

Remarque 2-2
Les opérateurs booléens "**true**" et "**false**" sont les constructeurs du type `boolean` : toute expression de ce type est évaluée vers une forme normale unique égale à "**true**" ou à "**false**". ■

Type "void" Le langage XTL permet d'afficher des informations sur un fichier de sortie (pour une implémentation du compilateur XTL sous le système d'exploitation UNIX, il s'agit du fichier standard de sortie `stdout`). Le type `void` est associé aux fonctions XTL permettant de modifier par effet de bord le fichier de sortie. Il existe trois opérateurs prédéfinis de type `void`, qui sont aussi des mots-clés du langage :

- **nop** `: -> void`, qui dénote une action sans aucun effet. Cet opérateur est le constructeur unique du type `void` ; il est utile dans certaines situations, notamment dans les expressions d'itération "**for**" (voir la section 2.6.2) ;

- **;** `: void, void -> void`, qui dénote la composition séquentielle de ses deux opérandes. Cet opérateur est infixé (ses appels peuvent être écrits E_1 **;** E_2), ce qui permet d'avoir une syntaxe du séquencement proche des instructions de composition séquentielle rencontrées dans les langages de programmation comme Pascal ou C ;

- **print** `: T -> void`, qui imprime la valeur de son argument sur le fichier de sortie. Cet opérateur est surchargé : le type `T` de son argument peut être instancié vers tout type XTL prédéfini et, plus généralement, vers tout type muni d'un opérateur d'impression (par exemple, les types BCG décrits à la section 2.3.2).

Le langage XTL est déterministe : chaque exécution d'un programme PG doit produire les mêmes résultats sur le fichier de sortie. Ceci revient à dire que toutes les expressions de type `void` contenues dans PG (qui sont les seules à pouvoir modifier le fichier de sortie par effet de bord) sont évaluées suivant une séquence unique. La sémantique dynamique des expressions XTL (voir l'annexe B) est définie de manière à assurer le déterminisme : pour chaque expression E, l'ordre d'évaluation de ses sous-expressions E_i est fixé et connu statiquement. En particulier, les arguments des appels de fonctions sont évalués de gauche à droite (dans le cas de l'opérateur de composition séquentielle "**;**", on obtient ainsi l'effet escompté).

Remarque 2-3
Dans le but d'optimiser l'évaluation des expressions, l'implémentation du compilateur XTL ne garantit un ordre d'évaluation fixé que pour les expressions de type `void` (par exemple, l'ordre d'évaluation des arguments de type différent de `void` d'un appel de fonction n'est pas précisé). Ceci n'affecte pas la sémantique dynamique du langage qui, au niveau utilisateur, reste déterministe. ■

Remarque 2-4
La sémantique des formules XTL (voir le chapitre 3) ne spécifie pas l'ordre d'évaluation des expressions contenues dans les formules (bien que les algorithmes d'évaluation des formules soient déterministes). En particulier, la séquence d'évaluation des expressions de type `void` contenues dans une formule α ou φ ne peut pas être connue statiquement.

Néanmoins, l'opérateur "**print**" peut être utilisé dans une formule φ afin d'imprimer sur le fichier de sortie des informations sur le déroulement de l'évaluation de φ. Ceci permet d'effectuer un "traçage" des formules, qui peut s'avérer utile pour la mise au point et pour l'estimation de la complexité des algorithmes d'évaluation des formules. ■

L'impression d'une séquence de plusieurs valeurs sur le fichier de sortie peut être exprimée de manière plus concise en utilisant la construction abrégée suivante :

$$\boxed{\textbf{print}\ (E_0, \ldots, E_n) \ \stackrel{\mathrm{d}}{=}\ \textbf{print}\ (E_0)\ \textbf{;}\ \cdots\ \textbf{; print}\ (E_n)}$$

L'évaluation de gauche à droite des arguments assure que les valeurs des expressions $E_0, \ldots, E_n$ seront imprimées dans cet ordre sur le fichier de sortie.

Méta-types Ce sont les types spéciaux `state`, `label` et `trans`, qui représentent respectivement les états, les étiquettes et les transitions du modèle STE correspondant au programme parallèle à vérifier[7]. Les types `stateset`, `labelset` et `transset`, dénotant respectivement des ensembles d'états, d'étiquettes et de transitions du STE, sont également prédéfinis.

OPÉRATEUR	SIGNIFICATION
`init : -> state`	état initial du STE
`pred : state -> stateset` `succ : state -> stateset`	états prédécesseurs et successeurs d'un état
`=, <> : state, state -> boolean`	comparaison des états
`pred : trans -> transset` `succ : trans -> transset`	transitions prédécesseurs et successeurs d'une transition
`=, <> : trans, trans -> boolean`	comparaison des transitions
`in  : state -> transset` `out : state -> transset`	transitions arrivant et partant d'un état
`source : trans -> state` `target : trans -> state`	état de départ et d'arrivée d'une transition
`label  : trans -> label`	étiquette d'une transition
`visible : label -> boolean`	test d'étiquette visible
`=, <> : label, label -> boolean`	comparaison des étiquettes

Table 2.5: Opérations prédéfinies sur les méta-types

[7]Nous utilisons le terme "méta-types" afin de préciser que ces types font référence au modèle STE du programme à vérifier plutôt qu'aux données manipulées par ce programme.

Dans la suite du document, tous les opérateurs ayant arguments et/ou résultat d'un méta-type seront appelés, par extension, *méta-opérateurs*. Les méta-opérateurs prédéfinis usuels sont donnés dans la table 2.5. D'autres méta-opérateurs prédéfinis, comme les opérateurs "**current**" sur actions et sur états, seront présentés aux sections 2.8.3 et 2.10.2.

Les méta-opérateurs donnés dans la table 2.5, dont certains sont inspirés du calcul de Dicky [Dic86], permettent d'explorer la relation de transition du modèle STE : `init` donne l'accès à l'état initial du STE ; `pred` et `succ` (qui sont surchargés sur états et transitions) permettent d'accéder aux états (*resp.* transitions) prédécesseurs et successeurs d'un état (*resp.* transition) du STE ; `in` et `out` font la liaison entre états et transitions, donnant accès aux transitions rentrant et sortant d'un état ; `source` et `target` font la liaison entre transitions et états, donnant accès aux états de départ et d'arrivée d'une transition. La figure 2.2 donne une représentation graphique de ces divers opérateurs.

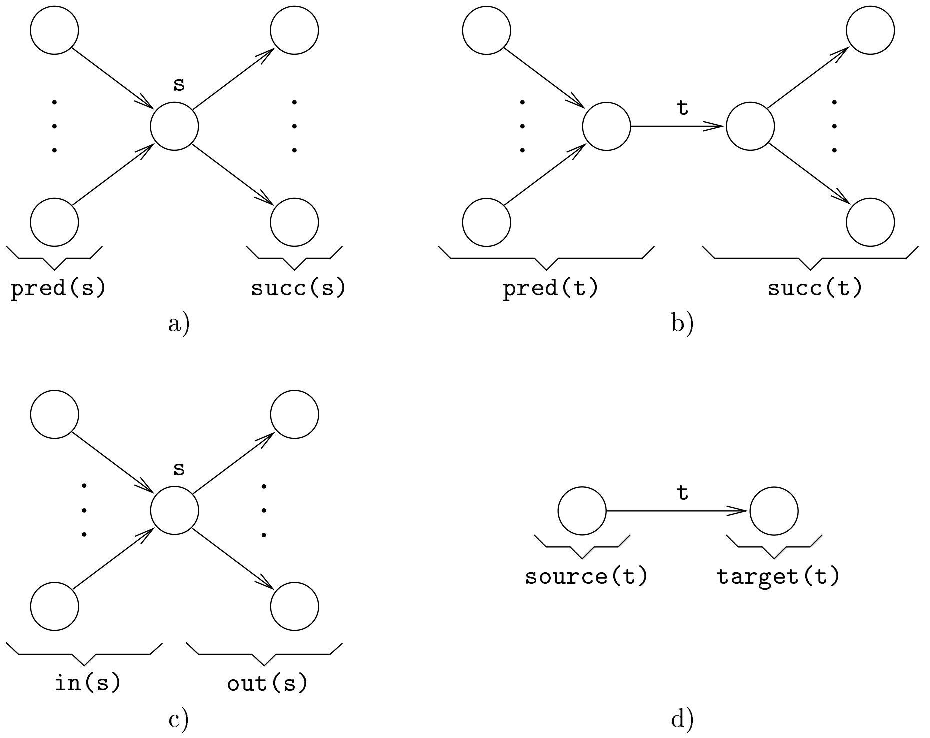

Figure 2.2: a) états successeurs et prédécesseurs d'un état ; b) transitions successeurs et prédécesseurs d'une transition ; c) transitions arrivant à et partant d'un état ; d) états origine et but d'une transition

Les méta-opérateurs permettent de calculer diverses informations sur le modèle STE, ainsi que de décrire des opérateurs temporels par exploration de la relation de transition. Différents exemples d'utilisation des méta-opérateurs seront présentés aux sections 2.6.2, 2.6.3, 2.6.4, 2.6.5 et 2.7.

Types ensembles XTL fournit les types ensembles prédéfinis `intset`, `charset`, `stateset`, `labelset` et `transset`, mentionnés aux paragraphes précédents. Ces types sont munis des opérations ensemblistes usuelles, données dans la table 2.6. Toutes ces opérations sont surchargées : les types génériques `elem` et `set` utilisés dans la table 2.6 peuvent être instanciés deux à deux par `integer` et `intset`, `character` et `charset`, `state` et `stateset`, `label` et `labelset`, `trans` et `transset`. En outre, pour faciliter l'écriture des opérateurs temporels, les fonctions ensemblistes `empty`, `full`, `union`, `inter` et `comp` ont aussi des synonymes booléens (`false`, `true`, `or`, `and` et `not`).

Opérateur	Signification
`empty, false : -> set`	ensemble vide
`full, true : -> set`	ensemble total
`union, or : set, set -> set`	union
`inter, and : set, set -> set`	intersection
`diff : set, set -> set`	différence
`comp, not : set -> set`	complément
`includes : set, set -> boolean`	test d'inclusion
`insert : set, elem -> set`	insertion d'un élément
`remove : set, elem -> set`	suppression d'un élément
`isin : elem, set -> boolean`	appartenance d'un élément
`card : set -> integer`	cardinal

Table 2.6: Opérations prédéfinies sur les types ensembles

Les valeurs de type ensemble peuvent être représentées en *extension*, c'est-à-dire en donnant la liste de leurs éléments, selon la notation traditionnelle :

$$\{E_1, \ldots, E_n\}$$

Cette notation aussi est surchargée, pouvant être appliquée pour tous les types ensembles prédéfinis. La notation "`{ }`" dénote un ensemble vide : elle est donc synonyme de l'opérateur `empty`.

En outre, des ensembles de nombres entiers ou de caractères peuvent également être définis comme *sous-domaines* des types `intset` et `charset`, utilisant la notation suivante :

$$\{E_1 \ \ldots \ E_2\}$$

Si v_1 et v_2 sont les valeurs respectives des expressions E_1 et E_2 (appelées *bornes* du sous-domaine), l'expression ci-dessus contient tous les éléments v du type respectif tels que $v_1 \leq v \leq v_2$; si $v_1 > v_2$, le sous-domaine est vide.

Exemple 2-4
Le sous-domaine `{0 ... 10}` dénote l'ensemble des nombres entiers entre 0 et 10. Le sous-domaine `{'a' ... 'z'}` dénote l'ensemble des lettres minuscules. ■

Les sous-domaines sont particulièrement utiles dans les constructions d'itération Xtl, qui seront présentées aux sections 2.6.2, 2.6.3, 2.6.4 et 2.6.5.

2.3.2 Types et fonctions BCG

Le format de fichiers Bcg fournit l'accès à diverses informations provenant du programme source à vérifier (en particulier, les types et les fonctions) qui peuvent être utilisées, suivant certaines conventions, dans les programmes Xtl.

Types BCG Ce sont les types définis dans le programme à vérifier. Ils sont contenus dans la zone des types (*type area*) du fichier Bcg et une implémentation en langage C est fournie pour chacun d'entre eux dans la zone des inclusions (*include area*) du fichier Bcg. Bien qu'ils soient considérés comme externes au langage Xtl, ces types peuvent être désignés par des identificateurs internes (voir la section 2.2), sauf si leurs noms entrent en conflit avec d'autres noms de types XTL ou avec la syntaxe des identificateurs XTL, auquel cas ils doivent être désignés au moyen d'identificateurs externes (entourés par des caractères ‘‘’). Cette convention permet d'employer dans les programmes XTL les notations du programme source à vérifier tant qu'elles n'entrent pas en conflit avec les noms des types XTL eux-mêmes.

Exemple 2-5
Un type BCG nommé `integer` doit être désigné dans les programmes XTL par `‘integer‘`, afin d'éviter le conflit avec le type XTL de même nom. De la même manière, un type BCG nommé `Type-Msg` doit être désigné par `‘Type-Msg‘`, car `Type-Msg` n'est pas un identificateur XTL valide. ■

Type “gate” Parmi les types BCG, il existe un type particulier, noté `gate`, associé aux portes (symbole terminal G) contenues dans les étiquettes du modèle STE (voir la section 1.1).

Le type `gate` est un type énuméré : pour chaque porte G contenue dans le modèle STE, il existe un opérateur constructeur nullaire (c'est-à-dire sans paramètre) de type `gate` ayant le même nom que G. Ceci permet de désigner les portes de façon naturelle dans les programmes XTL, en utilisant autant que possible les notations du programme parallèle à vérifier.

Exemple 2-6
Une porte `SEND` définie dans le programme à vérifier est désignée dans les programmes XTL par un appel du constructeur nullaire `SEND` de type `gate`. ■

Le type `gate` est muni des opérations de comparaison `=` et `<>`, ainsi que de l'opérateur d'impression **print**. L'utilisation du type `gate` est adaptée principalement pour la vérification de programmes LOTOS ; néanmoins, il peut servir aussi pour tout langage de description utilisant des canaux de communication nommés et traduisible vers un modèle STE en format BCG.

Fonctions BCG Les fonctions définies dans le programme à vérifier, appelées fonctions BCG, sont également fournies par le fichier BCG représentant le STE du programme. Leurs noms et profils sont contenus dans la zone des fonctions (*function area*) et leur implémentation (en langage C) réside dans la zone d'inclusion (*include area*) du fichier BCG. Ces fonctions sont désignées dans les programmes XTL suivant des conventions de nommage similaires à celles utilisées pour les types BCG.

Le format de fichiers BCG permet d'identifier les fonctions BCG qui sont des opérateurs constructeurs (c'est notamment le cas des fichiers BCG produits avec le compilateur CÆSAR à partir de programmes LOTOS). Si un type BCG T possède des opérateurs constructeurs, les valeurs de type T peuvent être représentées de façon unique sous *forme normale*, c'est-à-dire comme termes algébriques constitués d'appels de constructeurs. Les valeurs de type T peuvent être manipulées en XTL par des constructions de filtrage (*pattern-matching*) utilisant les opérateurs constructeurs de T (voir les sections 2.5.4, 2.5.5, 2.8.1 et 2.8.2).

2.3.3 Types tuples

Les types tuples (symbole non-terminal RT) dénotent des n-uplets construits à partir de types XTL et/ou de types BCG. Les types tuples sont notés à l'aide de parenthèses, qui jouent le rôle de “constructeurs de n-uplet”. Ces types n'ont pas de nom et leurs champs ne sont pas nommés.

Exemple 2-7
Un message peut être modélisé comme valeur d'un type tuple `(gate, string, integer)` contenant une porte (le canal de communication), une chaîne de caractères (le contenu du message) et un nombre entier (le code correcteur). ■

Un type tuple `(T)` à un seul champ est équivalent au type `T` de son champ. L'équivalence des types tuples est définie de façon structurelle : deux types tuples sont égaux ssi ils ont le même nombre de champs et les types de leurs champs respectifs sont égaux. Les valeurs de type tuple sont désignées au moyen d'expressions tuples, construites elles aussi à l'aide de parenthèses.

Exemple 2-8
L'expression `(SEND, "message", 11)` dénote un tuple ayant le type défini à l'exemple 2-7. ■

Les expressions de type tuple sont utiles dans plusieurs cas :

- Pour définir une fonction qui renvoie plusieurs résultats. En XTL, ceci s'obtient en spécifiant une fonction qui renvoie un résultat de type tuple ;
- Pour calculer simultanément plusieurs résultats. En XTL, ceci s'obtient au moyen des expressions d'itération "**for**" (voir la section 2.6.2) ayant un accumulateur de type tuple ;
- Pour effectuer le filtrage simultané de plusieurs expressions. En XTL, ceci s'obtient en filtrant une expression unique, de type tuple.

Toutefois, pour simplifier le langage et les algorithmes de compilation sous-jacents, l'utilisation des types tuples en XTL est limitée par les contraintes suivantes :

- Il est interdit d'imbriquer les types tuples, c'est-à-dire que les champs d'un type tuple peuvent être de type XTL ou BCG, mais pas de type tuple. Nous avons imposé cette restriction car, en présence des fonctions surchargées (*overloading*), le typage des expressions tuples imbriquées (en particulier, la résolution des surcharges de fonctions) nécessite des algorithmes de compilation dont la complexité peut s'avérer prohibitive.
- Il est interdit de mémoriser des expressions tuples ; l'utilisateur ne peut pas définir des variables XTL (voir la section 2.4) de type tuple, ni des fonctions XTL (voir la section 2.7) ayant des paramètres de type tuple[8]. En particulier, la comparaison et l'impression des valeurs de type tuple doivent être effectuées au moyen d'expressions "**let**" destructurantes (voir la section 2.5.3). En revanche, la définition de fonctions XTL renvoyant des résultats de type tuple est autorisée.

Ces restrictions sont vérifiées par l'analyse de la sémantique statique du langage (voir l'annexe A.7).

2.4 Variables

Il existe deux classes de variables pouvant être utilisées dans un programme XTL : les variables *simples* (symbole terminal x), dénotant des valeurs typées, et les variables *propositionnelles* (symbole terminal Y), dénotant des formules sur états (voir la section 2.10.6). Les variables simples, présentées dans les paragraphes suivants, sont de deux sortes : les variables BCG et les variables XTL. Dans la suite du document (sauf en cas d'ambiguïté) nous utiliserons le terme "variables" pour désigner les variables simples.

Les variables BCG sont définies dans le programme parallèle à vérifier. Ces variables se retrouvent dans le fichier BCG contenant le modèle STE du programme sous forme de champs du vecteur d'état, ces champs étant nommés (voir la section 1.1.3). Au niveau du programme XTL, ces variables sont désignées à l'aide de l'opérateur "**.**" : l'expression "$E.x$", où E doit dénoter une valeur s de type `state`, renvoie la valeur du champ de nom x dans le vecteur d'état correspondant à s.

[8] Cette restriction pourrait facilement être levée, si le besoin s'impose.

Exemple 2-9
L'expression suivante imprime la valeur de la variable `x` contenue dans l'état initial du STE :

```
print ("valeur de x dans l'etat initial : ", init.x)
```

La valeur de `x` est imprimée au moyen de l'opérateur "**print**" prédéfini associé au type de `x`. ■

Ceci permet de caractériser des états du modèle STE à l'aide de prédicats portant sur les variables BCG contenues dans ces états (voir les sections 2.10.1 et 2.10.2). Bien entendu, dans un programme XTL les variables BCG ne peuvent pas être modifiées, mais uniquement consultées.

Les variables XTL sont définies dans le programme XTL : ce sont des noms permettant de désigner des valeurs typées. XTL est un langage applicatif : on ne dispose pas de la notion de *mémoire* (ensemble de cellules identifiables par des noms et pouvant stocker des valeurs) ni de celle d'*affectation* (changement du contenu des cellules de la mémoire). Les variables XTL peuvent être déclarées et initialisées, mais ne peuvent plus être modifiées ensuite (variables *write-once*).

La surcharge des variables est généralement interdite en XTL : plusieurs variables x de même nom (même ayant des types différents) ne peuvent pas être simultanément visibles. Cependant, il existe deux constructions XTL qui font exception à cette règle : les expressions "**case**" (section 2.5.5) et les formules sur actions (section 2.8), qui permettent que plusieurs occurrences de définition d'une variable x de type T aient la même portée. Toutefois, il est garanti qu'à l'exécution du programme, une seule de ces occurrences sera utilisée.

Il est possible de définir des variables XTL ayant un type XTL (voir la section 2.3.1) ou un type BCG (voir la section 2.3.2) ; en revanche, il est interdit de définir des variables de type tuple (voir la section 2.3.3).

Il existe plusieurs constructions XTL permettant de définir et d'initialiser des variables : l'opérateur "**let**" (section 2.5.3), les filtres (section 2.5.4) utilisés dans les expressions "**case**" (section 2.5.5) et les filtres d'actions (section 2.8.2).

2.5 Expressions conditionnelles et de filtrage

XTL permet de décrire des traitements conditionnels des données, au moyen de constructions "**if**", "**assert**", "**let**" et "**case**" similaires à celles rencontrées dans les langages de programmation. Les sections suivantes présentent en détail chacune de ces constructions.

2.5.1 Expression "if"

La construction "**if**", qui permet d'effectuer des calculs conditionnels, est présente dans la quasi-totalité des langages de programmation. En XTL elle a la syntaxe suivante :

$$
\begin{array}{l}
\textbf{if } E_0 \textbf{ then } E'_0 \\
\quad \textbf{elsif } E_1 \textbf{ then } E'_1 \\
\quad \ldots \\
\quad \textbf{elsif } E_n \textbf{ then } E'_n \\
\;\big[\textbf{else } E'_{n+1}\big] \\
\textbf{endif}
\end{array}
$$

où les expressions $E'_0, \ldots, E'_{n+1}$ doivent avoir le même type et les expressions $E_0, \ldots, E_n$, appelées *conditions*, sont de type `boolean`.

L'évaluation d'une expression "**if**" est effectuée de la manière suivante. Les conditions E_i (pour i allant de 0 à n) sont évaluées successivement jusqu'à ce que l'on trouve une expression E_{i0} qui soit vraie. Si c'est le cas, la valeur de l'expression "**if**" est égale à celle de l'expression E'_i correspondant à la première condition E_{i0} qui est vraie. Si toutes les conditions $E_0, \ldots, E_n$ sont fausses, la valeur de l'expression "**if**" est égale à celle de E'_{n+1} ; si celle-ci est absente, l'exécution du programme XTL est arrêtée et une erreur est signalée.

Exemple 2-10
L'expression "**if**" suivante calcule le maximum de deux nombres entiers `x` et `y` :

```
if x >= y then x else y endif
```

■

Les expressions "**if**" ne sont pas des constructions XTL primitives : elles peuvent être traduites en termes d'expressions "**case**" (voir l'annexe B.1.2).

2.5.2 Expression "assert"

La construction conditionnelle "**assert**" permet de vérifier, lors de l'exécution, que certaines conditions sont respectées à des points précis du programme XTL. Cette expression a la syntaxe suivante :

assert $E_0, \ldots, E_n$ **in**
  E
endassert

où E est appelée *corps* de l'expression "**assert**" et les expressions $E_0, \ldots, E_n$, appelées *conditions*, doivent être de type `boolean`.

Si les conditions $E_0, \ldots, E_n$ sont toutes évaluées à vrai, la valeur d'une expression "**assert**" est égale à celle de son corps E ; dans le cas contraire, l'exécution du programme XTL est arrêtée et une erreur est signalée.

Exemple 2-11
L'expression suivante renvoie le premier élément d'une liste `l` de nombres entiers, à condition que celle-ci soit non vide :

```
assert l <> nil in
    head (l)
endassert
```

Si la liste `l` est vide, un message d'erreur est issu et le programme est terminé. ■

Remarque 2-5
L'expression "**assert**" pourrait être vue comme un cas particulier d'expression "**if**" :

		if E_0 **then**
		...
assert $E_0, \ldots, E_n$ **in**		**if** E_n **then**
E	$\stackrel{\mathrm{d}}{=}$	E
endassert		**endif**
		...
		endif

Toutefois, dans la sémantique dénotationnelle (voir l'annexe B.1.2) nous avons préféré distinguer ces deux constructions, notamment puisqu'elles produisent des messages d'erreur différents. ■

Les expressions "**assert**" sont utiles pour détecter les situations d'exception, mais ne permettent pas de reprendre l'exécution du programme après qu'une erreur ait été signalée. Pour bénéficier d'une gestion plus souple des cas d'exception, un mécanisme explicite de traitement d'exceptions doit être rajouté dans une version future du langage XTL.

2.5.3 Expressions "let"

Les expressions "**let**" constituent le plus simple moyen de déclarer et d'initialiser des variables XTL. Elles ont la syntaxe suivante :

$$\begin{array}{l}\textbf{let } x_0\textbf{:}T_0\textbf{:=}E_0\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n \textbf{ in}\\ \quad E\\ \textbf{endlet}\end{array}$$

où pour chaque $0 \leq i \leq n$, la variable x_i de type T_i est initialisée avec la valeur de l'expression E_i. Toutes les variables x_i sont visibles dans l'expression E (appelée *corps* du "**let**"), mais aucune n'est visible dans les expressions E_i.

Exemple 2-12
L'expression suivante imprime sur le fichier de sortie l'état initial du STE ainsi que ses états successeurs :

```
let s : state := init in
    print ("Etat initial : ", s, "\nEtats successeurs : ", succ (s))
endlet
```

L'impression de l'état `s` et de l'ensemble `succ (s)` de ses états successeurs est effectuée à l'aide des fonctions "**print**" prédéfinies associées aux types `state` et `stateset`. ■

Les expressions "**let**" peuvent être imbriquées, permettant de créer une structure de blocs similaire à celle rencontrée dans les langages algorithmiques (comme ADA) ou fonctionnels (comme ML). La visibilité des variables suit les règles habituelles : une variable x définie dans une expression "**let**" "masque" les autres variables x éventuellement définies dans des expressions "**let**" englobantes.

Le langage XTL offre aussi des constructions spéciales, appelées "**let** destructurant", permettant de récupérer les champs des tuples :

$$\begin{array}{l}\textbf{let } \textbf{(}x_0^0\textbf{:}T_0^0\textbf{,}\ldots\textbf{,}x_0^{n_0}\textbf{:}T_0^{n_0}\textbf{):=}E_0\textbf{,}\ldots\textbf{,(}x_m^0\textbf{:}T_m^0\textbf{,}\ldots\textbf{,}x_m^{n_m}\textbf{:}T_m^{n_m}\textbf{):=}E_m \textbf{ in}\\ \quad E\\ \textbf{endlet}\end{array}$$

Dans l'expression ci-dessus, pour chaque $0 \leq i \leq m$, l'expression E_i doit être de type $\textbf{(}T_i^0\textbf{,}\ldots\textbf{,}T_i^{n_i}\textbf{)}$. Les règles de visibilité des variables x_i^j $(0 \leq i \leq m, 0 \leq j \leq n_i)$ sont les mêmes que pour l'expression "**let**" normale.

Exemple 2-13
L'expression suivante destructure une valeur de type tuple et imprime les valeurs de ses composantes sur le fichier de sortie :

```
let (msg : string, val : integer) := ("SEND", 1) in
    print ("Message : ", msg, "Contenu : ", val)
endlet
```

■

Les expressions "**let**" ne sont pas des constructions XTL primitives : elles peuvent être traduites en termes d'expressions "**case**" (voir l'annexe B.1.2).

2.5.4 Filtres

Un filtre (*pattern*) est une construction permettant d'obtenir des informations sur la structure d'une valeur typée. En XTL, pour qu'une valeur v de type T puisse être filtrée, elle doit être représentée sous *forme normale*, c'est-à-dire comme un terme algébrique contenant uniquement des opérateurs constructeurs (voir la section 2.3). Les filtres utilisés en XTL (symbole non-terminal P) ont la syntaxe suivante :

$$
\begin{array}{rcl}
P & ::= & x\mathbf{:}T \\
 & | & (x_0\mathbf{:}T_0\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n) \\
 & | & \mathbf{any}\ T \\
 & | & P_1\ \mathbf{of}\ RT \\
 & | & C\ (P_1\mathbf{,}\ldots\mathbf{,}P_n)
\end{array}
$$

où C est un opérateur constructeur. Les variables x contenues dans un filtre P dénotent des occurrences de définition ayant la même portée dans le programme : il est donc interdit d'avoir la même variable x définie plusieurs fois dans P.

L'*application* d'un filtre P sur une valeur v détermine si le terme algébrique de v a une structure compatible avec celle spécifiée par P (on dit alors que P *filtre* v). Si tel est le cas, les variables x définies dans P (s'il en existe) sont initialisées avec les valeurs correspondantes extraites du terme algébrique de v.

Les filtres sont utilisés dans les expressions "**case**" (section 2.5.5) et dans les offres faisant partie des formules sur actions (section 2.8.1). Chaque fois qu'un filtre P est appliqué sur une valeur v, les variables définies dans P ne seront utilisées dans le programme que si v est correctement filtrée par P. Ceci assure le fait que les variables définies dans un filtre seront toujours initialisées avant d'être utilisées.

Les différents filtres XTL, appliqués sur une valeur v de type T', produisent l'effet suivant :

- "$x\mathbf{:}T$" filtre v (et initialise la variable x à v) ssi $T' = T$;

- "$(x_0\mathbf{:}T_0\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n)$" filtre v (et initialise les variables $x_0, \ldots, x_n$ avec les champs correspondants de v) ssi $T' = (T_0, \ldots, T_n)$;

- "**any** T" filtre v ssi $T' = T$;

- "P **of** RT" a le même effet que P ; il sert uniquement à résoudre les ambiguïtés provoquées par la surcharge des opérateurs constructeurs éventuellement contenus dans P ;

- "$C\ (P_1\mathbf{,}\ldots\mathbf{,}P_n)$" filtre v (et initialise toutes les variables définies dans $P_1, \ldots, P_n$ avec des valeurs extraites de v) ssi v a la forme $C(v_1, \ldots, v_n)$ et pour tout $1 \leq i \leq n$, P_i filtre v_i.

Exemple 2-14

En supposant l'existence du type externe "liste de nombres entiers" `intlist`, muni des opérateurs constructeurs `nil` et `cons`, le filtre suivant :

```
cons (n : integer, l : intlist)
```

permet de filtrer toutes les listes non vides. Dans ce cas, la variable `n` reçoit la valeur du premier élément de la liste et la variable `l` est initialisée avec le reste de la liste. ■

2.5.5 Expression "case" sur valeurs

La construction conditionnelle la plus générale offerte par le langage XTL est l'expression "**case**". Elle a la syntaxe suivante :

$$
\begin{array}{l}
\textbf{case } E_0 \textbf{ in} \\
\quad P_1^0 \mid \ldots \mid P_1^{n_1} \; [\textbf{where } E_1] \; \text{->} \; E'_1 \\
\quad \ldots \\
\mid P_m^0 \mid \ldots \mid P_m^{n_m} \; [\textbf{where } E_m] \; \text{->} \; E'_m \\
[\mid \textbf{otherwise} \; \text{->} \; E'_{m+1}] \\
\textbf{endcase}
\end{array}
$$

où les expressions $E'_1, \ldots, E'_{m+1}$ doivent avoir le même type et les expressions optionnelles $E_1, \ldots, E_m$ sont de type `boolean`. Pour tout $1 \leq i \leq m$ et $0 \leq j, k \leq n_i$, les variables définies dans les filtres P_i^j et P_i^k doivent être identiques. Les variables définies dans un filtre P_i^j ne sont visibles que dans les expressions E_i et E'_i. Les constructions "$P_i^0 \mid \ldots \mid P_i^{n_i}$ [**where** E_i] -> E'_i" sont appelées *branches* de l'expression "**case**".

L'évaluation d'une expression "**case**" est effectuée de la manière suivante. L'expression E_0 est d'abord évaluée, rendant comme résultat une valeur v_0. Ensuite, les branches i (pour i allant de 1 à m) sont considérées successivement afin de trouver la première branche i_0 pour laquelle il existe un $0 \leq j_0 \leq n_{i_0}$ tel que $P_{i_0}^{j_0}$ filtre v_0 et l'expression E_{i_0} (si elle est présente) est évaluée à vrai dans le contexte des variables initialisées par $P_{i_0}^{j_0}$. Si une telle branche existe, alors la valeur de l'expression "**case**" est égale à celle de l'expression E'_{i0} (évaluée dans le contexte des variables initialisées par $P_{i_0}^{j_0}$). Si une telle branche n'existe pas, la valeur de l'expression "**case**" est égale à celle de E'_{m+1} si la clause "**otherwise**" est présente ; si celle-ci est absente, l'exécution du programme XTL est interrompue et une erreur est signalée.

Exemple 2-15
L'expression "**case**" suivante renvoie le premier élément d'une liste de nombres entiers `l` contenant entre 1 et 3 éléments, ou renvoie 0 si la liste est vide :

```
case l in
    nil -> 0
  | cons (n : integer, nil)
  | cons (n : integer, cons (any integer, nil))
  | cons (n : integer, cons (any integer, cons (any integer, nil))) -> n
endcase
```

Le filtre `any integer` est utilisé pour filtrer des valeurs qui ne seront plus utilisées par la suite. Si le nombre d'éléments de `l` est différent de 1, 2, ou 3, une erreur sera signalée à l'exécution et le programme XTL sera arrêté. ■

Remarque 2-6
Les expressions dont le type ne possède pas d'opérateurs constructeurs (par exemple, le type XTL prédéfini `real`) peuvent être filtrées dans les expressions "**case**" à condition de ne pas utiliser des filtres avec constructeur "C **(**P_1**,**$\ldots$**,**P_n**)**" (voir la section 2.5.4). ■

2.5.6 Expression "case" sur actions

Le langage XTL offre aussi une autre forme d'expression "**case**", adaptée pour le filtrage des étiquettes du modèle STE. Cette expression spéciale, appelée "**case action**", a la syntaxe suivante :

$$
\begin{array}{l}
\textbf{case action } E_0 \textbf{ in} \\
\quad \alpha_1 \; [\textbf{where } E_1] \; \texttt{->} \; E'_1 \\
\quad \ldots \\
\; | \; \alpha_m \; [\textbf{where } E_m] \; \texttt{->} \; E'_m \\
[| \; \textbf{otherwise} \; \texttt{->} \; E'_{m+1}] \\
\textbf{endcase}
\end{array}
$$

où les expressions optionnelles $E_1, \ldots, E_m$ doivent être de type booléen, les expressions $E'_1, \ldots, E'_{m+1}$ doivent avoir le même type, E_0 doit avoir le type `label` et $\alpha_1, \ldots, \alpha_m$ sont des formules sur actions (voir la section 2.8). Les formules sur actions α peuvent, d'une manière similaire aux filtres P, définir et initialiser des variables x utilisables ultérieurement dans le programme. Pour chaque branche $1 \leq i \leq m$ de l'expression "**case action**", les variables initialisées par la formule α_i ne sont visibles que dans les expressions E_i et E'_i.

L'évaluation d'une expression "**case action**" est similaire à celle d'une expression "**case**" normale. Après avoir calculé la valeur v_0 de l'expression E_0, chaque branche i (pour i allant de 1 à m) est considérée afin de trouver la première branche i_0 pour laquelle v_0 satisfait α_{i_0} et l'expression E_{i_0} est évaluée à vrai dans le contexte des variables initialisées par α_{i_0}. Si une telle branche existe, la valeur de l'expression "**case action**" est égale à celle de l'expression E'_{i_0} (évaluée dans le contexte des variables initialisées par α_{i_0}). Si v_0 ne satisfait aucune des formules α_i, la valeur de l'expression "**case action**" est égale à celle de E'_{m+1} si la clause "**otherwise**" est présente ; si celle-ci est absente, une erreur est signalée et l'exécution du programme XTL est arrêtée.

Exemple 2-16
L'expression "**case action**" suivante examine la structure d'une étiquette `a` du STE et imprime sur le fichier de sortie des informations concernant son contenu :

```
case action a in
    SEND ? n : integer -> print ("Emission de ", n, "\n")
  | RECV ? n : integer -> print ("Reception de ", n, "\n")
  | otherwise -> print ("Autre action\n")
endcase
```

Les deux formules sur actions `SEND ? n : integer` et `RECV ? n : integer`, où `SEND` et `RECV` sont des portes (fonctions constantes de type `gate`), caractérisent les actions d'envoi et de réception d'un nombre entier `n`. ■

2.6 Expressions d'itération

XTL permet de décrire des traitements répétitifs des données, au moyen de différentes constructions d'itération. Il existe des constructions générales comme les expressions "**loop**" et "**for**", ainsi que des constructions plus spécialisées comme les itérateurs, les ensembles et les quantificateurs. XTL étant un langage fonctionnel, toutes les constructions d'itération sont conçues en fonction de ce paradigme et, notamment, elles envoient un résultat.

2.6.1 Expression "loop"

La construction d'itération la plus générale offerte par le langage XTL est l'expression "**loop**". Elle a la syntaxe suivante :

$$\textbf{loop}\ (x_0{:}T_0{:=}E_0, \ldots, x_n{:}T_n{:=}E_n) : RT\ \textbf{in}\\ E\\ \textbf{endloop}$$

où les variables $x_0, \ldots, x_n$, appelées *paramètres*, sont visibles dans l'expression E de type RT, appelée *corps* de l'opérateur "**loop**". L'expression E peut contenir des expressions "**continue**", qui déterminent un "rebouclage" (similaire à l'instruction correspondante du langage C) :

$$\textbf{continue}\ (E'_0, \ldots, E'_n)$$

Le nombre et types des expressions $E'_0, \ldots, E'_n$ doivent correspondre aux paramètres de l'expression "**loop**" respective.

Le résultat d'une expression "**loop**" est égal à la valeur de son corps E calculée dans le contexte des paramètres $x_0, \ldots, x_n$ initialisés avec les valeurs de $E_0, \ldots, E_n$. Chaque occurrence d'une expression **continue**($E'_0, \ldots, E'_n$) dans E dénote la valeur de la boucle "**loop**" calculée avec les valeurs de $E'_0, \ldots, E'_n$ associées aux paramètres $x_0, \ldots, x_n$. L'expression E est donc évaluée de manière itérative, avec des valeurs différentes pour les paramètres $x_0, \ldots, x_n$ à chaque itération. La boucle se termine lorsque l'expression E peut être évaluée sans rencontrer d'expressions "**continue**".

Les expressions "**continue**" ne peuvent être utilisées que dans les corps des expressions "**loop**". Une occurrence de "**continue**" est associée à la plus petite expression "**loop**" qui la contient.

Exemple 2-17
La factorielle `n`! d'un nombre naturel `n` peut être calculée au moyen de l'expression "**loop**" suivante :

```
loop (fact : integer := 1, k : integer := n) : integer in
    if k = 0 then
        fact                          (* resultat de la boucle *)
    else
        continue (fact * k, k - 1)    (* nouvelle iteration *)
    endif
endloop
```

La propriété `fact` = `n`!/`k`! est invariante et la terminaison est assurée par le fait que `k` décroît à chaque itération ; le résultat de l'expression "**loop**" est égal à `fact` = `n`!/`0`! = `n`!. ■

Une expression "**loop**" est équivalente à la définition et l'appel *in situ* d'une fonction récursive F ayant les paramètres $x_0, \ldots, x_n$, le corps E et le résultat de type RT. Les occurrences d'expressions **continue**($E'_0, \ldots, E'_n$) dans E correspondent aux appels récursifs de F avec les arguments $E'_0, \ldots, E'_n$. La valeur de l'expression "**loop**" est égale au résultat de l'appel de F avec les arguments $E_0, \ldots, E_n$.

Remarque 2-7
Il serait donc possible d'implémenter les expressions "**loop**" par traduction vers des fonctions récursives. Toutefois, cette implémentation ne serait pas nécessairement optimale, notamment pour les expressions "**loop**" ayant la propriété de *récursion terminale* (*tail recursion*), c'est-à-dire pour lesquelles toutes les occurrences d'expressions "**continue**" dans leur corps E sont des *points de sortie* de E (sous-expressions qui sont calculées en dernier lors de l'évaluation de E). ■

Utilisées conjointement avec les méta-opérateurs d'exploration de la relation de transition (voir la section 2.3.1), les expressions "**loop**" permettent de définir des opérateurs de logique temporelle par calcul itératif d'ensembles d'états du STE.

Exemple 2-18
L'opérateur **pot**(φ) de LTAC, qui est similaire à la modalité **EF**φ de CTL, caractérise les états du STE à partir desquels il existe un chemin menant à un état qui satisfait φ. Cet opérateur peut être implémenté en XTL au moyen de l'expression "**loop**" suivante (où la variable `P` de type `stateset` contient les états satisfaisant φ) :

```
loop (acc : stateset := empty) : stateset in
    let acc2 : stateset := P union pred (acc) in
        if acc2 = acc then
            acc
        else
            continue (acc2)
        endif
    endlet
endloop
```

Les états satisfaisant **pot**(φ) sont accumulés dans le paramètre `acc` (initialisé à l'ensemble vide). Après la k-ième itération, `acc` contient les états qui peuvent mener, après au plus $k-1$ transitions, à un état dans `P` ; les itérations sont arrêtées lorsque `acc` devient stable, ceci signifiant qu'il contient tous les états satisfaisant **pot**(φ). ■

Les expressions "**loop**" peuvent être utilisées pour effectuer plusieurs calculs en même temps et renvoyer plusieurs résultats ; dans ce cas, leur type doit être un tuple à plusieurs champs.

Exemple 2-19
Nous voulons calculer simultanément deux ensembles d'états du STE, qui sont origine d'un chemin infini dont les états d'indice pair (*resp.* impair) satisfont φ_1 et ceux d'indice impair (*resp.* pair) satisfont φ_2. Ces ensembles d'états peuvent être caractérisés dans la logique de Dicky par le système d'équations suivant (où le signe $-$ indique qu'il s'agit de variables de plus grand point fixe) :

$$\begin{cases} X_1^- \; = \; \varphi_1 \; \wedge \; \mathbf{src}(\mathbf{in}(X_2)) \\ X_2^- \; = \; \varphi_2 \; \wedge \; \mathbf{src}(\mathbf{in}(X_1)) \end{cases}$$

Utilisant les méta-opérateurs disponibles en XTL, la solution de ce système d'équations peut être calculée itérativement au moyen de l'expression "**loop**" suivante (où les variables `P1` et `P2` de type `stateset` contiennent respectivement les états satisfaisant φ_1 et φ_2) :

```
loop (X1, X2 : stateset := full) : (stateset, stateset) in
    let Y1 : stateset := P1 inter pred (X2),
        Y2 : stateset := P2 inter pred (X1)
    in
        if X1 = Y1 and X2 = Y2 then
            (X1, X2)
        else
            continue (Y1, Y2)
        endif
    endlet
endloop
```

Les deux ensembles d'états recherchés sont calculés dans les paramètres `X1` et `X2` (initialisés avec l'ensemble de tous les états du STE). Après la k-ième itération, `X1` et `X2` contiennent respectivement les états qui sont origine d'un chemin de longueur k dont les états d'indice pair (*resp.* impair) sont dans `P1` et les états d'indice impair (*resp.* pair) sont dans `P2`. Le calcul est arrêté quand `X1` et `X2` deviennent stables, c'est-à-dire égaux à la solution du système d'équations correspondant. ■

2.6.2 Expression "for"

XTL contient une seconde construction d'itération permettant de décrire les traitements répétitifs portant sur des domaines de valeurs finis. Il s'agit de l'expression "**for**", ayant la syntaxe suivante :

for x'_0:T'_0[**among** E'_0], ... ,x'_m:T'_m[**among** E'_m]
  [**var** x_0:T_0:=E_0, ... ,x_n:T_n:=E_n]
  [**where** E''_1]
  [**while** E''_2]
   in E''_3
  [**result** E''_4]
endfor

où les variables $x'_0, \ldots, x'_m$, appelées *variables d'itération*, sont visibles dans les expressions booléennes E''_1 et E''_2 et dans le *corps* E''_3 de l'expression "**for**", mais pas dans l'expression E''_4. Les variables optionnelles $x_0, \ldots, x_n$, appelées *accumulateurs*, sont visibles dans les expressions E''_1, E''_2, E''_3 et E''_4. Si $n \geq 0$ accumulateurs sont présents, le corps E''_3 doit avoir le type $(T_0, \ldots, T_n)$. Les constructions T'_0[**among** E'_0], ... , T'_m[**among** E'_m], où les expressions optionnelles $E'_0, \ldots, E'_m$ dénotent des sous-ensembles des types respectifs, sont appelées *domaines d'itération*.

Une expression "**for**" est évaluée de la manière suivante. Les accumulateurs $x_0, \ldots, x_n$ sont initialisés avec les valeurs des expressions $E_0, \ldots, E_n$. Ensuite, pour chaque valeur des variables $x'_0, \ldots, x'_m$ dans leurs domaines respectifs (qui doivent être finis), une itération est effectuée : elle consiste à évaluer le corps E''_3 (en utilisant les valeurs courantes des variables d'itération et des accumulateurs) et à mémoriser sa valeur dans les accumulateurs. La clause optionnelle "**where**" permet d'effectuer uniquement les itérations pour lesquelles l'expression E''_1 est vraie. La clause optionnelle "**while**" permet d'arrêter les itérations dès que l'expression E''_2 devient fausse. Une fois les itérations finies, si la clause optionnelle "**result**" est présente, le résultat de l'expression "**for**" est égal à la valeur de E''_4 (calculée en utilisant les valeurs des accumulateurs obtenues après la dernière itération) ; sinon, il est égal au tuple contenant les valeurs des accumulateurs.

Exemple 2-20
La factorielle `n`! d'un nombre naturel `n` peut être calculée au moyen de l'expression "**for**" suivante :

```
for k : integer among { 1 ... n }
    var fact : integer := 1
    in  fact * k
endfor
```

La propriété `fact` = `k`! est invariante pour toutes les itérations de `k` entre `1` et `n` ; le résultat de l'expression "**for**" est la valeur de l'accumulateur `fact` à la dernière itération, c'est-à-dire `n`!. ■

Les types autorisés pour les variables d'itération sont tous les types T finis *énumérables*, c'est-à-dire les types pour lesquels on dispose d'un ordonnancement de leurs valeurs $v_1, \ldots, v_n$ et des trois opérateurs suivants :

- $init_T : \rightarrow T$, qui renvoie v_1, la "première" valeur du domaine de T ;
- $inc_T : T \rightarrow T$, qui, appliqué sur une valeur v_i $(i < n)$, renvoie la valeur "suivante" v_{i+1} ;
- $end_T : T \rightarrow$ `boolean`, qui renvoie vrai pour v_n, la dernière valeur du domaine de T.

Les types énumérables comprennent les types XTL prédéfinis de base `integer`, `boolean` et `character`, les méta-types `state`, `label` et `trans`, ainsi que les types BCG énumérables provenant du programme parallèle à vérifier (par exemple, certains des types produits par le compilateur CÆSAR.ADT à partir de programmes LOTOS).

Exemple 2-21
L'opérateur $\mathbf{EX}\varphi$ de CTL caractérise les états du STE qui ont (au moins) un état successeur satisfaisant la formule φ. En supposant que la variable `p` de type `stateset` contient les états qui satisfont φ, l'ensemble des états satisfaisant $\mathbf{EX}\varphi$ peut être calculé au moyen de l'expression "**for**" suivante :

```
for s : state among p
    var acc : stateset := empty
    in  acc union pred (s)
endfor
```

La variable d'itération `s` parcourt tous les états appartenant à `p` et chaque itération rajoute à l'accumulateur `acc` (initialisé à l'ensemble vide) les états prédécesseurs de `s`. ■

La clause "**result**" est utile quand le résultat de l'expression "**for**" dépend des valeurs calculées dans les accumulateurs après la dernière itération.

Exemple 2-22
L'expression "**for**" suivante calcule la différence entre la valeur maximale et la valeur minimale que peut prendre la variable entière `x` dans les états du STE :

```
for s : state
    var xmax, xmin : integer := init.x
    in  (max (xmax, s.x), min (xmin, s.x))
    result xmax - xmin
endfor
```

Les valeurs maximale et minimale de la variable `x` sont calculées dans les accumulateurs `xmax` et `xmin`, qui sont initialisés avec la valeur de `x` dans l'état initial du STE et sont mis à jour à chaque itération de la variable `s` qui parcourt les états du STE. Une fois les itérations finies, le résultat de l'expression "**for**" est produit à l'aide de la clause "**result**". ■

Les expressions "**for**" peuvent être traduites, quoique d'une façon assez fastidieuse, en termes d'expressions "**loop**". A titre d'exemple, nous donnons ci-dessous la traduction d'une expression "**for**" avec une seule variable d'itération et sans clause "**result**" :

for x:T[**among** E]
  var x_0:T_0:=E_0,
    ...
    x_n:T_n:=E_n
  [**where** E_1'']
  [**while** E_2'']
  in E_3''
endfor
	$\stackrel{\mathrm{d}}{=}$	**loop** (x:T := $init_T()$,x_0:T_0 := E_0, ... ,x_n:T_n := E_n)
		    : (T_0, ... ,T_n) **in**
		  if $end_T(x)$ [**or not**(E_2'')] **then**
		    (x_0, ... , x_n)
		  [**elsif not**(E_1'' [**and** x `isin` E]) **then**
		    continue ($inc_T(x)$,x_0, ... ,x_n)]
		  else
		    let (x_0':T_0, ... ,x_n':T_n):=E_3'' **in**
		      continue ($inc_T(x), x_0', \ldots, x_n'$)
		    endlet
		  endif
		endloop

Les itérations sont contrôlées au moyen des opérateurs $init_T$, inc_T et end_T appliqués sur la variable x. La traduction des expressions "**for**" ayant plusieurs variables d'itération nécessite plusieurs "**loop**" imbriqués et celle des expressions "**for**" avec clause "**result**" nécessite un "**let**" destructurant supplémentaire (voir l'annexe B.1.2).

Un cas particulier utile est constitué par les expressions "**for**" de type `void`, pour lesquelles la présence de l'accumulateur n'est pas obligatoire. Dans ce cas, un accumulateur implicite *acc*, initialisé à **nop**, est utilisé afin de mémoriser les résultats des itérations intermédiaires :

$$
\begin{array}{lcl}
 & & \textbf{loop}\ (x{:}T := init_T(),\ acc : \texttt{void} := \textbf{nop})\ :\ \texttt{void}\ \textbf{in} \\
 & & \quad \textbf{if}\ end_T(x)\ [\textbf{or not}(E_2''))]\ \textbf{then} \\
\textbf{for}\ x{:}T[\textbf{among}\ E] & & \qquad acc \\
\quad [\textbf{where}\ E_1''] & & \quad \left[\begin{array}{l} \textbf{elsif not}(E_1''\ [\textbf{and}\ x\ \texttt{isin}\ E])\ \textbf{then} \\ \quad \textbf{continue}\ (inc_T(x), acc) \end{array}\right] \\
\quad [\textbf{while}\ \ E_2''] & \stackrel{\mathrm{d}}{=} & \\
\quad \textbf{in}\ E_3'' & & \quad \textbf{else} \\
\textbf{endfor} & & \qquad \textbf{continue}\ (inc_T(x), E_3'') \\
 & & \quad \textbf{endif} \\
 & & \textbf{endloop}
\end{array}
$$

Exemple 2-23
L'expression "**for**" suivante imprime sur le fichier de sortie les valeurs de la variable `x` dans les états successeurs de l'état initial du STE :

```
for s : state among succ (init) in
    print (s.x)
endfor
```

où `print` est la fonction d'impression associée au type de la variable `x`. ■

Outre les constructions d'itération générales "**loop**" et "**for**", XTL offre aussi des constructions plus simples, permettant la description plus concise de traitements répétitifs particuliers. Ces constructions sont présentées en détail aux sections suivantes.

2.6.3 Itérateurs

Les notations mathématiques utilisent fréquemment des opérateurs de somme généralisée $\sum_i$, de produit généralisé Π_i, etc. XTL possède des notations équivalentes, appelées *itérateurs*, pour exprimer l'application itérative d'un même opérateur. Les itérateurs ont la syntaxe suivante :

$$\{\ F\ \textbf{on}\ x_0{:}T_0\ [\textbf{among}\ E_0], \ldots, x_n{:}T_n\ [\textbf{among}\ E_n]\ [\textbf{where}\ E_1']\ \}\ E_2'$$

où F est une fonction binaire ayant un profil *homogène à gauche* (c'est-à-dire $F : T_a \times T_b \to T_a$), les variables d'itération $x_0, \ldots, x_n$ parcourent les (sous-domaines finis $E_0, \ldots, E_n$ des) types $T_0, \ldots, T_n$, l'expression booléenne E_1' sert de garde pour les itérations et l'expression E_2', de type T_b, est appelée *corps* de l'itérateur.

Les itérateurs ne sont pas des constructions primitives XTL ; ils peuvent être traduits en termes d'expressions "**for**" de la manière suivante :

$$
\begin{array}{lcl}
\{\ F\ \textbf{on}\ x_0{:}T_0\ [\textbf{among}\ E_0], & & \textbf{for}\ x_0{:}T_0[\textbf{among}\ E_0], \ldots, x_n{:}T_n[\textbf{among}\ E_n] \\
\qquad \ldots & & \quad \textbf{var}\ acc\ :\ T_a\ :=\ init\_val(F) \\
\qquad x_n{:}T_n\ [\textbf{among}\ E_n] & \stackrel{\mathrm{d}}{=} & \quad [\textbf{where}\ E_1'] \\
\qquad [\textbf{where}\ E_1'] & & \quad \textbf{in}\ F(acc, E_2') \\
\}\ E_2' & & \textbf{endfor}
\end{array}
$$

Pour chaque valeur des variables d'itération $x_0, \ldots, x_n$ dans leurs domaines respectifs, le corps E_2' de l'itérateur est évalué, en appliquant la fonction F, et "accumulé" dans une variable interne *acc* ayant le même type T_a que le résultat de F. L'accumulateur *acc* est initialisé au début de l'itération avec une valeur *init_val*(F), associée par défaut à la fonction F, et sa valeur à la fin de l'itération est renvoyée comme résultat. La clause "**where**" permet d'effectuer les itérations uniquement pour les valeurs des variables $x_0, \ldots, x_n$ qui satisfont l'expression booléenne E_1'.

Les valeurs *init_val* associées par défaut aux opérateurs prédéfinis sont données dans la table 2.7. Dans tous les cas (sauf pour les opérateurs `diff`, `insert` et `remove`), il s'agit de l'*élément neutre à gauche* de l'opérateur respectif. Les opérateurs ensemblistes `union`, `inter`, `diff`, `insert` et `remove` sont surchargés : les types génériques `elem` *resp.* `set` utilisés dans la table peuvent être instanciés deux à deux par `integer` et `intset`, `character` et `charset`, `state` et `stateset`, `label` et `labelset`, `trans` et `transset`.

Opérateur d'itération	Valeur initiale
`or  : boolean, boolean -> boolean`	`false`
`and : boolean, boolean -> boolean`	`true`
`+   : integer, integer -> integer`	`0`
`*   : integer, integer -> integer`	`1`
`max : integer, integer -> integer`	`0`
`union : set, set -> set`	`empty`
`inter : set, set -> set`	`full`
`diff  : set, set -> set`	`full`
`insert : set, elem -> set`	`empty`
`remove : set, elem -> set`	`full`
`; : action, action -> action`	`nop`

Table 2.7: Valeurs initiales associées par défaut aux opérateurs d'itération

Exemple 2-24
La somme des nombres naturels inférieurs ou égaux à `n`, écrite mathématiquement $\sum_{\mathtt{k}=1}^{\mathtt{n}} \mathtt{k}$, peut être calculée avec l'itérateur XTL suivant :

```
{ + on k : integer among { 1 ... n } } k
```

où l'accumulateur implicite est initialisé avec *init_val*(`+`) $= 0$. ■

Les domaines des variables d'itération ne se limitent pas aux types prédéfinis de base ; les itérateurs sur les méta-types `state`, `label` et `trans` correspondant aux états, étiquettes et transitions du modèle STE sont également autorisés.

Exemple 2-25
Le facteur de branchement moyen du STE, défini comme le rapport entre le nombre de transitions et le nombre d'états du STE, peut être calculé au moyen d'itérateurs XTL comme suit :

```
({ + on t : trans } 1) / ({ + on s : state } 1)
```

■

Les itérateurs peuvent être imbriqués d'une manière similaire aux notations mathématiques : du point de vue syntaxique, ils sont considérés comme des opérateurs unaires préfixés.

Exemple 2-26
Le facteur de branchement maximal du STE, défini comme le maximum du nombre de transitions issues des états, peut être calculé à l'aide de deux itérateurs imbriqués :

```
{ max on s : state } { + on t : trans among out (s) } 1
```

où l'accumulateur implicite du premier itérateur est initialisé avec *init_val*(`max`) $= 0$. ■

2.6.4 Ensembles définis en compréhension

La notation "**{ }**" (voir la section 2.3.1) sert à décrire des ensembles en *extension*, c'est-à-dire en énumérant leurs éléments. XTL permet aussi de décrire des ensembles en *compréhension*, c'est-à-dire en caractérisant leurs éléments, grâce à l'expression suivante :

$$\textbf{\{ } x\textbf{:}T \text{ [}\textbf{among } E_1\text{]} \textbf{ where } E_2 \textbf{ \}}$$

où x est une variable itérant sur le (sous-domaine E_1 du) type T et l'expression booléenne E_2 représente le prédicat sur x qui permet de sélectionner les valeurs appartenant à l'ensemble voulu.

Les ensembles en compréhension peuvent être traduits vers des itérateurs XTL de la façon suivante :

$$\boxed{\textbf{\{ } x\textbf{:}T \text{ [}\textbf{among } E_1\text{]} \textbf{ where } E_2 \textbf{ \}} \quad \stackrel{\text{d}}{=} \quad \textbf{\{ } \texttt{insert} \textbf{ on } x\textbf{:}T \text{ [}\textbf{among } E_1\text{]} \textbf{ where } E_2 \textbf{ \}} \ x}$$

Un ensemble en compréhension est donc défini comme l'union des valeurs de la variable x appartenant au domaine considéré qui satisfont le prédicat E_2.

Exemple 2-27
L'ensemble des nombres naturels multiples de `3` et inférieurs ou égaux à `n`, écrit mathématiquement $\{\mathtt{k} \in [\mathtt{0}, \mathtt{n}] \mid (\mathtt{k}\ mod\ \mathtt{3}) = \mathtt{0}\}$, peut être calculé de la façon suivante :

```
{ k : integer among { 0 ... n } where (n mod 3) = 0 }
```

■

Les ensembles en compréhension ne se limitent pas aux types prédéfinis de base ; le calcul d'ensembles d'états, de transitions et d'étiquettes du STE est également autorisé.

Exemple 2-28
L'ensemble des états de blocage du STE, appelés aussi *états puits* (*sink states*), c'est-à-dire les états n'ayant pas de successeurs, peut être calculé de la manière suivante :

```
{ s : state where succ (s) = empty }
```

La condition contenue dans la clause "**where**" pourrait être exprimée aussi au moyen d'un quantificateur sur les transitions successeurs de `s` (voir l'exemple 2-30). ■

2.6.5 Quantificateurs

Une classe particulière d'opérateurs XTL d'itération est constituée par les expressions booléennes "**exists**" et "**forall**", appelées *quantificateurs* (par analogie avec ∃ et ∀) :

$$\textbf{exists } x_0\textbf{:}T_0 \text{ [}\textbf{among } E_0\text{]}, \ldots, x_n\textbf{:}T_n \text{ [}\textbf{among } E_n\text{]} \textbf{ in } E$$

$$\textbf{forall } x_0\textbf{:}T_0 \text{ [}\textbf{among } E_0\text{]}, \ldots, x_n\textbf{:}T_n \text{ [}\textbf{among } E_n\text{]} \textbf{ in } E$$

où les variables d'itération $x_0, \ldots, x_n$ parcourent les (sous-domaines finis $E_0, \ldots, E_n$ des) types $T_0, \ldots, T_n$ et sont visibles dans les expressions booléennes E, appelées *corps* des quantificateurs.

La sémantique des quantificateurs est évidente : une expression "**exists**" est vraie ssi il existe des valeurs $v_0, \ldots, v_n$ pour les variables $x_0, \ldots, x_n$ dans leurs domaines respectifs telles que le corps E de l'expression est évalué à vrai. La sémantique de "**forall**" est définie de manière duale.

Les quantificateurs peuvent être traduits en termes d'expressions "**for**" de la manière suivante :

exists x_0:T_0 [**among** E_0], ... x_n:T_n [**among** E_n] **in** E	$\stackrel{\mathrm{d}}{=}$	**for** x_0:T_0[**among** E_0], ... ,x_n:T_n[**among** E_n] **var** *acc* : `boolean` := **false** **while not** (*acc*) **in** *acc* **or** E **endfor**
forall x_0:T_0 [**among** E_0], ... x_n:T_n [**among** E_n] **in** E	$\stackrel{\mathrm{d}}{=}$	**for** x_0:T_0[**among** E_0], ... ,x_n:T_n[**among** E_n] **var** *acc* : `boolean` := **true** **while** *acc* **in** *acc* **and** E **endfor**

Le quantificateur "**exists**" (*resp.* "**forall**") est donc défini comme la disjonction (*resp.* la conjonction) généralisée de son corps E pour toutes les valeurs des variables $x_0, \ldots, x_n$ dans leurs domaines respectifs. Les clauses "**while**" sont utilisées afin d'optimiser l'évaluation : elles permettent d'arrêter les itérations dès que la valeur de vérité du quantificateur en cause a été déterminée.

Les expressions "**exists**" et "**forall**" ont une utilisation naturelle dans l'expression des propriétés temporelles impliquant la quantification sur des domaines (finis) de valeurs : sans disposer de ces expressions, l'utilisateur serait obligé d'écrire des suites fastidieuses de propriétés (une pour chaque tuple de valeurs des variables quantifiées).

Exemple 2-29
Soit un programme parallèle contenant `n` processus concurrents (numérotés de `1` à `n`) qui utilisent une ressource partagée. La propriété d'exclusion mutuelle est vérifiée ssi il n'existe pas deux processus différents `p1` et `p2` qui utilisent simultanément la ressource. Sachant que la variable `processes` (définie dans le programme à vérifier) contient l'information sur l'état courant des `n` processus, cette propriété peut être exprimée, sur un état `s` du modèle STE sous-jacent, de la façon suivante :

```
not exists p1, p2 : integer among { 1 ... n } in (
    active (s.processes, p1) and active (s.processes, p2) and
    (p1 <> p2)
)
```

où on suppose que le prédicat `active` (défini dans le programme à vérifier) caractérise les processus qui sont en train d'utiliser la ressource. ■

Les domaines des variables quantifiées ne se limitent pas aux types prédéfinis de base ; la quantification sur des états, transitions ou étiquettes du STE (et, plus généralement, sur tous les types énumérables) est également autorisée.

Exemple 2-30
La modalité $[\alpha]\,\varphi$ de HML caractérise les états du STE dont toutes les transitions successeur ayant une étiquette qui satisfait la formule α mènent à des états satisfaisant la formule φ. En supposant que la variable `a` de type `labelset` (*resp.* `p` de type `stateset`) contient les étiquettes (*resp.* les états) du STE satisfaisant α (*resp.* φ), l'expression suivante calcule l'ensemble d'états satisfaisant $[\alpha]\,\varphi$:

```
{ s : state where
    forall t : trans among out (s) in
        (label (t) isin a) implies (target (t) isin p)
}
```

La modalité $\langle\alpha\rangle\,\varphi$ de HML peut être définie de manière similaire. ■

2.7 Définitions de fonctions

Cette section est consacrée aux constructions du langage XTL permettant à l'utilisateur de définir et utiliser des fonctions. Les autres classes de fonctions utilisables dans un programme XTL (les fonctions XTL prédéfinies et les fonctions BCG) ont été présentées à la section 2.3. Une définition de fonction XTL a la syntaxe suivante :

$$\begin{array}{l}[\textbf{local}]\,\textbf{function}\ F\ (x_1{:}T_1,\ldots,x_m{:}T_m)\ \textbf{:}\ RT\ \textbf{is}\\ \quad E\\ \textbf{endfunc}\end{array}$$

où F est l'identificateur de la fonction, $x_1, \ldots, x_m$ sont ses *paramètres*, E est son *corps* et RT est le type de son résultat. Les appels de la fonction F dans le programme XTL ont la syntaxe habituelle utilisée dans les langages de programmation :

$$F\ (E_1, \ldots, E_n)$$

où $E_1, \ldots, E_n$ sont les *arguments* de l'appel de F. Dans les appels des fonctions sans paramètres, les parenthèses peuvent être omises. Le passage des paramètres est fait par valeur (*call-by-value*) : les expressions $E_1, \ldots, E_n$ sont évaluées de gauche à droite et leurs valeurs sont transmises comme arguments à F.

Exemple 2-31
En reprenant l'exemple 2-21, l'opérateur $\mathbf{EX}\varphi$ de CTL peut être défini par la fonction XTL ci-dessous :

```
function EX (p : stateset) : stateset is
    { insert on s : state among p, s2 : state among pred (s) } s2
endfunc
```

Sachant que le paramètre `p` dénote l'ensemble d'états du STE satisfaisant φ, la fonction renvoie comme résultat l'ensemble des états satisfaisant $\mathbf{EX}\varphi$. Par souci de concision, nous avons préféré d'utiliser un itérateur au lieu d'une expression "**for**" explicite comme dans l'exemple 2-21. ■

Le mot-clé optionnel "**local**" sert à contrôler la visibilité des fonctions définies : les fonctions *locales* (dont la définition est précédée par "**local**") ne sont visibles que dans les corps des autres fonctions définies dans le programme XTL, tandis que les fonctions *globales* (dont la définition n'est pas précédée par "**local**") sont visibles dans tout le programme XTL (voir la section 2.15 pour la structure d'un programme XTL). Les définitions de fonctions locales sont utiles pour la construction de *bibliothèques*, c'est-à-dire de fichiers séparés, contenant des définitions de fonctions, qui peuvent être inclus dans le programme XTL (voir la section 2.14). Le fait que les fonctions locales définies dans une bibliothèque ne puissent pas être utilisées dans le corps du programme XTL assure une certaine modularité et limite les risques d'appels erronés de fonctions dans les programmes XTL utilisant des bibliothèques.

Bien entendu, la définition de fonctions XTL (directement ou mutuellement) récursives est autorisée.

Exemple 2-32
La fonction `EU` définie ci-dessous implémente l'opérateur $\mathbf{E}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$ de CTL, qui caractérise les états du STE à partir desquels il existe un chemin menant à un état satisfaisant la formule φ_2 et dont tous les états intermédiaires satisfont la formule φ_1 :

```
function EU (p1, p2 : stateset) : stateset is
    EU2 (p1, p2, empty)
endfunc
```

```
local function EU2 (p1, p2, acc : stateset) : stateset is
    let acc2 : stateset := p2 union (p1 inter EX (acc)) in
        if acc2 = acc then
            acc
        else
            EU2 (p1, p2, acc2)
        endif
    endlet
endfunc
```

Les paramètres p1 et p2 dénotent les ensembles d'états du STE satisfaisant respectivement les formules φ_1 et φ_2. Le calcul proprement dit est effectué au moyen de la fonction récursive locale EU2. L'ensemble d'états constituant le résultat est accumulé dans le paramètre acc (initialisé à l'ensemble vide). Au k-ième appel de la fonction EU2, acc contient tous les états s du STE qui sont source d'un chemin de longueur inférieure à $k-1$, dont tous les états intermédiaires (y compris s) satisfont φ_1 et qui mène à un état satisfaisant φ_2. La fonction EX, implémentant l'opérateur $\mathbf{EX}\varphi$ (voir l'exemple 2-31) est utilisée pour calculer la nouvelle valeur du paramètre acc à chaque appel de EU2. La terminaison est assurée par le fait que la longueur d'un chemin ne peut pas dépasser le nombre d'états du STE ; la valeur stable du paramètre acc, contenant exactement les états qui satisfont $\mathbf{E}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$, est renvoyée comme résultat. ■

La surcharge des fonctions est également autorisée : plusieurs fonctions (ayant le même nom mais des profils différents) peuvent être simultanément visibles. Ce mécanisme a une double utilité : d'une part, il permet de définir des synonymes pour les divers opérateurs temporels (à ce sujet, voir aussi la section 2.13) et, d'autre part, il assure la compatibilité avec les langages de description des programmes à vérifier (comme LOTOS), qui permettent la définition de fonctions surchargées.

Remarque 2-8
Les éventuelles ambiguïtés provoquées par les surcharges de fonctions peuvent être résolues à l'aide de l'opérateur de typage "**of**". Par exemple, l'expression suivante :

```
print (full of stateset)
```

imprime l'ensemble des états du STE sur le fichier de sortie, tandis que

```
print (full of labelset)
```

imprime l'ensemble des étiquettes du STE. L'expression `print (full)` serait ambiguë, car l'opérateur `full` est surchargé (voir la section 2.3.1). ■

XTL permet aussi de définir des opérateurs *infixés*, c'est-à-dire des fonctions ayant deux arguments et dont les appels peuvent être écrits en notation infixée. Les définitions d'opérateurs infixés ont la syntaxe suivante :

$$\begin{array}{l}[\textbf{local}]\,\textbf{function}\ \_\ F\ \_\ (x_1\textbf{:}T_1\textbf{,}x_2\textbf{:}T_2)\ \textbf{:}\ RT\ \textbf{is}\\ \quad E\\ \textbf{endfunc}\end{array}$$

Les appels d'une fonction F définie comme opérateur infixé peuvent être écrits en utilisant soit la notation préfixée $F(E_1, E_2)$, soit la notation infixée $E_1\ F\ E_2$. Les opérateurs binaires infixés définis dans les programmes XTL sont associatifs à droite : une expression $E_1\ F\ E_2\ F\ E_3$ est interprétée comme $E_1\ F\ (E_2\ F\ E_3)$. Cette règle est aussi valable pour les opérateurs prédéfinis qui sont des mots-clés du langage, comme "**or**", "**and**", "**xor**", "**implies**", etc. (voir la section 2.10.3).

Les définitions d'opérateurs infixés obéissent aux mêmes règles de visibilité qui sont associées aux fonctions XTL normales.

Exemple 2-33
En reprenant l'exemple 2-30, l'opérateur infixé `box` ci-dessous implémente la modalité $[\alpha]\,\varphi$ de HML :

```
function _ box _ (a : labelset, p : stateset) : stateset is
    { s : state where
        forall t : trans among out (s) in
            (label (t) isin a) implies (target (t) isin p)
    }
endfunc
```

Sachant que le paramètre `a` (*resp.* `p`) dénote l'ensemble d'étiquettes (*resp.* états) du STE satisfaisant la formule α (*resp.* φ), la fonction renvoie comme résultat l'ensemble des états satisfaisant $[\alpha]\,\varphi$. A titre d'exemple, la formule $[\alpha]\,[\alpha]\,[\alpha]\,\mathit{false}$ est dénotée par l'expression `a box a box a box false`, où la variable `a` de type `labelset` contient l'ensemble des étiquettes qui satisfont α. ■

Les fonctions XTL peuvent renvoyer des résultats de type tuple, ce qui est particulièrement utile quand il est nécessaire d'effectuer le calcul simultané de plusieurs résultats.

Exemple 2-34
La fonction `AU` ci-dessous implémente l'opérateur $\mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2]$ de CTL, qui caractérise les états du STE à partir desquels tous les chemins mènent à des états satisfaisant la formule φ_2, en passant par des états qui satisfont la formule φ_1 :

```
function AU (p1, p2 : stateset) : stateset is
   for s : state
      var    r, v : stateset := empty
      where  not (s isin v)
      in     AU2 (s, p1, p2, r, v)
      result r
   endfor
endfunc

local function AU2 (s : state, p1, p2, r, v : stateset) : (stateset, stateset) is
   let v2 : stateset := insert (v, s) in
      if s isin p2 then
         (insert (r, s), v2)
      elsif s isin p1 then
         for s2 : state among succ (s)
            var    b : boolean := true, ar : stateset := r, av : stateset := v2
            while  b
            in     if s2 isin av then (b and (s2 isin ar), ar, av)
                   else let (r3, v3 : stateset) := AU2 (s2, p1, p2, ar, av) in
                           (b and (s2 isin r3), r3, v3)
                        endlet
                   endif
            result if b then (insert (ar, s), av) else (ar, av) endif
         endfor
      else
         (r, v2)
      endif
   endlet
endfunc
```

Les paramètres `p1` et `p2` dénotent respectivement les ensembles d'états satisfaisant φ_1 et φ_2. A la différence de la fonction `EU` de l'exemple 2-32, qui implémente l'opérateur $\mathbf{E}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$ de CTL par calcul itératif d'ensembles d'états, la fonction `AU` ci-dessus calcule les états satisfaisant $\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$ par un parcours en profondeur du STE, suivant un algorithme donné en [Arn92, chapitre 5].

La fonction `AU` effectue une itération "**for**" sur tous les états `s` du STE, en accumulant respectivement dans les variables `r` et `v` les états satisfaisant $\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$ et ceux déjà visités durant le parcours. Le calcul proprement dit est effectué par la fonction récursive auxiliaire `AU2` : pour chaque état `s` qui n'a pas encore été visité, `AU2 (s, p1, p2, r, v)` parcourt en profondeur le STE à partir de `s`, afin de déterminer si `s` satisfait ou non $\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$. Les successeurs `s2` de `s` sont parcourus au moyen d'une expression "**for**" qui utilise une variable booléenne `b` pour indiquer si tous les successeurs `s2` déjà examinés satisfont $\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$ et deux variables `ar` et `av` pour accumuler les nouveaux états satisfaisant $\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$ et les nouveaux états visités. Le résultat d'`AU2` est un tuple contenant les nouvelles valeurs de `r` et `v` obtenues après le parcours.

Si $(\mathtt{r}_k, \mathtt{v}_k)$ est le résultat du k-ième appel de `AU2`, il est possible de montrer (par induction sur k) que $\mathtt{r}_k = \mathtt{v}_k \cap [\![\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]]\!]$, où $[\![\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]]\!]$ est l'ensemble d'états satisfaisant $\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$. Puisque `AU2` est exécutée une seule fois pour chaque état du STE et à la fin de l'algorithme tous les états ont été visités, la variable `r` (renvoyée comme résultat par `AU` au moyen de la clause "**result**" de l'expression "**for**") contient exactement les états satisfaisant $\mathbf{A}\,[\varphi_1\ \mathbf{U}\ \varphi_2]$. ■

2.8 Formules sur actions

XTL permet d'exprimer des propriétés caractérisant les actions du STE au moyen de *formules sur actions* (symbole non-terminal α). Ces formules, qui généralisent les constructions similaires de la logique ACTL, sont construites à l'aide d'opérateurs booléens et de prédicats atomiques permettant d'extraire les valeurs contenues dans les actions du STE. La sémantique d'une formule α est définie en termes des actions du STE qui la satisfont. Par la suite, nous utiliserons la terminologie suivante : "*évaluation d'une formule α sur une action a*" désigne le fait de déterminer si a satisfait α ou non ; en revanche, "*évaluation d'une formule α (sur un STE)*" signifie le calcul de l'ensemble des actions du STE qui satisfont α.

Les sections suivantes décrivent les différents opérateurs contenus dans les formules sur actions.

2.8.1 Offres

Les *offres*[9] sont utilisées dans les filtres d'actions (section 2.8.2) afin d'exprimer des prédicats sur les valeurs contenues dans les actions du STE. Une offre (symbole non-terminal O) est une construction permettant soit d'effectuer le filtrage d'une valeur v par un filtre P, soit de tester si v est égale à la valeur d'une expression E donnée. Les offres XTL ont la syntaxe suivante (similaire à la syntaxe des offres utilisées en LOTOS) :

$$\begin{array}{rcl} O & ::= & \mathbf{any} \\ & \mid & \mathbf{!}\ E \\ & \mid & \mathbf{?}\ P_0 \mid \ldots \mid P_n \end{array}$$

L'*application* d'une offre O sur une valeur v détermine soit si v a une structure compatible avec un filtre contenu dans O, soit si v est égale à la valeur de l'expression contenue dans O (on dit alors que *O filtre v*).

[9]Nous utilisons le terme "offre" par souci de compatibilité avec les constructions similaires du langage LOTOS.

Les différentes offres XTL, appliquées sur une valeur v, produisent l'effet suivant :

- "**any**" filtre toujours v, quel que soit son type ;
- "**!** E" filtre v ssi la valeur de E est égale à v ;
- "**?** $P_0 \mid \ldots \mid P_n$" filtre v s'il existe $0 \leq i \leq n$ tel que v est filtrée par P_i. Dans ce cas, toutes les variables définies dans P_i sont initialisées avec les valeurs correspondantes extraites de v (les filtres $P_0, \ldots, P_n$ doivent définir les mêmes variables).

Remarque 2-9
Comme pour les filtres (voir la section 2.5.4), chaque fois qu'une offre O est appliquée sur une valeur v, les variables définies dans O ne seront utilisées dans le programme XTL que si v est correctement filtrée par O. ■

2.8.2 Filtres d'actions

Un *filtre d'action* est une construction permettant d'exprimer un prédicat sur le contenu d'une action $a \in A$ du STE. Les filtres d'actions XTL, qui généralisent les constructions similaires de la logique temporelle R$\star$ICO [Gar89a, pages 181–183], ont la syntaxe suivante :

$$(G_0|O_0)\ O_1 \ldots O_m\ [\mathbf{\ldots}]\ O_{m+1} \ldots O_{m+n}\ [\textbf{where } E]$$

où G_0 représente une porte et l'expression optionnelle E doit être de type `boolean`. La porte G_0 est une abréviation pour l'offre "**!** G_0" ; nous l'avons introduite comme alternative à l'offre O_0 afin de permettre une écriture plus aisée des filtres d'actions pour la vérification de programmes LOTOS, dont les actions ont la forme "$G\ v_1 \ldots\ v_p$". Les offres $O_0, O_1, \ldots, O_{m+n}$ sont destinées à filtrer les champs de l'action a. Pour tout $0 \leq i, j \leq m + n$, les variables définies dans O_i ne sont pas visibles dans O_j ; de surcroît, elles doivent être différentes de toutes les variables définies dans O_j. Toutes les variables contenues dans les offres $O_0, \ldots, O_{m+n}$ sont visibles dans l'expression booléenne E, qui permet (optionnellement) d'exprimer une condition supplémentaire sur les valeurs de ces variables. La construction optionnelle "**...**" permet de filtrer un nombre arbitraire de champs de a ayant n'importe quel type.

Une action $a = v_0\ \ldots\ v_p$ du STE satisfait un filtre d'action ssi les conditions suivantes sont vérifiées :

a) $p = m + n$ si la construction "**...**" est absente et $p \geq m + n$ sinon ;

b) v_0 est égale à G_0 ou est filtrée par O_0 ; pour tout $1 \leq i \leq m$, v_i est filtrée par O_i, et pour tout $1 \leq j \leq n$, v_{p-n+j} est filtrée par O_{m+j} ;

c) l'expression booléenne E, si elle est présente, est évaluée à vrai dans le contexte des variables initialisées par les offres $O_0, \ldots, O_{m+n}$.

Remarque 2-10
Afin d'obtenir une sémantique dynamique correcte, lorsqu'un filtre d'action est évalué sur une action $a \in A$, les conditions a), b) et c) ci-dessus doivent être obligatoirement vérifiées dans cet ordre et l'évaluation doit être interrompue (dans ce cas, l'action a ne satisfaisant pas le filtre) dès qu'une d'entre elles est fausse. Ceci assure le fait que toutes les offres sont appliquées sur des valeurs bien définies et que toutes les variables utilisées dans l'expression "**where**" sont initialisées.

Par contre, puisque les variables contenues dans une offre O_i ne sont visibles dans aucune des autres offres O_j, l'application des offres $O_0, \ldots, O_{m+n}$ sur les champs de a peut être effectuée dans n'importe quel ordre. ■

Si a satisfait le filtre d'action, toutes les variables contenues dans les offres $O_0, \ldots, O_{m+n}$ sont initialisées et peuvent être utilisées à l'extérieur du filtre d'action, suivant les règles de visibilité qui seront définies aux sections 2.8.4, 2.9 et 2.10.4. Dans le cas contraire, aucune variable n'est initialisée.

Remarque 2-11
La sémantique du langage XTL assure le fait que les variables contenues dans un filtre d'action évalué sur une action $a \in A$ ne seront utilisées par la suite que si a satisfait le filtre d'action, de telle sorte que toutes les variables seront initialisées avant d'être utilisées. ■

Nous donnons ci-dessous quelques exemples d'utilisation des filtres d'actions ; d'autres pourront être trouvés dans la suite du document.

Exemple 2-35
Différents types d'offres (avec expressions ou avec variables) peuvent être combinées dans le même filtre d'action. Ainsi, l'accès d'un processus `p` à une ressource `r` en mode `Write` sur la porte `ACCESS`, avec écriture d'une donnée quelconque, peut être exprimé par le filtre d'action suivant :

```
ACCESS ! Write ? r : Resource ? p : Pid any
```

Cette formule est satisfaite par toutes les actions du STE ayant la forme "`ACCESS Write` v_1 v_2 v_3", où `Write` est une constante et v_1, v_2 sont respectivement des valeurs de type `Resource` et `Pid` (v_3 est une valeur d'un type quelconque). Les valeurs v_1 et v_2 pourront être utilisées à l'extérieur du filtre d'action, car elles sont affectées aux variables `r` et `p` ; par contre, la valeur v_3 ne sera pas visible, car elle est filtrée par une offre générique "**any**". ■

Exemple 2-36
L'expression optionnelle "**where**" permet de préciser des conditions supplémentaires sur les champs d'une action. Ainsi, l'émission sur une porte `SEND` d'un nombre entier supérieur à 3 peut être exprimée par le filtre d'action suivant :

```
SEND ? n : integer where n >= 3
```

Cette formule est satisfaite par toutes les actions du STE ayant la forme "`SEND` n" avec $n \geq 3$. ■

Exemple 2-37
La construction optionnelle "`...`" permet d'ignorer une portion de taille quelconque d'une action. Ainsi, la réception d'un message (dont on ignore le contenu) ayant l'adresse de l'expéditeur différente de celle du destinataire peut être exprimée par le filtre d'action suivant :

```
RECV ? src : Addr ? dest : Addr ... where src <> dest
```

Cette formule est satisfaite par toutes les actions du STE ayant la forme "`RECV` v_1 v_2 $\ldots$ v_p", où $p \geq 2$ et $v_1 \neq v_2$. ■

2.8.3 Méta-opérateur "current" sur actions

De la même façon que les langages orientés-objet permettent à une méthode d'accéder à l'objet courant auquel elle est attachée, il est possible d'accéder, dans une formule α, à l'action a du STE sur laquelle α est couramment évaluée. Ceci est réalisé grâce au méta-opérateur "**current**" sur actions, qui dénote une fonction nullaire de type `label`. Ce méta-opérateur peut être utilisé comme une fonction usuelle dans les expressions de valeur. Il obéit aux mêmes règles de visibilité que les variables exportées par les offres O_i contenues dans le filtre d'action respectif.

Chaque fois qu'un filtre d'action est évalué sur une action a du STE, une définition du méta-opérateur "**current**" est produite implicitement, de telle sorte qu'un appel de "**current**" renvoie la valeur a.

Ce méta-opérateur permet d'exprimer certains prédicats sur actions nécessitant l'utilisation de fonctions prédéfinies de type `label`.

Exemple 2-38
La fonction prédéfinie `visible : label -> boolean` permet de tester si une action du STE est visible (c'est-à-dire, différente de l'action invisible, notée τ en CCS et **i** en LOTOS). Le prédicat caractérisant l'action invisible peut être exprimé à l'aide du méta-opérateur "**current**" sur actions comme suit :

```
any where not visible (current)
```

L'offre générique "**any**" filtre toutes les actions ayant un seul champ ; parmi celles-ci, l'expression booléenne "**where**" filtre celles qui sont invisibles. ■

D'autres exemples d'utilisation de ce méta-opérateur pourront être trouvés dans la suite du document.

2.8.4 Opérateurs booléens

Des propriétés plus complexes sur les actions du STE peuvent être exprimées en combinant les filtres d'actions avec les opérateurs booléens classiques "**true**", "**false**", "**not**", "**or**", "**and**", "**implies**", "**iff**" et "**xor**". Tous les opérateurs binaires sont infixés et associatifs à droite. Les opérateurs ont les priorités usuelles : l'opérateur "**not**" est le plus prioritaire, suivi de "**and**", puis de "**or**" et "**xor**", puis de "**implies**", puis de "**iff**".

La sémantique de ces opérateurs est définie de manière habituelle : une action a satisfait la formule **not** α ssi elle ne satisfait pas α ; elle satisfait α_1 **or** α_2 ssi elle satisfait α_1 ou elle satisfait α_2, etc. Nous considérons "**not**" et "**or**" comme opérateurs booléens primitifs, les autres opérateurs ("**and**", "**implies**", "**iff**" et "**xor**") étant définis comme opérateurs dérivés (voir la section 3.5.2). L'opérateur "**true**" (qui est satisfait par toutes les actions a du STE) est équivalent au filtre d'action "**any ...**".

Les combinaisons booléennes de filtres d'actions permettent d'exprimer des propriétés utiles sur les actions du STE.

Exemple 2-39
La formule ci-dessous caractérise les actions d'émission ou de réception d'un message (valeur de type `Msg`, défini dans le programme à vérifier et importé du fichier BCG) :

```
(SEND ? any Msg) or (RECV ? any Msg)
```

De façon similaire, la formule suivante décrit les actions qui ne sont pas des émissions de messages :

```
not (SEND ? any Msg)
```

Il convient de remarquer que la présence des offres (même sans variables) dans les filtres d'actions permet une description plus concise des propriétés que les logiques classiques basées sur actions : par exemple, pour exprimer le prédicat `SEND ? any Msg` en ACTL (qui ne dispose pas de mécanismes de filtrage des actions), il aurait fallu une disjonction contenant autant de formules `SEND ! m` que d'émissions de messages `m` dans le STE. ■

De la même manière qu'un filtre d'action, une formule booléenne α peut exporter des variables simples, initialisées avec des valeurs extraites de l'action a sur laquelle α est couramment évaluée. Les règles de propagation des variables exportées à travers les opérateurs booléens seront définies formellement à la section 3.5.1. Intuitivement, une formule "α_1 **or** α_2" (*resp.* "α_1 **and** α_2") exporte les variables qui sont initialisées par α_1 et (*resp.* ou) par α_2.

La sémantique dénotationnelle des formules sur actions (voir la section 3.5) garantit que, chaque fois qu'une formule α est satisfaite par une action a du STE, toutes les variables exportées par α seront initialisées (donc utilisables de façon sûre à l'extérieur de α) avec des valeurs extraites de a.

Exemple 2-40
La formule suivante caractérise les actions qui peuvent être soit des envois de messages, soit des réceptions correctes de messages :

```
(SEND ? m : Msg)
or
(RECV ? m : Msg ? crc : integer where is_ok (m, crc))
```

Le prédicat `is_ok` est supposé défini dans le programme à vérifier. La variable `m` est visible à l'extérieur de la formule, étant exportée par les deux arguments de l'opérateur "**or**" ; en revanche, `crc` n'est visible que dans l'expression "**where**" du deuxième filtre d'action. ■

Outre les variables simples qu'elle peut exporter lorsqu'elle est évaluée sur une action a du STE, une formule α exporte aussi la valeur de a, qui pourra être utilisée à l'extérieur de α à l'aide du méta-opérateur "**current**" sur actions. D'autres exemples d'utilisation des formules sur actions peuvent être trouvés aux sections suivantes.

2.9 Expressions régulières

Les formules sur actions ne permettent d'exprimer que des propriétés portant sur les actions individuelles du programme (séquences d'actions de longueur 1). Toutefois, l'expérience montre qu'en pratique il est utile de pouvoir caractériser des séquences d'actions (de longueur quelconque) contenues dans le STE. Ceci est réalisé en XTL au moyen d'*expressions régulières* (symbole non-terminal R) construites sur le vocabulaire des formules sur actions α. Les expressions régulières XTL, qui généralisent les constructions similaires de la logique PDL, ont la syntaxe suivante :

$$
\begin{array}{rcl}
R & ::= & \alpha \\
& | & R_1 \,.\, R_2 \\
& | & R_1 \mid R_2 \\
& | & R^* \\
& | & R^+
\end{array}
$$

où "." est l'opérateur de concaténation de séquences, "|" est l'opérateur de choix et "*" (*resp.* "+") dénote la répétition d'une séquence zéro (*resp.* une) ou plusieurs fois. Les opérateurs binaires "." et "|" sont associatifs à gauche, "|" étant moins prioritaire que ".", qui à son tour est moins prioritaire que "*" et "+".

La sémantique des expressions régulières est définie de façon habituelle sur des séquences d'actions du STE : une séquence $s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_n} s_n$ satisfait R ssi le mot constitué des actions $a_1 \ldots a_n$ appartient au langage régulier défini par R. Plus précisément :

- une séquence $s_1 \xrightarrow{a_1} s_2$ satisfait α ssi a_1 satisfait α ;
- une séquence $s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_{n-1}} s_n$ satisfait "$R_1 \,.\, R_2$" ssi il existe un état $k \in [1, n]$ tel que $s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} s_k$ satisfait R_1 et $s_k \xrightarrow{a_k} \cdots \xrightarrow{a_{n-1}} s_n$ satisfait R_2 ;
- une séquence $s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_{n-1}} s_n$ satisfait "$R_1 \mid R_2$" ssi elle satisfait R_1 ou elle satisfait R_2 ;
- une séquence $s_1 \xrightarrow{a_1} s_2 \cdots \xrightarrow{a_{n-1}} s_n$ satisfait "R^*" (*resp.* "R^+") ssi elle représente la concaténation de $p \geq 0$ (*resp.* $p > 0$) sous-séquences, chacune satisfaisant R.

Exemple 2-41
L'expression régulière suivante décrit une séquence de quatre actions `REQUEST`, `INDICATION`, `RESPONSE` et `CONFIRM` (modélisant l'ouverture d'une connexion OSI), séparées par des séquences (éventuellement vides) d'actions invisibles `i` :

```
REQUEST . i* . INDICATION . i* . RESPONSE . i* . CONFIRM
```

Les priorités associées aux opérateurs "*" et "." permettent d'éviter l'utilisation de parenthèses. ■

Contrairement aux formules sur actions, qui peuvent être utilisées dans les expressions et les formules "**case action**" (voir les sections 2.5.6 et 2.10.11) ainsi que dans les méta-opérateurs d'évaluation de formules (voir la section 2.12), les expressions régulières ne peuvent être employées que dans les opérateurs modaux (voir la section 2.10.4). Par ailleurs, les expressions régulières ne sont pas des constructions primitives XTL : les formules modales contenant des R peuvent être traduites en termes d'opérateurs de point fixe et de modalités contenant des formules α (voir la section 3.8).

De la même manière qu'une formule sur actions, une expression régulière R peut définir et exporter des variables, initialisées avec des valeurs extraites des actions contenues dans la séquence sur laquelle R est couramment évaluée. En outre, la dernière action de la séquence (si elle peut être déterminée) est aussi exportée, pouvant être utilisée à l'extérieur de R au moyen du méta-opérateur "**current**" sur actions.

Les règles de propagation des variables et des actions exportées à travers les expressions régulières seront définies formellement à la section 3.6.1. Intuitivement, une expression régulière "R_1 **.** R_2" exporte toutes les variables exportées par R_1 et par R_2 ; une expression régulière "$R_1 \mid R_2$" exporte les variables simultanément exportées par R_1 et par R_2 ; une expression régulière "R^{+}" exporte les variables exportées par R. Les expressions régulières "R^{*}" n'exportent pas de variables, puisqu'elles peuvent être satisfaites par des séquences vides d'actions du STE. En outre, pour faciliter l'utilisation des variables dans les propriétés portant sur les séquences d'actions, dans toute expression régulière de la forme "R_1 **.** R_2", les variables exportées par R_1 sont visibles dans R_2.

Ces règles de propagation et de visibilité peuvent paraître compliquées, mais elles constituent une extension naturelle de la sémantique des expressions régulières utilisées dans la logique PDL.

Exemple 2-42
L'expression régulière suivante caractérise les séquences contenant l'émission d'un message `m` suivie, après un nombre quelconque d'actions invisibles `i`, de la réception du même message :

```
(SEND ? m : Msg) . i* . (RECV ! m)
```

La variable `m` exportée par le filtre d'action `SEND ? m : Msg` est visible dans les expressions régulières qui lui sont concaténées à l'aide d'opérateurs ".". L'expression régulière ci-dessus exporte la variable `m`, ainsi que l'action filtrée par `RECV ! m` (qui pourra être utilisée à l'extérieur à l'aide du méta-opérateur "**current**" sur actions). ■

Remarque 2-12
La sémantique des formules modales (voir la section 3.7.2) est définie de manière à assurer que chaque fois qu'une séquence d'actions du STE satisfait une expression régulière R, toutes les variables exportées par R (celles-ci pouvant contenir, si tel est le cas, la dernière action de la séquence, désignable par "**current**") seront initialisées. ■

D'autres exemples d'utilisation des expressions régulières seront présentés aux sections 2.10.4 et 2.10.5.

2.10 Formules sur états

Les propriétés temporelles sur les états du STE peuvent être exprimées en XTL au moyen de *formules sur états* (symbole non-terminal φ). La sémantique d'une formule φ est définie en terme des états du STE qui la satisfont. Le terme "*évaluation d'une formule φ sur un état s*" désigne le fait de déterminer si s satisfait φ ou non ; en revanche, "*évaluation d'une formule φ (sur un STE)*" signifie le calcul de l'ensemble des états du STE qui satisfont φ. Le sous-langage des formules sur états est une extension du μ-calcul modal avec des constructions permettant la manipulation de valeurs typées. Ainsi, les formules φ sont construites à partir de prédicats de base, opérateurs booléens, opérateurs modaux contenant des expressions régulières, opérateurs de point fixe paramétrés par des variables typées, quantificateurs et opérateurs inspirés des langages de programmation, comme "**let**", "**if**" et "**case**".

Les formules φ constituent une catégorie sémantique différente des expressions E. Néanmoins, pour des raisons de simplicité syntaxique, les expressions E de type `boolean` sont admises comme un cas particulier de formules φ, ce qui fait que le langage généré par le symbole non-terminal E est inclus dans celui généré par φ (la grammaire abstraite présentée à la section 2.1.2 contient effectivement une production $\varphi ::= E$).

Les sections suivantes présentent chaque construction contenue dans les formules sur états.

2.10.1 Prédicats de base

On appelle *prédicat de base* une formule constituée d'une expression E de type `boolean`. Les prédicats de base permettent de caractériser individuellement les états du STE : du fait qu'ils ne contiennent pas d'opérateurs modaux ou de point fixe, leur évaluation ne nécessite pas la connaissance d'autres états du STE ni de la relation de transition. Ils peuvent utiliser des variables XTL (définies dans des expressions, des formules sur actions ou des formules sur états) ainsi que des variables BCG contenues dans les états sur lesquels ils sont évalués.

Il existe des prédicats de base dont l'évaluation ne dépend pas d'un état considéré : c'est le cas du prédicat "**true**" (qui est satisfait par tous les états) et du prédicat "**false**" (qui n'est satisfait par aucun état).

2.10.2 Méta-opérateur "current" sur états

XTL offre la possibilité d'accéder, dans une formule φ, à l'état s sur lequel φ est couramment évaluée. Ceci est réalisé grâce au méta-opérateur "**current**" sur états, qui dénote une fonction nullaire de type `state`.

Chaque fois qu'une formule φ est évaluée sur un état s du STE, une définition de l'opérateur "**current**" est produite implicitement, de telle sorte qu'un appel de "**current**" dans φ renvoie la valeur s.

L'opérateur "**current**" sur états offre un moyen sémantique propre pour exprimer des propriétés sur les états du modèle STE. Il est donc particulièrement adapté pour la vérification de programmes écrits en des langages dont le modèle sous-jacent contient de l'information pertinente dans les états (comme c'est le cas d'ESTELLE ou de SDL).

Remarque 2-13
L'opérateur "**current**" est surchargé : celui sur actions (section 2.8.3) est une fonction nullaire de type `label`, tandis que celui sur états est une fonction nullaire de type `state`. Si le contexte d'utilisation d'un opérateur "**current**" ne permet pas de déterminer son type, l'utilisateur doit le préciser explicitement à l'aide de la notation "**of**" (par exemple, "**current of** `state`"). ■

L'opérateur "**current**", utilisé dans une formule φ conjointement avec la notation pointée "**.**", permet d'accéder aux variables BCG contenues dans l'état courant s sur lequel la formule φ est évaluée. Ceci permet, en particulier, d'exprimer des prédicats de base sur les états s du STE en utilisant les valeurs des variables contenues dans s.

Exemple 2-43
Le prédicat de base suivant est satisfait par tous les états du STE dans lesquels la variable `x` est inférieure ou égale à `y` :

```
current.x <= current.y
```

Dans la formule ci-dessus, `x` et `y` sont des variables BCG (voir la section 2.4) ; les éventuels conflits de nom avec d'autres variables XTL visibles dans la formule peuvent être résolus en désignant les variables BCG à l'aide de la notation externe (`current.‘x‘ <= current.‘y‘`). ■

Dans un programme XTL, l'opérateur "**current**" sur états ne peut être utilisé que dans les formules φ. Une occurrence d'un opérateur "**current**" dans une formule φ évaluée sur un état s est visible dans toutes les sous-formules de φ qui s'évaluent aussi sur s.

D'autres applications de l'opérateur "**current**" sur états seront présentées à la section 2.10.6 et dans l'annexe C.4.

2.10.3 Opérateurs booléens

Les formules sur états φ peuvent être combinées à l'aide des opérateurs booléens classiques "**not**", "**or**", "**and**", "**implies**", "**iff**" et "**xor**". Les associativités et les priorités de ces opérateurs sont les mêmes que celles des opérateurs correspondants sur les formules α (voir la section 2.8.4). Tous les opérateurs booléens binaires sont infixés et sont moins prioritaires que les opérateurs binaires infixés prédéfinis (voir la section 2.3) ou définis par l'utilisateur (voir la section 2.7). Ces conventions permettent de réduire la quantité de parenthèses utilisées dans les formules booléennes sur états.

La sémantique de ces opérateurs est définie de manière usuelle : un état s satisfait la formule **not** φ ssi il ne satisfait pas φ ; il satisfait φ_1 **or** φ_2 ssi il satisfait φ_1 ou il satisfait φ_2, etc. Nous considérons "**not**" et "**or**" comme opérateurs booléens de base. Les autres opérateurs ("**and**", "**implies**", "**iff**" et "**xor**") peuvent être définis en termes des opérateurs de base (voir la section 3.7.2).

Exemple 2-44
Soit un programme parallèle implémentant le problème classique des lecteurs et des rédacteurs, et supposons que le nombre de lecteurs et de rédacteurs couramment actifs sont mémorisés dans deux variables d'état `readers` et `writers`. La formule suivante caractérise l'exclusion mutuelle des lecteurs et des rédacteurs concernant l'accès à la bibliothèque :

```
current.writers <= 1 and (current.writers = 1 implies current.readers = 0)
```

Cette formule est satisfaite par tous les états du STE dans lesquels il existe au plus un rédacteur actif et, dans ce cas, tous les lecteurs sont passifs. ■

Tous les opérateurs booléens ci-dessus sont surchargés : ils peuvent dénoter des opérateurs sur les formules φ aussi bien que des fonctions XTL prédéfinies de type `boolean`. L'ambiguïté sur l'appel d'un opérateur *op* dans une formule φ est résolue selon le principe suivant : si tous les arguments de *op* sont des prédicats de base, alors *op* est interprété comme une fonction de type `boolean` et son appel comme un prédicat de base ; dans le cas contraire, *op* est interprété comme un opérateur sur formules φ. Intuitivement, cette règle permet d'obtenir les prédicats de base "maximaux" (au sens de l'inclusion syntaxique des formules) apparaissant dans une formule φ, ce qui assure une interprétation "canonique" des formules sur états.

Exemple 2-45
La formule suivante est interprétée comme un prédicat de base, "**and**" dénotant la fonction prédéfinie respective de type `boolean` :

```
current.x < current.y and current.y < current.z
```

Une autre interprétation possible (mais non conforme à la règle énoncée plus haut) serait de considérer `current.x < current.y` et `current.y < current.z` comme des formules sur états et "**and**" comme un opérateur booléen sur formules φ. ■

Suivant la règle donnée ci-dessus, toutes les occurrences des opérateurs nullaires "**true**" et "**false**" dans une formule φ sont interprétées comme des appels des fonctions respectives de type `boolean` (dénotant des prédicats de base). Ceci constitue une différence par rapport aux logiques temporelles classiques, dont la sémantique habituelle définit les prédicats de base en termes de propositions atomiques et non en termes d'expressions booléennes sur valeurs.

2.10.4 Opérateurs modaux

Les prédicats de base, construits à l'aide de l'opérateur "**current**" sur états et des opérateurs booléens, permettent d'exprimer des propriétés portant sur des états individuels du STE. Pour décrire des propriétés prenant en compte les transitions entre les états, XTL offre des opérateurs modaux, qui généralisent les opérateurs similaires de la logique PDL avec des mécanismes de définition de variables. Ces opérateurs ont la syntaxe suivante :

$$\langle R \rangle\ \varphi$$

$$[R]\ \varphi$$

où R est une expression régulière (voir la section 2.9), "$\langle\ \rangle$" dénote l'opérateur de possibilité (*losange*) et "$[\ ]$" dénote l'opérateur de nécessité (*carré*). Toutes les variables simples exportées par R sont visibles dans φ. Du point de vue syntaxique, les opérateurs modaux sont considérés comme des opérateurs unaires préfixés sur les formules φ, ayant la même priorité que l'opérateur booléen "**not**".

Un état s du STE satisfait la formule $\langle R \rangle\ \varphi$ ssi il existe un chemin $s \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n$ satisfaisant l'expression régulière R et dont l'état but s_n satisfait la formule φ. L'opérateur de nécessité "$[\ ]$" est défini comme le dual de "$\langle\ \rangle$" : un état s satisfait $[R]\ \varphi$ ssi tous les chemins $s \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n$ satisfaisant R mènent à des états s_n satisfaisant φ.

Les opérateurs modaux de XTL permettent d'obtenir comme cas particuliers les modalités de la logique HML, en considérant des expressions régulières R réduites à des formules sur actions α.

Exemple 2-46
Reprenant l'exemple 1-8, les états de blocage du STE, c'est-à-dire les états n'ayant pas de successeurs, peuvent être caractérisés au moyen de la formule modale `[ true ] false`. ■

L'utilisation des variables simples x définies dans les formules sur actions α permet d'exprimer des propriétés modales prenant en compte les valeurs contenues dans les étiquettes du STE.

Exemple 2-47
La propriété ci-dessous est inspirée de la spécification d'une unité de contrôle du son d'un poste de télévision [ABC94, pages 78–95]. Le volume courant du son, mémorisé dans une variable `V`, peut être modifié au moyen d'actions "`WRITE_V` v", où v est la nouvelle valeur du volume. Le fait que (suite à une action "`WRITE_V` v") le volume du son ne dépasse jamais une valeur maximale `max`, peut être caractérisé en XTL par la formule modale suivante :

```
[ WRITE_V ? x : Nat ] (x <= max)
```

Le type `Nat`, l'opérateur `<=`, ainsi que la constante `max` sont supposés définis dans le programme parallèle à vérifier. Cette propriété peut être décrite aussi par la formule équivalente suivante :

```
[ WRITE_V ? x : Nat where x > max ] false
```

exprimant, de manière duale, le fait qu'il n'existe aucune action "`WRITE_V` v" contenant une valeur v strictement supérieure à `max`. Bien entendu, pour une vérification exhaustive il faut s'assurer que tous les états du STE satisfont la formule ci-dessus ; pour cela, XTL offre des opérateurs spéciaux, présentés à la section 2.11. ∎

Les modalités de la logique HML ne permettent de décrire que des propriétés temporelles portant sur un voisinage borné des états du STE (en termes de la relation de transition). Par contre, les opérateurs modaux de XTL permettent, grâce aux expressions régulières, d'exprimer de manière naturelle des propriétés temporelles faisant intervenir des chemins de longueur arbitraire dans le STE.

Exemple 2-48
Le fait que l'émission d'un message (action `SEND m`) est éventuellement suivie de la réception du même message (action `RECV m`) peut être exprimé à l'aide de la formule suivante :

```
[ SEND ? m : Msg ] < (not (? SEND | RECV any))* . (RECV ! m) > true
```

où le type `Msg` est supposé défini dans le programme parallèle à vérifier. Un état s_1 du STE satisfait la formule ci-dessus ssi, pour toutes les transitions $s_1 \xrightarrow{\texttt{SEND m}} s_2$, il existe un chemin $s_2 \xrightarrow{a_2} s_3 \cdots \xrightarrow{a_{n-1}} s_n \xrightarrow{\texttt{RECV m}} s_{n+1}$. L'offre `? SEND | RECV` (où `SEND` et `RECV` sont des constructeurs de type `gate`) permet de caractériser de façon concise les actions qui ne sont pas des émissions ou des réceptions. Le mot $a_2 \ldots a_{n-1}$ appartient au langage défini par l'expression régulière `(not (? SEND | RECV any))*`, ce qui assure qu'aucun autre message n'a été émis ou reçu jusqu'à la réception de `m`.

Il convient de remarquer que, pour exprimer la propriété ci-dessus en PDL (qui ne dispose pas de mécanismes de filtrage des actions), il aurait fallu écrire autant de formules que de messages `m` contenus dans les actions `SEND` du STE. ∎

D'autres exemples d'utilisation des opérateurs modaux XTL peuvent être trouvés au chapitre 5.

Les opérateurs modaux contenant des expressions régulières ("$\langle R \rangle\ \varphi$" et "$[R]\ \varphi$") ne sont pas des constructions primitives XTL : ils peuvent être traduits en termes d'opérateurs de point fixe (voir la section 2.10.6) et de modalités "$\langle \alpha \rangle\ \varphi$" et "$[\alpha]\ \varphi$" de HML (voir la section 3.8).

2.10.5 Opérateur "@"

Il est parfois utile de pouvoir décrire des propriétés concernant la répétition infinie de certaines actions du modèle STE. Pour cela, on dispose en XTL de l'opérateur "**@**", ayant une sémantique similaire à l'opérateur de bouclage infini de la logique PDL-Δ. Cet opérateur a la syntaxe suivante :

$$\textbf{@}\ (R)$$

où R est une expression régulière. Les variables simples éventuellement exportées par R ne sont pas visibles à l'extérieur de la formule "**@**".

Un état s_1 du STE satisfait "**@** (R)" ssi il existe un chemin infini $s_1 \xrightarrow{a_1^0} s_1^0 \cdots \xrightarrow{a_1^{n_1}} s_2 \xrightarrow{a_2^0} s_2^0 \cdots \xrightarrow{a_2^{n_2}} s_3 \cdots$ où toutes les séquences $s_i \xrightarrow{a_i^0} s_i^0 \cdots \xrightarrow{a_i^{n_i}} s_{i+1}$ (pour $i \geq 1$) satisfont R. Bien entendu, étant donné que le modèle STE est fini, tous les chemins infinis doivent obligatoirement aboutir à un circuit, ce qui fait que le chemin ci-dessus contient forcément deux états s_i et s_k (avec $i < k$) identiques.

Exemple 2-49
Les états de *divergence* d'un STE sont les états à partir desquels il existe un chemin infini contenant uniquement des actions invisibles (notées `i` en LOTOS). Ils peuvent être caractérisés par la formule suivante :

```
@ (i)
```

Les états de *famine* (*livelock*) d'un STE sont les états contenus dans des `i`-circuits du STE ; puisque chaque séquence infinie de `i`-actions conduit forcément à un `i`-circuit, l'absence de famine peut être exprimée par la formule ci-dessous :

```
not @ (i)
```

qui interdit la présence d'états de divergence (et, par conséquent, la présence d'états de famine). ■

L'opérateur "@" permet aussi d'exprimer l'atteignabilité inévitable de certains états ou de certaines actions du STE.

Exemple 2-50
Le fait qu'après chaque action `QUIT` la terminaison du programme soit inévitable (c'est-à-dire que toutes les séquences aboutissent, après un nombre fini d'actions quelconques, à un état puits) peut être exprimé par la formule XTL suivante :

```
[ QUIT ] not @ (true)
```

qui spécifie qu'après une action `QUIT`, il n'existe pas de séquence infinie contenue dans le STE. ■

Exemple 2-51
Le fait que toute action `SEND` soit inévitablement suivie d'une action `RECV` peut être exprimé par la formule XTL suivante :

```
[ SEND ] (
    not @ (not RECV)
    and
    [ (not RECV)* ] < true > true
)
```

qui spécifie qu'après une action `SEND`, il n'existe pas de séquence infinie ne contenant pas d'action `RECV` et qu'il est impossible d'atteindre un blocage (état puits) avant d'avoir effectué un `RECV`. ■

Les propriétés illustrées dans les exemples 2-50 et 2-51 ci-dessus peuvent être exprimées de manière similaire en utilisant la logique R$\stackrel{\star}{\text{I}}$CO [Gar89a, pages 191–192], qui contient elle aussi un opérateur de bouclage "@".

Utilisé conjointement avec les opérateurs modaux de XTL, l'opérateur "@" permet de caractériser des séquences d'actions appartenant à toute la classe des langages ω-réguliers [Cho74]. En effet, sachant que tout langage ω-régulier U peut être caractérisé par une expression ω-régulière de la forme $L_1.L_2^\omega$ (où L_1 et L_2 sont des langages réguliers), la formule suivante exprime l'existence d'une séquence d'actions satisfaisant U :

$$\boxed{\langle R_1 \rangle \ \texttt{@}\ (R_2)}$$

où R_1 et R_2 sont les expressions régulières correspondant respectivement aux langages L_1 et L_2.

Enfin, signalons que l'opérateur "@" n'est pas primitif : il peut être traduit en termes d'opérateurs modaux et de point fixe (voir la section 3.7.2).

2.10.6 Opérateurs de point fixe

Les propriétés temporelles peuvent être exprimées en XTL à l'aide d'opérateurs de point fixe, qui généralisent les opérateurs similaires du μ-calcul modal. Ces opérateurs ont la syntaxe suivante :

$$\textbf{mu}\ Y\ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi$$

$$\textbf{nu}\ Y\ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi$$

où "**mu**" (*resp.* "**nu**") dénote l'opérateur de plus petit (*resp.* plus grand) point fixe, Y est une *variable propositionnelle*[10] définie par l'opérateur de point fixe respectif, $x_1, \ldots, x_n$ ($n \geq 0$) sont les paramètres, $E_1, \ldots, E_n$ sont les arguments et φ est le corps de Y. La variable Y, ainsi que les paramètres $x_1, \ldots, x_n$, sont visibles dans le corps φ. Les occurrences d'utilisation (ou *appels*) de Y ont la même syntaxe que les appels de fonctions :

$$Y\ (E'_1, \ldots, E'_n)$$

où les expressions $E'_1, \ldots, E'_n$ doivent être compatibles (en nombre et types) avec les paramètres $x_1, \ldots, x_n$ de la définition de Y. La visibilité des variables Y dans les formules de point fixe imbriquées suit les règles habituelles : une variable Y définie dans un opérateur de point fixe "masque" les autres variables Y éventuellement définies dans des opérateurs de point fixe englobants. Les variables Y peuvent être surchargées, les éventuelles ambiguïtés étant résolues suivant le nombre et les types des paramètres. Du point de vue syntaxique, les formules "**mu**" et "**nu**" sont des opérateurs unaires, ayant la même priorité que l'opérateur "**not**" ou les opérateurs modaux. Par la suite, nous utiliserons le symbole σ pour dénoter (indifféremment) "**mu**" ou "**nu**".

Afin d'assurer une sémantique correcte des opérateurs de point fixe (voir la section 3.7.2), les formules φ doivent satisfaire la condition de *monotonie syntaxique* [Koz83] : pour chaque formule "$\sigma Y\ (\ldots)\ .\ \varphi$", toutes les occurrences de Y dans φ doivent être placées sous un nombre pair de négations ou de parties gauches d'implications.

Une variable propositionnelle Y définie dans un opérateur "$\textbf{mu}\ Y\ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi$" (*resp.* "$\textbf{nu}\ Y\ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi$") dénote une formule sur états paramétrée par les variables simples $x_1, \ldots, x_n$, définie comme la plus petite (*resp.* la plus grande) solution de l'équation de point fixe suivante :

$$Y\ (x_1{:}T_1, \ldots, x_n{:}T_n) = \varphi$$

Un état s satisfait une formule "$\sigma Y\ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi$" ssi il satisfait la variable propositionnelle Y avec les valeurs de $E_1, \ldots, E_n$ substituées aux paramètres $x_1, \ldots, x_n$. Les formules de point fixe dénotent les définitions et appels *in situ* des variables Y correspondantes ; de ce point de vue, elles sont similaires aux expressions d'itération "**loop**" (voir la section 2.6.1). Les opérateurs "**mu**" et "**nu**" sont duaux : ils peuvent s'exprimer l'un en fonction de l'autre (voir la section 3.7.2).

Intuitivement, les opérateurs "**mu**" et "**nu**", combinés avec des formules modales, permettent de décrire respectivement des arborescences finies et infinies dans le STE.

Exemple 2-52
Une propriété de base des protocoles de communication est la transmission correcte d'un message `m` entre une entité émettrice et une entité réceptrice du protocole. La formule suivante exprime le fait qu'un message `m` émis sur la porte `SEND` sera inévitablement reçu sur la porte `RECV` :

```
[ SEND ? m : Msg ] mu Y . (< true > true and [ not (RECV ! m) ] Y)
```

[10] Bien qu'en XTL les "variables" Y puissent être paramétrées par des variables simples x, nous employons le terme "variable propositionnelle" par souci de compatibilité avec la terminologie utilisée dans la littérature consacrée au μ-calcul standard [Koz83, EL86, Cle90].

Un état s_1 du STE satisfait la formule ci-dessus ssi toutes les transitions $s_1 \xrightarrow{\texttt{SEND m}} s_2$ mènent à des états s_2 à partir desquels tous les chemins ont des préfixes $s_2 \xrightarrow{a_3} s_3 \xrightarrow{a_4} \cdots s_{n-1} \xrightarrow{\texttt{RECV m}} s_n$. En μ-calcul standard, la propriété ci-dessus nécessiterait une formule différente pour chaque message `m` contenu dans les actions `SEND` du STE. ■

Le paramétrage des opérateurs de point fixe permet (entre autres) d'exprimer le fait qu'une certaine valeur contenue dans les états ou les étiquettes du STE suit une loi d'évolution précise durant l'exécution du programme.

Exemple 2-53
La formule suivante est satisfaite par les états s d'un STE tels que, sur toutes les séquences d'exécution issues de s, les actions `OUTPUT` successives contiennent des nombres naturels croissants :

```
nu Y (last_n : Nat := 0) . (
    [ OUTPUT ? n : Nat ] (n >= last_n and Y (n))
    and
    [ not (OUTPUT any) ] Y (last_n)
)
```

Le type `Nat`, ainsi que l'opérateur `>=`, sont supposés être définis dans le programme à vérifier. Le paramètre `last_n`, initialisé à `0`, mémorise le dernier nombre naturel émis sur la porte `OUTPUT`. ■

Il est intéressant de remarquer que, même pour les propriétés portant sur des STEs sans valeurs typées (comme c'est le cas des STEs générés à partir de programmes écrits en *basic* LOTOS), les opérateurs de point fixe paramétrés permettent une expression plus concise des formules que le μ-calcul standard.

Exemple 2-54
La propriété suivante, typique des protocoles de communication, caractérise les états du STE à partir desquels tous les chemins d'exécution contiennent une alternance stricte d'envois (actions `SEND`) et de réceptions de messages (actions `RECV`), commençant par un envoi :

```
nu Y (expect_send : boolean := true) . (
    [ SEND ] (expect_send and Y (false))
    and
    [ RECV ] (not expect_send and Y (true))
    and
    [ not (SEND or RECV) ] Y (expect_send)
)
```

Le paramètre booléen `expect_send` permet d'exprimer l'alternance des actions `SEND` et `RECV` : il est égal à `true` (*resp.* `false`) ssi les chemins issus de l'état courant doivent passer par une action `SEND` (*resp.* `RECV`) avant d'atteindre une action `RECV` (*resp.* `SEND`). La description de cette propriété en μ-calcul standard aurait nécessité deux opérateurs "**nu**" imbriqués. ■

Utilisés conjointement avec le méta-opérateur "**current**" sur états, les opérateurs XTL de point fixe permettent d'exprimer, en utilisant des formules d'alternance 1, des propriétés qui ne sont exprimables en μ-calcul standard qu'au moyen de formules d'alternance supérieure.

Exemple 2-55
Considérons un système de gestion d'une ressource partagée, où l'accès d'un processus à la ressource est accordé à l'aide d'une action `GRANT`. Un chemin infini dans le STE sous-jacent est dit *fortement inéquitable* par rapport à l'action `GRANT` ssi il ne contient pas de transition étiquetée par `GRANT`, tout en passant infiniment souvent par des états où l'action `GRANT` est exécutable. La propriété d'équité forte (*strong fairness*), c'est-à-dire l'absence de chemins fortement inéquitables par rapport à `GRANT`, peut être exprimée au moyen de la formule XTL suivante :

```
not mu Y1 (first_state : stateset := current) . (
        (< GRANT > true and
         < not GRANT >
            mu Y2 . ((current = first_state) or < not GRANT > Y2)
        ) or
        < not GRANT > Y1 (first_state)
    )
```

Lorsque cette formule est interprétée sur un état s, celui-ci est capté à l'aide du méta-opérateur "**current**", mémorisé dans le paramètre `first_state` de type `state` de la variable `Y1` et propagé à chaque appel de `Y1`. Les deux opérateurs "**mu**" imbriqués spécifient qu'à partir de s il existe un chemin non vide qui ne contient pas d'actions `GRANT`, passe par un état où l'action `GRANT` est exécutable, et aboutit à un état (dénoté par l'occurrence de "**current**" dans le corps de `Y2`) identique à `first_state`, autrement dit à s. Ceci caractérise les chemins non-équitables par rapport à `GRANT`, qui sont interdits par la formule ci-dessus.

En μ-calcul standard, cette propriété aurait été exprimée par la formule suivante, où la répétition infinie est décrite au moyen d'un opérateur "**nu**" supplémentaire :

```
not nu Y0 . (
        mu Y1 . (
            (< GRANT > true and
             < not GRANT > mu Y2 . (Y0 or < not GRANT > Y2)
            ) or
            < not GRANT > Y1 or
        )
    )
```

Cette formule est d'alternance 2, car la variable de plus grand point fixe `Y0` est appelée récursivement à travers les variables de plus petit point fixe `Y1` et `Y2`. La description des formules de μ-calcul d'alternance supérieure comme formules XTL d'alternance 1 permet de vérifier ces formules à la volée, au moyen de l'algorithme d'évaluation présenté à la section 4.2. ■

D'autres exemples d'utilisation des opérateurs de point fixe XTL peuvent être trouvés au chapitre 5 ainsi qu'à l'annexe C.

2.10.7 Quantificateurs

La manipulation de valeurs typées dans les formules φ rend nécessaire l'écriture de prédicats du premier ordre sur valeurs. Ces prédicats sont exprimables grâce aux quantificateurs, qui ont une syntaxe similaire aux quantificateurs utilisés dans les expressions E (voir la section 2.6.5) :

$$\textbf{exists } x_0\textbf{:}T_0 \text{ [}\textbf{among } E_0\text{]}, \ldots, x_n\textbf{:}T_n \text{ [}\textbf{among } E_n\text{]} \textbf{ in } \varphi$$

$$\textbf{forall } x_0\textbf{:}T_0 \text{ [}\textbf{among } E_0\text{]}, \ldots, x_n\textbf{:}T_n \text{ [}\textbf{among } E_n\text{]} \textbf{ in } \varphi$$

où les variables x_i sont visibles dans le corps φ du quantificateur respectif et où chaque expression optionnelle E_i ($0 \leq i \leq n$), appelée *sous-domaine* de la variable x_i, doit être de type ensemble d'éléments de T_i. Du point de vue syntaxique, les quantificateurs sont des opérateurs unaires préfixés, ayant la même priorité que la négation, les opérateurs modaux et les opérateurs de point fixe.

La sémantique des quantificateurs est évidente : un état s du STE satisfait une formule "**exists**" (*resp.* "**forall**") ssi, pour certaines valeurs (*resp.* toutes les valeurs) v_i (pour $0 \leq i \leq n$) dans les domaines "T_i [**among** E_i]", s satisfait φ dans le contexte des variables x_i initialisées avec v_i.

Remarque 2-14
Les domaines des variables quantifiées étant supposés finis, la formule "**exists**" (*resp.* "**forall**") peut être vue comme une disjonction (*resp.* une conjonction) généralisée :

$$\textbf{exists } x_0{:}T_0 \text{ [}\textbf{among } E_0\text{]}, \ldots, x_n{:}T_n \text{ [}\textbf{among } E_n\text{]} \textbf{ in } \varphi \stackrel{\mathrm{d}}{=} \bigvee_{x_0:T_0[\textbf{among } E_0],\ldots,x_n:T_n[\textbf{among } E_n]} \varphi$$

$$\textbf{forall } x_0{:}T_0 \text{ [}\textbf{among } E_0\text{]}, \ldots, x_n{:}T_n \text{ [}\textbf{among } E_n\text{]} \textbf{ in } \varphi \stackrel{\mathrm{d}}{=} \bigwedge_{x_0:T_0[\textbf{among } E_0],\ldots,x_n:T_n[\textbf{among } E_n]} \varphi$$

Cependant, l'utilisation des quantificateurs offre un gain considérable en concision, évitant à l'utilisateur les répétitions fastidieuses ou l'écriture de formules de longueur prohibitive. ■

Les quantificateurs permettent d'exprimer certaines classes de propriétés portant sur les valeurs contenues dans le STE (notamment, des propriétés de vivacité), qui ne sont pas directement exprimables à l'aide des opérateurs modaux étendus de XTL.

Exemple 2-56
Considérons un programme parallèle implémentant un protocole de communication entre plusieurs sites, identifiés par des éléments d'un type énuméré `Site`. La formule suivante exprime le fait qu'un message `m` diffusé au moyen d'une action `BROADCAST` sera inévitablement reçu par tous les sites :

```
[ BROADCAST ? m : Msg ]
    forall d : Site in
        mu Y . (< true > true and [ not (RECV ! d ! m) ] Y)
```

où la variable `d` de type `Site` sert à quantifier les sites destinataires. Un état s_1 du STE satisfait la formule ci-dessus ssi, pour toutes les transitions $s_1 \xrightarrow{\texttt{BROADCAST m}} s_2$, tous les chemins issus de s_2 contiennent des préfixes $s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots s_{n-1} \xrightarrow{\texttt{RECV d m}} s_n$ pour chaque site `d`. ■

Certaines propriétés exprimables par les opérateurs modaux de XTL peuvent être décrites aussi en termes de quantificateurs. A titre d'exemple, nous donnons ci-dessous les traductions de deux formules modales définissant une variable x de type T vers les formules équivalentes quantifiant x :

$$\langle G \ ?\ x{:}T \rangle\ \varphi(x) \stackrel{\mathrm{d}}{=} \textbf{exists } x{:}T \textbf{ in } \langle G \ !\ x \rangle\ \varphi(x)$$
$$[G \ ?\ x{:}T]\ \varphi(x) \stackrel{\mathrm{d}}{=} \textbf{forall } x{:}T \textbf{ in } [G \ !\ x]\ \varphi(x)$$

où G est un nom de porte et $\varphi(x)$ dénote une formule φ qui utilise la variable x. Cependant, pour des raisons d'efficacité, nous préférons autant que possible utiliser les opérateurs modaux de XTL à la place des quantificateurs. En effet, l'évaluation d'une formule "$\langle G \ ?\ x{:}T \rangle\ \varphi(x)$" ou "$[G \ ?\ x{:}T]\ \varphi(x)$" sur un état s requiert uniquement l'inspection des transitions du STE issues de s (dont le nombre ne dépasse pas le facteur de branchement maximal du STE, qui en pratique s'avère petit), tandis que l'évaluation de la formule "**exists**" ou "**forall**" équivalente nécessite une itération sur tout le domaine T de la variable x (ce qui, en pratique, risque de s'avérer prohibitif).

2.10.8 Opérateur "let"

La manipulation des valeurs dans les formules φ nécessite la définition et l'initialisation de variables. Ceci est réalisé grâce à l'opérateur "**let**", ayant une syntaxe similaire à l'expression E correspondante (voir la section 2.5.3) :

$$\begin{array}{l} \textbf{let } x_0{:}T_0{:=}E_0, \ldots, x_n{:}T_n{:=}E_n \textbf{ in} \\ \quad \varphi \\ \textbf{endlet} \end{array}$$

où les variables $x_0, \ldots, x_n$ sont visibles dans le corps φ de l'opérateur "**let**" et pour tout $0 \leq i \leq n$, l'expression E_i doit être de type T_i. Bien entendu, les formules "**let**" peuvent être imbriquées, les règles de visibilité des variables x_i étant les mêmes que pour les expressions "**let**".

Un état s du STE satisfait une formule "**let**" ssi il satisfait son corps φ, interprété dans le contexte des variables $x_0, \ldots, x_n$ initialisées respectivement avec les valeurs des expressions $E_0, \ldots, E_n$.

Outre le fait d'éviter la duplication de certains calculs en mémorisant leurs résultats dans des variables, l'opérateur "**let**" (utilisé conjointement avec le méta-opérateur "**current**" sur états) permet également de mémoriser les valeurs des variables BCG appartenant à un état du STE afin de les utiliser ultérieurement.

Exemple 2-57
Considérons un programme parallèle qui lit une séquence de nombres naturels sur la porte `INPUT` et calcule leur somme dans une variable `sum`. La formule suivante exprime le fait qu'après la lecture d'un nombre `n`, la variable `sum` est incrémentée de `n` :

```
let last_sum : Nat := current.sum in
    [ INPUT ? n : Nat ] (current.sum = last_sum + n)
endlet
```

Le type `Nat`, ainsi que les opérateurs `+` et `=`, sont supposés définis dans le programme à vérifier. Un état s du STE satisfait la formule ci-dessus ssi, dans chacun de ses successeurs s' tel que $s \xrightarrow{\texttt{INPUT n}} s'$, la valeur de la variable `sum` (dénotée par la deuxième occurrence de `current.sum`) est égale à la valeur de `sum` dans s (mémorisée dans la variable `last_sum`) incrémentée de `n`. ■

Comme pour l'expression "**let**", il existe aussi une forme d'opérateur "**let**" destructurante (voir la section 2.5.3) permettant d'extraire les champs de valeurs de type tuple et de les mémoriser dans des variables simples. Cet opérateur a la syntaxe suivante :

$$\begin{array}{l}\textbf{let } (x_0^0{:}T_0^0, \ldots, x_0^{n_0}{:}T_0^{n_0}) {:=} E_0, \ldots, (x_m^0{:}T_m^0, \ldots, x_m^{n_m}{:}T_m^{n_m}) {:=} E_m \textbf{ in} \\ \quad \varphi \\ \textbf{endlet}\end{array}$$

où pour chaque $0 \leq i \leq m$, l'expression E_i doit être de type $(T_i^0, \ldots, T_i^{n_i})$.

Utilisé conjointement avec les expressions "**let**", le méta-opérateur "**current**" sur états permet aussi d'exprimer des propriétés temporelles non-standard.

Exemple 2-58
La formule ci-dessous caractérise les états de blocage tels qu'ils sont définis dans le langage synchrone S/R utilisé dans l'outil COSPAN [Kur94], c'est-à-dire les états s du STE dont toutes les transitions successeur sont des boucles revenant sur s :

```
let s : state := current in
    [ true ] (current = s)
endlet
```

Lorsque cette formule est évaluée sur un état s, celui-ci est capté par la première occurrence de l'opérateur "**current**" et mémorisé dans la variable `s` de type `state` ; ensuite, la valeur de `s` est utilisée dans la formule modale afin d'exprimer que tous les états successeurs de s (dénotés par la deuxième occurrence de l'opérateur "**current**") sont identiques à s. Pour exprimer cette propriété en μ-calcul standard, il aurait fallu définir un prédicat de base P_s et une formule modale différente pour chaque état s du STE. ■

Les opérateurs "**let**" ne sont pas primitifs : ils peuvent être traduits en termes d'opérateurs "**case**" (voir la section 3.7.2).

2.10.9 Opérateur "if"

Une construction permettant l'évaluation conditionnelle des formules sur états est l'opérateur "**if**", ayant une syntaxe analogue à l'expression correspondante sur valeurs (voir la section 2.5.1) :

$$\begin{array}{l} \textbf{if } E_0 \textbf{ then } \varphi_0 \\ \quad \textbf{elsif } E_1 \textbf{ then } \varphi_1 \\ \quad \ldots \\ \quad \textbf{elsif } E_n \textbf{ then } \varphi_n \\ \ [\textbf{else } \varphi_{n+1}] \\ \textbf{endif} \end{array}$$

où les expressions $E_0, \ldots, E_n$ sont de type `boolean`. L'évaluation d'une formule "**if**" est effectuée de manière similaire à une expression "**if**" sur valeurs (voir la section 2.5.1).

Cet opérateur permet d'exprimer de manière naturelle les propriétés temporelles qui nécessitent l'analyse de plusieurs alternatives suivant des conditions booléennes sur valeurs.

Exemple 2-59
Considérons un protocole de communication sur un médium non fiable. Après la réception sur la porte `GET` d'un message `m` en provenance du médium, le protocole doit inévitablement transmettre au client récepteur soit une indication d'erreur sur la porte `INDICATE`, soit le message `m` sur la porte `DELIVER`, suivant que `m` a été corrompu ou non. Ceci peut être exprimé par la formule suivante :

```
[ GET ? m : Msg ]
    if corrupted (m) then
       mu Y . (< true > true and [ not (INDICATE ! nok) ] Y)
    else
       mu Y . (< true > true and [ not (DELIVER ! m) ] Y)
    endif
```

Le type `Msg` et la fonction `corrupted` sont définis dans le programme à vérifier. Les deux opérateurs "**mu**" ci-dessus expriment respectivement le fait qu'il est inévitable d'exécuter les actions `INDICATE nok` et `DELIVER m`. ■

L'opérateur "**if**" n'est pas primitif : il peut être traduit en termes de l'opérateur "**case**" sur valeurs (voir la section 3.7.2).

2.10.10 Opérateur "case" sur valeurs

Une autre construction utile permettant de manipuler des valeurs dans les formules φ est l'opérateur "**case**", défini de façon similaire à la construction utilisée dans les expressions E (voir la section 2.5.5) :

$$\begin{array}{l} \textbf{case } E_0 \textbf{ in} \\ \quad P_1^0 \ | \ldots | \ P_1^{n_1} \ \ [\textbf{where } E_1] \ \texttt{->} \ \varphi_1 \\ \quad \ldots \\ \ \ | \ P_m^0 \ | \ldots | \ P_m^{n_m} \ [\textbf{where } E_m] \ \texttt{->} \ \varphi_m \\ \ \ [| \ \textbf{otherwise} \ \texttt{->} \ \varphi_{m+1}] \\ \textbf{endcase} \end{array}$$

où les expressions optionnelles $E_1, \ldots, E_m$ sont de type `boolean`. La visibilité des variables est définie selon les mêmes règles que pour l'expression "**case**" : les variables définies dans un filtre P_i^j ne sont visibles que dans l'expression E_i et la formule φ_i ; en outre, pour chaque $1 \leq i \leq m$ et $0 \leq j, k \leq n_i$, les variables définies dans les filtres P_i^j et P_i^k doivent être identiques. L'évaluation d'une formule "**case**" est effectuée de manière similaire à l'expression "**case**" (voir la section 2.5.5).

Exemple 2-60
Considérons un programme parallèle implémentant le gestionnaire de processus d'un système d'exploitation. Les processus (désignés comme éléments d'un type énuméré `Pid`) en attente d'exécution sont mémorisés dans une file `wait_queue` de type `PQueue`. Le fait qu'un processus planifié pour exécution deviendra inévitablement actif (ce qui est modélisé par l'insertion du processus dans une file `run_queue`) peut être exprimé à l'aide de la formule "**case**" suivante :

```
case current.wait_queue in
   empty -> true
 | put (p : Pid, any PQueue) ->
       mu Y . (member (p, current.run_queue) or < true > true and [ true ] Y)
endcase
```

où les opérateurs `empty : -> PQueue` et `put : Pid, PQueue -> PQueue` sont des constructeurs du type `PQueue` et le prédicat `member` (supposé défini dans le programme à vérifier) dénote l'appartenance d'un processus à une file. L'opérateur "**mu**" ci-dessus exprime le fait qu'il est inévitable d'atteindre un état du STE où le processus `p` sera actif. ■

2.10.11 Opérateur "case" sur actions

XTL offre aussi un opérateur "**case action**", similaire à la construction correspondante utilisée dans les expressions (voir la section 2.5.6), permettant de filtrer les étiquettes du STE dans les formules φ :

$$\begin{array}{l} \textbf{case action } E_0 \textbf{ in} \\ \quad \alpha_1 \ [\textbf{where } E_1] \ \text{–>} \ \varphi_1 \\ \quad \ldots \\ \ | \ \alpha_m \ [\textbf{where } E_m] \ \text{–>} \ \varphi_m \\ [| \ \textbf{otherwise} \ \text{–>} \ \varphi_{m+1}] \\ \textbf{endcase} \end{array}$$

L'évaluation d'une formule "**case action**" est effectuée de manière similaire à l'expression "**case action**" (voir la section 2.5.6). Utilisé conjointement avec les opérateurs modaux (voir la section 2.10.4) et le méta-opérateur "**current**" sur actions (voir la section 2.8.3), cet opérateur permet d'exprimer de manière concise des propriétés modales complexes.

Exemple 2-61
Reprenant l'exemple 2-54, la formule "**case action**" suivante exprime l'alternance des actions `SEND` et `RECV` dans un protocole de communication :

```
nu Y (expect_send : boolean := true) .
    [ true ]
        case action current in
           SEND      -> expect_send and Y (false)
         | RECV      -> not expect_send and Y (true)
         | otherwise -> Y (expect_send)
        endcase
```

Lorsque le corps de l'opérateur "**nu**" est évalué sur un état s du STE, la formule "**case action**" est évaluée sur tous les états s' tels que $s \xrightarrow{a} s'$. L'étiquette a peut être manipulée dans la formule "**case action**" grâce au méta-opérateur "**current**" sur actions positionné implicitement par la modalité `[ true ]`. ■

En particulier, l'opérateur "**case action**" permet d'éliminer les formules α des opérateurs modaux en utilisant les identités suivantes :

$\langle\alpha\rangle\ \varphi$	$\stackrel{\text{d}}{=}$	$\langle$ **true** $\rangle$ **case action current in** α –> φ | **otherwise** –> **false** **endcase**
$[\alpha]\ \varphi$	$\stackrel{\text{d}}{=}$	[**true**] **case action current in** α –> φ | **otherwise** –> **true** **endcase**

Les méta-opérateurs "**current**" sur actions, positionnés implicitement par les formules modales "$\langle$**true**$\rangle$" et "[**true**]" ci-dessus, permettent de récupérer la valeur de l'étiquette a chaque fois que ces modalités sont évaluées sur une transition $s \xrightarrow{a} s'$ du STE.

2.11 Méta-opérateurs d'évaluation des formules sur états

Le lien entre les formules φ et les expressions E est réalisé grâce aux méta-opérateurs "|=" et "[[. . .]]" d'évaluation des formules. Ces opérateurs, décrits dans la table 2.8, constituent le "moteur" du langage XTL : ils permettent d'évaluer des formules φ sur le modèle STE d'un programme parallèle et de récupérer les résultats en termes de valeurs booléennes ou d'ensembles d'états du STE.

MÉTA-OPÉRATEUR D'ÉVALUATION	TYPE DE L'ARGUMENT E	TYPE DU RÉSULTAT	SIGNIFICATION
$\models \varphi$	—	`boolean`	vrai ssi tous les états satisfont φ
$E \models \varphi$	`state`	`boolean`	vrai ssi l'état E satisfait φ
$[[\varphi]]$	—	`stateset`	ensemble d'états satisfaisant φ

Table 2.8: Méta-opérateurs d'évaluation des formules φ

Bien entendu, les résultats renvoyés par les méta-opérateurs d'évaluation peuvent être affichés sur le fichier de sortie à l'aide de l'opérateur "**print**".

Le méta-opérateur "$\models \varphi$" est largement utilisé dans les spécifications XTL, car il constitue le moyen "standard" pour tester la validité d'une formule φ sur un modèle STE.

Exemple 2-62
L'expression ci-dessous imprime sur le fichier de sortie le message "`Absence de blocage : TRUE`" ssi le STE ne contient pas d'états de blocage :

```
print ("Absence de blocage : ", |= < true > true)
```

Le résultat renvoyé par l'opérateur "|=" est affiché en utilisant la fonction "**print**" prédéfinie associée au type `boolean` (voir la section 2.3.1). ■

Le méta-opérateur "$E \models \varphi$" permet de tester si l'état s dénoté par E satisfait une formule φ ; en particulier, il est utile pour vérifier des propriétés portant sur l'état initial du STE.

Exemple 2-63
L'expression suivante vérifie l'atteignabilité inévitable d'une action START à partir de l'état initial :

```
init |= mu Y . (< true > true and [ not START ] Y)
```

Le méta-opérateur "|=" renvoie vrai ssi l'état initial `init` satisfait la formule "**mu**". ■

Utilisé conjointement avec les opérateurs de point fixe paramétrés, le méta-opérateur "$E \models \varphi$" permet d'exprimer aussi des propriétés portant sur le passé (c'est-à-dire, sur l'ordonnancement des actions du programme exécutées *avant* d'atteindre un certain état).

Exemple 2-64
La propriété suivante exprime le fait que chaque réception d'un message `m` sur la porte `RECV m` doit être précédée par l'émission du même message sur la porte `SEND` :

```
init |= nu Y (m_sent : MsgSet := empty) . (
            [ SEND ? m : Msg ] Y (insert (m_sent, m))
            and
            [ RECV ? m : Msg ] ((m isin m_sent) and Y (remove (m_sent, m)))
            and
            [ not ((SEND any) or (RECV any)) ] Y (m_sent)
        )
```

Le paramètre `m_sent` de type `MsgSet` (initialisé à l'ensemble vide) mémorise l'ensemble des messages émis (et pas encore reçus) depuis l'état initial jusqu'à l'état courant. L'expression XTL ci-dessus spécifie que, sur toutes les séquences d'exécution issues de l'état initial du programme, tous les messages `m` contenus dans des actions `RECV m` ont été émis auparavant par des actions `SEND`. ■

Il existe des cas où il est nécessaire de connaître précisément l'ensemble d'états du STE satisfaisant une formule φ. Ceci est possible grâce au méta-opérateur "$[[\varphi]]$".

Exemple 2-65
L'expression suivante imprime sur le fichier de sortie l'ensemble des états de blocage du STE :

```
print ("Etats de blocage : ", [[ [ true ] false ]])
```

Le résultat de l'opérateur "$[[\ldots]]$" est affiché à l'aide de la fonction "**print**" prédéfinie associée au type `stateset`. ■

Remarque 2-15
Afin de simplifier la sémantique dénotationnelle des formules, ainsi que les algorithmes d'évaluation associés, nous avons interdit l'utilisation des méta-opérateurs présentés ci-dessus dans les formules α ou φ elles-mêmes. Cette restriction semble raisonnable, l'expérience n'ayant pas révélé la nécessité d'employer les méta-opérateurs d'évaluation dans les formules. Toutefois, si le besoin se fait sentir, cette limitation pourra être éliminée dans une future version du langage XTL. ■

Le méta-opérateur "$E \models \varphi$" est considéré primitif. Les deux autres méta-opérateurs d'évaluation des formules sur états "$\models \varphi$" et "$[[\varphi]]$" peuvent être traduits respectivement en termes de l'opérateur "$E \models \varphi$", de quantificateurs sur états et d'ensembles d'états, de la manière suivante :

$$\boxed{\begin{array}{lcl} \models \varphi & \stackrel{\mathrm{d}}{=} & \textbf{forall}\ s : \texttt{state}\ \textbf{in}\ s \models \varphi \\[2ex] [[\varphi]] & \stackrel{\mathrm{d}}{=} & \{\ s : \texttt{state}\ \textbf{where}\ s \models \varphi\ \} \end{array}}$$

Bien que sémantiquement correctes, les traductions ci-dessus ne sont pas suffisamment efficaces pour être utilisables en pratique ; une implémentation réaliste de ces opérateurs doit s'appuyer sur des algorithmes spécialisés, comme ceux décrits au chapitre 4.

2.12 Méta-opérateurs d'évaluation des formules sur actions

XTL permet aussi d'évaluer des formules α sur un modèle et de récupérer les résultats en termes de valeurs booléennes ou d'ensembles d'étiquettes du STE. Ceci est réalisé au moyen des méta-opérateurs "$\models$" et "[[...]]" d'évaluation des formules sur actions, donnés dans la table 2.9.

MÉTA-OPÉRATEUR D'ÉVALUATION	TYPE DE L'ARGUMENT E	TYPE DU RÉSULTAT	SIGNIFICATION
$\models$ **action** α	—	`boolean`	vrai ssi toutes les étiquettes satisfont α
$E \models$ **action** α	`label`	`boolean`	vrai ssi l'étiquette E satisfait α
[[**action** α]]	—	`labelset`	ensemble d'étiquettes satisfaisant α

Table 2.9: Méta-opérateurs d'évaluation des formules α

Le méta-opérateur "$\models$ **action**" permet de vérifier que toutes les actions du STE satisfont une formule α ; en particulier, ceci est utile pour exprimer certaines propriétés de sûreté.

Exemple 2-66
L'expression XTL suivante vérifie le fait qu'il n'existe pas d'émission de signal (action `SIGNAL`) ayant l'adresse de l'expéditeur identique à celle du destinataire :

```
|= not (SIGNAL ? src : Addr ? dest : Addr where src = dest)
```

Des propriétés similaires peuvent être exprimées au moyen des opérateurs sur actions de la logique R$\overset{\star}{\text{I}}$CO [Gar89a, pages 186–187]. ■

Le méta-opérateur "$E \models$ **action**", utilisé conjointement avec les méta-opérateurs manipulant les étiquettes et les transitions du STE (voir la section 2.3), permet d'exprimer des propriétés temporelles non-standard.

Exemple 2-67
Le compilateur CÆSAR [Gar89a] modélise la terminaison correcte d'un programme LOTOS par des actions "`EXIT` $v_1 \dots v_n$", où `EXIT` est une porte spéciale et $v_1, \dots, v_n$ ($n \geq 0$) sont les résultats renvoyés par le programme. L'expression suivante caractérise les états de terminaison dans le STE d'un programme LOTOS, c'est-à-dire les états `s` immédiatement précédés par une action "`EXIT` $v_1 \dots v_n$" :

```
exists t : trans among in (s) in
    label (t) |= action (EXIT ...)
```

Le méta-opérateur `label` renvoie l'étiquette de la transition `t` menant à `s`. L'offre générique "`...`" utilisée dans le prédicat `(EXIT ...)` permet d'ignorer les éventuels résultats $v_1, \dots, v_n$. ■

Il existe des situations (notamment, pour la mise au point des spécifications XTL) où il est nécessaire de connaître l'ensemble de toutes les actions du modèle satisfaisant une certaine propriété. Ceci peut être réalisé grâce au méta-opérateur "[[**action** α ...]]".

Exemple 2-68
L'expression suivante imprime sur le fichier de sortie l'ensemble d'étiquettes du STE qui représentent des émissions (actions `SEND`) ou des réceptions (actions `RECV`) de messages :

```
print ("Actions d'emission ou de reception de messages : ",
    [[ action (SEND ? any Msg) or (RECV ? any Msg) ]]
)
```

Le résultat du méta-opérateur "[[**action** ...]]" est affiché à l'aide de la fonction "**print**" prédéfinie associée au type `labelset`. ■

De la même manière que les opérateurs d'évaluation des formules sur états (voir la remarque 2-15), il est interdit d'utiliser les méta-opérateurs présentés ci-dessus dans les formules α ou φ elles-mêmes. Cette restriction pourra être éliminée dans une version future du langage XTL.

Les trois méta-opérateurs d'évaluation des formules sur actions peuvent être traduits respectivement en termes d'expressions "**case action**", de quantificateurs sur étiquettes et d'ensembles d'étiquettes, de la manière suivante :

E |= **action** α	$\stackrel{\mathrm{d}}{=}$	**case action** E **in** α **->** **true** | **otherwise** **->** **false** **endcase**
|= **action** α	$\stackrel{\mathrm{d}}{=}$	**forall** l **:** `label` **in** l |= **action** α
[[**action** α]]	$\stackrel{\mathrm{d}}{=}$	**{** l **:** `label` **where** l |= **action** α **}**

Une implémentation "à la volée" de l'opérateur "|= **action** α" (c'est-à-dire, ne nécessitant pas la construction préalable du modèle STE correspondant au programme parallèle à vérifier) sera indiquée à la section 4.4.2.

2.13 Définitions de formules

D'une manière similaire aux fonctions (voir la section 2.7), le langage XTL permet de définir et d'utiliser des formules sur actions et sur états. Les définitions des formules sur actions et des formules sur états ont respectivement la syntaxe suivante :

formula AF ($PAR_1, \ldots, PAR_n$) **is**
  α
endform

et

formula SF ($PAR_1, \ldots, PAR_n$) **is**
  φ
endform

où AF et SF sont les noms des formules définies, $PAR_1, \ldots, PAR_n$ ($n \geq 0$) sont leurs paramètres formels et α, φ sont leurs corps respectifs. Les noms des formules et les noms des paramètres sont des identificateurs XTL internes (voir la section 2.2). Les appels de formules XTL ont une syntaxe identique à celle des appels préfixés de fonctions :

$$AF\ (ARG_1, \ldots, ARG_n)$$

et

$$SF\ (ARG_1, \ldots, ARG_n)$$

où $ARG_1, \ldots, ARG_n$ sont les arguments de AF et SF. Cependant, les appels de formules ont une sémantique radicalement différente de celle des appels de fonctions, le passage des paramètres étant effectué par substitution syntaxique (*call-by-text* ou *in-line expansion*) :

$$AF\ (ARG_1, \ldots, ARG_n) \stackrel{\mathrm{d}}{=} \alpha\,[ARG_1/PAR_1, \ldots, ARG_n/PAR_n]$$

$$SF\ (ARG_1, \ldots, ARG_n) \stackrel{\mathrm{d}}{=} \varphi\,[ARG_1/PAR_1, \ldots, ARG_n/PAR_n]$$

où $\alpha\,[ARG_1/PAR_1,\ldots,ARG_n/PAR_n]$ et $\varphi\,[ARG_1/PAR_1,\ldots,ARG_n/PAR_n]$ dénotent respectivement la substitution syntaxique de $PAR_1,\ldots,PAR_n$ par $ARG_1,\ldots,ARG_n$ dans α et φ.

Ce mécanisme de passage de paramètres, utilisé aussi dans certains langages de programmation (comme C et C++), permet d'obtenir, de manière simple, des définitions de formules d'ordre supérieur, c'est-à-dire des formules α et φ paramétrées par d'autres formules. Bien que ce mécanisme ne permet de simuler qu'une partie restreinte de l'ordre supérieur (par exemple, le passage d'une formule comme l'un de ses propres paramètres est interdit), il s'avère suffisant en pratique, tout en offrant l'avantage d'une sémantique et d'une implémentation simples.

Exemple 2-69
Reprenant l'exemple 2-38, la formule `TAU` définie ci-dessous caractérise les actions invisibles du STE :

```
formula TAU () is
    (any where not visible (current))
endform
```

Afin de faciliter les notations, l'utilisation des parenthèses dans les appels de formules nullaires n'est pas obligatoire : par exemple, les états du STE permettant l'exécution d'une action invisible peuvent être caractérisés par la formule `< TAU > true`. ■

Les formules définies au moyen des constructions "**formula**" peuvent être paramétrées par d'autres formules ; ceci permet de décrire de manière naturelle les opérateurs de logique temporelle.

Exemple 2-70
Reprenant l'exemple 2-32, la formule `EU` ci-dessous définit l'opérateur $\mathbf{E}\,[\varphi_1 \;\mathbf{U}\; \varphi_2]$ de CTL :

```
formula EU (P1, P2) is
    mu Y . ((P2) orelse (P1) and < true > Y)
endform
```

Les paramètres `P1` et `P2` dénotent respectivement les formules φ_1 et φ_2. A titre d'exemple, la formule `EU (true, [ true ] false)` caractérise les états conduisant potentiellement à un blocage. ■

L'expansion des formules est effectuée au moyen d'un outil XTL auxiliaire, appelé *expanseur*. Cet outil remplace chaque appel de formule par son corps, dans lequel les paramètres ont été syntaxiquement substitués avec les arguments respectifs. Pour des raisons d'efficacité, l'expansion des formules est effectuée en un seul passage sur le texte source du programme XTL ; ceci impose que les définitions de formules (contrairement aux définitions de fonctions XTL) doivent être placées dans le programme *avant* leurs appels respectifs.

L'expansion des formules a lieu avant l'analyse syntaxique ; ce choix est inspiré des implémentations existantes pour divers langages de programmation, tels que C ou C++, qui possèdent un préprocesseur (`/lib/cpp` sous UNIX). Cependant, à la différence des préprocesseurs des langages C et C++, l'expanseur XTL autorise la surcharge des formules, résolue suivant le nombre de paramètres. Cette facilité est particulièrement utile lorsqu'il s'agit de définir, pour un opérateur temporel donné, des opérateurs dérivés ayant le même nom, mais un nombre d'arguments différent.

Exemple 2-71
Dans les spécifications écrites en LTAC, il est courant d'employer, outre les opérateurs temporels $\mathbf{pot}(\varphi_1, \varphi_2)$ et $\mathbf{inev}(\varphi_1, \varphi_2)$ à deux arguments (qui sont équivalents aux opérateurs $\mathbf{E}\,[\varphi_1 \;\mathbf{U}\; \varphi_2]$ et $\mathbf{A}\,[\varphi_1 \;\mathbf{U}\; \varphi_2]$ de CTL), leurs opérateurs dérivés à un seul argument, définis comme suit :

```
formula POT (P)  is POT (true, P)  endform

formula INEV (P) is INEV (true, P) endform
```

La même situation se présente avec les opérateurs $\mathbf{E}\left[{\varphi_1}_{\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2\right]$ et $\mathbf{A}\left[{\varphi_1}_{\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2\right]$ d'ACTL. ■

2.14 Inclusions de bibliothèques

L'expanseur XTL autorise un certain degré de modularité : l'utilisateur peut définir, dans des fichiers séparés, des bibliothèques réutilisables d'opérateurs de logique temporelle qui peuvent être incluses dans d'autres programmes XTL. La construction permettant d'effectuer l'inclusion de fichiers externes dans un programme XTL a la syntaxe suivante :

library
$FN_0, \ldots, FN_n$
endlib

où $FN_0, \ldots, FN_n$ représentent des noms de fichiers. Chaque occurrence d'une construction "**library**" est remplacée syntaxiquement dans le programme XTL par les contenus des fichiers $FN_0, \ldots, FN_n$ (concaténés dans cet ordre).

Exemple 2-72
Supposant qu'on dispose d'une implémentation des opérateurs de la logique CTL dans un fichier XTL appelé `ctl.xtl` (comme celui fourni dans l'annexe C.1), la directive suivante :

```
library ctl.xtl endlib
```

permet d'utiliser les opérateurs de CTL dans les spécifications écrites en XTL. ■

L'inclusion des fichiers est effectuée au moyen de l'expanseur XTL, simultanément avec l'expansion des formules. Si un même fichier est inclus (directement ou transitivement) plusieurs fois de suite, uniquement la première inclusion est effectuée. Les inclusions de fichiers (directement ou mutuellement) récursives sont naturellement prohibées.

Des exemples de bibliothèques implémentant en XTL divers opérateurs de logique temporelle peuvent être trouvés dans l'annexe C.

2.15 Programme XTL

XTL étant un langage applicatif, un programme XTL est essentiellement constitué d'une expression, appelée aussi *corps* du programme. L'exécution d'un programme XTL consiste à évaluer son corps E en présence d'un fichier de sortie vide ; le résultat du programme est représenté par le contenu du fichier de sortie après l'évaluation de E. Le seul moyen de récupérer des résultats est donc de les imprimer sur le fichier de sortie à l'aide des opérateurs "**print**". Un programme XTL a la syntaxe suivante :

$$[L_0 \ldots L_m]\ [M_0 \ldots M_n]\ [D_0 \ldots D_p]$$
$$E$$
$$[\textbf{where}\ [L_{m+1} \ldots L_{m+q}]\ [M_{n+1} \ldots M_{n+r}]\ D_{p+1} \ldots D_{p+s}]$$

où $L_0, \ldots, L_{m+q}$ sont des inclusions de bibliothèques, $M_0, \ldots, M_{n+r}$ sont des définitions de formules, $D_0, \ldots, D_{p+s}$ sont des définitions de fonctions et l'expression E est le *corps* du programme. Toutes les constructions ci-dessus, à l'exception de E, sont optionnelles.

Les fonctions définies dans $D_0, \ldots, D_{p+s}$ obéissent aux règles de visibilité précisées à la section 2.7 : les fonctions locales (précédées par le mot-clé "**local**") sont visibles uniquement dans les corps des autres fonctions, tandis que les fonctions globales sont visibles aussi dans le corps E du programme. Comme nous avons précisé à la section 2.13, les définitions de formules doivent précéder leur utilisation.

L'exemple suivant illustre la structure typique d'un programme XTL.

Exemple 2-73
Le programme XTL ci-dessous représente la spécification en ACTL d'un protocole de communication. La bibliothèque `actl.xtl`, contenant les opérateurs ACTL (voir la section C.2), est incluse en tête du programme. Ensuite, les propriétés temporelles sont évaluées à l'aide de méta-opérateurs "|=" et leurs valeurs de vérité, accompagnées de messages explicatifs, sont imprimées sur le fichier de sortie à l'aide d'opérateurs "**print**".

```
library actl.xtl endlib

print ("Atteignabilite inevitable d'une emission : ",
    |= AU_A_B (true, SEND ? any Msg)
);

print ("Emission obligatoire avant la premiere reception : ",
    init |= not EU_A_B (not (SEND ? any Msg), RECV ? any Msg)
);

print ("Reception obligatoire entre deux emissions successives : ",
    |= [ SEND ? any Msg ] not EU_A_B (not (RECV ? any Msg), SEND ? any Msg)
);

print ("Emission obligatoire entre deux receptions successives : ",
    |= [ RECV ? any Msg ] not EU_A_B (not (SEND ? any Msg), RECV ? any Msg)
);

print ("Transmission correcte des messages : ",
    |= [ SEND ? m : Msg ] AU_A_B (not (SEND ? any Msg), RECV ! m)
)
```

Les opérateurs temporels `EU_A_B (A1, A2)` et `AU_A_B (A1, A2)` (implémentant les opérateurs $\mathbf{E}\,[\mathit{true}_{\alpha_1}\mathbf{U}_{\alpha_2}\mathit{true}]$ et $\mathbf{A}\,[\mathit{true}_{\alpha_1}\mathbf{U}_{\alpha_2}\mathit{true}]$ d'ACTL) expriment respectivement l'atteignabilité potentielle et inévitable, après une séquence d'actions satisfaisant α_1, d'une action satisfaisant α_2.

La 1$^{\text{ère}}$ propriété ci-dessus exprime le fait qu'à partir de tout état du programme, il est inévitable d'atteindre l'émission d'un message (le client émetteur du protocole peut toujours envoyer un message) ; c'est une propriété de vivacité (*liveness property*) qui implique l'absence de blocage.

Les 2$^{\text{ème}}$, 3$^{\text{ème}}$ et 4$^{\text{ème}}$ propriétés expriment l'alternance entre les émissions et réceptions de messages, en commençant par une émission ; il s'agit de propriétés de sûreté (*safety properties*).

La 5$^{\text{ème}}$ propriété exprime qu'après chaque émission d'un message `m`, il est inévitable d'atteindre la réception du même message ; c'est une propriété "mixte" qui combine des aspects de sûreté et de vivacité. ■

Chapitre 3

Sémantique dénotationnelle des formules

Le langage XTL contient des formules permettant d'exprimer des propriétés portant sur les états, les actions et/ou les valeurs contenues dans le modèle STE étendu généré à partir d'un programme parallèle. La sémantique des formules XTL a été définie informellement aux sections 2.8, 2.9 et 2.10.

Ce chapitre contient la définition formelle de la sémantique dénotationnelle des formules XTL. Cette formalisation servira de base pour la présentation des algorithmes d'évaluation des formules XTL sur un modèle STE étendu, qui feront l'objet du chapitre 4. En plus de la définition sémantique des opérateurs XTL, nous précisons également leurs propriétés. En particulier, nous identifions les *opérateurs primitifs* constituant un sous-ensemble minimal complet du langage des formules XTL et nous justifions la traduction des *opérateurs dérivés* en termes des opérateurs primitifs.

Finalement, nous décrivons différentes phases préliminaires de transformation des formules XTL (élimination des opérateurs dérivés sur actions et sur états, transformation en forme normale positive) qui sont effectuées dans le but de faciliter l'évaluation des formules sur un modèle.

3.1 Préliminaires

Suivant l'approche utilisée dans la littérature consacrée au μ-calcul standard [Koz83, EL86, CS91b, And94], nous définissons la sémantique des formules XTL sous forme dénotationnelle. Cette méthode s'avère plus adaptée que les méthodes opérationnelles pour définir la sémantique des opérateurs modaux et de point fixe. En outre, elle offre un niveau d'abstraction plus élevé : pour une même sémantique dénotationnelle des formules, il peut exister plusieurs algorithmes d'évaluation (ce qui est effectivement le cas pour le μ-calcul standard), chacun d'entre eux implémentant une sémantique opérationnelle différente.

Dans la définition sémantique des formules, nous supposons que toutes les phases d'analyse statique ont été effectuées (voir l'annexe A) : en particulier, tous les objets manipulés ont des noms uniques et tous les attributs statiques (notamment l'information de type, dénotée par l'attribut *type*) ont été calculés. Afin de simplifier les définitions sémantiques, la grammaire abstraite des formules que nous utilisons dans ce chapitre est légèrement différente de celle présentée à la section 2.1 (les deux différences ci-dessous n'affectant pas la sémantique des formules) :

- la règle syntaxique "$P ::= P$ **of** RT", qui sert uniquement pour le typage des filtres, est supprimée, car elle n'est utile que pendant l'analyse statique ;
- la règle syntaxique "$\alpha ::= (G_0|O_0)\ O_1 \ldots O_m$ [**...**] $O_{m+1} \ldots O_{m+n}$ [**where** E]" est remplacée par "$\alpha ::= O_0 \ldots O_m$ [**...**] $O_{m+1} \ldots O_{m+n}$ [**where** E]", car l'occurrence du symbole terminal G_0 (introduite pour compatibilité avec le format des actions LOTOS) est sémantiquement équivalente à une offre "**!** G_0".

Nous commençons par indiquer les domaines et les fonctions syntaxiques associés aux symboles contenus dans les formules XTL, ainsi que les domaines et les fonctions sémantiques utilisés dans ce chapitre. Ensuite, nous définissons formellement la sémantique des différentes constructions constituant le langage des formules XTL, en précisant aussi leurs propriétés.

3.1.1 Domaines et fonctions syntaxiques

Pour chaque symbole M de la grammaire abstraite des formules XTL, nous introduisons un domaine syntaxique représentant l'ensemble des phrases du langage générées par M. La table 3.1 indique les domaines syntaxiques utilisés.

SYMBOLE	DOMAINE SYNTAXIQUE
x	*DVar*
Y	*PVar*
E	*Exp*
P	*Pattern*
O	*Offer*
α	*AForm*
R	*RegExp*
φ	*SForm*

Table 3.1: Domaines syntaxiques

Le domaine *DVar* des variables simples contient, outre les variables XTL et BCG respectivement définies et utilisées dans le programme XTL, les variables spéciales *c_a* et *c_s* (supposées différentes de toutes les autres variables simples), dénotant respectivement les occurrences des méta-opérateurs "**current**" sur actions et sur états. Bien qu'au niveau du langage utilisateur ces méta-opérateurs soient interprétés comme des fonctions constantes surchargées (voir les sections 2.8.3 et 2.10.2), dans la définition sémantique il est plus commode de les représenter comme des variables (voir la section 3.2).

A chacun des symboles non-terminaux E, P, O, α, R et φ indiqués dans la table 3.1, nous associons des fonctions syntaxiques (définies respectivement aux sections 3.2.1, 3.3.1, 3.4.1, 3.5.1, 3.6.1 et 3.7.1) qui calculent diverses informations extraites des phrases générées par le symbole en cause (variables libres et liées, nombre d'opérateurs, ...). Chaque fonction syntaxique est définie inductivement sur la structure syntaxique du symbole non-terminal respectif.

3.1.2 Domaines et fonctions sémantiques

Tout au long de ce chapitre, nous supposons la présence d'un modèle STE étendu $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$ (voir la définition 1-4) sur lequel les formules XTL sont interprétées. Nous introduisons aussi les domaines sémantiques suivants, utilisés pour définir la sémantique dénotationnelle des formules XTL :

- S, A et T sont les domaines associés respectivement aux états, actions et transitions du modèle STE étendu. Les domaines 2^S, 2^A et 2^T correspondent respectivement aux ensembles d'états, d'actions et de transitions du STE.

- $\mathbf{Bool} \stackrel{\mathrm{d}}{=} \{\mathbf{ff}, \mathbf{tt}\}$ est le domaine des valeurs booléennes, muni des opérations classiques *and*, *or* et *not*.

- A chaque type de données T_i, nous associons un domaine représentant les valeurs de ce type. Pour simplifier les notations, nous utilisons le même symbole T_i pour dénoter ce domaine.

- **Val** est le domaine de toutes les valeurs dénotées par les expressions ou variables simples. Ce domaine est l'union de tous les domaines T_i associés aux types des expressions et des variables simples. Cette union inclut les domaines S, A et T associés au modèle STE étendu, ainsi que les domaines ensemblistes correspondants 2^S, 2^A et 2^T : en particulier, ceci permet de prendre en compte les valeurs renvoyées par les méta-opérateurs d'évaluation de formules.

- $\mathbf{DEnv} \stackrel{\mathrm{d}}{=} \mathit{DVar} \rightarrow \mathbf{Val}$ est le domaine des *environnements simples*. Un environnement simple $\varepsilon \in \mathbf{DEnv}$ est une fonction partielle associant à chaque variable simple $x \in \mathit{supp}(\varepsilon)$ une valeur v. Le domaine des environnements est muni des opérations $\oslash$, $\oplus$, etc. sur les fonctions partielles, définies au chapitre de notations.

- **Param** est le domaine des paramètres des variables propositionnelles. Ce domaine est l'union de tous les produits cartésiens des domaines associés aux types des paramètres des variables Y. Par extension des opérations $\cup$, $\cap$ et de la relation $\subseteq$ sur 2^S, nous définissons les opérations $\sqcup$, $\sqcap$ et la relation $\sqsubseteq$ sur $\mathbf{Param} \rightarrow 2^S$. La structure de treillis complet de $\langle 2^S, \cup, \cap, \subseteq \rangle$ induit une structure de treillis complet pour $\langle \mathbf{Param} \rightarrow 2^S, \sqcup, \sqcap, \sqsubseteq \rangle$.

- $\mathbf{PEnv} \stackrel{\mathrm{d}}{=} \mathit{PVar} \rightarrow \mathbf{Param} \rightarrow 2^S$ est le domaine des *environnements propositionnels*. Un environnement propositionnel $\rho \in \mathbf{PEnv}$ est une fonction partielle associant à chaque variable propositionnelle $Y(x_1{:}T_1, \ldots, x_n{:}T_n) \in \mathit{supp}(\rho)$ une fonction $\rho(Y) : T_1 \times \cdots \times T_n \rightarrow 2^S$. Pour une variable propositionnelle Y et des valeurs $(v_1, \ldots, v_n) \in T_1 \times \cdots \times T_n$, $(\rho(Y))(v_1, \ldots, v_n)$ renvoie l'ensemble d'états du STE qui satisfont Y lorsque $v_1, \ldots, v_n$ sont substituées aux variables $x_1, \ldots, x_n$.

Remarque 3-1
Une définition dénotationnelle rigoureuse exige que les domaines sémantiques (ici, les domaines T_i, **Val**, **DEnv**, **Param** et **PEnv**) possèdent une structure d'ordre partiel complet (*cpo*), obtenue en leur rajoutant un plus petit élément $\perp$, qui modélise la "non définition". Par souci de simplicité, nous supposerons que chacun de ces domaines possède un élément $\perp$ et nous n'utiliserons pas la notation $D_\perp$, qui désigne le domaine D auquel on rajoute l'élément $\perp$. ■

A chacun des symboles non-terminaux E, P, O, α, R et φ indiqués dans la table 3.1, nous associons une fonction d'interprétation (présentée respectivement à la section 3.2.2, 3.3.2, 3.4.2, 3.5.2, 3.6.2 et 3.7.2) qui définit la sémantique du symbole en cause. Suivant l'approche classique en sémantique dénotationnelle [Sch88], chaque fonction d'interprétation est définie inductivement sur la structure syntaxique du symbole non-terminal respectif.

Afin de simplifier la présentation, nous utilisons la même notation "$[\![.]\!]$" pour toutes les fonctions sémantiques, les surcharges pouvant être levées suivant les types des arguments et/ou du résultat. L'opérateur $\rightarrow$ est supposé associatif à droite et moins prioritaire que l'opérateur $\times$. Pour définir les fonctions sémantiques, outre les prédicats de la logique du premier ordre, nous utilisons également les constructions "*let*" et "*if-then-else*" communément employées dans les langages de programmation fonctionnels.

Remarque 3-2
Toutes les fonctions sémantiques sont supposées être *strictes*, c'est-à-dire que la sémantique d'une phrase du langage est indéfinie (égale à $\perp$) chaque fois que la sémantique d'une de ses sous-phrases l'est aussi. En particulier, les constructions "*if-then-else*" utilisées pour définir les fonctions sémantiques ont la clause "*else*" optionnelle : si celle-ci est absente, la fonction sémantique respective sera indéfinie lorsque la clause "*if*" est fausse. Dans une implémentation du langage XTL, ceci produira un arrêt du programme avec impression d'un message d'erreur sur le fichier de sortie. ■

Chaque fonction sémantique associée à un symbole non-terminal N possède comme argument un modèle STE étendu $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$ sur lequel N est interprété ; toutefois, afin d'alléger les notations, cet argument est considéré implicite, car identique pour toutes les fonctions.

3.2 Expressions

Comme il a été précisé aux sections 2.11 et 2.12, les expressions E contenues dans les formules α et φ ne couvrent pas tout le langage des expressions XTL : elles ne contiennent pas de méta-opérateurs d'évaluation de formules. Cette restriction, qui pourrait être éliminée dans une version future du langage, permet une meilleure séparation de la sémantique des formules et des expressions, tout en simplifiant les algorithmes d'évaluation. Par souci de concision, nous ne présentons ci-dessous que les aspects de la syntaxe et de la sémantique des expressions strictement nécessaires pour définir l'interprétation des formules XTL. La définition complète de la sémantique dénotationnelle des expressions figure dans l'annexe A.

3.2.1 Aspects syntaxiques

Les éléments syntaxiques associés aux expressions E sont définis par les fonctions suivantes :

$$fdv, bdv : Exp \to 2^{DVar}$$

Etant donné une expression E, les dénotations $fdv(E)$ et $bdv(E)$ renvoient respectivement l'ensemble des variables simples libres et liées dans E. Ces deux fonctions syntaxiques sont définies formellement à la section B.1.1. Informellement, une occurrence de variable x dans E est *liée* ssi elle est contenue dans une sous-expression E' de E qui définit x (c'est le cas, par exemple, si E a la forme "**let** x:T:=E_0 **in** E_1 **endlet**"). Toutes les autres occurrences des variables x dans E sont *libres*. L'ensemble $fdv(E)$ peut aussi contenir les variables spéciales *c_a* et *c_s*, associées respectivement aux occurrences d'opérateurs "**current**" sur actions et sur états contenues dans E.

3.2.2 Aspects sémantiques

Pour définir l'interprétation des formules XTL, il suffit de connaître le profil de la fonction d'interprétation des expressions contenues dans les formules (pour plus de détails, voir l'annexe B.1.2) :

$$[\![.]\!] : Exp \to \mathbf{DEnv} \to \mathbf{Val}$$

Etant donné une expression E et un environnement ε tel que $fdv(E) \subseteq supp(\varepsilon)$, la dénotation $[\![E]\!]\varepsilon$ renvoie la valeur de E calculée dans le contexte de ε.

Remarque 3-3
La sémantique statique des formules XTL (voir l'annexe A.8) assure le fait que l'état courant (*resp.* l'action courante) dénoté(e) par chaque occurrence d'un opérateur "**current**" sera toujours initialisé(e) par l'environnement ε dans le contexte duquel l'opérateur respectif est évalué. ■

A titre d'exemple, nous donnons ci-dessous l'interprétation des méta-opérateurs "**current**" sur actions et sur états qui, eux, peuvent être utilisés dans les formules XTL :

$$[\![\mathbf{current}]\!]\,\varepsilon \stackrel{\mathrm{d}}{=} \mathit{if}\ \mathit{type}(\mathbf{current}) = \mathtt{label}\ \mathit{then}\ \varepsilon(c\_a)\ \mathit{else}\ \varepsilon(c\_s)\ \mathit{endif}$$

L'attribut *type* a été positionné lors du typage des expressions et des formules (voir l'annexe A.7).

3.3 Filtres

Les filtres P (présentés informellement à la section 2.5.4) permettent de tester la structure d'une valeur typée (représentée par un terme algébrique sous forme normale) et d'en extraire éventuellement des informations, en les affectant à des variables afin d'utilisation ultérieure. En particulier, les filtres XTL utilisés (par l'intermédiaire des offres O) dans les formules d'actions α et les expressions régulières R permettent d'extraire les valeurs contenues dans les actions du modèle STE et de les propager à l'intérieur des formules modales XTL.

3.3.1 Aspects syntaxiques

Les éléments syntaxiques associés aux filtres P sont définis par la fonction suivante :

$$\mathit{bdv} : \mathit{Pattern} \to 2^{DVar}$$

Etant donné un filtre P, la dénotation $\mathit{bdv}(P)$ renvoie l'ensemble des variables simples liées par P. Cette fonction syntaxique est définie inductivement dans la table 3.2.

P	$\mathit{bdv}(P)$
$x{:}T$	$\{x\}$
$(x_0{:}T_0, \ldots, x_n{:}T_n)$	$\{x_0, \ldots, x_n\}$
any T	$\emptyset$
$C\ (P_1, \ldots, P_n)$	$\bigcup_{i=1}^{n} \mathit{bdv}(P_i)$

Table 3.2: Variables simples liées dans les filtres

Comme il a été précisé informellement à la section 2.5.4, pour chaque filtre P, les variables contenues dans $\mathit{bdv}(P)$ doivent être deux à deux disjointes, car elles dénotent des occurrences de définition ayant la même portée dans le programme XTL.

3.3.2 Aspects sémantiques

La sémantique des filtres est définie par la fonction d'interprétation suivante :

$$[\![.]\!] : \mathit{Pattern} \to \mathbf{Val} \to \mathbf{Bool} \times \mathbf{DEnv}$$

Etant donné un filtre P et une valeur v, la dénotation $[\![P]\!]\,v$ renvoie un tuple ayant deux champs : (1) une valeur booléenne, indiquant si P filtre v et (2) un environnement simple, contenant les variables initialisées par P avec des valeurs extraites de v. La fonction sémantique est définie inductivement de la manière suivante :

$$
\begin{array}{rcl}
[\![x{:}T]\!]\, v & \stackrel{\mathrm{d}}{=} & \textit{if type}(v) = T \textit{ then } (\mathbf{tt}, [v/x]) \textit{ else } (\mathbf{ff}, [\,]) \textit{ endif} \\
[\![(x_0{:}T_0, \ldots, x_n{:}T_n)]\!]\, v & \stackrel{\mathrm{d}}{=} & \textit{if type}(v) = (T_0, \ldots, T_n) \textit{ then} \\
 & & \quad (\mathbf{tt}, [(v)_0/x_0, \ldots, (v)_n/x_n]) \\
 & & \textit{else} \\
 & & \quad (\mathbf{ff}, [\,]) \\
 & & \textit{endif} \\
[\![\mathbf{any}\ T]\!]\, v & \stackrel{\mathrm{d}}{=} & \textit{if type}(v) = T \textit{ then } (\mathbf{tt}, [\,]) \textit{ else } (\mathbf{ff}, [\,]) \textit{ endif} \\
[\![C\ (P_1, \ldots, P_n)]\!]\, v & \stackrel{\mathrm{d}}{=} & \textit{if } \exists v_1 : \textit{type}(P_1), \ldots, v_n : \textit{type}(P_n).v = C(v_1, \ldots, v_n) \ \wedge \\
 & & \quad\quad \forall i \in [0, n].([\![P_i]\!]\, v_i)_1 = \mathbf{tt} \\
 & & \textit{then} \\
 & & \quad (\mathbf{tt}, \bigoplus_{i=1}^{n} ([\![P_i]\!]\, v_i)_2) \\
 & & \textit{else} \\
 & & \quad (\mathbf{ff}, [\,]) \\
 & & \textit{endif}
\end{array}
$$

3.4 Offres

Les offres O (présentées informellement à la section 2.8.1) sont utilisées dans les filtres d'actions XTL afin de définir des prédicats sur la structure des actions du modèle STE. Les offres XTL peuvent contenir des expressions E ou des filtres P, ce qui leur permet soit de tester si un champ d'une action a une certaine valeur, soit d'en extraire des informations et les mémoriser dans des variables.

3.4.1 Aspects syntaxiques

Les éléments syntaxiques associés aux offres O sont définis par les fonctions suivantes :

$$\textit{fdv}, \textit{bdv} : \textit{Offer} \rightarrow 2^{DVar}$$

Etant donné une offre O, la dénotation $\textit{fdv}(O)$ (*resp.* $\textit{bdv}(O)$) renvoie l'ensemble des variables simples libres (*resp.* liées) dans O. Ces fonctions syntaxiques sont définies inductivement dans la table 3.3.

O	$\textit{fdv}(O)$	$\textit{bdv}(O)$
any	$\emptyset$	$\emptyset$
! E	$\textit{fdv}(E)$	$\emptyset$
? $P_0 \mid \ldots \mid P_n$	$\emptyset$	$\textit{bdv}(P_0)$

Table 3.3: Variables simples libres et liées dans les offres

Comme il a été précisé informellement à la section 2.8.1, pour chaque offre avec filtres "**?** $P_0 \mid \ldots \mid P_n$" et pour tous $0 \leq i, j \leq n$, $\textit{bdv}(P_i) = \textit{bdv}(P_j)$.

3.4.2 Aspects sémantiques

La sémantique des offres est définie par la fonction suivante :

$$[\![.]\!] : \mathit{Offer} \to \mathbf{DEnv} \to \mathbf{Val} \to \mathbf{Bool} \times \mathbf{DEnv}$$

Etant donné une offre O, un environnement ε tel que $\mathit{fdv}(O) \subseteq \mathit{supp}(\varepsilon)$ et une valeur v, la dénotation $[\![O]\!]\,\varepsilon v$ renvoie un tuple ayant deux champs : (1) une valeur booléenne, indiquant si O filtre v dans le contexte de ε et (2) un environnement simple, contenant les variables initialisées par O avec des valeurs extraites de v. La fonction sémantique est définie inductivement comme suit :

$$\begin{array}{rcl}
[\![\mathbf{any}]\!]\,\varepsilon v & \stackrel{\mathrm{d}}{=} & (\mathbf{tt}, [\,]) \\
[\![\mathbf{!}\ E]\!]\,\varepsilon v & \stackrel{\mathrm{d}}{=} & \mathit{if}\ \ [\![E]\!]\,\varepsilon = v\ \mathit{then}\ (\mathbf{tt}, [\,])\ \mathit{else}\ (\mathbf{ff}, [\,])\ \mathit{endif} \\
[\![\mathbf{?}\ P_0 \mid \ldots \mid P_n]\!]\,\varepsilon v & \stackrel{\mathrm{d}}{=} & \mathit{if}\ \exists i \in [0, n].\,(([\![P_i]\!]\,v)_1 = \mathbf{tt}\ \wedge\ \forall j \in [0, i-1].([\![P_j]\!]\,v)_1 = \mathbf{ff})\ \ \mathit{then} \\
& & \quad (\mathbf{tt}, ([\![P_i]\!]\,v)_2) \\
& & \mathit{else} \\
& & \quad (\mathbf{ff}, [\,]) \\
& & \mathit{endif}
\end{array}$$

Remarque 3-4
Lorsqu'une offre multiple "**?** $P_0 \mid \ldots \mid P_n$" est appliquée sur une valeur v, les filtres sont évalués de gauche à droite et les variables exportées par l'offre sont initialisées avec les valeurs extraites par le premier P_i qui filtre v. Ceci assure une sémantique déterministe pour les offres : chaque variable exportée par une offre est initialisée avec une seule valeur. ■

3.5 Formules sur actions

La sémantique que nous avons proposé pour les formules α (présentées informellement à la section 2.8) est une généralisation de la sémantique booléenne des formules sur actions rencontrées dans ACTL [NV90] ou dans certaines variantes du μ-calcul standard [Bra92]. A la différence de ces formalismes, les formules XTL sur actions permettent d'extraire et de manipuler les valeurs contenues dans les actions du STE, au moyen d'offres qui mémorisent ces valeurs dans des variables simples.

Comme il a été mentionné à la section 2.10.7, les modalités contenant des formules α qui exportent des variables peuvent être traduites en termes de quantificateurs et de modalités dont les formules α ne contiennent que des occurrences d'utilisation de variables simples. Cependant, la spécification avec des formules α qui exportent des variables présente plusieurs avantages par rapport à l'utilisation explicite de quantificateurs :

Indépendance des formules par rapport à la description. Les formules α qui exportent des variables permettent de spécifier des propriétés temporelles qui ne dépendent pas des paramètres de l'application à vérifier (nombre de processus, taille des messages, ...). En effet, les formules α permettent d'extraire l'information *localement*, en explorant les actions individuelles du STE, sans avoir à connaître le modèle dans sa totalité. De cette manière, une spécification XTL peut être utilisée, sans modification, pour vérifier plusieurs instances d'une même application, obtenues en donnant des valeurs différentes aux paramètres. En revanche, l'utilisation explicite des quantificateurs nécessite la connaissance des domaines des variables quantifiées, lesquels dépendent souvent des paramètres de l'application.

Efficacité d'évaluation. En général, les modalités contenant des formules α qui exportent des variables s'évaluent beaucoup plus efficacement que les quantificateurs. En effet, les premières nécessitent d'examiner uniquement les transitions issues d'un état du STE, tandis que les dernières imposent, dans le pire des cas, une itération sur tout le domaine de la variable quantifiée (qui peut être beaucoup plus grand que le facteur de branchement du STE).

Facilité d'implémentation. Pour évaluer les formules avec quantificateurs, il est nécessaire de disposer de fonctions spéciales capables d'énumérer toutes les valeurs des domaines (supposés bornés) des variables quantifiées. Pour certains domaines (listes, arbres, ensembles, ...), les fonctions d'énumération peuvent être très compliquées à mettre en œuvre. En revanche, l'évaluation des formules α ne requiert que des mécanismes simples de filtrage des valeurs contenues dans les actions du modèle STE.

Ceci montre l'intérêt des formules sur actions telles que nous les avons conçues en XTL. Leur sémantique, ainsi que leurs propriétés, sont définies et étudiées dans les sections suivantes.

3.5.1 Aspects syntaxiques

Les éléments syntaxiques associés aux formules α sur actions sont définis par les fonctions suivantes :

$$fdv, v_{tt}, v_{ff} : AForm \to 2^{DVar}$$

Etant donné une formule α, la dénotation $fdv(\alpha)$ renvoie l'ensemble de variables simples libres dans α. La dénotation $v_{tt}(\alpha)$ (*resp.* $v_{ff}(\alpha)$) renvoie l'ensemble des variables simples exportées par α quand celle-ci est satisfaite (*resp.* quand elle ne l'est pas) dans le contexte d'un environnement ε et d'une action a. Ces fonctions syntaxiques sont définies inductivement dans la table 3.4.

α	$fdv(\alpha)$	$v_{tt}(\alpha)$	$v_{ff}(\alpha)$
$O_0 \dots O_m[\dots]$ $O_{m+1} \dots O_{m+n}$ [**where** E]	$\bigcup_{i=0}^{m+n} fdv(O_i) \cup$ $(fdv(E) \setminus \bigcup_{i=0}^{m+n} bdv(O_i))$	$\bigcup_{i=0}^{m+n} bdv(O_i)$	$\emptyset$
true	$\emptyset$	$\emptyset$	$\emptyset$
false	$\emptyset$	$\emptyset$	$\emptyset$
not α	$fdv(\alpha)$	$v_{ff}(\alpha)$	$v_{tt}(\alpha)$
α_1 **or** α_2	$fdv(\alpha_1) \cup fdv(\alpha_2)$	$v_{tt}(\alpha_1) \cap v_{tt}(\alpha_2)$	$v_{ff}(\alpha_1) \cup v_{ff}(\alpha_2)$
α_1 **and** α_2	$fdv(\alpha_1) \cup fdv(\alpha_2)$	$v_{tt}(\alpha_1) \cup v_{tt}(\alpha_2)$	$v_{ff}(\alpha_1) \cap v_{ff}(\alpha_2)$
α_1 **implies** α_2	$fdv(\alpha_1) \cup fdv(\alpha_2)$	$v_{ff}(\alpha_1) \cap v_{tt}(\alpha_2)$	$v_{tt}(\alpha_1) \cup v_{ff}(\alpha_2)$
α_1 **iff** α_2	$fdv(\alpha_1) \cup fdv(\alpha_2)$	$(v_{ff}(\alpha_1) \cap v_{tt}(\alpha_2)) \cup$ $(v_{tt}(\alpha_1) \cap v_{ff}(\alpha_2))$	$(v_{tt}(\alpha_1) \cup v_{ff}(\alpha_2)) \cap$ $(v_{ff}(\alpha_1) \cup v_{tt}(\alpha_2))$
α_1 **xor** α_2	$fdv(\alpha_1) \cup fdv(\alpha_2)$	$(v_{tt}(\alpha_1) \cup v_{ff}(\alpha_2)) \cap$ $(v_{ff}(\alpha_1) \cup v_{tt}(\alpha_2))$	$(v_{ff}(\alpha_1) \cap v_{tt}(\alpha_2)) \cup$ $(v_{tt}(\alpha_1) \cap v_{ff}(\alpha_2))$

Table 3.4: Variables simples libres et exportées par les formules sur actions

Les ensembles $v_{tt}(\alpha)$ et $v_{ff}(\alpha)$ sont définis de manière à assurer que, lors de l'évaluation de α, toutes les variables exportées seront initialisées. Par exemple, lorsqu'une formule "α_1 **or** α_2" est satisfaite par une action a, il est certain que les variables *communes* exportées par α_1 et par α_2 sont initialisées (car au moins une de ces sous-formules est vraie). Pour une formule "α_1 **and** α_2", il est certain que *toutes* les variables exportées par α_1 et par α_2 sont initialisées (car les deux sous-formules sont vraies). Les ensembles $v_{ff}(\alpha)$ sont définis de manière duale.

Il est facile de vérifier (par induction structurelle sur α) que $v_{tt}(\alpha) \cap v_{ff}(\alpha) = \emptyset$ pour toute $\alpha \in AForm$.

3.5.2 Aspects sémantiques

La sémantique des formules sur actions est définie par la fonction suivante :

$$[\![.]\!] : AForm \to \mathbf{DEnv} \to A \to \mathbf{Bool} \times 2^{\mathbf{DEnv}}$$

Etant donné une formule sur actions α, un environnement ε tel que $fdv(\alpha) \subseteq supp(\varepsilon)$ et une action a, la dénotation $[\![\alpha]\!]\,\varepsilon a$ renvoie un tuple ayant deux champs : (1) une valeur booléenne, indiquant si a satisfait α dans le contexte de ε et (2) un ensemble d'environnements simples, chacun initialisant les variables exportées par α avec des valeurs extraites de a.

Remarque 3-5
Le second champ du tuple $[\![\alpha]\!]\,\varepsilon a$ est un ensemble d'environnements simples, et non pas un environnement simple, car l'évaluation d'une formule α sur une action a peut produire plusieurs environnements initialisant les variables exportées par α. Ce non-déterminisme apparaît dans le cas des formules "α_1 **or** α_2" : puisque les deux sous-formules peuvent être simultanément satisfaites par a, il est nécessaire de prendre en compte les affectations des variables $v_{tt}(\alpha_1\ \mathbf{or}\ \alpha_2)$ produites par α_1 aussi bien que celles produites par α_2. Considérons, par exemple, la formule suivante :

```
(G ? x1 : Nat ? x2 : Nat) or (G ? x2 : Nat ? x3 : Nat)
```

interprétée sur l'action "`G 0 1`" dans le contexte d'un environnement vide. Les sous-formules "`G ? x1 : Nat ? x2 : Nat`" et "`G ? x2 : Nat ? x3 : Nat`" produisent respectivement les environnements $[0/\mathtt{x1}, 1/\mathtt{x2}]$ et $[0/\mathtt{x2}, 1/\mathtt{x3}]$. Puisque les deux affectations de la variable `x2` correspondent au fait que la formule est satisfaite par l'action, elles doivent être propagées à l'extérieur ; l'ensemble d'environnements produit sera donc $\{[0/\mathtt{x2}], [1/\mathtt{x2}]\}$. ■

Quelques notions auxiliaires sont nécessaires. Soit un ensemble d'environnements $\mathcal{E} \in 2^{\mathbf{DEnv}}$ tel que tous les environnements contenus dans $\mathcal{E}$ aient le même support, noté, par extension, $supp(\mathcal{E})$. La restriction de $\mathcal{E}$ à un domaine $D \subseteq DVar$, notée $\mathcal{E}|_D$, est définie par $\mathcal{E}|_D \stackrel{\mathrm{d}}{=} \{\varepsilon|_D \mid \varepsilon \in \mathcal{E}\}$. Pour manipuler ces ensembles d'environnements, nous introduisons les opérations binaires $\uplus$ et $\Cap$ sur $2^{\mathbf{DEnv}}$, définies comme suit :

$$\begin{aligned}
\mathcal{E}_1 \uplus \mathcal{E}_2 &\stackrel{\mathrm{d}}{=} \{\varepsilon \mid supp(\varepsilon) = supp(\mathcal{E}_1) \cup supp(\mathcal{E}_2)\ \wedge\ \varepsilon|_{supp(\mathcal{E}_1)} \in \mathcal{E}_1\ \wedge\ \varepsilon|_{supp(\mathcal{E}_2)} \in \mathcal{E}_2\} \\
\mathcal{E}_1 \Cap \mathcal{E}_2 &\stackrel{\mathrm{d}}{=} \{\varepsilon \mid supp(\varepsilon) = supp(\mathcal{E}_1) \cap supp(\mathcal{E}_2)\ \wedge\ \big(\varepsilon \in \mathcal{E}_1|_{supp(\mathcal{E}_2)}\ \vee\ \varepsilon \in \mathcal{E}_2|_{supp(\mathcal{E}_1)}\big)\}
\end{aligned}$$

Les environnements produits par $\mathcal{E}_1 \uplus \mathcal{E}_2$ positionnent toutes les variables de $supp(\mathcal{E}_1)$ et $supp(\mathcal{E}_2)$; chaque environnement dans $\mathcal{E}_1 \uplus \mathcal{E}_2$ positionne les variables appartenant à $supp(\mathcal{E}_1)$ avec les valeurs données par un environnement de $\mathcal{E}_1$ et les variables appartenant à $supp(\mathcal{E}_2)$ avec les valeurs données par un environnement de $\mathcal{E}_2$. Les environnements produits par $\mathcal{E}_1 \Cap \mathcal{E}_2$ positionnent les variables qui sont communes à $supp(\mathcal{E}_1)$ et à $supp(\mathcal{E}_2)$; chaque environnement dans $\mathcal{E}_1 \Cap \mathcal{E}_2$ positionne les variables appartenant à $supp(\mathcal{E}_1)$ et à $supp(\mathcal{E}_2)$ avec les valeurs données par un environnement de $\mathcal{E}_1$ ou par un environnement de $\mathcal{E}_2$.

$\mathcal{E} \uplus \mathcal{E} = \mathcal{E}$	$\mathcal{E} \Cap \mathcal{E} = \mathcal{E}$
$\mathcal{E}_1 \uplus \mathcal{E}_2 = \mathcal{E}_2 \uplus \mathcal{E}_1$	$\mathcal{E}_1 \Cap \mathcal{E}_2 = \mathcal{E}_2 \Cap \mathcal{E}_1$
$(\mathcal{E}_1 \uplus \mathcal{E}_2) \uplus \mathcal{E}_3 = \mathcal{E}_1 \uplus (\mathcal{E}_2 \uplus \mathcal{E}_3)$	$(\mathcal{E}_1 \Cap \mathcal{E}_2) \Cap \mathcal{E}_3 = \mathcal{E}_1 \Cap (\mathcal{E}_2 \Cap \mathcal{E}_3)$
$\mathcal{E} \uplus \{[\,]\} = \mathcal{E}$	$\mathcal{E} \Cap \{[\,]\} = \{[\,]\}$

Table 3.5: Propriétés des opérations $\uplus$ et $\Cap$

La table 3.5 indique les propriétés des opérateurs $\uplus$ et $\Cap$ qui seront utilisées par la suite afin d'établir certaines propriétés des formules sur actions.

La fonction sémantique associée aux formules sur actions est définie inductivement comme suit :

$$
\begin{aligned}
\left[\!\!\left[\begin{matrix} O_0 \ldots O_m \\ [\mathbf{where}\ E] \end{matrix}\right]\!\!\right] \varepsilon a \;\stackrel{\mathrm{d}}{=}\;& \mathit{if}\ a = v_0 \ldots v_m \ \wedge\ \textstyle\bigwedge_{i=0}^{m}((\llbracket O_i \rrbracket\, \varepsilon v_i)_1 = \mathbf{tt}) \\
& \quad [\ \wedge\ \llbracket E \rrbracket\, (\varepsilon \oslash \textstyle\bigoplus_{i=0}^{m}(\llbracket O_i \rrbracket\, \varepsilon v_i)_2 \oslash [a/c\_a]) = \mathbf{tt}] \\
& \mathit{then} \\
& \quad \big(\mathbf{tt}, \{\textstyle\bigoplus_{i=0}^{m}(\llbracket O_i \rrbracket\, \varepsilon v_i)_2\}\big) \\
& \mathit{else} \\
& \quad \big(\mathbf{ff}, \{[\ ]\}\big) \\
& \mathit{endif}
\end{aligned}
$$

$$
\begin{aligned}
\left[\!\!\left[\begin{matrix} O_0 \ldots O_m\ \ldots \\ O_{m+1} \ldots O_{m+n} \\ [\mathbf{where}\ E] \end{matrix}\right]\!\!\right] \varepsilon a \;\stackrel{\mathrm{d}}{=}\;& \mathit{if}\ a = v_0 \ldots v_p \ \wedge\ p \geq m+n \ \wedge\ \textstyle\bigwedge_{i=0}^{m}((\llbracket O_i \rrbracket\, \varepsilon v_i)_1 = \mathbf{tt}) \ \wedge \\
& \quad \textstyle\bigwedge_{j=1}^{n}(\llbracket O_{m+j} \rrbracket\, \varepsilon v_{p-n+j})_1 = \mathbf{tt} \\
& \quad \Big[\wedge \llbracket E \rrbracket\, (\varepsilon \oslash (\textstyle\bigoplus_{i=0}^{m}(\llbracket O_i \rrbracket\, \varepsilon v_i)_2 \oplus \bigoplus_{j=1}^{n}(\llbracket O_{m+j} \rrbracket\, \varepsilon v_{p-n+j})_2) \oslash [a/c\_a]) = \mathbf{tt}\Big] \\
& \mathit{then} \\
& \quad \big(\mathbf{tt}, \{\textstyle\bigoplus_{i=0}^{m}(\llbracket O_i \rrbracket\, \varepsilon v_i)_2 \oplus \bigoplus_{j=1}^{n}(\llbracket O_{m+j} \rrbracket\, \varepsilon v_{p-n+j})_2\}\big) \\
& \mathit{else} \\
& \quad \big(\mathbf{ff}, \{[\ ]\}\big) \\
& \mathit{endif}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \mathbf{true} \rrbracket\, \varepsilon a \;&\stackrel{\mathrm{d}}{=}\; (\mathbf{tt}, \{[\ ]\}) \\
\llbracket \mathbf{false} \rrbracket\, \varepsilon a \;&\stackrel{\mathrm{d}}{=}\; (\mathbf{ff}, \{[\ ]\}) \\
\llbracket \mathbf{not}\ \alpha \rrbracket\, \varepsilon a \;&\stackrel{\mathrm{d}}{=}\; (\mathit{not}((\llbracket \alpha \rrbracket\, \varepsilon a)_1), (\llbracket \alpha \rrbracket\, \varepsilon a)_2) \\
\llbracket \alpha_1\ \mathbf{or}\ \alpha_2 \rrbracket\, \varepsilon a \;&\stackrel{\mathrm{d}}{=}\; \mathit{if}\ (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \wedge\ (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \mathit{then} \\
& \quad (\mathbf{tt}, (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_2 \mathbin{\overset{*}{\cap}} (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_2) \\
& \mathit{elsif}\ (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \mathit{then} \\
& \quad (\mathbf{tt}, (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_2|_{v_{tt}(\alpha_2)}) \\
& \mathit{elsif}\ (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \mathit{then} \\
& \quad (\mathbf{tt}, (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_2|_{v_{tt}(\alpha_1)}) \\
& \mathit{else} \\
& \quad (\mathbf{ff}, (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_2 \mathbin{\overset{*}{\cup}} (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_2) \\
& \mathit{endif} \\
\llbracket \alpha_1\ \mathbf{and}\ \alpha_2 \rrbracket\, \varepsilon a \;&\stackrel{\mathrm{d}}{=}\; \mathit{if}\ (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \wedge\ (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \mathit{then} \\
& \quad (\mathbf{tt}, (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_2 \mathbin{\overset{*}{\cup}} (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_2) \\
& \mathit{elsif}\ (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \mathit{then} \\
& \quad (\mathbf{ff}, (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_2|_{v_{ff}(\alpha_1)}) \\
& \mathit{elsif}\ (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_1 = \mathbf{tt}\ \mathit{then} \\
& \quad (\mathbf{ff}, (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_2|_{v_{ff}(\alpha_2)}) \\
& \mathit{else} \\
& \quad (\mathbf{ff}, (\llbracket \alpha_1 \rrbracket\, \varepsilon a)_2 \mathbin{\overset{*}{\cap}} (\llbracket \alpha_2 \rrbracket\, \varepsilon a)_2) \\
& \mathit{endif}
\end{aligned}
$$

$$\begin{aligned}
[\![\alpha_1 \textbf{ implies } \alpha_2]\!]\,\varepsilon a &\stackrel{\mathrm{d}}{=} [\![\textbf{not } \alpha_1 \textbf{ or } \alpha_2]\!]\,\varepsilon a \\
[\![\alpha_1 \textbf{ iff } \alpha_2]\!]\,\varepsilon a &\stackrel{\mathrm{d}}{=} [\![(\alpha_1 \textbf{ implies } \alpha_2) \textbf{ and } (\alpha_2 \textbf{ implies } \alpha_1)]\!]\,\varepsilon a \\
[\![\alpha_1 \textbf{ xor } \alpha_2]\!]\,\varepsilon a &\stackrel{\mathrm{d}}{=} [\![\textbf{not } (\alpha_1 \textbf{ iff } \alpha_2)]\!]\,\varepsilon a
\end{aligned}$$

Les règles sémantiques associées aux opérateurs "**or**" et "**and**", bien qu'ayant une apparence compliquée, peuvent être facilement interprétées en tenant compte des observations suivantes (où *op* dénote "**or**" ou "**and**") :

- la valeur booléenne $([\![\alpha_1 \; op \; \alpha_2]\!]\,\varepsilon a)_1$ respecte la table de vérité de l'opérateur *op* ;
- l'ensemble d'environnements simples $([\![\alpha_1 \; op \; \alpha_2]\!]\,\varepsilon a)_2$ a le support égal à $v_{tt}(\alpha_1 \; op \; \alpha_2)$ si $([\![\alpha_1 \; op \; \alpha_2]\!]\,\varepsilon a)_1 = \mathbf{tt}$ et à $v_{ff}(\alpha_1 \; op \; \alpha_2)$ sinon ;
- l'ensemble $([\![\alpha_1 \; op \; \alpha_2]\!]\,\varepsilon a)_2$ est synthétisé à partir de la (ou des) sous-formule(s) α_1 et α_2 qui décide(nt) la valeur de vérité de $\alpha_1 \; op \; \alpha_2$.

Une meilleure compréhension de cette sémantique est fournie à travers ses propriétés, étudiées dans les paragraphes suivants. Ces propriétés concernent plusieurs aspects : (1) la traduction des opérateurs dérivés en termes des opérateurs primitifs, (2) la préservation des identités booléennes et (3) l'initialisation correcte des variables exportées.

Traduction des opérateurs dérivés Certains opérateurs des formules XTL sur actions peuvent être exprimés en termes d'autres opérateurs, plus primitifs. La proposition suivante précise ces traductions.

Proposition 3-1 (Traduction des opérateurs dérivés sur actions)
Les identités suivantes sont valides.

true	$\equiv$	**any ...**
false	$\equiv$	**not true**
α_1 **and** α_2	$\equiv$	**not** (**not** α_1 **or not** α_2)

■

Preuve Etant donné $\varepsilon \in \mathbf{DEnv}$ et $a \in A$, il s'agit de montrer que, pour chaque règle de traduction, les formules en partie gauche et droite ont la même sémantique dans le contexte de ε et a. A titre d'exemple, nous justifions la traduction de l'opérateur "**and**" :

$$\begin{array}{ll}
[\![\alpha_1 \textbf{ and } \alpha_2]\!]\,\varepsilon a = & \text{par définition de } [\![\alpha]\!] \\
\textit{if } ([\![\alpha_1]\!]\,\varepsilon a)_1 = \mathbf{tt} \wedge ([\![\alpha_2]\!]\,\varepsilon a)_1 = \mathbf{tt} \textit{ then} & \\
\quad (\mathbf{tt}, ([\![\alpha_1]\!]\,\varepsilon a)_2 \uplus ([\![\alpha_2]\!]\,\varepsilon a)_2) & \\
\textit{elsif } ([\![\alpha_1]\!]\,\varepsilon a)_1 = \mathbf{tt} \textit{ then} & \\
\quad (\mathbf{ff}, ([\![\alpha_2]\!]\,\varepsilon a)_2|_{v_{ff}(\alpha_1)}) & \\
\textit{elsif } ([\![\alpha_2]\!]\,\varepsilon a)_1 = \mathbf{tt} \textit{ then} & \\
\quad (\mathbf{ff}, ([\![\alpha_1]\!]\,\varepsilon a)_2|_{v_{ff}(\alpha_2)}) & \\
\textit{else} & \\
\quad (\mathbf{ff}, ([\![\alpha_1]\!]\,\varepsilon a)_2 \Cap ([\![\alpha_2]\!]\,\varepsilon a)_2) & \\
\textit{endif} = & \text{par définition de } [\![\alpha]\!] \\
& \text{et de } v_{tt}(\alpha), v_{ff}(\alpha)
\end{array}$$

$$\begin{array}{l}
\textit{if } (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_1 = \textbf{ff} \wedge (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_1 = \textbf{ff} \textit{ then} \\
\quad (\textbf{tt}, (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_2 \uplus (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_2) \\
\textit{elsif } (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_1 = \textbf{ff} \textit{ then} \\
\quad (\textbf{ff}, (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_2|_{v_{tt}(\textbf{not } \alpha_1)}) \\
\textit{elsif } (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_1 = \textbf{ff} \textit{ then} \\
\quad (\textbf{ff}, (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_2|_{v_{tt}(\textbf{not } \alpha_2)}) \\
\textit{else} \\
\quad (\textbf{ff}, (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_2 \Cap (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_2) \\
\textit{endif} = \\
\textit{if } (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_1 = \textbf{tt} \wedge (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_1 = \textbf{tt} \textit{ then} \\
\quad (\textbf{ff}, (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_2 \Cap (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_2) \\
\textit{elsif } (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_1 = \textbf{tt} \textit{ then} \\
\quad (\textbf{ff}, (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_2|_{v_{tt}(\textbf{not } \alpha_2)}) \\
\textit{elsif } (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_1 = \textbf{tt} \textit{ then} \\
\quad (\textbf{ff}, (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_2|_{v_{tt}(\textbf{not } \alpha_1)}) \\
\textit{else} \\
\quad (\textbf{tt}, (\llbracket \textbf{not } \alpha_1 \rrbracket \varepsilon a)_2 \uplus (\llbracket \textbf{not } \alpha_2 \rrbracket \varepsilon a)_2) \\
\textit{endif} = \qquad\qquad \text{par définition de } \llbracket \alpha \rrbracket \\
\llbracket \textbf{not } (\textbf{not } \alpha_1 \textbf{ or not } \alpha_2) \rrbracket \varepsilon a
\end{array}$$

□

La proposition 3-1 permet de préciser trois constructions primitives pour les formules XTL sur actions : le filtre d'action "$O_0 \ldots O_m$ [**...**] $O_{m+1} \ldots O_{m+n}$ [**where** E]" et les opérateurs "**not**" et "**or**".

Préservation de la sémantique booléenne Les formules α qui ne contiennent pas de définitions de variables simples préservent toutes les identités de l'algèbre booléenne (c'est-à-dire, les propriétés algébriques des opérateurs "**and**", "**or**" et "**not**"). En général, il n'en est pas de même pour les formules α qui exportent des variables (c'est-à-dire, qui ont des ensembles $v_{tt}(\alpha)$ et $v_{ff}(\alpha)$ non vides). La raison intuitive de ce fait est que les variables exportées par une (sous)formule α peuvent "interagir" avec les autres variables simples définies dans le contexte de α. La table 3.6 présente de façon synthétique les conditions nécessaires et suffisantes sur les variables exportées par les formules sur actions afin de préserver les identités booléennes.

Les identités données dans la table 3.6 signifient que (moyennant les éventuelles conditions sur les variables exportées) les formules en partie gauche et droite ont la même sémantique dans le contexte d'un environnement $\varepsilon \in$ **DEnv** et d'une action $a \in A$ quelconques et que, par conséquent, elles peuvent être librement substituées une avec l'autre. La preuve de ces propriétés est élémentaire (mais assez fastidieuse), ne faisant appel qu'à la définition de la sémantique des formules α et aux propriétés des opérateurs $\uplus$ et $\Cap$ (voir la table 3.5).

Les conditions imposées sur les variables exportées par les formules α sont trivialement respectées pour les formules n'ayant pas de variables ($v_{tt}(\alpha) = v_{ff}(\alpha) = \emptyset$). Ces conditions peuvent paraître restrictives, mais d'après notre expérience, en pratique il est rarement nécessaire d'effectuer des calculs booléens sur les formules α. En outre, la hauteur des formules sur actions ne dépasse généralement pas 2 ou 3, ce qui rend facile pour l'utilisateur la gestion des variables exportées.

Identité booléenne			Conditions
α **or** α	$\equiv$	α	*true*
α **and** α	$\equiv$	α	
α_1 **or** α_2	$\equiv$	α_2 **or** α_1	
α_1 **and** α_2	$\equiv$	α_2 **and** α_1	
$(\alpha_1$ **or** $\alpha_2)$ **or** α_3	$\equiv$	α_1 **or** $(\alpha_2$ **or** $\alpha_3)$	
$(\alpha_1$ **and** $\alpha_2)$ **and** α_3	$\equiv$	α_1 **and** $(\alpha_2$ **and** $\alpha_3)$	
α_1 **or** $(\alpha_2$ **and** $\alpha_3)$	$\equiv$	$(\alpha_1$ **or** $\alpha_2)$ **and** $(\alpha_1$ **or** $\alpha_3)$	$v_{ff}(\alpha_1) \cap (v_{ff}(\alpha_2) \cup v_{ff}(\alpha_3)) = \emptyset$
α_1 **and** $(\alpha_2$ **or** $\alpha_3)$	$\equiv$	$(\alpha_1$ **and** $\alpha_2)$ **or** $(\alpha_1$ **and** $\alpha_3)$	$v_{tt}(\alpha_1) \cap (v_{tt}(\alpha_2) \cup v_{tt}(\alpha_3)) = \emptyset$
α_1 **and** $(\alpha_1$ **or** $\alpha_2)$	$\equiv$	α_1	$v_{ff}(\alpha_1) \cap v_{ff}(\alpha_2) = \emptyset$
α_1 **or** $(\alpha_1$ **and** $\alpha_2)$	$\equiv$	α_1	$v_{tt}(\alpha_1) \cap v_{tt}(\alpha_2) = \emptyset$
α **and false**	$\equiv$	**false**	$v_{tt}(\alpha) = \emptyset$
α **or false**	$\equiv$	α	
α **and true**	$\equiv$	α	$v_{ff}(\alpha) = \emptyset$
α **or true**	$\equiv$	**true**	
α **and not** α	$\equiv$	**false**	$v_{tt}(\alpha) = v_{ff}(\alpha) = \emptyset$
α **or not** α	$\equiv$	**true**	
not not α	$\equiv$	α	*true*

Table 3.6: Préservation des identités booléennes par les formules sur actions

Initialisation des variables exportées Les formules sur actions sont utilisées dans les expressions régulières faisant partie des formules modales, ainsi que dans les opérateurs “**case action**”. La sémantique de ces constructions (voir les sections 3.6.2 et 3.7.2) est définie de telle manière que, lorsqu'une formule α est satisfaite par une action du STE, les variables exportées par α peuvent être utilisées à l'extérieur de α. La proposition suivante garantit que toutes les variables exportées par une formule α seront initialisées.

Proposition 3-2 (Initialisation des variables exportées)
Soit une formule $\alpha \in \mathit{AForm}$. Alors, pour tout environnement $\varepsilon \in \mathbf{DEnv}$ et pour toute action $a \in A$:

$$\begin{array}{rl} \text{i)} & (\llbracket\alpha\rrbracket\,\varepsilon a)_1 = \mathbf{tt} \Rightarrow \forall \varepsilon' \in (\llbracket\alpha\rrbracket\,\varepsilon a)_2 . \mathit{supp}(\varepsilon') = v_{tt}(\alpha) \\ \text{ii)} & (\llbracket\alpha\rrbracket\,\varepsilon a)_1 = \mathbf{ff} \Rightarrow \forall \varepsilon' \in (\llbracket\alpha\rrbracket\,\varepsilon a)_2 . \mathit{supp}(\varepsilon') = v_{ff}(\alpha). \end{array}$$

■

Preuve Par induction structurelle sur α, utilisant les définitions de la fonction sémantique $\llbracket\alpha\rrbracket\,\varepsilon a$ et des ensembles de variables $v_{tt}(\alpha)$, $v_{ff}(\alpha)$. □

En pratique, seulement la condition i) énoncée dans la proposition 3-2 est intéressante, puisque les variables initialisées par les environnements contenus dans $(\llbracket\alpha\rrbracket\,\varepsilon a)_2$ sont utilisées uniquement lorsque a satisfait α (c'est-à-dire, lorsque $(\llbracket\alpha\rrbracket\,\varepsilon a)_1 = \mathbf{tt}$).

3.6 Expressions régulières

Les expressions régulières R (présentées informellement à la section 2.9) sont construites sur le vocabulaire des formules XTL sur actions, ce qui leur permet, à la différence des constructions similaires rencontrées dans la logique PDL, d'extraire les valeurs contenues dans les actions du STE et de les propager le long des séquences de transitions. La sémantique que nous proposons pour les expressions régulières XTL est une extension de la sémantique des expressions régulières classiques (donnée habituellement en termes de relations binaires) avec des informations permettant de modéliser les mécanismes d'extraction et de propagation de valeurs.

3.6.1 Aspects syntaxiques

Les éléments syntaxiques associés aux expressions régulières R sont définis par les fonctions suivantes :

$$fdv, v : RegExp \to 2^{DVar}$$

Etant donné une expression régulière R, la dénotation $fdv(R)$ (*resp.* $v(R)$) renvoie l'ensemble des variables simples libres dans R (*resp.* exportées par R) quand elle est satisfaite par une séquence de transitions du modèle STE. Ces fonctions syntaxiques sont définies inductivement dans la table 3.7.

R	$fdv(R)$	$v(R)$
α	$fdv(\alpha)$	$v_{tt}(\alpha) \cup \{c\_a\}$
$R_1 \,.\, R_2$	$fdv(R_1) \cup (fdv(R_2) \setminus v(R_1))$	$v(R_1) \cup v(R_2)$
$R_1 \mid R_2$	$fdv(R_1) \cup fdv(R_2)$	$v(R_1) \cap v(R_2)$
R^*	$fdv(R)$	$\emptyset$
R^+	$fdv(R)$	$v(R)$

Table 3.7: Variables simples libres et exportées par les expressions régulières

Remarque 3-6
Une expression régulière "R^*" n'exporte pas de variable simple (ni d'action courante c_a), car elle peut être satisfaite par une séquence vide de transitions du STE (et donc il est possible qu'aucune action a ne soit traversée). En revanche, une expression régulière "R^+" exporte les variables initialisées par R, car toute séquence du STE qui satisfait "R^+" contient un suffixe satisfaisant R. ■

3.6.2 Aspects sémantiques

La sémantique des expressions régulières est définie par la fonction suivante :

$$[\![.]\!] : RegExp \to \mathbf{DEnv} \to 2^{S \times S \times 2^{\mathbf{DEnv}}}$$

Etant donné une expression régulière R et un environnement ε tel que $fdv(R) \subseteq supp(\varepsilon)$, la dénotation $[\![R]\!]\,\varepsilon$ renvoie un ensemble de tuples ayant trois champs : (1) l'état de départ et (2) l'état d'arrivée d'une séquence de transitions du STE qui satisfait R dans le contexte de ε ; (3) un ensemble d'environnements initialisant les variables simples exportées par R avec des valeurs extraites des actions appartenant à la séquence respective (la variable spéciale "c_a" étant initialisée avec la dernière action a qui a pu être déterminée). La fonction sémantique est définie inductivement comme suit :

$$\begin{aligned}
[\![\alpha]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} \big\{(s, s_0, \mathcal{E}_0) \mid \exists a \in A.s \xrightarrow{a} s_0 \;\wedge\; ([\![\alpha]\!]\,\varepsilon a)_1 = \mathbf{tt} \;\wedge \\
&\qquad \mathcal{E}_0 = \{\varepsilon_0 \oslash [a/c_a] \mid \varepsilon_0 \in ([\![\alpha]\!]\,\varepsilon a)_2\}\big\} \\
[\![R_1 \,.\, R_2]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} \big\{(s, s_0, \mathcal{E}_0) \mid \exists s_1 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.(s, s_1, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge \\
&\qquad \exists \varepsilon_1 \in \mathcal{E}_1.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s_1, s_2, \mathcal{E}_2) \in [\![R_2]\!]\,(\varepsilon \oslash \varepsilon_1) \;\wedge\; \mathcal{E}_0 = \{\varepsilon_1 \oslash \varepsilon_2 \mid \varepsilon_2 \in \mathcal{E}_2\})\big\} \\
[\![R_1 \mid R_2]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} \big\{(s, s_0, \mathcal{E}_0) \mid \exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge\; \mathcal{E}_0 = \mathcal{E}_1|_{v(R_2)}) \;\vee \\
&\qquad \exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_2) \in [\![R_2]\!]\,\varepsilon \;\wedge\; \mathcal{E}_0 = \mathcal{E}_2|_{v(R_1)})\big\} \\
[\![R^*]\!]\varepsilon &\stackrel{\mathrm{d}}{=} \big\{(s, s_0, \{[\,]\}) \mid \exists k \geq 0.(s, s_0) \in (([\![R]\!]\,\varepsilon)_{1,2})^k\big\} \\
[\![R^+]\!]\varepsilon &\stackrel{\mathrm{d}}{=} \big\{(s, s_0, \mathcal{E}_0) \mid \exists s_1 \in S.\exists k \geq 0.(s, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^k \;\wedge\; (s_1, s_0, \mathcal{E}_0) \in [\![R]\!]\,\varepsilon\big\}
\end{aligned}$$

Propriétés De la même façon que son correspondant utilisé dans les expressions régulières classiques, l'opérateur **+** peut être exprimé en termes des opérateurs $*$ et **.** comme suit :

$$\boxed{R^{+} \quad \equiv \quad R^{*}.\, R}$$

Pour tout environnement $\varepsilon \in \mathbf{DEnv}$, les expressions régulières en partie gauche et droite de l'identité ci-dessus, interprétées dans un contexte quelconque ε, ont la même sémantique :

$$\begin{array}{lr}
[\![R^{*}.\, R]\!]\varepsilon = & \text{par définition de } [\![R]\!] \\
\big\{(s, s_0, \mathcal{E}_0) \mid \exists s_1 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.(s, s_1, \mathcal{E}_1) \in [\![R^{*}]\!]\varepsilon \;\wedge & \\
\quad \exists \varepsilon_1 \in \mathcal{E}_1.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s_1, s_2, \mathcal{E}_2) \in [\![R]\!]\,(\varepsilon \oslash \varepsilon_1) \;\wedge\; \mathcal{E}_0 = \{\varepsilon_1 \oslash \varepsilon_2 \mid \varepsilon_2 \in \mathcal{E}_2\})\big\} = & \\
\big\{(s, s_0, \mathcal{E}_0) \mid \exists s_1 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.(\mathcal{E}_1 = \{[\,]\} \;\wedge\; \exists k \geq 0.(s, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^k) \;\wedge & \\
\quad \exists \varepsilon_1 \in \mathcal{E}_1.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s_1, s_2, \mathcal{E}_2) \in [\![R]\!]\,(\varepsilon \oslash \varepsilon_1) \;\wedge\; \mathcal{E}_0 = \{\varepsilon_1 \oslash \varepsilon_2 \mid \varepsilon_2 \in \mathcal{E}_2\})\big\} = & \\
\big\{(s, s_0, \mathcal{E}_0) \mid \exists s_1 \in S.\exists k \geq 0.(s, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^k) \;\wedge & \\
\quad \exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s_1, s_2, \mathcal{E}_2) \in [\![R]\!]\,\varepsilon \;\wedge\; \mathcal{E}_0 = \{\varepsilon_2 \mid \varepsilon_2 \in \mathcal{E}_2\})\big\} = & \\
\big\{(s, s_0, \mathcal{E}_0) \mid \exists s_1 \in S.\exists k \geq 0.(s, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^k \;\wedge & \\
\quad (s_1, s_0, \mathcal{E}_0) \in [\![R]\!]\,\varepsilon\big\} = & \text{par définition de } [\![R]\!] \\
[\![R^{+}]\!]\varepsilon. &
\end{array}$$

3.7 Formules sur états

Les formules sur états φ (présentées informellement à la section 2.10) permettent d'exprimer des propriétés booléennes, modales et temporelles sur les états du modèle STE. Les opérateurs présents dans les formules XTL sur états généralisent les opérateurs modaux de PDL-Δ et les opérateurs de point fixe du μ-calcul modal standard avec des variables typées. La manipulation des données dans les formules φ est effectuée au moyen des constructions "**let**", "**if**" et "**case**", inspirées des langages de programmation fonctionnels. Par conséquent, la sémantique des formules XTL sur états est une extension naturelle de la sémantique des opérateurs de point fixe du μ-calcul standard et de celle des fonctions récursives des langages de programmation.

3.7.1 Aspects syntaxiques

Les éléments syntaxiques associés aux formules φ sur états sont définis par les fonctions suivantes :

$$\mathit{fdv}, \mathit{bdv}, \mathit{fpv}, \mathit{bpv} : \mathit{SForm} \to 2^{PVar}$$

Etant donné une formule φ, la dénotation $\mathit{fdv}(\varphi)$ (*resp.* $\mathit{bdv}(\varphi)$) renvoie l'ensemble des variables simples libres (*resp.* liées) dans φ. La dénotation $\mathit{fpv}(\varphi)$ (*resp.* $\mathit{bpv}(\varphi)$) renvoie l'ensemble des variables propositionnelles libres (*resp.* liées) dans φ. Ces fonctions syntaxiques sont définies inductivement dans les tables 3.8 et 3.9 (où **bool_op**, σ et **quantif** dénotent respectivement les opérateurs booléens binaires, les opérateurs de point fixe et les quantificateurs).

Comme il a été précisé à la section 2.10.10, pour chaque branche "$P_i^0 \mid \ldots \mid P_i^{n_i}$ [**where** E_i] **->** φ_i" d'une formule "**case**" et pour tous $0 \leq j, k \leq n_i$, $\mathit{bdv}(P_i^j) = \mathit{bdv}(P_i^k)$.

φ	$fdv(\varphi)$	$bdv(\varphi)$
E	$fdv(E)$	$bdv(E)$
Y **(**E_1**,** $\ldots$ **,**E_n**)**	$\bigcup_{i=1}^{n} fdv(E_i)$	$\bigcup_{i=1}^{n} bdv(E_i)$
not φ	$fdv(\varphi)$	$bdv(\varphi)$
φ_1 **bool_op** φ_2	$fdv(\varphi_1) \cup fdv(\varphi_2)$	$bdv(\varphi_1) \cup bdv(\varphi_2)$
$\langle R \rangle\ \varphi$, $[R]\ \varphi$	$fdv(R) \cup (fdv(\varphi) \setminus v(R))$	$v(R) \cup bdv(\varphi)$
@ **(**R**)**	$fdv(R)$	$v(R)$
σY **(**x_1**:**T_1**:=**E_1**,** $\ldots$ **,** x_n**:**T_n**:=**E_n**)** **.** φ	$(fdv(\varphi) \setminus \bigcup_{i=1}^{n}\{x_i\}) \cup$ $\bigcup_{i=1}^{n} fdv(E_i)$	$bdv(\varphi) \cup \bigcup_{i=1}^{n}\{x_i\}$
quantif x_0**:**T_0 [**among** E_0] **,** $\ldots$ **,** x_n**:**T_n [**among** E_n] **in** φ	$(fdv(\varphi) \setminus \bigcup_{i=0}^{n}\{x_i\}) \cup$ $\bigcup_{i=0}^{n} fdv(E_i)$	$bdv(\varphi) \cup$ $\bigcup_{i=0}^{n}\{x_i\} \cup$ $\bigcup_{i=0}^{n} bdv(E_i)$
let x_0**:**T_0**:=**E_0**,** $\ldots$ **,** x_n**:**T_n**:=**E_n **in** φ **endlet**	$(fdv(\varphi) \setminus \bigcup_{i=0}^{n}\{x_i\}) \cup$ $\bigcup_{i=0}^{n} fdv(E_i)$	$bdv(\varphi) \cup$ $\bigcup_{i=0}^{n}\{x_i\} \cup$ $\bigcup_{i=0}^{n} bdv(E_i)$
let **(**x_0^0**:**T_0^0**,** $\ldots$ **,**$x_0^{n_0}$**:**$T_0^{n_0}$**):=**E_0 **,** $\ldots$ **,** **(**x_m^0**:**T_m^0**,** $\ldots$ **,**$x_m^{n_m}$**:**$T_m^{n_m}$**):=**E_m **in** φ **endlet**	$(fdv(\varphi) \setminus \bigcup_{i=0}^{m}\bigcup_{j=0}^{n_i}\{x_i^j\}) \cup$ $\bigcup_{i=0}^{m} fdv(E_i)$	$bdv(\varphi) \cup$ $\bigcup_{i=0}^{m} bdv(E_i) \cup$ $\bigcup_{i=0}^{m}\bigcup_{j=0}^{n_i}\{x_i^j\}$
if E_0 **then** φ_0 **elsif** E_1 **then** φ_1 $\ldots$ **elsif** E_n **then** φ_n [**else** φ_{n+1}] **endif**	$\bigcup_{i=1}^{n+1} fdv(\varphi_i) \cup \bigcup_{j=1}^{n} fdv(E_j)$	$\bigcup_{i=0}^{n+1} bdv(\varphi_i) \cup$ $\bigcup_{j=1}^{n} bdv(E_j)$
case E_0 **in** $P_1^0 \mid \ldots \mid P_1^{n_1}$ [**where** E_1] **->** φ_1 $\ldots$ $\mid P_m^0 \mid \ldots \mid P_m^{n_m}$ [**where** E_m] **->** φ_m [$\mid$ **otherwise** **->** φ_{m+1}] **endcase**	$((\bigcup_{i=1}^{m} fdv(E_i) \cup \bigcup_{j=1}^{m+1} fdv(\varphi_j)) \setminus$ $\bigcup_{k=1}^{m} bdv(P_k^0)) \cup fdv(E_0)$	$bdv(E_0) \cup$ $\bigcup_{i=1}^{m} bdv(E_i) \cup$ $\bigcup_{j=1}^{m+1} bdv(\varphi_j) \cup$ $\bigcup_{k=1}^{m} bdv(P_k^0)$
case action E_0 **in** α_1 [**where** E_1] **->** φ_1 $\ldots$ $\mid \alpha_m$ [**where** E_m] **->** φ_m [$\mid$ **otherwise** **->** φ_{m+1}] **endcase**	$((\bigcup_{i=1}^{m} fdv(E_i) \cup \bigcup_{j=1}^{m+1} fdv(\varphi_j)) \setminus$ $\bigcup_{k=1}^{m} v_{tt}(\alpha_k)) \cup fdv(E_0)$	$bdv(E_0) \cup$ $\bigcup_{i=1}^{m} bdv(E_i) \cup$ $\bigcup_{j=1}^{m+1} bdv(\varphi_j) \cup$ $\bigcup_{k=1}^{m} v_{tt}(\alpha_k)$

Table 3.8: Variables simples libres et liées dans les formules sur états

φ	$fpv(\varphi)$	$bpv(\varphi)$
E	$\emptyset$	$\emptyset$
$Y\ (E_1,\ldots,E_n)$	$\{Y\}$	$\emptyset$
not φ	$fpv(\varphi)$	$bpv(\varphi)$
φ_1 **bool_op** φ_2	$fpv(\varphi_1) \cup fpv(\varphi_2)$	$bpv(\varphi_1) \cup bpv(\varphi_2)$
$\langle R \rangle\ \varphi$, $[R]\ \varphi$	$fpv(\varphi)$	$bpv(\varphi)$
@ (R)	$\emptyset$	$\emptyset$
σY $(x_1$:T_1:=$E_1, \ldots,$ x_n:T_n:=$E_n)$. φ	$fpv(\varphi) \setminus \{Y\}$	$bpv(\varphi) \cup \{Y\}$
quantif x_0:T_0 [**among** E_0] $, \ldots ,$ x_n:T_n [**among** E_n] **in** φ	$fpv(\varphi)$	$bpv(\varphi)$
let x_0:T_0:=$E_0, \ldots,$ x_n:T_n:=E_n **in** φ **endlet**	$fpv(\varphi)$	$bpv(\varphi)$
let $(x_0^0$:$T_0^0, \ldots, x_0^{n_0}$:$T_0^{n_0})$:=E_0 $, \ldots ,$ $(x_m^0$:$T_m^0, \ldots, x_m^{n_m}$:$T_m^{n_m})$:=E_m **in** φ **endlet**	$fpv(\varphi)$	$bpv(\varphi)$
if E_0 **then** φ_0 **elsif** E_1 **then** φ_1 $\ldots$ **elsif** E_n **then** φ_n [**else** φ_{n+1}] **endif**	$\bigcup_{i=0}^{n+1} fpv(\varphi_i)$	$\bigcup_{i=0}^{n+1} bpv(\varphi_i)$
case E_0 **in** P_1^0 | ... | $P_1^{n_1}$ [**where** E_1] -> φ_1 $\ldots$ | P_m^0 | ... | $P_m^{n_m}$ [**where** E_m] -> φ_m [| **otherwise** -> φ_{m+1}] **endcase**	$\bigcup_{i=1}^{m+1} fpv(\varphi_i)$	$\bigcup_{i=1}^{m+1} bpv(\varphi_i)$
case action E_0 **in** α_1 [**where** E_1] -> φ_1 $\ldots$ | α_m [**where** E_m] -> φ_m [| **otherwise** -> φ_{m+1}] **endcase**	$\bigcup_{i=1}^{m+1} fpv(\varphi_i)$	$\bigcup_{i=1}^{m+1} bpv(\varphi_i)$

Table 3.9: Variables propositionnelles libres et liées dans les formules sur états

L'analyse statique, notamment la liaison des variables propositionnelles (voir la section A.5), assure que toutes les variables propositionnelles (libres ou liées) contenues dans les formules φ ont des noms uniques. Ceci garantit, en particulier, qu'une formule φ ne contient pas d'occurrences libres et liées d'une même variable Y.

Une formule φ telle que $fdv(\varphi) = \emptyset$ est dite *fermée d.p.d.v. des variables simples*. Une formule φ telle que $fpv(\varphi) = \emptyset$ est dite *fermée d.p.d.v. des variables propositionnelles*. Une formule φ est dite *fermée* si elle est fermée d.p.d.v. des variables simples et d.p.d.v. des variables propositionnelles.

Remarque 3-7
L'analyse statique (voir la section A.5) garantit que toutes les formules XTL sur états passées comme arguments aux méta-opérateurs d'évaluation "[[...]]" et "|=" sont fermées d.p.d.v. des variables propositionnelles. Par contre, ces formules peuvent contenir des variables simples libres, qui doivent être définies dans le programme XTL. ■

Les formules φ doivent vérifier la condition de monotonie syntaxique mentionnée à la section 2.10.6 : dans chaque sous-formule de point fixe "**mu** $Y(\ldots)$ **.** φ'" ou "**nu** $Y(\ldots)$ **.** φ'" de φ, chaque occurrence de la variable propositionnelle Y dans φ' doit être placée sous un nombre pair de négations (opérateurs "**not**") ou de parties gauches d'implications (opérateurs "**implies**").

La *taille* d'une formule φ est égale au nombre d'opérateurs (booléens, modaux, de point fixe, quantificateurs, "**let**", "**if**" ou "**case**") contenus dans φ. Les formules φ qui ne contiennent aucun opérateur (parmi ceux énumérés ci-dessus) sont dites *atomiques*.

3.7.2 Aspects sémantiques

La sémantique des formules sur états est définie par la fonction suivante :

$$[\![.]\!] : \mathit{SForm} \to \mathbf{PEnv} \to \mathbf{DEnv} \to 2^S$$

Etant donné une formule φ, un environnement ρ tel que $fpv(\varphi) \subseteq supp(\rho)$ et un environnement ε tel que $fdv(\varphi) \subseteq supp(\varepsilon)$, la dénotation $[\![\varphi]\!]\,\rho\varepsilon$ renvoie l'ensemble d'états du STE qui satisfont φ dans le contexte de ρ et de ε. La fonction sémantique est définie inductivement comme suit :

$$\begin{array}{rcl}
[\![E]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & \{s \in S \mid [\![E]\!]\,(\varepsilon \oslash [s/c_s]) = \mathbf{tt}\} \\
[\![Y\ (E_1,\ldots,E_n)]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & \{s \in S \mid s \in (\rho(Y))([\![E_1]\!]\,(\varepsilon \oslash [s/c_s]),\ldots, \\
 & & \qquad [\![E_n]\!]\,(\varepsilon \oslash [s/c_s]))\} \\
[\![\mathbf{not}\ \varphi]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & S \setminus [\![\varphi]\!]\,\rho\varepsilon \\
[\![\varphi_1\ \mathbf{or}\ \varphi_2]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & [\![\varphi_1]\!]\,\rho\varepsilon \cup [\![\varphi_2]\!]\,\rho\varepsilon \\
[\![\varphi_1\ \mathbf{and}\ \varphi_2]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & [\![\varphi_1]\!]\,\rho\varepsilon \cap [\![\varphi_2]\!]\,\rho\varepsilon \\
[\![\varphi_1\ \mathbf{implies}\ \varphi_2]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & [\![\mathbf{not}\ \varphi_1\ \mathbf{or}\ \varphi_2]\!]\,\rho\varepsilon \\
[\![\varphi_1\ \mathbf{iff}\ \varphi_2]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & [\![(\varphi_1\ \mathbf{implies}\ \varphi_2)\ \mathbf{and}\ (\varphi_2\ \mathbf{implies}\ \varphi_1)]\!]\,\rho\varepsilon \\
[\![\varphi_1\ \mathbf{xor}\ \varphi_2]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & [\![\mathbf{not}\ (\varphi_1\ \mathbf{iff}\ \varphi_2)]\!]\,\rho\varepsilon \\
[\![\langle R\rangle\ \varphi]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & \{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}. \\
 & & \quad (s, s_0, \mathcal{E}_0) \in [\![R]\!]\,\varepsilon\ \wedge\ \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)\} \\
[\![[R]\ \varphi]\!]\,\rho\varepsilon & \stackrel{\mathrm{d}}{=} & \{s \in S \mid \forall s_0 \in S.\forall \mathcal{E}_0 \in 2^{\mathbf{DEnv}}. \\
 & & \quad (s, s_0, \mathcal{E}_0) \in [\![R]\!]\,\varepsilon \Rightarrow \forall \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)\}
\end{array}$$

$$\llbracket \textbf{@ } (R) \rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \left\{ s \in S \mid \forall k \geq 0.\exists s_0 \in S.(s, s_0) \in ((\llbracket R \rrbracket \varepsilon)_{1,2})^k \right\}$$

$$\left\llbracket \begin{array}{l} \textbf{mu } Y \ (x_1{:}T_1{:=}E_1, \ldots, \\ \qquad x_n{:}T_n{:=}E_n) \ \textbf{.} \ \varphi \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \left\{ s \in S \mid s \in \left(\bigsqcap \{ F : T_1 \times \cdots \times T_n \to 2^S \mid \Phi_{\rho\varepsilon}(F) \sqsubseteq F \} \right) \right. \\ \qquad \left. (\llbracket E_1 \rrbracket (\varepsilon \oslash [s/c\_s]), \ldots, \llbracket E_n \rrbracket (\varepsilon \oslash [s/c\_s])) \right\} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{nu } Y \ (x_1{:}T_1{:=}E_1, \ldots, \\ \qquad x_n{:}T_n{:=}E_n) \ \textbf{.} \ \varphi \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \left\{ s \in S \mid s \in \left(\bigsqcup \{ F : T_1 \times \cdots \times T_n \to 2^S \mid F \sqsubseteq \Phi_{\rho\varepsilon}(F) \} \right) \right. \\ \qquad \left. (\llbracket E_1 \rrbracket (\varepsilon \oslash [s/c\_s]), \ldots, \llbracket E_n \rrbracket (\varepsilon \oslash [s/c\_s])) \right\} \end{array}$$

où $\Phi_{\rho\varepsilon} : (T_1 \times \cdots \times T_n \to 2^S) \to (T_1 \times \cdots \times T_n \to 2^S)$ est définie par

$$\Phi_{\rho\varepsilon}(F) \stackrel{\mathrm{d}}{=} \lambda v_1{:}T_1, \ldots, v_n{:}T_n. \llbracket \varphi \rrbracket (\rho \oslash [F/Y])(\varepsilon \oslash [v_1/x_1, \ldots, v_n/x_n])$$

$$\left\llbracket \begin{array}{l} \textbf{exists } x_0{:}T_0 \ [\textbf{among } E_0] \\ \qquad\quad , \ldots , \\ \qquad\quad x_n{:}T_n \ [\textbf{among } E_n] \\ \textbf{in } \varphi \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \{ s \in S \mid \exists v_0{:}T_0 \, [\in \llbracket E_0 \rrbracket (\varepsilon \oslash [s/c\_s])] . \ \ldots \\ \qquad\qquad \exists v_n{:}T_n \, [\in \llbracket E_n \rrbracket (\varepsilon \oslash [s/c\_s])] . \\ \qquad\qquad s \in \llbracket \varphi \rrbracket \rho(\varepsilon \oslash [v_0/x_0, \ldots, v_n/x_n]) \\ \} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{forall } x_0{:}T_0 \ [\textbf{among } E_0] \\ \qquad\quad , \ldots , \\ \qquad\quad x_n{:}T_n \ [\textbf{among } E_n] \\ \textbf{in } \varphi \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \{ s \in S \mid \forall v_0{:}T_0 \, [\in \llbracket E_0 \rrbracket (\varepsilon \oslash [s/c\_s])] . \ \ldots \\ \qquad\qquad \forall v_n{:}T_n \, [\in \llbracket E_n \rrbracket (\varepsilon \oslash [s/c\_s])] . \\ \qquad\qquad s \in \llbracket \varphi \rrbracket \rho(\varepsilon \oslash [v_0/x_0, \ldots, v_n/x_n]) \\ \} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{let } x_0{:}T_0{:=}E_0 \\ \qquad , \ldots , \\ \qquad x_n{:}T_n{:=}E_n \ \textbf{in} \\ \quad \varphi \\ \textbf{endlet} \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \{ s \in S \mid s \in \llbracket \varphi \rrbracket \rho(\varepsilon \oslash [\llbracket E_0 \rrbracket (\varepsilon \oslash [s/c\_s])/x_0, \ldots, \\ \qquad\qquad\qquad\qquad \llbracket E_n \rrbracket (\varepsilon \oslash [s/c\_s])/x_n]) \\ \} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{let } (x_0^0{:}T_0^0, \ldots, x_0^{n_0}{:}T_0^{n_0}){:=}E_0 \\ \quad , \ldots , \\ \quad (x_m^0{:}T_m^0, \ldots, x_m^{n_m}{:}T_m^{n_m}){:=}E_m \\ \textbf{in} \\ \quad \varphi \\ \textbf{endlet} \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \{ s \in S \mid s \in \llbracket \varphi \rrbracket \rho(\varepsilon \oslash [(\llbracket E_0 \rrbracket (\varepsilon \oslash [s/c\_s]))_0/x_0^0, \ldots, \\ \qquad\qquad\qquad\qquad (\llbracket E_0 \rrbracket (\varepsilon \oslash [s/c\_s]))_{n_0}/x_0^{n_0}, \ldots, \\ \qquad\qquad\qquad\qquad \ldots \\ \qquad\qquad\qquad\qquad (\llbracket E_m \rrbracket (\varepsilon \oslash [s/c\_s]))_0/x_m^0, \ldots, \\ \qquad\qquad\qquad\qquad (\llbracket E_m \rrbracket (\varepsilon \oslash [s/c\_s]))_{n_m}/x_m^{n_m}]) \\ \} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{if } E_0 \textbf{ then } \varphi_0 \\ \quad \textbf{elsif } E_1 \textbf{ then } \varphi_1 \\ \quad \ldots \\ \quad \textbf{elsif } E_n \textbf{ then } \varphi_n \\ \ [\textbf{else } \varphi_{n+1}] \\ \textbf{endif} \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \{ s \in S \mid \textit{if } \exists i \in [0,n]. \llbracket E_i \rrbracket (\varepsilon \oslash [s/c\_s]) = \mathbf{tt} \land \\ \qquad\qquad \forall k \in [0, i-1]. \llbracket E_k \rrbracket (\varepsilon \oslash [s/c\_s]) = \mathbf{ff} \\ \qquad\quad \textit{then } s \in \llbracket \varphi_i \rrbracket \rho\varepsilon \\ \qquad\quad [\textit{else } s \in \llbracket \varphi_{n+1} \rrbracket \rho\varepsilon \ ] \\ \qquad\quad \textit{endif} \\ \} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{case } E_0 \textbf{ in} \\ \quad P_1^0 \ | \ldots | \ P_1^{n_1} \\ \qquad [\textbf{where } E_1] \ \texttt{->} \ \varphi_1 \\ \quad \ldots \\ | \ P_m^0 \ | \ldots | \ P_m^{n_m} \\ \qquad [\textbf{where } E_m] \ \texttt{->} \ \varphi_m \\ [| \ \textbf{otherwise} \ \texttt{->} \ \varphi_{m+1}] \\ \textbf{endcase} \end{array} \right\rrbracket \rho\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \{ s \in S \mid \textit{let } v_0 : \textit{type}(E_0) := \llbracket E_0 \rrbracket (\varepsilon \oslash [s/c\_s]) \ \textit{in} \\ \qquad\qquad \textit{if } \exists i \in [1,m]. \exists j \in [0, n_i]. ok_i^j \land \\ \qquad\qquad\quad \forall l \in [0, i-1]. \forall k \in [0, n_l]. \neg ok_l^k \\ \qquad\qquad \textit{then } s \in \llbracket \varphi_i \rrbracket \rho(\varepsilon \oslash (\llbracket P_i^j \rrbracket v_0)_2) \\ \qquad\qquad [\textit{else } s \in \llbracket \varphi_{m+1} \rrbracket \rho\varepsilon \ ] \\ \qquad\qquad \textit{endif} \\ \qquad\quad \textit{endlet} \\ \} \end{array}$$

où $ok_i^j \stackrel{\mathrm{d}}{=} (\llbracket P_i^j \rrbracket v_0)_1 = \mathbf{tt} [\ \land \ \llbracket E_i \rrbracket (\varepsilon \oslash (\llbracket P_i^j \rrbracket v_0)_2 \oslash [s/c\_s]) = \mathbf{tt} \ ]$

$$\left[\!\!\left[\begin{array}{l}\textbf{case action } E_0 \textbf{ in}\\ \quad \alpha_1 \ [\textbf{where } E_1] \ \texttt{->} \ \varphi_1\\ \quad \ldots\\ \mid \alpha_m \ [\textbf{where } E_m] \ \texttt{->} \ \varphi_m\\ [\mid \textbf{otherwise} \ \texttt{->} \ \varphi_{m+1}]\\ \textbf{endcase}\end{array}\right]\!\!\right] \rho\varepsilon \ \stackrel{\mathrm{d}}{=} \ \begin{array}{l}\{ s \in S \mid \textit{let } a_0 \in A := [\![E_0]\!]\,(\varepsilon \oslash [s/c\_s]) \textit{ in}\\ \qquad\qquad \textit{if } \exists i \in [1,m].\exists \varepsilon_i \in ([\![\alpha_i]\!]\,\varepsilon a_0)_2.ok_i(\varepsilon_i) \wedge\\ \qquad\qquad\quad \forall j \in [1,i-1].\forall \varepsilon_j \in ([\![\alpha_j]\!]\,\varepsilon a_0)_2.\neg ok_j(\varepsilon_j)\\ \qquad\qquad \textit{then } s \in [\![\varphi_i]\!]\,\rho(\varepsilon \oslash \varepsilon_i)\\ \qquad\qquad [\textit{else } s \in [\![\varphi_{m+1}]\!]\,\rho\varepsilon\ ]\\ \qquad\qquad \textit{endif}\\ \qquad\quad \textit{endlet}\\ \}\end{array}$$

$$\text{où } ok_i(\varepsilon_i) \stackrel{\mathrm{d}}{=} ([\![\alpha_i]\!]\,\varepsilon a_0)_1 = \mathbf{tt}\,[\ \wedge\ [\![E_i]\!]\,(\varepsilon \oslash \varepsilon_i \oslash [s/c\_s]) = \mathbf{tt}\ ]$$

Un STE étendu $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$ satisfait une formule φ dans le contexte de deux environnements $\rho \in \mathbf{PEnv}$ et $\varepsilon \in \mathbf{DEnv}$ ssi $[\![\varphi]\!]\,\rho\varepsilon = S$.

Les propriétés de la sémantique des formules XTL sur états sont étudiées dans les paragraphes suivants. Ces propriétés concernent, d'une part, la traduction des opérateurs dérivés en termes des opérateurs primitifs et, d'autre part, la monotonie des formules.

Traduction des opérateurs dérivés Certains opérateurs des formules XTL sur états peuvent être exprimés en termes d'autres opérateurs, plus primitifs. La proposition suivante précise ces traductions.

Proposition 3-3 (Traduction des opérateurs dérivés sur états)
Les identités suivantes sont valides.

i) prédicats de base :

$$\boxed{E \ \equiv\ \textbf{if } E \textbf{ then true else false endif}}$$

ii) opérateurs booléens :

$$\boxed{\begin{array}{lcl}\textbf{false} & \equiv & \textbf{not true}\\ \varphi_1 \textbf{ and } \varphi_2 & \equiv & \textbf{not } (\textbf{not } \varphi_1 \textbf{ or not } \varphi_2)\end{array}}$$

iii) opérateurs modaux :

$$\boxed{[\,R\,]\ \varphi \ \equiv\ \textbf{not } \langle R \rangle \textbf{ not } \varphi}$$

iv) opérateurs réguliers :

$$\boxed{\begin{array}{lcl}\langle R_1 \ \textbf{.}\ R_2 \rangle\ \varphi & \equiv & \langle R_1 \rangle\, \langle R_2 \rangle\ \varphi\\ \langle R_1 \mid R_2 \rangle\ \varphi & \equiv & \langle R_1 \rangle\ \varphi \textbf{ or } \langle R_2 \rangle\ \varphi\\ \langle R^{*} \rangle\ \varphi & \equiv & \textbf{mu } Y\ \textbf{.}\ (\varphi \textbf{ or } \langle R \rangle\ Y)\\ \textbf{@}\ (R) & \equiv & \textbf{nu } Y\ \textbf{.}\ \langle R \rangle\ Y\end{array}}$$

où Y est supposée différente de toutes les autres variables propositionnelles contenues dans φ.

v) opérateurs de point fixe :

$$\boxed{\begin{array}{l}\textbf{nu } Y\ (x_1{:}T_1{:=}E_1,\\ \qquad \ldots\\ \qquad x_n{:}T_n{:=}E_n)\ \textbf{.}\ \varphi\end{array} \ \equiv\ \begin{array}{l}\textbf{not mu } Y\ (x_1{:}T_1{:=}E_1,\\ \qquad \ldots\\ \qquad x_n{:}T_n{:=}E_n)\ \textbf{. not } \varphi[\textbf{not } Y/Y]\end{array}}$$

où $\varphi\,[\textbf{not } Y/Y]$ représente la substitution syntaxique de Y par $\textbf{not } Y$ dans la formule φ.

vi) quantificateurs :

forall x_0:T_0 [**among** E_0], ... x_n:T_n [**among** E_n] **in** φ	$\equiv$	**not exists** x_0:T_0 [**among** E_0], ... x_n:T_n [**among** E_n] **in not** φ

vii) opérateur "**let**" :

let x_0:T_0:=E_0, ... ,x_n:T_n:=E_n **in** φ **endlet**	$\equiv$	**case** (E_0, ... ,E_n) **in** (x_0:T_0, ... ,x_n:T_n) –> φ **endcase**
let (x_0^0:T_0^0, ... ,$x_0^{n_0}$:$T_0^{n_0}$):=E_0, ... (x_m^0:T_m^0, ... ,$x_m^{n_m}$:$T_m^{n_m}$):=E_m **in** φ **endlet**	$\equiv$	**case** E_0 **in** (x_0^0:T_0^0, ... ,$x_0^{n_0}$:$T_0^{n_0}$) –> ... **case** E_m **in** (x_m^0:T_m^0, ... ,$x_m^{n_m}$:$T_m^{n_m}$) –> φ **endcase** ... **endcase**

viii) opérateur "**if**" :

if E_0 **then** φ_0 **elsif** E_1 **then** φ_1 ... **elsif** E_n **then** φ_n [**else** φ_{n+1}] **endif**	$\equiv$	**case** E_0 **in** **true** –> φ_0 | **otherwise** –> **case** E_1 **in** **true** –> φ_1 | **otherwise** –> ... **case** E_n **in** **true** –> φ_n [| **otherwise** –> φ_{n+1}] **endcase** ... **endcase** **endcase**

■

Preuve Etant donné $\rho \in$ **PEnv** et $\varepsilon \in$ **DEnv**, il s'agit de montrer que, pour chaque règle de traduction énoncée, les formules en partie gauche et droite ont la même sémantique dans le contexte de ρ et ε. Nous justifions ci-dessous les identités iv), qui concernent les opérateurs réguliers. Les autres traductions peuvent être démontrées facilement à partir de la sémantique des formules φ. La preuve des identités iv) est similaire à la traduction des opérateurs de PDL-Δ en μ-calcul standard (voir par exemple [EL86, Bra92]), mais plus élaborée, à cause de la présence des valeurs.

• Traduction de l'opérateur "**.**" :

$$
\begin{array}{ll}
[\![\langle\, R_1 \;\mathbf{.}\; R_2 \,\rangle\; \varphi]\!]\,\rho\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
\big\{s \in S \mid \exists s_2 \in S.\exists \mathcal{E}_2' \in 2^{\mathbf{DEnv}}.(s, s_2, \mathcal{E}_2') \in [\![R_1 \;\mathbf{.}\; R_2]\!]\,\varepsilon \;\wedge & \\
\quad \exists \varepsilon_2' \in \mathcal{E}_2'.s_2 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_2')\big\} = & \text{par définition de } [\![R]\!] \\
\big\{s \in S \mid \exists s_2 \in S.\exists \mathcal{E}_2' \in 2^{\mathbf{DEnv}}.(\exists s_1 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}. & \\
\quad (s, s_1, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge\; \exists \varepsilon_1 \in \mathcal{E}_1.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s_1, s_2, \mathcal{E}_2) \in [\![R_2]\!]\,(\varepsilon \oslash \varepsilon_1) \;\wedge & \\
\quad \mathcal{E}_2' = \{\varepsilon_1 \oslash \varepsilon_2 \mid \varepsilon_2 \in \mathcal{E}_2\}) \;\wedge\; \exists \varepsilon_2' \in \mathcal{E}_2'.s_2 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_2')\big\} = & \\
\big\{s \in S \mid \exists s_1 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.(s, s_1, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge & \\
\quad \exists \varepsilon_1 \in \mathcal{E}_1.(\exists s_2 \in S.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.(s_1, s_2, \mathcal{E}_2) \in [\![R_2]\!]\,(\varepsilon \oslash \varepsilon_1) \;\wedge & \\
\quad \exists \mathcal{E}_2' \in 2^{\mathbf{DEnv}}.(\mathcal{E}_2' = \{\varepsilon_1 \oslash \varepsilon_2 \mid \varepsilon_2 \in \mathcal{E}_2\} \;\wedge\; \exists \varepsilon_2' \in \mathcal{E}_2'.s_2 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_2')))\big\} = & \\
\big\{s \in S \mid \exists s_1 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.(s, s_1, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge & \\
\quad \exists \varepsilon_1 \in \mathcal{E}_1.(\exists s_2 \in S.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.(s_1, s_2, \mathcal{E}_2) \in [\![R_2]\!]\,(\varepsilon \oslash \varepsilon_1) \;\wedge & \\
\quad \exists \varepsilon_2 \in \mathcal{E}_2.s_2 \in [\![\varphi]\!]\,\rho((\varepsilon \oslash \varepsilon_1) \oslash \varepsilon_2))\big\} = & \text{par définition de } [\![\varphi]\!] \\
\big\{s \in S \mid \exists s_1 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.(s, s_1, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge & \text{et associativité de } \oslash \\
\quad \exists \varepsilon_1 \in \mathcal{E}_1.s_1 \in [\![\langle\, R_2 \,\rangle\; \varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_1)\big\} = & \text{par définition de } [\![\varphi]\!] \\
[\![\langle\, R_1 \,\rangle\; \langle\, R_2 \,\rangle\; \varphi]\!]\,\rho\varepsilon &
\end{array}
$$

• Traduction de l'opérateur "|". La liaison des variables simples (voir la section A.4) dans la formule "$\langle\, R_1 \mid R_2 \,\rangle\; \varphi$" assure que les variables exportées par R_1 et R_2 et les variables simples libres dans φ vérifient la condition suivante :

$$v(R_1) \cap \mathit{fdv}(\varphi) = v(R_2) \cap \mathit{fdv}(\varphi) \tag{3.1}$$

La sémantique de la formule "$\langle\, R_1 \mid R_2 \,\rangle\; \varphi$" peut être calculée comme suit :

$$
\begin{array}{ll}
[\![\langle\, R_1 \mid R_2 \,\rangle\; \varphi]\!]\,\rho\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
\big\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(s, s_0, \mathcal{E}_0) \in [\![R_1 \mid R_2]\!]\,\varepsilon \;\wedge & \\
\quad \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)\big\} = & \text{par définition de } [\![R]\!] \\
\big\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}. & \\
\quad (\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge\; \mathcal{E}_0 = \mathcal{E}_1|_{v(R_2)}) \;\vee & \\
\quad \exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_2) \in [\![R_2]\!]\,\varepsilon \;\wedge\; \mathcal{E}_0 = \mathcal{E}_2|_{v(R_1)})) \;\wedge & \\
\quad \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)\big\} = & \\
\big\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge & \\
\quad \exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(\mathcal{E}_0 = \mathcal{E}_1|_{v(R_2)} \;\wedge\; \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0))) \;\vee & \\
\quad \exists s_0 \in S.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_2) \in [\![R_2]\!]\,\varepsilon \;\wedge & \\
\quad \exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(\mathcal{E}_0 = \mathcal{E}_2|_{v(R_1)} \;\wedge\; \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)))\big\} = & \\
\big\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \;\wedge & \\
\quad \exists \varepsilon_1 \in \mathcal{E}_1.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash (\varepsilon_1|_{v(R_2)}))) \;\vee & \\
\quad \exists s_0 \in S.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_2) \in [\![R_2]\!]\,\varepsilon \;\wedge & \\
\quad \exists \varepsilon_2 \in \mathcal{E}_2.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash (\varepsilon_2|_{v(R_1)})))\big\} = & \text{d'après (3.1) et les} \\
& \text{propriétés de } \oslash
\end{array}
$$

$$\begin{array}{ll}
\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_1 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_1) \in [\![R_1]\!]\,\varepsilon \wedge & \\
\quad \exists \varepsilon_1 \in \mathcal{E}_1.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_1)) \vee & \\
\quad \exists s_0 \in S.\exists \mathcal{E}_2 \in 2^{\mathbf{DEnv}}.((s, s_0, \mathcal{E}_2) \in [\![R_2]\!]\,\varepsilon \wedge & \\
\quad \exists \varepsilon_2 \in \mathcal{E}_2.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_2))\} = & \text{par définition de } [\![\varphi]\!] \\
[\![\langle R_1 \rangle\ \varphi]\!]\,\rho\varepsilon \cup [\![\langle R_2 \rangle\ \varphi]\!]\,\rho\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
[\![\langle R_1 \rangle\ \varphi \textbf{ or } \langle R_2 \rangle\ \varphi]\!]\,\rho\varepsilon &
\end{array}$$

- Traduction de l'opérateur "$*$". La sémantique de la formule en partie gauche est égale à :

$$\begin{array}{ll}
[\![\langle R^* \rangle\ \varphi]\!]\rho\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(s, s_0, \mathcal{E}_0) \in [\![R^*]\!]\varepsilon \wedge & \\
\quad \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)\} = & \text{par définition de } [\![R]\!] \\
\{s \in S \mid \exists s_0 \in S.(\exists k \geq 0.(s, s_0) \in (([\![R]\!]\,\varepsilon)_{1,2})^k)\ \wedge\ s_0 \in [\![\varphi]\!]\,\rho\varepsilon\} = & \\
\bigcup_{k \geq 0} \{s \in S \mid \exists s_0 \in S.(s, s_0) \in (([\![R]\!]\,\varepsilon)_{1,2})^k\ \wedge\ s_0 \in [\![\varphi]\!]\,\rho\varepsilon\} \stackrel{\mathrm{d}}{=} \bigcup_{k \geq 0} A_k &
\end{array}$$

La fonctionnelle $\Phi_{\rho\varepsilon} : 2^S \to 2^S$, associée à la formule en partie droite, est calculée comme suit (pour tout $F \in 2^S$) :

$$\begin{array}{ll}
\Phi_{\rho\varepsilon}(F) \stackrel{\mathrm{d}}{=} [\![\varphi \textbf{ or } \langle R \rangle\ Y]\!]\,(\rho \oslash [F/Y])\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
[\![\varphi]\!]\,(\rho \oslash [F/Y])\varepsilon \cup [\![\langle R \rangle\ Y]\!]\,(\rho \oslash [F/Y])\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
[\![\varphi]\!]\,\rho\varepsilon \cup \{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(s, s_0, \mathcal{E}_0) \in [\![R]\!]\,\varepsilon \wedge & \text{et puisque } Y \notin \mathit{fpv}(\varphi) \\
\quad \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![Y]\!]\,(\rho \oslash [F/Y])(\varepsilon \oslash \varepsilon_0)\} = & \text{par définition de } [\![\varphi]\!] \\
[\![\varphi]\!]\,\rho\varepsilon \cup \{s \in S \mid \exists s_0 \in S.(s, s_0) \in ([\![R]\!]\,\varepsilon)_{1,2}\ \wedge\ s_0 \in F\} &
\end{array}$$

Il est facile de voir que $\Phi_{\rho\varepsilon}$ est monotone, c'est-à-dire que pour tous $F_1, F_2 \in 2^S$ tels que $F_1 \subseteq F_2$, on a $\Phi_{\rho\varepsilon}(F_1) \subseteq \Phi_{\rho\varepsilon}(F_2)$. Puisque le treillis 2^S est complet, le théorème de Tarski [Tar55] assure que la sémantique de la formule "**mu** Y **.** $(\varphi$ **or** $\langle R \rangle\ Y)$" (voir la définition de $[\![\varphi]\!]\,\rho\varepsilon$) est égale au plus petit point fixe $\mu\Phi_{\rho\varepsilon}$ de la fonctionnelle $\Phi_{\rho\varepsilon}$. En outre, 2^S étant fini, $\mu\Phi_{\rho\varepsilon}$ a aussi la caractérisation itérative suivante [Kle52] :

$$[\![\textbf{mu } Y \textbf{ . } (\varphi \textbf{ or } \langle R \rangle\ Y)]\!]\,\rho\varepsilon = \mu\Phi_{\rho\varepsilon} = \bigcup_{k \geq 0} \Phi^k_{\rho\varepsilon}(\emptyset) = \bigcup_{k \geq 0} \Phi^{k+1}_{\rho\varepsilon}(\emptyset) \stackrel{\mathrm{d}}{=} \bigcup_{k \geq 0} B_k$$

Nous allons montrer, par induction sur k, que la propriété $P(k) \Leftrightarrow B_k = \cup_{i=0}^k A_i$ est vraie pour tout $k \geq 0$, ce qui implique l'égalité sémantique des formules "$\langle R^* \rangle\ \varphi$" et "**mu** Y **.** $(\varphi$ **or** $\langle R \rangle\ Y)$".

– $P(0)$: $B_0 \stackrel{\mathrm{d}}{=} \Phi_{\rho\varepsilon}(\emptyset) = [\![\varphi]\!]\,\rho\varepsilon = \{s \in S \mid \exists s_0 \in S.s = s_0\ \wedge\ s_0 \in [\![\varphi]\!]\,\rho\varepsilon\} \stackrel{\mathrm{d}}{=} A_0 = \bigcup_{i=0}^0 A_i$.

– $P(k) \Rightarrow P(k+1)$:

$$\begin{array}{l}
B_{k+1} \stackrel{\mathrm{d}}{=} \Phi^{k+2}_{\rho\varepsilon}(\emptyset) = \Phi_{\rho\varepsilon}(\Phi^{k+1}_{\rho\varepsilon}(\emptyset)) = \Phi_{\rho\varepsilon}(B_k) \stackrel{P(k)}{=} \Phi_{\rho\varepsilon}(\bigcup_{i=0}^k A_i) = \\
[\![\varphi]\!]\,\rho\varepsilon \cup \{s \in S \mid \exists s_0 \in S.(s, s_0) \in ([\![R]\!]\,\varepsilon)_{1,2}\ \wedge\ s_0 \in \bigcup_{i=0}^k A_i\} = \\
[\![\varphi]\!]\,\rho\varepsilon \cup \{s \in S \mid \exists s_0 \in S.(s, s_0) \in ([\![R]\!]\,\varepsilon)_{1,2}\ \wedge \\
\qquad\qquad \exists 0 \leq i \leq k.\exists s_1 \in S.((s_0, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^i\ \wedge\ s_1 \in [\![\varphi]\!]\,\rho\varepsilon)\} = \\
[\![\varphi]\!]\,\rho\varepsilon \cup \{s \in S \mid \exists s_1 \in S.\exists s_0 \in S.((s, s_0) \in ([\![R]\!]\,\varepsilon)_{1,2}\ \wedge \\
\qquad\qquad \exists 0 \leq i \leq k.(s_0, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^i)\ \wedge\ s_1 \in [\![\varphi]\!]\,\rho\varepsilon\} = \\
[\![\varphi]\!]\,\rho\varepsilon \cup \{s \in S \mid \exists s_1 \in S.\exists 1 \leq i \leq k.(s, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^i\ \wedge\ s_1 \in [\![\varphi]\!]\,\rho\varepsilon\} \stackrel{\mathrm{d}}{=} \\
A_0 \cup \bigcup_{i=1}^k A_i = \bigcup_{i=0}^k A_i.
\end{array}$$

• Traduction de l'opérateur "@". La sémantique de la formule en partie gauche est égale à :

$$[\![\mathbf{@}\ (R)]\!]\,\rho\varepsilon \stackrel{\mathrm{d}}{=} \{s \in S \mid \forall k \geq 0 . \exists s_0 \in S.(s, s_0) \in (([\![R]\!]\,\varepsilon)_{1,2})^k\} =$$
$$\bigcap_{k \geq 0}\{s \in S \mid \exists s_0 \in S.(s, s_0) \in (([\![R]\!]\,\varepsilon)_{1,2})^k\} \stackrel{\mathrm{d}}{=} \bigcap_{k\geq 0} A_k$$

La fonctionnelle $\Phi_{\rho\varepsilon} : 2^S \to 2^S$, associée à la formule en partie droite, est calculée de la façon suivante (pour tout $F \in 2^S$) :

$$\begin{array}{ll}
\Phi_{\rho\varepsilon}(F) \stackrel{\mathrm{d}}{=} [\![\langle\, R\,\rangle\ Y]\!]\,(\rho \oslash [F/Y])\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
\{s \in S \mid \exists s_0 \in S . \exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(s, s_0, \mathcal{E}_0) \in [\![R]\!]\,\varepsilon\ \wedge & \\
\quad \exists \varepsilon_0 \in \mathcal{E}_0 . s_0 \in [\![Y]\!]\,(\rho \oslash [F/Y])(\varepsilon \oslash \varepsilon_0)\} = & \text{par définition de } [\![\varphi]\!] \\
\{s \in S \mid \exists s_0 \in S.(s, s_0) \in ([\![R]\!]\,\varepsilon)_{1,2}\ \wedge\ s_0 \in F\} &
\end{array}$$

Il est facile de voir que $\Phi_{\rho\varepsilon}$ est monotone, c'est-à-dire que pour tous $F_1, F_2 \in 2^S$ tels que $F_1 \subseteq F_2$, $\Phi_{\rho\varepsilon}(F_1) \subseteq \Phi_{\rho\varepsilon}(F_2)$. Puisque le treillis 2^S est complet, le théorème de Tarski [Tar55] assure que la sémantique de la formule "**nu** Y . $\langle\, R\,\rangle\ Y$" (voir la définition de $[\![\varphi]\!]\,\rho\varepsilon$) est égale au plus grand point fixe $\nu\Phi_{\rho\varepsilon}$ de la fonctionnelle $\Phi_{\rho\varepsilon}$. En outre, 2^S étant fini, $\nu\Phi_{\rho\varepsilon}$ a aussi la caractérisation itérative suivante [Kle52] :

$$[\![\mathbf{nu}\ Y\ .\ \langle\, R\,\rangle\ Y]\!]\,\rho\varepsilon = \nu\Phi_{\rho\varepsilon} = \bigcap_{k\geq 0}\Phi^k_{\rho\varepsilon}(S) \stackrel{\mathrm{d}}{=} \bigcap_{k \geq 0} B_k$$

Nous allons montrer, par induction sur k, que la propriété $P(k) \Leftrightarrow A_k = B_k$ est vraie pour tout $k \geq 0$, ce qui implique l'égalité sémantique des formules "**@** (R)" et "**nu** Y . $\langle\, R\,\rangle\ Y$".

– $P(0)$: $B_0 \stackrel{\mathrm{d}}{=} \Phi^0_{\rho\varepsilon}(S) = S = \{s \in S \mid \exists s_0 \in S . s = s_0\} \stackrel{\mathrm{d}}{=} A_0$.

– $P(k) \Rightarrow P(k+1)$:

$$\begin{array}{ll}
B_{k+1} \stackrel{\mathrm{d}}{=} \Phi^{k+1}_{\rho\varepsilon}(S) = \Phi_{\rho\varepsilon}(\Phi^k_{\rho\varepsilon}(S)) \stackrel{\mathrm{d}}{=} & \\
\{s \in S \mid \exists s_0 \in S.(s, s_0) \in ([\![R]\!]\,\varepsilon)_{1,2}\ \wedge\ s_0 \in \Phi^k_{\rho\varepsilon}(S)\} = & \text{par } P(k) \\
\{s \in S \mid \exists s_0 \in S.(s, s_0) \in ([\![R]\!]\,\varepsilon)_{1,2}\ \wedge\ \exists s_1 \in S.(s_0, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^k\} = & \\
\{s \in S \mid \exists s_1 \in S.(s, s_1) \in (([\![R]\!]\,\varepsilon)_{1,2})^{k+1}\} \stackrel{\mathrm{d}}{=} A_{k+1}. &
\end{array}$$

□

La proposition 3-3 permet de préciser huit opérateurs primitifs pour les formules XTL sur états : "**true**", "**not**", "**or**", "$\langle\ \rangle$", "**mu**", "**exists**", "**case**" et "**case action**".

Monotonie des formules sur états La sémantique des opérateurs de point fixe, telle qu'elle a été définie, n'offre (pour l'instant) aucune indication quant à la correspondance avec des points fixes. Il est donc nécessaire de s'assurer que les ensembles d'états satisfaisant les formules "**mu**" (*resp.* "**nu**") correspondent effectivement aux plus petits (*resp.* plus grands) points fixes des fonctionnelles associées aux formules respectives. Ceci est garanti par la proposition suivante.

Proposition 3-4 (Monotonie des formules sur états)
Soit $\sigma Y(x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)\ .\ \varphi \in \mathit{SForm}$ ($\sigma \in \{\mathbf{mu}, \mathbf{nu}\}$) et $F_1, F_2 : T_1 \times \cdots \times T_n \to 2^S$. Alors, pour tout $\rho \in \mathbf{PEnv}$ et $\varepsilon \in \mathbf{DEnv}$:

$$F_1 \sqsubseteq F_2 \Rightarrow [\![\varphi]\!]\,(\rho \oslash [F_1/Y])\varepsilon \subseteq [\![\varphi]\!]\,(\rho \oslash [F_2/Y])\varepsilon.$$

■

Preuve Par induction structurelle sur φ, en utilisant la définition de la fonction d'interprétation $[\![\varphi]\!]\,\rho\varepsilon$ et en tenant compte de la monotonie syntaxique de φ. $\square$

La proposition 3-4 permet d'établir la monotonie des fonctionnelles $\Phi_{\rho\varepsilon} : (T_1\times\cdots\times T_n \to 2^S) \to (T_1\times\cdots\times T_n \to 2^S)$ associées aux formules "σY $(x_1$**:**T_1**:=**E_1**,** $\ldots$ **,**x_n**:**T_n**:=**$E_n)$ **.** φ" (où σ dénote "**mu**" ou "**nu**"). En effet, pour toutes les fonctions $F_1, F_2 : T_1\times\cdots\times T_n \to 2^S$ telles que $F_1 \sqsubseteq F_2$ et pour toutes les valeurs $v_1 \in T_1, \ldots, v_n \in T_n$:

$$\begin{array}{ll}
(\Phi_{\rho\varepsilon}(F_1))(v_1,\ldots,v_n) = & \text{par définition de } \Phi_{\rho\varepsilon} \\
[\![\varphi]\!]\,(\rho \oslash [F_1/Y])(\varepsilon \oslash [v_1/x_1,\ldots,v_n/x_n]) \subseteq & \text{par proposition 3-4} \\
[\![\varphi]\!]\,(\rho \oslash [F_2/Y])(\varepsilon \oslash [v_1/x_1,\ldots,v_n/x_n]) = & \text{par définition de } \Phi_{\rho\varepsilon} \\
(\Phi_{\rho\varepsilon}(F_2))(v_1,\ldots,v_n) &
\end{array}$$

ce qui implique (moyennant la définition de "$\sqsubseteq$" sur $T_1\times\cdots\times T_n \to 2^S$) la monotonie de $\Phi_{\rho\varepsilon}$:

$$\forall F_1, F_2 : T_1\times\cdots\times T_n \to 2^S . F_1 \sqsubseteq F_2 \Rightarrow \Phi_{\rho\varepsilon}(F_1) \sqsubseteq \Phi_{\rho\varepsilon}(F_2)$$

Chaque domaine $\langle T_1\times\cdots\times T_n \to 2^S, \sqcup, \sqcap, \sqsubseteq\rangle$ étant un treillis complet, le théorème de Tarski [Tar55] assure l'existence et l'unicité du plus petit et du plus grand point fixe de $\Phi_{\rho\varepsilon}$, qui sont donnés par :

$$\begin{array}{rcl}
\mu\Phi_{\rho\varepsilon} & = & \bigsqcap\{F : T_1\times\cdots\times T_n \to 2^S \mid \Phi_{\rho\varepsilon}(F) \sqsubseteq F\} \\
\nu\Phi_{\rho\varepsilon} & = & \bigsqcup\{F : T_1\times\cdots\times T_n \to 2^S \mid F \sqsubseteq \Phi_{\rho\varepsilon}(F)\}
\end{array}$$

Par conséquent, la sémantique des opérateurs "**mu**" et "**nu**" définie précédemment est égale à :

$$\begin{array}{rcl}
[\![\mathbf{mu}\ Y\ (x_1\mathbf{:}T_1\mathbf{:=}E_1\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n\mathbf{:=}E_n)\ \mathbf{.}\ \varphi]\!]\,\rho\varepsilon & = & \{s \in S \mid s \in (\mu\Phi_{\rho\varepsilon})([\![E_1]\!]\,(\varepsilon \oslash [s/c_s]),\ldots, \\
& & [\![E_n]\!]\,(\varepsilon \oslash [s/c_s]))\} \\
[\![\mathbf{nu}\ Y\ (x_1\mathbf{:}T_1\mathbf{:=}E_1\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n\mathbf{:=}E_n)\ \mathbf{.}\ \varphi]\!]\,\rho\varepsilon & = & \{s \in S \mid s \in (\nu\Phi_{\rho\varepsilon})([\![E_1]\!]\,(\varepsilon \oslash [s/c_s]),\ldots, \\
& & [\![E_n]\!]\,(\varepsilon \oslash [s/c_s]))\}
\end{array}$$

ce qui assure que ces opérateurs dénotent effectivement le plus petit et le plus grand point fixe des fonctionnelles correspondantes $\Phi_{\rho\varepsilon}$.

3.8 Transformations préliminaires sur les formules

Dans les sections précédentes, nous avons défini la sémantique de tous les opérateurs contenus dans les formules XTL. Dans le but de faciliter leur évaluation, il est utile de traduire les formules sous une forme plus simple, comportant un nombre réduit d'opérateurs. Les différentes phases de traduction effectuées sur les formules α et φ, ainsi que les opérateurs contenus dans les formules XTL après chaque phase, sont illustrées dans la figure 3.1. Les paragraphes suivants décrivent brièvement chacune de ces phases de traduction.

Elimination des opérateurs dérivés sur actions En utilisant les règles de traduction énoncées à la proposition 3-1, il est possible d'éliminer complètement les opérateurs dérivés sur actions ("**true**", "**false**", "**and**", "**implies**", "**iff**", "**xor**") contenus dans une formule φ : il suffit d'appliquer ces règles sur les sous-formules α de φ autant que possible. Après cette phase de traduction, les sous-formules α de φ ne contiendront que les trois opérateurs primitifs "$O_0 \ldots O_m$ **[...]** $O_{m+1} \ldots O_{m+n}$ [**where** E]", "**not**" et "**or**". En pratique, cette traduction peut être implémentée par un parcours en profondeur des formules φ.

Elimination des opérateurs dérivés sur états En utilisant les règles de traduction énoncées à la proposition 3-3, il est possible d'éliminer les opérateurs dérivés sur états (E, "**and**", "**implies**", "**iff**", "**xor**", "[ ]", "**@**", "**nu**", "**forall**", "**let**", "**if**") contenus dans une formule φ : il suffit d'appliquer ces règles sur les sous-formules de φ autant que possible. Après cette phase de traduction, φ ne contiendra que les huit opérateurs primitifs "**true**", "**not**", "**or**", "$\langle\ \rangle$", "**mu**", "**exists**", "**case**" et "**case action**". En pratique, cette traduction peut être implémentée par un parcours en profondeur des formules φ.

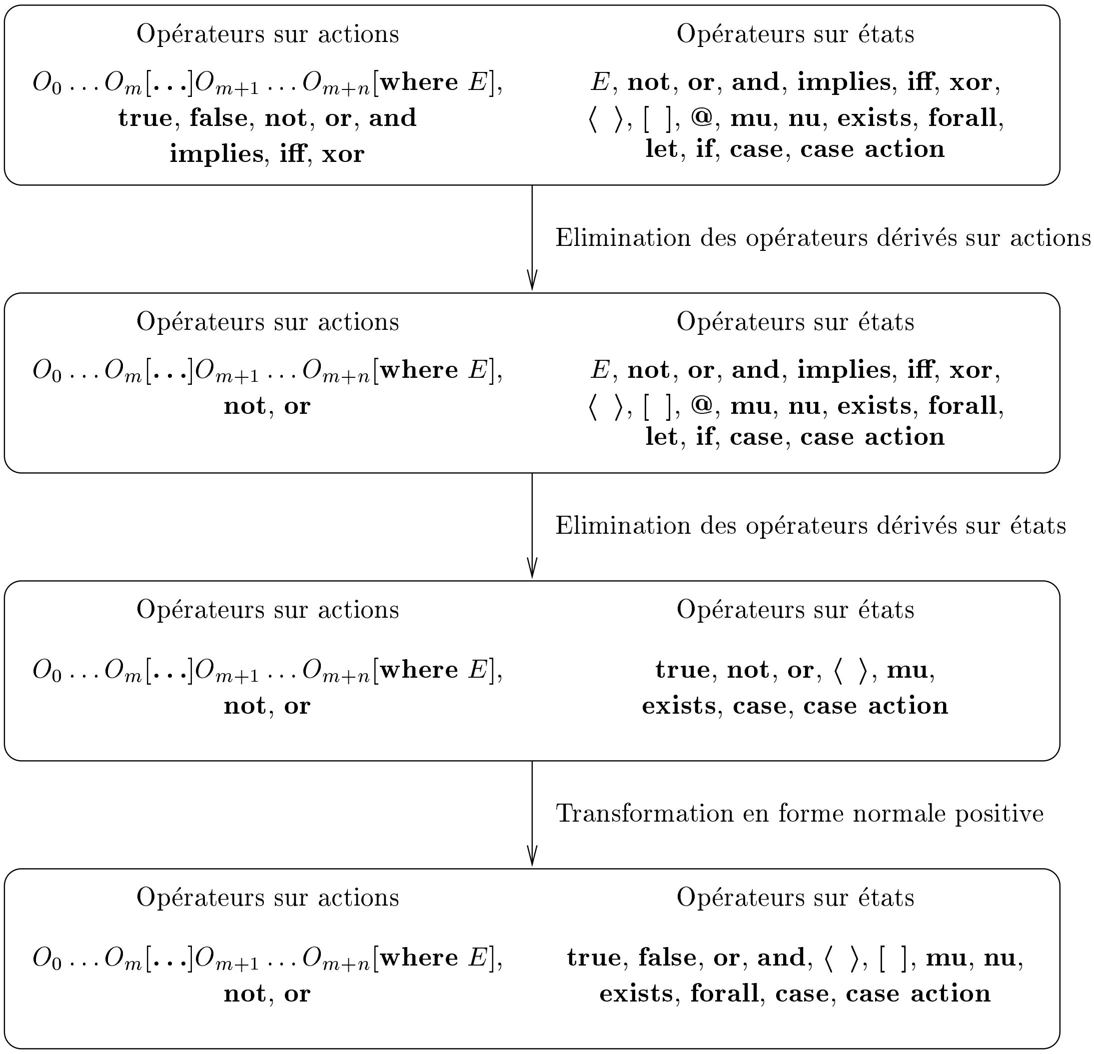

Figure 3.1: Transformations préliminaires sur les formules

Etant donné qu'après cette traduction les expressions régulières ont été éliminées, les opérateurs modaux ne contiendront que des formules sur actions α. Leur sémantique (définie à la section 3.7.2) peut donc être simplifiée en conséquence :

$$
\begin{array}{ll}
[\![\langle\, \alpha \,\rangle\ \varphi]\!]\,\rho\varepsilon = & \text{par définition de } [\![\varphi]\!] \\
\big\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(s, s_0, \mathcal{E}_0) \in [\![\alpha]\!]\,\varepsilon\ \wedge & \\
\quad \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)\big\} = & \text{par définition de } [\![R]\!] \\
\big\{s \in S \mid \exists s_0 \in S.\exists \mathcal{E}_0 \in 2^{\mathbf{DEnv}}.(\exists a \in A.s \xrightarrow{a} s_0\ \wedge & \\
\quad ([\![\alpha]\!]\,\varepsilon a)_1 = \mathbf{tt}\ \wedge\ \mathcal{E}_0 = \{\varepsilon_1 \oslash [a/c\_a] \mid \varepsilon_1 \in ([\![\alpha]\!]\,\varepsilon a)_2\})\ \wedge & \\
\quad \exists \varepsilon_0 \in \mathcal{E}_0.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0)\big\} = & \\
\big\{s \in S \mid \exists s_0 \in S.\exists a \in A.s \xrightarrow{a} s_0\ \wedge\ ([\![\alpha]\!]\,\varepsilon a)_1 = \mathbf{tt}\ \wedge & \\
\quad \exists \varepsilon_0 \in ([\![\alpha]\!]\,\varepsilon a)_2.s_0 \in [\![\varphi]\!]\,\rho(\varepsilon \oslash \varepsilon_0 \oslash [a/c\_a])\big\}. &
\end{array}
$$

Dans la suite du document, nous utiliserons cette sémantique simplifiée pour les formules modales.

Transformation en forme normale positive Comme il a été mentionné à la section 3.7.1, les formules φ vérifient la condition de monotonie syntaxique, qui assure la monotonie des fonctionnelles $\Phi_{\rho\varepsilon}$ associées aux formules de point fixe. Il est possible de montrer (par induction structurelle sur φ) que l'élimination des opérateurs dérivés dans une formule φ préserve la monotonie syntaxique de φ.

De la même manière que pour le μ-calcul standard (voir par exemple [EL86, CKS92]), la monotonie syntaxique permet de transformer les formules φ fermées d.p.d.v. des variables propositionnelles en *forme normale positive* (FNP), c'est-à-dire d'éliminer toutes les occurrences de l'opérateur "**not**" sur états contenues dans φ. Pour cela, il suffit de propager les négations "en bas", jusqu'aux sous-formules atomiques de φ, en utilisant les identités énoncées dans la proposition 3-3 (la taille de la formule ainsi obtenue ne peut pas excéder 2 fois la taille de φ). La monotonie syntaxique et le fait que φ soit fermée d.p.d.v. propositionnel (donc que tout appel de variable propositionnelle Y soit contenu dans une sous-formule de point fixe définissant Y) assurent que φ ne contiendra pas de sous-formules de la forme "**not** $Y(E_1, \ldots, E_n)$". En pratique, la traduction en FNP peut être implémentée par un parcours en profondeur des formules φ.

Une formule φ traduite en FNP contient, en plus des opérateurs primitifs sur états, les opérateurs "**and**", "[ ]", "**nu**" et "**forall**", qui sont respectivement les duaux de "**or**", "⟨ ⟩", "**mu**" et "**exists**". Dans la suite du document, nous ne considérerons que des formules φ en FNP.

Chapitre 4

Algorithmes d'évaluation

Le langage XTL permet d'exprimer les propriétés temporelles au moyen de formules sur états et sur actions, présentées au chapitre 2 et définies formellement au chapitre 3. XTL offre également la possibilité de vérifier des propriétés sur un modèle, à l'aide des méta-opérateurs "$[[\ \ldots\ ]]$" et "$|=$", qui permettent respectivement de calculer l'ensemble d'états (ou actions) satisfaisant une formule et de tester si un certain état (ou une certaine action) satisfait une formule.

Ce chapitre est consacré aux algorithmes d'évaluation des formules XTL sur un modèle STE étendu. Nous commençons par préciser le principe d'évaluation des formules XTL de point fixe, qui combine l'évaluation des appels de fonction (avec passage de paramètres par valeur) dans les langages de programmation fonctionnels et le calcul itératif des formules de point fixe du μ-calcul standard.

Nous abordons ensuite le problème de l'évaluation des formules XTL d'alternance 1 (c'est-à-dire sans opérateurs de plus petit et de plus grand point fixe mutuellement récursifs), qui réalisent un compromis pratique entre la puissance d'expression et l'efficacité d'évaluation. Généralisant les approches utilisées dans la littérature pour le fragment correspondant du μ-calcul standard, nous effectuons l'évaluation d'une formule XTL d'alternance 1 par étapes successives, réduisant le problème à la résolution d'un système d'équations booléennes paramétrées par des variables typées. Nous proposons une méthode de résolution semi-décidable de ces systèmes qui, sous des conditions suffisantes de terminaison, conduit à des algorithmes d'évaluation globale ou locale. Pour les formules XTL d'alternance quelconque, nous proposons une méthode d'évaluation globale qui, toujours sous l'hypothèse de terminaison, permet de calculer itérativement l'ensemble d'états satisfaisant une formule.

Finalement, nous précisons quelques aspects concernant l'implémentation des méta-opérateurs "$[[\ \ldots\ ]]$" et "$|=$" au moyen des algorithmes d'évaluation exposés précédemment.

4.1 Principe de l'évaluation des formules de point fixe

La sémantique des formules XTL de point fixe (voir la section 3.7.2) constitue une généralisation naturelle de la sémantique des formules de point fixe du μ-calcul standard et de celle des fonctions récursives des langages fonctionnels. Par conséquent, la méthode d'évaluation que nous proposons pour les formules XTL s'inspire, d'une part, des méthodes itératives d'évaluation des formules du μ-calcul et, d'autre part, des méthodes d'évaluation des appels de fonction avec passage de paramètres par valeur (*value-passing*) employées dans les langages fonctionnels. Nous illustrons le principe de la méthode au moyen de deux exemples : une fonction récursive et une formule de point fixe.

Exemple 4-1
Considérons une fonction récursive `F : Nat -> Nat` qui, appliquée sur un nombre naturel `n`, calcule le terme d'indice `n` de la suite de Fibonacci :

```
function F (n : Nat) : Nat is
    if n = 0 or n = 1 then
        1
    else
        F (n - 1) + F (n - 2)
    endif
endfunc
```

Pour calculer, par exemple, `F(5)`, il est nécessaire de calculer d'abord `F(4)` et `F(3)`, qui à leur tour dépendent de `F(3)` et `F(2)`, et ainsi de suite. En d'autres termes, le calcul de `F(5)` est équivalent à la résolution du système d'équations suivant :

```
F(5) = F(4) + F(3)
F(4) = F(3) + F(2)
F(3) = F(2) + F(1)
F(2) = F(1) + F(0)
F(1) = 1
F(0) = 1
```

Nous appelons `F(i)` une *instance* de la fonction `F` avec l'argument `i`. Pour chaque instance `F(i)` à calculer, nous définissons une équation ayant la partie droite égale au corps de `F` partiellement évalué avec la valeur `i` substituée au paramètre `n` ; ceci peut générer de nouvelles instances `F(j)` qui restent à calculer, pour lesquelles le processus sera répété. La construction du système d'équations contenant les instances nécessaires au calcul de `F(i)` est appelée *dépliage* (*unfolding*) de l'instance `F(i)`.

Un système d'équations généré par dépliage d'une instance `F(k)` induit un graphe de dépendance entre les instances, ayant un sommet pour chaque instance `F(i)` et un arc de `F(i)` à `F(j)` ssi `F(j)` apparaît en partie droite de l'équation associée à `F(i)`. La figure 4.1 montre le graphe de dépendance correspondant au système obtenu par dépliage de `F(5)`. Les graphes orientés produits par le dépliage des instances $F(v_1, \ldots, v_n)$ de fonctions récursives F avec les arguments $v_1, \ldots, v_n$ sont des graphes connexes acycliques (DAGs) ayant une seule racine (en l'occurrence, le sommet correspondant à l'instance $F(v_1, \ldots, v_n)$) à partir de laquelle tous les sommets sont atteignables. Normalement, ces graphes ne contiennent aucun circuit : en effet, la présence d'un chemin $F(w_1, \ldots, w_n) \to \cdots \to F(w_1, \ldots, w_n)$ signifie que l'évaluation de l'appel de F avec les arguments $w_1, \ldots, w_n$ ne termine pas.

$$\begin{tikzcd}
\texttt{F(5)} \arrow[rr] \arrow[dr] & & \texttt{F(3)} \arrow[rr] \arrow[dr] & & \texttt{F(1)} & \\
& \texttt{F(4)} \arrow[ur] \arrow[rr] & & \texttt{F(2)} \arrow[ur] \arrow[rr] & & \texttt{F(0)}
\end{tikzcd}$$

Figure 4.1: Graphe des instances produit par dépliage de `F(5)`

Puisque les systèmes d'équations générés par dépliage des instances de fonctions récursives ne contiennent pas de dépendances cycliques, ils peuvent être résolus simplement par substitution. En termes du graphe de dépendance entre instances, ceci revient à effectuer un parcours en profondeur, en calculant la valeur de chaque instance lorsque toutes ses instances successeurs ont été calculées. Cette méthode d'évaluation (appel par valeur) est couramment utilisée : le parcours en profondeur du graphe de dépendance est implémenté habituellement à l'aide d'une pile qui mémorise les instances de la fonction visitées depuis l'instance racine à calculer [ASU86]. ■

Exemple 4-2
Considérons maintenant une formule XTL de point fixe exprimant le fait qu'à partir de l'état courant, toutes les séquences d'exécution contiennent une alternance stricte d'actions `SEND` et `RECV`, commençant par un `SEND` (voir aussi l'exemple 2-54) :

```
nu Y (b : Bool = true) . (
    [ SEND ] (b and Y (false)) and
    [ RECV ] (not b and Y (true)) and
    [ not (SEND or RECV) ] Y (b)
)
```

Le paramètre booléen `b` est égal à `true` (*resp.* à `false`) selon qu'une action `SEND` (*resp.* `RECV`) est attendue sur le chemin courant d'exécution. L'évaluation de cette formule nécessite le calcul des états du STE satisfaisant `Y(true)`, qui dépendent des états satisfaisant `Y(false)`, et ainsi de suite. En d'autres termes, ceci revient à calculer la plus grande solution du système d'équations suivant :

```
Y (true)  = [ SEND ] Y (false) and [ RECV ] false and
            [ not (SEND or RECV) ] Y (true)
Y (false) = [ SEND ] false and [ RECV ] Y (true) and
            [ not (SEND or RECV) ] Y (false)
```

Par analogie avec les fonctions, nous appelons `Y(b)` l'instance de `Y` avec l'argument `b`. Le système ci-dessus est obtenu par dépliage de `Y(true)` ; le graphe de dépendance induit entre les instances de `Y` est illustré par la figure 4.2.

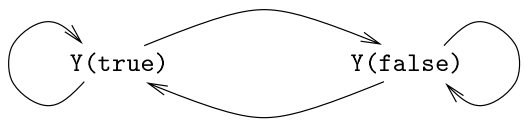

Figure 4.2: Graphe des instances produit par dépliage de `Y(true)`

Les graphes orientés produits par dépliage des instances $Y(v_1, \ldots, v_n)$ de formules XTL de point fixe sont des graphes connexes ayant (au moins) une racine (en l'occurrence, le sommet correspondant à l'instance $Y(v_1, \ldots, v_n)$) à partir de laquelle tous les sommets sont atteignables. Cependant, il existe une différence essentielle avec les graphes produits par dépliage des instances de fonctions récursives : les graphes correspondant aux formules de point fixe peuvent contenir des circuits (comme c'est le cas de la figure 4.2). Par conséquent, les systèmes d'équations obtenus par dépliage des formules de point fixe ne peuvent pas être résolus comme dans l'appel par valeur : il faut appliquer des méthodes itératives d'approximation des points fixes, comme celles utilisées, par exemple, pour le μ-calcul standard [EL86, LBC+94]. ■

Naturellement, la méthode d'évaluation illustrée dans l'exemple 4-2 repose sur le fait que l'ensemble d'instances obtenues par dépliage des formules de point fixe est fini (on dit alors que les formules *convergent*). Ceci peut ne pas être toujours le cas, comme le montre la formule suivante[11] :

```
mu Y (n : Nat := 1) . ((n = 0) or < SEND > Y (n + 1))
```

L'évaluation de cette formule nécessite le dépliage de `Y(1)`, qui boucle indéfiniment, générant l'ensemble infini d'instances {`Y(1)`,`Y(2)`,...}. De toute évidence, le problème de la terminaison du dépliage est indécidable (tout comme la terminaison des fonctions récursives). La méthode d'évaluation des formules XTL de point fixe est donc semi-décidable : si le dépliage de l'instance $Y(v_1, \ldots, v_n)$ à calculer termine, alors la sémantique de la formule respective peut être évaluée sur un

[11] Cet exemple est assez artificiel, car le paramètre `n` n'a aucune "liaison" avec les valeurs contenues dans le STE ; c'est souvent le cas pour les formules de point fixe qui ne convergent pas.

modèle STE fini. En revanche, l'évaluation d'une formule de point fixe qui ne converge pas s'arrêtera (avec un message d'erreur approprié) lorsque les instances produites par dépliage auront épuisé tout l'espace mémoire disponible sur la machine hôte.

Bien entendu, les diverses conditions suffisantes de terminaison exhibées pour les fonctions récursives ou pour les règles de réécriture [Der95] sont aussi valables pour les formules de point fixe paramétrées. Cependant, les conditions suffisantes pour la convergence des formules XTL de point fixe semblent moins restrictives que les conditions correspondantes pour les fonctions récursives, puisque le bouclage (*looping*) est autorisé : une instance $Y(v_1, \ldots, v_n)$ peut dépendre (directement ou transitivement) d'elle-même, sans pour autant entraîner la non-convergence de la formule de point fixe (comme c'est le cas pour la formule de l'exemple 4-2). Par contre, dans le cas d'une fonction récursive, un bouclage aurait entraîné la non-terminaison. Cette dernière remarque s'avère importante en pratique, car elle signifie que le fait d'écrire une formule XTL qui converge est "plus facile" que d'écrire une fonction récursive qui termine.

4.2 Evaluation des formules d'alternance 1

Le fragment du μ-calcul standard d'alternance 1 a bénéficié d'une attention considérable dans la littérature. De nombreux algorithmes d'évaluation (globale ou locale) ont été proposés pour cette logique et différents outils de vérification associés ont été développés. Ce fragment est intéressant pour une double raison. D'une part, il est assez expressif pour décrire des propriétés de sûreté et de vivacité ; en particulier, les opérateurs de plusieurs logiques temporelles largement utilisées, comme CTL, ACTL ou PDL-Δ, peuvent être exprimés comme formules du μ-calcul d'alternance 1 (voir la section 1.2.4). D'autre part, les algorithmes d'évaluation associés ont une complexité linéaire en taille du modèle (nombre d'états et de transitions) et de la formule (nombre d'opérateurs), ce qui permet d'évaluer les logiques temporelles mentionnées, par traduction vers le μ-calcul d'alternance 1, avec la même efficacité que celle de leurs algorithmes particuliers d'évaluation (voir la section 1.3).

A la différence du μ-calcul standard d'alternance 1, le fragment correspondant des formules XTL permet aussi de décrire, outre des propriétés portant sur les valeurs, certaines propriétés d'équité (voir les sections 2.10.6 et C.4), ce qui justifie d'autant plus son intérêt pratique.

Cette section est consacrée aux algorithmes d'évaluation des formules XTL d'alternance 1 sur un modèle STE étendu $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$. Dans la présentation, nous supposons que les phases d'analyse statique (voir l'annexe A.5) et de transformation préliminaire des formules φ (voir la section 3.8) ont été effectuées. En particulier, l'attribut $sign(Y) \in \{\mu, \nu\}$, indiquant le signe de Y, a été calculé pour chaque variable propositionnelle $Y \in PVar$. Les formules φ considérées sont en forme normale positive (FNP) et sont fermées d.p.d.v. des variables propositionnelles (voir la remarque 3-7). Le symbole σ est utilisé pour dénoter "**mu**" ou "**nu**".

Nous commençons par définir le fragment des formules XTL d'alternance 1.

Définition 4-1 (Formules XTL d'alternance 1)
Une formule $\varphi \in SForm$ est d'alternance 1 ssi chaque sous-formule $\sigma Y(x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n).\varphi'$ de φ satisfait la propriété suivante :

$$\forall Y' \in fpv(\varphi').sign(Y') = \sigma.$$

■

Intuitivement, cette définition exprime l'absence de récursion mutuelle entre des variables propositionnelles de plus petit et de plus grand point fixe (par contre, la récursion mutuelle entre des variables de même signe est autorisée).

L'approche que nous proposons pour l'évaluation des formules XTL d'alternance 1 généralise l'approche utilisée dans [AC88, CS91b, VL92, VWL94, And94] pour le fragment correspondant du μ-calcul standard. Elle procède par étapes successives, illustrées sur la figure 4.3.

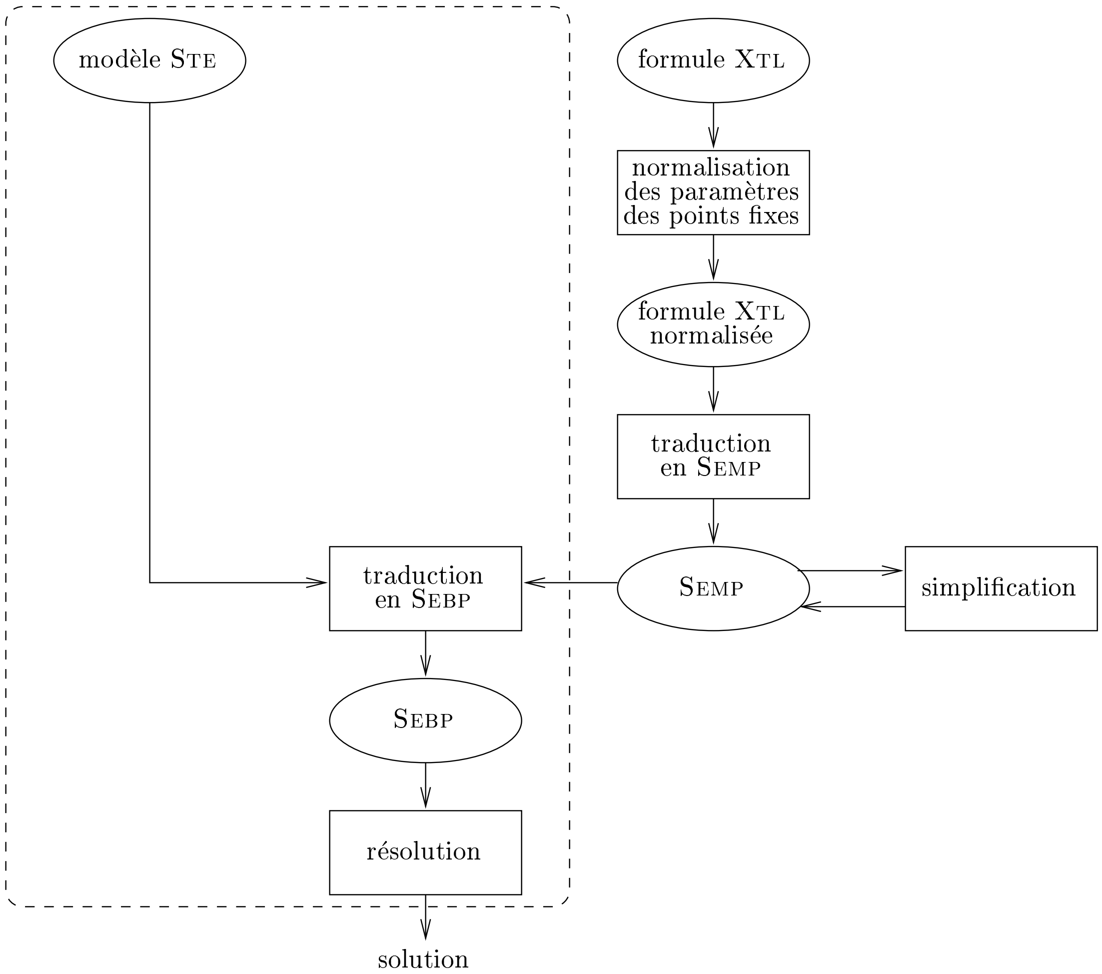

Figure 4.3: Evaluation des formules XTL d'alternance 1

Les formules XTL subissent d'abord une transformation préliminaire, appelée *normalisation des paramètres des opérateurs de point fixe* (voir la section 4.2.1), destinée à faciliter les phases ultérieures de traduction et d'évaluation.

Les formules obtenues, appelées *formules normalisées*, sont ensuite traduites vers des *systèmes d'équations modales paramétrées* par des variables typées (SEMPs), c'est-à-dire des systèmes dont les parties gauches des équations sont des variables propositionnelles paramétrées et les parties droites sont des formules sur états ne contenant pas d'opérateurs de point fixe. Cette phase de traduction (voir la section 4.2.2) est une généralisation naturelle de la traduction vers des systèmes d'équations modales utilisée dans les algorithmes pour le μ-calcul standard d'alternance 1 [AC88, CS91b, VL94, And94]. Afin de permettre leur résolution efficace, les SEMPs ainsi obtenus sont *simplifiés*, par introduction de nouvelles variables, de façon à ce que chaque formule en partie droite d'une équation ne contienne qu'au plus un seul opérateur.

Le problème de la résolution des SEMPS simplifiés est ramené à la résolution de *systèmes d'équations booléennes paramétrées* (SEBPS) construits à partir du modèle STE et des SEMPS (voir la section 4.2.3), c'est-à-dire des systèmes dont les parties gauches des équations sont des variables booléennes paramétrées et les parties droites sont des prédicats portant sur les paramètres. Les SEBPS obtenus sont résolus au moyen d'une méthode itérative semi-décidable (voir la section 4.2.4). L'algorithme de résolution proposé permet de traiter les deux approches (qui ont été mentionnées à la section 1.3) utilisées pour l'évaluation des formules :

Evaluation globale, consistant à calculer tous les états du STE qui satisfont une formule. Ceci nécessite la construction complète du STE et du SEBP avant de commencer la résolution ;

Evaluation locale, consistant à déterminer si l'état initial du STE satisfait une formule. Ceci peut être effectué à la volée, en générant le STE et le SEBP au fur et à mesure de la résolution.

Sur la figure 4.3, les phases d'évaluation entourées de pointillés (génération du modèle STE, construction et résolution des SEBPS) peuvent être effectuées soit globalement, soit localement.

4.2.1 Normalisation des paramètres des opérateurs de point fixe

Afin de permettre une traduction des formules XTL d'alternance 1 vers des systèmes d'équations modales paramétrées bien définis (voir la section 4.2.2), il est nécessaire de transformer chaque formule de point fixe $\sigma Y\,(x_1{:}T_1{:=}E_1,\ldots,x_n{:}T_n{:=}E_n).\varphi$ de manière à ce que toutes les variables simples libres dans φ figurent dans la liste de paramètres $x_1,\ldots,x_n$:

$$fdv(\varphi) \subseteq \{x_1,\ldots,x_n\} \tag{4.1}$$

Ceci assure que dans chaque équation du SEMP obtenu après traduction, les variables libres dans la partie droite de l'équation figurent parmi les paramètres de la variable propositionnelle définie en partie gauche de l'équation. Une formule φ est dite *avec paramètres normalisés*[12], ou simplement *normalisée*, si toutes ses sous-formules de point fixe satisfont la propriété (4.1).

Exemple 4-3
Considérons la formule suivante, exprimant qu'après l'émission d'un message (action `SEND`) il est toujours possible d'atteindre, avant d'effectuer une nouvelle émission, la réception du même message (action `RECV`) :

$$\begin{array}{l}
\textbf{nu } Y_1 \textbf{ . (}\\
\quad [\texttt{SEND} \textbf{ ? } m_1 \textbf{ : } \texttt{Msg}]\ \textbf{mu } Y_2 \textbf{ . (}\langle \texttt{RECV} \textbf{ ! } m_1 \rangle \textbf{ true or } \langle \textbf{not } (\texttt{SEND} \textbf{ any}) \rangle\ Y_2\textbf{) and}\\
\quad [\textbf{true}]\ Y_1\\
\textbf{)}
\end{array}$$

La sous-formule "**mu** Y_2" n'est pas normalisée, car la variable m_1, qui est libre dans cette sous-formule, n'est pas un paramètre de Y_2. ∎

Toute formule φ d'alternance 1 peut être transformée, au moyen de substitutions syntaxiques appropriées, sous forme normalisée, sans que la sémantique de φ soit modifiée. La normalisation des paramètres est effectuée au moyen de la fonction syntaxique suivante :

$$\textrm{TRN} : \mathit{SForm} \to \mathit{SForm}$$

Pour chaque formule $\sigma Y\,(x_1{:}T_1{:=}E_1,\ldots,x_n{:}T_n{:=}E_n).\varphi$, notons $x'_1,\ldots,x'_n$ les variables libres de φ qui ne figurent pas parmi les paramètres formels de Y :

$$\{x'_1,\ldots,x'_p\} \stackrel{\mathrm{d}}{=} fdv(\varphi) \setminus \{x_1,\ldots,x_n\}.$$

[12] Dans [Dam94b], ces formules sont appelées *fully parameterized.*

La fonction TRN est définie inductivement comme suit :

$$\begin{array}{rcl}
\mathrm{T{\scriptstyle R}N}(\textbf{true}) & \stackrel{\mathrm{d}}{=} & \textbf{true} \\
\mathrm{T{\scriptstyle R}N}(\textbf{false}) & \stackrel{\mathrm{d}}{=} & \textbf{false} \\
\mathrm{T{\scriptstyle R}N}(\varphi_1 \textbf{ or } \varphi_2) & \stackrel{\mathrm{d}}{=} & \mathrm{T{\scriptstyle R}N}(\varphi_1) \textbf{ or } \mathrm{T{\scriptstyle R}N}(\varphi_2) \\
\mathrm{T{\scriptstyle R}N}(\varphi_1 \textbf{ and } \varphi_2) & \stackrel{\mathrm{d}}{=} & \mathrm{T{\scriptstyle R}N}(\varphi_1) \textbf{ and } \mathrm{T{\scriptstyle R}N}(\varphi_2) \\
\mathrm{T{\scriptstyle R}N}(Y(E_1,\ldots,E_n)) & \stackrel{\mathrm{d}}{=} & Y(E_1,\ldots,E_n) \\
\mathrm{T{\scriptstyle R}N}(\langle\alpha\rangle\ \varphi) & \stackrel{\mathrm{d}}{=} & \langle\alpha\rangle\ \mathrm{T{\scriptstyle R}N}(\varphi) \\
\mathrm{T{\scriptstyle R}N}([\alpha]\ \varphi) & \stackrel{\mathrm{d}}{=} & [\alpha]\ \mathrm{T{\scriptstyle R}N}(\varphi) \\
\mathrm{T{\scriptstyle R}N}\left(\begin{array}{l}\textbf{exists } x_0\textbf{:}T_0 \textbf{ [among } E_0\textbf{]} \\ \quad ,\ldots, \\ \quad x_n\textbf{:}T_n \textbf{ [among } E_n\textbf{]} \\ \textbf{in } \varphi\end{array}\right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l}\textbf{exists } x_0\textbf{:}T_0 \textbf{ [among } E_0\textbf{]} \\ \quad ,\ldots, \\ \quad x_n\textbf{:}T_n \textbf{ [among } E_n\textbf{]} \\ \textbf{in } \mathrm{T{\scriptstyle R}N}(\varphi)\end{array} \\
\mathrm{T{\scriptstyle R}N}\left(\begin{array}{l}\textbf{forall } x_0\textbf{:}T_0 \textbf{ [among } E_0\textbf{]} \\ \quad ,\ldots, \\ \quad x_n\textbf{:}T_n \textbf{ [among } E_n\textbf{]} \\ \textbf{in } \varphi\end{array}\right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l}\textbf{forall } x_0\textbf{:}T_0 \textbf{ [among } E_0\textbf{]} \\ \quad ,\ldots, \\ \quad x_n\textbf{:}T_n \textbf{ [among } E_n\textbf{]} \\ \textbf{in } \mathrm{T{\scriptstyle R}N}(\varphi)\end{array} \\
\mathrm{T{\scriptstyle R}N}\left(\begin{array}{l}\textbf{case } E_0 \textbf{ in} \\ \quad P_1^0 \mid \ldots \mid P_1^{n_1} \\ \qquad \textbf{[where } E_1\textbf{]} \ \texttt{->}\ \varphi_1 \\ \quad \ldots \\ \mid P_m^0 \mid \ldots \mid P_m^{n_m} \\ \qquad \textbf{[where } E_m\textbf{]} \ \texttt{->}\ \varphi_m \\ \textbf{[} \mid \textbf{otherwise } \texttt{->}\ \varphi_{m+1}\textbf{]} \\ \textbf{endcase}\end{array}\right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l}\textbf{case } E_0 \textbf{ in} \\ \quad P_1^0 \mid \ldots \mid P_1^{n_1} \\ \qquad \textbf{[where } E_1\textbf{]} \ \texttt{->}\ \mathrm{T{\scriptstyle R}N}(\varphi_1) \\ \quad \ldots \\ \mid P_m^0 \mid \ldots \mid P_m^{n_m} \\ \qquad \textbf{[where } E_m\textbf{]} \ \texttt{->}\ \mathrm{T{\scriptstyle R}N}(\varphi_m) \\ \textbf{[} \mid \textbf{otherwise } \texttt{->}\ \mathrm{T{\scriptstyle R}N}(\varphi_{m+1})\textbf{]} \\ \textbf{endcase}\end{array} \\
\mathrm{T{\scriptstyle R}N}\left(\begin{array}{l}\textbf{case action } E_0 \textbf{ in} \\ \quad \alpha_1\ \textbf{[where } E_1\textbf{]} \ \texttt{->}\ \varphi_1 \\ \quad \ldots \\ \mid \alpha_m \textbf{ [where } E_m\textbf{]} \ \texttt{->}\ \varphi_m \\ \textbf{[} \mid \textbf{otherwise } \texttt{->}\ \varphi_{m+1}\textbf{]} \\ \textbf{endcase}\end{array}\right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l}\textbf{case action } E_0 \textbf{ in} \\ \quad \alpha_1\ \textbf{[where } E_1\textbf{]} \ \texttt{->}\ \mathrm{T{\scriptstyle R}N}(\varphi_1) \\ \quad \ldots \\ \mid \alpha_m \textbf{ [where } E_m\textbf{]} \ \texttt{->}\ \mathrm{T{\scriptstyle R}N}(\varphi_m) \\ \textbf{[} \mid \textbf{otherwise } \texttt{->}\ \mathrm{T{\scriptstyle R}N}(\varphi_{m+1})\textbf{]} \\ \textbf{endcase}\end{array} \\
\mathrm{T{\scriptstyle R}N}\left(\begin{array}{l}\sigma Y(x_1\textbf{:}T_1\textbf{:=}E_1,\ldots, \\ \qquad x_n\textbf{:}T_n\textbf{:=}E_n)\textbf{.}\varphi\end{array}\right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l}\sigma Y(x_1^Y\textbf{:}T_1'\textbf{:=}x_1',\ldots,x_p^Y\textbf{:}T_p'\textbf{:=}x_p', \\ \qquad x_1\textbf{:}T_1\textbf{:=}E_1,\ldots,x_n\textbf{:}T_n\textbf{:=}E_n)\textbf{.(} \\ \quad \big((\mathrm{T{\scriptstyle R}N}(\varphi))\,[x_1^Y/x_1',\ldots,x_p^Y/x_p']\big) \\ \qquad [Y(x_1^Y,\ldots,x_p^Y,E_1',\ldots,E_n')/Y(E_1',\ldots,E_n')] \\ )\end{array}
\end{array}$$

où pour chaque formule $\sigma Y(x_1\textbf{:}T_1\textbf{:=}E_1,\ldots,x_n\textbf{:}T_n\textbf{:=}E_n)\textbf{.}\varphi$, les variables $x_1^Y,\ldots,x_p^Y$ sont de nouvelles variables, différentes de toutes les autres variables de φ. Intuitivement, $x_1^Y,\ldots,x_p^Y$ remplacent les variables $x_1',\ldots,x_p'$, qui sont libres dans φ mais ne figurent pas parmi $x_1,\ldots,x_n$. La transformation procède de la manière suivante : après la normalisation de φ, toutes les occurrences de $x_1',\ldots,x_p'$ dans $\mathrm{T{\scriptstyle R}N}(\varphi)$ sont remplacées respectivement par $x_1^Y,\ldots,x_p^Y$; ensuite tous les appels $Y(E_1',\ldots,E_n')$ dans la formule obtenue sont remplacés par $Y(x_1^Y,\ldots,x_p^Y,E_1',\ldots,E_n')$, de façon à propager les valeurs des paramètres formels $x_1^Y,\ldots,x_p^Y$, qui sont initialisés dans la formule de point fixe avec les valeurs des variables $x_1',\ldots,x_p'$.

Exemple 4-4
Appliquée sur la formule de l'exemple 4-3, la fonction TrN produit la formule normalisée suivante :

$$\begin{array}{l}\textbf{nu}\ Y_1\ \textbf{.}\ ([\texttt{SEND ?}\ m_1\ \texttt{: Msg}] \\ \quad \textbf{mu}\ Y_2\ (m_1^{Y_2}\ \texttt{: Msg :=}\ m_1)\ \textbf{.}\ (\langle \texttt{RECV !}\ m_1^{Y_2} \rangle\ \textbf{true or}\ \langle \textbf{not}\ (\texttt{SEND}\ \textbf{any}) \rangle\ Y_2(m_1^{Y_2}))\ \textbf{and} \\ \quad [\,\textbf{true}\,]\ Y_1 \\ )\end{array}$$

■

Le reste de cette section (qui peut être évité en première lecture) contient la preuve que la transformation TrN(φ) préserve la sémantique des formules φ.

Plusieurs résultats préliminaires sont nécessaires. Soit une formule $\varphi \in \mathit{SForm}$, un environnement simple $\varepsilon \in \mathbf{DEnv}$ tel que $\mathit{fdv}(\varphi) \subseteq \mathit{supp}(\varepsilon)$, un environnement propositionnel $\rho \in \mathbf{PEnv}$ tel que $\mathit{fpv}(\varphi) \subseteq \mathit{supp}(\rho)$ et une variable $Y(x_1{:}T_1, \ldots, x_n{:}T_n) \in \mathit{PVar}$. La fonctionnelle $\Phi_{\rho\varepsilon Y}$: $(T_1 \times \cdots \times T_n \to 2^S) \to (T_1 \times \cdots \times T_n \to 2^S)$ associée à φ, ρ, ε et Y est définie comme suit :

$$\Phi_{\rho\varepsilon Y}(F) \stackrel{\mathrm{d}}{=} \lambda v_1{:}T_1, \ldots, v_n{:}T_n.\, [\![\varphi]\!]\, (\rho \oslash [F/Y])(\varepsilon \oslash [v_1/x_1, \ldots, v_n/x_n])$$

pour tout $F : T_1 \times \cdots \times T_n \to 2^S$. Le lemme suivant exprime le fait que les fonctionnelles $\Phi_{\rho\varepsilon Y}$ associées aux formules φ d'alternance 1 sont $\sqcup$- et $\sqcap$-continues.

Lemme 4-1 (Continuité des formules avec alternance 1)
Soient $\varphi \in \mathit{SForm}$ une formule d'alternance 1, $\varepsilon \in \mathbf{DEnv}$ tel que $\mathit{fdv}(\varphi) \subseteq \mathit{supp}(\varepsilon)$, $\rho \in \mathbf{PEnv}$ tel que $\mathit{fpv}(\varphi) \subseteq \mathit{supp}(\rho)$ et $Y(x_1{:}T_1, \ldots, x_n{:}T_n) \in \mathit{PVar}$ telle que $Y \notin \mathit{bpv}(\varphi)$. Soit $F_i : T_1 \times \cdots \times T_n \to 2^S$ une suite croissante, $G_i : T_1 \times \cdots \times T_n \to 2^S$ une suite décroissante et $\Phi_{\rho\varepsilon Y} : (T_1 \times \cdots \times T_n \to 2^S) \to (T_1 \times \cdots \times T_n \to 2^S)$ la fonctionnelle associée à φ, ρ, ε et Y. Alors :

$$\begin{aligned}\Phi_{\rho\varepsilon Y}\Big(\bigsqcup_{i\geq 0} F_i\Big) &= \bigsqcup_{i\geq 0} \Phi_{\rho\varepsilon Y}(F_i) \\ \Phi_{\rho\varepsilon Y}\Big(\bigsqcap_{i\geq 0} G_i\Big) &= \bigsqcap_{i\geq 0} \Phi_{\rho\varepsilon Y}(G_i)\end{aligned}$$

signifiant que $\Phi_{\rho\varepsilon Y}$ est $\sqcup$- et $\sqcap$-continue. ■

Preuve Par induction structurelle sur φ. Les arguments essentiels sont (1) le fait que le degré de branchement du Ste est fini, ce qui assure la continuité des modalités "$\langle\ \rangle$" et "$[\ ]$", (2) le fait que les domaines des variables quantifiées sont finis (voir la section 2.6.5), ce qui assure la continuité des quantificateurs "**exists**" et "**forall**" et (3) le fait que les formules φ sont d'alternance 1, ce qui assure la continuité des opérateurs de point fixe "**mu**" et "**nu**". □

Le lemme suivant exprime une propriété des fonctionnelles continues définies sur des treillis complets.

Lemme 4-2
Soit $\langle D, \sqcup, \sqcap, \sqsubseteq \rangle$ un treillis complet, D_1 et D_2 des ensembles non vides, $F_1 : (D_1 \to D) \to (D_1 \to D)$, $F_2 : (D_2 \times D_1 \to D) \to (D_2 \times D_1 \to D)$ deux fonctionnelles $\sqcup$- et $\sqcap$-continues et $v_2 \in D_2$ tels que :

$$\begin{aligned}&\forall v_1 \in D_1. \forall f_1 : D_1 \to D. \forall f_2 : D_2 \times D_1 \to D. \\ &\quad f_1(v_1) = f_2(v_2, v_1) \Rightarrow (F_1(f_1))(v_1) = (F_2(f_2))(v_2, v_1)\end{aligned} \tag{4.2}$$

Alors, pour tout $v_1 \in D_1$ et pour tout $\sigma \in \{\mu, \nu\}$:

$$(\sigma F_1)(v_1) = (\sigma F_2)(v_2, v_1).$$

■

Preuve Nous démontrons le lemme pour $\sigma = \mu$, le cas $\sigma = \nu$ étant dual. Les fonctionnelles F_1 et F_2 étant continues et $\langle D, \sqcup, \sqcap, \sqsubseteq \rangle$ étant un treillis complet, les plus petits points fixes de F_1 et F_2 ont la caractérisation itérative suivante [Kle52] :

$$\begin{array}{lcl} \mu F_1 & = & \bigsqcup_{k \geq 0} F_1^k(\lambda x_1{:}D_1.\bot) \\ \mu F_2 & = & \bigsqcup_{k \geq 0} F_2^k(\lambda x_1{:}D_1, x_2{:}D_2.\bot) \end{array} \tag{4.3}$$

Nous montrons par induction sur k que, pour tout $v_1 \in D_1$:

$$(F_1^k(\lambda x_1{:}D_1.\bot))(v_1) = (F_2^k(\lambda x_1{:}D_1, x_2{:}D_2.\bot))(v_2, v_1) \tag{4.4}$$

- $k = 0$. Pour tout $v_1 \in D_1$, $(F_1^0(\lambda x_1{:}D_1.\bot))(v_1) \stackrel{\text{d}}{=} \bot \stackrel{\text{d}}{=} (F_2^0(\lambda x_1{:}D_1, x_2{:}D_2.\bot))(v_2, v_1)$.
- $k = n + 1$. Par hypothèse d'induction pour $k = n$, nous avons, pour tout $v_1 \in D_1$:

  $$(F_1^n(\lambda x_1{:}D_1.\bot))(v_1) = (F_2^n(\lambda x_1{:}D_1, x_2{:}D_2.\bot))(v_2, v_1)$$

  En prenant $f_1 \stackrel{\text{d}}{=} F_1^n(\lambda x_1{:}D_1.\bot)$ et $f_2 \stackrel{\text{d}}{=} F_2^n(\lambda x_1{:}D_1, x_2{:}D_2.\bot)$ et en appliquant l'hypothèse (4.2) du lemme, nous avons que, pour tout $v_1 \in D_1$:

  $$\begin{array}{l} (F_1(F_1^n(\lambda x_1{:}D_1.\bot)))(v_1) = (F_2(F_2^n(\lambda x_1{:}D_1, x_2{:}D_2.\bot)))(v_2, v_1) \Leftrightarrow \\ (F_1^{n+1}(\lambda x_1{:}D_1.\bot))(v_1) = (F_2^{n+1}(\lambda x_1{:}D_1, x_2{:}D_2.\bot))(v_2, v_1). \end{array}$$

La valeur $(\mu F_1)(v_1)$ peut être maintenant calculée comme suit :

$$\begin{array}{ll} (\mu F_1)(v_1) = & \text{par } (4.3) \\ \big(\bigsqcup_{k \geq 0} F_1^k(\lambda x_1{:}D_1.\bot)\big)(v_1) = & \text{par définition de } \sqcup \\ \bigsqcup_{k \geq 0} \big(F_1^k(\lambda x_1{:}D_1.\bot)\big)(v_1) = & \text{par } (4.4) \\ \bigsqcup_{k \geq 0} \big(F_2^k(\lambda x_2{:}D_2, x_1{:}D_1.\bot)\big)(v_2, v_1) = & \text{par définition de } \sqcup \\ \big(\bigsqcup_{k \geq 0} F_2^k(\lambda x_1{:}D_1.\bot)\big)(v_2, v_1) = & \text{par } (4.3) \\ (\mu F_2)(v_2, v_1) & \end{array}$$

□

Le lemme suivant exprime la préservation de la sémantique des formules φ après certaines substitutions utilisées dans la définition de $\textsc{TrN}(\varphi)$.

Lemme 4-3
Soit une formule $\varphi \in \mathit{SForm}$, un environnement propositionnel $\rho \in \mathbf{PEnv}$, un environnement simple $\varepsilon \in \mathbf{DEnv}$, une variable propositionnelle $Y(x_1{:}T_1, \ldots, x_n{:}T_n) \in \mathit{supp}(\rho) \cap \mathit{fpv}(\varphi)$ et des variables simples $x_1'{:}T_1', \ldots, x_p'{:}T_p' \in \mathit{supp}(\varepsilon)$ telles que $\{x_1', \ldots, x_p'\} \cap \mathit{bdv}(\varphi) = \emptyset$. Soit une fonction G : $T_1' \times \cdots \times T_p' \times T_1 \times \cdots \times T_n \to 2^S$ telle que :

$$\forall v_1 \in T_1 \ldots \forall v_n \in T_n . G(\varepsilon(x_1'), \ldots, \varepsilon(x_p'), v_1, \ldots, v_n) = (\rho(Y))(v_1, \ldots, v_n).$$

Alors :

$$\left[\!\!\left[\begin{array}{l} (\varphi\, [x_1^Y/x_1', \ldots, x_p^Y/x_p'] \,) \\ \quad [Y(x_1^Y, \ldots, x_p^Y, E_1, \ldots, E_n)/Y(E_1, \ldots, E_n)] \end{array} \right]\!\!\right] \left(\begin{array}{l} (\rho \oslash [G/Y]) \\ \quad (\varepsilon \oslash [\varepsilon(x_1')/x_1^Y, \ldots, \varepsilon(x_p')/x_p^Y]) \end{array} \right) = [\![\varphi]\!] \rho\varepsilon$$

où $x_1^Y, \ldots, x_p^Y$ sont de nouvelles variables simples, différentes des autres variables de φ. ■

Preuve Elémentaire (mais fastidieuse), par induction structurelle sur φ. □

La préservation de la sémantique des formules φ suite à la normalisation des paramètres des opérateurs de point fixe est exprimée formellement par la proposition suivante.

Proposition 4-1 (Normalisation des paramètres des opérateurs de point fixe)
Soit une formule $\varphi \in \mathit{SForm}$, un environnement propositionnel $\rho \in \mathbf{PEnv}$ tel que $\mathit{fpv}(\varphi) \subseteq \mathit{supp}(\rho)$ et un environnement simple $\varepsilon \in \mathbf{DEnv}$ tel que $\mathit{fdv}(\varphi) \subseteq \mathit{supp}(\varepsilon)$. Alors :

$$[\![\mathrm{TRN}(\varphi)]\!]\,\rho\varepsilon = [\![\varphi]\!]\,\rho\varepsilon.$$

■

Preuve Soient $\varphi \in \mathit{SForm}$, $\rho \in \mathbf{PEnv}$ tel que $\mathit{fpv}(\varphi) \subseteq \mathit{supp}(\rho)$ et $\varepsilon \in \mathbf{DEnv}$ tel que $\mathit{fdv}(\varphi) \subseteq \mathit{supp}(\varepsilon)$. Nous procédons par induction structurelle sur φ. Le seul cas intéressant est $\varphi ::= \sigma Y(x_1{:}T_1{:=}E_1,\ldots,x_n{:}T_n{:=}E_n).\varphi_1$, tous les autres cas étant des conséquences immédiates de la définition de $\mathrm{TRN}(\varphi)$ et/ou de l'hypothèse d'induction.

La sémantique de φ est calculée de la manière suivante :

$$\begin{array}{ll} [\![\varphi]\!]\,\rho\varepsilon = [\![\sigma Y(x_1{:}T_1{:=}E_1,\ldots,x_n{:}T_n{:=}E_n).\varphi_1]\!]\,\rho\varepsilon = & \text{par définition de } [\![.]\!] \\ (\sigma\Phi_{\rho\varepsilon})([\![E_1]\!]\,\varepsilon,\ldots,[\![E_n]\!]\,\varepsilon) & \end{array}$$

où la fonctionnelle $\Phi_{\rho\varepsilon} : (T_1\times\cdots\times T_n \to 2^S) \to (T_1\times\cdots\times T_n \to 2^S)$ est définie ci-dessous :

$$\Phi_{\rho\varepsilon}(F) \stackrel{\mathrm{d}}{=} \lambda v_1{:}T_1,\ldots,v_n{:}T_n.\,[\![\varphi_1]\!]\,(\rho\oslash[F/Y])(\varepsilon\oslash[v_1/x_1,\ldots,v_n/x_n])$$

pour chaque $F : T_1\times\cdots\times T_n \to 2^S$. Supposant que $\{x'_1,\ldots,x'_p\} = \mathit{fdv}(\varphi)\setminus\{x_1,\ldots,x_n\}$ et que les types des variables $x'_1,\ldots,x'_p$ sont respectivement $T'_1,\ldots,T'_p$, la sémantique de $\mathrm{TRN}(\varphi)$ est calculée de la façon suivante :

$$\begin{array}{ll} [\![\mathrm{TRN}(\varphi)]\!]\,\rho\varepsilon = & \text{par définition de TRN} \\ \left[\!\!\left[\begin{array}{l} \sigma Y(x_1^Y{:}T'_1{:=}x'_1,\ldots,x_p^Y{:}T'_p{:=}x'_p,x_1{:}T_1{:=}E_1,\ldots,x_n{:}T_n{:=}E_n).(\\ \quad((\mathrm{TRN}(\varphi_1))\,[x_1^Y/x'_1,\ldots,x_p^Y/x'_p]) \\ \quad\;[Y(x_1^Y,\ldots,x_p^Y,E'_1,\ldots,E'_n)/Y(E'_1,\ldots,E'_n)] \\ ) \end{array}\right]\!\!\right]\rho\varepsilon = & \text{par définition de } [\![.]\!] \\ (\sigma\Phi_{1\rho\varepsilon})([\![x'_1]\!]\,\varepsilon,\ldots,[\![x'_n]\!]\,\varepsilon,[\![E_1]\!]\,\varepsilon,\ldots,[\![E_n]\!]\,\varepsilon) = & \text{par définition de } [\![.]\!] \\ (\sigma\Phi_{1\rho\varepsilon})(\varepsilon(x'_1),\ldots,\varepsilon(x'_n),[\![E_1]\!]\,\varepsilon,\ldots,[\![E_n]\!]\,\varepsilon) & \end{array}$$

où la fonctionnelle $\Phi_{1\rho\varepsilon} : (T'_1\times\cdots\times T'_p\times T_1\times\cdots\times T_n \to 2^S) \to (T'_1\times\cdots\times T'_p\times T_1\times\cdots\times T_n \to 2^S)$ est définie comme suit :

$$\begin{array}{rcl} \Phi_{1\rho\varepsilon}(G) & \stackrel{\mathrm{d}}{=} & \lambda v_Y^1{:}T'_1,\ldots,v_Y^p{:}T'_p,v_1{:}T_1,\ldots,v_n{:}T_n. \\ & & [\![((\mathrm{TRN}(\varphi_1))\,[x_1^Y/x'_1,\ldots,x_p^Y/x'_p]\,)\,[Y(x_1^Y,\ldots,x_p^Y,E'_1,\ldots,E'_n)/Y(E'_1,\ldots,E'_n)]]\!] \\ & & \quad(\rho\oslash[G/Y])(\varepsilon\oslash[v_Y^1/x_1^Y,\ldots,v_Y^p/x_p^Y,v_1/x_1,\ldots,v_n/x_n]) \end{array}$$

pour chaque $G : T'_1\times\cdots\times T'_p\times T_1\times\cdots\times T_n \to 2^S$. La formule φ étant d'alternance 1, le lemme 4-1 assure que les deux fonctionnelles $\Phi_{\rho\varepsilon}$ et $\Phi_{1\rho\varepsilon}$ sont $\sqcup$- et $\sqcap$-continues. Supposons que pour tout $v_1 \in T_1, \ldots, v_n \in T_n$, $F(v_1,\ldots,v_n) = G(\varepsilon(x'_1),\ldots,\varepsilon(x'_p),v_1,\ldots,v_n)$. Alors :

$$\begin{array}{ll} (\Phi_{1\rho\varepsilon}(G))(\varepsilon(x'_1),\ldots,\varepsilon(x'_p),v_1,\ldots,v_n) = & \text{par définition de } \Phi_{1\rho\varepsilon} \\ [\![((\mathrm{TRN}(\varphi_1))\,[x_1^Y/x'_1,\ldots,x_p^Y/x'_p]\,)\,[Y(x_1^Y,\ldots,x_p^Y,E'_1,\ldots,E'_n)/Y(E'_1,\ldots,E'_n)]]\!] & \\ \quad(\rho\oslash[G/Y])(\varepsilon\oslash[v_Y^1/x_1^Y,\ldots,v_Y^p/x_p^Y,v_1/x_1,\ldots,v_n/x_n]) = & \text{par propriétés de } \oslash \\ [\![((\mathrm{TRN}(\varphi_1))\,[x_1^Y/x'_1,\ldots,x_p^Y/x'_p]\,)\,[Y(x_1^Y,\ldots,x_p^Y,E'_1,\ldots,E'_n)/Y(E'_1,\ldots,E'_n)]]\!] & \\ \quad((\rho\oslash[F/Y])\oslash[G/Y])((\varepsilon\oslash[v_1/x_1,\ldots,v_n/x_n])\oslash[v_Y^1/x_1^Y,\ldots,v_Y^p/x_p^Y]) = & \text{par le lemme (4-3)} \\ [\![\mathrm{TRN}(\varphi_1)]\!]\,(\rho\oslash[F/Y])(\varepsilon\oslash[v_1/x_1,\ldots,v_n/x_n]) = & \text{par induction} \\ [\![\varphi_1]\!]\,(\rho\oslash[F/Y])(\varepsilon\oslash[v_1/x_1,\ldots,v_n/x_n]) = & \text{par définition de } \Phi_{\rho\varepsilon} \\ (\Phi_{\rho\varepsilon}(F))(v_1,\ldots,v_n). & \end{array}$$

Les fonctionnelles $\Phi_{\rho\varepsilon}$ et $\Phi_{1\rho\varepsilon}$ satisfont donc les conditions requises dans l'hypothèse du lemme 4-2 et, par conséquent :

$$\forall v_1 \in T_1 \ldots \forall v_n \in T_n.(\sigma\Phi_{\rho\varepsilon})(v_1, \ldots, v_n) = (\sigma\Phi_{1\rho\varepsilon})(\varepsilon(x'_1), \ldots, \varepsilon(x'_p), v_1, \ldots, v_n)$$

ce qui implique $[\![\textsc{TrN}(\varphi)]\!]\,\rho\varepsilon = [\![\varphi]\!]\,\rho\varepsilon$. □

Dans les sections suivantes, nous ne considérons que des formules XTL normalisées d'alternance 1.

4.2.2 Transformation en systèmes d'équations modales paramétrées

La traduction des formules de point fixe vers des systèmes d'équations modales est utilisée dans la plupart des algorithmes d'évaluation dédiés au μ-calcul standard d'alternance 1 [AC88, CS91a, CS91b, VL92, And94]. L'approche que nous suivons ici est une généralisation naturelle de cette traduction : puisque les formules XTL de point fixe sont paramétrées par des variables typées, il est naturel de les traduire vers des systèmes d'équations modales paramétrées (SEMPs). Nous commençons par définir la syntaxe et la sémantique des SEMPs et ensuite nous présentons la traduction des formules XTL normalisées d'alternance 1 vers ces systèmes.

Systèmes d'équations modales paramétrées Un système d'équations modales paramétrées (SEMP) SM est un ensemble d'équations de point fixe ayant la syntaxe suivante :

$$\left\{Y_i(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma_i}{=} \varphi_i\right\}_{1 \le i \le n}$$

où, pour chaque $1 \le i \le n$, $Y_i(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \in PVar$ est une variable propositionnelle, $\sigma_i \in \{\mu, \nu\}$ est le *signe* de l'équation i et $\varphi_i \in SForm$ dénote une formule *modale*, c'est-à-dire ne contenant pas d'opérateurs "**mu**" et "**nu**". Chaque formule φ_i $(1 \le i \le n)$ doit être *positive*, c'est-à-dire qu'elle ne doit pas contenir de négations. En plus, pour chaque équation $1 \le i \le n$, $fdv(\varphi_i) \subseteq \{x_i^1, \ldots, x_i^{n_i}\}$. Une variable Y_i est appelée μ-variable (*resp.* ν-variable) si $\sigma_i = \mu$ (*resp.* $\sigma_i = \nu$).

Le domaine syntaxique associé aux SEMPs est noté *MSys*. Les variables propositionnelles libres et liées dans un SEMP $SM \in MSys$ sont définies respectivement par les fonctions syntaxiques suivantes :

$$fpv, bpv : MSys \to PVar$$

Ces fonctions sont données dans la table 4.1. Un SEMP SM ne contient pas de variables simples libres. Un SEMP SM est dit *fermé* ssi $fpv(SM) = \emptyset$.

SM	$fpv(SM)$	$bpv(SM)$
$\left\{Y_i(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma_i}{=} \varphi_i\right\}_{1<i<n}$	$\bigcup_{i=1}^{n} (fpv(\varphi_i) \setminus \{Y_i\})$	$\{Y_1, \ldots, Y_n\}$

Table 4.1: Variables propositionnelles libres et liées dans les SEMPs

A chaque SEMP SM est associé un graphe de dépendance G_{SM} entre les variables propositionnelles contenues dans SM, ayant un sommet pour chaque variable Y_i et un arc de Y_i à Y_j ssi Y_i est définie en fonction de Y_j, c'est-à-dire $Y_j \in fpv(\varphi_i)$. Nous ne considérons ici que des SEMPs d'alternance 1, c'est-à-dire ne contenant pas de μ-variables et de ν-variables mutuellement récursives (définies une en fonction de l'autre). Les graphes de dépendance des SEMPs d'alternance 1 ne contiennent pas de circuits entre les μ- et les ν-variables.

Un SEMP dont toutes les variables ont le même signe $\sigma \in \{\mu, \nu\}$ est appelé σ-bloc. Un sous-système d'un SEMP SM est un SEMP contenant un sous-ensemble des équations de SM.

Une *partition en σ-blocs* d'un SEMP SM est un ensemble de σ-blocs $\{B_1, \dots, B_m\}$ tel que $SM = \bigcup_{j=1}^{m} B_j$ et pour tous $1 \leq j, k \leq m$, $B_j \cap B_k = \emptyset$. A chaque partition $\{B_1, \dots, B_m\}$ d'un SEMP SM est associé un graphe de dépendance entre les σ-blocs de la partition, ayant un sommet pour chaque σ-bloc B_j et un arc de B_j à B_k ssi il existe une variable libre de B_j qui est définie par une équation de B_k, c'est-à-dire $\mathit{fpv}(B_j) \cap \mathit{bpv}(B_k) \neq \emptyset$.

Soit un SEMP SM d'alternance 1 et soit $\{C_1, \dots, C_p\}$ les composantes fortement connexes maximales de G_{SM}. Chaque composante C_i induit un sous-système B_i de SM, contenant toutes les équations de SM qui définissent des variables contenues dans C_i. Puisque SM est d'alternance 1 et $\{C_1, \dots, C_p\}$ est une partition de l'ensemble des sommets de G_{SM}, l'ensemble des sous-systèmes induits $\{B_1, \dots, B_p\}$ forme une partition du système SM, appelée *partition canonique* de SM. Le graphe de dépendance associé à la partition canonique d'un SEMP d'alternance 1 est acyclique.

La sémantique des SEMPs ayant $n \geq 1$ équations est définie par la fonction d'interprétation suivante :

$$[\![.]\!] : \mathit{MSys} \to \mathbf{PEnv} \to (\mathbf{Param} \to 2^S)^n$$

Etant donné un SEMP $SM = \{Y_i(x_i^1{:}T_i^1, \dots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma_i}{=} \varphi_i\}_{1 \leq i \leq n}$ et un environnement propositionnel ρ tel que $\mathit{fpv}(SM) \subseteq \mathit{supp}(\rho)$, la dénotation $[\![SM]\!]\,\rho$ renvoie un tuple de fonctions $(G_1, \dots, G_n)$, où $G_i : T_i^1 \times \cdots \times T_i^{n_i} \to 2^S$ pour $1 \leq i \leq n$, représentant la solution de SM dans le contexte de ρ. Nous définissons d'abord la sémantique des σ-blocs et ensuite la sémantique des SEMPs d'alternance 1.

La sémantique d'un σ-bloc dans le contexte d'un environnement ρ est définie comme suit :

$$\left[\!\!\left[\{Y_i(x_i^1{:}T_i^1, \dots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma}{=} \varphi_i\}_{1 \leq i \leq n}\right]\!\!\right] \rho \stackrel{\mathrm{d}}{=} \sigma \overline{\Phi}_\rho$$

où $\overline{\Phi}_\rho : (\mathbf{Param} \to 2^S)^n \to (\mathbf{Param} \to 2^S)^n$ est la fonctionnelle associée au σ-bloc :

$$\overline{\Phi}_\rho(F_1, \dots, F_n) \stackrel{\mathrm{d}}{=} (\lambda v_i^1{:}T_i^1, \dots, v_i^{n_i}{:}T_i^{n_i}.\,[\![\varphi_i]\!]\,(\rho \oslash [F_1/Y_1, \dots, F_n/Y_n])[v_i^1/x_i^1, \dots, v_i^{n_i}/x_i^{n_i}])_{1 \leq i \leq n}$$

pour $F_i : T_i^1 \times \cdots \times T_i^{n_i} \to 2^S$ $(1 \leq i \leq n)$. Puisque la fonctionnelle $\overline{\Phi}_\rho$ est monotone (toutes les formules φ_i étant monotones) et le treillis $\langle (\mathbf{Param} \to 2^S)^n, \sqcup, \sqcap, \sqsubseteq \rangle$ est complet (les opérations $\sqcup$ et $\sqcap$ et la relation d'ordre $\sqsubseteq$ étant définies par extension des opérations correspondantes sur $\mathbf{Param} \to 2^S$), le théorème de Tarski assure l'existence et l'unicité des points fixes $\mu\overline{\Phi}_\rho$ et $\nu\overline{\Phi}_\rho$.

La sémantique d'un SEMP SM d'alternance 1 est définie à partir de la partition canonique en σ-blocs $\{B_1, \dots, B_p\}$ de SM. Puisque le graphe de dépendance associé à cette partition est acyclique, les σ-blocs peuvent être triés topologiquement suivant la relation de dépendance et renumérotés de façon à ce que pour chaque $1 \leq j \leq p$, tous les arcs qui sortent de B_j mènent à des σ-blocs B_k avec $k < j$. Pour tout $1 \leq j \leq p$, la sémantique de B_j peut être calculée en fonction des sémantiques de $B_1, \dots, B_{j-1}$:

$$[\![B_j]\!]\,\rho_j = (F_j^1, \dots, F_j^{r_j}) \tag{4.5}$$

où $\rho_1 = \rho$ et $\rho_{j+1} = \rho_j \oslash [F_j^1/Y_j^1, \dots, F_j^{r_j}/Y_j^{r_j}]$ pour tout $1 \leq j < p$.

La sémantique de SM est obtenue en réunissant les sémantiques de ses σ-blocs $B_1, \dots, B_p$:

$$\left[\!\!\left[\{Y_i(x_i^1{:}T_i^1, \dots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma_i}{=} \varphi_i\}_{1 \leq i \leq n}\right]\!\!\right] \rho \stackrel{\mathrm{d}}{=} (F_1, \dots, F_n) \tag{4.6}$$

où $\{F_1, \dots, F_n\} = \bigcup_{j=1}^{m} \{F_j^1, \dots, F_j^{r_j}\}$.

Traduction des formules XTL d'alternance 1 vers des SEMPs Soit une formule $\varphi \in \mathit{SForm}$ normalisée d'alternance 1 et soient $\sigma Y_i(x_i^1{:}T_i^1{:=}E_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}{:=}E_i^{n_i}).\varphi_i$, pour $1 \leq i \leq m$, les sous-formules maximales de point fixe de φ (c'est-à-dire, qui ne sont pas contenues dans d'autres sous-formules de point fixe de φ). L'*aplatissement* de φ, noté $\mathit{flat}(\varphi)$, est une formule définie comme suit :

$$\mathit{flat}(\varphi) \stackrel{\mathrm{d}}{=} \varphi\left[Y_i(E_i^1, \ldots, E_i^{n_i}) / \sigma Y_i(x_i^1{:}T_i^1{:=}E_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}{:=}E_i^{n_i}).\varphi_i\right]_{1 \leq i \leq m}$$

où $\varphi\left[Y_i(E_i^1, \ldots, E_i^{n_i}) / \sigma Y_i(x_i^1{:}T_i^1{:=}E_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}{:=}E_i^{n_i}).\varphi_i\right]_{1 \leq i \leq m}$ dénote la substitution dans φ de toutes ses sous-formules maximales de point fixe par leurs appels correspondants. Pour toute formule φ normalisée d'alternance 1, l'aplatissement de φ satisfait la propriété suivante :

$$\mathit{bpv}(\mathit{flat}(\varphi)) = \emptyset \ \wedge\ \mathit{fdv}(\mathit{flat}(\varphi)) = \mathit{fdv}(\varphi) \tag{4.7}$$

La traduction des formules XTL normalisées d'alternance 1 vers des SEMPs est effectuée au moyen de la fonction syntaxique suivante :

$$\textsc{TrM} : \mathit{SForm} \rightarrow \mathit{MSys}$$

Etant donné une formule $\varphi \in \mathit{SForm}$ avec $\mathit{bpv}(\varphi) = \{Y_1, \ldots, Y_n\}$, $\textsc{TrM}(\varphi)$ renvoie un SEMP $SM = \left\{Y_i(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma_i}{=} \mathit{flat}(\varphi_i)\right\}_{1 \leq i \leq n}$ contenant une équation pour chaque sous-formule $\sigma Y_i(x_i^1{:}T_i^1{:=}E_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}{:=}E_i^{n_i}).\varphi_i$ de φ. La fonction TRM est définie inductivement ci-dessous :

$$\begin{array}{rcl}
\textsc{TrM}(\mathbf{true}) & \stackrel{\mathrm{d}}{=} & \{\} \\
\textsc{TrM}(\mathbf{false}) & \stackrel{\mathrm{d}}{=} & \{\} \\
\textsc{TrM}(Y\ (E_1, \ldots, E_n)) & \stackrel{\mathrm{d}}{=} & \{\} \\
\textsc{TrM}(\varphi_1 \ \mathbf{or}\ \varphi_2) & \stackrel{\mathrm{d}}{=} & \textsc{TrM}(\varphi_1) \cup \textsc{TrM}(\varphi_2) \\
\textsc{TrM}(\varphi_1 \ \mathbf{and}\ \varphi_2) & \stackrel{\mathrm{d}}{=} & \textsc{TrM}(\varphi_1) \cup \textsc{TrM}(\varphi_2) \\
\textsc{TrM}(\langle \alpha \rangle\ \varphi) & \stackrel{\mathrm{d}}{=} & \textsc{TrM}(\varphi) \\
\textsc{TrM}([\alpha]\ \varphi) & \stackrel{\mathrm{d}}{=} & \textsc{TrM}(\varphi) \\
\textsc{TrM}(\sigma Y(x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n).\varphi) & \stackrel{\mathrm{d}}{=} & \left\{Y(x_1{:}T_1, \ldots, x_n{:}T_n) \stackrel{\sigma}{=} \mathit{flat}(\varphi)\right\} \cup \textsc{TrM}(\varphi) \\
\textsc{TrM}\begin{pmatrix} \mathbf{exists}\ x_0{:}T_0\ [\mathbf{among}\ E_0], \ldots, \\ \qquad x_n{:}T_n\ [\mathbf{among}\ E_n] \\ \mathbf{in}\ \varphi \end{pmatrix} & \stackrel{\mathrm{d}}{=} & \textsc{TrM}(\varphi) \\
\textsc{TrM}\begin{pmatrix} \mathbf{forall}\ x_0{:}T_0\ [\mathbf{among}\ E_0], \ldots, \\ \qquad x_n{:}T_n\ [\mathbf{among}\ E_n] \\ \mathbf{in}\ \varphi \end{pmatrix} & \stackrel{\mathrm{d}}{=} & \textsc{TrM}(\varphi) \\
\textsc{TrM}\begin{pmatrix} \mathbf{case}\ E_0\ \mathbf{in} \\ \quad P_1^0 \mid \ldots \mid P_1^{n_1}\ [\mathbf{where}\ E_1]\ \text{->}\ \varphi_1 \\ \quad \ldots \\ \mid P_m^0 \mid \ldots \mid P_m^{n_m}\ [\mathbf{where}\ E_m]\ \text{->}\ \varphi_m \\ [\mid \mathbf{otherwise}\ \text{->}\ \varphi_{m+1}] \\ \mathbf{endcase} \end{pmatrix} & \stackrel{\mathrm{d}}{=} & \displaystyle\bigcup_{i=1}^{m} \textsc{TrM}(\varphi_i)\,[\cup \textsc{TrM}(\varphi_{m+1})] \\
\textsc{TrM}\begin{pmatrix} \mathbf{case\ action}\ E_0\ \mathbf{in} \\ \quad \alpha_1\ [\mathbf{where}\ E_1]\ \text{->}\ \varphi_1 \\ \quad \ldots \\ \mid \alpha_m\ [\mathbf{where}\ E_m]\ \text{->}\ \varphi_m \\ [\mid \mathbf{otherwise}\ \text{->}\ \varphi_{m+1}] \\ \mathbf{endcase} \end{pmatrix} & \stackrel{\mathrm{d}}{=} & \displaystyle\bigcup_{i=1}^{m} \textsc{TrM}(\varphi_i)\,[\cup \textsc{TrM}(\varphi_{m+1})].
\end{array}$$

Les SEMPs $SM = \mathrm{TRM}(\varphi)$ produits par traduction des formules φ normalisées d'alternance 1 sont aussi d'alternance 1 et satisfont la propriété suivante :

$$fpv(SM) = fpv(\varphi) \ \wedge \ bpv(SM) = bpv(\varphi) \tag{4.8}$$

En outre, puisque les formules φ considérées ici sont fermées d.p.d.v. des variables propositionnelles (voir la remarque 3-7), les SEMPs $SM = \mathrm{TRM}(\varphi)$ produits par traduction sont fermés.

La complexité des algorithmes de résolution des SEMPs (voir les sections 4.2.3 et 4.2.4) dépend directement de leur taille (nombre de variables et d'opérateurs contenus dans les parties droites des équations). Il est facile de montrer (par induction structurelle sur φ) que la taille du SEMP $SM = \mathrm{TRM}(\varphi)$ produit par traduction ne peut pas dépasser la taille (nombre d'opérateurs) de φ.

Exemple 4-5
Nous reprenons la formule normalisée φ de l'exemple 4-4 (en renommant la variable $m_1^{Y_2}$ en m_2) :

$$\begin{aligned}\varphi \ = \ & \textbf{nu}\ Y_1\ \textbf{.}\ ([\mathtt{SEND}\ \textbf{?}\ m_1\ \textbf{:}\ \mathtt{Msg}] \\ & \quad \textbf{mu}\ Y_2\ (m_2\ \textbf{:}\ \mathtt{Msg}\ \textbf{:=}\ m_1)\ \textbf{.}\ (\langle \mathtt{RECV}\ \textbf{!}\ m_2 \rangle\ \textbf{true or}\ \langle \textbf{not}\ (\mathtt{SEND}\ \textbf{any}) \rangle\ Y_2(m_2))\ \textbf{and} \\ & \quad [\textbf{true}]\ Y_1 \\ &)\end{aligned}$$

La traduction $\mathrm{TRM}(\varphi)$ produit le SEMP suivant :

$$SM \ = \ \left\{ \begin{array}{rcl} Y_1 & \stackrel{\nu}{=} & [\mathtt{SEND}\ \textbf{?}\ m_1\ \textbf{:}\ \mathtt{Msg}]\ Y_2(m_1)\ \textbf{and}\ [\textbf{true}]\ Y_1 \\ Y_2\ (m_2\ \textbf{:}\ \mathtt{Msg}) & \stackrel{\mu}{=} & \langle \mathtt{RECV}\ \textbf{!}\ m_2 \rangle\ \textbf{true or}\ \langle \textbf{not}\ (\mathtt{SEND}\ \textbf{any}) \rangle\ Y_2\ (m_2) \end{array} \right.$$

La partition canonique de SM contient un ν-bloc $B_1 = \{Y_1\}$ et un μ-bloc $B_2 = \{Y_2\}$. Le graphe de dépendance entre les variables de SM, ainsi que sa partition canonique, sont illustrés sur la figure 4.4.

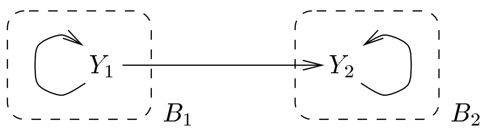

Figure 4.4: Graphe de dépendance avec partition canonique en σ-blocs

■

La relation entre une formule φ et le SEMP $\mathrm{TRM}(\varphi)$ obtenu par traduction est exprimée formellement pas la proposition suivante. Intuitivement, la sémantique d'une formule φ est égale à la sémantique de *flat*(φ), calculée dans le contexte de la solution du SEMP $\mathrm{TRM}(\varphi)$.

Proposition 4-2 (Traduction des formules d'alternance 1 vers des SEMPs)
Soit $\varphi \in SForm$ une formule normalisée d'alternance 1 telle que $bpv(\varphi) = \{Y_1, \ldots, Y_p\}$. Alors, pour tous les environnements $\rho \in \mathbf{PEnv}$ et $\varepsilon \in \mathbf{DEnv}$ tels que $fpv(\varphi) \subseteq supp(\rho)$ et $fdv(\varphi) \subseteq supp(\varepsilon)$:

$$[\![\varphi]\!]\,\rho\varepsilon = [\![flat(\varphi)]\!]\,(\rho \oslash [F_1/Y_1, \ldots, F_p/Y_p])\varepsilon$$

où $(F_1, \ldots, F_p) \stackrel{\mathrm{d}}{=} [\![\mathrm{TRM}(\varphi)]\!]\,\rho$. ■

Preuve Par induction sur la hauteur des formules φ (c'est-à-dire, la longueur maximale des chemins syntaxiques menant de la racine de φ à ses sous-formules atomiques), en utilisant le théorème de Bekić [Bek84] qui permet de transformer les points fixes imbriqués en systèmes d'équations. □

La proposition 4-2 permet de réduire le problème de l'évaluation d'une formule XTL φ à celui de la résolution (partielle) d'un SEMP. Pour préciser les idées, considérons une formule φ d'alternance 1, de point fixe (toute formule peut être amenée sous cette forme en la préfixant par un plus petit point fixe **mu** Y_0 définissant une nouvelle variable Y_0), normalisée et fermée d.p.d.v. des variables propositionnelles. Supposons que $\varphi = \sigma Y_1\,(x_1^1{:}T_1^1{:=}E_1^1,\ldots,x_1^{n_1}{:}T_1^{n_1}{:=}E_1^{n_1}).\varphi_1$ avec $bpv(\varphi) = \{Y_1,\ldots,Y_n\}$ et soit $\varepsilon \in \mathbf{DEnv}$ tel que $fdv(\varphi) \subseteq supp(\varepsilon)$. Appliquée à φ, à un environnement propositionnel vide et à ε, la proposition 4-2 conduit au résultat suivant :

$$[\![\sigma Y_1\,(x_1^1{:}T_1^1{:=}E_1^1,\ldots,x_1^{n_1}{:}T_1^{n_1}{:=}E_1^{n_1}).\varphi_1]\!][\,]\varepsilon = F_1\left([\![E_1^1]\!]\varepsilon,\ldots,[\![E_1^{n_1}]\!]\varepsilon\right)$$

où $(F_1,\ldots,F_n) = [\![\mathrm{TRM}(\varphi)]\!]\,[\,]$. Le calcul de la sémantique de φ est donc réduit au calcul de l'instance $F_1(v_1^1,\ldots,v_1^{n_1})$, où $F_1 : T_1^1\times\cdots\times T_1^{n_1} \to 2^S$ est la composante[13] de la sémantique de $\mathrm{TRM}(\varphi)$ associée à Y_1 et $v_1^1 \in T_1^1,\ldots,v_1^{n_1} \in T_1^{n_1}$ sont les valeurs des expressions $E_1^1,\ldots,E_1^{n_1}$ calculées dans le contexte de ε. L'évaluation globale et l'évaluation locale de φ se réduisent respectivement au calcul de $F_1(v_1^1,\ldots,v_1^{n_1})$ et au test $s_{init} \in F_1(v_1^1,\ldots,v_1^{n_1})$.

Cette résolution partielle des SEMPs peut être effectuée à partir de leur définition sémantique. Considérons un SEMP fermé $SM = \{Y_i(x_i^1{:}T_i^1,\ldots,x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma_i}{=} \varphi_i\}_{1\leq i\leq n}$ d'alternance 1 et soit $\{B_1,\ldots,B_p\}$ la partition canonique de SM en σ-blocs, triés topologiquement de façon à ce que $fpv(B_j) \subseteq \bigcup_{k=1}^{j-1} bpv(B_k)$ pour tout $1 \leq j \leq p$. Soit B_j le σ-bloc qui définit Y_1, c'est-à-dire qui vérifie $Y_1 \in bpv(B_j)$. Conformément à la définition sémantique des SEMPs (égalités (4.5) et (4.6)), la composante F_1 de la sémantique de SM est égale à la composante respective de la sémantique de B_j, définie dans le contexte des blocs $B_1,\ldots,B_{j-1}$:

$$F_1 = ([\![SM]\!]\,[\,])_1 \stackrel{\mathrm{d}}{=} ([\![B_j]\!]\,\rho_j)_1$$

où $\rho_1 \stackrel{\mathrm{d}}{=} [\,]$ et $\rho_{j+1} \stackrel{\mathrm{d}}{=} \rho_j \oslash [F_j^1/Y_j^1,\ldots,F_j^{r_j}/Y_j^{r_j}]$ pour tout $1 \leq j < p$.

Ce schéma de calcul est utilisé dans les algorithmes d'évaluation du μ-calcul standard d'alternance 1 basés sur des systèmes d'équations modales [CS93, And94] : la sémantique d'un σ-bloc B_j est calculée explicitement, en évaluant d'abord les sémantiques de tous les σ-blocs $B_1,\ldots,B_{j-1}$. Pour les formules XTL, la situation est plus compliquée : les domaines des paramètres des opérateurs de point fixe pouvant être infinis, il est impossible de calculer explicitement la sémantique des blocs $B_1,\ldots,B_{j-1}$ (par ailleurs, ceci n'est pas nécessaire, puisqu'il s'agit de calculer uniquement l'instance $F_1(v_1^1,\ldots,v_1^{n_1})$ et non pas la composante F_1 dans sa totalité).

La méthode de résolution que nous proposons utilise comme opération primitive l'évaluation d'une instance $F_k(v_k^1,\ldots,v_k^{r_k})$ appartenant à un σ-bloc B_j dans le contexte ρ_j des autres blocs $B_1,\ldots,B_{j-1}$. Le contexte ρ_j est représenté implicitement : chaque fois que la valeur d'une composante $(\rho_j(F_l))(v_l^1,\ldots,v_l^{n_l})$ est nécessaire (où $F_l \in fpv(B_j)$), celle-ci est calculée en évaluant partiellement le σ-bloc B_m définissant Y_l. Les blocs étant triés topologiquement, chaque bloc B_j ne dépend que des blocs B_m avec $1 \leq m < j$, ce qui assure (sous réserve du fait que chaque évaluation d'instance d'un σ-bloc termine) la terminaison du processus d'évaluation.

Remarque 4-1
Une implémentation naturelle et efficace de cette méthode de résolution peut être réalisée au moyen de *coroutines*. Ce schéma associe à chaque σ-bloc B_j du SEMP une coroutine chargée de calculer les instances $F_k(v_k^1,\ldots,v_k^{r_k})$ des variables définies dans B_j, au fur et à mesure qu'elles sont requises par d'autres σ-blocs qui dépendent de B_j. Grâce aux mécanismes de sauvegarde du contexte, l'utilisation de coroutines (ou d'autres constructions équivalentes) permet d'optimiser l'évaluation des σ-blocs, en gardant et en réutilisant les instances déjà calculées. ■

[13] La composante de la solution de $\mathrm{TRM}(\varphi)$ recherchée correspond à la variable de point fixe située à la racine de φ.

L'évaluation d'un Semp fermé SM d'alternance 1 peut donc être réduite à la résolution partielle d'un σ-bloc B_j dans un contexte $\rho_j \in \mathbf{PEnv}$ tel que $\mathit{fpv}(B_j) \subseteq \mathit{supp}(\rho_j)$, tout en sachant que ρ_j est représenté implicitement, car il est calculé à partir des autres σ-blocs de SM. La résolution partielle des σ-blocs sera traitée aux sections 4.2.3 et 4.2.4.

Simplification des SEMPs La complexité de la résolution des Semps dépend directement de la taille des systèmes d'équations booléennes paramétrées (Sebps) vers lesquels ils seront traduits (voir la section 4.2.4). Afin d'obtenir une taille des Sebps qui soit linéaire par rapport à la taille du Ste (nombre d'états et de transitions) et des Semps (nombre de variables et d'opérateurs contenus dans les formules en partie droite des équations), les Semps doivent être *simplifiés* avant d'être traduits vers des Sebps.

La simplification d'un Semp SM consiste à substituer les sous-formules contenues dans les parties droites des équations de SM par de nouvelles variables propositionnelles, définies en rajoutant de nouvelles équations à SM, de façon à ce que chaque formule du Semp SM' obtenu ne contienne qu'au plus un seul opérateur. Intuitivement, ceci permet de factoriser (et, par conséquent, de mieux réutiliser) les sous-formules modales "$\langle\ \rangle$" et "$[\ ]$" contenues en partie droite des équations de SM. Proposée en [AC88], cette approche est utilisée dans pratiquement tous les algorithmes d'évaluation dédiés au μ-calcul standard d'alternance 1 qui procèdent par traduction des formules vers des systèmes d'équations modales ou booléennes [CS91b, VL92, CS93, And94, VWL94, VL94].

La simplification des Semps produits par traduction des formules φ peut être effectuée au moyen de transformations syntaxiques simples, que nous ne détaillons pas ici. En revanche, nous illustrons cette transformation au moyen d'un exemple.

Exemple 4-6
Nous reprenons le Semp SM de l'exemple 4-5 :

$$SM = \left\{ \begin{array}{rcl} Y_1 & \stackrel{\nu}{=} & [\texttt{SEND ?}\ m_1 : \texttt{Msg}]\ Y_2(m_1)\ \mathbf{and}\ [\mathbf{true}]\ Y_1 \\ Y_2\ (m_2 : \texttt{Msg}) & \stackrel{\mu}{=} & \langle \texttt{RECV !}\ m_2 \rangle\ \mathbf{true\ or}\ \langle \mathbf{not}\ (\texttt{SEND}\ \mathbf{any}) \rangle\ Y_2\ (m_2) \end{array} \right.$$

La simplification de SM conduit au Semp SM' suivant :

$$SM' = \left\{ \begin{array}{rcl} Y_1 & \stackrel{\nu}{=} & Y_3\ \mathbf{and}\ Y_4 \\ Y_3 & \stackrel{\nu}{=} & [\texttt{SEND ?}\ m_1 : \texttt{Msg}]\ Y_2\ (m_1) \\ Y_4 & \stackrel{\nu}{=} & [\mathbf{true}]\ Y_1 \\ Y_2\ (m_2 : \texttt{Msg}) & \stackrel{\mu}{=} & Y_5\ (m_2)\ \mathbf{or}\ Y_6\ (m_2) \\ Y_5\ (m_5 : \texttt{Msg}) & \stackrel{\mu}{=} & \langle \texttt{RECV !}\ m_5 \rangle\ \mathbf{true} \\ Y_6\ (m_6 : \texttt{Msg}) & \stackrel{\mu}{=} & \langle \mathbf{not}\ (\texttt{SEND}\ \mathbf{any}) \rangle\ Y_2\ (m_6) \end{array} \right.$$

La simplification de SM a nécessité l'introduction des variables propositionnelles Y_3, Y_4, Y_5 et Y_6, ainsi que le renommage de la variable simple m_2, qui était libre dans les sous-formules "$\langle \texttt{RECV !}\ m_2 \rangle\ \mathbf{true}$" et "$\langle \mathbf{not}\ (\texttt{SEND}\ \mathbf{any}) \rangle\ Y_2\ (m_2)$", devenues respectivement les parties droites des équations associées à Y_5 et Y_6 dans SM'. Le ν-bloc $B_1' = \{Y_1, Y_3, Y_4\}$ et le μ-bloc $B_2' = \{Y_2, Y_5, Y_6\}$, produits par simplification des blocs B_1 et B_2 de SM, constituent une partition acyclique du graphe de dépendance associé à SM'. ■

Il est aisé de montrer que la simplification préserve la sémantique des Semps. La taille du Semp simplifié SM' ne peut excéder que d'un facteur linéaire la taille du Semp SM (qui, à son tour, ne peut pas dépasser la taille de la formule initiale φ) : le nombre d'opérateurs en partie droite des équations de SM' est égal à celui de SM et le nombre de variables propositionnelles rajoutées à SM' ne peut pas excéder le nombre d'opérateurs contenus dans SM.

4.2.3 Transformation en systèmes d'équations booléennes paramétrées

Généralisant l'approche utilisée dans plusieurs algorithmes efficaces dédiés au μ-calcul standard d'alternance 1 [VL92, VL94, And94], nous effectuons l'évaluation partielle d'un σ-bloc en le traduisant vers un système d'équations booléennes paramétrées (SEBP). Nous définissons d'abord les formules booléennes utilisées dans les SEBPs, ensuite nous présentons la syntaxe et la sémantique des SEBPs et finalement nous donnons la traduction des σ-blocs vers des SEBPs.

Formules booléennes Nous introduisons les domaines syntaxiques suivants :

- *BVar* est le domaine des *variables booléennes.* Une variable booléenne $Z(x_1{:}T_1, \ldots, x_n{:}T_n) \in$ *BVar*, paramétrée par les variables simples $x_1, \ldots, x_n$ ayant respectivement les types $T_1, \ldots, T_n$, dénote un prédicat défini sur $T_1 \times \cdots \times T_n$.

- *BForm* est le domaine des *formules booléennes.* Les formules booléennes $\psi \in$ *BForm* (données directement en forme positive, c'est-à-dire sans négations) ont la syntaxe suivante :

$$
\begin{array}{lll}
\psi & ::= & \textbf{true} \\
& | & \textbf{false} \\
& | & Z\ (E_1, \ldots, E_n) \\
& | & \psi_1 \textbf{ or } \psi_2 \\
& | & \psi_1 \textbf{ and } \psi_2 \\
& | & \textbf{exists } x_0{:}T_0 \ [\textbf{among } E_0], \ldots, x_n{:}T_n \ [\textbf{among } E_n] \textbf{ in } \psi_1 \\
& | & \textbf{forall } x_0{:}T_0 \ [\textbf{among } E_0], \ldots, x_n{:}T_n \ [\textbf{among } E_n] \textbf{ in } \psi_1 \\
& | & \textbf{case } E_0 \textbf{ in} \\
& & \quad P_1^0 \mid \ldots \mid P_1^{n_1} \ [\textbf{where } E_1] \ \text{->} \ \psi_1 \\
& & \quad \ldots \\
& & \mid P_m^0 \mid \ldots \mid P_m^{n_m} \ [\textbf{where } E_m] \ \text{->} \ \psi_m \\
& & [\mid \textbf{otherwise} \ \text{->} \ \psi_{m+1}] \\
& & \textbf{endcase} \\
& | & \textbf{case action } E_0 \textbf{ in} \\
& & \quad \alpha_1 \ [\textbf{where } E_1] \ \text{->} \ \psi_1 \\
& & \quad \ldots \\
& & \mid \alpha_m \ [\textbf{where } E_m] \ \text{->} \ \psi_m \\
& & [\mid \textbf{otherwise} \ \text{->} \ \psi_{m+1}] \\
& & \textbf{endcase}
\end{array}
$$

Les opérateurs booléens, les quantificateurs et les constructions "**case**" et "**case action**" sont similaires aux opérateurs correspondants utilisés dans les formules φ sur états. Une formule "$Z\ (E_1, \ldots, E_n)$" dénote l'appel d'une variable booléenne Z avec les arguments $E_1, \ldots, E_n$. Les variables (booléennes et simples) libres dans une formule ψ sont définies respectivement par les fonctions syntaxiques suivantes :

$$
\begin{array}{l}
\mathit{fbv} : \mathit{BForm} \to \mathit{BVar} \\
\mathit{fdv} : \mathit{BForm} \to \mathit{DVar}
\end{array}
$$

Ces fonctions sont données dans la table 4.2 (où **quantif** dénote les quantificateurs). Une formule ψ ne contient pas d'occurrences liées de variables booléennes.

ψ	$\mathit{fdv}(\psi)$	$\mathit{fbv}(\psi)$
true, **false**	$\emptyset$	$\emptyset$
Z $(E_1, \ldots, E_n)$	$\bigcup_{i=1}^{n} \mathit{fdv}(E_i)$	$\{Z\}$
ψ_1 **or** ψ_2, ψ_1 **and** ψ_2	$\mathit{fdv}(\psi_1) \cup \mathit{fdv}(\psi_2)$	$\mathit{fbv}(\psi_1) \cup \mathit{fbv}(\psi_2)$
quantif x_0**:**T_0 [**among** E_0] , . . . , x_n**:**T_n [**among** E_n] **in** ψ	$(\mathit{fdv}(\psi) \setminus \bigcup_{i=0}^{n} \{x_i\}) \cup$ $\bigcup_{i=0}^{n} \mathit{fdv}(E_i)$	$\mathit{fbv}(\psi)$
case E_0 **in** P_1^0 | . . . | $P_1^{n_1}$ [**where** E_1] -> ψ_1 . . . | P_m^0 | . . . | $P_m^{n_m}$ [**where** E_m] -> ψ_m [| **otherwise** -> ψ_{m+1}] **endcase**	$((\bigcup_{i=1}^{m} \mathit{fdv}(E_i) \cup \bigcup_{j=1}^{m+1} \mathit{fdv}(\psi_j)) \setminus$ $\bigcup_{k=1}^{m} \mathit{bdv}(P_k^0)) \cup \mathit{fdv}(E_0)$	$\bigcup_{j=1}^{m+1} \mathit{fbv}(\psi_j)$
case action E_0 **in** α_1 [**where** E_1] -> ψ_1 . . . | α_m [**where** E_m] -> ψ_m [| **otherwise** -> ψ_{m+1}] **endcase**	$((\bigcup_{i=1}^{m} \mathit{fdv}(E_i) \cup \bigcup_{j=1}^{m+1} \mathit{fdv}(\psi_j)) \setminus$ $\bigcup_{k=1}^{m} v_{tt}(\alpha_k)) \cup \mathit{fdv}(E_0)$	$\bigcup_{j=1}^{m+1} \mathit{fbv}(\psi_j)$

Table 4.2: Variables libres dans les formules booléennes

Nous introduisons aussi les domaines sémantiques suivants :

- $\langle \mathbf{Bool}, \vee, \wedge, \Rightarrow \rangle$ est le domaine des valeurs booléennes utilisé pour interpréter les formules ψ. Ce domaine a une structure de treillis complet (**ff** et **tt** étant respectivement le plus petit et le plus grand élément du treillis). De ce point de vue, il est différent du domaine **Bool** défini à la section 3.1.1, qui est utilisé dans l'interprétation des constructions XTL (formules sur actions, formules sur états, expressions, ...), celui-ci étant un ordre partiel complet dans lequel les éléments **ff** et **tt** ne sont pas comparables selon la relation d'ordre. Par souci de simplicité, nous utiliserons les mêmes notations pour les deux domaines sémantiques, les ambiguïtés étant résolues suivant le contexte.

- $\mathbf{BEnv} = BVar \to (\mathbf{Param} \to \mathbf{Bool})$ est le domaine des *environnements booléens*. Un environnement booléen $\delta \in \mathbf{BEnv}$ est une application partielle associant à chaque variable booléenne $Z(x_1{:}T_1, \ldots, x_n{:}T_n) \in \mathit{supp}(\delta)$ un prédicat $\delta(Z) : T_1 \times \cdots \times T_n \to \mathbf{Bool}$.

  La structure de treillis complet de $\langle \mathbf{Bool}, \vee, \wedge, \Rightarrow \rangle$ induit une structure de treillis complet pour le domaine $\langle \mathbf{Param} \to \mathbf{Bool}, \sqcup, \sqcap, \sqsubseteq \rangle$, où les opérations $\sqcup$, $\sqcap$ et la relation d'ordre $\sqsubseteq$ sont respectivement définies par extension de $\vee$, $\wedge$ et $\Rightarrow$.

La sémantique des formules booléennes est définie par la fonction d'interprétation suivante :

$$[\![.]\!] : BForm \to \mathbf{BEnv} \to \mathbf{DEnv} \to \mathbf{Bool}$$

Etant donné une formule ψ, un environnement δ tel que $\mathit{fbv}(\psi) \subseteq \mathit{supp}(\delta)$ et un environnement ε tel que $\mathit{fdv}(\psi) \subseteq \mathit{supp}(\psi)$, la dénotation $[\![\psi]\!]\,\delta\varepsilon$ renvoie **tt** ssi ψ est satisfaite dans le contexte de δ et ε. La fonction sémantique est définie inductivement comme suit :

$$\llbracket \textbf{true} \rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \textbf{tt}$$
$$\llbracket \textbf{false} \rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \textbf{ff}$$
$$\llbracket Z\ (E_1, \ldots, E_n) \rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} (\delta(Z))(\llbracket E_1 \rrbracket \varepsilon, \ldots, \llbracket E_n \rrbracket \varepsilon)$$
$$\llbracket \psi_1 \textbf{ or } \psi_2 \rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \llbracket \psi_1 \rrbracket \delta\varepsilon \vee \llbracket \psi_2 \rrbracket \delta\varepsilon$$
$$\llbracket \psi_1 \textbf{ and } \psi_2 \rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \llbracket \psi_1 \rrbracket \delta\varepsilon \wedge \llbracket \psi_2 \rrbracket \delta\varepsilon$$

$$\left\llbracket \begin{array}{l} \textbf{exists } x_0{:}T_0\ [\textbf{among } E_0] \\ \quad ,\ldots, \\ \quad x_n{:}T_n\ [\textbf{among } E_n] \\ \textbf{in } \psi \end{array} \right\rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \mathit{if}\ \exists v_0{:}T_0\,[\in \llbracket E_0 \rrbracket \varepsilon] \ldots \exists v_n{:}T_n\,[\in \llbracket E_n \rrbracket \varepsilon]\,. \\ \quad \llbracket \psi \rrbracket\, \delta(\varepsilon \oslash [v_0/x_0, \ldots, v_n/x_n]) = \textbf{tt} \\ \mathit{then}\ \textbf{tt}\ \mathit{else}\ \textbf{ff} \\ \mathit{endif} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{forall } x_0{:}T_0\ [\textbf{among } E_0] \\ \quad ,\ldots, \\ \quad x_n{:}T_n\ [\textbf{among } E_n] \\ \textbf{in } \psi \end{array} \right\rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \mathit{if}\ \forall v_0{:}T_0\,[\in \llbracket E_0 \rrbracket \varepsilon] \ldots \forall v_n{:}T_n\,[\in \llbracket E_n \rrbracket \varepsilon]\,. \\ \quad \llbracket \psi \rrbracket\, \delta(\varepsilon \oslash [v_0/x_0, \ldots, v_n/x_n]) = \textbf{tt} \\ \mathit{then}\ \textbf{tt}\ \mathit{else}\ \textbf{ff} \\ \mathit{endif} \end{array}$$

$$\left\llbracket \begin{array}{l} \textbf{case } E_0 \textbf{ in} \\ \quad P_1^0\ |\ldots|\ P_1^{n_1} \\ \qquad [\textbf{where } E_1]\ \texttt{->}\ \psi_1 \\ \quad \ldots \\ |\ P_m^0\ |\ldots|\ P_m^{n_m} \\ \qquad [\textbf{where } E_m]\ \texttt{->}\ \psi_m \\ {[}|\ \textbf{otherwise}\ \texttt{->}\ \psi_{m+1}] \\ \textbf{endcase} \end{array} \right\rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \mathit{let}\ v_0 : \mathit{type}(E_0) := \llbracket E_0 \rrbracket \varepsilon\ \mathit{in} \\ \quad \mathit{if}\ \exists i \in [1,m].\exists j \in [0,n_i].\mathit{ok}_i^j\ \wedge \\ \qquad \forall l \in [0,i-1].\forall k \in [0,n_l].\neg \mathit{ok}_l^k \\ \quad \mathit{then}\ \llbracket \psi_i \rrbracket\, \delta(\varepsilon \oslash (\llbracket P_i^j \rrbracket v_0)_2) \\ \quad [\mathit{else}\ \llbracket \psi_{m+1} \rrbracket\, \delta\varepsilon\ ] \\ \quad \mathit{endif} \\ \mathit{endlet} \end{array}$$

$$\text{où } \mathit{ok}_i^j \stackrel{\mathrm{d}}{=} (\llbracket P_i^j \rrbracket v_0)_1 = \textbf{tt}\,[\ \wedge\ \llbracket E_i \rrbracket\,(\varepsilon \oslash (\llbracket P_i^j \rrbracket v_0)_2) = \textbf{tt}\ ]$$

$$\left\llbracket \begin{array}{l} \textbf{case action } E_0 \textbf{ in} \\ \quad \alpha_1\ [\textbf{where } E_1]\ \texttt{->}\ \psi_1 \\ \quad \ldots \\ |\ \alpha_m\ [\textbf{where } E_m]\ \texttt{->}\ \psi_m \\ {[}|\ \textbf{otherwise}\ \texttt{->}\ \psi_{m+1}] \\ \textbf{endcase} \end{array} \right\rrbracket \delta\varepsilon \stackrel{\mathrm{d}}{=} \begin{array}{l} \mathit{let}\ a_0 \in A := \llbracket E_0 \rrbracket \varepsilon\ \mathit{in} \\ \quad \mathit{if}\ \exists i \in [1,m].\exists \varepsilon_i \in (\llbracket \alpha_i \rrbracket\, \varepsilon a_0)_2.\mathit{ok}_i(\varepsilon_i)\ \wedge \\ \qquad \forall j \in [1,i-1].\forall \varepsilon_j \in (\llbracket \alpha_j \rrbracket\, \varepsilon a_0)_2.\neg \mathit{ok}_j(\varepsilon_j) \\ \quad \mathit{then}\ \llbracket \psi_i \rrbracket\, \delta(\varepsilon \oslash \varepsilon_i) \\ \quad [\mathit{else}\ \llbracket \psi_{m+1} \rrbracket\, \delta\varepsilon\ ] \\ \quad \mathit{endif} \\ \mathit{endlet} \end{array}$$

$$\text{où } \mathit{ok}_i(\varepsilon_i) \stackrel{\mathrm{d}}{=} (\llbracket \alpha_i \rrbracket\, \varepsilon a_0)_1 = \textbf{tt}\,[\ \wedge\ \llbracket E_i \rrbracket\,(\varepsilon \oslash \varepsilon_i) = \textbf{tt}\ ].$$

Il est aisé de vérifier (par induction structurelle sur ψ) que la sémantique des formules booléennes est monotone sur **Param** $\to$ **Bool**, c'est-à-dire que pour toute formule $\psi \in \mathit{BForm}$, variable $Z(x_1{:}T_1, \ldots, x_n{:}T_n) \in \mathit{BVar}$, environnements $\delta \in \mathbf{BEnv}$ et $\varepsilon \in \mathbf{DEnv}$ tels que $\mathit{fbv}(\psi) \subseteq \mathit{supp}(\delta)$ et $\mathit{fdv}(\psi) \subseteq \mathit{supp}(\varepsilon)$, et fonctions $G_1, G_2 : T_1 \times \cdots \times T_n \to \mathbf{Bool}$:

$$G_1 \sqsubseteq G_2 \Rightarrow \big(\llbracket \psi \rrbracket\,(\delta \oslash [G_1/Z])\varepsilon \Rightarrow \llbracket \psi \rrbracket\,(\delta \oslash [G_2/Z])\varepsilon \big).$$

Systèmes d'équations booléennes paramétrées Un système d'équations booléennes paramétrées (Sebp) SB est un ensemble d'équations de point fixe ayant la syntaxe suivante :

$$\big\{ Z_k(x_k^1{:}T_k^1, \ldots, x_k^{n_k}{:}T_k^{n_k}) \stackrel{\sigma}{=} \psi_k \big\}_{1 \le k \le p}$$

où $\sigma \in \{\mu, \nu\}$ est le *signe* du système et, pour chaque $1 \le k \le p$, $Z_k(x_k^1{:}T_k^1, \ldots, x_k^{n_k}{:}T_k^{n_k}) \in \mathit{BVar}$ est une variable booléenne et $\psi_k \in \mathit{BForm}$ dénote une formule booléenne. Nous supposons que, pour chaque équation $1 \le k \le p$, $\mathit{fdv}(\psi_k) \subseteq \{x_k^1, \ldots, x_k^{n_k}\}$.

Le domaine syntaxique associé aux SEBPs est noté $BSys$. Les variables booléennes libres et liées dans un SEBP $SB \in BSys$ sont définies respectivement par les fonctions syntaxiques suivantes :

$$fbv, bbv : BSys \to BVar$$

Ces fonctions sont données dans la table 4.3. Un SEBP SB ne contient pas de variables simples libres. Un SEBP SB est dit *fermé* ssi $fbv(SB) = \emptyset$.

SB	$fbv(SB)$	$bbv(SB)$
$\{Z_k(x_k^1{:}T_k^1, \ldots, x_k^{n_k}{:}T_k^{n_k}) \stackrel{\sigma}{=} \psi_k\}_{1 \leq k \leq p}$	$\bigcup_{k=1}^{p} (fbv(\psi_k) \setminus \{Z_k\})$	$\{Z_1, \ldots, Z_p\}$

Table 4.3: Variables booléennes libres et liées dans les SEBPs

La sémantique des SEBPs ayant $p \geq 1$ équations est définie par la fonction d'interprétation suivante :

$$[\![.]\!] : BSys \to \mathbf{BEnv} \to (\mathbf{Param} \to \mathbf{Bool})^p$$

Etant donné un SEBP SB et un environnement booléen δ tel que $fbv(SB) \subseteq supp(\delta)$, la dénotation $[\![SB]\!]\,\delta$ renvoie un tuple de fonctions $(G_k)_{1 \leq k \leq p}$, où $G_k : T_k^1 \times \cdots \times T_k^{n_k} \to \mathbf{Bool}$ pour $1 \leq k \leq p$, représentant la solution de SB dans le contexte de δ. La fonction sémantique est définie ci-dessous :

$$\left[\!\!\left[\{Z_k(x_k^1{:}T_k^1, \ldots, x_k^{n_k}{:}T_k^{n_k}) \stackrel{\sigma}{=} \psi_k\}_{1 \leq k \leq p}\right]\!\!\right] \delta \stackrel{\mathrm{d}}{=} \sigma \overline{\Psi}_\delta$$

où $\overline{\Psi}_\delta : (\mathbf{Param} \to \mathbf{Bool})^p \to (\mathbf{Param} \to \mathbf{Bool})^p$ est la fonctionnelle associée à SB :

$$\overline{\Psi}_\delta(G_1, \ldots, G_p) \stackrel{\mathrm{d}}{=} (\lambda v_k^1{:}T_k^1, \ldots, v_k^{n_k}{:}T_k^{n_k}.\,[\![\psi_k]\!]\,(\delta \oslash [G_1/Z_1, \ldots, G_p/Z_p])[v_k^1/x_k^1, \ldots, v_k^{n_k}/x_k^{n_k}])_{1 \leq k \leq p}$$

pour $G_k : T_k^1 \times \cdots \times T_k^{n_k} \to \mathbf{Bool}$ $(1 \leq k \leq p)$. Puisque la fonctionnelle $\overline{\Psi}_\delta$ est monotone (toutes les formules ψ_k étant monotones) et le treillis $\langle (\mathbf{Param} \to \mathbf{Bool})^p, \sqcup, \sqcap, \sqsubseteq \rangle$ est complet (les opérations $\sqcup$ et $\sqcap$ et la relation d'ordre $\sqsubseteq$ étant définies par extension des opérations correspondantes sur $\mathbf{Param} \to \mathbf{Bool}$), le théorème de Tarski assure l'existence et l'unicité des points fixes $\mu\overline{\Psi}_\delta$ et $\nu\overline{\Psi}_\delta$.

Traduction des SEMPs vers des SEBPs La traduction des σ-blocs vers des SEBPs est effectuée au moyen des fonctions suivantes :

$$\begin{aligned} \mathrm{TrMb} &: MSys \to BSys \\ \mathrm{TrB} &: SForm \times S \to BForm \end{aligned}$$

où S est l'ensemble d'états du modèle STE sur lequel le σ-bloc est évalué. Etant donné un σ-bloc $SM = \{Y_i(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma}{=} \varphi_i\}_{1 \leq i \leq n}$ et un modèle STE $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$, la traduction $\mathrm{TrMb}(SM)$ renvoie un SEBP SB de signe σ ayant une variable booléenne $Z_{i,s}$ associée à chaque couple (Y_i, s), pour $1 \leq i \leq n$ et $s \in S$:

$$\mathrm{TrMb}\big(\{Y_i(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma}{=} \varphi_i\}_{1 \leq i \leq n}\big) \stackrel{\mathrm{d}}{=} \{Z_{i,s}(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma}{=} \mathrm{TrB}(\varphi_i, s)\}_{1 \leq i \leq n, s \in S}$$

Intuitivement, $Z_{i,s}(v_i^1, \ldots, v_i^{n_i})$ renvoie **tt** ssi l'état s satisfait $Y_i(v_i^1, \ldots, v_i^{n_i})$. La fonction TrB, qui traduit une formule modale, dans le contexte d'un état $s \in S$, vers une formule booléenne, est définie inductivement comme suit :

$$
\begin{array}{rcl}
\textsc{TrB}(\mathbf{true}, s) & \stackrel{\mathrm{d}}{=} & \mathbf{true} \\
\textsc{TrB}(\mathbf{false}, s) & \stackrel{\mathrm{d}}{=} & \mathbf{false} \\
\textsc{TrB}(Y_i\ (E_i^1, \ldots, E_i^{n_i}), s) & \stackrel{\mathrm{d}}{=} & Z_{i,s}(E_i^1[s/c\_s], \ldots, E_i^{n_i}[s/c\_s]) \\
\textsc{TrB}(\varphi_1\ \mathbf{or}\ \varphi_2, s) & \stackrel{\mathrm{d}}{=} & \textsc{TrB}(\varphi_1, s)\ \mathbf{or}\ \textsc{TrB}(\varphi_2, s) \\
\textsc{TrB}(\varphi_1\ \mathbf{and}\ \varphi_2, s) & \stackrel{\mathrm{d}}{=} & \textsc{TrB}(\varphi_1, s)\ \mathbf{and}\ \textsc{TrB}(\varphi_2, s) \\
\textsc{TrB}(\langle\, \alpha \,\rangle\ \varphi, s) & \stackrel{\mathrm{d}}{=} & \mathbf{or}_{s \xrightarrow{a} s'} \left(\begin{array}{l} \mathbf{case\ action}\ a\ \mathbf{in} \\ \quad \alpha \ \texttt{->}\ \textsc{TrB}(\varphi, s')[a/c\_a] \\ \mid \mathbf{otherwise} \ \texttt{->}\ \mathbf{false} \\ \mathbf{endcase} \end{array} \right) \\
\textsc{TrB}([\,\alpha\,]\ \varphi, s) & \stackrel{\mathrm{d}}{=} & \mathbf{and}_{s \xrightarrow{a} s'} \left(\begin{array}{l} \mathbf{case\ action}\ a\ \mathbf{in} \\ \quad \alpha \ \texttt{->}\ \textsc{TrB}(\varphi, s')[a/c\_a] \\ \mid \mathbf{otherwise} \ \texttt{->}\ \mathbf{true} \\ \mathbf{endcase} \end{array} \right) \\
\textsc{TrB}\left(\begin{array}{l} \mathbf{exists}\ x_0{:}T_0\ \ [\mathbf{among}\ E_0] \\ \qquad , \ldots , \\ \qquad x_n{:}T_n\ [\mathbf{among}\ E_n] \\ \mathbf{in}\ \varphi \end{array} , s \right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l} \mathbf{exists}\ x_0{:}T_0\ \ [\mathbf{among}\ E_0[s/c\_s]] \\ \qquad , \ldots , \\ \qquad x_n{:}T_n\ [\mathbf{among}\ E_n[s/c\_s]] \\ \mathbf{in}\ \textsc{TrB}(\varphi, s) \end{array} \\
\textsc{TrB}\left(\begin{array}{l} \mathbf{forall}\ x_0{:}T_0\ \ [\mathbf{among}\ E_0] \\ \qquad , \ldots , \\ \qquad x_n{:}T_n\ [\mathbf{among}\ E_n] \\ \mathbf{in}\ \varphi \end{array} , s \right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l} \mathbf{forall}\ x_0{:}T_0\ \ [\mathbf{among}\ E_0[s/c\_s]] \\ \qquad , \ldots , \\ \qquad x_n{:}T_n\ [\mathbf{among}\ E_n[s/c\_s]] \\ \mathbf{in}\ \textsc{TrB}(\varphi, s) \end{array} \\
\textsc{TrB}\left(\begin{array}{l} \mathbf{case}\ E_0\ \mathbf{in} \\ \quad P_1^0\ \mid \ldots \mid P_1^{n_1} \\ \qquad [\mathbf{where}\ E_1]\ \texttt{->}\ \varphi_1 \\ \quad \ldots \\ \mid P_m^0\ \mid \ldots \mid P_m^{n_m} \\ \qquad [\mathbf{where}\ E_m]\ \texttt{->}\ \varphi_m \\ [\mid \mathbf{otherwise}\ \texttt{->}\ \varphi_{m+1}] \\ \mathbf{endcase} \end{array} , s \right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l} \mathbf{case}\ E_0[s/c\_s]\ \mathbf{in} \\ \quad P_1^0\ \mid \ldots \mid P_1^{n_1} \\ \qquad [\mathbf{where}\ E_1[s/c\_s]]\ \texttt{->}\ \textsc{TrB}(\varphi_1, s) \\ \quad \ldots \\ \mid P_m^0\ \mid \ldots \mid P_m^{n_m} \\ \qquad [\mathbf{where}\ E_m[s/c\_s]]\ \texttt{->}\ \textsc{TrB}(\varphi_m, s) \\ [\mid \mathbf{otherwise}\ \texttt{->}\ \textsc{TrB}(\varphi_{m+1}, s)] \\ \mathbf{endcase} \end{array} \\
\textsc{TrB}\left(\begin{array}{l} \mathbf{case\ action}\ E_0\ \mathbf{in} \\ \quad \alpha_1\ \ [\mathbf{where}\ E_1]\ \texttt{->}\ \varphi_1 \\ \quad \ldots \\ \mid \alpha_m\ [\mathbf{where}\ E_m]\ \texttt{->}\ \varphi_m \\ [\mid \mathbf{otherwise}\ \texttt{->}\ \varphi_{m+1}] \\ \mathbf{endcase} \end{array} , s \right) & \stackrel{\mathrm{d}}{=} & \begin{array}{l} \mathbf{case\ action}\ E_0[s/c\_s]\ \mathbf{in} \\ \quad \alpha_1\ \ [\mathbf{where}\ E_1[s/c\_s]]\ \texttt{->}\ \textsc{TrB}(\varphi_1, s) \\ \quad \ldots \\ \mid \alpha_m\ [\mathbf{where}\ E_m[s/c\_s]]\ \texttt{->}\ \textsc{TrB}(\varphi_m, s) \\ [\mid \mathbf{otherwise}\ \texttt{->}\ \textsc{TrB}(\varphi_{m+1}, s)] \\ \mathbf{endcase} \end{array}
\end{array}
$$

où $E[s/c\_s]$ dénote la substitution dans E des occurrences de l'opérateur "**current**" sur états par s (valeur de type `state`) et $\psi[a/c\_a]$ dénote la substitution dans ψ des occurrences de l'opérateur "**current**" sur actions par a (valeur de type `label`). Lorsque l'ensemble des transitions issues de s est vide, les opérateurs généralisés "$\mathbf{or}_{s \xrightarrow{a} s'}$" et "$\mathbf{and}_{s \xrightarrow{a} s'}$" renvoient respectivement **false** et **true**.

Remarque 4-2

Pour un σ-bloc simplifié SM (voir la section 4.2.2), les formules en partie droite des équations ne contiennent que des sous-formules atomiques. Ceci assure la même propriété pour les formules booléennes $\textsc{TrB}(\varphi, s)$, à l'exception des formules obtenues par traduction des modalités "$\langle\ \rangle$" et "[ ]", qui néanmoins ne contiennent que des opérateurs "**or**" ou "**and**" et des opérateurs "**case**" et "**case action**" ayant des sous-formules atomiques. ■

Exemple 4-7
Considérons le SEMP simplifié SM' de l'exemple 4-6 et le modèle STE étendu illustré dans la figure 4.5 (s_1 étant l'état initial).

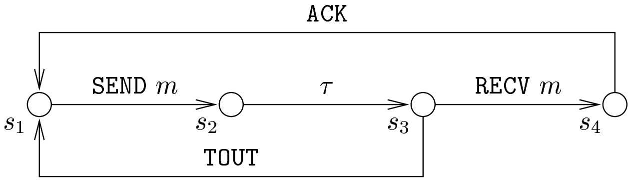

Figure 4.5: Exemple de modèle STE

Les traductions $\text{TrMb}(B'_1)$ et $\text{TrMb}(B'_2)$ des σ-blocs de SM' dans le contexte de cet STE produisent les SEBPS SB_1 et SB_2 ci-dessous (les mots-clés "**act**" et "**othw**" utilisés dans les formules booléennes de ces SEBPS étant respectivement des abréviations pour "**action**" et "**otherwise**").

$$\left\{\begin{array}{llll}
Z_{1,s_i} & \stackrel{\nu}{=} & Z_{2,s_i} \textbf{ and } Z_{3,s_i} & \text{où } 1 \le i \le 4 \\
Z_{3,s_1} & \stackrel{\nu}{=} & \textbf{case act } \texttt{SEND}\ m & \textbf{in } \texttt{SEND ?}m_1\texttt{:Msg} \textbf{ -> } Z_{2,s_2}(m_1) \mid \textbf{othw -> true endcase} \\
Z_{3,s_2} & \stackrel{\nu}{=} & \textbf{case act } \tau & \textbf{in } \texttt{SEND ?}m_1\texttt{:Msg} \textbf{ -> } Z_{2,s_3}(m_1) \mid \textbf{othw -> true endcase} \\
Z_{3,s_3} & \stackrel{\nu}{=} & \textbf{case act } \texttt{TOUT} & \textbf{in } \texttt{SEND ?}m_1\texttt{:Msg} \textbf{ -> } Z_{2,s_1}(m_1) \mid \textbf{othw -> true endcase and} \\
 & & \textbf{case act } \texttt{RECV}\ m & \textbf{in } \texttt{SEND ?}m_1\texttt{:Msg} \textbf{ -> } Z_{2,s_4}(m_1) \mid \textbf{othw -> true endcase} \\
Z_{3,s_4} & \stackrel{\nu}{=} & \textbf{case act } \texttt{ACK} & \textbf{in } \texttt{SEND ?}m_1\texttt{:Msg} \textbf{ -> } Z_{2,s_1}(m_1) \mid \textbf{othw -> true endcase} \\
Z_{4,s_1} & \stackrel{\nu}{=} & \textbf{case act } \texttt{SEND}\ m & \textbf{in true -> } Z_{1,s_2} \mid \textbf{othw -> true endcase} \\
Z_{4,s_2} & \stackrel{\nu}{=} & \textbf{case act } \tau & \textbf{in true -> } Z_{1,s_3} \mid \textbf{othw -> true endcase} \\
Z_{4,s_3} & \stackrel{\nu}{=} & \textbf{case act } \texttt{TOUT} & \textbf{in true -> } Z_{1,s_1} \mid \textbf{othw -> true endcase and} \\
 & & \textbf{case act } \texttt{RECV}\ m & \textbf{in true -> } Z_{1,s_4} \mid \textbf{othw -> true endcase} \\
Z_{4,s_4} & \stackrel{\nu}{=} & \textbf{case act } \texttt{ACK} & \textbf{in true -> } Z_{1,s_1} \mid \textbf{othw -> true endcase}
\end{array}\right.$$

$$\left\{\begin{array}{llll}
Z_{2,s_i}(m_2\texttt{:Msg}) & \stackrel{\mu}{=} & Z_{5,s_i}(m_2) \textbf{ and } Z_{6,s_i}(m_2) & \text{où } 1 \le i \le 4 \\
Z_{5,s_1}(m_5\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \texttt{SEND}\ m & \textbf{in } \texttt{RECV ?}m_1\texttt{:Msg} \textbf{ -> true} \mid \textbf{othw -> false endcase} \\
Z_{5,s_2}(m_5\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \tau & \textbf{in } \texttt{RECV ?}m_1\texttt{:Msg} \textbf{ -> true} \mid \textbf{othw -> false endcase} \\
Z_{5,s_3}(m_5\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \texttt{TOUT} & \textbf{in } \texttt{RECV ?}m_1\texttt{:Msg} \textbf{ -> true} \mid \textbf{othw -> false endcase or} \\
 & & \textbf{case act } \texttt{RECV}\ m & \textbf{in } \texttt{RECV ?}m_1\texttt{:Msg} \textbf{ -> true} \mid \textbf{othw -> false endcase} \\
Z_{5,s_4}(m_5\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \texttt{ACK} & \textbf{in } \texttt{RECV ?}m_1\texttt{:Msg} \textbf{ -> true} \mid \textbf{othw -> false endcase} \\
Z_{6,s_1}(m_6\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \texttt{SEND}\ m & \textbf{in not } (\texttt{SEND } \textbf{any}) \textbf{ -> } Z_{2,s_2}(m_6) \mid \textbf{othw -> false endcase} \\
Z_{6,s_2}(m_6\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \tau & \textbf{in not } (\texttt{SEND } \textbf{any}) \textbf{ -> } Z_{2,s_3}(m_6) \mid \textbf{othw -> false endcase} \\
Z_{6,s_3}(m_6\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \texttt{TOUT} & \textbf{in not } (\texttt{SEND } \textbf{any}) \textbf{ -> } Z_{2,s_1}(m_6) \mid \textbf{othw -> false endcase} \\
 & & \textbf{case act } \texttt{RECV}\ m & \textbf{in not } (\texttt{SEND } \textbf{any}) \textbf{ -> } Z_{2,s_4}(m_6) \mid \textbf{othw -> false endcase} \\
Z_{6,s_4}(m_6\texttt{:Msg}) & \stackrel{\mu}{=} & \textbf{case act } \texttt{ACK} & \textbf{in not } (\texttt{SEND } \textbf{any}) \textbf{ -> } Z_{2,s_1}(m_6) \mid \textbf{othw -> false endcase}
\end{array}\right.$$

Pour simplifier la présentation, nous avons pris la liberté de représenter explicitement[14] (en donnant la liste de leurs champs) les actions du STE présentes dans les expressions "**case action**" (bien que le langage XTL n'offre pas de primitives de représentation explicite des valeurs de type `label`). ■

[14] Il s'agit de la représentation interne des valeurs de type `label` dans la zone des étiquettes (*label area*) du fichier BCG contenant le modèle STE. Les informations contenues dans cette zone sont directement accessibles durant la construction du SEBP.

Remarque 4-3
La taille du SEBP SB produit par traduction d'un σ-bloc SM dans le contexte d'un modèle STE $\mathcal{M}$ est linéaire par rapport à la taille de SM et de $\mathcal{M}$: le nombre de variables booléennes de SB est égal au nombre de variables propositionnelles de SM multiplié par le nombre d'états de $\mathcal{M}$, et le nombre d'opérateurs en partie droite des équations de SB ne peut pas dépasser celui de SM, multiplié par le nombre de transitions de $\mathcal{M}$. ■

Il est important d'observer qu'en faisant abstraction des expressions contenues dans φ, la traduction $\text{TRB}(\varphi, s)$ est définie en fonction des transitions successeurs de l'état s (cas des modalités "$\langle\ \rangle$" et "$[\ ]$") et en fonction de s lui-même (cas des autres opérateurs). Cette traduction pourrait donc être effectuée à la volée, en construisant simultanément le modèle STE et le SEBP. Cependant, ceci n'est pas possible lorsque les expressions contenues dans φ contiennent, par exemple, des méta-opérateurs de type `stateset` qui font référence à des états prédécesseurs de l'état courant, comme c'est le cas de l'opérateur `pred` sur états (voir la section 2.3.1). Ces observations nous conduisent à la définition d'un sous-ensemble des programmes XTL dont l'évaluation ne nécessite pas la construction préalable du STE (bien entendu, ces programmes peuvent être évalués aussi sur des STEs déjà construits). La définition suivante précise des conditions suffisantes identifiant cette classe de programmes XTL.

Définition 4-2 (Programmes XTL évaluables à la volée)
Un programme XTL P est dit *évaluable à la volée* si toute expression E contenue dans P satisfait les conditions suivantes :

1. $\mathit{type}(E) = \texttt{state} \Rightarrow (E \equiv \textbf{current} \ \vee\ E \equiv \texttt{init} \ \vee\ E \equiv \texttt{target}(E_1) \ \vee\ E \equiv x_1)$;
2. $\mathit{type}(E) = \texttt{stateset} \Rightarrow (E \equiv \texttt{succ}(E_2) \ \vee\ E \equiv x_2)$;
3. $\mathit{type}(E) = \texttt{label} \Rightarrow (E \equiv \textbf{current} \ \vee\ E \equiv \texttt{label}(E_3) \ \vee\ E \equiv x_3)$;
4. $\mathit{type}(E) = \texttt{transset} \Rightarrow (E \equiv \texttt{succ}(E_4) \ \vee\ E \equiv \texttt{out}(E_4') \ \vee\ E \equiv x_4)$

où les variables x_1, x_2, x_3 et x_4 sont respectivement de type `state`, `stateset`, `label` et `transset`, et les expressions E_1, E_2, E_3, E_4 et E_4' sont respectivement de type `trans`, `state`, `trans`, `trans` et `state`. ■

Intuitivement, la définition 4-2 garantit que tous les méta-opérateurs font référence soit à un état (ou une étiquette) courant(e), soit à des états ou transitions successeurs d'un état donné. En pratique, ces conditions ne sont pas excessivement restrictives : toutes les formules temporelles XTL présentées aux sections 2.15, 5.3 et 5.4, ainsi qu'à l'annexe C, sont évaluables à la volée.

Quelques notions auxiliaires sont nécessaires afin d'exprimer la relation entre la sémantique d'un σ-bloc et du SEBP produit par traduction. Nous introduisons la fonction $\text{TRB} : \mathbf{PEnv} \to \mathbf{BEnv}$, qui traduit les environnements propositionnels vers des environnements booléens :

$$\mathit{supp}(\text{TRB}(\rho)) = \{Z_{i,s} \mid Y_i \in \mathit{supp}(\rho) \ \wedge\ s \in S\}, \quad (\text{TRB}(\rho))(Z_{i,s}) = \big(s \in \rho(Y_i)\big)$$

pour chaque $\rho \in \mathbf{PEnv}$ et $Z_{i,s} \in \mathit{supp}(\text{TRB}(\rho))$. Nous introduisons aussi la fonction $h : (\mathbf{Param} \to 2^S)^n \to (\mathbf{Param} \to \mathbf{Bool})^{n \cdot |S|}$, définie comme suit :

$$h\big((F_1, \ldots, F_n)\big) \stackrel{\mathrm{d}}{=} \big(\lambda v_i^1{:}T_i^1, \ldots, v_i^{n_i}{:}T_i^{n_i}.(s \in F_i(v_i^1, \ldots, v_i^{n_i}))\big)_{1 \le i \le n, s \in S} \tag{4.9}$$

pour $F_i : T_i^1 \times \cdots \times T_i^{n_i} \to 2^S$ $(1 \le i \le n)$. Il est facile de vérifier que h est un isomorphisme entre les treillis complets $\big\langle (\mathbf{Param} \to 2^S)^n, \sqcup, \sqcap, \sqsubseteq \big\rangle$ et $\big\langle (\mathbf{Param} \to \mathbf{Bool})^{n \cdot |S|}, \sqcup, \sqcap, \sqsubseteq \big\rangle$ (les opérations $\sqcup$, $\sqcap$ et la relation $\sqsubseteq$ étant définies par extension des opérations correspondantes sur 2^S et $\mathbf{Bool}$).

Nous utilisons aussi le lemme de traduction suivant.

Lemme 4-4
Soit $\varphi \in \mathit{SForm}$, $\rho \in \mathbf{PEnv}$ et $\varepsilon \in \mathbf{DEnv}$ tels que $\mathit{fpv}(\varphi) \subseteq \mathit{supp}(\rho)$ et $\mathit{fdv}(\varphi) \subseteq \mathit{supp}(\varepsilon)$. Alors :

$$[\![\varphi]\!]\,\rho\varepsilon = \big\{s \in S \mid [\![\mathrm{TRB}(\varphi, s)]\!]\,\mathrm{TRB}(\rho)\varepsilon = \mathbf{tt}\big\}.$$

■

Preuve Elémentaire, par induction structurelle sur φ. □

La proposition suivante précise la relation entre les sémantiques des σ-blocs et des SEBPs correspondants obtenus après traduction.

Proposition 4-3 (Traduction des σ-blocs vers des SEBPs)
Soit un σ-bloc $SM = \big\{Y_i(x_i^1{:}T_i^1, \ldots, x_i^{n_i}{:}T_i^{n_i}) \stackrel{\sigma}{=} \varphi_i\big\}_{1 \le i \le n}$ et un environnement $\rho \in \mathbf{PEnv}$ tel que $\mathit{fpv}(SM) \subseteq \mathit{supp}(\rho)$. Alors, pour tout $1 \le i \le n$ et pour toutes les valeurs $v_i^1{:}T_i^1, \ldots, v_i^{n_i}{:}T_i^{n_i}$:

$$\big([\![SM]\!]\,\rho\big)_i(v_i^1, \ldots, v_i^{n_i}) = \big\{s \in S \mid \big([\![\mathrm{TRMB}(SM)]\!]\,\mathrm{TRB}(\rho)\big)_{i,s}(v_i^1, \ldots, v_i^{n_i}) = \mathbf{tt}\big\}.$$

■

Preuve Utilisant les définitions des fonctionnelles $\overline{\Phi}_\rho$ et $\overline{\Psi}_{\mathrm{TRB}(\rho)}$, associées respectivement à SM et à $\mathrm{TRMB}(SM)$, et la définition (4.9) de l'isomorphisme h, l'énoncé de la proposition se ramène à l'égalité ci-dessous :

$$h(\sigma\overline{\Phi}_\rho) = \sigma\overline{\Psi}_{\mathrm{TRB}(\rho)} \tag{4.10}$$

où $\sigma \in \{\mu, \nu\}$ est le signe de SM. En appliquant le lemme 4-4 à chaque formule φ_i, pour $1 \le i \le n$, et en étendant le résultat aux fonctionnelles vectorielles $\overline{\Phi}_\rho$ et $\overline{\Psi}_{\mathrm{TRB}(\rho)}$, nous obtenons :

$$\big(h\big(\overline{\Phi}_\rho(F_1, \ldots, F_n)\big)\big) = \overline{\Psi}_{\mathrm{TRB}(\rho)}\big(h(F_1, \ldots, F_n)\big)$$

pour tous $F_i : T_i^1 \times \cdots \times T_i^{n_i} \to 2^S$ $(1 \le i \le n)$. Puisque h est un isomorphisme de treillis complets et il préserve les fonctionnelles monotones $\overline{\Phi}_\rho$ et $\overline{\Psi}_{\mathrm{TRB}(\rho)}$, il préserve aussi leurs plus petits et plus grands points fixes, ce qui implique l'égalité (4.10). □

La proposition 4-3 permet donc de réduire la résolution partielle d'un σ-bloc à la résolution partielle d'un SEBP. La section suivante présente un algorithme permettant d'effectuer cette résolution.

4.2.4 Résolution des systèmes d'équations booléennes paramétrées

Considérons un SEBP $SB = \{Z_k(x_k^1{:}T_k^1, \ldots, x_k^{n_k}{:}T_k^{n_k}) \stackrel{\sigma}{=} \psi_k\}_{1 \le k \le p}$ produit à partir d'un σ-bloc et d'un modèle STE et soit un environnement booléen $\delta \in \mathbf{BEnv}$ tel que $\mathit{fbv}(SB) \subseteq \mathit{supp}(\delta)$. Pour simplifier la présentation, nous supposons que SB et δ sont donnés explicitement, tout en sachant qu'ils peuvent être aussi construits à la volée, conjointement avec la génération du STE (voir les sections 4.2.2 et 4.2.3). Soit $J \subseteq [1, p]$ et soit $v_j^1 \in T_j^1, \ldots, v_j^{n_j} \in T_j^{n_j}$ pour chaque $j \in J$. L'objectif est de résoudre SB partiellement, c'est-à-dire de calculer $([\![SB]\!]\,\delta)_j(v_j^1, \ldots, v_j^{n_j})$ pour chaque $j \in J$. Nous ne traitons ici que le cas $\sigma = \mu$, le cas $\sigma = \nu$ étant complètement dual.

Quelques définitions auxiliaires sont nécessaires. Suivant la terminologie utilisée à la section 4.1, nous appelons $Z_j(v_j^1, \ldots, v_j^{n_j})$ une *instance* de la variable Z_j avec les arguments $v_j^1, \ldots, v_j^{n_j}$. Les instances sont représentées dans l'algorithme sous forme de couples $(Z_j, (v_j^1, \ldots, v_j^{n_j})) \in \mathit{BVar} \times \mathbf{Param}$; par abus de langage, nous utiliserons le même terme "instance" pour désigner ces couples. Une instance $Z_j(v_j^1, \ldots, v_j^{n_j})$ est *expansée* lorsque la formule ψ_j est évaluée avec les valeurs $v_j^1, \ldots, v_j^{n_j}$ substituées aux variables $x_j^1, \ldots, x_j^{n_j}$. Une instance $Z_j(v_j^1, \ldots, v_j^{n_j})$ *dépend* d'une instance $Z_l(v_l^1, \ldots, v_l^{n_l})$ ssi l'évaluation de ψ_j avec les arguments $v_j^1, \ldots, v_j^{n_j}$ doit utiliser la valeur de $Z_l(v_l^1, \ldots, v_l^{n_l})$. Cette relation induit un graphe de dépendance entre les instances, ayant un sommet associé à chaque instance et un arc de $Z_j(v_j^1, \ldots, v_j^{n_j})$ à $Z_l(v_l^1, \ldots, v_l^{n_l})$ ssi $Z_j(v_j^1, \ldots, v_j^{n_j})$ dépend de $Z_l(v_l^1, \ldots, v_l^{n_l})$. Les termes "instance successeur" et "instance prédécesseur" ont l'interprétation usuelle dans le contexte du graphe de dépendance.

Afin de simplifier les notations, nous introduisons aussi un opérateur auxiliaire sur fonctions partielles ensemblistes. Soit A et B deux ensembles et soit $e_1, e_2 : A \to 2^B$ deux fonctions partielles. L'*union* de e_1 et e_2, notée $e_1 \cup e_2$, est une fonction partielle définie comme suit :

$$(e_1 \cup e_2)(x) \stackrel{\mathrm{d}}{=} \begin{cases} e_1(x) & \text{si } x \in \mathit{supp}(e_1) \setminus \mathit{supp}(e_2) \\ e_1(x) \cup e_2(x) & \text{si } x \in \mathit{supp}(e_1) \cap \mathit{supp}(e_2) \\ e_2(x) & \text{si } x \in \mathit{supp}(e_2) \setminus \mathit{supp}(e_1). \end{cases}$$

La résolution partielle d'un Sebp de signe μ est effectuée au moyen de l'algorithme 4-1 que nous proposons ci-dessous. En partant des instances à calculer, l'algorithme explore "en avant" le graphe de dépendance entre les instances des variables du Sebp, en effectuant l'expansion des instances déjà visitées (mais pas encore expansées) pendant le calcul. La valeur courante de chaque nouvelle instance visitée est initialisée à faux (car il s'agit de calculer un plus petit point fixe). Lorsqu'une instance expansée est évaluée à vrai, elle devient "stable", ceci signifiant qu'elle a atteint sa valeur finale. Chaque fois qu'une telle instance est évaluée à vrai, les autres instances qui en dépendent deviennent potentiellement "instables" (c'est-à-dire qu'elles peuvent passer de faux à vrai si elles sont réévaluées dans le nouveau contexte) et donc elles doivent être réévaluées. Ce processus s'arrête lorsque toutes les instances visitées ont été expansées et il n'existe plus d'instances "instables", ceci signifiant que les valeurs de toutes les instances parcourues correspondent au plus petit point fixe du Sebp. Bien que l'algorithme 4-1 présente des aspects communs avec plusieurs autres algorithmes dédiés à la résolution locale [VWL94, VL94, And94] ou globale [AC88, CS93, And94] des systèmes d'équations booléennes "pures" (sans valeurs typées), il permet de traiter un formalisme d'entrée plus général (les Sebps). En outre, suivant que le Sebp à résoudre est construit complètement ou généré à la volée, cet algorithme permet d'effectuer une résolution globale ou locale.

Algorithme 4-1 (Résolution des SEBPs)
La résolution partielle d'un Sebp de signe μ est effectuée au moyen d'une fonction principale EvalSB qui utilise une fonction auxiliaire EvalB. La fonction EvalSB prend en entrée :

- un Sebp $SB = \{Z_k(x_k^1{:}T_k^1, \ldots, x_k^{n_k}{:}T_k^{n_k}) \stackrel{\mu}{=} \psi_k\}_{1 \le k \le p}$;
- un ensemble $\mathit{wanted} = \{(Z_j, (v_j^1, \ldots, v_j^{n_j})) \mid \forall j \in J. v_j^1 \in T_j^1, \ldots, v_j^{n_j} \in T_j^{n_j}\}$, où $J \subseteq [1, p]$;
- un environnement booléen $\delta \in \mathbf{BEnv}$ tel que $\mathit{fbv}(SB) \subseteq \mathit{supp}(\delta)$

et produit en sortie un ensemble $\{(Z_j, (v_j^1, \ldots, v_j^{n_j}), b_j) \mid (Z_j, (v_j^1, \ldots, v_j^{n_j})) \in \mathit{wanted}\}$ tel que $b_j = (\llbracket SB \rrbracket\, \delta)_j(v_j^1, \ldots, v_j^{n_j})$ pour chaque $j \in J$. EvalSB utilise les variables locales suivantes :

- *visited* $\subseteq \mathit{BVar} \times \mathbf{Param}$ est un ensemble d'instances $(Z_j, (v_j^1, \ldots, v_j^{n_j}))$ qui ont été déjà rencontrées pendant le calcul, mais qui n'ont pas encore été expansées.
- *expanded* : $(\mathit{BVar} \times \mathbf{Param}) \to (\mathbf{Bool} \times \mathbf{Nat})$ est une fonction partielle qui associe à chaque instance expansée $(Z_j, (v_j^1, \ldots, v_j^{n_j}))$ un couple contenant deux champs : un drapeau $b \in \mathbf{Bool}$, représentant la valeur courante qui a été calculée pour l'instance $Z_j(v_j^1, \ldots, v_j^{n_j})$, et un compteur $c \in \mathbf{Nat}$, mémorisant le nombre courant d'instances successeur de $Z_j(v_j^1, \ldots, v_j^{n_j})$ qui doivent être vraies pour que ψ_j, réévaluée avec les arguments $v_j^1, \ldots, v_j^{n_j}$, devienne vraie.

- *depend* : $(BVar \times \mathbf{Param}) \to 2^{BVar \times \mathbf{Param}}$ est une fonction partielle qui associe à une instance $(Z_l, (v_l^1, \ldots, v_l^{n_l}))$ l'ensemble de toutes les instances $(Z_j, (v_j^1, \ldots, v_j^{n_j}))$ qui en dépendent (*depend* modélise la relation "prédécesseur" entre instances).

  Nous avons modélisé *expanded* et *depend* comme des fonctions partielles, car ces objets sont construits au fur et à mesure de la résolution du SEBP.

- *to_be_propagated* $\subseteq BVar \times \mathbf{Param}$ est un ensemble d'instances $(Z_l, (v_l^1, \ldots, v_l^{n_l}))$ expansées qui ont été évaluées à vrai et dont l'effet n'a pas été propagé aux instances qui en dépendent.

La résolution est effectuée de la manière suivante. La variable *visited* est initialisée avec l'ensemble *wanted* des instances à calculer, les autres variables *to_be_propagated*, *expanded* et *depend* étant initialement vides. Ensuite, deux actions sont effectuées de manière répétitive :

- Si *visited* est non vide, alors une instance $(Z_j, (v_j^1, \ldots, v_j^{n_j}))$ à calculer est extraite de *visited* et expansée, en évaluant (partiellement) ψ_j avec les arguments $v_j^1, \ldots, v_j^{n_j}$ à l'aide d'EVALB. Les nouvelles instances éventuellement rencontrées pendant l'évaluation de ψ_j sont rajoutées à *visited* et les nouvelles dépendances créées entre ces instances et $(Z_j, (v_j^1, \ldots, v_j^{n_j}))$ sont mémorisées dans *depend*. Si ψ_j (donc $Z_j(v_j^1, \ldots, v_j^{n_j})$) est évaluée à vrai, les instances qui dépendent de $(Z_j, (v_j^1, \ldots, v_j^{n_j}))$ deviennent potentiellement "instables" et, par conséquent, $(Z_j, (v_j^1, \ldots, v_j^{n_j}))$ est rajoutée à *to_be_propagated*.

- Si *to_be_propagated* est non vide, alors une instance $Z_l(v_l^1, \ldots, v_l^{n_l})$ est extraite de *to_be_propagated* et toutes les instances $Z_j(v_j^1, \ldots, v_j^{n_j})$ qui en dépendent sont réévaluées. La structure simple des formules ψ_j (voir la remarque 4-2) permet d'optimiser leur réévaluation au moyen des compteurs c attachés aux instances expansées (des techniques d'optimisation similaires sont utilisées dans les algorithmes efficaces de résolution des systèmes d'équations booléennes présentés en [CS93, And94]). Chaque instance $Z_j(v_j^1, \ldots, v_j^{n_j})$ qui est réévaluée à vrai est mémorisée dans *to_be_propagated*.

Ce processus s'arrête lorsque les variables *visited* et *to_be_propagated* sont vides, ceci signifiant que toutes les instances à calculer ont été expansées et que toutes les instances expansées sont "stables". La fonction EVALSB est présentée ci-dessous.

fonction EVALSB $(\{Z_k(x_k^1{:}T_k^1, \ldots, x_k^{n_k}{:}T_k^{n_k}) \stackrel{\mu}{=} \psi_k\}_{1 \le k \le p} : BSys,\ wanted : 2^{BVar \times \mathbf{Param}},\ \delta : \mathbf{BEnv})$
$: 2^{BVar \times \mathbf{Param} \times \mathbf{Bool}}$

est
  var *expanded* : $(BVar \times \mathbf{Param}) \to (\mathbf{Bool} \times \mathbf{Nat})$,
    visited, *to_be_propagated*, *needed* : $2^{BVar \times \mathbf{Param}}$,
    depend : $(BVar \times \mathbf{Param}) \to 2^{BVar \times \mathbf{Param}}$, b : **Bool**, c : **Nat**;
  expanded := [ ];
  visited := *wanted*;
  to_be_propagated := $\emptyset$;
  depend := [ ];
  repeter
    si *visited* $\neq \emptyset$ **alors**
      soit $(Z_j, (v_j^1, \ldots, v_j^{n_j})) \in visited$;
      visited := *visited* $\setminus \{(Z_j, (v_j^1, \ldots, v_j^{n_j}))\}$;
      $(needed, b, c)$:= EVALB $(\psi_j,\ \delta,\ [v_j^1/x_j^1, \ldots, v_j^{n_j}/x_j^{n_j}],\ visited,\ expanded)$;
      expanded := *expanded* $\cup [(b,c)/(Z_j, (v_j^1, \ldots, v_j^{n_j}))]$;
      si b **alors**
        to_be_propagated := *to_be_propagated* $\cup \{(Z_j, (v_j^1, \ldots, v_j^{n_j}))\}$;
      fin_si;

pour_tous $(Z_l, (v_l^1, \ldots, v_l^{n_l})) \in \mathit{needed}$ **faire**
$\mathit{depend} := \mathit{depend} \cup [\{(Z_j, (v_j^1, \ldots, v_j^{n_j}))\} / (Z_l, (v_l^1, \ldots, v_l^{n_l}))]$;
si $(Z_l, (v_l^1, \ldots, v_l^{n_l})) \notin \mathit{visited}$ **alors**
$\mathit{visited} := \mathit{visited} \cup \{(Z_l, (v_l^1, \ldots, v_l^{n_l}))\}$
fin_si
fin
sinon_si $\mathit{to\_be\_propagated} \neq \emptyset$ **alors**
soit $(Z_l, (v_l^1, \ldots, v_l^{n_l})) \in \mathit{to\_be\_propagated}$;
$\mathit{to\_be\_propagated} := \mathit{to\_be\_propagated} \setminus \{(Z_l, (v_l^1, \ldots, v_l^{n_l}))\}$;
pour_tous $(Z_j, (v_j^1, \ldots, v_j^{n_j})) \in \mathit{depend}((Z_l, (v_l^1, \ldots, v_l^{n_l})))$ **faire**
$(b, c) := \mathit{expanded}((Z_j, (v_j^1, \ldots, v_j^{n_j})))$;
si $\neg b$ **alors**
si $c = 1$ **alors**
$\mathit{expanded} := \mathit{expanded} \oslash [(\mathbf{tt}, 0) / (Z_j, (v_j^1, \ldots, v_j^{n_j}))]$;
$\mathit{to\_be\_propagated} := \mathit{to\_be\_propagated} \cup \{(Z_j, (v_j^1, \ldots, v_j^{n_j}))\}$;
sinon
$\mathit{expanded} := \mathit{expanded} \oslash [(\mathbf{ff}, c-1) / (Z_j, (v_j^1, \ldots, v_j^{n_j}))]$
fin_si
fin_si
fin
fin_si
jusqu_a $\mathit{visited} = \emptyset \wedge \mathit{to\_be\_propagated} = \emptyset$;
retourner $\{(Z_j, (v_j^1, \ldots, v_j^{n_j}), b) \mid (Z_j, (v_j^1, \ldots, v_j^{n_j})) \in \mathit{wanted} \wedge$
$b = (\mathit{expanded}((Z_j, (v_j^1, \ldots, v_j^{n_j}))))_1\}$
fin

La fonction auxiliaire EVALB effectue l'évaluation d'une formule ψ, située en partie droite d'une équation de SB, dans un contexte fourni par la fonction EVALSB. Elle prend en entrée :

- une formule $\psi \in \mathit{BForm}$;
- deux environnements $\delta \in \mathbf{BEnv}$ et $\varepsilon \in \mathbf{DEnv}$ tels que $\mathit{fbv}(\psi) \subseteq \mathit{supp}(\delta)$ et $\mathit{fdv}(\psi) \subseteq \mathit{supp}(\varepsilon)$;
- les valeurs courantes des variables *visited* et *expanded* utilisées dans EVALSB

et produit en sortie un tuple contenant trois champs :

- l'ensemble d'instances $(Z_l, (v_l^1, \ldots, v_l^{n_l}))$ utilisées pendant l'évaluation de ψ ;
- la valeur booléenne b qui a pu être calculée pour la formule ψ ;
- le nombre c des instances utilisées qui doivent devenir vraies afin que ψ soit réévaluée à vrai.

L'évaluation est effectuée récursivement sur la structure de ψ, conformément à la sémantique des formules booléennes définie à la section 4.2.3. La fonction EVALB est présentée ci-dessous.

fonction EVALB $(\psi : \mathit{BForm}, \delta : \mathbf{BEnv}, \varepsilon : \mathbf{DEnv}, \mathit{visited} : 2^{\mathit{BVar} \times \mathbf{Param}},$
$\mathit{expanded} : (\mathit{BVar} \times \mathbf{Param}) \to (\mathbf{Bool} \times \mathbf{Nat})) : 2^{\mathbf{BVar} \times \mathbf{Param}} \times \mathbf{Bool} \times \mathbf{Nat}$
est
var $\mathit{needed}_1, \mathit{needed}_2 : 2^{\mathit{BVar} \times \mathbf{Param}}, b_1, b_2 : \mathbf{Bool}, c_1, c_2 : \mathbf{Nat}$;
cas ψ **dans**
true $\to$ **retourner** $(\emptyset, \mathbf{tt}, 0)$
false $\to$ **retourner** $(\emptyset, \mathbf{ff}, 0)$

$Z(E_1, \ldots, E_n) \rightarrow$
var v_1:*type*$(E_1) := [\![E_1]\!]\,\varepsilon, \ldots, v_n$:*type*$(E_n) := [\![E_n]\!]\,\varepsilon$;
retourner $(\{(Z, (v_1, \ldots, v_n))\}$,
si $Z \in$ *supp*(δ) **alors** $(\delta(Z))(v_1, \ldots, v_n)$
sinon_si $(Z, (v_1, \ldots, v_n)) \in$ *visited* **alors ff**
sinon_si $(Z, (v_1, \ldots, v_n)) \in$ *supp*(*expanded*) **alors** $(expanded(Z, (v_1, \ldots, v_n)))_1$
sinon ff fin_si,
1)
ψ_1 **or** $\psi_2 \rightarrow$
$(needed_1, b_1, c_1) :=$ EVALB $(\psi_1, \delta, \varepsilon$, *visited*, *expanded*);
$(needed_2, b_2, c_2) :=$ EVALB $(\psi_2, \delta, \varepsilon$, *visited*, *expanded*);
retourner $(needed_1 \cup needed_2,\ b_1 \lor b_2$, **si** $b_1 \lor b_2$ **alors** 0 **sinon** 1 **fin_si**)
ψ_1 **and** $\psi_2 \rightarrow$
$(needed_1, b_1, c_1) :=$ EVALB $(\psi_1, \delta, \varepsilon$, *visited*, *expanded*);
$(needed_2, b_2, c_2) :=$ EVALB $(\psi_2, \delta, \varepsilon$, *visited*, *expanded*);
retourner $(needed_1 \cup needed_2,\ b_1 \land b_2,\ \sum_{i \in [1,2]}$ (**si** $\neg b_i$ **alors** c_i **sinon** 0 **fin_si**))
exists x_0:T_0 [**among** E_0], $\ldots, x_n$:T_n [**among** E_n] **in** $\psi_1 \rightarrow$
$needed_2 := \emptyset$; $b_2 :=$ **ff**;
pour_tous v_0:$T_0[\in [\![E_0]\!]\,\varepsilon], \ldots, v_n$:$T_n[\in [\![E_n]\!]\,\varepsilon]$ **faire**
$(needed_1,\ b_1,\ c_1) :=$ EVALB $(\psi_1, \delta, \varepsilon \oslash [v_0/x_0, \ldots, v_n/x_n]$, *visited*, *expanded*);
$needed_2 := needed_2 \cup needed_1$; $b_2 := b_2 \lor b_1$
fin;
retourner $(needed_2,\ b_2$, **si** b_2 **alors** 0 **sinon** 1 **fin_si**)
forall x_0:T_0 [**among** E_0], $\ldots, x_n$:T_n [**among** E_n] **in** $\psi_1 \rightarrow$
$needed_2 := \emptyset$; $b_2 :=$ **tt**; $c_2 := 0$;
pour_tous v_0:$T_0[\in [\![E_0]\!]\,\varepsilon], \ldots, v_n$:$T_n[\in [\![E_n]\!]\,\varepsilon]$ **faire**
$(needed_1,\ b_1,\ c_1) :=$ EVALB $(\psi_1, \delta, \varepsilon \oslash [v_0/x_0, \ldots, v_n/x_n]$, *visited*, *expanded*);
$needed_2 := needed_2 \cup needed_1$; $b_2 := b_2 \land b_1$;
$c_2 := c_2 +$ (**si** $\neg b_1$ **alors** c_1 **sinon** 0 **fin_si**)
fin;
retourner $(needed_2,\ b_2,\ c_2)$
case E_0 **in**
$P_1^0 \mid \ldots \mid P_1^{n_1}$ [**where** E_1] -> ψ_1
...
$\mid P_m^0 \mid \ldots \mid P_m^{n_m}$ [**where** E_m] -> ψ_m
[| **otherwise** -> ψ_{m+1}]
endcase $\rightarrow$
var v_0:*type*$(E_0) := [\![E_0]\!]\,\varepsilon$;
si $\exists i \in [1, m].\exists j \in [0, n_i].(([\![P_i^j]\!]v_0)_1 = \mathbf{tt}\,[\ \land\ [\![E_i]\!]\,(\varepsilon \oslash ([\![P_i^j]\!]v_0)_2) = \mathbf{tt}\ ]) \land$
$\forall l \in [0, i-1].\forall k \in [0, n_l].(([\![P_l^k]\!]v_0)_1 = \mathbf{ff}\,[\ \lor\ [\![E_l]\!]\,(\varepsilon \oslash ([\![P_l^k]\!]v_0)_2) = \mathbf{ff}\ ])$
alors retourner EVALB $(\psi_i, \delta, \varepsilon \oslash ([\![P_i^j]\!]v_0)_2$, *visited*, *expanded*)
[**sinon retourner** EVALB $(\psi_{m+1}, \delta, \varepsilon$, *visited*, *expanded*)]
fin_si
case action E_0 **in**
α_1 [**where** E_1] -> ψ_1
...
$\mid \alpha_m$ [**where** E_m] -> ψ_m
[| **otherwise** -> ψ_{m+1}]
endcase $\rightarrow$

var a_0:`label` := $[\![E_0]\!]\,\varepsilon$;
si $\exists i \in [1,m].(([\![\alpha_i]\!]\,\varepsilon a_0)_1 = \mathbf{tt}\,[\ \wedge\ [\![E_i]\!]\,(\varepsilon \oslash ([\![\alpha_i]\!]\,\varepsilon a_0)_2) = \mathbf{tt}\ ]) \wedge$
$\forall l \in [1, i-1].(([\![\alpha_l]\!]\,\varepsilon a_0)_1 = \mathbf{ff}[\ \vee\ [\![E_l]\!]\,(\varepsilon \oslash ([\![\alpha_l]\!]\,\varepsilon a_0)_2) = \mathbf{ff}\ ])$
alors retourner EVALB (ψ_i, δ, $\varepsilon \oslash ([\![\alpha_i]\!]\,\varepsilon a_0)_2$, *visited*, *expanded*)
[**sinon retourner** EVALB (ψ_{m+1}, δ, ε, *visited*, *expanded*)]
fin_si
fin_cas
fin

Nous n'avons pas défini explicitement l'évaluation des expressions E et des formules α contenues dans la formule ψ : il s'agit respectivement d'une implémentation de la sémantique des expressions XTL (définie à l'annexe B) et des formules sur actions (définie à la section 3.5). ■

Nous illustrons l'exécution de l'algorithme 4-1 au moyen d'un exemple.

Exemple 4-8
En reprenant les SEBPS SB_1 et SB_2 et le STE de l'exemple 4-7, nous voulons évaluer, sur l'état initial s_1 du STE, la formule φ de l'exemple 4-4 à partir de laquelle SB_1 et SB_2 sont générés. Ceci revient à évaluer l'appel[15] EVALSB($SB_1, \{(Z_{1,s_1})\}, \delta$), où l'environnement $\delta \in \mathbf{BEnv}$, ayant le support $\{Z_{2,s_i} \mid 1 \leq i \leq 4\}$, sera construit au moyen d'appels d'EVALSB avec SB_2. Le graphe de dépendance entre les instances générées pendant l'évaluation est illustré dans la figure 4.6.

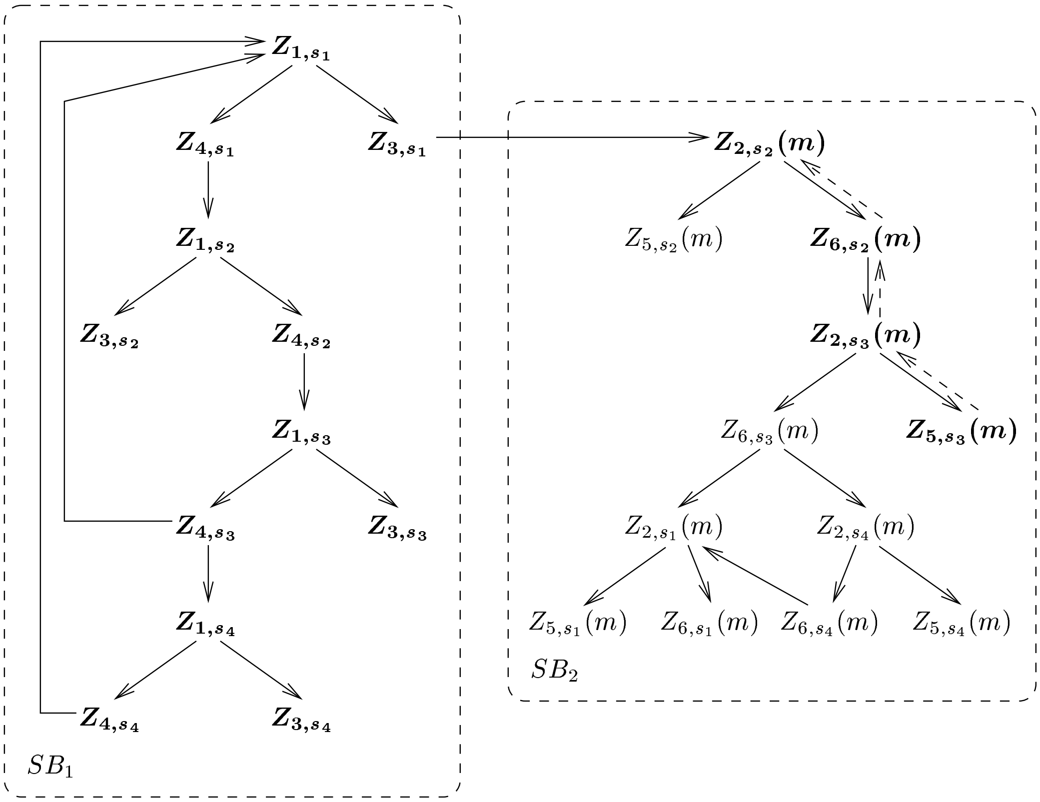

Figure 4.6: Evaluation locale de SB_1 dans le contexte de SB_2

[15]Il s'agit ici de la variante duale d'EVALSB pour les SEBPS de signe ν.

Les instances représentées en caractères gras sont évaluées à **tt** après terminaison de l'algorithme et les autres instances sont évaluées à **ff**. La formule φ est satisfaite par l'état initial du STE, puisque l'instance Z_{1,s_1}, correspondant à l'évaluation de φ sur s_1, est évaluée à **tt**. Sur la figure 4.6, les arcs représentés en pointillés indiquent la propagation de l'effet des instances contenues dans *to_be_propagated* pendant l'exécution de l'algorithme. ■

L'algorithme 4-1 a une complexité linéaire en taille du graphe de dépendance entre les instances des variables du SEBP générées pendant l'évaluation, puisque chaque arc de ce graphe est traversé au plus deux fois : premièrement, lorsque l'instance de départ de l'arc est expansée, et deuxièmement, lorsque l'instance d'arrivée de l'arc est (éventuellement) propagée. La terminaison de l'algorithme 4-1 étant indécidable lorsque les domaines des paramètres des équations sont infinis (voir la discussion en section 4.1), la taille du graphe de dépendance entre instances est (extrêmement) difficile à estimer dans le cas général. Néanmoins, nous pouvons examiner diverses classes particulières de formules XTL d'alternance 1 pour lesquelles l'algorithme termine :

- pour les formules φ n'ayant pas d'opérateurs de point fixe paramétrés (mais pouvant contenir des variables simples dans les modalités), la taille du graphe de dépendance entre instances ne peut pas dépasser la taille du SEBP obtenu par traduction, qui est linéaire par rapport à la taille de la formule φ et du STE (voir la remarque 4-3) :

$$O\big(|\varphi| \cdot (|S| + |T|)\big)$$

  En particulier, ce fragment de XTL inclut le μ-calcul standard d'alternance 1, auquel cas la complexité de l'algorithme 4-1 se ramène à celle des meilleurs algorithmes disponibles pour cette logique [CS93, And94, VL94].

- pour les formules φ dont tous les opérateurs de point fixe ont des *paramètres restreints* [RH96] (c'est-à-dire que les arguments de tous les appels de variables propositionnelles dans φ sont des variables simples préalablement initialisées avec des valeurs extraites des actions du STE), la taille du graphe de dépendance devient :

$$O\big(|\varphi| \cdot (|S| + |T|) \cdot |A|^{arity(\varphi)}\big)$$

  où *arity*(φ) représente l'arité maximale des opérateurs de point fixe contenus dans φ. Le facteur $|A|^{arity(\varphi)}$ provient du fait que les paramètres des instances générées peuvent couvrir tout le domaine des valeurs contenues dans les actions du STE. En pratique, ce facteur devrait rester raisonnablement petit, puisque le nombre d'actions $|A|$ est habituellement (beaucoup) plus petit que le nombre d'états $|S|$ du STE et l'arité maximale des opérateurs de point fixe ne dépasse généralement pas 2 ou 3.

Il est utile de remarquer que les estimations de complexité indiquées ci-dessus sont calculées dans le pire des cas (c'est-à-dire, lorsque toutes les instances des variables du SEBP sont expansées par l'algorithme). La complexité moyenne de l'algorithme 4-1 peut être diminuée en appliquant diverses techniques d'optimisation [VWL94] visant à réduire la portion du graphe de dépendances entre instances qu'il faut explorer afin de décider la valeur de vérité de la formule à vérifier.

En général, la terminaison de l'algorithme 4-1 peut être prouvée en examinant les domaines des valeurs possibles que puissent prendre les paramètres des opérateurs de point fixe de la formule φ à vérifier (qui se retrouvent comme paramètres des équations du SEBP obtenu par traduction). Dans beaucoup de cas (voir notamment les exemples de formules XTL donnés aux sections 5.3 et 5.4), les domaines de ces paramètres sont limités par l'ensemble des valeurs contenues dans le STE (nombre de processus, taille des messages, ...), ce qui assure la terminaison de l'algorithme et peut fournir des indications sur la complexité de l'évaluation.

4.3 Evaluation des formules d'alternance quelconque

Nous abordons dans cette section le problème de l'évaluation des formules XTL d'alternance $n \geq 1$ sur un modèle STE étendu $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$. Il existe dans la littérature consacrée au μ-calcul standard de nombreux algorithmes pour l'évaluation des formules d'alternance quelconque [EL86, Cle90, SW91, Win91, CKS92, And94, LBC+94]. Par souci de simplicité, nous avons choisi de généraliser un algorithme global proposé en [EL86], qui calcule itérativement l'ensemble d'états d'un modèle STE qui satisfont une formule.

Tout comme à la section 4.2, nous considérons ici uniquement des formules XTL φ en forme normale positive (FNP) et fermées d.p.d.v. des variables propositionnelles. Nous supposons aussi que les transformations préliminaires sur φ ont été effectuées (voir la section 3.8) et que l'attribut $sign(Y) \in \{\mu, \nu\}$, indiquant le signe de Y, a été calculé (voir l'annexe A.5) pour chaque variable propositionnelle $Y \in PVar$. Le symbole σ dénote "**mu**" ou "**nu**".

Quelques définitions auxiliaires sont nécessaires. Nous introduisons le domaine $\mathbf{PSub} \stackrel{\mathrm{d}}{=} PVar \to 2^{\mathbf{Param}}$ des *sous-domaines propositionnels*. Un sous-domaine $\theta \in \mathbf{PSub}$ est une fonction partielle associant à chaque variable $Y(x_1{:}T_1, \ldots, x_n{:}T_n)$ un ensemble $\theta(Y) = V_1 \times \cdots \times V_n \subseteq T_1 \times \cdots \times T_n$. Intuitivement, les sous-domaines θ permettent de représenter les instances des variables Y rencontrées pendant l'évaluation des formules. Pour combiner les sous-domaines propositionnels, nous utilisons l'opérateur d'union $\cup$ sur fonctions partielles ensemblistes défini à la section 4.2.4.

L'évaluation d'une formule XTL φ d'alternance quelconque est effectuée au moyen de l'algorithme 4-2 que nous proposons ci-dessous. Les formules booléennes "**true**", "**false**", "**or**" et "**and**", les formules modales "$\langle\ \rangle$" et "[ ]", les quantificateurs "**exists**" et "**forall**" et les constructions "**case**" et "**case action**" sont évalués par un calcul direct de leurs fonctions sémantiques, qui ont été définies à la section 3.7. Pour les formules de point fixe σY $(x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n)$ **.** φ', l'algorithme calcule itérativement l'ensemble d'états R du STE qui satisfont l'instance $Y(v_1, \ldots, v_n)$, où $v_1, \ldots, v_n$ sont les valeurs des arguments $E_1, \ldots, E_n$. L'évaluation de $Y(v_1, \ldots, v_n)$ peut entraîner le calcul d'autres instances $Y(v'_1, \ldots, v'_n)$, provenant des appels de Y contenus dans φ'. Ces instances, initialisées à l'ensemble vide ou à S suivant que σ dénote un plus petit ou un plus grand point fixe, sont mémorisées dans un sous-domaine propositionnel $\theta(Y)$ et sont réévaluées à chaque itération. Ce processus est arrêté lorsque toutes les instances rencontrées sont "stables" (c'est-à-dire que leur réévaluation produirait le même ensemble d'états du STE) et elles ne dépendent pas d'instances autres que celles contenues dans $\theta(Y)$, ceci signifiant que toutes les instances nécessaires pour l'évaluation de $Y(v_1, \ldots, v_n)$ ont été calculées.

Algorithme 4-2 (Evaluation des formules XTL d'alternance quelconque)
L'évaluation des formules XTL d'alternance quelconque est effectuée au moyen d'une fonction EVALS. Cette fonction prend en entrée :

- une formule $\varphi \in SForm$;
- un environnement $\rho \in \mathbf{PEnv}$ tel que $fpv(\varphi) \subseteq supp(\rho)$;
- un environnement $\varepsilon \in \mathbf{DEnv}$ tel que $fdv(\varphi) \subseteq supp(\varepsilon)$

et produit en sortie un tuple contenant deux champs :

- l'ensemble d'états satisfaisant φ dans le contexte de ρ et de ε ;
- le sous-domaine propositionnel contenant les instances des variables propositionnelles de φ qui ont été rencontrées pendant l'évaluation.

La fonction EVALS est présentée ci-dessous.

fonction EVALS (φ : *SForm*, ρ : **PEnv**, ε : **DEnv**) : (2^S, **PSub**) **est**
  var R, R', R'' : 2^S; θ, θ', θ'', θ''' : **PSub**; ρ' : **PEnv**; *stable* : **Bool**;
  cas φ **dans**
    true $\rightarrow$ **retourner** $(S, [\,])$
    false $\rightarrow$ **retourner** $(\emptyset, [\,])$
  | Y $(E_1, \ldots, E_n) \rightarrow$
      $R := \emptyset$;
      $\theta := [\,]$;
      pour_tous $s \in S$ **faire**
        var $v_1 : type(E_1) := [\![E_1]\!] (\varepsilon \oslash [s/c\_s]), \ldots, v_n : type(E_n) := [\![E_n]\!] (\varepsilon \oslash [s/c\_s])$;
        si $(v_1, \ldots, v_n) \in supp(\rho(Y))$ **alors**
          si $s \in (\rho(Y))(v_1, \ldots, v_n)$ **alors**
            $R := R \cup \{s\}$
          fin_si
        sinon
          si *sign* $(Y) =$ **nu alors**
            $R := R \cup \{s\}$
          fin_si;
          $\theta := \theta \cup [\ \{(v_1, \ldots, v_n)\}\ /\ Y\ ]$
        fin_si
      fin;
      retourner (R, θ)
  | φ_1 **or** $\varphi_2 \rightarrow$
      $(R', \theta') :=$ EVALS $(\varphi_1, \rho, \varepsilon)$;
      $(R'', \theta'') :=$ EVALS $(\varphi_2, \rho, \varepsilon)$;
      retourner $(R' \cup R'', \theta' \cup \theta'')$
  | φ_1 **and** $\varphi_2 \rightarrow$
      $(R', \theta') :=$ EVALS $(\varphi_1, \rho, \varepsilon)$;
      $(R'', \theta'') :=$ EVALS $(\varphi_2, \rho, \varepsilon)$;
      retourner $(R' \cap R'', \theta' \cup \theta'')$
  | $\langle \alpha \rangle\ \varphi_1 \rightarrow$
      $R := \emptyset$;
      $\theta := [\,]$;
      pour_tous $s \xrightarrow{a} s' \in T$ **faire**
        var $(b : \mathbf{Bool}, \mathcal{E}_0 : 2^{\mathbf{DEnv}}) := [\![\alpha]\!] \varepsilon a$;
        si b **alors**
          repeter
            soit $\varepsilon_0 \in \mathcal{E}_0$; $\mathcal{E}_0 := \mathcal{E}_0 \setminus \{\varepsilon_0\}$;
            $(R', \theta') :=$ EVALS $(\varphi_1, \rho, \varepsilon \oslash \varepsilon_0 \oslash [a/c\_a])$;
            si $s' \in R'$ **alors** $R := R \cup \{s\}$ **fin_si**;
            $\theta := \theta \cup \theta'$
          jusqu_a $\mathcal{E}_0 = \emptyset \ \vee\ s' \in R'$;
        fin_si
      fin;
      retourner (R, θ)
  | $[\alpha]\ \varphi_1 \rightarrow$
      $R := S$;
      $\theta := [\,]$;

```
      pour_tous s --a--> s' ∈ T faire
         var (b : Bool, ℰ₀ : 2^DEnv) := ⟦α⟧ ε a;
         si b alors
            repeter
               soit ε₀ ∈ ℰ₀; ℰ₀ := ℰ₀ \ {ε₀};
               (R', θ') := EvalS (φ₁, ρ, ε ⊘ ε₀ ⊘ [a/c_a]);
               si s' ∉ R' alors R := R \ {s} fin_si;
               θ := θ ∪ θ'
            jusqu_a ℰ₀ = ∅ ∨ s' ∉ R';
         fin_si
      fin;
      retourner (R, θ)
 |  σY (x₁:T₁:=E₁, ..., xₙ:Tₙ:=Eₙ) . φ₁ →
      R := ∅; θ := [ ];
      pour_tous s ∈ S faire
         var v₁:T₁ := ⟦E₁⟧ (ε ⊘ [s/c_s]), ..., vₙ:Tₙ := ⟦Eₙ⟧ (ε ⊘ [s/c_s]);
         θ' := θ ∪ [ {(v₁, ..., vₙ)} / Y ];
         ρ' := ρ ⊘ [ [si σ = mu alors ∅ sinon S fin_si / (v₁, ..., vₙ) ] / Y ];
         repeter
            θ'' := [ ]; stable := tt;
            pour_tous (w₁, ..., wₙ) ∈ supp(ρ'(Y)) faire
               (R', θ''') := EvalS (φ₁, ρ', ε ⊘ [w₁/x₁, ..., wₙ/xₙ]);
               si R' ≠ (ρ'(Y))(w₁, ..., wₙ) alors (ρ'(Y))(w₁, ..., wₙ) := R'; stable := ff fin_si;
               si θ''' ⊈ θ' alors θ'' := θ'' ∪ θ'''; stable := ff fin_si
            fin;
            θ' := θ' ∪ θ''
         jusqu_a stable;
         si s ∈ (ρ'(Y))(v₁, ..., vₙ) alors R := R ∪ {s} fin_si;
         θ := θ ∪ θ'
      fin;
      retourner (R, θ)
 |  exists x₀:T₀ [among E₀], ..., xₙ:Tₙ [among Eₙ] in φ₁ →
      R := ∅; θ := [ ];
      pour_tous s ∈ S faire
         pour_tous v₀:T₀[∈ ⟦E₀⟧ (ε ⊘ [s/c_s])], ..., vₙ:Tₙ[∈ ⟦Eₙ⟧ (ε ⊘ [s/c_s])] faire
            (R', θ') := EvalS (φ₁, ρ, ε ⊘ [v₀/x₀, ..., vₙ/xₙ]);
            R := R ∪ R'; θ := θ ∪ θ'
         fin
      fin;
      retourner (R, θ)
 |  forall x₀:T₀ [among E₀], ..., xₙ:Tₙ [among Eₙ] in φ₁ →
      R := S; θ := [ ];
      pour_tous s ∈ S faire
         pour_tous v₀:T₀[∈ ⟦E₀⟧ (ε ⊘ [s/c_s])], ..., vₙ:Tₙ[∈ ⟦Eₙ⟧ (ε ⊘ [s/c_s])] faire
            (R', θ') := EvalS (φ₁, ρ, ε ⊘ [v₀/x₀, ..., vₙ/xₙ]);
            R := R ∩ R'; θ := θ ∪ θ'
         fin
      fin;
      retourner (R, θ)
```

| **case** E_0 **in**
    $P_1^0 \mid \ldots \mid P_1^{n_1}$ [**where** E_1] –> φ_1
    ...
  | $P_m^0 \mid \ldots \mid P_m^{n_m}$ [**where** E_m] –> φ_m
  [| **otherwise** –> φ_{m+1}]
  endcase $\rightarrow$
    $R := \emptyset$; $\theta := [\ ]$;
    pour_tous $s \in S$ **faire**
      var v_0:$type(E_0) := [\![E_0]\!]\,(\varepsilon \oslash [s/c\_s])$;
      $(R', \theta') :=$ **si** $\exists i \in [1, m].\exists j \in [0, n_i].ok_i^j \wedge$
          $\forall l \in [0, i-1].\forall k \in [0, n_l].\neg ok_l^k$
        alors EVALS $(\varphi_i, \rho, \varepsilon \oslash ([\![P_i^j]\!] v_0)_2)$;
        [**sinon** EVALS $(\varphi_{m+1}, \rho, \varepsilon)$]
        fin_si;
      si $s \in R'$ **alors** $R := R \cup \{s\}$ **fin_si**;
      $\theta := \theta \cup \theta'$
    fin;
    retourner (R, θ)
| **case action** E_0 **in**
    α_1 [**where** E_1] –> φ_1
    ...
  | α_m [**where** E_m] –> φ_m
  [| **otherwise** –> φ_{m+1}]
  endcase $\rightarrow$
    $R := \emptyset$; $\theta := [\ ]$;
    pour_tous $s \in S$ **faire**
      var a_0:`label` $:= [\![E_0]\!]\,(\varepsilon \oslash [s/c\_s])$;
      $(R', \theta') :=$ **si** $\exists i \in [1, m].\exists \varepsilon_i \in ([\![\alpha_i]\!]\,\varepsilon a_0)_2.ok_i(\varepsilon_i) \wedge$
          $\forall l \in [1, i-1].\forall \varepsilon_j \in ([\![\alpha_j]\!]\,\varepsilon a_0)_2.\neg ok_j(\varepsilon_j)$
        alors EVALS $(\varphi_i, \rho, \varepsilon \oslash \varepsilon_i)$;
        [**sinon** EVALS $(\varphi_{m+1}, \rho, \varepsilon)$]
        fin_si;
      si $s \in R'$ **alors** $R := R \cup \{s\}$ **fin_si**;
      $\theta := \theta \cup \theta'$
    fin;
    retourner (R, θ)
fin_cas
fin

Nous ne précisons pas l'implémentation de l'évaluation des expressions et des formules sur actions, celle-ci pouvant être dérivée directement à partir des fonctions sémantiques définies respectivement à l'annexe B et à la section 3.5. Les prédicats ok_i^j et $ok_i(\varepsilon_i)$, utilisés respectivement pour l'évaluation des formules "**case**" et "**case action**", ont été définis à la section 3.7.2. ■

Nous illustrons le comportement de l'algorithme 4-2 au moyen d'un exemple.

Exemple 4-9
Considérons un programme implémentant l'accès concurrent de plusieurs processus (identifiés comme éléments d'un type énuméré `Pid`) à une ressource partagée. Les demandes et les autorisations d'accès à la ressource sont modélisées respectivement par des actions `REQUEST` et `GRANT`. Nous voulons vérifier qu'après chaque demande d'accès à la ressource partagée émise par un processus `p`, tous les chemins

menant à une autorisation d'accès de `p` sont équitables (c'est-à-dire qu'il n'existe pas de chemin qui ne contient pas d'action `GRANT p` et qui passe infiniment souvent par des états où cette action est exécutable). Cette propriété peut être exprimée par la formule φ d'alternance 2 ci-dessous.

$$
\begin{array}{l}
[\texttt{REQUEST ?}\ p : \texttt{Pid}]\ \textbf{mu}\ Y_1\ .\ \textbf{nu}\ Y_2\ .\ (\\
\quad ([\texttt{GRANT !}\ p]\ \textbf{false or}\ [\textbf{not}\ (\texttt{GRANT !}\ p)]\ Y_1)\ \textbf{and} \\
\quad [\textbf{not}\ (\texttt{GRANT !}\ p)]\ Y_2 \\
)
\end{array}
$$

Considérons le modèle STE illustré sur la figure 4.7 (s_1 étant l'état initial).

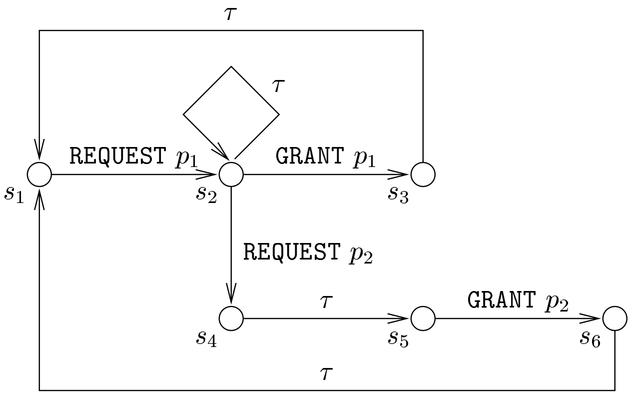

Figure 4.7: Exemple de modèle STE

Les résultats de l'évaluation de φ sur cet STE au moyen de l'algorithme 4-2 sont donnés dans la table 4.4 (où φ_1 et φ_2 dénotent respectivement les sous-formules "**mu**" et "**nu**" de φ). Pour φ, la table indique les valeurs des arguments ρ et ε d'EVALS, les transitions significatives sur lesquelles est évaluée la modalité "[`GRANT ?` p `: Pid`]" et le résultat R de l'évaluation de φ. Pour φ_1 et φ_2, outre les valeurs des arguments ρ et ε, la table indique respectivement la valeur de la variable locale $\rho'(Y_1)$ et $\rho'(Y_2)$ pendant les itérations effectuées pour évaluer ces deux sous-formules.

φ	φ_1			φ_2		
$\rho,\ \varepsilon,\ s \xrightarrow{a} s'$	$\rho,\ \varepsilon$	itér.	$\rho'(Y_1)$	$\rho,\ \varepsilon$	itér.	$\rho'(Y_2)$
$[\,],[\,],$ $s_1 \xrightarrow{\texttt{REQUEST}\ p_1} s_2$	$[\,],[p_1/p]$	1	$\emptyset$	$[\emptyset/Y_1],[p_1/p]$	1	$\{s_1,s_3,s_4,s_5,s_6\}$
					2	$\{s_3,s_4,s_5,s_6\}$
					3	$\{s_4,s_5\}$
					4	$\{s_4\}$
					5	$\emptyset$
					6	$\emptyset$
$[\,],[\,],$ $s_2 \xrightarrow{\texttt{REQUEST}\ p_2} s_4$	$[\,],[p_2/p]$	1	S	$[\emptyset/Y_1],[p_2/p]$	1	S
		2	S	$[S/Y_1],[p_2/p]$	1	S
$R = S \setminus \{s_1\}$						

Table 4.4: Evaluation globale de φ

La formule φ est satisfaite par tous les états du STE, sauf s_1 : en effet, il existe à partir de s_1 le chemin infini $s_1 \xrightarrow{\texttt{REQUEST}\ p_1} s_2 \xrightarrow{\tau} s_2 \xrightarrow{\tau} s_2 \cdots$, qui est inéquitable par rapport à l'action `GRANT` p_1. ■

La terminaison de l'algorithme 4-2 n'est pas garantie dans le cas général (voir la section 4.1), mais peut être assurée pour des classes particulières de formules XTL, comme celles indiquées à la section 4.2.4, pour lesquelles l'algorithme a une complexité exponentielle en la profondeur d'imbrication des opérateurs de point fixe. Nous pouvons envisager deux classes d'optimisations possibles permettant d'améliorer cette complexité. Premièrement, le graphe de dépendance entre les instances des variables Y peut être exploité afin d'optimiser le calcul des instances, en réévaluant uniquement celles qui sont devenues potentiellement "instables". Deuxièmement, la notion d'alternance [EL86] peut être utilisée afin de réduire le nombre d'itérations, suivant les techniques utilisées pour le μ-calcul standard d'alternance quelconque [EL86, CKS92, And94, LBC+94].

4.4 Implémentation des méta-opérateurs "$\models$" et "$[[\ldots]]$"

Nous concluons ce chapitre avec une discussion sur la sémantique et l'implémentation des méta-opérateurs "$\models$" et "$[[\ldots]]$" qui évaluent les formules XTL φ et α respectivement sur les états et sur les actions d'un modèle STE. Nous définissons la sémantique dénotationnelle de ces méta-opérateurs et nous précisons leur implémentation (par évaluation globale et/ou locale) au moyen des algorithmes présentés aux sections 4.2 et 4.3.

4.4.1 Méta-opérateurs "$\models$" et "$[[\ldots]]$" sur états

La sémantique dénotationnelle des méta-opérateurs d'évaluation des formules φ, qui a été décrite informellement à la section 2.11, est définie formellement ci-dessous.

$$\begin{aligned}
[\![E \models \varphi]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} [\![E]\!]\,\varepsilon \in [\![\varphi]\!]\,[\,]\varepsilon \\
[\![\models \varphi]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} [\![\varphi]\!]\,[\,]\varepsilon = S \\
[\![\,[[\varphi]]\,]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} [\![\varphi]\!]\,[\,]\varepsilon
\end{aligned}$$

où $type(E) = \texttt{state}$ et $fdv(\varphi) \cup fdv(E) \subseteq supp(\varepsilon)$.

Ces méta-opérateurs peuvent être implémentés de la manière suivante :

Opérateur "$E \models \varphi$". En général, le calcul de cette expression revient à évaluer φ globalement (au moyen de l'algorithme 4-1 ou de l'algorithme 4-2, suivant que φ est d'alternance 1 ou supérieure à 1) et à tester si l'état dénoté par E appartient à la sémantique de φ. Si φ est d'alternance 1 et le programme XTL est évaluable à la volée (voir la définition 4-2), φ peut être évaluée localement sur l'état dénoté par E, au moyen de l'algorithme 4-1.

Opérateur "$\models \varphi$". En général, le calcul de cette expression revient à évaluer φ globalement (utilisant un des algorithmes 4-1 ou 4-2) et à tester si l'ensemble d'états obtenu est identique à l'ensemble S de tous les états du STE. Lorsque φ est d'alternance 1 et le programme XTL est évaluable à la volée, le problème revient à évaluer l'expression équivalente suivante :

$$\texttt{init} \models [\,\mathbf{true}^*\,]\;\varphi$$

spécifiant que φ doit être satisfaite par tous les états du STE atteignables à partir de l'état initial (donc par tous les états du STE).

Opérateur "$[[\varphi]]$". Le calcul de cette expression revient à déterminer tous les états du STE satisfaisant φ. En général, ceci peut être effectué globalement, au moyen de l'algorithme 4-1 ou 4-2. Si φ est d'alternance 1 et si le programme XTL est évaluable à la volée, le calcul peut être effectué localement, au moyen de l'algorithme 4-1.

4.4.2 Méta-opérateurs "$|=$" et "$[[\ldots]]$" sur actions

La sémantique dénotationnelle des méta-opérateurs d'évaluation des formules α, qui a été décrite informellement à la section 2.12, est définie formellement ci-dessous.

$$\begin{aligned}
[\![E \models \textbf{action}\ \alpha]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} ([\![\alpha]\!]\,\varepsilon([\![E]\!]\,\varepsilon))_1 \\
[\![\models \textbf{action}\ \alpha]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} \{a \in A \mid ([\![\alpha]\!]\,\varepsilon a)_1 = \textbf{tt}\} = A \\
[\![[[\textbf{action}\ \alpha]]]\!]\,\varepsilon &\stackrel{\mathrm{d}}{=} \{a \in A \mid ([\![\alpha]\!]\,\varepsilon a)_1 = \textbf{tt}\}
\end{aligned}$$

où $\mathit{type}(E) = \texttt{label}$ et $\mathit{fdv}(\alpha) \cup \mathit{fdv}(E) \subseteq \mathit{supp}(\varepsilon)$.

Ces méta-opérateurs peuvent être implémentés de la manière suivante :

Opérateur "$E \models$ action α". Le calcul de cette expression revient à tester si l'étiquette du STE dénotée par E satisfait α, ce qui est implémentable directement par traduction vers une expression "**case action**" (voir la section 2.12).

Opérateur "$\models$ action α". Le calcul de cette expression revient à tester si toutes les étiquettes du STE satisfont α. Si le modèle est déjà construit, ceci peut être fait en testant chaque étiquette a au moyen de l'expression "$a \models$ **action** α". Ce calcul peut aussi être effectué à la volée, en testant les étiquettes générées au fur et à mesure de la construction du STE.

Opérateur "$[[$action $\alpha]]$". Le calcul de cette expression revient à déterminer l'ensemble de toutes les étiquettes du STE satisfaisant α. Ceci est implémentable directement par traduction vers un itérateur XTL qui accumule toutes les étiquettes satisfaisant α (voir la section 2.12). L'itération peut être effectuée localement ou globalement, suivant que le programme XTL est évaluable à la volée ou non.

Chapitre 5

Réalisation et applications

Ce chapitre décrit brièvement le travail de développement d'un évaluateur pour le langage XTL. Deux études de cas industrielles illustrant l'utilisation de XTL sont présentées : le protocole BRP (*Bounded Retransmission Protocol*) développé par Philips et le protocole de la couche liaison du bus série IEEE-1394 ("FireWire").

5.1 Le fragment de XTL version 1.1

Il convient de remarquer que la définition de XTL n'a pas été obtenue *ex nihilo*, mais suivant un processus expérimental comportant plusieurs améliorations successives du langage. Le travail de conception a été confronté aux contraintes posées, d'une part, par l'expressivité des constructions introduites dans le langage et, d'autre part, par l'effort d'implémentation et de connexion avec l'environnement logiciel CADP [FGK+96, GJM+97] pour la vérification de protocoles.

Plus précisément, nous avons implémenté un évaluateur capable de traiter une première version (désignée par le numéro 1.1) du langage XTL. Ensuite, tenant compte du retour fourni par l'expérimentation de l'outil XTL version 1.1 dans l'enseignement universitaire ainsi que par deux études de cas de taille "industrielle" [Mat96, SM97], nous avons étendu le langage XTL afin d'aboutir à la version (désignée par le numéro 2.0) qui est présentée dans ce document. Le langage XTL version 1.1 constitue donc un sous-ensemble du langage XTL version 2.0, composé des éléments suivants :

- types de base couramment utilisés dans les langages de programmation (booléens, nombres entiers et réels, chaînes de caractères), munis des fonctions prédéfinies associées (opérations booléennes, arithmétiques, relationnelles, ...) ;
- types tuples anonymes (structures à plusieurs champs), utilisés lors du calcul simultané de plusieurs résultats ;
- types et fonctions définis dans le programme source à vérifier, utilisés avec les mêmes notations dans les programmes XTL ;
- méta-types faisant référence aux éléments du modèle STE (états, étiquettes, transitions, ensembles d'états, d'étiquettes et de transitions), munis des méta-opérateurs d'exploration de la relation de transition (accès à l'état initial du STE et aux successeurs et prédécesseurs des états et des transitions) ;

- opérateurs spéciaux permettant d'extraire les informations contenues dans les états et les étiquettes du modèle STE et de les affecter à des variables typées ;
- constructions apparentées aux langages de programmation fonctionnels ("**let**", "**if**", "**case**", "**loop**"), permettant de définir des variables et d'effectuer des calculs conditionnels ou répétitifs ;
- constructions d'itération spécialisées (itérateurs abrégés, ensembles en compréhension, quantificateurs), permettant la description concise des prédicats et des opérateurs modaux ;
- définitions de fonctions récursives et d'opérateurs infixés (renvoyant un ou plusieurs résultats), permettant l'expression d'opérateurs temporels par exploration de la relation de transition et par calcul itératif d'ensembles d'états et de transitions ;
- définitions de macro-opérateurs et inclusions de bibliothèques prédéfinies d'opérateurs temporels, expansées syntaxiquement lors de l'évaluation des programmes XTL sur des STEs.

L'outil dédié à la version 1.1 du langage XTL est présenté brièvement à la section suivante.

5.2 Développement de l'évaluateur XTL version 1.1

Nous avons conçu, implémenté et testé un outil capable d'évaluer les programmes XTL sur un modèle STE étendu représenté en format BCG. L'évaluateur, dont le schéma fonctionnel est donné sur la figure 5.1, est composé des modules suivants :

Expanseur de formules : ce module effectue des traitements au niveau du texte source XTL. Il remplit deux fonctions : analyse et macro-expansion des définitions de formules "**formula**" (voir la section 2.13), et inclusion syntaxique des bibliothèques XTL "**library**" (voir la section 2.14). Les définitions de formules ou les inclusions de fichiers (directement ou mutuellement) récursives sont détectées et signalées. L'expanseur de formules peut être utilisé indépendamment de l'évaluateur XTL : ceci permet d'observer les effets de l'expansion des formules afin de faciliter la mise au point des programmes XTL.

Analyseur lexical et syntaxique : ce module, chargé de l'analyse syntaxique des programmes XTL obtenus après expansion, est produit en utilisant le générateur de compilateurs SYNTAX[16]. Il produit en sortie un arbre abstrait du programme XTL, qui servira de base pour les phases de compilation suivantes.

Analyseur sémantique : ce module effectue les différents traitements liés à la sémantique statique : liaison des types, des variables et des fonctions, résolution des surcharges et typage des expressions. Différentes informations provenant du fichier BCG (notamment les noms et les types des variables et des fonctions provenant du programme à vérifier) sont utilisées dans ces traitements. L'analyseur sémantique produit en sortie un arbre abstrait enrichi avec divers attributs (notamment, un nom unique et un type pour chaque objet du programme) qui seront utilisés lors de la génération de code.

Générateur de code : ce module parcourt l'arbre abstrait fourni par l'analyse sémantique et produit en sortie un programme C représentant la traduction du programme XTL initial. Le code C généré est ensuite compilé et exécuté. Pendant l'exécution, les informations contenues dans le fichier BCG (notamment la table des états, des étiquettes et des transitions) sont consultées. Les résultats de l'évaluation du programme C sont affichés sur le fichier standard de sortie (`stdout`).

[16]SYNTAX est une marque déposée de l'INRIA

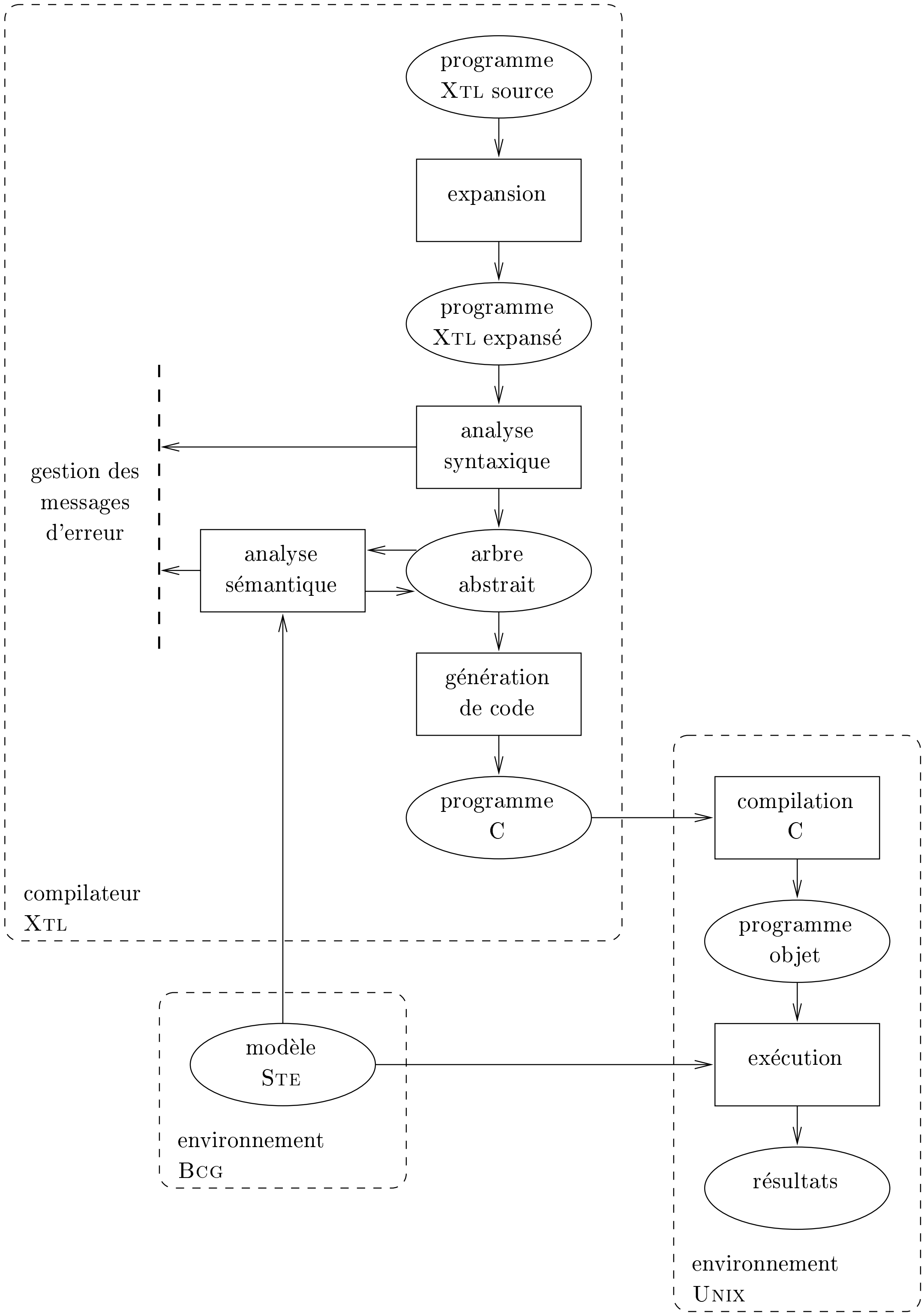

Figure 5.1: Architecture de l'évaluateur XTL

Bibliothèques C : il s'agit de modules auxiliaires d'interfaçage de l'évaluateur XTL avec l'environnement BCG, ainsi que de bibliothèques implémentant les types XTL prédéfinis (`state`, `label`, `trans`, etc.).

Nous avons choisi d'implémenter une large partie du compilateur XTL (arbres abstraits, analyse sémantique, génération de code) en types abstraits algébriques du langage LOTOS, et non directement en langage C. L'utilisation des types LOTOS permet en effet une description plus aisée des structures de données complexes et des algorithmes de parcours correspondants, tout en garantissant un typage strict, qui n'existe pas en langage C. Les types LOTOS sont ensuite traduits automatiquement en code C en utilisant le compilateur CÆSAR.ADT [Gar89b, GT93, Mat93, Sig94].

L'implémentation de l'évaluateur XTL comporte plus de 26000 lignes de grammaires SYNTAX, types abstraits LOTOS et code C (voir table 5.1).

MODULE	LIGNES DE SYNTAX	LIGNES DE LOTOS	LIGNES DE C
Expanseur de formules			4601
Analyseur lexical	114		136
Analyseur syntaxique	2406		
Analyseur sémantique		6869	6440
Générateur de code		2634	2372
Bibliothèques C			950
Total	2520	9503	14443
Total général	26522		

Table 5.1: Lignes de code source dans l'évaluateur XTL

Nous avons aussi développé plusieurs bibliothèques d'opérateurs XTL (totalisant environ 1000 lignes de code XTL) implémentant les logiques temporelles HML, CTL, ACTL et LTAC, ainsi que le μ-calcul standard. En outre, pendant le développement de l'évaluateur, nous avons créé et maintenu une base d'environ 500 programmes de test (totalisant plus de 9000 lignes de code source XTL) qui nous a permis de tester intensivement les différentes unités fonctionnelles de l'outil.

L'évaluateur XTL version 1.1 a été intégré à la version 97b de la boîte à outils CADP et est actuellement disponible sur les machines à processeur Sparc, sous système d'exploitation SunOS ou Solaris.

5.3 Application 1 : le protocole BRP

Dans cette section, nous présentons une première application du langage XTL, au cas du protocole à retransmission bornée BRP [GvdP93] conçu par Philips pour être utilisé dans les télécommandes de ses postes de télévision. Nous donnons d'abord une description en LOTOS du protocole BRP, puis nous décrivons les propriétés de sûreté et de vivacité qui en garantissent le bon fonctionnement. Nous avons déjà publié certains résultats obtenus en utilisant XTL pour vérifier le protocole BRP [Mat96, Mat97]. Dans ces deux publications, les formules étaient écrites dans une variante de la logique ACTL étendue avec valeurs (définie en XTL version 1.1) et vérifiées en utilisant l'évaluateur XTL version 1.1.

Par rapport à ces publications, les formules temporelles que nous présentons ici sont écrites en XTL version 2.0 (tel que défini dans le chapitre 2). En utilisant toute la puissance du langage XTL (notamment les filtres d'actions, les expressions régulières et les opérateurs de point fixe paramétrés), nous obtenons un gain substantiel, tant en concision qu'en généralité, pour la spécification des propriétés du protocole BRP.

5.3.1 Description du protocole BRP

Le protocole Brp est destiné à transmettre des messages (de grande taille) à travers un médium de communication non fiable en les partitionnant en (petits) paquets qui sont envoyés séquentiellement. Après chaque paquet transmis, l'émetteur attend un acquittement de la part du récepteur avant d'envoyer le paquet suivant. S'il y a une erreur de transmission et que l'acquittement n'arrive plus, l'émetteur retransmet le dernier paquet après avoir attendu un délai de garde. Le nombre de retransmissions est borné (*bounded retransmission*) : si la limite est atteinte, le protocole abandonne la transmission du message en informant de façon appropriée l'émetteur et le récepteur.

Dans les paragraphes suivants, nous présentons une modélisation de ce protocole en Lotos, produite à partir d'une description en μCrl [GvdP93] et d'une description en automates d'entrée/sortie (*I/O automata*) donnée en [HSV94].

Architecture du protocole Comme la plupart des protocoles de communication, le protocole Brp peut être décrit à l'aide de deux entités, un émetteur et un récepteur, qui échangent des messages à travers deux canaux. Suivant l'approche adoptée dans les protocoles Osi, nous examinons le comportement du protocole Brp par rapport à l'interaction avec un client émetteur et un client récepteur, qui utilisent respectivement les primitives de communication offertes par l'entité émetteur et l'entité récepteur (en termes d'architecture Osi, ceci correspond à la couche supérieure qui utilise les primitives de communication fournies par le protocole Brp). L'architecture du protocole modélisé en Lotos est illustrée en figure 5.2.

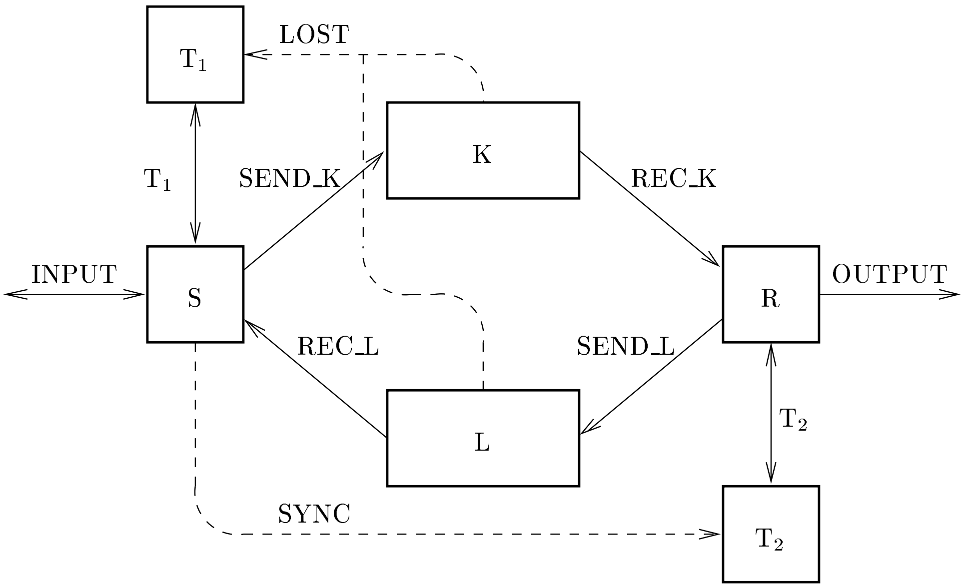

Figure 5.2: Architecture du protocole Brp

L'émetteur `S` communique avec le client émetteur sur la porte `INPUT`, à travers laquelle il reçoit les messages et renvoie les indications. Les messages sont décomposés en paquets, qui sont envoyés au canal `K` par la porte `SEND_K`. Les acquittements correspondants sont reçus du canal `L` par la porte `REC_L`. L'émetteur est équipé avec une horloge de garde `T1` qu'il peut démarrer et réinitialiser par des signaux envoyés à la porte `T1`. L'horloge peut envoyer un signal d'expiration du délai de garde à travers la même porte.

Le récepteur R reçoit les paquets issus du canal K à travers la porte REC_K, les délivre au client récepteur par la porte OUTPUT et renvoie les acquittements au canal L par la porte SEND_L. Il est aussi équipé avec une horloge de garde T2 qu'il peut démarrer et réinitialiser par des signaux envoyés à la porte T2. L'horloge peut émettre un signal d'expiration du délai de garde à travers la même porte.

Le langage LOTOS ne permettant pas la manipulation explicite du temps, nous utilisons des synchronisations auxiliaires (représentées en pointillé dans la figure 5.2) afin d'assurer que certaines contraintes de causalité sont respectées [MV93].

Ainsi, il faut s'assurer que l'horloge T1, qui est démarrée lors de l'émission d'un paquet, n'émet un signal d'expiration du délai de garde que si l'acquittement correspondant ne peut pas arriver (ce qui peut être causé soit par une perte du paquet dans le canal K, soit par une perte de l'acquittement dans le canal L). Comme il a été souligné dans [MV93], si on admet des acquittements qui arrivent après que le délai de garde correspondant s'est écoulé, le protocole peut exhiber des comportements erronés conduisant à la perte silencieuse et irrémédiable des paquets. Suivant la description μCRL donnée en [GvdP93], nous modélisons cette contrainte causale en permettant aux canaux K et L de signaler à l'horloge T1 les pertes de paquets ou d'acquittements à travers une porte auxiliaire LOST. Une fois démarrée, T1 n'est "autorisée" à émettre un signal d'expiration du délai de garde que lorsqu'elle a reçu un signal sur la porte LOST.

Une autre synchronisation auxiliaire, que nous expliquerons plus loin, est rajoutée entre l'émetteur S et l'horloge T2 sur la porte SYNC. Cette synchronisation a pour but d'assurer un redémarrage propre de l'émetteur et du récepteur après que la transmission d'un message a été abandonnée.

L'architecture du protocole BRP est décrite en LOTOS par l'expression de comportement ci-dessous.

```
hide SEND_K, REC_K, SEND_L, REC_L, T1, T2, LOST, SYNC in
      (
         (
         S [INPUT, SEND_K, REC_L, T1, SYNC] (false)
         |||
         R [OUTPUT, REC_K, SEND_L, T2]
         )
      |[SEND_K, REC_K, SEND_L, REC_L]|
         (
         K [SEND_K, REC_K, LOST]
         |||
         L [SEND_L, REC_L, LOST]
         )
      )
   |[T1, T2, LOST, SYNC]|
      (
         T1 [T1, LOST]
         |||
         T2 [T2, SYNC]
      )
```

Les entités illustrées dans la figure 5.2 sont modélisées par des processus LOTOS qui s'exécutent en parallèle. Puisque nous nous intéressons uniquement aux actions INPUT et OUTPUT effectuées par le protocole, nous avons caché toutes les autres portes à l'aide de l'opérateur "**hide**".

Nous utilisons l'opérateur de composition parallèle asynchrone "|||" pour composer les processus qui ne se synchronisent pas directement (l'émetteur et le récepteur, les deux canaux de communication et les deux horloges) et l'opérateur "|[...]|" pour connecter les groupes de processus qui se synchronisent sur les portes appropriées. Ainsi, l'émetteur et le récepteur se synchronisent avec les deux canaux de communication sur les portes SEND_K, REC_K, SEND_L et REC_L ; les deux horloges se synchronisent avec l'émetteur, le récepteur et les canaux sur les portes T1, T2, LOST et SYNC.

Les paragraphes suivants décrivent chacun de ces six processus.

L'émetteur L'émetteur est décrit par le processus S ci-dessous. Il lit sur la porte INPUT un message M provenant du client émetteur et, s'il n'est pas vide, appelle le processus auxiliaire S_1 qui gère l'envoi du message. Le paramètre booléen ALT représente la valeur courante du *bit alterné* utilisé par le protocole pour détecter la duplication des paquets.

```
process S [INPUT, SEND_K, REC_L, T1, SYNC] (ALT:Bool) : noexit :=
   INPUT ?M:Message;
      (
      [len (M) == 0] -> (* rejet des messages vides *)
         S [INPUT, SEND_K, REC_L, T1, SYNC] (ALT)
      []
      [len (M) > 0] ->
         S_1 [INPUT, SEND_K, REC_L, T1, SYNC] (ALT, M, len (M), 0)
      )
endproc
```

Le processus S_1 parcourt le message M (qui est modélisé par une liste de paquets) et envoie les paquets un par un vers le canal K. Chaque paquet est accompagné par trois bits (modélisés ici par des valeurs booléennes) : les deux premiers indiquent respectivement si le paquet est le premier ou le dernier du message courant et le troisième est le bit alterné. Le paramètre L représente la longueur initiale du message et le paramètre RN dénote le nombre courant de retransmissions qui ont été effectuées. L'horloge T1 est démarrée avant d'envoyer chaque paquet. Deux situations sont possibles.

Dans le premier cas, un acquittement arrive sur la porte REC_L. L'émetteur réinitialise l'horloge T1 et continue la transmission du message courant (appel récursif du processus S_1 avec la partie du message `tail (M)` restant à envoyer, le compteur de retransmissions réinitialisé[17] et le bit alterné inversé). Si le paquet était le dernier du message courant, une indication I_OK est issue sur la porte INPUT et l'émetteur retourne à son état initial (appel du processus S).

Dans le second cas, l'horloge T1 signale l'expiration du délai de garde, signifiant une erreur de transmission dans un des canaux. Si le compteur RN est inférieur à la valeur limite max du nombre de retransmissions, le paquet courant est retransmis (appel récursif du processus S_1 avec RN incrémenté). Si la valeur max est atteinte, l'émetteur abandonne la transmission du message courant et envoie une indication I_NOK ou I_DK au client émetteur. Avant d'accepter un nouveau message, l'émetteur doit s'assurer que le récepteur a détecté aussi l'abandon de la transmission, afin de redémarrer proprement la transmission du message suivant. Dans une implémentation réelle du protocole, ceci peut être assuré en fixant une valeur appropriée pour le délai de garde de l'horloge T2 (par exemple, une valeur supérieure à $\mathtt{max} * t$, où t représente le temps d'aller-retour d'un message sur les deux canaux). Pour modéliser cet aspect temporisé en LOTOS, nous utilisons la synchronisation auxiliaire sur la porte SYNC afin que l'horloge T2 n'expire pas avant que l'émetteur abandonne la transmission du message. Ainsi, l'émetteur envoie un signal S_READY (*sender ready*) par la porte SYNC, autorisant l'horloge T2 à expirer, puis attend une réponse R_READY (*receiver ready*) à la même porte et finalement retourne à son état initial. Il faut noter que le bit ALT est inversé à chaque appel du processus S et donc le schéma du bit alterné est poursuivi pour les messages suivants.

[17]Nous suivons ici la description en automates d'entrée-sortie [HSV94]. Dans la description μCRL [GvdP93], le compteur des retransmissions n'est pas réinitialisé après la transmission correcte d'un paquet.

```
process S_1 [INPUT, SEND_K, REC_L, T1, SYNC]
            (ALT:Bool, M:Message, L, RN:Nat) : noexit :=
   T1 !START;
      SEND_K !(len (M) == L) !(len (M) == 1) !ALT !head (M);
      (
      REC_L;                  (* reception d'un acquittement *)
         T1 !RESET;
            (
            [len (M) == 1] -> (* fin du message courant *)
               INPUT !I_OK;
                  S [INPUT, SEND_K, REC_L, T1, SYNC] (not (ALT))
            []
            [len (M) > 1] ->  (* paquets restant a envoyer *)
               S_1 [INPUT, SEND_K, REC_L, T1, SYNC]
                   (not (ALT), tail (M), L, 0)
            )
      []
      T1 !TIMEOUT;            (* expiration du delai de garde *)
         (
         [RN < max] ->
            S_1 [INPUT, SEND_K, REC_L, T1, SYNC] (ALT, M, L, RN + 1)
         []
         [RN == max] ->       (* nombre maximal de retransmissions *)
            INPUT !conf (M);
               SYNC !S_READY;
                  SYNC !R_READY;
                     S [INPUT, SEND_K, REC_L, T1, SYNC] (not (ALT))
         )
      )
endproc
```

L'horloge associée à l'émetteur est modélisée par le processus `T1` ci-dessous. Elle est démarrée par un signal `START` à la porte `T1`. Ensuite, elle peut soit être réinitialisée par un signal `RESET` à la même porte et retourner à son état initial (appel récursif du processus `T1`), soit recevoir un signal de perte d'un des canaux sur la porte `LOST`, puis émettre un signal `TIMEOUT` d'expiration du délai de garde et revenir à l'état initial.

```
process T1 [T1, LOST] : noexit :=
   T1 !START;         (* demarrage *)
      (
      T1 !RESET;      (* reinitialisation *)
         T1 [T1, LOST]
      []
      LOST;           (* indication de perte *)
         T1 !TIMEOUT; (* expiration du delai de garde *)
            T1 [T1, LOST]
      )
endproc
```

Les canaux de communication Le canal de communication non fiable K est modélisé comme un tampon à une place qui, de façon cyclique, lit un paquet à la porte SEND_K, le transmet éventuellement à la porte REC_K et retourne à son état initial. En cas de perte d'un paquet, le canal K envoie un signal d'indication sur la porte LOST. Le choix entre transmettre ou perdre un paquet est précédé par des actions silencieuses i pour assurer que l'environnement (c'est-à-dire le récepteur) ne puisse pas forcer la transmission correcte d'un paquet.

```
process K [SEND_K, REC_K, LOST] : noexit :=
   SEND_K ?FST, LST, ALT:Bool ?P:Packet; (* reception d'un paquet *)
      (
      i; REC_K !FST !LST !ALT !P;        (* transmission correcte *)
            K [SEND_K, REC_K, LOST]
      []
      i; LOST;                           (* perte avec indication *)
            K [SEND_K, REC_K, LOST]
      )
endproc
```

Le canal L a un comportement similaire, excepté le fait qu'il reçoit des acquittements à la porte SEND_L et les transmet éventuellement à la porte REC_L.

```
process L [SEND_L, REC_L, LOST] : noexit :=
   SEND_L;                               (* reception d'un acquittement *)
      (
      i; REC_L;                          (* transmission correcte *)
            L [SEND_L, REC_L, LOST]
      []
      i; LOST;                           (* perte avec indication *)
            L [SEND_L, REC_L, LOST]
      )
endproc
```

Le récepteur Le récepteur est modélisé par le processus R ci-dessous. Il attend de recevoir un paquet (le premier d'un nouveau message) à la porte REC_K, le délivre au client récepteur à travers la porte OUTPUT (accompagné par une indication I_FST ou I_OK), renvoie un acquittement au canal L et appelle le processus auxiliaire R_1 qui gère la réception du paquet courant. L'horloge T2 est démarrée chaque fois qu'un acquittement est envoyé à la porte SEND_L.

```
process R [OUTPUT, REC_K, SEND_L, T2] : noexit :=
   REC_K ?FST, LST, ALT:Bool ?P:Packet [FST];
      OUTPUT !P !ind (FST, LST);
         T2 !START;
            SEND_L;
               R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
endproc
```

Le paramètre booléen END du processus R_1 indique si le dernier paquet délivré était ou non le dernier du message courant. Le paramètre ALT représente la valeur du bit alterné attendue pour le prochain paquet à venir. Après l'envoi d'un acquittement, deux situations sont possibles.

Dans le premier cas, un paquet est reçu à la porte REC_K. S'il est nouveau (c'est-à-dire, a un bit alterné égal à ALT), l'horloge T2 est réinitialisée, le paquet est délivré au client récepteur avec l'indication appropriée (noter qu'il peut être le premier d'un nouveau message), un acquittement est envoyé au canal L, l'horloge T2 est redémarrée et le processus est répété. Si un paquet dupliqué arrive, il est simplement ignoré, un acquittement est envoyé (sans arrêter l'horloge T2) et le processus est répété.

```
process R_1 [OUTPUT, REC_K, SEND_L, T2] (END, ALT:Bool) : noexit :=
   REC_K ?FST, LST:Bool !ALT ?P:Packet;       (* nouveau paquet *)
      T2 !RESET;
         OUTPUT !P !ind (FST, LST);
            T2 !START;
               SEND_L;
                  R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
   []
   REC_K ?FST, LST:Bool !not (ALT) ?P:Packet; (* paquet duplique *)
      SEND_L;
         R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, ALT)
   []
   T2 !TIMEOUT;
      (
      [not (END)] ->                          (* paquets restant *)
         OUTPUT !I_NOK;                       (* a arriver       *)
            T2 !R_READY;
               R [OUTPUT, REC_K, SEND_L, T2]
      []
      [END] ->                                (* dernier paquet  *)
         T2 !R_READY;                         (* deja delivre    *)
            R [OUTPUT, REC_K, SEND_L, T2]
      )
endproc
```

Dans le second cas, un signal d'expiration du délai de garde est reçu de l'horloge T2, signifiant la perte du contact avec l'émetteur. Si le message courant n'a pas été complètement délivré, une indication I_NOK est fournie au client récepteur. Un signal R_READY est envoyé à T2 comme réponse au signal d'expiration et le récepteur retourne à son état initial (appel du processus R).

```
process T2 [T2, SYNC] : noexit :=
   T2 !START;
      (
      T2 !RESET;
         T2 [T2, SYNC]
      []
      SYNC !S_READY;
         T2 !TIMEOUT;
            T2 !R_READY;
               SYNC !R_READY;
                  T2 [T2, SYNC]
      )
   []
   SYNC !S_READY;
      SYNC !R_READY;
         T2 [T2, SYNC]
endproc
```

L'horloge T2 a un comportement plus compliqué que T1. Si elle est démarrée, elle peut être réinitialisée (fonctionnement normal du protocole) ou elle peut recevoir un signal S_READY sur la porte SYNC, signifiant que l'émetteur a abandonné la transmission du message courant et "autorise" T2 à émettre un signal d'expiration du délai de garde. Après un signal TIMEOUT, afin d'assurer la synchronisation de l'émetteur et du récepteur pour une nouvelle transmission, T2 attend un signal R_READY du récepteur sur la porte T2 et le retransmet à l'émetteur sur la porte SYNC.

La modélisation des contraintes causales en utilisant la synchronisation sur la porte SYNC rend possible le fait qu'un signal S_READY puisse arriver par cette porte à T2 même si celle-ci n'est pas démarrée (par exemple, ceci peut arriver quand le premier paquet est systématiquement perdu par le canal K et l'émetteur abandonne la transmission). L'horloge T2 doit réagir correctement dans ce cas et renvoyer un signal R_READY à la porte SYNC, informant l'émetteur qu'il peut commencer la transmission d'un nouveau message.

Dans les sections suivantes, nous supposons qu'un modèle STE fini a été généré à partir du programme LOTOS présenté (pour une certaine valeur maximale du nombre de retransmissions et une certaine longueur maximale des messages envoyés) à l'aide du compilateur CÆSAR. Toutes les propriétés temporelles seront interprétées sur ce modèle.

5.3.2 Propriétés de sûreté

Informellement, une propriété de sûreté d'un programme parallèle exprime le fait que "rien de mal n'arrivera". Les propriétés de sûreté que nous avons identifié assurent le bon fonctionnement du protocole BRP : l'alternance des émissions et des réceptions est respectée, le contenu des messages est transmis correctement et des indications appropriées sont délivrées au clients émetteur et récepteur. Ces propriétés sont présentées en détail dans les paragraphes suivants.

1.1. Séquencement des actions du côté émetteur. Le client émetteur, après avoir émis un message sur la porte INPUT, doit recevoir, sur la même porte, une indication I_OK, I_NOK ou I_DK provenant de l'entité émettrice du protocole. Ce comportement peut être cyclique ; dans ce cas, il faut assurer l'alternance des envois de messages et des réceptions d'indications. En outre, à partir de l'état initial du programme, il n'est pas possible que le client émetteur, avant d'avoir envoyé son premier message, reçoive spontanément une indication de la part de l'entité émettrice du protocole. Le séquencement correct des actions du côté de l'émetteur peut être décrit en XTL comme suit :

```
init |= nu Y (expect_msg : boolean := true) . (
            [ INPUT ? any Message ] (expect_msg and Y (false)) and
            [ INPUT ? any Indication ] (not expect_msg and Y (true)) and
            [ not (INPUT any) ] Y (expect_msg)
        )
```

Cette formule exprime le fait que, sur toutes les séquences d'exécution issues de l'état initial du STE, les émissions de messages (filtre d'action "INPUT ? any Message") doivent alterner avec les réceptions d'indications (filtre d'action "INPUT ? any Indication"), en commençant par un envoi. Le paramètre booléen expect_msg sert à exprimer cette alternance : il est égal à true (*resp.* false) si une émission de message (*resp.* une réception d'indication) est attendue à partir de l'état courant du STE.

La propriété ci-dessus ne porte pas sur les valeurs contenues dans les actions INPUT du STE (on peut remarquer que la formule en cause ne contient que des filtres "**any**") ; elle pourrait être exprimée aussi en ACTL ou en μ-calcul standard. Cependant, aucun de ces formalismes ne permet d'obtenir la concision de la formule XTL ci-dessus :

- Une spécification équivalente en ACTL [Mat96, Mat97] nécessiterait trois formules, exprimant les faits suivants : (a) entre deux émissions consécutives de messages, il doit y avoir une réception d'indication ; (b) entre deux réceptions consécutives d'indications, il doit y avoir une émission de message ; (c) à partir de l'état initial du STE, il ne peut pas y avoir de réception d'indication avant l'envoi d'un premier message.

- Une spécification équivalente en μ-calcul standard pourrait s'écrire en une seule formule, mais ayant une taille deux fois plus grande que la formule XTL : en effet, deux opérateurs "**nu**" imbriqués sont nécessaires pour exprimer l'alternance des envois de messages et des réceptions d'indications.

En outre, pour décrire effectivement la propriété en ACTL ou en μ-calcul standard, ces formalismes doivent être étendus avec des mécanismes permettant le filtrage du contenu des actions du STE ; si ce n'est pas le cas, chaque filtre d'action contenant une offre "**any**" doit être expansé en une disjonction sur toutes les actions du STE ayant le profil adéquat.

1.2. Séquencement des actions du côté récepteur. Le client récepteur doit observer, sur la porte `OUTPUT`, une alternance des actions marquant les débuts de messages (paquets accompagnés par des indications `I_FST`) et les fins de messages (paquets accompagnés d'indications `I_OK`, ou indications d'abandon `I_NOK`). Les messages de longueur 1 doivent être traités comme des cas particuliers, car la réception du seul paquet (accompagné d'une indication `I_OK`) marque le début aussi bien que la fin du message. En outre, à partir de l'état initial, le client récepteur ne peut pas recevoir spontanément un paquet marquant une fin de message avant d'avoir reçu un paquet marquant un début de message. Le séquencement correct des actions du côté récepteur peut être décrit en XTL comme suit :

```
init |= nu Y (expect_begin : boolean := true) . (
            [ OUTPUT any ! I_FST ] (expect_begin and Y (false)) and
            [ OUTPUT ! I_NOK ] (not expect_begin and Y (true)) and
            [ (OUTPUT any ! I_OK) or not (OUTPUT ...) ] Y (true)
        )
```

Cette formule exprime le fait que, sur chaque chemin d'exécution issu de l'état initial du STE, les débuts de messages (filtre d'action "`OUTPUT any ! I_FST`") doivent alterner avec des fins de messages (filtre d'action "`OUTPUT any ! I_OK`" ou "`OUTPUT ! I_NOK`"), en commençant par un début de message. Le paramètre booléen `expect_begin` est utilisé pour exprimer cette alternance : il est égal à `true` (*resp.* `false`) si un début (*resp.* une fin) de message est attendu(e) à partir de l'état courant du STE. L'offre générique "`...`" est utilisée pour filtrer de manière concise toutes les actions sur la porte `OUTPUT`, qui peuvent contenir une ou deux valeurs échangées par rendez-vous.

La propriété ci-dessus ne porte pas sur les valeurs contenues dans les actions `OUTPUT` du STE ; tout comme la propriété précédente, elle est exprimable en ACTL ou en μ-calcul standard, mais de façon moins concise qu'en XTL.

1.3. Transmission des paquets. Le protocole effectue la transmission de la façon suivante : chaque message émis sur la porte `INPUT` est partitionné en paquets, qui sont transmis séquentiellement sur la porte `OUTPUT`. Le message peut être reçu intégralement (si le nombre maximal de retransmissions n'est dépassé pour aucun paquet) ou partiellement (si l'entité émettrice du protocole abandonne la transmission). Néanmoins, dans un cas comme dans l'autre, la séquence des paquets reçus doit former un préfixe du message émis ; en outre, chaque paquet reçu doit être accompagné par une indication appropriée (`I_FST`, `I_INC` ou `I_OK`), suivant sa position dans le message. Cette propriété peut être exprimée en XTL de la façon suivante :

```
|= [ INPUT ? m0 : Message ] nu Y (m : Message := m0, l : Nat := len (m0)) .
       if len (m) == 0 then
           true
       else
           [ INPUT ? any Message ] false and
           [ OUTPUT ? p : Packet ? i : Indication ] (
               (p = head (m)) and
               (i = ind (len (m) == l, len (m) == 1)) and
               Y (tail (m), l)
           ) and
           [ not (OUTPUT ...) ] Y (m, l)
       endif
```

Cette formule exprime le fait qu'après chaque émission d'un message `m0` par le client émetteur (filtre d'action "`INPUT ? m0 : Message`"), les paquets reçus par le client récepteur (filtre d'action "`OUTPUT ? p : Packet ? i : Indication`") forment un préfixe de `m0` et sont accompagnés par les indications appropriées. Les paramètres `m` et `l` de l'opérateur "**nu**" mémorisent respectivement le suffixe de `m0` qu'il reste à transmettre et la longueur de `m0` ; ils servent à déterminer si le message a été complètement transmis (`m` est vide) ou si la dernière réception contient le paquet `p` et l'indication `i` attendus. L'émission d'un autre message (filtre d'action "`INPUT ? any Message`") est interdite tant que `m0` n'a pas été complètement transmis et que la transmission n'a pas été abandonnée.

L'opérateur "**if-then-else**" est nécessaire afin d'exprimer que la formule modale ne doit être évaluée que si `m` est non vide. Le méta-opérateur "|=" unaire signifie que la formule est invariante sur tous les états du STE.

La propriété ci-dessus porte sur les valeurs contenues dans les actions `INPUT` et `OUTPUT` du modèle STE ; par conséquent, elle n'est pas (directement) exprimable dans une logique temporelle classique. Une spécification équivalente en μ-calcul standard nécessiterait une formule différente pour chaque message `m0` émis dans les actions `INPUT` du STE ; en plus, chacune de ces formules devrait contenir autant d'opérateurs "**nu**" imbriqués que de paquets contenus dans `m0`.

1.4. Indications transmises à l'émetteur. Après avoir émis un message sur la porte `INPUT`, le client émetteur ne doit être informé du succès de la transmission (indication `I_OK`) que si le message a été complètement transmis au client récepteur. Ceci peut être exprimé en XTL comme suit :

```
|= [ INPUT ? any Message ] nu Y . (
       [ INPUT ! I_OK ] false and
       [ not ((INPUT ? any Message) or (OUTPUT any ! I_OK)) ] Y
   )
```

Le filtre d'action "`INPUT ? any Message`" a été rajouté à la seconde modalité du point fixe afin d'assurer que la formule "**nu**" ne prend en compte que le dernier message émis (aucun autre message n'a été émis depuis celui filtré par la modalité "[ ]" précédant l'opérateur de point fixe).

D'une façon similaire, après l'émission d'un message, le client émetteur ne doit recevoir une indication `I_DK` (signifiant la perte du dernier paquet ou du dernier acquittement) que si tous les paquets du message (sauf peut-être le dernier) ont été transmis au client récepteur. Cette propriété est exprimée par la formule XTL ci-dessous. Le paramètre `l` de l'opérateur "**nu**" mémorise le nombre de paquets de `m0` qu'il reste à transmettre. Une indication `I_DK` peut être délivrée au client émetteur lorsque `l` est inférieur ou égal à 1, ce qui signifie que les paquets de `m0` (du moins jusqu'à l'avant-dernier inclus) ont été transmis au client récepteur.

```
|= [ INPUT ? m0 : Message ] nu Y (l : Nat := len (m0)) . (
        [ INPUT ! I_DK ] (l <= 1) and
        [ OUTPUT ... ] Y (l - 1) and
        [ not ((INPUT any) or (OUTPUT ...)) ] Y (l)
   )
```

La propriété ci-dessus utilise les valeurs des messages contenus dans les actions `INPUT` du STE ; elle n'est donc pas (directement) exprimable en logique temporelle classique. Une spécification équivalente en μ-calcul standard exigerait une formule différente, ayant `len (m0)` opérateurs "**nu**" imbriqués, pour chaque message `m0` contenu dans les actions `INPUT` du STE.

1.5. Indications transmises au récepteur. Après avoir reçu le premier paquet d'un message de longueur supérieure à 1, le client récepteur ne peut recevoir une indication d'abandon `I_NOK` (signifiant la perte du contact avec l'émetteur) que si le client émetteur a reçu auparavant une indication `I_NOK` ou `I_DK`. Ceci est exprimé en XTL par la formule suivante :

```
|= [ OUTPUT ? any Packet ! I_FST ] nu Y . (
        [ OUTPUT ! I_NOK ] false and
        [ not (INPUT ? I_NOK | I_DK) ] Y
   )
```

Les opérateurs `I_NOK` et `I_DK` étant des constructeurs du type énuméré `Indication`, ils peuvent être utilisés dans des filtres avec alternatives, comme celui de l'offre "`? I_NOK | I_DK`", afin d'exprimer de façon concise le fait que l'action `INPUT` contient une indication `I_NOK` ou `I_DK`.

5.3.3 Propriétés de vivacité

Informellement, une propriété de vivacité d'un programme parallèle exprime le fait que "quelque chose de souhaitable arrivera". Les propriétés de vivacité que nous avons exhibé pour le protocole BRP concernent, d'une part, l'atteignabilité de certaines actions et, d'autre part, les réponses du protocole aux (séquences d') actions effectuées par les clients émetteur et récepteur. Ces propriétés sont décrites en détail dans les paragraphes suivants.

Afin de faciliter l'écriture de certaines propriétés de réponse de forme similaire, nous introduisons la notation abrégée "`RESPONSE`", définie comme suit :

```
formula RESPONSE (R, A) is
    < R > true and
    [ R ] mu Y . (
        < true > true and [ not (i or (A)) ] false and [ not (A) ] Y
    )
endform
```

Un état s du STE satisfait `RESPONSE (R, A)` s'il existe une séquence d'exécution issue de s, qui vérifie l'expression régulière `R`, et si, après chaque telle séquence, il est inévitable d'exécuter (éventuellement après des actions invisibles, caractérisées par le filtre d'action `i`) une action satisfaisant `A`.

1.6. Atteignabilité de l'émission d'un message. Une condition de vivacité élémentaire consiste à assurer le fait qu'à tout instant, un client émetteur connecté avec l'entité émettrice du protocole sur la porte `INPUT` soit capable, au bout d'un temps fini, d'émettre un message. Ceci peut être exprimé en XTL par la formule suivante :

```
|= mu Y . (
      < true > true and
      [ not (INPUT ? any Message) ] Y
   )
```

qui spécifie qu'à partir de tout état du STE, il est inévitable (après un nombre fini d'autres actions), d'émettre un message. Cette propriété de vivacité est assez puissante, puisqu'elle implique à la fois l'absence de blocage et l'atteignabilité inévitable de l'émission d'un message, indépendamment de la planification (équitable ou pas) des actions.

1.7. Réponse à l'émission d'un message. Le client émetteur, après avoir envoyé un message, doit inévitablement recevoir une indication (soit de transmission avec succès, soit de perte d'un paquet intermédiaire, soit de perte du dernier paquet ou du dernier acquittement) de la part du protocole. Ceci est exprimé par la formule XTL suivante :

```
|= [ INPUT ? any Message ] mu Y . (
      < true > true and
      [ INPUT ? any Message ] false and
      [ not (INPUT ? I_OK | I_NOK | I_DK) ] Y
   )
```

La deuxième modalité "[ ]" ci-dessus a été rajoutée afin d'assurer que la formule "**mu**" porte sur un seul message (c'est-à-dire qu'aucun autre message n'a été reçu depuis celui filtré par la première modalité "[ ]").

1.8. Réponse à la perte du premier paquet d'un message. Le médium de communication étant non fiable, il est possible que le premier paquet d'un message émis soit systématiquement perdu ; dans ce cas, l'entité émettrice du protocole abandonne la transmission du message et envoie une indication de perte (I_NOK) au client émetteur. Par contre, le client récepteur ne doit recevoir aucune information concernant le message (vu que le contact avec le client émetteur n'a pas été établi) et le fonctionnement du protocole doit continuer par l'attente d'une nouvelle émission de message. Ce comportement peut être caractérisé en XTL comme suit :

```
|= [ INPUT ? any Message ]
      RESPONSE (
          i* . (INPUT ! I_NOK),
          INPUT ? any Message
      )
```

Bien que la formule ci-dessus ne porte pas sur la valeur du message émis, pour l'exprimer en ACTL il aurait fallu trois opérateurs temporels (**EF**, **AG** et **A**[.**U**.]). De même, pour l'exprimer en μ-calcul standard, il aurait fallu trois opérateurs de point fixe. Ceci confirme le fait que les expressions régulières permettent une description plus naturelle des propriétés portant sur des séquences d'actions.

1.9. Réponse à la perte d'un paquet intermédiaire d'un message. Il est possible qu'un préfixe du message émis soit transmis correctement, mais qu'un paquet intermédiaire soit perdu ; dans ce cas, la transmission du message est abandonnée et une indication de perte (I_DK ou I_NOK, suivant que le paquet perdu est le dernier ou non) est envoyée au client émetteur. Le client récepteur, qui a déjà reçu les paquets précédents du message, doit obligatoirement être informé de l'abandon de la transmission par une indication I_NOK. Ceci est exprimé en XTL de la façon suivante :

```
|= [ INPUT ? any Message ]
      RESPONSE (
          i* . (OUTPUT ? any Packet ! I_FST) .
              (i* . (OUTPUT ? any Packet ! I_INC))* .
                  (INPUT ? I_NOK | I_DK),
          OUTPUT ! I_NOK
      )
```

La formule ci-dessus contient une expression régulière avec deux opérateurs "*" imbriqués, exprimant la réception des paquets intermédiaires séparés par des séquences d'actions invisibles `i`. Une spécification équivalente en ACTL nécessiterait une formule différente, contenant k opérateurs **EF**, k opérateurs **AG** et un opérateur **A**[.**U**.] pour chaque séquence de réceptions de k paquets susceptible d'apparaître dans le STE. En μ-calcul standard, une formule équivalente aurait contenu sept opérateurs de point fixe.

1.10. Réponse à la transmission correcte d'un message. Enfin, il est possible (et ce devrait être la situation normale !) qu'un message émis soit entièrement et correctement reçu par le client récepteur ; dans ce cas, le client émetteur doit obligatoirement recevoir une indication `I_DK` ou `I_OK`, suivant que le dernier acquittement a été perdu ou non. Cette propriété peut être exprimée en XTL comme suit :

```
|= [ INPUT ? any Message ]
      RESPONSE (
          i* . (OUTPUT ? any Packet ! I_FST) .
              (i* . (OUTPUT ? any Packet ! I_INC))* .
                  (OUTPUT ? any Packet ! I_OK),
          INPUT ? I_OK | I_DK
      )
```

Les mêmes remarques que pour la propriété **1.9** sont valables pour la formule ci-dessus.

5.4 Application 2 : le bus série IEEE-1394 ("FireWire")

Cette section contient une deuxième application du langage XTL, au cas du protocole de la couche liaison du bus série IEEE-1394 ("Firewire"). Nous donnons une présentation informelle de ce protocole (des présentations détaillées peuvent être trouvées dans [Lut97, SM97]), suivie de la description de ses propriétés de bon fonctionnement. Nous avons déjà publié certains résultats concernant la vérification du protocole IEEE-1394 avec XTL [SM97]. Dans cette publication, le protocole était décrit en E-LOTOS [Que97] et ses propriétés attendues étaient spécifiées en une variante d'ACTL étendue avec des valeurs (définie en XTL version 1.1) et vérifiées à l'aide de l'évaluateur XTL version 1.1.

A la différence de [SM97], nous spécifions ici les propriétés temporelles du protocole IEEE-1394 comme formules décrites en XTL version 2.0 (tel que défini dans le chapitre 2), ce qui nous permet d'obtenir, grâce aux expressions régulières, aux opérateurs de point fixe paramétrés et aux définitions de formules, des descriptions plus concises et plus naturelles de ces propriétés.

5.4.1 Description du protocole IEEE-1394

Le protocole défini par la norme IEEE-1394 [IEE95] ("FireWire") est le résultat de l'unification de différents bus série tels que VME, MULTIBUS II et FUTURE BUS. Il s'agit d'un bus rapide particulièrement adapté à la transmission des données pour des périphériques multimédias haut débit connectés aux postes de travail. A cette norme se sont ralliés un bon nombre de constructeurs, parmi lesquels AT&T, Compaq, Hewlett-Packard, IBM, Kodak, Microsoft, Sony, Texas Instruments, etc.

Architecture du protocole Nous résumons ci-dessous les principales caractéristiques de ce protocole. Une description plus détaillée peut être trouvée en [Lut97, SM97]. L'architecture du bus suppose l'existence de `n` nœuds répartis (entités ayant une adresse unique) connectés par une ligne série (que nous appellerons CABLE dans la suite). Le protocole exécuté par chaque nœud connecté au bus IEEE-1394 est composé de trois niveaux empilés :

le niveau supérieur, ou *couche transaction* (notée TRANS), implémente le protocole "demande-réponse" (*request-response*) conforme au standard IEEE-13313 (*Control and Status Register Architecture for Microcomputer Buses*). Il offre aux applications exécutées par le nœud les primitives de lecture, écriture et verrouillage transactionnel des données.

le niveau intermédiaire, ou *couche liaison* (notée LINK), implémente un protocole "datagramme avec confirmation de réception" (*acknowledged datagram*). Il gère la transmission et de la réception des paquets, ainsi que le contrôle d'horloge pour les transmissions "isochrones" (c'est-à-dire effectuées à des moments de temps bien précisés).

le niveau inférieur, ou *couche physique* (notée PHY), fournit les services d'initialisation et d'arbitrage nécessaires pour assurer qu'un seul nœud émet sur le CABLE à un moment donné. Ce niveau effectue aussi la conversion entre les données manipulées par le LINK et les signaux électriques manipulés par le CABLE. Par la suite, nous utiliserons le terme BUS pour désigner l'ensemble constitué de la couche physique PHY et du CABLE.

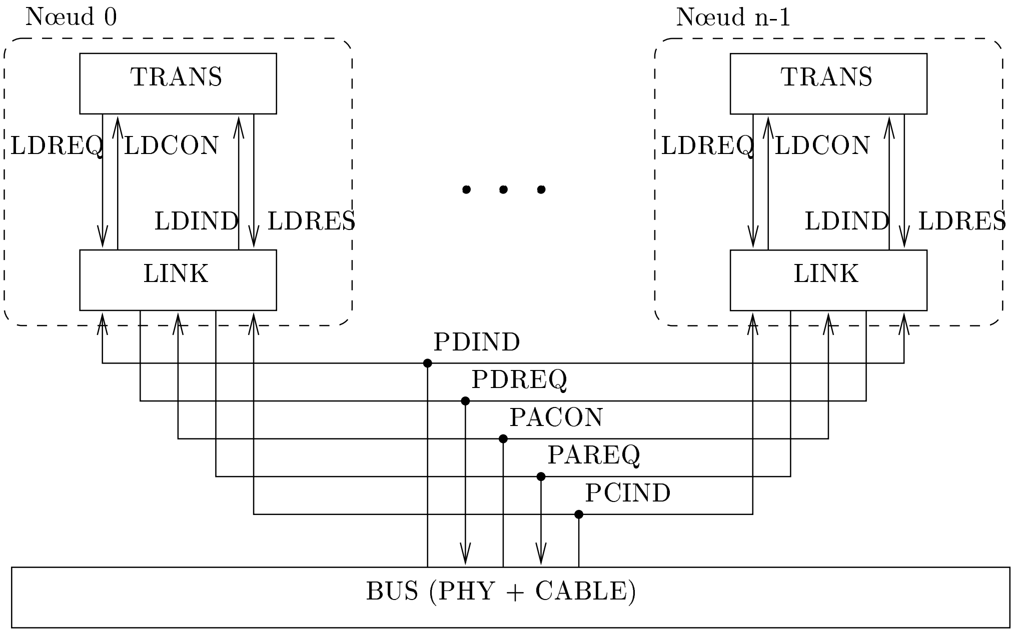

Figure 5.3: Architecture du bus série

L'architecture du bus IEEE-1394 est représentée sur la Figure 5.3. Chaque nœud est modélisé par une couche TRANS et une couche LINK qui s'exécutent en parallèle, en se synchronisant sur les primitives du niveau LINK : LDREQ, LDCON, LDIND et LDRES. L'ensemble de n nœuds (indexés de 0 à n-1) se synchronise avec la couche PHY sur les primitives du niveau PHY : PDIND, PDREQ, PACON, PAREQ et PCIND.

La couche liaison Le protocole de la couche liaison a été conçu pour transmettre les paquets de données sur un milieu non fiable, en les divisant en *signaux*. Ces signaux sont ensuite envoyés séquentiellement à la couche physique d'une manière asynchrone ou isochrone. Dans notre étude de cas, nous avons considéré seulement la partie asynchrone du protocole.

Le protocole asynchrone du LINK assure la transmission d'un paquet à un nœud précis (*point-to-point communication*) ou à tous les nœuds (*broadcast*). Le niveau liaison offre les primitives suivantes :

- LDREQ (*Link Data Request*) est utilisée par le TRANS pour demander au LINK la transmission d'un paquet asynchrone. La couche liaison doit confirmer cette demande (voir la primitive LDCON ci-dessous). La primitive LDREQ a quatre paramètres : l'identificateur du nœud source, l'identificateur du nœud destination (ou n s'il s'agit d'une diffusion à tous les nœuds), l'entête et la donnée de la demande.
- LDCON (*Link Data Confirmation*) est utilisée par le LINK pour confirmer la transmission d'un paquet. La couche liaison doit envoyer cette confirmation après la terminaison de la transmission du paquet. La primitive LDCON a deux paramètres : l'identificateur du nœud et un code de confirmation.
- LDIND (*Link Data Indication*) est utilisée par le LINK pour indiquer au TRANS l'arrivée d'un paquet. La couche transaction doit répondre à cette indication (voir la primitive LDRES ci-dessous). La primitive LDIND a quatre paramètres : l'identificateur du nœud, le type du paquet (personnel ou diffusion), le contenu du paquet (entête et donnée) et un code de réception (réception normale ou erronée).
- LDRES (*Link Data Response*) est utilisée par le TRANS pour répondre à la réception d'un paquet et compléter la transaction en envoyant une confirmation de réception. Cette primitive a trois paramètres : l'identificateur du nœud, la confirmation et un code pour indiquer au LINK si le TRANS veut ou non transmettre un paquet en réponse. Ce code peut avoir les valeurs suivantes : "release" si le TRANS ne veut pas envoyer un paquet de réponse, "hold" dans le cas contraire, ou "no_op" si le paquet reçu est une diffusion à tous les nœuds.

La couche physique La couche physique du protocole a deux fonctions principales : l'arbitrage de l'accès des LINKs au CABLE (voir la primitive PAREQ ci-dessous) ainsi que la transmission et la réception des signaux (voir les primitives PDREQ et PDIND ci-dessous). Nous décrivons les principales caractéristiques de ces fonctions et nous précisons ensuite les services offerts par la couche physique.

Le protocole d'arbitrage implémenté par la couche physique est basé sur le concept d'*intervalle d'équité* (*fairness interval*), illustré sur la Figure 5.4. Pendant un intervalle d'équité, chaque LINK ne peut transmettre qu'au plus un paquet asynchrone sur le BUS ; en revanche, il peut transmettre plusieurs paquets d'acquittement (*acknowledge*) confirmant des réceptions. Le temps nécessaire pour la transmission d'un paquet suivi par une séquence (éventuellement vide) de paquets de confirmation est appelé *sous-action*. Un intervalle d'équité peut contenir une ou plusieurs sous-actions délimitées par des signaux d'*interstice de sous-action* (*subaction gap*, en abrégé SUBACTGAP) envoyés par le BUS à l'aide du service PDIND ci-dessous. Les intervalles d'équité sont séparés par des signaux d'*interstice de demande d'arbitrage* (*arbitration reset gap*, en abrégé ARBRESGAP) envoyés par le BUS. Un signal ARBRESGAP est émis lorsque le BUS, après quelques sous-actions, a été inoccupé pendant un certain temps.

Le protocole de transmission suppose un milieu de communication non fiable, les signaux pouvant être corrompus ou perdus.

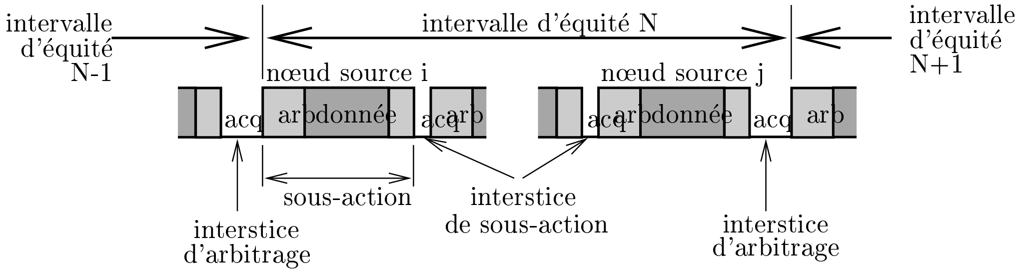

Figure 5.4: Structure de l'intervalle d'équité

Le niveau physique fournit les primitives suivantes :

- PAREQ (*PHY arbitration request*) est utilisée par le Link pour demander au Phy de commencer une procédure d'arbitrage du bus. Cette demande doit être confirmée par la couche physique quand la procédure d'arbitrage est terminée (voir la primitive PACON ci-dessous). La primitive PAREQ a deux paramètres : l'identificateur du nœud et la méthode d'arbitrage qui doit être utilisée par la couche physique. Ce dernier paramètre peut avoir deux valeurs : IMMEDIATE, signifiant que la couche physique doit accorder l'accès au Bus dès qu'il est libre ; FAIR, signifiant que la couche physique doit commencer l'arbitrage soit après un interstice de sous-action (si le Link n'a pas encore transmis un paquet dans l'intervalle d'équité courant), soit dans l'intervalle d'équité suivant (si le Link a déjà gagné l'accès au Bus dans l'intervalle d'équité courant).

- PACON (*PHY arbitration confirmation*) est utilisée par le Phy pour confirmer le résultat d'un arbitrage. Cette primitive a deux paramètres : l'identificateur du Link demandeur et le résultat de l'arbitrage (WON si le Link a gagné l'arbitrage ou LOST sinon).

- PCIND (*PHY clock indication*) est utilisée par le Phy pour signaler au Link qu'un signal peut être transmis sur le Cable. Le niveau liaison doit répondre à cette indication en demandant la transmission d'un signal (voir la primitive PDREQ ci-dessous). La primitive PCIND a comme paramètre l'identificateur du nœud transmetteur.

- PDREQ (*PHY data request*) est utilisée par le Link pour transmettre des signaux à la couche physique. Un signal doit être transmis à chaque indication PCIND émise par l'horloge. La primitive PDREQ a deux paramètres : l'identificateur du nœud et le signal à transmettre.

- PDIND (*PHY data indication*) est utilisée par le Phy pour signaler au Link une modification dans l'état de la couche physique. Ce changement peut être soit la réception d'un signal, soit un interstice de sous-action, ou encore d'autres événements (l'interstice d'arbitrage étant modélisé par un signal spécial ARBRESGAP). La primitive PDIND a deux paramètres : l'identificateur du nœud et le signal qui code le changement d'état du Phy.

La couche transaction Pour chaque nœud, la couche Trans offre aux applications exécutées par le nœud les primitives de lecture, écriture et verrouillage transactionnel des données. Les transactions utilisent les quatre primitives du Link en suivant le protocole Osi d'établissement de connexion (illustré sur la Figure 5.5) :

demande (primitive LDREQ), qui est utilisée par le Trans demandeur afin de commencer une transaction ;

indication (primitive LDIND), qui est utilisée pour indiquer au Trans répondeur l'arrivée d'une demande ;

réponse (primitive LDRES), qui est utilisée par le Trans répondeur pour transmettre des codes de réponse ou des paquets de données au Trans demandeur ;

confirmation (primitive LDCON), qui est utilisée pour indiquer au Trans demandeur l'arrivée de la réponse correspondante.

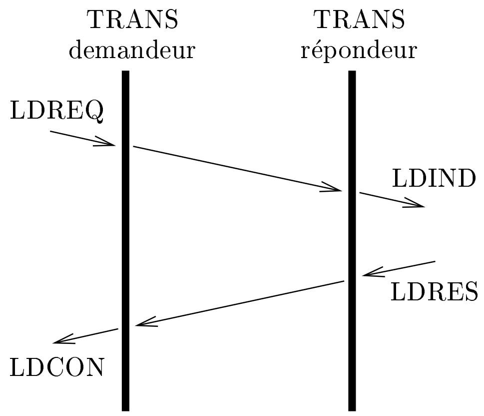

Figure 5.5: Déroulement d'une transaction

A tout moment, la couche Trans d'un nœud peut traiter des transactions sortantes (demandes) et des transactions entrantes (réponses).

Dans la section suivante, nous supposons qu'un modèle Ste fini a été généré à partir d'une description du protocole Ieee-1394, pour un nombre fixé de nœuds et un nombre limité de paquets circulant sur le bus (pour plus de détails, voir [SM97]). Toutes les propriétés temporelles du protocole seront interprétées sur ce modèle.

5.4.2 Propriétés

Les propriétés de bon fonctionnement du protocole de la couche liaison du bus Ieee-1394 ont été formulées en langage naturel par Luttik [Lut97]. Dans les paragraphes suivants, nous caractérisons ces propriétés comme formules temporelles décrites en Xtl.

Afin de faciliter l'écriture de certaines propriétés ayant une forme similaire, nous introduisons la notation abrégée "AFTER_INEV", définie comme suit :

```
formula AFTER_INEV (R, A, B) is
    [ R ] mu Y . (
        < true > true and [ not ((A) or (B)) ] false and [ not (B) ] Y
    )
endform
```

Un état s du STE satisfait `AFTER_INEV (R, A, B)` si, après chaque séquence d'exécution issue de s qui satisfait l'expression régulière `R`, il est inévitable d'exécuter (éventuellement après des actions satisfaisant `A`) une action satisfaisant `B`.

2.1. Absence de blocage. Afin d'obtenir un modèle du protocole de taille raisonnable, nous avons limité le nombre de paquets circulant sur le BUS (pour plus de détails sur les scénarios considérés, voir [SM97]). Par conséquent, le modèle contient des comportements finis : lorsque tous les paquets émis par les LINKs ont été transmis sur le BUS, le protocole atteint un état de terminaison normale.

L'absence de blocage doit être exprimée dans le contexte de ces comportements finis : il s'agit de distinguer les états de terminaison normale des états de blocage. Un examen attentif de la description du protocole a révélé le fait que les états de terminaison normale apparaissent après un signal d'interstice d'arbitrage (action `ARBRESGAP`), éventuellement suivi par une séquence de confirmations (actions `LDCON`). La formule XTL suivante exprime qu'il n'est pas possible d'atteindre, à partir de l'état initial, un état de blocage qui ne soit pas un état de terminaison normale :

```
init |= not < true* . not (ARBRESGAP or (LDCON ...)) . (LDCON ...)* >
            [ true ] false
```

Une formulation équivalente en ACTL (étendue avec des filtres d'actions) nécessiterait, à la place de la modalité "⟨ ⟩" ci-dessus, deux opérateurs **EF** imbriqués à travers d'une modalité "⟨ ⟩". Bien que la formule ACTL permettrait une spécification tout aussi concise que la formule XTL ci-dessus, l'expression régulière XTL nous semble plus naturelle pour décrire la propriété.

2.2 Transmission des paquets entre deux interstices de sous-actions. Entre deux signaux consécutifs d'interstice de sous-actions (`SUBACTGAP`) émis sur la porte `PDIND`, au plus deux paquets asynchrones peuvent circuler sur le BUS. Cette propriété de sûreté du protocole peut être spécifiée en XTL par la formule suivante :

```
|= [ PDIND ? any Node ! SUBACTGAP ]
       nu Y (p : Nat := 0) . (
           [ LDCON ... ] if p < 2 then Y (p + 1) else false endif
           and
           [ not ((PDIND ? any Node ! SUBACTGAP) or (LDCON ...)) ] Y (p)
       )
```

Pour exprimer le fait qu'un paquet a circulé sur le BUS, nous avons utilisé le filtre d'action "`LDCON ...`", dénotant la confirmation d'un envoi de paquet reçue par la couche TRANS d'un nœud quelconque. Le paramètre `p` de l'opérateur "**nu**" mémorise le nombre de paquets qui ont été déjà envoyés sur le BUS depuis le dernier signal délimitant un interstice de sous-actions (filtre d'action "`PDIND ? any Node ! SUBACTGAP`"). Une fois que `p` est égal à 2, la formule de point fixe interdit qu'un autre paquet soit émis sur le BUS avant le prochain signal d'interstice de sous-actions.

Une formulation équivalente en ACTL nécessiterait, à la place de l'opérateur "**nu**", trois opérateurs **AG** (chacun d'entre eux paramétré par la formule d'actions utilisée dans la dernière modalité "[ ]" ci-dessus) imbriqués à travers trois modalités "[ ]". Cette propriété aurait pu être exprimée aussi au moyen d'expressions régulières XTL, mais d'une manière moins concise que la formule de point fixe ci-dessus.

2.3. Réponse à la requête de transmission d'un paquet. Si le niveau TRANS d'un nœud $0 \leq \mathtt{id} \leq n-1$ a émis une demande de transmission de paquet (action `LDREQ`) et le niveau LINK de `id` fait une demande d'arbitrage du BUS (action `PAREQ`) après chaque signal d'interstice de sous-action — et avant l'interstice d'arbitrage suivant (action ARBRESGAP) — alors le niveau TRANS du nœud respectif doit obligatoirement recevoir une confirmation de transmission (action LDCON). Cette propriété, qui combine des aspects de sûreté et de vivacité, peut être exprimée par la formule XTL suivante :

```
|= AFTER_INEV (
        (LDREQ ? id : Node) .
            not (ARBRESGAP or (PDIND ! id ! SUBACTGAP) or (LDCON ! id))* .
                (PDIND ! id ! SUBACTGAP) .
                    not (ARBRESGAP or (PAREQ ! id))* .
                        (PAREQ ! id),
        not (ARBRESGAP),
        LDCON ! id
    )
```

Les occurrences du filtre d'action ARBRESGAP dans l'expression régulière et dans le second argument de l'opérateur AFTER_INEV assurent que la formule porte sur le même intervalle d'équité (aucun signal ARBRESGAP n'a été émis pendant ce temps). La formule ci-dessus utilise les facilités de capture et de propagation des valeurs dans les expressions régulières XTL : le numéro id du nœud qui fait la demande de transmission est extrait par le filtre d'action "LDREQ ? id : Node" et utilisé par la suite (la définition de l'opérateur AFTER_INEV autorise qu'une variable affectée par son premier argument puisse être utilisée dans ses deux autres arguments). Puisque cette propriété porte sur les valeurs, elle n'est pas directement exprimable en ACTL : une formulation équivalente nécessiterait une formule différente, comportant deux opérateurs **AG**, trois modalités "[ ]" et un opérateur **A** [.**U**.], pour chaque nœud contenu dans les requêtes LDREQ du modèle STE.

2.4. Réponse à la requête d'un arbitrage immédiat. Chaque demande d'arbitrage immédiat émise par le niveau LINK d'un nœud $0 \leq$ id $\leq n-1$ (action PAREQ avec paramètres id et IMMEDIATE) est obligatoirement suivie par une confirmation d'accès au BUS (action PACON avec le paramètre WON). Cette propriété, qui combine des aspects de sûreté et de vivacité, peut être spécifiée par la formule XTL suivante :

```
|= AFTER_INEV (
        PAREQ ? id : Node ! IMMEDIATE,
        not (PAREQ ! id ! IMMEDIATE),
        PACON ! id ! WON
    )
```

Tout comme la formule XTL spécifiant la propriété **2.3**, la formule ci-dessus utilise l'extraction et la propagation de valeurs à travers les arguments de l'opérateur AFTER_INEV. Une formulation équivalente en ACTL nécessiterait une formule différente, ayant une modalité "[ ]" et un opérateur **A** [.**U**.], pour chaque nœud id présent dans les requêtes PAREQ d'arbitrage immédiat contenues dans le modèle STE.

2.5. Réponse à la requête d'un arbitrage équitable. Dans chaque intervalle d'équité (délimité par deux actions ARBRESGAP consécutives), aucun nœud $0 \leq$ id $\leq n-1$ ne peut recevoir une confirmation d'accès au BUS (action PACON avec paramètre WON) après une demande d'arbitrage équitable (action PAREQ avec paramètres id et FAIR) plus d'une fois. Cette propriété de sûreté peut être exprimée en XTL à l'aide de la formule suivante :

```
|= [ ARBRESGAP .
       not (ARBRESGAP)* .
           (PAREQ ? id : Node ! FAIR) . (PACON ! id ! WON) .
               not (ARBRESGAP)* .
                   (PAREQ ! id ! FAIR) . (PACON ! id ! WON)
   ] false
```

Tout comme les formules décrivant les propriétés **2.3** et **2.4**, la formule ci-dessus utilise l'extraction et la propagation des valeurs à travers une expression régulière contenue dans une modalité XTL. Une formulation équivalente en ACTL devrait contenir une formule différente, comportant cinq modalités "[ ]" et deux opérateurs **AG** imbriqués, pour chaque nœud `id` présent dans les requêtes `PAREQ` d'arbitrage équitable contenues dans le modèle STE. Cette propriété pourrait être décrite aussi au moyen d'opérateurs de point fixe, mais d'une manière moins naturelle qu'avec l'expression régulière XTL ci-dessus.

5.5 Discussion

La spécification en XTL des propriétés de bon fonctionnement du protocole BRP (voir les sections 5.3.2 et 5.3.3) et du protocole IEEE-1394 (voir la section 5.4.2) confirme les avantages de notre approche par rapport aux formalismes classiques :

- Même pour les propriétés qui ne portent pas sur les valeurs contenues dans le STE (comme les propriétés **1.1**, **1.2**, **1.4** et **2.2**), le paramétrage des opérateurs de point fixe en XTL permet une spécification plus concise que les logiques temporelles classiques ou le μ-calcul standard. De la même façon, le filtrage des actions structurées (mécanisme absent dans les logiques temporelles standard) permet une description plus concise des formules modales (voir les propriétés **1.5**, **1.6**, **1.7** et **2.1**).

- Les expressions régulières, qui permettent une description naturelle des propriétés sur séquences d'actions, et les facilités de définition et d'expansion de formules (voir les propriétés **1.8**, **1.9**, **1.10**, **2.3**, **2.4** et **2.5**), constituent également deux autres facteurs de concision par rapport aux logiques temporelles classiques.

- La syntaxe et la sémantique statique de XTL permettent d'utiliser dans les formules temporelles les objets du programme source à vérifier (notamment, les fonctions et les types). Ceci permet une expression naturelle des propriétés temporelles comportant des valeurs (voir à ce sujet l'utilisation des fonctions `len`, `ind`, `head` et `tail` dans la propriété **1.3**), d'autant plus que ces objets sont désignés en utilisant les mêmes notations que dans le programme à vérifier.

- Les spécifications que nous avons produit sont indépendantes des paramètres des applications (nombre maximal de retransmissions, contenu et taille maximale des messages pour le protocole BRP, nombre de nœuds pour le protocole IEEE-1394), ce qui fait que l'utilisateur peut vérifier plusieurs variantes des descriptions de ces protocoles, chacune ayant des paramètres différents, sans modifier les spécifications XTL. Ceci est réalisé essentiellement grâce aux formules sur actions, qui permettent d'extraire les valeurs échangées par rendez-vous, évitant l'utilisation de quantificateurs qui seraient dépendants des domaines des valeurs contenues dans les modèles STE (et, par conséquent, des paramètres des applications) et qui s'évalueraient moins efficacement.

Conclusion

Bilan

La spécification et la vérification des propriétés des programmes parallèles constituent une étape indispensable pour en garantir le bon fonctionnement. Depuis plus de quinze ans, les logiques temporelles ont été proposées et reconnues comme un formalisme adapté à la spécification de propriétés. Toutefois, les travaux consacrés aux logiques temporelles se sont orientés dans de multiples directions. Un grand nombre de logiques ont été définies, la plupart d'entre elles se focalisant sur un aspect précis du problème, par exemple l'expressivité ou, au contraire, l'efficacité des algorithmes d'évaluation. De surcroît, des approches antagonistes (par exemple, l'opposition des logiques temporelles linéaires et arborescentes) ont encore accentué l'aspect fragmentaire des travaux dans ce domaine.

Dans cette étude, nous avons cherché, au contraire, une démarche unificatrice basée sur une approche pragmatique adaptée aux besoins de la spécification et de la vérification :

- Nous avons étudié attentivement la littérature scientifique consacrée aux logiques temporelles, ce qui nous a permis d'établir une classification des différentes logiques temporelles existantes, ainsi que des algorithmes d'évaluation proposés (voir chapitre 1).

- En confrontant l'état de l'art avec notre propre expérience acquise lors de l'étude de cas réalistes, nous avons constaté que les logiques temporelles "classiques", qui sont interprétées sur un vocabulaire d'actions atomiques (noms de portes ou de canaux de communication), ne sont pas adaptées à la vérification des programmes LOTOS dont les actions sont structurées (contenant des noms de portes et des listes de valeurs échangées par rendez-vous). Ceci nous a incité à étendre les logiques temporelles avec des constructions permettant de manipuler les valeurs contenues dans les états et les actions du modèle à vérifier.

- Ce travail a débouché sur la définition de la version 1.1 du langage XTL (*eXecutable Temporal Language*), un formalisme permettant de spécifier les propriétés temporelles des programmes parallèles écrits dans des langages avec valeurs. Cette première version du langage comporte un certain nombre de traits originaux qui la distinguent des logiques temporelles classiques :

  - un modèle de base général, les systèmes de transitions étiquetées (STES) étendus (voir la définition 1-4), qui englobent comme cas particuliers les structures de Kripke et les systèmes de transitions étiquetées. Concrètement, ce modèle est implémenté par le format d'automates BCG (*Binary Coded Graph*) [Gar94] ;
  - types de données, soit prédéfinis (booléens, entiers, réels, chaînes de caractères, munis des opérations usuelles), soit définis dans le programme source à vérifier (et repris dans le format BCG), soit structurés (types tuples anonymes) construits à partir des types précédents ;

- méta-types et opérations associées permettant de manipuler les éléments du modèle (états, étiquettes, transitions, ensembles d'états, ensembles d'étiquettes, ensembles de transitions, état initial, successeurs et prédécesseurs des états et transitions) ;
- opérateurs spéciaux permettant d'extraire les informations contenues dans les états et les étiquettes du modèle et de les affecter à des variables typées ;
- constructions inspirées des langages de programmation fonctionnels ("**let**", "**if**", "**case**", "**loop**"), permettant de définir des variables et d'effectuer des calculs conditionnels ou répétitifs. Des constructions d'itération spécialisées (itérateurs abrégés, ensembles en compréhension, quantificateurs), permettant la description concise des prédicats et des opérateurs modaux, sont également fournies ;
- définitions de fonctions récursives, permettant l'expression d'opérateurs temporels par exploration de la relation de transition et par calcul itératif d'ensembles d'états ;
- définitions d'opérateurs temporels paramétrés et inclusions de bibliothèques prédéfinies, expansées syntaxiquement lors de l'évaluation des programmes XTL sur des STEs étendus.

• Nous avons entièrement implémenté ces idées, ce qui a donné naissance à la version 1.1 de l'évaluateur XTL. Cette mise en œuvre a nécessité un important travail d'implémentation (plus de 26000 lignes de programmes C, LOTOS et SYNTAX), puisqu'il s'agit d'un compilateur complet, incluant : un préprocesseur (expanseur), un analyseur lexical et syntaxique, un vérificateur de sémantique statique, un générateur de code C et les bibliothèques C correspondantes.

• Nous avons développé plusieurs bibliothèques qui implémentent les logiques temporelles classiques (HML, CTL, ACTL, LTAC, ainsi qu'un fragment du μ-calcul standard, ces bibliothèques totalisant environ 1000 lignes de code XTL) dans la version 1.1 du langage XTL. Nous avons aussi créé et maintenu une série de jeux de tests servant à valider notre implémentation.

• Nous avons utilisé l'évaluateur XTL avec succès pour la validation de deux études de cas industrielles (voir le chapitre 5) : le protocole BRP [Mat96] développé par Philips et le protocole de la couche liaison du bus série IEEE-1394 ("FireWire") [SM97]. Nous avons aussi employé cet outil pour l'enseignement : les étudiants du magistère informatique de l'Université Joseph Fourier de Grenoble ont ainsi utilisé l'évaluateur XTL 1.1 pour valider des algorithmes répartis (exclusion mutuelle, élection sur un réseau à jeton, ...). Outre la validation de l'évaluateur, ces différentes applications ont mis en évidence l'intérêt de notre approche : les propriétés temporelles comportant des valeurs ont pu être exprimées de manière concise et naturelle, tout en étant évaluées avec des performances comparables aux autres outils existants. C'est pourquoi l'évaluateur XTL a pu être intégré à la version 97b de la boîte à outils CADP.

• Ce travail d'implémentation et d'expérimentation nous a permis d'évaluer concrètement la version 1.1 du langage XTL sur des applications pratiques. C'est ainsi que nous avons été conduits à proposer une nouvelle version 2.0 du langage, présentée dans ce document. Par rapport à la version précédente, la version 2.0 introduit des améliorations notables, qui autorisent une description encore plus aisée des propriétés temporelles :

  - modalités "$\langle\ \rangle$" et "[ ]" étendues avec des variables typées et des expressions régulières ;
  - opérateurs de point fixe "**mu**" et "**nu**" paramétrés par des variables typées ;
  - opérateur de bouclage "**@**" permettant de caractériser des séquences ω-régulières d'actions ;
  - méta-opérateurs "**current**" permettant d'accéder, dans une formule, à l'état ou à l'action sur laquelle la formule est couramment évaluée ;
  - méta-opérateurs "[[ ]]" et "|=" d'évaluation des formules sur états et sur actions.

- Nous avons défini formellement la syntaxe (voir la section 2.1), la sémantique statique (voir l'annexe A) et la sémantique dénotationnelle (voir le chapitre 3 et l'annexe B) de la version 2.0 du langage XTL. La sémantique proposée pour les opérateurs de point fixe paramétrés constitue une extension naturelle de la sémantique du μ-calcul standard et des fonctions récursives des langages de programmation.

- Nous avons ensuite abordé le problème de l'implémentation efficace de la version 2.0 de XTL. Pour cela, nous avons identifié un ensemble de constructions primitives constituant un sous-langage minimal des formules XTL, ainsi que la traduction des constructions dérivées en termes des constructions de base (voir la section 3.8).

  Nous avons également proposé des algorithmes d'évaluation des formules XTL sur des modèles finis (voir le chapitre 4). Généralisant l'approche utilisée pour le μ-calcul standard, nous traduisons les formules de point fixe d'alternance 1 (l'alternance étant une mesure du degré de récursion mutuelle des opérateurs de plus petit et de plus grand point fixe) vers des systèmes d'équations booléennes paramétrées par des variables typées. Nous proposons une procédure semi-décidable permettant (sous des conditions suffisantes de terminaison) de traduire ces systèmes paramétrés vers des systèmes booléens “purs”, qui peuvent être résolus en adaptant les algorithmes efficaces existants [AC88, CS91b, And94, VL94]. Cette méthode conduit à des algorithmes d'évaluation globaux (opérant sur un modèle STE déjà construit) aussi bien que locaux (le modèle STE étant construit au fur et à mesure de l'évaluation de la formule). Pour les formules XTL d'alternance quelconque, nous avons proposé un schéma de traduction permettant (toujours sous des conditions suffisantes de terminaison) de réduire les domaines des paramètres des points fixes à des ensembles finis et de calculer ces points fixes de manière itérative, en utilisant des algorithmes similaires à ceux présentés dans [EL86, And92, LBC+94].

- Enfin, nous avons réécrit en XTL version 2.0 les bibliothèques définissant les logiques temporelles CTL et ACTL précédemment écrites en XTL version 1.1 et nous avons également proposé une traduction pour un fragment “purement arborescent” de la logique modale dédiée à μCRL [GvV94] (voir l'annexe C). Nous avons aussi reformulé en XTL version 2.0 les propriétés de correction des protocoles BRP et IEEE-1394 (voir le chapitre 5) précédemment décrites en XTL version 1.1. Ceci a mis en évidence le gain en concision et en généralité apporté par les constructions introduites en XTL version 2.0 (notamment les expressions régulières et les opérateurs de point fixe paramétrés).

Les résultats de ce travail ont confirmé l'intérêt de notre approche : le langage XTL permet une description concise et naturelle des propriétés temporelles portant sur les valeurs, tout en bénéficiant d'une efficacité d'évaluation comparable aux autres outils existants.

Perspectives

Le travail présenté dans ce document peut être continué dans plusieurs directions.

Tout d'abord, il faudrait implémenter les algorithmes d'évaluation des formules XTL de point fixe présentés au chapitre 4, en particulier ceux proposés pour les formules d'alternance 1. Ce fragment présente un intérêt pratique considérable car, d'une part, il permet une traduction directe de plusieurs logiques temporelles (comme CTL, ACTL, LTAC ou PDL-Δ) et, d'autre part, grâce aux méta-opérateurs “**current**” et au paramétrage des opérateurs de point fixe, il autorise l'expression de certaines propriétés d'équité (comme celles exprimables en ECTL), ce qui n'est pas possible en μ-calcul standard d'alternance 1. Bien que la complexité des algorithmes présentés aux sections 4.2 et 4.3 soit difficile à analyser dans le cas général, nous espérons de bonnes performances en pratique (pour les formules XTL ne contenant pas de variables typées, la complexité rejoint celle des algorithmes linéaires du μ-calcul standard d'alternance 1). Pour l'algorithme d'évaluation locale, les outils de l'environnement CADP (en particulier, OPEN/CÆSAR) devraient être étendus afin de permettre la manipulation des valeurs contenues dans les états et les actions du programme à vérifier durant la génération "à la volée" de son modèle STE.

D'autre part, il serait souhaitable d'étendre le langage XTL avec de nouvelles constructions. En effet, certaines classes de propriétés avec valeurs (en particulier, les propriétés du passé) nécessitent la manipulation d'ensembles ou de listes d'objets définis dans le programme à vérifier (par exemple, l'ensemble des messages émis depuis l'état initial jusqu'à l'état courant du programme). Ces types auxiliaires, ainsi que leurs opérations associées, peuvent ne pas être définis dans le programme à vérifier, ce qui rend nécessaire l'introduction en XTL d'un mécanisme de définition de types. Ceci pourrait être réalisé, par exemple, en rajoutant des types définis par des opérateurs constructeurs, de manière similaire aux types abstraits algébriques ou aux types des langages fonctionnels comme ML (ceci est facilité par la présence de l'expression "**case**" permettant d'effectuer le filtrage des valeurs sous forme normale).

L'expérience a montré qu'en pratique il faut fournir à l'utilisateur un diagnostic expliquant la valeur de vérité d'une formule en termes de séquences (ou arborescences) du modèle STE. Un problème intéressant serait donc de développer des algorithmes d'évaluation des formules XTL capables de produire un diagnostic, ce qui constitue un problème en soi [Ras90]. Ceci a d'ores et déjà été entrepris, pour l'évaluateur XTL version 1.1, par Charles Pecheur, qui a implémenté une nouvelle bibliothèque d'opérateurs ACTL capables de générer des traces de diagnostic expliquant la valeur de vérité des formules [Pec98]. En ce qui concerne le langage XTL version 2.0, ses mécanismes (notamment, la possibilité de capturer l'état courant dans une formule et de le passer en paramètre aux opérateurs de point fixe) semblent adaptés à la génération de diagnostics. Par exemple, il serait possible d'accumuler, dans un paramètre auxiliaire des opérateurs de point fixe, le chemin (mémorisé comme une liste de transitions) parcouru depuis l'état initial du STE jusqu'à l'état courant sur lequel la formule est évaluée. Suivant la structure de la formule et sa valeur de vérité, l'algorithme d'évaluation sous-jacent peut imprimer (une partie de) ce chemin sur le fichier de sortie. Une autre facilité utile en pratique est d'afficher récursivement, pour une formule φ donnée, la valeur de vérité de ses sous-formules : ceci permet de détecter les formules trivialement vraies (*false positives*), qui peuvent induire l'utilisateur en erreur.

Finalement, l'implémentation d'autres logiques temporelles en XTL peut être envisagée. Par exemple, les logiques temporelles (contenant des fragments) linéaires, comme PTL [GPSS80], CTL* [EH86] ou ACTL* [NV90] peuvent être traduites, moyennant un passage intermédiaire par des automates de Büchi, vers des formules du μ-calcul standard [Dam94a]. Des traductions similaires pourraient être développées pour des logiques temporelles avec valeurs, comme la logique modale (complète) dédiée au langage μCRL [GvV94], probablement en passant par des automates de Büchi étendus avec des valeurs, qui seront ensuite traduits vers des formules de point fixe paramétrées (ceci pourrait constituer aussi une passerelle vers la vérification basée sur des automates observateurs, à travers leurs formules caractéristiques [IS94]). En ce sens, l'environnement XTL constitue une plate-forme logicielle ouverte, permettant la définition et l'évaluation efficace de nouveaux opérateurs temporels.

Annexe A

Sémantique statique

Nous présentons ici une définition de la sémantique statique du langage XTL, décrite au moyen de grammaires attribuées. Il s'agit d'une annexe technique, qui peut être évitée en première lecture, mais qui illustre bien les difficultés posées par le caractère original de XTL, qui doit offrir, d'une part, une syntaxe des formules temporelles proche des notations mathématiques couramment employées et, d'autre part, une syntaxe des expressions proche des langages de programmation fonctionnels.

Cette annexe est organisée de la manière suivante. La section A.1 précise les notations utilisées. La section A.2 définit la grammaire utilisée comme base pour les différentes phases d'analyse : la liaison des types, la liaison des variables simples, la liaison des variables propositionnelles, la liaison des fonctions et le typage des expressions et des formules, qui sont respectivement présentées aux sections A.3, A.4, A.5, A.6 et A.7. Finalement, la section A.8 décrit plusieurs vérifications statiques complémentaires requises par la sémantique des formules et des expressions régulières XTL.

A.1 Préliminaires

Nous avons choisi de décrire les phases d'analyse au moyen de grammaires attribuées [Knu68] qui, à la différence d'autres formalismes comme les règles d'inférence [Plo81], présentent l'avantage d'être facilement implémentables. Nous utilisons les notations suivantes :

- les attributs synthétisés sont précédés par le symbole $\uparrow$ et les attributs hérités sont précédés par le symbole $\downarrow$; les attributs locaux *attr* associés aux symboles N sont dénotés par $attr(N)$;

- les actions sémantiques qui calculent les attributs associés aux symboles N sont décrites dans les parties droites des règles syntaxiques associées à N ;

- le langage des actions sémantiques contient des affectations, des séquences d'actions, des constructions conditionnelles "**si-alors-sinon**" et des constructions répétitives "**pour_tous**" ;

- l'action sémantique spéciale *error*( ) est utilisée pour arrêter l'analyse de la sémantique statique et imprimer un message d'erreur.

Par souci de concision, les grammaires attribuées ne contiennent que les règles syntaxiques des symboles non-terminaux qui ont des actions sémantiques associées non vides et/ou qui propagent des attributs synthétisés ou hérités.

Remarque A-1
Les actions sémantiques effectuées dans les différentes phases de liaison utilisent des *environnements syntaxiques* afin de modéliser la structure des blocs et la visibilité des différents objets (variables, fonctions, ...). A la différence des environnements utilisés pour définir la sémantique dénotationnelle des formules et des expressions XTL (voir le chapitre 3 et l'annexe B), les environnements syntaxiques contiennent uniquement des informations statiques (identificateurs uniques, types, ...). ■

A.2 Grammaire semi-concrète

Nous présentons ici une grammaire du langage XTL qui sera utilisée par la suite comme base pour les différentes phases d'analyse de la sémantique statique. Le terme "semi-concrète" utilisé pour désigner cette grammaire est justifié par les deux observations suivantes.

Premièrement, cette grammaire a été obtenue à partir d'une grammaire concrète[18] du langage XTL en faisant abstraction autant que possible des détails syntaxiques :

- Seulement un sous-ensemble significatif de symboles non-terminaux a été gardé (entre autres, tous les symboles non-terminaux auxiliaires, utilisés pour modéliser les priorités et associativités des différents opérateurs, ont été éliminés) ;

- Les déclarations multiples de variables (utilisées dans les en-têtes des définitions de fonctions, dans les expressions "**let**", etc.) ont été "mises à plat" : chaque déclaration multiple de la forme "x_1**,** ... **,**x_n**:**T" a été transformée en "x_1**:**T**,** ... **,**x_n**:**T" ;

- Les appels de l'opérateur d'impression "**print**" avec plusieurs arguments (voir la section 2.3) ont été expansés vers des appels de "**print**" avec un seul argument, séparés par des opérateurs de séquencement "**;**" : chaque occurrence de "**print(**E_1**,** ... **,**E_n**)**" a été remplacée par "**print(**E_1**) ;** $\cdots$ **; print(**E_n**)**".

Deuxièmement, nous ne considérons pas cette grammaire comme "abstraite", car elle n'est pas suffisamment précise pour servir de base à une définition propre de la sémantique dénotationnelle :

- Les formules φ sur états ne peuvent pas être séparées syntaxiquement des expressions E, car certaines constructions comme "**let**", "**if**", "**case**", etc. sont communes aux formules aussi bien qu'aux expressions. En outre, certaines expressions booléennes contenues dans les formules peuvent dénoter des prédicats de base (voir la section 2.10.1) ;

- Les occurrences d'utilisation des variables simples x sont syntaxiquement identiques aux appels de variables propositionnelles Y ou de fonctions F sans arguments. Dans la grammaire semi-concrète, ces occurrences sont désignées par le même symbole terminal I (identificateur) ;

- Les appels des variables propositionnelles sont syntaxiquement identiques aux appels de fonctions ayant le même nombre d'arguments. Dans la grammaire semi-concrète, ces occurrences sont désignées par la même construction "I **(**E_1**,** ... **,**E_n**)**".

Ces ambiguïtés sont dues au caractère original du langage XTL, qui doit contenir des constructions apparentées aux logiques temporelles et aux langages fonctionnels, tout en restant compatible avec les langages utilisés pour décrire les programmes source à vérifier (par exemple LOTOS). Au fur et à mesure des différentes phases d'analyse sémantique présentées dans cette annexe, ces ambiguïtés seront résolues afin d'aboutir à la grammaire abstraite (voir la section 2.1) utilisée comme base pour la sémantique dénotationnelle de XTL (voir le chapitre 3 et l'annexe B).

[18] Il s'agit de la grammaire $LALR(1)$ fournie à l'outil SYNTAX.

A.2.1 Notations

La grammaire semi-concrète de XTL utilise les symboles terminaux et non-terminaux définis à la section 2.1.1, ainsi que d'autres symboles auxiliaires, donnés dans la table A.1.

TERMINAL	SIGNIFICATION
I	identificateur

NON-TERMINAL	SIGNIFICATION
const_bool_op	opérateur booléen constant
unary_bool_op	opérateur booléen unaire
binary_bool_op	opérateur booléen binaire
quantifier	quantificateur
σ	opérateur de point fixe

Table A.1: Symboles terminaux et non-terminaux auxiliaires

Remarque A-2
Pour certains symboles terminaux, les attributs suivants sont calculés lors de l'analyse syntaxique :

- l'attribut *type*, dénotant le type d'une construction XTL, peut être calculé pour les constantes littérales K à partir de leur syntaxe concrète (définie informellement à la section 2.3.1) : il peut prendre les valeurs `integer`, `real`, `character` ou `string` ;
- l'attribut *internal_id*, caractérisant les identificateurs internes (voir la section 2.2), est positionné à faux pour les identificateurs T, F et I entourés par des caractères '' et à vrai pour les autres identificateurs.

Ces attributs seront utilisés dans les phases ultérieures d'analyse sémantique. ■

Remarque A-3
La grammaire semi-concrète ne contient pas les constructions "**formula**" et "**library**", celles-ci étant traitées à la phase d'expansion des définitions de formules et d'inclusion des bibliothèques (voir les sections 2.13 et 2.14), qui est effectuée avant l'analyse syntaxique. ■

A.2.2 Règles syntaxiques

Types résultat

$$\begin{array}{rcl} RT & ::= & T \\ & | & (T_0, \ldots, T_n) \end{array}$$

Filtres

$$\begin{array}{rcl} P & ::= & x\mathbf{:}T \\ & | & (x_0\mathbf{:}T_0, \ldots, x_n\mathbf{:}T_n) \\ & | & \mathbf{any}\ T \\ & | & P_1\ \mathbf{of}\ RT \\ & | & C\ (P_1, \ldots, P_n) \end{array}$$

Offres

$$
\begin{array}{rcl}
O & ::= & \textbf{any} \\
& | & \textbf{!}\ E \\
& | & \textbf{?}\ P_0 \mid \ldots \mid P_n
\end{array}
$$

Opérateurs

$$
\begin{array}{rcl}
\textit{const\_bool\_op} & ::= & \textbf{true} \\
& | & \textbf{false} \\
\\
\textit{unary\_bool\_op} & ::= & \textbf{not} \\
\\
\textit{binary\_bool\_op} & ::= & \textbf{or} \\
& | & \textbf{and} \\
& | & \textbf{implies} \\
& | & \textbf{iff} \\
& | & \textbf{xor} \\
\\
\textit{quantifier} & ::= & \textbf{exists} \\
& | & \textbf{forall} \\
\\
\sigma & ::= & \textbf{mu} \\
& | & \textbf{nu}
\end{array}
$$

Formules sur actions

$$
\begin{array}{rcl}
\alpha & ::= & (G_0|O_0)\ O_1 \ldots O_m\ [\textbf{...}]\ O_{m+1} \ldots O_{m+n}\ [\textbf{where}\ E] \\
& | & \textit{const\_bool\_op} \\
& | & \textit{unary\_bool\_op}\ \alpha_1 \\
& | & \alpha_1\ \textit{binary\_bool\_op}\ \alpha_2
\end{array}
$$

Expressions régulières

$$
\begin{array}{rcl}
R & ::= & \alpha \\
& | & R_1\ \textbf{.}\ R_2 \\
& | & R_1 \mid R_2 \\
& | & R_1^{\boldsymbol{*}} \\
& | & R_1^{\boldsymbol{+}}
\end{array}
$$

Expressions

$$
\begin{array}{rcl}
E & ::= & \mathit{const\_bool\_op} \\
 & | & \mathit{unary\_bool\_op}\ E_1 \\
 & | & E_1\ \mathit{binary\_bool\_op}\ E_2 \\
 & | & \langle R \rangle\ E_1 \\
 & | & [R]\ E_1 \\
 & | & \mathbf{@}\ \mathbf{(}R\mathbf{)} \\
 & | & \mathit{quantifier}\ x_0\mathbf{:}T_0\ [\mathbf{among}\ E_0]\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n\ [\mathbf{among}\ E_n]\ \mathbf{in}\ E' \\
 & | & \sigma Y\ \mathbf{(}x_1\mathbf{:}T_1\mathbf{:=}E_1\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n\mathbf{:=}E_n\mathbf{)}\ \mathbf{.}\ E' \\
 & | & [E_1] \models E_2 \\
 & | & [E_1] \models \mathbf{action}\ \alpha \\
 & | & [[E_1]] \\
 & | & [[\mathbf{action}\ \alpha]] \\
 & | & K \\
 & | & E_1\ F\ E_2 \\
 & | & I \\
 & | & I\ \mathbf{(}E_1\mathbf{,}\ldots\mathbf{,}E_n\mathbf{)} \\
 & | & \mathbf{current} \\
 & | & E_1\ \mathbf{.}\ x \\
 & | & E_1\ \mathbf{of}\ RT \\
 & | & \mathbf{nop} \\
 & | & E_1\ \mathbf{;}\ E_2 \\
 & | & \mathbf{print}\ \mathbf{(}E_1\mathbf{)} \\
 & | & \mathbf{(}E_0\mathbf{,}\ldots\mathbf{,}E_n\mathbf{)} \\
 & | & \mathbf{let}\ x_0\mathbf{:}T_0\mathbf{:=}E_0\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n\mathbf{:=}E_n\ \mathbf{in} \\
 & & \quad E' \\
 & & \mathbf{endlet} \\
 & | & \mathbf{let}\ \mathbf{(}x_0^0\mathbf{:}T_0^0\mathbf{,}\ldots\mathbf{,}x_0^{n_0}\mathbf{:}T_0^{n_0}\mathbf{)}\mathbf{:=}E_0\mathbf{,}\ldots\mathbf{,}\mathbf{(}x_m^0\mathbf{:}T_m^0\mathbf{,}\ldots\mathbf{,}x_m^{n_m}\mathbf{:}T_m^{n_m}\mathbf{)}\mathbf{:=}E_m\ \mathbf{in} \\
 & & \quad E' \\
 & & \mathbf{endlet} \\
 & | & \mathbf{assert}\ E_0\mathbf{,}\ldots\mathbf{,}E_n\ \mathbf{in} \\
 & & \quad E' \\
 & & \mathbf{endassert} \\
 & | & \mathbf{if}\ E_0\ \mathbf{then}\ E'_0 \\
 & & \quad \mathbf{elsif}\ E_1\ \mathbf{then}\ E'_1 \\
 & & \quad \ldots \\
 & & \quad \mathbf{elsif}\ E_n\ \mathbf{then}\ E'_n \\
 & & \ \big[\mathbf{else}\ E'_{n+1}\big] \\
 & & \mathbf{endif}
\end{array}
$$

$$
\begin{array}{lll}
 & \mid & \textbf{case } E_0 \textbf{ in} \\
 & & \quad P_1^0 \mid \ldots \mid P_1^{n_1} \text{ [}\textbf{where } E_1\text{]} \ \textbf{->} \ E'_1 \\
 & & \quad \ldots \\
 & & \ \mid P_m^0 \mid \ldots \mid P_m^{n_m} \text{ [}\textbf{where } E_m\text{]} \ \textbf{->} \ E'_m \\
 & & \text{[}\mid \textbf{otherwise} \ \textbf{->} \ E'_{m+1}\text{]} \\
 & & \textbf{endcase} \\
 & \mid & \textbf{case action } E_0 \textbf{ in} \\
 & & \quad \alpha_1 \ \text{[}\textbf{where } E_1\text{]} \ \textbf{->} \ E'_1 \\
 & & \quad \ldots \\
 & & \ \mid \alpha_m \ \text{[}\textbf{where } E_m\text{]} \ \textbf{->} \ E'_m \\
 & & \text{[}\mid \textbf{otherwise} \ \textbf{->} \ E'_{m+1}\text{]} \\
 & & \textbf{endcase} \\
 & \mid & \textbf{loop } (x_0\textbf{:}T_0\textbf{:=}E_0, \ldots, x_n\textbf{:}T_n\textbf{:=}E_n) \textbf{ : } RT \textbf{ in} \\
 & & \quad E' \\
 & & \textbf{endloop} \\
 & \mid & \textbf{continue } (E_0, \ldots, E_n) \\
 & \mid & \textbf{for } x'_0\textbf{:}T'_0\text{[}\textbf{among } E'_0\text{]}, \ldots, x'_m\textbf{:}T'_m\text{[}\textbf{among } E'_m\text{]} \\
 & & \quad \text{[}\textbf{var } x_0\textbf{:}T_0\textbf{:=}E_0, \ldots, x_n\textbf{:}T_n\textbf{:=}E_n\text{]} \\
 & & \quad \text{[}\textbf{where } E''_1\text{]} \\
 & & \quad \text{[}\textbf{while } E''_2\text{]} \\
 & & \quad \ \textbf{in } E''_3 \\
 & & \quad \text{[}\textbf{result } E''_4\text{]} \\
 & & \textbf{endfor} \\
 & \mid & \{E_1, \ldots, E_n\} \\
 & \mid & \{E_1 \textbf{ ... } E_2\} \\
 & \mid & \{\ F \textbf{ on } x_0\textbf{:}T_0 \text{ [}\textbf{among } E_0\text{]}, \ldots, x_n\textbf{:}T_n \text{ [}\textbf{among } E_n\text{] [}\textbf{where } E'_1\text{]} \ \} \ E'_2 \\
 & \mid & \{\ x\textbf{:}T \text{ [}\textbf{among } E_1\text{] } \textbf{where } E_2 \ \}
\end{array}
$$

Définitions de fonctions

$$
\begin{array}{lll}
D & ::= & \text{[}\textbf{local}\text{]}\,\textbf{function } F \ (x_1\textbf{:}T_1, \ldots, x_n\textbf{:}T_n) \textbf{ : } RT \textbf{ is} \\
 & & \quad E \\
 & & \textbf{endfunc} \\
 & \mid & \text{[}\textbf{local}\text{]}\,\textbf{function } \_\ F \ \_ \ (x_1\textbf{:}T_1, x_2\textbf{:}T_2) \textbf{ : } RT \textbf{ is} \\
 & & \quad E \\
 & & \textbf{endfunc}
\end{array}
$$

Programme

$$
\begin{array}{lll}
PG & ::= & [D_0 \ldots D_m] \\
 & & \quad E \\
 & & \text{[}\textbf{where } D_{m+1} \ldots D_{m+p}\text{]}
\end{array}
$$

A.3 Liaison des types

Du point de vue de leur nommage, les types utilisés dans le langage XTL peuvent être groupés en deux classes :

Types nommés (symbole terminal T) : ce sont les types XTL prédéfinis, ainsi que les types BCG définis dans le programme source à vérifier (voir la section 2.3) ;

Types anonymes (symbole non-terminal RT) : ce sont des types tuples, n'ayant pas de nom associé, construits à partir de types nommés (voir la section 2.3.3).

La version actuelle du langage XTL ne permet pas de définir de nouveaux types ; par conséquent, un programme XTL ne peut contenir que des occurrences d'utilisation de types. Les types XTL prédéfinis et les types BCG sont visibles partout, étant considérés comme définis dans un bloc fictif incluant tout le programme.

Le but de la liaison des types est d'associer un type unique à chaque occurrence de symbole dénotant un type (symboles T et RT) dans un programme XTL.

A.3.1 Attributs

Les types uniques associés aux symboles T et RT sont représentés comme *numéros uniques de type* appartenant au domaine **NumType** et/ou comme *expressions de type* appartenant au domaine **ExpType**. Les expressions de type $t \in$ **ExpType** sont définies inductivement comme suit :

$$\begin{array}{rcl} t & \stackrel{\mathrm{d}}{=} & nt \\ & | & (nt_0, \ldots, nt_n) \end{array}$$

où $nt_i \in$ **NumType** sont des numéros uniques de type. L'équivalence des expressions de type est définie de manière structurelle, composante par composante.

Remarque A-4
Un type tuple ayant une seule composante est assimilé au type de sa composante ; par conséquent, une expression de type (nt) est équivalente à nt. ■

La liaison des types consiste à calculer l'attribut local $type \in$ **ExpType** associé aux occurrences des symboles T et RT, qui identifie de manière unique le type respectif.

Cette phase d'analyse est effectuée au moyen d'*environnements de types* appartenant au domaine **EnvType** $\stackrel{\mathrm{d}}{=}$ **Idf** $\rightarrow$ **NumType**. Un environnement de types $\mathcal{T} \in$ **EnvType** est une fonction partielle associant aux identificateurs T des numéros uniques de type.

Les types XTL prédéfinis et les types BCG sont contenus respectivement dans les environnements de types $\mathcal{T}_{xtl}$ et $\mathcal{T}_{bcg}$. Ces environnements sont initialisés avant de commencer la liaison des types et ne sont plus modifiés ensuite : il n'est donc pas nécessaire de les propager comme attributs hérités dans les règles syntaxiques.

Remarque A-5
Les images de $\mathcal{T}_{xtl}$ et de $\mathcal{T}_{bcg}$ sont disjointes : un type XTL prédéfini et un type BCG ayant le même nom auront des numéros uniques différents associés dans les deux environnements. ■

A.3.2 Actions sémantiques

La liaison des types est effectuée au moyen de l'action sémantique use_T : **Idf** $\to$ **Void**, qui positionne l'attribut local *type* associé à une occurrence d'identificateur de type :

$$use_T(T) \stackrel{\mathrm{d}}{=} \left\{ \begin{array}{l} \textbf{si } \mathit{internal\_id}(T) \ \wedge \ T \in \mathit{supp}(\mathcal{T}_{xtl}) \textbf{ alors} \\ \quad \mathit{type}(T) := \mathcal{T}_{xtl}(T) \\ \textbf{sinon\_si } T \in \mathit{supp}(\mathcal{T}_{bcg}) \textbf{ alors} \\ \quad \mathit{type}(T) := \mathcal{T}_{bcg}(T) \\ \textbf{sinon} \\ \quad \mathit{error}(\texttt{"undefined type"}) \\ \textbf{fin\_si} \end{array} \right.$$

Nous utilisons aussi la notation abrégée suivante :

$$use_T(T_1, \ldots, T_n) \stackrel{\mathrm{d}}{=} \left\{ \begin{array}{l} use_T(T_1) \ ; \\ \ldots \\ use_T(T_n) \end{array} \right.$$

Remarque A-6
L'action sémantique $use_T(\ )$ respecte les conventions de liaison mentionnées à la section 2.3.2 : en cas de conflit entre un type XTL et un type BCG (par exemple, un type nommé `integer` défini dans le programme source à vérifier), la priorité est donnée par défaut au type XTL ; cependant, l'utilisateur peut forcer la liaison avec le type BCG en utilisant pour celui-ci un identificateur externe (entouré par des caractères '‘’'). ■

A.3.3 Grammaire attribuée

Types résultat

$$\begin{array}{lll} RT & ::= & T \\ & & \quad \{ \ use_T(T) \\ & | & \mathbf{(}T_0\mathbf{,} \ldots \mathbf{,}T_n\mathbf{)} \\ & & \quad \left\{ \begin{array}{l} use_T(T_0, \ldots, T_n); \\ \mathit{type}(RT) := (\mathit{type}(T_0), \ldots, \mathit{type}(T_n)) \end{array} \right. \end{array}$$

Filtres

$$\begin{array}{lll} P & ::= & x\mathbf{:}T \\ & & \quad \{ \ use_T(T) \\ & | & \mathbf{(}x_0\mathbf{:}T_0\mathbf{,} \ldots \mathbf{,}x_n\mathbf{:}T_n\mathbf{)} \\ & & \quad \{ \ use_T(T_0, \ldots, T_n) \\ & | & \textbf{any } T \\ & & \quad \{ \ use_T(T) \end{array}$$

Expressions

$$
\begin{array}{lll}
E & ::= & \textit{quantifier}\ x_0\textbf{:}T_0\ [\textbf{among}\ E_0]\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\ [\textbf{among}\ E_n]\ \textbf{in}\ E' \\
 & & \quad \{\ \mathit{use}_T(T_0,\ldots,T_n) \\
 & | & \sigma Y\ \textbf{(}x_1\textbf{:}T_1\textbf{:=}E_1\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n\textbf{)}\ \textbf{.}\ E' \\
 & & \quad \{\ \mathit{use}_T(T_1,\ldots,T_n) \\
 & | & \textbf{let}\ x_0\textbf{:}T_0\textbf{:=}E_0\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n\ \textbf{in} \\
 & & \quad E' \\
 & & \textbf{endlet} \\
 & & \quad \{\ \mathit{use}_T(T_0,\ldots,T_n) \\
 & | & \textbf{let}\ \textbf{(}x_0^0\textbf{:}T_0^0\textbf{,}\ldots\textbf{,}x_0^{n_0}\textbf{:}T_0^{n_0}\textbf{):=}E_0\textbf{,}\ldots\textbf{,(}x_m^0\textbf{:}T_m^0\textbf{,}\ldots\textbf{,}x_m^{n_m}\textbf{:}T_m^{n_m}\textbf{):=}E_m\ \textbf{in} \\
 & & \quad E' \\
 & & \textbf{endlet} \\
 & & \quad \{\ \mathit{use}_T(T_0^0,\ldots,T_0^{n_0},\ldots,T_m^0,\ldots,T_m^{n_m}) \\
 & | & \textbf{loop}\ \textbf{(}x_0\textbf{:}T_0\textbf{:=}E_0\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n\textbf{)}\ \textbf{:}\ RT\ \textbf{in} \\
 & & \quad E' \\
 & & \textbf{endloop} \\
 & & \quad \{\ \mathit{use}_T(T_0,\ldots,T_n) \\
 & | & \textbf{for}\ x_0'\textbf{:}T_0'[\textbf{among}\ E_0']\textbf{,}\ldots\textbf{,}x_m'\textbf{:}T_m'[\textbf{among}\ E_m'] \\
 & & \quad [\textbf{var}\ x_0\textbf{:}T_0\textbf{:=}E_0\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n] \\
 & & \quad [\textbf{where}\ E_1''] \\
 & & \quad [\textbf{while}\ \ E_2''] \\
 & & \quad\ \textbf{in}\ E_3'' \\
 & & \quad [\textbf{result}\ E_4''] \\
 & & \textbf{endfor} \\
 & & \quad \{\ \mathit{use}_T(T_0',\ldots,T_m',T_0,\ldots,T_n) \\
 & | & \boldsymbol{\{}\ F\ \textbf{on}\ x_0\textbf{:}T_0\ [\textbf{among}\ E_0]\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\ [\textbf{among}\ E_n]\ [\textbf{where}\ E_1']\ \boldsymbol{\}}\ E_2' \\
 & & \quad \{\ \mathit{use}_T(T_0,\ldots,T_n) \\
 & | & \boldsymbol{\{}\ x\textbf{:}T\ [\textbf{among}\ E_1]\ \textbf{where}\ E_2\ \boldsymbol{\}} \\
 & & \quad \{\ \mathit{use}_T(T)
\end{array}
$$

Définitions de fonctions

$$
\begin{array}{lll}
D & ::= & [\textbf{local}]\,\textbf{function}\ F\ \textbf{(}x_1\textbf{:}T_1\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{)}\ \textbf{:}\ RT\ \textbf{is} \\
 & & \quad E \\
 & & \textbf{endfunc} \\
 & & \quad \{\ \mathit{use}_T(T_1,\ldots,T_n) \\
 & | & [\textbf{local}]\,\textbf{function}\ \_\ F\ \_\ \textbf{(}x_1\textbf{:}T_1\textbf{,}x_2\textbf{:}T_2\textbf{)}\ \textbf{:}\ RT\ \textbf{is} \\
 & & \quad E \\
 & & \textbf{endfunc} \\
 & & \quad \{\ \mathit{use}_T(T_1,T_2)
\end{array}
$$

A.4 Liaison des variables simples

Les variables simples (symbole terminal x) peuvent être définies, initialisées et utilisées par diverses constructions XTL. Par la suite, sauf en cas d'ambiguïté avec les variables propositionnelles, nous emploierons le terme "variable" pour désigner les variables simples.

La liaison des variables a deux objectifs : (1) identifier de manière unique chaque variable contenue dans un programme XTL et (2) associer chaque occurrence d'identificateur susceptible de dénoter l'utilisation d'une variable avec une occurrence de définition de la variable respective.

A.4.1 Attributs

Les informations statiques associées aux variables sont représentées par des *profils de variables* appartenant au domaine **ProfDVar** $\stackrel{\mathrm{d}}{=}$ **NumDVar** $\times$ **ExpType**, où **NumDVar** est le domaine des numéros uniques de variables. Un profil de variable $p \in$ **ProfDVar** est composé de deux champs :

- *p.num* $\in$ **NumDVar** est le numéro unique de la variable ;
- *p.type* $\in$ **ExpType** représente le type de la variable.

La liaison des variables est effectuée au moyen d'*environnements de variables* appartenant au domaine **EnvDVar** $\stackrel{\mathrm{d}}{=}$ **Idf** $\to$ **ProfDVar**. Un environnement de variables $\mathcal{X} \in$ **EnvDVar** est une fonction partielle associant des profils de variables aux identificateurs de variables.

Remarque A-7
Un programme XTL ne contient pas de variables de type tuple (voir la section 2.3.3). Par conséquent, les champs *p.type* des profils de variables sont (équivalents à) des numéros uniques de type. ■

La table A.2 décrit les attributs gérés pendant la liaison des variables.

CLASSE D'ATTRIBUTS	NOM	SIGNIFICATION	SYMBOLES D'ATTACHEMENT
hérités	$\mathcal{X} \in$ **EnvDVar**	environnement de variables visibles	P, O, α, R, E
	$\mathcal{X}_{loc} \in$ **EnvDVar**	environnement de variables locales	P, O, α
synthétisés	$\mathcal{V}_{loc} \in 2^{\mathbf{Idf} \times \mathbf{ExpType}}$	ensemble de déclarations de variables locales	P, O, α
	$\mathcal{X}_{tt} \in$ **EnvDVar**	environnement de variables exportées en cas de succès	α, R
	$\mathcal{X}_{ff} \in$ **EnvDVar**	environnement de variables exportées en cas d'échec	α
locaux	*profile* $\in$ **ProfDVar**	profil unique de variable	x, I

Table A.2: Attributs gérés pendant la liaison des variables simples

Les variables BCG provenant du modèle STE sont contenues dans un environnement $\mathcal{X}_{bcg}$. Cet environnement est initialisé avant de commencer la liaison des variables et n'est plus modifié ensuite : il n'est donc pas nécessaire de le transmettre comme attribut hérité dans les règles syntaxiques.

A la fin de cette phase de liaison, chaque occurrence d'identificateur de variable x dans le programme XTL aura associé un profil unique de variable *profile*(x).

A.4.2 Actions sémantiques

La liaison des variables simples est effectuée au moyen des actions sémantiques suivantes :

- def_X : **Idf** × **ExpType** → **EnvDVar**, qui renvoie l'environnement associant à un identificateur de variable un profil contenant un numéro unique de variable :

$$def_X(x,t) \stackrel{\mathrm{d}}{=} [(uniq_X(\,),t)/x]$$

L'action auxiliaire $uniq_X$:→ **NumDVar** renvoie à chaque appel un numéro unique de variable.

Nous utilisons aussi la notation abrégée suivante :

$$def_X(x_1,t_1,\ldots,x_n,t_n) \stackrel{\mathrm{d}}{=} def_X(x_1,t_1) \oplus \cdots \oplus def_X(x_n,t_n)$$

- use_X : **Idf** × **EnvDVar** → **Void**, qui positionne, en utilisant un environnement de variables, l'attribut local *profile* associé à une occurrence d'identificateur de variable :

$$use_X(x,\mathcal{X}) \stackrel{\mathrm{d}}{=} \left\{ \begin{array}{l} \textbf{si } x \in supp(\mathcal{X}) \textbf{ alors} \\ \quad profile(x) := \mathcal{X}(x) \\ \textbf{fin\_si} \end{array} \right.$$

Nous utilisons aussi la notation abrégée suivante :

$$use_X(x_1,\ldots,x_n,\mathcal{X}) \stackrel{\mathrm{d}}{=} \left\{ \begin{array}{l} use_X(x_1,\mathcal{X}) \\ \ldots \\ use_X(x_n,\mathcal{X}) \end{array} \right.$$

Remarque A-8
Lors de la liaison d'un identificateur I susceptible de dénoter une occurrence d'utilisation de variable, on ne signale pas d'erreur si I n'est pas défini dans l'environnement $\mathcal{X}$ courant. En effet, I pourrait dénoter tout aussi bien un appel de variable propositionnelle ou de fonction sans arguments ; ces cas seront traités aux phases de liaison respectives (voir les sections A.5 et A.6) et l'erreur sera signalée uniquement s'il n'existe aucune possibilité de liaison pour I. ■

A.4.3 Grammaire attribuée

Filtres

$$\begin{array}{lll}
P \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X}_{loc} & ::= & x\mathbf{:}T \\
& & \left\{ \begin{array}{l} \mathcal{V}_{loc} := \{(x, type(T))\}; \\ use_X(x,\mathcal{X}_{loc}) \end{array} \right. \\
& | & \mathbf{(}x_0\mathbf{:}T_0\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n\mathbf{)} \\
& & \left\{ \begin{array}{l} \mathcal{V}_{loc} := \{(x_0, type(T_0)),\ldots,(x_n, type(T_n))\}; \\ use_X(x_0,\ldots,x_n,\mathcal{X}_{loc}) \end{array} \right. \\
& | & \mathbf{any}\ T \\
& & \{\ \mathcal{V}_{loc} := \emptyset \\
& | & P_1 \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X}_{loc}\ \mathbf{of}\ RT \\
& | & C\ \mathbf{(}P_1 \uparrow \mathcal{V}_{loc1} \downarrow \mathcal{X}_{loc}\mathbf{,}\ldots\mathbf{,}P_n \uparrow \mathcal{V}_{loc\,n} \downarrow \mathcal{X}_{loc}\mathbf{)} \\
& & \{\ \mathcal{V}_{loc} := \bigcup_{i=1}^{n} \mathcal{V}_{loc\,i}
\end{array}$$

Offres

$$
\begin{array}{lcl}
O \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} & ::= & \textbf{any} \\
 & & \quad \{ \ \mathcal{V}_{loc} := \emptyset \\
 & | & \textbf{!}\ E \downarrow \mathcal{X} \\
 & & \quad \{ \ \mathcal{V}_{loc} := \emptyset \\
 & | & \textbf{?}\ P_0 \uparrow \mathcal{V}_{loc\,0} \downarrow \mathcal{X}_{loc} \ |\dots|\ P_n \uparrow \mathcal{V}_{loc\,n} \downarrow \mathcal{X}_{loc} \\
 & & \quad \{ \ \mathcal{V}_{loc} := \bigcup_{i=0}^{n} \mathcal{V}_{loc\,i}
\end{array}
$$

Formules sur actions

$$
\begin{array}{lcl}
\alpha \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} & ::= & (G_0 | O_0 \uparrow \mathcal{V}_{loc\,0} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc})\ O_1 \uparrow \mathcal{V}_{loc\,1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc}\ \dots \\
\quad \uparrow \mathcal{X}_{tt} \uparrow \mathcal{X}_{\mathit{ff}} & & O_m \uparrow \mathcal{V}_{loc\,m} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc}\ [\dots]\ O_{m+1} \uparrow \mathcal{V}_{loc\,m+1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc}\ \dots \\
 & & O_{m+n} \uparrow \mathcal{V}_{loc\,m+n} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc}\ [\textbf{where}\ E \downarrow \mathcal{X}'] \\
 & & \quad \left\{ \begin{array}{l} \mathcal{V}_{loc} := \bigcup_{i=0}^{m+n} \mathcal{V}_{loc\,i}; \\ \textbf{soit}\ \mathcal{X}_{O\,loc} := \mathcal{X}_{loc}|_{\{x \in \textbf{Idf} \mid \exists t \in \textbf{ExpType}.(x,t) \in \mathcal{V}_{loc}\}}\ \textbf{dans} \\ \quad \mathcal{X}' := \mathcal{X} \oslash \mathcal{X}_{O\,loc}; \\ \quad \mathcal{X}_{tt} := \mathcal{X}_{O\,loc} \\ \textbf{fin}; \\ \mathcal{X}_{\mathit{ff}} := [\,] \end{array} \right. \\
 & | & \textbf{true} \\
 & & \quad \left\{ \begin{array}{l} \mathcal{V}_{loc} := \emptyset; \\ \mathcal{X}_{tt} := \mathcal{X}_{\mathit{ff}} := [\,] \end{array} \right. \\
 & | & \textbf{false} \\
 & & \quad \left\{ \begin{array}{l} \mathcal{V}_{loc} := \emptyset; \\ \mathcal{X}_{tt} := \mathcal{X}_{\mathit{ff}} := [\,] \end{array} \right. \\
 & | & \textbf{not}\ \alpha_1 \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,1} \uparrow \mathcal{X}_{\mathit{ff}\,1} \\
 & & \quad \left\{ \begin{array}{l} \mathcal{X}_{tt} := \mathcal{X}_{\mathit{ff}\,1}; \\ \mathcal{X}_{\mathit{ff}} := \mathcal{X}_{tt\,1} \end{array} \right. \\
 & | & \alpha_1 \uparrow \mathcal{V}_{loc\,1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,1} \uparrow \mathcal{X}_{\mathit{ff}\,1}\ \textbf{or} \\
 & & \alpha_2 \uparrow \mathcal{V}_{loc\,2} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,2} \uparrow \mathcal{X}_{\mathit{ff}\,2} \\
 & & \quad \left\{ \begin{array}{l} \mathcal{V}_{loc} := \mathcal{V}_{loc\,1} \cup \mathcal{V}_{loc\,2}; \\ \mathcal{X}_{tt} := \mathcal{X}_{tt\,1}|_{supp(\mathcal{X}_{tt\,2})}; \\ \mathcal{X}_{\mathit{ff}} := \mathcal{X}_{\mathit{ff}\,1} \oslash \mathcal{X}_{\mathit{ff}\,2} \end{array} \right. \\
 & | & \alpha_1 \uparrow \mathcal{V}_{loc\,1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,1} \uparrow \mathcal{X}_{\mathit{ff}\,1}\ \textbf{and} \\
 & & \alpha_2 \uparrow \mathcal{V}_{loc\,2} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,2} \uparrow \mathcal{X}_{\mathit{ff}\,2} \\
 & & \quad \left\{ \begin{array}{l} \mathcal{V}_{loc} := \mathcal{V}_{loc\,1} \cup \mathcal{V}_{loc\,2}; \\ \mathcal{X}_{tt} := \mathcal{X}_{tt\,1} \oslash \mathcal{X}_{tt\,2}; \\ \mathcal{X}_{\mathit{ff}} := \mathcal{X}_{\mathit{ff}\,1}|_{supp(\mathcal{X}_{\mathit{ff}\,2})} \end{array} \right. \\
 & | & \alpha_1 \uparrow \mathcal{V}_{loc\,1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,1} \uparrow \mathcal{X}_{\mathit{ff}\,1}\ \textbf{implies} \\
 & & \alpha_2 \uparrow \mathcal{V}_{loc\,2} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,2} \uparrow \mathcal{X}_{\mathit{ff}\,2} \\
 & & \quad \left\{ \begin{array}{l} \mathcal{V}_{loc} := \mathcal{V}_{loc\,1} \cup \mathcal{V}_{loc\,2}; \\ \mathcal{X}_{tt} := \mathcal{X}_{\mathit{ff}\,1}|_{supp(\mathcal{X}_{tt\,2})}; \\ \mathcal{X}_{\mathit{ff}} := \mathcal{X}_{tt\,1} \oslash \mathcal{X}_{\mathit{ff}\,2} \end{array} \right.
\end{array}
$$

$$
\begin{array}{rcl}
 & | & \alpha_1 \uparrow \mathcal{V}_{loc\,1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,1} \uparrow \mathcal{X}_{\mathit{ff}\,1}\ \textbf{iff} \\
 & & \alpha_2 \uparrow \mathcal{V}_{loc\,2} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,2} \uparrow \mathcal{X}_{\mathit{ff}\,2} \\
 & & \left\{ \begin{array}{l} \mathcal{V}_{loc} := \mathcal{V}_{loc\,1} \cup \mathcal{V}_{loc\,2}; \\ \mathcal{X}_{tt} := \mathcal{X}_{tt\,1}|_{supp(\mathcal{X}_{\mathit{ff}\,2})} \oplus \mathcal{X}_{\mathit{ff}\,1}|_{supp(\mathcal{X}_{tt2})}; \\ \mathcal{X}_{\mathit{ff}} := \mathcal{X}_{tt\,1}|_{supp(\mathcal{X}_{tt2})} \oplus \mathcal{X}_{\mathit{ff}\,1}|_{supp(\mathcal{X}_{\mathit{ff}\,2})} \end{array} \right. \\
 & | & \alpha_1 \uparrow \mathcal{V}_{loc\,1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,1} \uparrow \mathcal{X}_{\mathit{ff}\,1}\ \textbf{xor} \\
 & & \alpha_2 \uparrow \mathcal{V}_{loc\,2} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt\,2} \uparrow \mathcal{X}_{\mathit{ff}\,2} \\
 & & \left\{ \begin{array}{l} \mathcal{V}_{loc} := \mathcal{V}_{loc\,1} \cup \mathcal{V}_{loc\,2}; \\ \mathcal{X}_{tt} := \mathcal{X}_{tt\,1}|_{supp(\mathcal{X}_{tt2})} \oplus \mathcal{X}_{\mathit{ff}\,1}|_{supp(\mathcal{X}_{\mathit{ff}\,2})}; \\ \mathcal{X}_{\mathit{ff}} := \mathcal{X}_{tt\,1}|_{supp(\mathcal{X}_{\mathit{ff}\,2})} \oplus \mathcal{X}_{\mathit{ff}\,1}|_{supp(\mathcal{X}_{tt2})} \end{array} \right.
\end{array}
$$

Expressions régulières

$$
\begin{array}{rcl}
R \downarrow \mathcal{X} \uparrow \mathcal{X}_{tt} & ::= & \alpha \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt} \uparrow \mathcal{X}_{\mathit{ff}} \\
 & & \quad \{\ \mathcal{X}_{loc} := \bigoplus \{ \mathit{def}_X(x,t) \mid (x,t) \in \mathcal{V}_{loc} \} \\
 & | & R_1 \downarrow \mathcal{X} \uparrow \mathcal{X}_{tt1}\ \textbf{.}\ R_2 \downarrow \mathcal{X}_2 \uparrow \mathcal{X}_{tt2} \\
 & & \left\{ \begin{array}{l} \mathcal{X}_2 := \mathcal{X} \oslash \mathcal{X}_{tt\,1}; \\ \mathcal{X}_{tt} := \mathcal{X}_{tt\,1} \oslash \mathcal{X}_{tt\,2} \end{array} \right. \\
 & | & R_1 \downarrow \mathcal{X} \uparrow \mathcal{X}_{tt1} \mid R_2 \downarrow \mathcal{X} \uparrow \mathcal{X}_{tt2} \\
 & & \quad \{\ \mathcal{X}_{tt} := \mathcal{X}_{tt\,1}|_{supp(\mathcal{X}_{tt2})} \\
 & | & R_1 \downarrow \mathcal{X} \uparrow {\mathcal{X}_{tt1}}^{*} \\
 & & \quad \{\ \mathcal{X}_{tt} := [\,] \\
 & | & R_1 \downarrow \mathcal{X} \uparrow {\mathcal{X}_{tt1}}^{+} \\
 & & \quad \{\ \mathcal{X}_{tt} := \mathcal{X}_{tt1}
\end{array}
$$

Expressions

$$
\begin{array}{rcl}
E \downarrow \mathcal{X} & ::= & \mathit{unary\_bool\_op}\ E_1 \downarrow \mathcal{X} \\
 & | & E_1 \downarrow \mathcal{X}\ \mathit{binary\_bool\_op}\ E_2 \downarrow \mathcal{X} \\
 & | & \langle R \downarrow \mathcal{X} \uparrow \mathcal{X}_{tt} \rangle\ E_1 \downarrow \mathcal{X}_1 \\
 & & \quad \{\ \mathcal{X}_1 := \mathcal{X} \oslash \mathcal{X}_{tt} \\
 & | & [R \downarrow \mathcal{X} \uparrow \mathcal{X}_{tt}]\ E_1 \downarrow \mathcal{X}_1 \\
 & & \quad \{\ \mathcal{X}_1 := \mathcal{X} \oslash \mathcal{X}_{tt} \\
 & | & \textbf{@}\ (R \downarrow \mathcal{X} \uparrow \mathcal{X}_{tt}) \\
 & | & \mathit{quantifier}\ x_0\textbf{:}T_0\ [\textbf{among}\ E_0 \downarrow \mathcal{X}]\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\ [\textbf{among}\ E_n \downarrow \mathcal{X}]\ \textbf{in}\ E' \downarrow \mathcal{X}' \\
 & & \left\{ \begin{array}{l} \textbf{soit}\ \mathcal{X}'' := \mathit{def}_X(x_0, \mathit{type}(T_0), \ldots, x_n, \mathit{type}(T_n))\ \textbf{dans} \\ \quad \mathit{use}_X(x_0, \ldots, x_n, \mathcal{X}''); \\ \quad \mathcal{X}' := \mathcal{X} \oslash \mathcal{X}'' \\ \textbf{fin} \end{array} \right. \\
 & | & \sigma Y\ (x_1\textbf{:}T_1\textbf{:=}E_1 \downarrow \mathcal{X}\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n \downarrow \mathcal{X})\ \textbf{.}\ E' \downarrow \mathcal{X}' \\
 & & \left\{ \begin{array}{l} \textbf{soit}\ \mathcal{X}'' := \mathit{def}_X(x_1, \mathit{type}(T_1), \ldots, x_n, \mathit{type}(T_n))\ \textbf{dans} \\ \quad \mathit{use}_X(x_1, \ldots, x_n, \mathcal{X}''); \\ \quad \mathcal{X}' := \mathcal{X} \oslash \mathcal{X}'' \\ \textbf{fin} \end{array} \right.
\end{array}
$$

$|$ $[E_1 \downarrow \mathcal{X}]$ $|=$ $E_2 \downarrow \mathcal{X}$

$|$ $[E_1 \downarrow \mathcal{X}]$ $|=$ **action** $\alpha \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt} \uparrow \mathcal{X}_{ff}$

$|$ $[[E_1 \downarrow \mathcal{X}]]$

$|$ $[[$**action** $\alpha \uparrow \mathcal{V}_{loc} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc} \uparrow \mathcal{X}_{tt} \uparrow \mathcal{X}_{ff}]]$

$|$ $E_1 \downarrow \mathcal{X}\ F\ E_2 \downarrow \mathcal{X}$

$|$ I

$\{\ use_X(I, \mathcal{X})$

$|$ I **(**$E_1 \downarrow \mathcal{X}$**,** $\ldots$ **,**$E_n \downarrow \mathcal{X}$**)**

$|$ $E_1 \downarrow \mathcal{X}$ **.** x

{ **si** $x \in supp(\mathcal{X}_{bcg})$ **alors**
  $profile(x) := \mathcal{X}_{bcg}(x)$
sinon
  $error($`"undefined BCG variable"`$)$
fin_si

$|$ $E_1 \downarrow \mathcal{X}$ **of** RT

$|$ $E_1 \downarrow \mathcal{X}$ **;** $E_2 \downarrow \mathcal{X}$

$|$ **print (**$E_1 \downarrow \mathcal{X}$**)**

$|$ **(**$E_0 \downarrow \mathcal{X}$**,** $\ldots$ **,**$E_n \downarrow \mathcal{X}$**)**

$|$ **let** x_0**:**T_0**:=**$E_0 \downarrow \mathcal{X}$**,** $\ldots$ **,**x_n**:**T_n**:=**$E_n \downarrow \mathcal{X}$ **in**
  $E' \downarrow \mathcal{X}'$
endlet

{ **soit** $\mathcal{X}'' := def_X(x_0, type(T_0), \ldots, x_n, type(T_n))$ **dans**
  $use_X(x_0, \ldots, x_n, \mathcal{X}'')$;
  $\mathcal{X}' := \mathcal{X} \oslash \mathcal{X}''$
fin

$|$ **let (**x_0^0**:**T_0^0**,** $\ldots$ **,**$x_0^{n_0}$**:**$T_0^{n_0}$**):=**$E_0 \downarrow \mathcal{X}$**,** $\ldots$ **,(**x_m^0**:**T_m^0**,** $\ldots$ **,**$x_m^{n_m}$**:**$T_m^{n_m}$**):=**$E_m \downarrow \mathcal{X}$ **in**
  $E' \downarrow \mathcal{X}'$
endlet

{ **soit** $\mathcal{X}'' := def_X(x_0^0, type(T_0^0), \ldots, x_0^{n_0}, type(T_0^{n_0}), \ldots,$
      $x_m^0, type(T_m^0), \ldots, x_m^{n_m}, type(T_m^{n_m})))$ **dans**
  $use_X(x_0^0, \ldots, x_0^{n_0}, \ldots, x_m^0, \ldots, x_m^{n_m}, \mathcal{X}'')$;
  $\mathcal{X}' := \mathcal{X} \oslash \mathcal{X}''$
fin

$|$ **assert** $E_0 \downarrow \mathcal{X}$**,** $\ldots$ **,**$E_n \downarrow \mathcal{X}$ **in**
  $E' \downarrow \mathcal{X}$
endassert

$|$ **if** $E_0 \downarrow \mathcal{X}$ **then** $E'_0 \downarrow \mathcal{X}$
  elsif $E_1 \downarrow \mathcal{X}$ **then** $E'_1 \downarrow \mathcal{X}$
  $\ldots$
  elsif $E_n \downarrow \mathcal{X}$ **then** $E'_n \downarrow \mathcal{X}$
  $[$**else** $E'_{n+1} \downarrow \mathcal{X}]$
endif

$|$ **case** $E_0 \downarrow \mathcal{X}$ **in**

$P_1^0 \uparrow \mathcal{V}_{loc}{}_1^0 \downarrow \mathcal{X}_{loc}{}_1^0 \quad | \ldots | \; P_1^{n_1} \uparrow \mathcal{V}_{loc}{}_1^{n_1} \downarrow \mathcal{X}_{loc}{}_1^{n_1}$

[**where** $E_1 \downarrow \mathcal{X}_1$] $\;$ -> $E_1' \downarrow \mathcal{X}_1'$

$\ldots$

$| \; P_m^0 \uparrow \mathcal{V}_{loc}{}_m^0 \downarrow \mathcal{X}_{loc}{}_m^0 \; | \ldots | \; P_m^{n_m} \uparrow \mathcal{V}_{loc}{}_m^{n_m} \downarrow \mathcal{X}_{loc}{}_m^{n_m}$

[**where** $E_m \downarrow \mathcal{X}_m$] -> $E_m' \downarrow \mathcal{X}_m'$

[| **otherwise** -> $E_{m+1}' \downarrow \mathcal{X}$]

endcase

$$\left\{\begin{array}{l}
\forall i \in [1,m], \\
\quad \textbf{si } \forall j,k \in [0,n_i], \mathcal{V}_{loc}{}_i^j = \mathcal{V}_{loc}{}_i^k \textbf{ alors} \\
\quad\quad \textbf{soit } \mathcal{X}' := \bigoplus \left\{ \mathit{def}_X(x,t) \mid (x,t) \in \mathcal{V}_{loc}{}_i^0 \right\} \textbf{ dans} \\
\quad\quad\quad \forall j \in [0,n_i],\; \mathcal{X}_{loc}{}_i^j := \mathcal{X}'; \\
\quad\quad\quad [\mathcal{X}_i :=]\, \mathcal{X}_i' := \mathcal{X}' \\
\quad\quad \textbf{fin} \\
\quad \textbf{sinon} \\
\quad\quad \mathit{error}(\texttt{"alternative patterns should define} \\
\quad\quad\quad\quad \texttt{identical variables"}) \\
\quad \textbf{fin_si}
\end{array}\right.$$

$|$ **case action** $E_0 \downarrow \mathcal{X}$ **in**

$\alpha_1 \uparrow \mathcal{V}_{loc\,1} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc\,1} \uparrow \mathcal{X}_{tt\,1} \uparrow \mathcal{X}_{ff\,1}$

[**where** $E_1 \downarrow \mathcal{X}_1$] $\;$ -> $E_1' \downarrow \mathcal{X}_1'$

$\ldots$

$| \; \alpha_m \uparrow \mathcal{V}_{loc\,m} \downarrow \mathcal{X} \downarrow \mathcal{X}_{loc\,m} \uparrow \mathcal{X}_{tt\,m} \uparrow \mathcal{X}_{ff\,m}$

[**where** $E_m \downarrow \mathcal{X}_m$] -> $E_m' \downarrow \mathcal{X}_m'$

[| **otherwise** -> $E_{m+1}' \downarrow \mathcal{X}$]

endcase

$$\left\{\begin{array}{l}
\forall i \in [1,m], \mathcal{X}_{loc\,i} := \bigoplus \{ \mathit{def}_X(x,t) \mid (x,t) \in \mathcal{V}_{loc\,i} \}\,; \\
\forall i \in [1,m], [\mathcal{X}_i :=]\, \mathcal{X}_i' := \mathcal{X} \oslash \mathcal{X}_{tt\,i}
\end{array}\right.$$

$|$ **for** x_0'**:**T_0'[**among** $E_0' \downarrow \mathcal{X}$]$, \ldots,$ x_m'**:**T_m'[**among** $E_m' \downarrow \mathcal{X}$]

[**var** x_0**:**T_0**:=**$E_0 \downarrow \mathcal{X}, \ldots, x_n$**:**$T_n$**:=**$E_n \downarrow \mathcal{X}$]

[**where** $E_1'' \downarrow \mathcal{X}_1''$]

[**while** $E_2'' \downarrow \mathcal{X}_2''$]

in $E_3'' \downarrow \mathcal{X}_3''$

[**result** $E_4'' \downarrow \mathcal{X}_4''$]

endfor

$$\left\{\begin{array}{l}
\textbf{soit } \mathcal{X}' := \mathit{def}_X(x_0', \mathit{type}(T_0'), \ldots, x_m', \mathit{type}(T_m')) \\
\quad\quad [\mathcal{X}'' := \mathit{def}_X(x_0, \mathit{type}(T_0), \ldots, x_n, \mathit{type}(T_n))] \\
\textbf{dans} \\
\quad \mathit{use}_X(x_0', \ldots, x_m', \mathcal{X}'); \\
\quad [\mathit{use}_X(x_0, \ldots, x_n, \mathcal{X}'')]\,; \\
\quad [\mathcal{X}_1'' :=]\,[\mathcal{X}_2'' :=]\, \mathcal{X}_3'' := (\mathcal{X} \oslash \mathcal{X}')\, [\oslash \mathcal{X}'']\,; \\
\quad \mathcal{X}_4'' := \mathcal{X}\, [\oslash \mathcal{X}''] \\
\textbf{fin}
\end{array}\right.$$

| **loop** (x_0**:**T_0**:=**$E_0 \downarrow \mathcal{X}$**,** ...**,**$x_n$**:**$T_n$**:=**$E_n \downarrow \mathcal{X}$**)** **:** RT **in**
  $E' \downarrow \mathcal{X}'$
endloop

$$\left\{\begin{array}{l}\textbf{soit } \mathcal{X}'' := \mathit{def}_X(x_0, \mathit{type}(T_0) \ldots, x_n, \mathit{type}(T_n)) \textbf{ dans}\\ \quad \mathit{use}_X(x_0, \ldots, x_n, \mathcal{X}'');\\ \quad \mathcal{X}' := \mathcal{X} \oslash \mathcal{X}''\\ \textbf{fin}\end{array}\right.$$

| **continue** **(**$E_0 \downarrow \mathcal{X}$**,** ...**,**$E_n \downarrow \mathcal{X}$**)**

| **{**$E_1 \downarrow \mathcal{X}$**,** ...**,**$E_n \downarrow \mathcal{X}$**}**

| **{**$E_1 \downarrow \mathcal{X}$ **...** $E_2 \downarrow \mathcal{X}$**}**

| **{** F **on** x_0**:**T_0 [**among** $E_0 \downarrow \mathcal{X}$]**,** ...**,**$x_n$**:**$T_n$ [**among** $E_n \downarrow \mathcal{X}$]
  [**where** $E'_1 \downarrow \mathcal{X}'_1$] **}** $E'_2 \downarrow \mathcal{X}'_2$

$$\left\{\begin{array}{l}\textbf{soit } \mathcal{X}' := \mathit{def}_X(x_0, \mathit{type}(T_0), \ldots, x_n, \mathit{type}(T_n)) \textbf{ dans}\\ \quad \mathit{use}_X(x_0, \ldots, x_n, \mathcal{X}');\\ \quad [\mathcal{X}'_1 :=]\, \mathcal{X}'_2 := \mathcal{X} \oslash \mathcal{X}'\\ \textbf{fin}\end{array}\right.$$

| **{** x**:**T [**among** $E_1 \downarrow \mathcal{X}$] **where** $E_2 \downarrow \mathcal{X}_2$ **}**

$$\left\{\begin{array}{l}\textbf{soit } \mathcal{X}' := \mathit{def}_X(x, \mathit{type}(T)) \textbf{ dans}\\ \quad \mathit{use}_X(x, \mathcal{X}');\\ \quad \mathcal{X}_2 := \mathcal{X} \oslash \mathcal{X}'\\ \textbf{fin}\end{array}\right.$$

Définitions de fonctions

D ::= [**local**] **function** F **(**x_1**:**T_1**,** ...**,**x_n**:**T_n**)** **:** RT **is**
  $E \downarrow \mathcal{X}$
endfunc

$$\left\{\begin{array}{l}\textbf{soit } \mathcal{X}' := \mathit{def}_X(x_1, \mathit{type}(T_1), \ldots, x_n, \mathit{type}(T_n)) \textbf{ dans}\\ \quad \mathit{use}_X(x_1, \ldots, x_n, \mathcal{X}');\\ \quad \mathcal{X} := \mathcal{X}'\\ \textbf{fin}\end{array}\right.$$

| [**local**] **function** _ F _ **(**x_1**:**T_1**,**x_2**:**T_2**)** **:** RT **is**
  $E \downarrow \mathcal{X}$
endfunc

$$\left\{\begin{array}{l}\textbf{soit } \mathcal{X}' := \mathit{def}_X(x_1, \mathit{type}(T_1), x_2, \mathit{type}(T_2)) \textbf{ dans}\\ \quad \mathit{use}_X(x_1, x_2, \mathcal{X}');\\ \quad \mathcal{X} := \mathcal{X}'\\ \textbf{fin}\end{array}\right.$$

Programme

PG ::= [$D_0 \ldots D_m$]
  $E \downarrow \mathcal{X}$
[**where** $D_{m+1} \ldots D_{m+p}$]
  $\{\ \mathcal{X} := [\,]$

A.4.4 Discussion

Durant la liaison des variables simples, les trois propriétés ci-dessous sont invariantes.

1. Pour toute règle syntaxique et attributs courants $\uparrow \mathcal{V}_{loc}$ et $\downarrow \mathcal{X}_{loc}$:
$$supp(\mathcal{X}_{loc}) = \{x \in \mathbf{Idf} | \exists t \in \mathbf{ExpType}.(x,t) \in \mathcal{V}_{loc}\}\,.$$
2. Pour toute formule α avec les attributs $\uparrow \mathcal{X}_{tt}$ et $\uparrow \mathcal{X}_{ff}$:
$$supp(\mathcal{X}_{tt}) \cap supp(\mathcal{X}_{ff}) = \emptyset.$$
3. Pour toute formule α avec les attributs $\uparrow \mathcal{X}_{tt}$, $\uparrow \mathcal{X}_{ff}$ et $\downarrow \mathcal{V}_{loc}$:
$$\forall x \in supp(\mathcal{X}_{tt}).\big(x, (\mathcal{X}_{tt}(x)).type\big) \in \mathcal{V}_{loc} \ \wedge \ \forall x \in supp(\mathcal{X}_{ff}).\big(x, (\mathcal{X}_{ff}(x)).type\big) \in \mathcal{V}_{loc}.$$

Ces propriétés peuvent être facilement montrées par induction structurelle sur la grammaire.

La liaison des variables permet de lever partiellement l'ambiguïté entre les occurrences d'utilisation des variables simples et les appels de variables propositionnelles ou de fonctions sans arguments : les occurrences du symbole terminal I qui ont pu être liées comme variables simples sont remplacées dans l'arbre abstrait par des occurrences du symbole terminal x. Autrement dit, pour ces occurrences, la règle syntaxique "$E ::= I$" est remplacée par "$E ::= x$".

A.5 Liaison des variables propositionnelles

Les occurrences de définition des variables propositionnelles Y sont contenues dans les expressions E ayant la forme suivante :

$$E \quad ::= \quad \sigma Y \ (x_1{:}T_1{:=}E_1, \ldots, x_n{:}T_n{:=}E_n) \ . \ E'$$

La variable Y définie par l'opérateur σ est visible dans l'expression E'. La définition de Y "masque" les définitions d'autres variables propositionnelles, ayant le même nom et le même nombre de paramètres que Y, qui étaient éventuellement visibles dans E : celles-ci ne sont plus visibles dans E'.

Les occurrences d'utilisation (ou *appels*) des variables propositionnelles sont des expressions E ayant la syntaxe suivante :

$$\begin{array}{rcl} E & ::= & I \\ & | & I \ (E_1, \ldots, E_n) \end{array}$$

Nous utilisons le symbole terminal I à la place de Y puisqu'il est impossible de distinguer un appel de variable propositionnelle d'un appel de fonction ayant le même nom et le même nombre d'arguments (voir la section A.2). Chaque appel de variable propositionnelle doit être contenu dans une expression définissant la variable respective.

Les variables propositionnelles peuvent être surchargées : plusieurs variables Y ayant le même nom (mais ayant des paramètres différents en nombre et/ou types) peuvent être simultanément visibles. La résolution des surcharges sera faite à la phase de typage des expressions et des formules (voir la section A.7).

La liaison des variables propositionnelles a deux objectifs : (1) identifier de manière unique chaque variable propositionnelle contenue dans un programme XTL et (2) associer chaque occurrence d'identificateur susceptible de dénoter l'utilisation d'une variable propositionnelle avec un ensemble d'occurrences de définition surchargées de la variable respective.

A.5.1 Attributs

Les informations statiques associées aux variables propositionnelles sont représentées par des *profils de variables propositionnelles* appartenant au domaine $\mathbf{ProfPVar} \stackrel{\mathrm{d}}{=} \mathbf{NumPVar} \times \{\mu, \nu\} \times \mathbf{Nat} \times \mathbf{ExpType}$, où **NumPVar** est le domaine des numéros uniques de variables propositionnelles. Un profil de variable propositionnelle $p \in \mathbf{ProfPVar}$ est composé de quatre champs :

- $p.num \in \mathbf{NumPVar}$ est le numéro unique de la variable propositionnelle ;
- $p.sign \in \{\mu, \nu\}$ est le signe (plus petit ou plus grand point fixe) de la variable ;
- $p.arity \in \mathbf{Nat}$ est l'arité (nombre de paramètres) de la variable ;
- $p.arg\_types \in \mathbf{ExpType}$ représente les types des paramètres de la variable.

La liaison des variables propositionnelles utilise des *environnements de variables propositionnelles* appartenant au domaine $\mathbf{EnvPVar} \stackrel{\mathrm{d}}{=} \mathbf{Idf} \to 2^{\mathbf{NumPVar} \times \{\mu,\nu\} \times \mathbf{Nat} \times \mathbf{ExpType}}$. Un environnement $\mathcal{Y} \in \mathbf{EnvPVar}$ est une fonction partielle ensembliste associant aux identificateurs de variables propositionnelles des ensembles de profils surchargés. La table A.3 décrit les attributs gérés pendant la liaison des variables propositionnelles.

CLASSE D'ATTRIBUTS	NOM	SIGNIFICATION	SYMBOLES D'ATTACHEMENT
hérités	$\mathcal{Y} \in \mathbf{EnvPVar}$	environnement de variables visibles	E
locaux	$profiles \in 2^{\mathbf{ProfPVar}}$	ensemble de profils possibles	Y, I
	$sign \in \{\mu, \nu\}$	signe de variable	σ

Table A.3: Attributs gérés pendant la liaison des variables propositionnelles

A la fin de cette phase de liaison, chaque occurrence d'identificateur de variable propositionnelle Y dans le programme XTL aura associé un ensemble de profils surchargés possibles $profiles(Y)$.

A.5.2 Actions sémantiques

La liaison des variables propositionnelles est effectuée au moyen des actions sémantiques suivantes :

- $def_Y : \mathbf{Idf} \times \{\mu, \nu\} \times \mathbf{Nat} \times \mathbf{ExpType} \to \mathbf{EnvPVar}$, qui renvoie l'environnement associant à un identificateur de variable propositionnelle un seul profil contenant un numéro unique :

  $$def_Y(Y, sign, arity, arg\_types) \stackrel{\mathrm{d}}{=} [\{(uniq_Y(\,), sign, arity, arg\_types)\}/Y]$$

  L'action $uniq_Y(\,)$ renvoie à chaque appel un numéro unique de variable propositionnelle.

- $use_Y : \mathbf{Idf} \times \mathbf{Nat} \times \mathbf{EnvPVar} \to \mathbf{Void}$, qui positionne, en utilisant un environnement de variables propositionnelles, l'attribut local *profiles* associé à une occurrence de variable Y :

  $$use_Y(Y, arity, \mathcal{Y}) \stackrel{\mathrm{d}}{=} \left\{ \begin{array}{l} \textbf{si } Y \in supp(\mathcal{Y}) \textbf{ alors} \\ \quad profiles(Y) := \{p \in \mathcal{Y}(Y) \mid p.arity = arity\} \\ \textbf{fin\_si} \end{array} \right.$$

Remarque A-9
Lors de la liaison d'un identificateur I susceptible de dénoter un appel de variable propositionnelle, on ne signale pas d'erreur si I n'est pas défini dans l'environnement $\mathcal{Y}$ courant. En effet, I pourrait dénoter aussi bien un appel de fonction ; ce cas sera traité à la phase de liaison des fonctions (voir la section A.6) et l'erreur sera signalée uniquement s'il n'existe aucune possibilité de liaison pour I. ■

A.5.3 Grammaire attribuée

Opérateurs

$$
\begin{array}{lll}
\sigma & ::= & \textbf{mu} \\
 & & \quad \{ \; sign(\sigma) := \mu \\
 & | & \textbf{nu} \\
 & & \quad \{ \; sign(\sigma) := \nu
\end{array}
$$

Expressions

$$
\begin{array}{lll}
E \downarrow \mathcal{Y} & ::= & unary\_bool\_op \; E_1 \downarrow \mathcal{Y} \\
 & | & E_1 \downarrow \mathcal{Y} \; binary\_bool\_op \; E_2 \downarrow \mathcal{Y} \\
 & | & \langle R \rangle \; E_1 \downarrow \mathcal{Y} \\
 & | & [R] \; E_1 \downarrow \mathcal{Y} \\
 & | & quantifier \; x_0\textbf{:}T_0 \; [\textbf{among} \; E_0 \downarrow \mathcal{Y}]\textbf{,} \ldots \textbf{,} x_n\textbf{:}T_n \; [\textbf{among} \; E_n \downarrow \mathcal{Y}] \; \textbf{in} \; E' \downarrow \mathcal{Y} \\
 & | & \sigma Y \; \textbf{(}x_1\textbf{:}T_1\textbf{:=}E_1 \downarrow \mathcal{Y}\textbf{,} \ldots \textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n \downarrow \mathcal{Y}\textbf{)} \; \textbf{.} \; E' \downarrow \mathcal{Y}' \\
 & & \left\{ \begin{array}{l} \textbf{soit} \; \mathcal{Y}'' := def_Y(Y, sign(\sigma), n, (type(T_1), \ldots, type(T_n))) \; \textbf{dans} \\ \quad use_Y(Y, n, \mathcal{Y}''); \\ \quad \mathcal{Y}' := \mathcal{Y} \oslash \mathcal{Y}'' \\ \textbf{fin} \end{array} \right. \\
 & | & [E_1 \downarrow \mathcal{Y}] \; |\!= E_2 \downarrow \mathcal{Y} \\
 & | & [E_1 \downarrow \mathcal{Y}] \; |\!= \textbf{action} \; \alpha \\
 & | & [\,[E_1 \downarrow \mathcal{Y}]\,] \\
 & | & E_1 \downarrow \mathcal{Y} \; F \; E_2 \downarrow \mathcal{Y} \\
 & | & I \\
 & & \quad \{ \; use_Y(I, 0, \mathcal{Y}) \\
 & | & I \; \textbf{(}E_1 \downarrow \mathcal{Y}\textbf{,} \ldots \textbf{,}E_n \downarrow \mathcal{Y}\textbf{)} \\
 & & \quad \{ \; use_Y(I, n, \mathcal{Y}) \\
 & | & E_1 \downarrow \mathcal{Y} \; \textbf{.} \; x \\
 & | & E_1 \downarrow \mathcal{Y} \; \textbf{of} \; RT \\
 & | & E_1 \downarrow \mathcal{Y} \; \textbf{;} \; E_2 \downarrow \mathcal{Y} \\
 & | & \textbf{print} \; \textbf{(}E_1 \downarrow \mathcal{Y}\textbf{)} \\
 & | & \textbf{(}E_0 \downarrow \mathcal{Y}\textbf{,} \ldots \textbf{,}E_n \downarrow \mathcal{Y}\textbf{)} \\
 & | & \textbf{let} \; x_0\textbf{:}T_0\textbf{:=}E_0 \downarrow \mathcal{Y}\textbf{,} \ldots \textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n \downarrow \mathcal{Y} \; \textbf{in} \\
 & & \quad E' \downarrow \mathcal{Y} \\
 & & \textbf{endlet} \\
 & | & \textbf{let} \; \textbf{(}x_0^0\textbf{:}T_0^0\textbf{,} \ldots \textbf{,}x_0^{n_0}\textbf{:}T_0^{n_0}\textbf{):=}E_0 \downarrow \mathcal{Y}\textbf{,} \ldots \textbf{,(}x_m^0\textbf{:}T_m^0\textbf{,} \ldots \textbf{,}x_m^{n_m}\textbf{:}T_m^{n_m}\textbf{):=}E_m \downarrow \mathcal{Y} \; \textbf{in} \\
 & & \quad E' \downarrow \mathcal{Y} \\
 & & \textbf{endlet} \\
 & | & \textbf{assert} \; E_0 \downarrow \mathcal{Y}\textbf{,} \ldots \textbf{,}E_n \downarrow \mathcal{Y} \; \textbf{in} \\
 & & \quad E' \downarrow \mathcal{Y} \\
 & & \textbf{endassert}
\end{array}
$$

$|$ **if** $E_0 \downarrow \mathcal{Y}$ **then** $E'_0 \downarrow \mathcal{Y}$
elsif $E_1 \downarrow \mathcal{Y}$ **then** $E'_1 \downarrow \mathcal{Y}$
$\ldots$
elsif $E_n \downarrow \mathcal{Y}$ **then** $E'_n \downarrow \mathcal{Y}$
[**else** $E'_{n+1} \downarrow \mathcal{Y}$]
endif

$|$ **case** $E_0 \downarrow \mathcal{Y}$ **in**
P_1^0 $|\ldots|$ $P_1^{n_1}$ [**where** $E_1 \downarrow \mathcal{Y}$] **->** $E'_1 \downarrow \mathcal{Y}$
$\ldots$
$|$ P_m^0 $|\ldots|$ $P_m^{n_m}$ [**where** $E_m \downarrow \mathcal{Y}$] **->** $E'_m \downarrow \mathcal{Y}$
[$|$ **otherwise ->** $E'_{m+1} \downarrow \mathcal{Y}$]
endcase

$|$ **case action** $E_0 \downarrow \mathcal{Y}$ **in**
α_1 [**where** $E_1 \downarrow \mathcal{Y}$] **->** $E'_1 \downarrow \mathcal{Y}$
$\ldots$
$|$ α_m [**where** $E_m \downarrow \mathcal{Y}$] **->** $E'_m \downarrow \mathcal{Y}$
[$|$ **otherwise ->** $E'_{m+1} \downarrow \mathcal{Y}$]
endcase

$|$ **loop** (x_0:T_0:=$E_0 \downarrow \mathcal{Y}, \ldots, x_n$:$T_n$:=$E_n \downarrow \mathcal{Y}$) **:** RT **in**
$E' \downarrow \mathcal{Y}$
endloop

$|$ **continue** ($E_0 \downarrow \mathcal{Y}, \ldots, E_n \downarrow \mathcal{Y}$)

$|$ **for** x'_0:T'_0[**among** $E'_0 \downarrow \mathcal{Y}$]$, \ldots, x'_m$:$T'_m$[**among** $E'_m \downarrow \mathcal{Y}$]
[**var** x_0:T_0:=$E_0 \downarrow \mathcal{Y}, \ldots, x_n$:$T_n$:=$E_n \downarrow \mathcal{Y}$]
[**where** $E''_1 \downarrow \mathcal{Y}$]
[**while** $E''_2 \downarrow \mathcal{Y}$]
in $E''_3 \downarrow \mathcal{Y}$
[**result** $E''_4 \downarrow \mathcal{Y}$]
endfor

$|$ $\{E_1 \downarrow \mathcal{Y}, \ldots, E_n \downarrow \mathcal{Y}\}$

$|$ $\{E_1 \downarrow \mathcal{Y}$ **...** $E_2 \downarrow \mathcal{Y}\}$

$|$ { F **on** x_0:T_0 [**among** $E_0 \downarrow \mathcal{Y}$]$, \ldots, x_n$:$T_n$ [**among** $E_n \downarrow \mathcal{Y}$] [**where** $E'_1 \downarrow \mathcal{Y}$] } $E'_2 \downarrow \mathcal{Y}$

$|$ { x:T [**among** $E_1 \downarrow \mathcal{Y}$] **where** $E_2 \downarrow \mathcal{Y}$ }

Définitions de fonctions

D ::= [**local**] **function** F (x_1:$T_1, \ldots, x_n$:T_n) **:** RT **is**
$E \downarrow \mathcal{Y}$
endfunc
$\{\ \mathcal{Y} := [\,]$

| [**local**] **function** $\_ \, F \, \_$ $(x_1{:}T_1{,}x_2{:}T_2)$ **:** RT **is**
  $E \downarrow \mathcal{Y}$
endfunc
  $\{ \ \mathcal{Y} := [\,]$

Programme

PG ::= $[D_0 \ldots D_m]$
  $E \downarrow \mathcal{Y}$
[**where** $D_{m+1} \ldots D_{m+p}$]
  $\{ \ \mathcal{Y} := [\,]$

A.5.4 Discussion

La liaison des variables propositionnelles permet de lever l'ambiguïté concernant les appels de variables propositionnelles et de fonctions : les occurrences du symbole terminal I qui ont pu être liées comme appels de variables propositionnelles sont remplacées dans l'arbre abstrait par des occurrences du symbole terminal Y. Les occurrences de I qui restent sont censées dénoter des appels de fonctions, et donc elles peuvent d'ores et déjà être remplacées par des occurrences du symbole terminal F. Autrement dit, pour les occurrences de variables Y et de fonctions F, les règles syntaxiques "E ::= I | I $(E_1, \ldots, E_n)$" sont remplacées respectivement par "Y | Y $(E_1, \ldots, E_n)$" et par "F | F $(E_1, \ldots, E_n)$".

A.6 Liaison des fonctions

Dans le langage XTL il existe deux classes de fonctions (dénotées par le symbole terminal F) :

Les fonctions internes, prédéfinies ou définies dans le programme XTL (voir la section 2.3.1) ;

Les fonctions externes, définies dans le programme source à vérifier (voir la section 2.3.2).

Un programme XTL peut contenir des occurrences de définition de fonctions internes et des occurrences d'utilisation (ou *appels*) de fonctions internes et/ou externes. Les occurrences de définition des fonctions internes sont contenues dans les définitions de fonctions XTL (symbole non-terminal D) :

D ::= [**local**] **function** F $(x_1{:}T_1, \ldots, x_n{:}T_n)$ **:** RT **is** E **endfunc**
  | [**local**] **function** $\_ \, F \, \_$ $(x_1{:}T_1{,}x_2{:}T_2)$ **:** RT **is** E **endfunc**

La récursion est autorisée : les fonctions F définies ci-dessus sont visibles dans leurs corps E respectifs. Les fonctions F "masquent" les autres fonctions, ayant le même nom et le même nombre de paramètres, qui étaient éventuellement visibles dans D : celles-ci ne sont plus visibles dans E. Les fonctions internes dont la définition contient le mot-clé "**local**" sont appelées *locales*, par opposition aux autres fonctions internes (y compris celles prédéfinies XTL), qui sont appelées *globales*.

Soit un programme XTL $[D_0 \ldots D_m]$ E [**where** $D_{m+1} \ldots D_{m+p}$]. Les fonctions internes globales, ainsi que les fonctions externes, sont visibles dans tout le programme XTL. Les fonctions locales F_i définies par D_i $(0 \leq i \leq m+p)$ sont visibles dans les corps de toutes les fonctions définies par $D_0, \ldots, D_{m+p}$, mais pas dans le corps E du programme.

Le langage XTL permet la définition de fonctions surchargées. La résolution des surcharges sera faite à la phase de typage des expressions et des formules (voir la section A.7).

Une construction similaire à une définition de fonction est l'expression d'itération "**loop**" :

$$E \ ::= \ \mathbf{loop}\ \mathbf{(}x_0\mathbf{:}T_0\mathbf{:=}E_0\mathbf{,} \ldots \mathbf{,}x_n\mathbf{:}T_n\mathbf{:=}E_n\mathbf{)}\ \mathbf{:}\ RT\ \mathbf{in}\ E'\ \mathbf{endloop}$$

L'expression E peut être assimilée à la définition et l'appel *in situ* d'une fonction récursive appelée `loop` (cet identificateur étant un mot-clé, il n'entre pas en conflit avec les autres identificateurs de fonction) ayant les paramètres formels $x_0\mathbf{:}T_0, \ldots, x_n\mathbf{:}T_n$, le corps E', les arguments $E_0, \ldots, E_n$ et le résultat de type RT. Cette fonction n'est visible que dans l'expression E' et ses appels ont la forme "**continue** $\mathbf{(}E_0\mathbf{,} \ldots \mathbf{,}E_n\mathbf{)}$". La définition de la fonction `loop` "masque" les autres fonctions `loop`, ayant le même nombre de paramètres, qui étaient éventuellement visibles dans E : celles-ci ne sont plus visibles dans E'.

La liaison des fonctions a deux objectifs : (1) identifier de manière unique chaque chaque fonction définie dans un programme XTL et (2) associer à chaque appel de fonction (y compris aux expressions "**continue**") un ensemble d'occurrences de définition surchargées de la fonction respective.

A.6.1 Attributs

Les informations statiques associées aux fonctions sont représentées par des *profils de fonctions* appartenant au domaine $\mathbf{ProfFunc} \stackrel{\mathrm{d}}{=} \mathbf{NumFunc} \times \mathbf{Bool} \times \mathbf{Bool} \times \mathbf{Nat} \times \mathbf{ExpType} \times \mathbf{ExpType}$, où **NumFunc** est le domaine des numéros uniques de fonctions. Un profil de fonction $p \in \mathbf{ProfFunc}$ est composé de six champs :

- $p.num \in \mathbf{NumFunc}$ est le numéro unique de la fonction ;
- $p.cons \in \mathbf{Bool}$ est vrai ssi la fonction est un opérateur constructeur ;
- $p.infix \in \mathbf{Bool}$ est vrai ssi la fonction est un opérateur infixé ;
- $p.arity \in \mathbf{Nat}$ est l'arité (nombre de paramètres) de la fonction ;
- $p.arg\_types \in \mathbf{ExpType}$ représente les types des paramètres de la fonction ;
- $p.res\_type \in \mathbf{ExpType}$ représente le type du résultat de la fonction.

La liaison des fonctions utilise des *environnements de fonctions* appartenant au domaine $\mathbf{EnvFunc} \stackrel{\mathrm{d}}{=} \mathbf{Idf} \rightarrow 2^{\mathbf{ProfFunc}}$. Un environnement de fonctions $\mathcal{F} \in \mathbf{EnvFunc}$ est une fonction partielle ensembliste associant aux identificateurs de fonction des ensembles de profils surchargés. La table A.4 décrit les attributs gérés pendant la liaison des fonctions.

Les fonctions XTL prédéfinies et les fonctions externes sont contenues respectivement dans les environnements $\mathcal{F}_{xtl}$ et $\mathcal{F}_{bcg}$. Ces environnements sont initialisés avant de commencer la liaison des fonctions et ne sont plus modifiés ensuite : il n'est donc pas nécessaire de les transmettre comme attributs hérités dans les règles syntaxiques.

Remarque A-10
Les images de $\mathcal{F}_{xtl}$ et de $\mathcal{F}_{bcg}$ sont disjointes : une fonction XTL prédéfinie et une fonction externe ayant le même nom auront des numéros uniques différents associés dans les deux environnements. ■

A la fin de cette phase de liaison, chaque occurrence de fonction F aura associé un ensemble de profils surchargés possibles $\mathit{profiles}(F)$. Les occurrences des mots-clés "**loop**" et "**continue**" auront associé un profil unique, correspondant à la fonction implicitement définie et utilisée.

CLASSE D'ATTRIBUTS	NOM	SIGNIFICATION	SYMBOLES D'ATTACHEMENT
hérités	$\mathcal{F} \in$ **EnvFunc**	environnement de fonctions internes visibles	P, O, α, R, E, D
synthétisés	$\mathcal{F}_{def} \in$ **EnvFunc**	environnement de fonction définie	D
	$loc \in$ **Bool**	vrai ssi la définition est locale	D
locaux	$profiles \in 2^{\mathbf{ProfFunc}}$	ensemble de profils de fonction possibles	F, **print**
	$profile \in$ **ProfFunc**	profil unique de fonction	**loop**, **continue**, **nop**, **;**

Table A.4: Attributs gérés pendant la liaison des fonctions

A.6.2 Actions sémantiques

La liaison des fonctions est effectuée au moyen des actions sémantiques suivantes :

- def_F : **Idf** $\times$ **Bool** $\times$ **Bool** $\times$ **Nat** $\times$ **ExpType** $\times$ **ExpType** $\to$ **EnvFunc**, qui renvoie l'environnement associant à un identificateur de fonction un seul profil contenant un numéro unique :

$$def_F(F, cons, infix, arity, arg\_types, res\_type) \stackrel{\mathrm{d}}{=} [\{\langle uniq_F(\ ), cons, infix, arity, arg\_types, res\_type\rangle\}/F]$$

  L'action auxiliaire $uniq_F(\ )$ renvoie à chaque appel un nouveau numéro de fonction.

- use_F : **Idf** $\times$ **Bool** $\times$ **Bool** $\times$ **Nat** $\times$ **EnvFunc** $\to$ **Void**, qui positionne, en utilisant un environnement de fonctions, l'attribut local *profiles* associé à une occurrence de fonction :

$$use_F(F, cons, infix, arity, \mathcal{F}) \stackrel{\mathrm{d}}{=} \left\{ \begin{array}{l} \textbf{si } internal\_id(F) \ \wedge \ F \in supp(\mathcal{F}) \textbf{ alors} \\ \quad profiles(F) := \{p \in \mathcal{F}(F) \mid (cons \Rightarrow p.cons) \ \wedge \\ \qquad\qquad (infix \Rightarrow p.infix) \ \wedge \ p.arity = arity\} \\ \textbf{sinon\_si } F \in supp(\mathcal{F}_{bcg}) \textbf{ alors} \\ \quad profiles(F) := \{p \in \mathcal{F}_{bcg}(F) \mid (cons \Rightarrow p.cons) \ \wedge \\ \qquad\qquad (infix \Rightarrow p.infix) \ \wedge \ p.arity = arity\} \\ \textbf{sinon} \\ \quad error(\texttt{"undefined identifier"}) \\ \textbf{fin\_si} \end{array} \right.$$

Remarque A-11
L'action $use_F(\ )$ respecte les conventions de liaison mentionnées à la section 2.3.2 : si une fonction interne et une fonction externe BCG ont le même nom, la priorité est donnée par défaut à la fonction interne ; cependant, l'utilisateur peut forcer la liaison avec la fonction BCG en utilisant pour celle-ci un identificateur externe (entouré par des caractères '‘').

Il existe toutefois une exception à cette règle, destinée à faciliter l'écriture des portes G dans les filtres d'actions : en cas de conflit de nom avec une fonction XTL interne, il n'est pas nécessaire d'entourer l'identificateur de porte par des caractères '‘'. En effet, sachant précisément que G dénote une fonction externe BCG de type `gate`, le compilateur peut forcer par défaut la liaison externe en positionnant l'attribut $internal\_id(G)$ à faux avant de commencer la liaison des fonctions. ■

A.6.3 Grammaire attribuée

Filtres

$$
\begin{array}{lll}
P \downarrow \mathcal{F} & ::= & P_1 \downarrow \mathcal{F} \textbf{ of } RT \\
 & | & C\ \textbf{(}P_1 \downarrow \mathcal{F}\textbf{,}\ldots\textbf{,}P_n \downarrow \mathcal{F}\textbf{)} \\
 & & \quad \{\ use_F(C, true, false, n, \mathcal{F})
\end{array}
$$

Offres

$$
\begin{array}{lll}
O \downarrow \mathcal{F} & ::= & \textbf{!}\ E \downarrow \mathcal{F} \\
 & | & \textbf{?}\ P_0 \downarrow \mathcal{F} \mid \ldots \mid P_n \downarrow \mathcal{F}
\end{array}
$$

Formules sur actions

$$
\begin{array}{lll}
\alpha \downarrow \mathcal{F} & ::= & (G_0 | O_0 \downarrow \mathcal{F})\ O_1 \downarrow \mathcal{F} \ldots O_m \downarrow \mathcal{F}\ [\textbf{...}]\ O_{m+1} \downarrow \mathcal{F} \ldots O_{m+n} \downarrow \mathcal{F}\ [\textbf{where } E \downarrow \mathcal{F}] \\
 & & \quad \{\ use_F(G, true, false, 0, \mathcal{F}) \\
 & | & unary\_bool\_op\ \alpha_1 \downarrow \mathcal{F} \\
 & | & \alpha_1 \downarrow \mathcal{F}\ binary\_bool\_op\ \alpha_2 \downarrow \mathcal{F}
\end{array}
$$

Expressions régulières

$$
\begin{array}{lll}
R \downarrow \mathcal{F} & ::= & \alpha \downarrow \mathcal{F} \\
 & | & R_1 \downarrow \mathcal{F}\ \textbf{.}\ R_2 \downarrow \mathcal{F} \\
 & | & R_1 \downarrow \mathcal{F} \mid R_2 \downarrow \mathcal{F} \\
 & | & R_1 \downarrow \mathcal{F}\ ^{\textbf{*}} \\
 & | & R_1 \downarrow \mathcal{F}\ ^{\textbf{+}}
\end{array}
$$

Expressions

$$
\begin{array}{lll}
E \downarrow \mathcal{F} & ::= & unary\_bool\_op\ E_1 \downarrow \mathcal{F} \\
 & | & E_1 \downarrow \mathcal{F}\ binary\_bool\_op\ E_2 \downarrow \mathcal{F} \\
 & | & \langle R \downarrow \mathcal{F} \rangle\ E_1 \downarrow \mathcal{F} \\
 & | & [\, R \downarrow \mathcal{F}\,]\ E_1 \downarrow \mathcal{F} \\
 & | & \textbf{@}\ \textbf{(}R \downarrow \mathcal{F}\textbf{)} \\
 & | & quantifier\ x_0\textbf{:}T_0\ [\textbf{among } E_0 \downarrow \mathcal{F}]\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\ [\textbf{among } E_n \downarrow \mathcal{F}]\ \textbf{in } E' \downarrow \mathcal{F} \\
 & | & \sigma Y\ \textbf{(}x_1\textbf{:}T_1\textbf{:=}E_1 \downarrow \mathcal{F}\textbf{,}\ldots\textbf{,}x_n\textbf{:}T_n\textbf{:=}E_n \downarrow \mathcal{F}\textbf{)}\ \textbf{.}\ E' \downarrow \mathcal{F} \\
 & | & [E_1 \downarrow \mathcal{F}] \models E_2 \downarrow \mathcal{F} \\
 & | & [E_1 \downarrow \mathcal{F}] \models \textbf{action } \alpha \downarrow \mathcal{F} \\
 & | & [\,[E_1 \downarrow \mathcal{F}]\,] \\
 & | & [\,[\textbf{action } \alpha \downarrow \mathcal{F}]\,] \\
 & | & F \\
 & & \quad \{\ use_F(F, false, false, 0, \mathcal{F})
\end{array}
$$

$|\quad E_1 \downarrow \mathcal{F}\ F\ E_2 \downarrow \mathcal{F}$
$\quad\{\ use_F(F, \mathit{false}, \mathit{true}, 2, \mathcal{F})$
$|\quad F\ (E_1 \downarrow \mathcal{F}, \ldots, E_n \downarrow \mathcal{F})$
$\quad\{\ use_F(F, \mathit{false}, \mathit{false}, n, \mathcal{F})$
$|\quad E_1 \downarrow \mathcal{F}\ .\ x$
$|\quad E_1 \downarrow \mathcal{F}$ **of** RT
$|\quad$ **nop**
$\quad\{\ \mathit{profile}(\mathbf{nop}) := \mathcal{F}_{xtl}(\texttt{nop})$
$|\quad E_1 \downarrow \mathcal{F}\ \mathbf{;}\ E_2 \downarrow \mathcal{F}$
$\quad\{\ \mathit{profile}(\mathbf{;}) := \mathcal{F}_{xtl}(\texttt{;})$
$|\quad$ **print** $(E_1 \downarrow \mathcal{F})$
$\quad\{\ use_F(\mathbf{print}, \mathit{false}, \mathit{false}, 1, \mathcal{F})$
$|\quad (E_0 \downarrow \mathcal{F}, \ldots, E_n \downarrow \mathcal{F})$
$|\quad$ **let** $x_0{:}T_0{:=}E_0 \downarrow \mathcal{F}, \ldots, x_n{:}T_n{:=}E_n \downarrow \mathcal{F}$ **in**
$\quad E' \downarrow \mathcal{F}$
endlet
$|\quad$ **let** $(x_0^0{:}T_0^0, \ldots, x_0^{n_0}{:}T_0^{n_0}){:=}E_0 \downarrow \mathcal{F}, \ldots, (x_m^0{:}T_m^0, \ldots, x_m^{n_m}{:}T_m^{n_m}){:=}E_m \downarrow \mathcal{F}$ **in**
$\quad E' \downarrow \mathcal{F}$
endlet
$|\quad$ **assert** $E_0 \downarrow \mathcal{F}, \ldots, E_n \downarrow \mathcal{F}$ **in**
$\quad E' \downarrow \mathcal{F}$
endassert
$|\quad$ **if** $E_0 \downarrow \mathcal{F}$ **then** $E'_0 \downarrow \mathcal{F}$
elsif $E_1 \downarrow \mathcal{F}$ **then** $E'_1 \downarrow \mathcal{F}$
$\ldots$
elsif $E_n \downarrow \mathcal{F}$ **then** $E'_n \downarrow \mathcal{F}$
[**else** $E'_{n+1} \downarrow \mathcal{F}$]
endif
$|\quad$ **case** $E_0 \downarrow \mathcal{F}$ **in**
$\quad P_1^0 \downarrow \mathcal{F}\ |\ldots|\ P_1^{n_1} \downarrow \mathcal{F}$ [**where** $E_1 \downarrow \mathcal{F}$] –> $E'_1 \downarrow \mathcal{F}$
$\quad\ldots$
$|\ P_m^0 \downarrow \mathcal{F}\ |\ldots|\ P_m^{n_m} \downarrow \mathcal{F}$ [**where** $E_m \downarrow \mathcal{F}$] –> $E'_m \downarrow \mathcal{F}$
[| **otherwise** –> $E'_{m+1} \downarrow \mathcal{F}$]
endcase
$|\quad$ **case action** $E_0 \downarrow \mathcal{F}$ **in**
$\quad \alpha_1 \downarrow \mathcal{F}$ [**where** $E_1 \downarrow \mathcal{F}$] –> $E'_1 \downarrow \mathcal{F}$
$\quad\ldots$
$|\ \alpha_m \downarrow \mathcal{F}$ [**where** $E_m \downarrow \mathcal{F}$] –> $E'_m \downarrow \mathcal{F}$
[| **otherwise** –> $E'_{m+1} \downarrow \mathcal{F}$]
endcase

| **loop** (x_0:T_0:=$E_0 \downarrow \mathcal{F}$, ... ,$x_n$:$T_n$:=$E_n \downarrow \mathcal{F}$) **:** RT **in**
  $E' \downarrow \mathcal{F}'$
endloop

$$\left\{\begin{array}{l} \textbf{soit } \mathcal{F}'' := \mathit{def}_F(\texttt{loop}, \mathit{false}, \mathit{false}, n+1, (\mathit{type}(T_0), \ldots, \mathit{type}(T_n)), \mathit{type}(RT)) \\ \textbf{dans} \\ \quad \mathit{profile}(\textbf{loop}) := \mathcal{F}''(\texttt{loop}); \\ \quad \mathcal{F}' := \mathcal{F} \oslash \mathcal{F}'' \\ \textbf{fin} \end{array}\right.$$

| **continue** ($E_0 \downarrow \mathcal{F}$, ... ,$E_n \downarrow \mathcal{F}$)

$$\left\{\begin{array}{l} \textbf{si } \mathcal{F}(\texttt{loop}) = \{p\} \ \wedge \ p.\mathit{arity} = n+1 \textbf{ alors} \\ \quad \mathit{profile}(\textbf{continue}) := p \\ \textbf{sinon si } \mathcal{F}(\texttt{loop}) = \{p\} \textbf{ alors} \\ \quad \mathit{error}(\texttt{"wrong number of arguments in continue"}) \\ \textbf{sinon} \\ \quad \mathit{error}(\texttt{"continue outside loop"}) \\ \textbf{fin\_si} \end{array}\right.$$

| **for** x'_0:T'_0[**among** $E'_0 \downarrow \mathcal{F}$], ... ,$x'_m$:$T'_m$[**among** $E'_m \downarrow \mathcal{F}$]
  [**var** x_0:T_0:=$E_0 \downarrow \mathcal{F}$, ... ,$x_n$:$T_n$:=$E_n \downarrow \mathcal{F}$]
  [**where** $E''_1 \downarrow \mathcal{F}$]
  [**while** $E''_2 \downarrow \mathcal{F}$]
  in $E''_3 \downarrow \mathcal{F}$
  [**result** $E''_4 \downarrow \mathcal{F}$]
endfor

| {$E_1 \downarrow \mathcal{F}$, ... ,$E_n \downarrow \mathcal{F}$}

| {$E_1 \downarrow \mathcal{F}$ **...** $E_2 \downarrow \mathcal{F}$}

| { F **on** x_0:T_0 [**among** $E_0 \downarrow \mathcal{F}$], ... ,$x_n$:$T_n$ [**among** $E_n \downarrow \mathcal{F}$]
  [**where** $E'_1 \downarrow \mathcal{F}$] } $E'_2 \downarrow \mathcal{F}$

$$\left\{ \ \mathit{use}_F(F, \mathit{false}, \mathit{true}, 2, \mathcal{F}) \right.$$

| { x:T [**among** $E_1 \downarrow \mathcal{F}$] **where** $E_2 \downarrow \mathcal{F}$ }

Définitions de fonctions

$D \uparrow \mathcal{F}_{def} \uparrow loc \downarrow \mathcal{F}$ ::= [**local**] **function** F (x_1:T_1, ... ,x_n:T_n) **:** RT **is**
  $E \downarrow \mathcal{F}$
endfunc

$$\left\{\begin{array}{l} \textbf{soit } \mathcal{F}' := \mathit{def}_F(F, \mathit{false}, \mathit{false}, n, (\mathit{type}(T_1), \ldots, \mathit{type}(T_n)), \mathit{type}(RT)) \textbf{ dans} \\ \quad \mathit{use}_F(F, \mathit{false}, \mathit{false}, n, \mathcal{F}'); \\ \quad \mathcal{F}_{def} := \mathcal{F}'; \\ \quad \mathit{loc} := \mathit{false}\,[\ \vee\ \mathit{true}] \\ \textbf{fin} \end{array}\right.$$

$$
\begin{array}{l}
\mid \quad [\textbf{local}]\, \textbf{function} \ \_\ F \ \_\ (x_1\textbf{:}T_1\textbf{,}x_2\textbf{:}T_2) \ \textbf{:}\ RT \ \textbf{is} \\
\quad\quad E \downarrow \mathcal{F}' \\
\quad \textbf{endfunc} \\
\quad \left\{ \begin{array}{l}
\textbf{soit } \mathcal{F}' := \mathit{def}_F(F, \mathit{false}, \mathit{true}, 2, (\mathit{type}(T_1), \mathit{type}(T_2)), \mathit{type}(RT)) \textbf{ dans} \\
\quad \mathit{use}_F(F, \mathit{false}, \mathit{true}, 2, \mathcal{F}'); \\
\quad \mathcal{F}_{\mathit{def}} := \mathcal{F}'; \\
\quad \mathit{loc} := \mathit{false}\, [\, \vee\ \mathit{true}] \\
\textbf{fin}
\end{array} \right.
\end{array}
$$

Programme

$$
\begin{array}{rcl}
PG & ::= & [D_0 \uparrow \mathcal{F}_{\mathit{def}\,0} \uparrow \mathit{loc}_0 \downarrow \mathcal{F}_0 \ldots D_m \uparrow \mathcal{F}_{\mathit{def}\,m} \uparrow \mathit{loc}_m \downarrow \mathcal{F}_m] \\
 & & \quad E \downarrow \mathcal{F} \\
 & & [\textbf{where } D_{m+1} \uparrow \mathcal{F}_{\mathit{def}\,m+1} \uparrow \mathit{loc}_{m+1} \downarrow \mathcal{F}_{m+1} \ldots D_{m+p} \uparrow \mathcal{F}_{\mathit{def}\,m+p} \uparrow \mathit{loc}_{m+p} \downarrow \mathcal{F}_{m+p}] \\
 & & \left\{ \begin{array}{l}
\textbf{soit } \mathcal{F}' := \left[\bigoplus_{i=0}^{m} \left\{ \mathcal{F}_{\mathit{def}\,i} \mid \mathit{loc}_i = \mathit{true} \right\} \right] \oplus \left[\bigoplus_{j=m+1}^{m+p} \left\{ \mathcal{F}_{\mathit{def}\,j} \mid \mathit{loc}_j = \mathit{true} \right\} \right] \\
\qquad \mathcal{F}'' := \left[\bigoplus_{i=0}^{m} \left\{ \mathcal{F}_{\mathit{def}\,i} \mid \mathit{loc}_i = \mathit{false} \right\} \right] \oplus \left[\bigoplus_{j=m+1}^{m+p} \left\{ \mathcal{F}_{\mathit{def}\,j} \mid \mathit{loc}_j = \mathit{false} \right\} \right] \\
\textbf{dans} \\
\quad \left[\forall i \in [0, m],\ \mathcal{F}_i := \mathcal{F}_{\mathit{xtl}} \oslash \mathcal{F}'; \right] \\
\quad \left[\forall j \in [m+1, m+p],\ \mathcal{F}_j := \mathcal{F}_{\mathit{xtl}} \oslash \mathcal{F}'; \right] \\
\quad \mathcal{F} := \mathcal{F}_{\mathit{xtl}} \oslash \mathcal{F}'' \\
\textbf{fin}
\end{array} \right.
\end{array}
$$

A.6.4 Discussion

Durant la liaison des fonctions, les deux propriétés ci-dessous sont invariantes.

1. Pour toute règle syntaxique, environnement $\downarrow \mathcal{F}$ courant et identificateur $F \in \mathit{supp}(\mathcal{F})$:

   $$\forall p \in \mathcal{F}(F).\ p.\mathit{infix} \Rightarrow (p.\mathit{arity} = 2).$$

   En supposant que cette propriété est vraie aussi pour les profils des fonctions contenues dans les environnements $\mathcal{F}_{\mathit{xtl}}$ et $\mathcal{F}_{\mathit{bcg}}$, elle le sera également pour les profils $p \in \mathit{profiles}(F) \subseteq \mathcal{F}(F) \cup \mathcal{F}_{\mathit{xtl}}(F) \cup \mathcal{F}_{\mathit{bcg}}(F)$ obtenus après la liaison de chaque occurrence de fonction F.

2. Pour toute règle syntaxique et environnement $\downarrow \mathcal{F}$ courant :

   $$\texttt{loop} \in \mathit{supp}(\mathcal{F}) \Rightarrow |\mathcal{F}(\texttt{loop})| = 1.$$

   Autrement dit, les fonctions récursives implicites correspondant aux expressions d'itération "**loop**" ne peuvent pas être surchargées, ce qui signifie que chaque expression "**continue**" sera liée à la plus petite expression "**loop**" qui la contient.

Ces deux propriétés peuvent être montrées à partir de la grammaire attribuée donnée à la section A.6.3, par induction sur la longueur de la dérivation permettant d'atteindre la règle syntaxique courante à partir de l'axiome de la grammaire.

Remarque A-12
Les profils uniques des opérateurs prédéfinis "**;**" et "**nop**" qui, étant des mots-clés, ne peuvent pas être surchargés, ont pu être positionnés à l'aide de l'environnement $\mathcal{F}_{\mathit{xtl}}$. ■

Afin de réduire le nombre de règles sémantiques qui seront utilisées par la suite, les définitions et appels d'opérateurs infixés peuvent d'ores et déjà être remplacées par leurs correspondants préfixés, ceci n'ayant aucune influence sur les phases ultérieures d'analyse.

A.7 Typage des expressions et des formules

Le but de cette phase d'analyse est d'associer à chaque objet d'un programme XTL (qu'il s'agisse d'une variable, d'une fonction ou d'une expression) une information unique caractérisant son type. Plusieurs activités sont effectuées simultanément durant le typage :

- Calcul des types uniques associés aux expressions E et aux filtres P. Du fait de la présence des objets surchargés (variables propositionnelles Y et fonctions F), une expression peut avoir plusieurs types possibles ; en cas d'ambiguïté, le contexte de l'expression doit fournir suffisamment d'information pour restreindre ce choix à un seul type. Le calcul des types est fait en deux phases [ASU86] : (1) le calcul d'un ensemble de types possibles pour chaque E et P et (2) la restriction de cet ensemble à un type unique.
- Résolution des surcharges des variables propositionnelles et des fonctions : chaque appel de variable Y et de fonction F doit avoir un profil unique, choisi respectivement parmi l'ensemble des profils possibles $\mathit{profils}(Y)$ et $\mathit{profils}(F)$ qui ont été calculés lors des phases de liaison décrites aux sections A.5 et A.6. La levée des surcharges est effectuée conjointement avec le calcul des types : le profil final de chaque appel de variable propositionnelle ou de fonction est sélectionné lorsque les types uniques de ses arguments ont été déterminés.
- Séparation des formules comme catégorie syntaxique et sémantique disjointe des expressions. A cause de la similitude entre certaines formules et expressions (les constructions "**if-then-else**", "**let**", "**case**", etc.), cette séparation ne peut pas être faite syntaxiquement. En revanche, elle peut être effectuée grâce à l'information de type : suivant leur type et leur contexte d'utilisation, certaines expressions seront converties à des formules.
- Liaison des opérateurs $\mathit{const\_bool\_op}$, $\mathit{unary\_bool\_op}$ et $\mathit{binary\_bool\_op}$, qui peuvent dénoter soit des fonctions booléennes prédéfinies, soit des opérateurs booléens sur formules. Leur liaison ne peut donc être effectuée qu'après avoir séparé les formules des expressions.

Par souci de concision, nous avons fusionné toutes les actions sémantiques correspondant aux activités ci-dessus en une seule grammaire attribuée.

A.7.1 Attributs

Le typage des expressions et des formules utilise des ensembles d'expressions de type. La table A.5 décrit les attributs gérés pendant la phase de typage.

Classe d'attributs	Nom	Signification	Symboles d'attachement
hérités	*tu*	type unique imposé par le contexte	E, P
synthétisés	*tp*	ensemble de types possibles	E, P
locaux	*type*	type unique	E, P
	profile	profil unique de fonction ou de variable propositionnelle	C, F, Y, $\mathit{const\_bool\_op}$, $\mathit{unary\_bool\_op}$, $\mathit{binary\_bool\_op}$
	id	identificateur	$\mathit{const\_bool\_op}$, $\mathit{unary\_bool\_op}$, $\mathit{binary\_bool\_op}$

Table A.5: Attributs gérés pendant le typage

Afin de fusionner la séparation des formules et le calcul des types, nous introduisons les types auxiliaires `formula` et `bool_to_formula`. Le type `formula` sert à séparer les formules des expressions :

il sera associé aux occurrences du symbole non-terminal E qui dénotent des formules. Le type `bool_to_formula` sert à convertir des expressions booléennes à des formules : il sera associé aux expressions E ayant le type unique `boolean`, mais qui dénotent en fait des prédicats de base.

A.7.2 Actions sémantiques

Afin de simplifier les règles sémantiques, nous utilisons l'action sémantique *setof* : **ExpType** $\to$ **ExpType**, qui renvoie le type XTL dénotant les ensembles d'éléments d'un certain type :

$$\mathit{setof}(t) \stackrel{\mathrm{d}}{=} \begin{cases} \texttt{intset} & \text{si} \quad t = \texttt{integer} \\ \texttt{charset} & \text{si} \quad t = \texttt{character} \\ \texttt{stateset} & \text{si} \quad t = \texttt{state} \\ \texttt{labelset} & \text{si} \quad t = \texttt{label} \\ \texttt{transset} & \text{si} \quad t = \texttt{trans} \\ \mathit{error}(\texttt{"unauthorized set type"}) & \text{sinon} \end{cases}$$

Remarque A-13
Par souci de concision, les erreurs sémantiques rencontrées pendant la phase de typage sont signalées le plus tard possible (généralement, lors du calcul de l'attribut hérité $\downarrow tu$). Cette approche permet de simplifier les règles sémantiques, mais présente l'inconvénient de ne pas fournir de diagnostics très précis sur la nature des erreurs. En pratique, ceci peut être remédié en suivant l'approche complémentaire, qui consiste à signaler les erreurs le plus tôt possible. ■

A.7.3 Grammaire attribuée

Filtres

$$\begin{array}{lll}
P \uparrow tp \downarrow tu & ::= & x\mathbf{:}T \\
& & \left\{ \begin{array}{l} tp := \{\mathit{type}(T)\}; \\ \mathit{type}(P) := tu \end{array} \right. \\
& | & (x_0\mathbf{:}T_0, \ldots, x_n\mathbf{:}T_n) \\
& & \left\{ \begin{array}{l} tp := \{(\mathit{type}(T_0), \ldots, \mathit{type}(T_n))\}; \\ \mathit{type}(P) := tu \end{array} \right. \\
& | & \mathbf{any}\ T \\
& & \left\{ \begin{array}{l} tp := \{\mathit{type}(T)\}; \\ \mathit{type}(P) := tu \end{array} \right. \\
& | & P_1 \uparrow tp_1 \downarrow tu_1\ \mathbf{of}\ RT \\
& & \left\{ \begin{array}{l} tp := tp_1 \cap \{\mathit{type}(RT)\}; \\ \mathit{type}(P) := tu_1 := tu \end{array} \right. \\
& | & C\ (P_1 \uparrow tp_1 \downarrow tu_1, \ldots, P_n \uparrow tp_n \downarrow tu_n) \\
& & \left\{ \begin{array}{l} tp := \{p.\mathit{res_type} \mid p \in \mathit{profiles}(C) \wedge p.\mathit{arg_types} \in \times_{i=1}^{n} tp_i\}; \\ \mathit{type}(P) := tu; \\ \mathbf{si}\ \{p \in \mathit{profiles}(C) \mid p.\mathit{arg_types} \in \times_{i=1}^{n} tp_i \wedge p.\mathit{res_type} = tu\} = \{p\}\ \mathbf{alors} \\ \quad \mathit{profile}(C) := p \\ \mathbf{sinon} \\ \quad \mathit{error}(\texttt{"ambiguous type of constructor pattern"}) \\ \mathbf{fin_si} \end{array} \right.
\end{array}$$

Offres

$$O ::= \mathbf{!}\ E \uparrow tp \downarrow tu$$

$$\left\{\begin{array}{l}\textbf{si } tp = \{t\} \textbf{ alors}\\ \quad tu := t\\ \textbf{sinon}\\ \quad error(\texttt{"ambiguous type of expression in offer"})\\ \textbf{fin\_si}\end{array}\right.$$

$$\mid\ \mathbf{?}\ P_0 \uparrow tp_0 \downarrow tu_0 \mid \ldots \mid P_n \uparrow tp_n \downarrow tu_n$$

$$\left\{\begin{array}{l}\textbf{si } \bigcap_{i=0}^{n} tp_i = \{t\} \textbf{ alors}\\ \quad tu_0 := \cdots := tu_n := t\\ \textbf{sinon}\\ \quad error(\texttt{"ambiguous type of filters in offer"})\\ \textbf{fin\_si}\end{array}\right.$$

Opérateurs

$$const\_bool\_op ::= \textbf{true}$$

$$\left\{\ id(const\_bool\_op) := \texttt{true}\right.$$

$$\mid\ \textbf{false}$$

$$\left\{\ id(const\_bool\_op) := \texttt{false}\right.$$

$$unary\_bool\_op ::= \textbf{not}$$

$$\left\{\ id(unary\_bool\_op) := \texttt{not}\right.$$

$$binary\_bool\_op ::= \textbf{or}$$

$$\left\{\ id(binary\_bool\_op) := \texttt{or}\right.$$

$$\mid\ \textbf{and}$$

$$\left\{\ id(binary\_bool\_op) := \texttt{and}\right.$$

$$\mid\ \textbf{implies}$$

$$\left\{\ id(binary\_bool\_op) := \texttt{implies}\right.$$

$$\mid\ \textbf{iff}$$

$$\left\{\ id(binary\_bool\_op) := \texttt{iff}\right.$$

$$\mid\ \textbf{xor}$$

$$\left\{\ id(binary\_bool\_op) := \texttt{xor}\right.$$

Formules sur actions

$$\alpha ::= (G_0 | O_0)\ O_1 \ldots O_m\ [\mathbf{\ldots}]\ O_{m+1} \ldots O_{m+n}\ [\textbf{where } E \uparrow tp \downarrow tu]$$

$$\left\{\begin{array}{l}\textbf{si } \texttt{boolean} \in tp \textbf{ alors}\\ \quad tu := \texttt{boolean}\\ \textbf{sinon}\\ \quad error(\texttt{"guard type must be boolean"})\\ \textbf{fin\_si}\end{array}\right.$$

Expressions

$$E \uparrow tp \downarrow tu \quad ::= \quad \mathit{const\_bool\_op}$$

$$\left\{\begin{array}{l} tp := \{\texttt{boolean}, \texttt{formula}\}; \\ \textbf{si } tu = \texttt{formula} \textbf{ alors} \\ \quad type(E) := \texttt{bool\_to\_formula} \\ \textbf{sinon} \\ \quad type(E) := \texttt{boolean} \\ \textbf{fin\_si}; \\ \mathit{profile}(\mathit{const\_bool\_op}) := \mathcal{F}_{xtl}(id(\mathit{const\_bool\_op})) \end{array}\right.$$

$$\mid \quad \mathit{unary\_bool\_op}\ E_1 \uparrow tp_1 \downarrow tu_1$$

$$\left\{\begin{array}{l} tp := \{\texttt{boolean}, \texttt{formula}\} \cap tp_1; \\ \textbf{si } tp = \{\texttt{formula}\} \textbf{ alors} \\ \quad type(E) := tu_1 := \texttt{formula} \\ \textbf{sinon} \\ \quad \textbf{si } tu = \texttt{formula} \textbf{ alors} \\ \quad \quad type(E) := \texttt{bool\_to\_formula} \\ \quad \textbf{sinon} \\ \quad \quad type(E) := \texttt{boolean} \\ \quad \textbf{fin\_si}; \\ \quad tu_1 := \texttt{boolean}; \\ \quad \mathit{profile}(\mathit{unary\_bool\_op}) := \mathcal{F}_{xtl}(id(\mathit{unary\_bool\_op})) \\ \textbf{fin\_si} \end{array}\right.$$

$$\mid \quad E_1 \uparrow tp_1 \downarrow tu_1\ \mathit{binary\_bool\_op}\ E_2 \uparrow tp_2 \downarrow tu_2$$

$$\left\{\begin{array}{l} tp := \{\texttt{boolean}, \texttt{formula}\} \cap tp_1 \cap tp_2; \\ \textbf{si } tp = \{\texttt{formula}\} \textbf{ alors} \\ \quad type(E) := tu_1 := tu_2 := \texttt{formula} \\ \textbf{sinon} \\ \quad \textbf{si } tu = \texttt{formula} \textbf{ alors} \\ \quad \quad type(E) := \texttt{bool\_to\_formula} \\ \quad \textbf{sinon} \\ \quad \quad type(E) := \texttt{boolean} \\ \quad \textbf{fin\_si}; \\ \quad tu_1 := tu_2 := \texttt{boolean}; \\ \quad \mathit{profile}(\mathit{binary\_bool\_op}) := \mathcal{F}_{xtl}(id(\mathit{binary\_bool\_op})) \\ \textbf{fin\_si} \end{array}\right.$$

$$\mid \quad \langle R \rangle\ E_1 \uparrow tp_1 \downarrow tu_1$$

$$\left\{\begin{array}{l} tp := \{\texttt{formula}\} \cap tp_1; \\ type(E) := tu_1 := \texttt{formula} \end{array}\right.$$

$$\mid \quad [R]\ E_1 \uparrow tp_1 \downarrow tu_1$$

$$\left\{\begin{array}{l} tp := \{\texttt{formula}\} \cap tp_1; \\ type(E) := tu_1 := \texttt{formula} \end{array}\right.$$

$$| \quad \textit{quantifier}\ x_0{:}T_0\ \ [\textbf{among}\ E_0 \uparrow tp_0 \downarrow tu_0], \ldots,$$
$$x_n{:}T_n\ [\textbf{among}\ E_n \uparrow tp_n \downarrow tu_n]\ \textbf{in}\ E' \uparrow tp' \downarrow tu'$$

$$\left\{\begin{array}{l}
tp := \{\texttt{boolean}, \texttt{formula}\} \cap tp'; \\
\forall i \in [0, n], \\
\quad \textbf{si}\ \textit{setof}(\textit{type}(T_i)) \in tp_i\ \textbf{alors} \\
\quad\quad tu_i := \textit{setof}(\textit{type}(T_i)) \\
\quad \textbf{sinon} \\
\quad\quad \textit{error}(\texttt{"domain type uncompatible with the iteration variable"}) \\
\quad \textbf{fin_si}; \\
\textbf{si}\ tp = \{\texttt{formula}\}\ \textbf{alors} \\
\quad \textit{type}(E) := tu' := \texttt{formula} \\
\textbf{sinon} \\
\quad \textbf{si}\ tu = \texttt{formula}\ \textbf{alors} \\
\quad\quad \textit{type}(E) := \texttt{bool_to_formula} \\
\quad \textbf{sinon} \\
\quad\quad \textit{type}(E) := \texttt{boolean} \\
\quad \textbf{fin_si}; \\
\quad tu' := \texttt{boolean} \\
\textbf{fin_si}
\end{array}\right.$$

$$| \quad \sigma Y\ \ (x_1{:}T_1{:=}E_1 \uparrow tp_1 \downarrow tu_1, \ldots, x_n{:}T_n{:=}E_n \uparrow tp_n \downarrow tu_n)\ \textbf{.}\ E' \uparrow tp' \downarrow tu'$$

$$\left\{\begin{array}{l}
tp := \{\texttt{formula}\} \cap tp'; \\
\forall i \in [0, n], \\
\quad \textbf{si}\ \textit{type}(T_i) \in tp_i\ \textbf{alors} \\
\quad\quad tu_i := \textit{type}(T_i) \\
\quad \textbf{sinon} \\
\quad\quad \textit{error}(\texttt{"argument type uncompatible with formal parameter"}) \\
\quad \textbf{fin_si}; \\
\textit{type}(E) := tu' := \texttt{formula}
\end{array}\right.$$

$$| \quad [E_1 \uparrow tp_1 \downarrow tu_1]\ \models E_2 \uparrow tp_2 \downarrow tu_2$$

$$\left\{\begin{array}{l}
tp := \{\texttt{boolean}, \texttt{formula} \mid [\texttt{state} \in tp_1 \wedge]\, \texttt{formula} \in tp_2\}; \\
\textit{type}(E) := \texttt{boolean}; \\
[tu_1 := \texttt{state};] \\
tu_2 := \texttt{formula}
\end{array}\right.$$

$$| \quad [E_1 \uparrow tp_1 \downarrow tu_1]\ \models \textbf{action}\ \alpha$$

$$\left\{\begin{array}{l}
tp := \{\texttt{boolean}, \texttt{formula} \mid \textit{true}\, [\wedge\, \texttt{label} \in tp_1]\}; \\
\textit{type}(E) := \texttt{boolean} \\
[tu_1 := \texttt{label}]
\end{array}\right.$$

$$| \quad [[\, E_1 \uparrow tp_1 \downarrow tu_1\,]]$$

$$\left\{\begin{array}{l}
tp := \{\texttt{stateset} \mid \texttt{formula} \in tp_1\}; \\
\textit{type}(E) := tu; \\
tu_1 := \texttt{formula}
\end{array}\right.$$

$$| \quad [[\, \textbf{action}\ \alpha\,]]$$

$$\left\{\begin{array}{l}
tp := \{\texttt{labelset}\}; \\
\textit{type}(E) := tu
\end{array}\right.$$

$|\quad K$

$$\left\{\begin{array}{l} tp := \{type(K)\};\\ type(E) := tu \end{array}\right.$$

$|\quad x$

$$\left\{\begin{array}{l} tp := \{type(x)\} \cup \{\texttt{formula} \mid type(x) = \texttt{boolean}\};\\ \textbf{si } tu = \texttt{formula} \textbf{ alors}\\ \quad type(E) := \texttt{bool\_to\_formula}\\ \textbf{sinon}\\ \quad type(E) := \texttt{boolean}\\ \textbf{fin\_si} \end{array}\right.$$

$|\quad Y(E_1 \uparrow tp_1 \downarrow tu_1, \ldots, E_n \uparrow tp_n \downarrow tu_n)$

$$\left\{\begin{array}{l} tp := \{\texttt{formula} \mid \exists p \in profiles(Y) \text{ tel que } p.arg\_types \in \times_{i=1}^{n} tp_i\};\\ type(E) := \texttt{formula};\\ \textbf{si } \{p \in profiles(Y) \mid p.arg\_types \in \times_{i=1}^{n} tp_i\} = \{p\} \textbf{ alors}\\ \quad profile(Y) := p;\\ \quad (tu_1, \ldots, tu_n) := p.arg\_types\\ \textbf{sinon}\\ \quad error(\texttt{"ambiguous argument type(s)"})\\ \textbf{fin\_si} \end{array}\right.$$

$|\quad F(E_1 \uparrow tp_1 \downarrow tu_1, \ldots, E_n \uparrow tp_n \downarrow tu_n)$

$$\left\{\begin{array}{l} \textbf{soit } tp' := \{p.res\_type \mid p \in profiles(F) \ \wedge \ p.arg\_types \in \times_{i=1}^{n} tp_i\} \textbf{ dans}\\ \quad tp := tp' \cup \{\texttt{formula} \mid \texttt{boolean} \in tp'\}\\ \textbf{fin};\\ \textbf{si } tu = \texttt{formula} \textbf{ alors}\\ \quad type(E) := \texttt{bool\_to\_formula}\\ \textbf{sinon}\\ \quad type(E) := tu\\ \textbf{fin\_si};\\ \textbf{si } \{p \in profiles(F) \mid p.arg\_types \in \times_{i=1}^{n} tp_i \wedge\\ \qquad\qquad\qquad (tu = \texttt{formula} \Rightarrow p.res\_type = \texttt{boolean}) \wedge\\ \qquad\qquad\qquad (tu \neq \texttt{formula} \Rightarrow p.res\_type = tu)\} = \{p\}\\ \textbf{alors}\\ \quad profile(F) := p;\\ \quad (tu_1, \ldots, tu_n) := p.arg\_types\\ \textbf{sinon}\\ \quad error(\texttt{"ambiguous argument type(s)"})\\ \textbf{fin\_si} \end{array}\right.$$

$|\quad$ **current**

$$\left\{\begin{array}{l} tp := \{\texttt{state}, \texttt{label}\};\\ type(E) := tu \end{array}\right.$$

$|\quad E_1 \uparrow tp_1 \downarrow tu_1 \ \textbf{.}\ x$

$$\left\{\begin{array}{l} tp := \{type(x) \mid \texttt{state} \in tp_1\};\\ type(E) := tu;\\ tu_1 := \texttt{state} \end{array}\right.$$

| $E_1 \uparrow tp_1 \downarrow tu_1$ **of** RT

$$\left\{\begin{array}{l}
\textbf{soit } tp' := \{type(RT)\} \cap tp_1 \textbf{ dans} \\
\quad tp := tp' \cup \{\texttt{formula} \mid \texttt{boolean} \in tp'\} \\
\textbf{fin}; \\
\textbf{si } tu = \texttt{formula} \textbf{ alors} \\
\quad type(E) := \texttt{bool_to_formula};\ tu_1 := \texttt{boolean} \\
\textbf{sinon} \\
\quad tu_1 := tu \\
\textbf{fin_si}
\end{array}\right.$$

| **nop**

$$\left\{\begin{array}{l}
tp := \{\texttt{void}\}; \\
type(E) := \texttt{void}
\end{array}\right.$$

| $E_1 \uparrow tp_1 \downarrow tu_1$ **;** $E_2 \uparrow tp_2 \downarrow tu_2$

$$\left\{\begin{array}{l}
tp := \{\texttt{void}\} \cap tp_1 \cap tp_2; \\
type(E) := tu_1 := tu_2 := \texttt{void}
\end{array}\right.$$

| **print** $(E_1 \uparrow tp_1 \downarrow tu_1)$

$$\left\{\begin{array}{l}
tp := \{\texttt{void} \mid tp_1 \cap \mathbf{NumType} \neq \emptyset\}; \\
type(E) := \texttt{void}; \\
\textbf{si } tp_1 = \{t\} \textbf{ alors} \\
\quad tu_1 := t \\
\textbf{sinon} \\
\quad error(\texttt{"ambiguous argument type(s)"}) \\
\textbf{fin_si}
\end{array}\right.$$

| $(E_0 \uparrow tp_0 \downarrow tu_0, \ldots, E_n \uparrow tp_n \downarrow tu_n)$

$$\left\{\begin{array}{l}
tp := \times_{i=0}^{n}\{t \in tp_i \mid t \in \mathbf{NumType}\}; \\
type(E) := (tu_0, \ldots, tu_n) := tu
\end{array}\right.$$

| **let** x_0**:**T_0**:=**$E_0 \uparrow tp_0 \downarrow tu_0, \ldots, x_n$**:**$T_n$**:=**$E_n \uparrow tp_n \downarrow tu_n$ **in**
$E' \uparrow tp' \downarrow tu'$
endlet

$$\left\{\begin{array}{l}
tp := tp'; \\
\forall i \in [0, n], \\
\quad \textbf{si } type(T_i) \in tp_i \textbf{ alors} \\
\quad\quad tu_i := type(T_i) \\
\quad \textbf{sinon} \\
\quad\quad error(\texttt{"type of expression uncompatible with the variable"}) \\
\quad \textbf{fin_si}; \\
\textbf{si } tu = \texttt{formula} \wedge \texttt{boolean} \in tp \textbf{ alors} \\
\quad type(E) := \texttt{bool_to_formula};\ tu' := \texttt{boolean} \\
\textbf{sinon} \\
\quad type(E) := tu' := tu \\
\textbf{fin_si}
\end{array}\right.$$

| **let** $(x_0^0$**:**$T_0^0, \ldots, x_0^{n_0}$**:**$T_0^{n_0})$**:=**$E_0 \uparrow tp_0 \downarrow tu_0, \ldots,$
$(x_m^0$**:**$T_m^0, \ldots, x_m^{n_m}$**:**$T_m^{n_m})$**:=**$E_m \uparrow tp_m \downarrow tu_m$ **in**
$E' \uparrow tp' \downarrow tu'$
endlet

$$\left\{\begin{array}{l}
tp := tp';\\
\forall i \in [0,m],\\
\quad \textbf{si } (type(T_i^0),\ldots,type(T_i^{n_i})) \in tp_i \textbf{ alors}\\
\quad\quad tu_i := (type(T_i^0),\ldots,type(T_i^{n_i})) \;\; \textbf{sinon}\\
\quad\quad error(\texttt{"type of expression uncompatible with the variable"})\\
\quad \textbf{fin_si};\\
\textbf{si } tu = \texttt{formula} \wedge \texttt{boolean} \in tp \textbf{ alors}\\
\quad type(E) := \texttt{bool_to_formula};\\
\quad tu' := \texttt{boolean}\\
\textbf{sinon}\\
\quad type(E) := tu' := tu\\
\textbf{fin_si}
\end{array}\right.$$

| **assert** $E_0 \uparrow tp_0 \downarrow tu_0, \ldots, E_n \uparrow tp_0 \downarrow tu_0$ **in**
$E' \uparrow tp' \downarrow tu'$
endassert

$$\left\{\begin{array}{l}
tp := \textbf{si } tp' \neq \{\texttt{formula}\} \textbf{ alors } tp' \textbf{ sinon } \emptyset \textbf{ fin_si};\\
\forall i \in [0,n],\\
\quad \textbf{si } \texttt{boolean} \in tp_i \textbf{ alors}\\
\quad\quad tu_i := \texttt{boolean}\\
\quad \textbf{sinon}\\
\quad\quad error(\texttt{"assertion type must be boolean"})\\
\quad \textbf{fin_si};\\
\textbf{si } tu = \texttt{formula} \wedge \texttt{boolean} \in tp \textbf{ alors}\\
\quad type(E) := \texttt{bool_to_formula};\\
\quad tu' := \texttt{boolean}\\
\textbf{sinon}\\
\quad type(E) := tu' := tu\\
\textbf{fin_si}
\end{array}\right.$$

| **if** $E_0 \uparrow tp_0 \downarrow tu_0$ **then** $E'_0 \uparrow tp'_0 \downarrow tu'_0$
elsif $E_1 \uparrow tp_1 \downarrow tu_1$ **then** $E'_1 \uparrow tp'_1 \downarrow tu'_1$
...
elsif $E_n \uparrow tp_n \downarrow tu_n$ **then** $E'_n \uparrow tp'_n \downarrow tu'_n$
[**else** $E'_{n+1} \uparrow tp'_{n+1} \downarrow tu'_{n+1}$]
endif

$$\left\{\begin{array}{l}
tp := \bigcap_{i=0}^{n+1} tp'_i;\\
\forall i \in [0,n],\\
\quad \textbf{si } \texttt{boolean} \in tp_i \textbf{ alors}\\
\quad\quad tu_i := \texttt{boolean}\\
\quad \textbf{sinon}\\
\quad\quad error(\texttt{"condition type must be boolean"})\\
\quad \textbf{fin_si};\\
\textbf{si } tu = \texttt{formula} \wedge \texttt{boolean} \in tp \textbf{ alors}\\
\quad type(E) := \texttt{bool_to_formula};\\
\quad tu'_0 := \cdots := tu'_{n+1} := \texttt{boolean}\\
\textbf{sinon}\\
\quad type(E) := tu'_0 := \cdots := tu'_{n+1} := tu\\
\textbf{fin_si}
\end{array}\right.$$

$|$ **case** $E_0 \uparrow tp_0 \downarrow tu_0$ **in**
$P_1^0 \uparrow tp_1^0 \downarrow tu_1^0 \quad | \ldots | \; P_1^{n_1} \uparrow tp_1^{n_1} \downarrow tu_1^{n_1}$
$[$**where** $E_1 \uparrow tp_1 \downarrow tu_1] \;\texttt{->}\; E_1' \uparrow tp_1' \downarrow tu_1'$
$\ldots$
$| \; P_m^0 \uparrow tp_m^0 \downarrow tu_m^0 \; | \ldots | \; P_m^{n_m} \uparrow tp_m^{n_m} \downarrow tu_m^{n_m}$
$[$**where** $E_m \uparrow tp_m \downarrow tu_m] \;\texttt{->}\; E_m' \uparrow tp_m' \downarrow tu_m'$
$[|$ **otherwise** $\texttt{->}\; E_{m+1}' \uparrow tp_{m+1}' \downarrow tu_{m+1}']$
endcase

$tp := \bigcap_{i=1}^m tp_i' \left[\cap\, tp_{m+1}'\right]$;
si $tp_0 \cap \bigcap_{i=1}^m \bigcap_{j=0}^{n_i} tp_i^j = \{t\}$ **alors**
$tu_0 := tu_1^0 := \cdots := tu_1^{n_1} := \cdots := tu_m^0 := \cdots := tu_m^{n_m} := t$
sinon
error(`"ambiguous filter type(s)"`)
fin_si;
$\forall i \in [1, m]$,
si `boolean` $\in tp_i$ **alors**
$tu_i :=$ `boolean`
sinon
error(`"guard type must be boolean"`)
fin_si;
si $tu =$ `formula` $\wedge$ `boolean` $\in tp$ **alors**
$type(E) :=$ `bool_to_formula`;
$tu_1' := \cdots := tu_m' \left[:= tu_{m+1}'\right] :=$ `boolean`
sinon
$type(E) := tu_1' := \cdots := tu_m' \left[:= tu_{m+1}'\right] := tu$
fin_si

$|$ **case action** $E_0 \uparrow tp_0 \downarrow tu_0$ **in**
$\alpha_1 \;\; [$**where** $E_1 \uparrow tp_1 \downarrow tu_1] \quad \texttt{->}\; E_1' \uparrow tp_0' \downarrow tu_0'$
$\ldots$
$| \; \alpha_m \;\; [$**where** $E_m \uparrow tp_m \downarrow tu_m] \;\texttt{->}\; E_m' \uparrow tp_m' \downarrow tu_m'$
$[|$ **otherwise** $\texttt{->}\; E_{m+1}' \uparrow tp_{m+1}' \downarrow tu_{m+1}']$
endcase

$$\left\{\begin{array}{l}
tp := \bigcap_{i=1}^{m} tp'_i \left[\cap\, tp'_{m+1}\right];\\
\textbf{si } \texttt{label} \in tp_0 \textbf{ alors}\\
\quad tu_0 := \texttt{label}\\
\textbf{sinon}\\
\quad error(\texttt{"type of expression must be label"})\\
\textbf{fin_si};\\
\forall i \in [1,m],\\
\quad \textbf{si } \texttt{boolean} \in tp_i \textbf{ alors}\\
\quad\quad tu_i := \texttt{boolean}\\
\quad \textbf{sinon}\\
\quad\quad error(\texttt{"guard type must be boolean"})\\
\quad \textbf{fin_si};\\
\textbf{si } tu = \texttt{formula} \wedge \texttt{boolean} \in tp \textbf{ alors}\\
\quad type(E) := \texttt{bool_to_formula};\\
\quad tu'_1 := \cdots := tu'_m \left[:= tu'_{m+1}\right] := \texttt{boolean}\\
\textbf{sinon}\\
\quad type(E) := tu'_1 := \cdots := tu'_m \left[:= tu'_{m+1}\right] := tu\\
\textbf{fin_si}
\end{array}\right.$$

| **loop** $(x_0\textbf{:}T_0\textbf{:=}E_0 \uparrow tp_0 \downarrow tu_0, \ldots, x_n\textbf{:}T_n\textbf{:=}E_n \uparrow tp_n \downarrow tu_n)$ **:** RT **in**

$E' \uparrow tp' \downarrow tu'$

endloop

$$\left\{\begin{array}{l}
\textbf{soit } tp'' := \{type(RT)\} \cap tp' \textbf{ dans}\\
\quad tp := tp'' \cup \{\texttt{formula} \mid \texttt{boolean} \in tp''\}\\
\textbf{fin};\\
\forall i \in [0,n],\\
\quad \textbf{si } type(T_i) \in tp_i \textbf{ alors}\\
\quad\quad tu_i := type(T_i)\\
\quad \textbf{sinon}\\
\quad\quad error(\texttt{"type of expression uncompatible with the variable"})\\
\quad \textbf{fin_si};\\
\textbf{si } tu = \texttt{formula} \textbf{ alors}\\
\quad type(E) := \texttt{bool_to_formula};\\
\quad tu' := \texttt{boolean}\\
\textbf{sinon}\\
\quad type(E) := tu' := tu\\
\textbf{fin_si}
\end{array}\right.$$

| **continue** $(E_0 \uparrow tp_0 \downarrow tu_0, \ldots, E_n \uparrow tp_n \downarrow tu_n)$

$$\left\{\begin{array}{l}
\textbf{soit } tp' := \{p.res_type \mid p = profile(\textbf{continue}) \wedge p.arg_types \in \times_{i=0}^{n} tp_i\}\\
\textbf{dans}\\
\quad tp := tp' \cup \{\texttt{formula} \mid \texttt{boolean} \in tp'\}\\
\textbf{fin};\\
\textbf{si } tu = \texttt{formula} \textbf{ alors}\\
\quad type(E) := \texttt{bool_to_formula}\\
\textbf{sinon}\\
\quad type(E) := tu\\
\textbf{fin_si};\\
(tu_0, \ldots, tu_n) := profile(\textbf{continue}).arg_types
\end{array}\right.$$

| **for** x'_0:T'_0[**among** $E'_0 \uparrow tp'_0 \downarrow tu'_0$], ... ,$x'_m$:$T'_m$[**among** $E'_m \uparrow tp'_m \downarrow tu'_m$]
[**var** x_0:T_0:=$E_0 \uparrow tp_0 \downarrow tu_0$, ... ,$x_n$:$T_n$:=$E_n \uparrow tp_n \downarrow tu_n$]
[**where** $E''_1 \uparrow tp''_1 \downarrow tu''_1$]
[**while** $E''_2 \uparrow tp''_2 \downarrow tu''_2$]
in $E''_3 \uparrow tp''_3 \downarrow tu''_3$
endfor

$$
\left\{
\begin{array}{l}
\textbf{soit } tp' := tp''_3\,[\cap\{(type(T_0),\ldots,type(T_n))\}]\ \textbf{dans}\\
\quad tp := \textbf{si } tp' \neq \{\texttt{formula}\}\ \textbf{alors } tp'\ \textbf{sinon}\ \emptyset\ \textbf{fin\_si};\\
\textbf{fin};\\
\forall i \in [0,m],\\
\quad \left[\begin{array}{l}
\textbf{si } setof(type(T'_i)) \in tp'_i\ \textbf{alors}\\
\quad tu'_i := setof(type(T'_i))\\
\textbf{sinon}\\
\quad error(\texttt{"domain type uncompatible}\\
\qquad\qquad \texttt{with the iteration variable"})\\
\textbf{fin\_si};
\end{array}\right]\\
\left[\begin{array}{l}
\forall j \in [0,n],\\
\quad \textbf{si } type(T_j) \in tp_j\ \textbf{alors}\\
\qquad tu_j := type(T_j)\\
\quad \textbf{sinon}\\
\qquad error(\texttt{"type of expression uncompatible}\\
\qquad\qquad \texttt{with the variable"})\\
\quad \textbf{fin\_si};
\end{array}\right]\\
\left[\begin{array}{l}
\textbf{si } \texttt{boolean} \in tp''_1 \cap tp''_2\ \textbf{alors}\\
\quad tu''_1 := tu''_2 := \texttt{boolean}\\
\textbf{sinon}\\
\quad error(\texttt{"guard type must be boolean"})\\
\textbf{fin\_si};
\end{array}\right]\\
\textbf{si } tu = \texttt{formula}\ \textbf{alors}\\
\quad type(E) := \texttt{bool\_to\_formula};\\
\quad tu''_3 := \texttt{boolean}\\
\textbf{sinon}\\
\quad type(E) := tu;\\
\quad tu''_3 := tu\\
\textbf{fin\_si}
\end{array}
\right.
$$

| **for** x'_0:T'_0[**among** $E'_0 \uparrow tp'_0 \downarrow tu'_0$], ... ,$x'_m$:$T'_m$[**among** $E'_m \uparrow tp'_m \downarrow tu'_m$]
[**var** x_0:T_0:=$E_0 \uparrow tp_0 \downarrow tu_0$, ... ,$x_n$:$T_n$:=$E_n \uparrow tp_n \downarrow tu_n$]
[**where** $E''_1 \uparrow tp''_1 \downarrow tu''_1$]
[**while** $E''_2 \uparrow tp''_2 \downarrow tu''_2$]
in $E''_3 \uparrow tp''_3 \downarrow tu''_3$
result $E''_4 \uparrow tp''_4 \downarrow tu''_4$
endfor

```
{ tp := si tp''_4 ≠ {formula} alors tp''_4 sinon ∅ fin_si;
  ∀i ∈ [0, m],
    [ si setof(type(T'_i)) ∈ tp'_i alors
        tu'_i := setof(type(T'_i))
      sinon
        error("domain type uncompatible
               with the iteration variable")
      fin_si; ]
  [ ∀j ∈ [0, n],
      si type(T_j) ∈ tp_j alors
        tu_j := type(T_j)
      sinon
        error("type of expression uncompatible
               with the variable")
      fin_si; ]
  [ si boolean ∈ tp''_1 ∩ tp''_2 alors
      tu''_1 := tu''_2 := boolean
    sinon
      error("guard type must be boolean")
    fin_si; ]
  si tp''_3 [∩ {(type(T_0), ..., type(T_n))}] = {t} alors
    tu''_3 := t
  sinon
    error("ambiguous type of body expression")
  fin_si;
  si tu = formula alors
    type(E) := bool_to_formula;
    tu''_4 := boolean
  sinon
    type(E) := tu;
    tu''_4 := tu
  fin_si

|  {E_1 ↑ tp_1 ↓ tu_1, ..., E_n ↑ tp_n ↓ tu_n}
{ si n = 0 alors
    tp := {setof(t) | t ∈ {integer, character, state, label, trans}}
  sinon
    tp := {setof(t) | t ∈ {integer, character, state, label, trans} ∩ ⋂_{i=1}^{n} tp_i}
  fin_si;
  type(E) := tu;
  soit t ∈ ExpType tel que setof(t) = tu dans
    tu_1 := ··· := tu_n := t
  fin

|  {E_1 ↑ tp_1 ↓ tu_1 ... E_2 ↑ tp_2 ↓ tu_2}
{ tp := {setof(t) | t ∈ {integer, character} ∩ tp_1 ∩ tp_2};
  type(E) := tu;
  soit t ∈ ExpType tel que setof(t) = tu dans
    tu_1 := tu_2 := t
  fin
```

| { F **on** x_0:T_0 [**among** $E_0 \uparrow tp_0 \downarrow tu_0$], ... ,$x_n$:$T_n$ [**among** $E_n \uparrow tp_n \downarrow tu_n$]
[**where** $E'_1 \uparrow tp'_1 \downarrow tu'_1$] **}** $E'_2 \uparrow tp'_2 \downarrow tu'_2$

soit $tp' := \{t_1 \mid \exists p \in \mathit{profiles}(F), \exists t_2 \in$ **ExpType** $. p.\mathit{arg\_types} = (t_1, t_2) \wedge$
$p.\mathit{res\_type} = t_1 \ \wedge \ t_2 \in tp'_2\}$ **dans**
  $tp := tp' \cup \{$`formula` $\mid$ `boolean` $\in tp'\}$
fin;
$\forall i \in [0, n]$,
  si $\mathit{setof}(\mathit{type}(T_i)) \in tp_i$ **alors**
    $tu_i := \mathit{setof}(\mathit{type}(T_i))$
  sinon
    error("domain type uncompatible
        with the iteration variable")
  fin_si;
si `boolean` $\in tp'_1$ **alors**
  $tu'_1 :=$ `boolean`
sinon
  error("guard type must be boolean")
fin_si;
si $tu =$ `formula` **alors**
  $\mathit{type}(E) :=$ `bool_to_formula`
sinon
  $\mathit{type}(E) := tu$
fin_si;
si $\{p \in \mathit{profiles}(F) \mid p.\mathit{arg\_types} = (t_1, t_2) \wedge t_2 \in tp'_2 \wedge$
  $(tu =$ `formula` $\Rightarrow t_1 =$ `boolean`$) \wedge (tu \neq$ `formula` $\Rightarrow t_1 = tu)\} = \{p\}$
alors
  $\mathit{profile}(F) := p$;
  $tu'_2 := (p.\mathit{arg\_types})_2$
sinon
  error("ambiguous argument type(s)")
fin_si

| { x:T [**among** $E_1 \uparrow tp_1 \downarrow tu_1$] **where** $E_2 \uparrow tp_2 \downarrow tu_2$ **}**

$tp := \{\mathit{setof}(\mathit{type}(T))\}$;
si $tu \in tp_1$ **alors**
  $tu_1 := tu$
sinon
  error("domain type uncompatible
        with the iteration variable")
fin_si;
si `boolean` $\in tp_2$ **alors**
  $tu_2 :=$ `boolean`
sinon
  error("guard type must be boolean")
fin_si;
$\mathit{type}(E) := tu$

Définitions de fonctions

D ::= [**local**] **function** F $(x_1{:}T_1, \ldots, x_n{:}T_n)$ **:** RT **is**
  $E \uparrow tp \downarrow tu$
endfunc
  si $type(RT) \in tp$ **alors**
    $tu := type(RT)$
  sinon
    error("function return type uncompatible with its body")
  fin_si

Programme

PG ::= $[D_0 \ldots D_m]$
  $E \uparrow tp \downarrow tu$
[**where** $D_{m+1} \ldots D_{m+p}$]
  si $tp = \{t\}$ **alors**
    $tu := t$
  sinon
    error("ambiguous type of program body")
  fin_si

A.7.4 Discussion

Durant le typage des expressions et des formules, les quatre propriétés ci-dessous sont invariantes.

1. Pour toute règle syntaxique des symboles non-terminaux E et P ayant les attributs $\uparrow tp \downarrow tu$:

$$tu \in tp.$$

   Ceci est la condition de correction du typage : le type unique calculé pour chaque expression E et pour chaque filtre P doit figurer parmi ses types possibles.

2. Pour toute règle syntaxique de E ayant l'attribut $\uparrow tp$:

$$\texttt{boolean} \in tp \Rightarrow \texttt{formula} \in tp.$$

   Cette propriété signifie que chaque expression ayant `boolean` comme type possible est susceptible de dénoter une formule (prédicat de base).

3. Pour toute règle syntaxique de E ayant l'attribut $\uparrow tp$:

$$(\texttt{formula} \in tp \;\wedge\; \texttt{boolean} \notin tp) \Rightarrow tp = \{\texttt{formula}\}.$$

   Ceci permet d'identifier les expressions E dénotant des formules : si `formula` figure parmi les types possibles de E, mais `boolean` n'y figure pas, alors le type unique de E est `formula`.

4. Pour toute règle syntaxique de E ayant les attributs $\uparrow tp \downarrow tu$:

$$(\texttt{boolean} \in tp \;\wedge\; tu = \texttt{formula}) \Leftrightarrow type(E) = \texttt{bool\_to\_formula}.$$

   Ceci représente la condition de conversion des expressions booléennes vers des formules : une expression E est convertie à une formule ssi `boolean` figure parmi ses types possibles et le contexte de E lui impose le type unique `formula`.

Ces invariants peuvent être prouvés par induction, soit sur la longueur de la dérivation menant de l'axiome à la règle courante (propriétés 1 et 4), soit sur la structure de la grammaire (propriétés 2 et 3), en inspectant chaque règle syntaxique donnée dans la section A.7.3.

Après la phase de typage, la séparation entre formules et expressions peut être effectuée en remplaçant dans l'arbre abstrait toutes les occurrences d'expressions E ayant $type(E)$ = `formula` par des formules φ. Les occurrences de E ayant $type(E)$ = `bool_to_formula` correspondent à la règle syntaxique $\varphi ::= E$ de la grammaire abstraite du langage XTL.

Remarque A-14
Le typage des appels de "**print**" tient compte du fait que cet opérateur est surchargé (voir la section 2.3.1) et que son argument ne peut pas être de type tuple (voir la section 2.3.3). ■

A.8 Vérifications statiques complémentaires

Plusieurs vérifications statiques sont encore nécessaires afin d'assurer que certaines conditions imposées par la sémantique dynamique des formules et des expressions régulières XTL (voir les sections 3.5, 3.6 et 3.7) sont satisfaites. De la même manière que les phases d'analyse décrites dans les sections précédentes, ces vérifications peuvent être décrites au moyen de grammaires attribuées. Cependant, étant donné leur caractère simple (elles ne nécessitent pas le calcul d'attributs locaux, mais uniquement la propagation d'attributs synthétisés et hérités), nous nous contentons de les présenter informellement. Les vérifications statiques complémentaires peuvent être groupées en plusieurs classes :

Visibilité des méta-opérateurs "current" sur états. Ces méta-opérateurs (assimilés à des fonctions de type `state` sans arguments) permettent d'accéder, dans une formule φ, à l'état courant s sur lequel φ est évaluée (voir la section 2.10.2). Par conséquent, toutes les occurrences des méta-opérateurs "**current**" sur états doivent être contenues dans des formules φ. Cette vérification peut être effectuée au moyen d'un attribut booléen hérité $\downarrow$ *in_state_formula*, associé à tous les symboles N de la grammaire, qui indique si N est contenu dans une formule φ ou non. Une erreur est signalée pour chaque occurrence de méta-opérateur "**current** $\downarrow$ *in_state_formula*" telle que *type*(**current**) = `state` et *in_state_formula* = **ff**.

Visibilité des méta-opérateurs "current" sur actions. Ces méta-opérateurs (assimilés à des fonctions de type `label` sans arguments) permettent d'accéder à l'action courante sur laquelle une formule modale est évaluée. Toutes les occurrences des méta-opérateurs "**current**" sur actions doivent être contenues dans des formules α et/ou dans des formules φ précédées par des modalités "$\langle R \rangle$" ou "$[R]$" dont l'expression régulière R exporte la dernière action de la séquence de transitions sur laquelle R a été évaluée (voir la section 3.6). Cette vérification peut être effectuée au moyen d'un attribut synthétisé $\uparrow$ *exports_a*, associé aux expressions régulières R, indiquant si R exporte une action, et d'un attribut hérité $\downarrow$ *visible_a*, associé aux formules et aux expressions, indiquant si une action courante est visible ou non. Une erreur est signalée pour chaque occurrence de méta-opérateur "**current** $\downarrow$ *visible_a*" telle que *type*(**current**) = `label` et *visible_a* = **ff**.

Visibilité des méta-opérateurs "|=" et "[[...]]". Pour des raisons de simplicité, dans la version courante du langage XTL ces méta-opérateurs d'évaluation des formules ne peuvent pas être contenus dans les formules elles-mêmes (voir la remarque 2-15). Cette vérification peut être effectuée au moyen d'un attribut hérité $\downarrow$ *in_formula*, associé à tous les symboles N de la grammaire, qui indique si un symbole est contenu ou non dans une formule φ ou α. Une erreur est signalée pour chaque occurrence de méta-opérateur "... |= ... $\downarrow$ *in_formula*" ou "[[...]] $\downarrow$ *in_formula*" telle que *in_formula* = **tt**.

Identification des formules d'alternance 1. Cette phase consiste à implémenter la définition 4-1 des formules φ d'alternance 1. Ceci peut être effectué au moyen d'un attribut hérité $last\_sign$, associé aux formules φ, qui mémorise le signe de la variable Y définie par la plus petite formule de point fixe contenant φ. Une formule φ est d'alternance 1 ssi chaque occurrence d'utilisation de variable propositionnelle "$Y \downarrow last\_sign$" contenue dans φ vérifie $sign(Y) = last\_sign$.

Caractérisation des programmes XTL évaluables à la volée. Cette phase consiste à implémenter la définition 4-2 caractérisant les programmes XTL évaluables à la volée. Ceci ne nécessite la propagation d'aucun attribut synthétisé ou hérité, mais uniquement la comparaison structurelle d'expressions E, qui peut être implémentée de manière directe à partir de l'arbre abstrait. Utilisée conjointement avec l'identification des formules d'alternance 1 contenues dans le programme, cette information est utile pour choisir la stratégie d'évaluation des formules (voir la section 4.4).

Annexe B

Sémantique dénotationnelle des expressions

La sémantique des formules XTL a été définie, sous forme dénotationnelle, au chapitre 3. Par souci de complétude, nous présentons dans cette annexe la sémantique dénotationnelle des expressions XTL.

B.1 Expressions

Tout au long de cette annexe, nous supposons que toutes les phases d'analyse statique ont été effectuées (voir l'annexe A) : en particulier, tous les objets ont des noms uniques et tous les attributs statiques (notamment l'information de type, dénotée par l'attribut *type*) ont été calculés.

La syntaxe abstraite des expressions XTL considérée ici comporte quelques différences mineures par rapport à celle présentée à la section 2.1 :

- les appels infixés de fonctions "$E_1\ F\ E_2$" ont été remplacés par leurs équivalents préfixés "$F\ (E_1,E_2)$" ;
- les expressions "E **of** RT", qui servent uniquement à la résolution des surcharges, ont été éliminées ;
- les expressions "**current**" (sur états ou sur actions), qui ne peuvent apparaître que dans les formules, ont été éliminées ;
- les méta-opérateurs d'évaluation des formules ne sont pas traités dans cette annexe, car leur sémantique a déjà été définie à la section 4.4.

Les quatre simplifications énoncées ci-dessus n'ont aucune influence sur la sémantique dénotationnelle des expressions XTL.

B.1.1 Aspects syntaxiques

Les éléments syntaxiques associés aux expressions E sont définis par les fonctions suivantes (qui ont été mentionnées informellement à la section 3.2) :

$$fdv, bdv : Exp \rightarrow 2^{DVar}$$

E	$fdv(E)$
K, **true**, **false**, **nop**	$\emptyset$
not E, **print** (E), E **.** x	$fdv(E)$
E_1 **bool_op** E_2, E_1 **;** E_2	$fdv(E_1) \cup fdv(E_2)$
$(E_0,\ldots,E_n)$, **continue** $(E_0,\ldots,E_n)$	$\bigcup_{i=0}^{n} fdv(E_i)$
$\{E_1,\ldots,E_n\}$, F $(E_1,\ldots,E_n)$	$\bigcup_{i=1}^{n} fdv(E_i)$
x	$\{x\}$
let x_0**:**T_0**:=**$E_0,\ldots,x_n$**:**T_n**:=**E_n **in** E **endlet**	$(fdv(E) \setminus \bigcup_{i=0}^{n}\{x_i\}) \cup$ $\bigcup_{i=0}^{n} fdv(E_i)$
let $(x_0^0$**:**$T_0^0,\ldots,x_0^{n_0}$**:**$T_0^{n_0})$**:=**$E_0,\ldots,$ $(x_m^0$**:**$T_m^0,\ldots,x_m^{n_m}$**:**$T_m^{n_m})$**:=**E_m **in** E **endlet**	$(fdv(E) \setminus \bigcup_{i=0}^{m} \bigcup_{j=0}^{n_i}\{x_i^j\}) \cup$ $\bigcup_{i=0}^{m} fdv(E_i)$
case E_0 **in** $P_1^0 \mid \ldots \mid P_1^{n_1}$ [**where** E_1] -> E'_1 $\ldots$ $\mid P_m^0 \mid \ldots \mid P_m^{n_m}$ [**where** E_m] -> E'_m $[\mid$ **otherwise** -> $E'_{m+1}]$ **endcase**	$((\bigcup_{i=1}^{m} fdv(E_i) \cup \bigcup_{j=1}^{m+1} fdv(E'_j)) \setminus$ $\bigcup_{k=1}^{m} bdv(P_k^0)) \cup fdv(E_0)$
case action E_0 **in** α_1 [**where** E_1] -> E'_1 $\ldots$ $\mid \alpha_m$ [**where** E_m] -> E'_m $[\mid$ **otherwise** -> $E'_{m+1}]$ **endcase**	$((\bigcup_{i=1}^{m} fdv(E_i) \cup \bigcup_{j=1}^{m+1} fdv(E'_j)) \setminus$ $\bigcup_{k=1}^{m} v_{tt}(\alpha_k)) \cup fdv(E_0)$
if E_0 **then** E'_0 **elsif** E_1 **then** E'_1 $\ldots$ **elsif** E_n **then** E'_n [**else** E'_{n+1}] **endif**	$\bigcup_{i=0}^{n+1} fdv(E'_i) \cup \bigcup_{j=0}^{n} fdv(E_j)$
assert $E_0,\ldots,E_n$ **in** E **endassert**	$fdv(E) \cup \bigcup_{i=0}^{n} fdv(E_i)$
loop $(x_0$**:**T_0**:=**$E_0,\ldots,x_n$**:**T_n**:=**$E_n)$ **:** RT **in** E **endloop**	$(fdv(E) \setminus \bigcup_{i=0}^{n}\{x_i\}) \cup \bigcup_{i=0}^{n} fdv(E_i)$
for x'_0**:**T'_0[**among** E'_0]$,\ldots,x'_m$**:**T'_m[**among** E'_m] [**var** x_0**:**T_0**:=**$E_0,\ldots,x_n$**:**T_n**:=**E_n] [**where** E''_1] [**while** E''_2] **in** E''_3 [**result** E''_4] **endfor**	$(\bigcup_{i=0}^{m} fdv(E'_i) \cup \bigcup_{j=0}^{n} fdv(E_j) \cup$ $\bigcup_{k=1}^{4} fdv(E''_k)) \setminus$ $(\bigcup_{i=0}^{m}\{x'_i\} \cup \bigcup_{j=0}^{n}\{x_j\})$
{ F **on** x_0**:**T_0 [**among** E_0]$,\ldots,$ x_n**:**T_n [**among** E_n] [**where** E'_1] **}** E'_2	$(\bigcup_{i=0}^{n} fdv(E_i) \cup fdv(E'_1) \cup fdv(E'_2)) \setminus$ $\bigcup_{j=0}^{n}\{x_j\}$
{ x**:**T [**among** E_1] **where** E_2 **}**	$(fdv(E_1) \cup fdv(E_2)) \setminus \{x\}$
quantif x_0**:**T_0 [**among** E_0]$,\ldots,$ x_n**:**T_n [**among** E_n] **in** E	$(\bigcup_{i=0}^{n} fdv(E_i) \cup fdv(E)) \setminus \bigcup_{j=0}^{n}\{x_j\}$

Table B.1: Variables libres dans les expressions

E	$bdv(E)$
K, **true**, **false**, **nop**	$\emptyset$
not E, **print** (E), E **.** x	$bdv(E)$
E_1 **bool_op** E_2, E_1 **;** E_2	$bdv(E_1) \cup bdv(E_2)$
$(E_0,\ldots,E_n)$, **continue** $(E_0,\ldots,E_n)$	$\bigcup_{i=0}^{n} bdv(E_i)$
$\{E_1,\ldots,E_n\}$, $F\ (E_1,\ldots,E_n)$	$\bigcup_{i=1}^{n} bdv(E_i)$
x	$\emptyset$
let x_0**:**T_0**:=**$E_0,\ldots,x_n$**:**T_n**:=**E_n **in** E **endlet**	$\bigcup_{i=0}^{n}\{x_i\} \cup \bigcup_{i=0}^{n} bdv(E_i) \cup$ $bdv(E)$
let $(x_0^0$**:**$T_0^0,\ldots,x_0^{n_0}$**:**$T_0^{n_0})$**:=**$E_0,\ldots,$ $(x_m^0$**:**$T_m^0,\ldots,x_m^{n_m}$**:**$T_m^{n_m})$**:=**E_m **in** E **endlet**	$\bigcup_{i=0}^{m}\bigcup_{j=0}^{n_i}\{x_i^j\} \cup$ $\bigcup_{i=0}^{m} bdv(E_i) \cup bdv(E)$
case E_0 **in** $P_1^0 \mid \ldots \mid P_1^{n_1}$ [**where** E_1] **->** E'_1 $\ldots$ $\mid P_m^0 \mid \ldots \mid P_m^{n_m}$ [**where** E_m] **->** E'_m [| **otherwise** **->** E'_{m+1}] **endcase**	$\bigcup_{i=1}^{m} bdv(E_i) \cup \bigcup_{j=1}^{m+1} bdv(E'_j) \cup$ $\bigcup_{k=1}^{m} bdv(P_k^0) \cup bdv(E_0)$
case action E_0 **in** α_1 [**where** E_1] **->** E'_1 $\ldots$ $\mid \alpha_m$ [**where** E_m] **->** E'_m [| **otherwise** **->** E'_{m+1}] **endcase**	$\bigcup_{i=1}^{m} bdv(E_i) \cup \bigcup_{j=1}^{m+1} bdv(E'_j) \cup$ $\bigcup_{k=1}^{m} v_{tt}(\alpha_k) \cup bdv(E_0)$
if E_0 **then** E'_0 **elsif** E_1 **then** E'_1 $\ldots$ **elsif** E_n **then** E'_n [**else** E'_{n+1}] **endif**	$\bigcup_{i=0}^{n+1} bdv(E'_i) \cup \bigcup_{j=0}^{n} bdv(E_j)$
assert $E_0,\ldots,E_n$ **in** E **endassert**	$bdv(E) \cup \bigcup_{i=0}^{n} bdv(E_i)$
loop $(x_0$**:**T_0**:=**$E_0,\ldots,x_n$**:**T_n**:=**$E_n)$ **:** RT **in** E **endloop**	$bdv(E) \cup \bigcup_{i=0}^{n}\{x_i\} \cup \bigcup_{i=0}^{n} bdv(E_i)$
for x'_0**:**T'_0[**among** E'_0]$,\ldots,x'_m$**:**T'_m[**among** E'_m] [**var** x_0**:**T_0**:=**$E_0,\ldots,x_n$**:**T_n**:=**E_n] [**where** E''_1] [**while** E''_2] **in** E''_3 [**result** E''_4] **endfor**	$\bigcup_{i=0}^{m} bdv(E'_i) \cup \bigcup_{j=0}^{n} bdv(E_j) \cup$ $\bigcup_{k=1}^{4} bdv(E''_k) \cup$ $\bigcup_{i=0}^{m}\{x'_i\} \cup \bigcup_{j=0}^{n}\{x_j\}$
{ F **on** x_0**:**T_0 [**among** E_0]$,\ldots,$ x_n**:**T_n [**among** E_n] [**where** E'_1] **}** E'_2	$\bigcup_{i=0}^{n} bdv(E_i) \cup bdv(E'_1) \cup bdv(E'_2) \cup$ $\bigcup_{j=0}^{n}\{x_j\}$
{ x**:**T [**among** E_1] **where** E_2 **}**	$bdv(E_1) \cup bdv(E_2) \cup \{x\}$
quantif x_0**:**T_0 [**among** E_0]$,\ldots,$ x_n**:**T_n [**among** E_n] **in** E	$\bigcup_{i=0}^{n} bdv(E_i) \cup bdv(E) \cup \bigcup_{j=0}^{n}\{x_j\}$

Table B.2: Variables liées dans les expressions

Pour une expression E, les dénotations $fdv(E)$ et $bdv(E)$ renvoient respectivement l'ensemble des variables simples libres et liées dans E. Ces fonctions syntaxiques sont définies inductivement dans les tables B.1 et B.2 (dans ces tables, **bool_op** et **quantif** dénotent respectivement les opérateurs booléens binaires et les quantificateurs, et la fonction $v_{tt}(\alpha)$, qui a été définie à la section 3.5.1, renvoie les variables exportées par α).

Outre les domaines syntaxiques définis à la section 3.1.1, nous introduisons aussi le domaine *Func* des identificateurs de fonctions (symbole terminal F) XTL ou BCG et le domaine *Const* des constantes littérales (symbole terminal K) contenues dans les expressions XTL.

B.1.2 Aspects sémantiques

L'évaluation d'une expression XTL peut avoir trois effets : (1) renvoyer une valeur (comme c'est le cas des opérateurs booléens "**and**", "**or**" ou "**not**"), (2) imprimer une valeur sur le fichier de sortie (comme c'est le cas de l'opérateur "**print**") et (3) arrêter le programme, avec impression d'un message approprié, si une erreur se produit (comme c'est le cas des expressions "**assert**", "**if**" et "**case**"). Afin de décrire de manière naturelle ces différents effets, et à la différence du style direct illustré par les fonctions sémantiques définies au chapitre 3, nous avons choisi dans cette annexe un style de sémantique dénotationnelle avec des continuations [Sch88].

Outre les domaines sémantiques définis à la section 3.1.2, nous utiliserons aussi les domaines suivants :

- $\mathbf{OutFile} \stackrel{\mathrm{d}}{=} \mathbf{Val}^*$ est le domaine des *fichiers de sortie*. Un fichier de sortie $\omega \in \mathbf{OutFile}$ dénote une séquence (éventuellement vide) de valeurs typées. Le domaine **OutFile** est muni des opérations $nil :\to \mathbf{OutFile}$, qui renvoie un fichier vide, et $append : \mathbf{OutFile} \times \mathbf{Val} \to \mathbf{OutFile}$, qui rajoute une valeur à la fin d'un fichier.

- $\mathbf{ECont} \stackrel{\mathrm{d}}{=} \mathbf{Val} \to \mathbf{OutFile} \to \mathbf{Val} \times \mathbf{OutFile}$ est le domaine des *continuations*. Une continuation $\kappa \in \mathbf{ECont}$ représente l'effet de l'exécution d'un programme XTL à partir d'un certain point[19]. Une continuation prend comme arguments la valeur et le fichier de sortie produits par l'exécution du programme jusqu'au point respectif et renvoie la valeur et le fichier de sortie produits après l'exécution de tout le programme.

- $\mathbf{FEnv} \stackrel{\mathrm{d}}{=} Func \to \mathbf{Param} \to \mathbf{ECont} \to \mathbf{OutFile} \to \mathbf{Val} \times \mathbf{OutFile}$ est le domaine des *environnements de fonctions*. Un environnement $\eta \in \mathbf{FEnv}$ associe à chaque identificateur de fonction F contenu dans $supp(\eta)$ une fonction sémantique $\eta(F)$ ayant trois paramètres : (1) un tuple de valeurs $(v_1, \ldots, v_n)$ de **Param**, (2) une continuation κ représentant la partie du programme qui reste à évaluer *après* l'appel $F(v_1, \ldots, v_n)$ et (3) un fichier ω produit par l'évaluation de la partie du programme située *avant* l'appel $F(v_1, \ldots, v_n)$. La fonction $\eta(F)$ renvoie comme résultat la valeur et le fichier produits après l'exécution du programme.

Remarque B-1
En toute rigueur (voir aussi la remarque 3-1 à la section 3.1.2), les domaines sémantiques D doivent être munis d'une structure d'ordre partiel complet (*cpo*). Cette structure est habituellement obtenue en leur rajoutant un plus petit élément $\perp$, à l'aide de la construction $D_\perp$. Cependant, afin d'alléger les notations, nous considérons la présence de $\perp$ implicite et nous écrirons D à la place de $D_\perp$. ■

Outre les constructions "*let*" et "*if-then-else*" (utilisées aussi dans les fonctions sémantiques définies au chapitre 3), nous utiliserons l'opérateur $lfp : (D \to D) \to D$ dénotant le plus petit point fixe d'une fonctionnelle définie sur le domaine D. Les fonctions sémantiques définies sont supposées *strictes*

[19] En termes d'arbres syntaxiques, une continuation représente l'effet de l'évaluation de la portion du programme XTL située à l'extérieur d'une certaine sous-expression.

(voir la remarque 3-2). Tout au long de cette section, nous considérons un modèle STE étendu $\mathcal{M} = (S, val_S, A, val_A, T, s_{init})$ (voir la définition 1-4) utilisé comme argument implicite des fonctions sémantiques.

La sémantique des expressions XTL est définie au moyen de la fonction d'interprétation suivante :

$$[\![.]\!] : Exp \rightarrow \mathbf{FEnv} \rightarrow \mathbf{DEnv} \rightarrow \mathbf{ECont} \rightarrow \mathbf{OutFile} \rightarrow \mathbf{Val} \times \mathbf{OutFile}$$

Etant donné une expression E, un environnement $\eta \in \mathbf{FEnv}$ (qui doit initialiser toutes les fonctions appelées dans E), un environnement $\varepsilon \in \mathbf{DEnv}$ (tel que $fdv(E) \subseteq supp(\varepsilon)$), une continuation $\kappa \in \mathbf{ECont}$ (qui représente la partie du programme à évaluer *après* E) et un fichier $\omega \in \mathbf{OutFile}$ (produit par l'évaluation de la partie du programme située *avant* E), la dénotation $[\![E]\!]\,\eta\varepsilon\kappa\omega$ renvoie un tuple contenant la valeur et le fichier de sortie obtenus après l'exécution du programme. La valeur et le fichier produits par l'évaluation de E sont passés en paramètre à la continuation κ.

Remarque B-2
Pour simplifier la présentation, dans la section 3.2 nous avons utilisé une fonction d'interprétation des expressions ayant un profil différent de celle introduite ci-dessus (les paramètres η, κ et ω étant omis). Ceci est justifié par le fait que les expressions E contenues dans les formules α ou φ satisfont les conditions suivantes :

- elles ne contiennent pas de définitions de fonctions, donc l'environnement η (contenant les fonctions définies dans le programme XTL) n'est pas modifié au cours de l'évaluation de E ;
- elles ne produisent aucun effet sur le contenu du fichier de sortie, car l'ordre de leur évaluation n'est pas spécifié (cet ordre étant dépendant de l'algorithme d'évaluation des formules).

Plus précisément, le lien entre les deux fonctions sémantiques est exprimé par l'égalité suivante :

$$[\![E]\!]\,\varepsilon = ([\![E]\!]\,\eta\varepsilon(\lambda v_1.\lambda\omega_1.(v_1,\omega))\omega)_1$$

où η contient toutes les fonctions définies dans le programme XTL et ω dénote le fichier de sortie produit par l'exécution du programme avant de commencer l'évaluation de E. La continuation $\lambda v_1.\lambda\omega_1.(v_1,\omega)$ utilisée assure que, tout au long de l'évaluation de E, le fichier de sortie ω restera inchangé. ■

La fonction sémantique associée aux expressions XTL est définie inductivement dans les paragraphes suivants. Afin d'accroître la clarté de la présentation, des exemples d'évaluation de différentes expressions sont également fournis.

Constante littérale

$$[\![K]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \kappa([\![K]\!])(\omega)$$

Remarque B-3
Nous ne définissons pas ici la fonction $[\![.]\!] : Const \rightarrow \mathbf{Val}$ d'interprétation des constantes littérales. Dans le compilateur XTL, la valeur $[\![K]\!]$ est calculée directement à partir de la représentation lexicale de K. ■

Opérateurs booléens

$$[\![\mathbf{true}]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \kappa(\mathbf{tt})(\omega)$$

$$[\![\mathbf{false}]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \kappa(\mathbf{ff})(\omega)$$

$$\llbracket \textbf{not } E \rrbracket \eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \llbracket E \rrbracket \eta\varepsilon(\lambda b_1.\lambda\omega_1.\kappa(not(b_1))(\omega_1))(\omega)$$

$$\begin{array}{l}\llbracket E_1 \textbf{ or } E_2 \rrbracket \eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \llbracket E_1 \rrbracket \eta\varepsilon(\\ \quad \lambda b_1.\lambda\omega_1.\llbracket E_2 \rrbracket \eta\varepsilon(\\ \quad\quad \lambda b_2.\lambda\omega_2.\kappa(b_1 \ or \ b_2)(\omega_2) \\ \quad)(\omega_1) \\ )(\omega)\end{array}$$

$$\begin{array}{l}\llbracket E_1 \textbf{ and } E_2 \rrbracket \eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \llbracket E_1 \rrbracket \eta\varepsilon(\\ \quad \lambda b_1.\lambda\omega_1.\llbracket E_2 \rrbracket \eta\varepsilon(\\ \quad\quad \lambda b_2.\lambda\omega_2.\kappa(b_1 \ and \ b_2)(\omega_2) \\ \quad)(\omega_1) \\ )(\omega)\end{array}$$

$$\begin{array}{l}\llbracket E_1 \textbf{ implies } E_2 \rrbracket \eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \llbracket E_1 \rrbracket \eta\varepsilon(\\ \quad \lambda b_1.\lambda\omega_1.\llbracket E_2 \rrbracket \eta\varepsilon(\\ \quad\quad \lambda b_2.\lambda\omega_2.\kappa(b_1 \ implies \ b_2)(\omega_2) \\ \quad)(\omega_1) \\ )(\omega)\end{array}$$

$$\begin{array}{l}\llbracket E_1 \textbf{ iff } E_2 \rrbracket \eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \llbracket E_1 \rrbracket \eta\varepsilon(\\ \quad \lambda b_1.\lambda\omega_1.\llbracket E_2 \rrbracket \eta\varepsilon(\\ \quad\quad \lambda b_2.\lambda\omega_2.\kappa(b_1 \ iff \ b_2)(\omega_2) \\ \quad)(\omega_1) \\ )(\omega)\end{array}$$

$$\begin{array}{l}\llbracket E_1 \textbf{ xor } E_2 \rrbracket \eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \llbracket E_1 \rrbracket \eta\varepsilon(\\ \quad \lambda b_1.\lambda\omega_1.\llbracket E_2 \rrbracket \eta\varepsilon(\\ \quad\quad \lambda b_2.\lambda\omega_2.\kappa(b_1 \ xor \ b_2)(\omega_2) \\ \quad)(\omega_1) \\ )(\omega)\end{array}$$

où $b_1, b_2 \in$ **Bool** et *not*, *or*, *and*, *implies*, *iff* et *xor* sont les opérations usuelles sur le domaine **Bool**. L'évaluation des sous-expressions est modélisée au moyen de continuations imbriquées.

Exemple B-1
Conformément aux définitions ci-dessus, l'expression booléenne "**false or true**" est évaluée dans le contexte de η, ε, κ et ω de la manière suivante :

$$\begin{array}{ll}\llbracket \textbf{false or true} \rrbracket \eta\varepsilon\kappa\omega = & \text{par interprétation de “}\textbf{or}\text{”} \\ \llbracket \textbf{false} \rrbracket \eta\varepsilon(& \\ \quad \lambda b_1.\lambda\omega_1.\llbracket \textbf{true} \rrbracket \eta\varepsilon & \\ \quad\quad \lambda b_2.\lambda\omega_2.\kappa(b_1 \ or \ b_2)(\omega_2) & \\ \quad)(\omega_1) & \\ )(\omega) = & \text{par interprétation de “}\textbf{false}\text{”} \\ (\lambda b_1.\lambda\omega_1.\llbracket \textbf{true} \rrbracket \eta\varepsilon(& \\ \quad \lambda b_2.\lambda\omega_2.\kappa(b_1 \ or \ b_2)(\omega_2) & \\ )(\omega_1))(\textbf{ff})(\omega) = & \end{array}$$

$[\![\mathbf{true}]\!]\,\eta\varepsilon($
$\quad\lambda b_2.\lambda\omega_2.\kappa(\mathbf{ff}\ or\ b_2)(\omega_2)$
$)(\omega) =$ par interprétation de "**true**"
$(\lambda b_2.\lambda\omega_2.\kappa(\mathbf{ff}\ or\ b_2)(\omega_2))(\mathbf{tt})(\omega) =$
$\kappa(\mathbf{ff}\ or\ \mathbf{tt})(\omega) =$
$\kappa(\mathbf{tt})(\omega)$

La valeur de l'expression booléenne est passée à la continuation du programme ; par contre, le fichier de sortie reste inchangé. ■

Opérateur "nop"

$$[\![\mathbf{nop}]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \kappa(nop)(\omega)$$

où *nop* est le constructeur unique du type `void` (voir la section 2.3.1).

Opérateur ";"

$$\begin{aligned}[\![E_1\ ;\ E_2]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=}\ & [\![E_1]\!]\,\eta\varepsilon(\\ & \quad \lambda v_1.\lambda\omega_1.\,[\![E_2]\!]\,\eta\varepsilon(\\ & \quad\quad \lambda v_2.\lambda\omega_2.\kappa(a_2)(\omega_2) \\ & \quad)(\omega_1) \\ &)(\omega)\end{aligned}$$

où $v_1, v_2 \in$ `void`.

Opérateur "print"

$$[\![\mathbf{print}\ (E)]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} [\![E]\!]\,\eta\varepsilon(\lambda v.\lambda\omega'.\kappa(nop)(append(\omega', v)))(\omega)$$

où $v \in type(E)$. L'exemple suivant montre l'effet de la composition séquentielle des appels de "**print**" au moyen de l'opérateur "**;**".

Exemple B-2
L'expression "**print (1) ; print (2)**" est évaluée dans le contexte de η, ε, κ et ω comme suit :

$[\![\mathbf{print}\ (1)\ ;\ \mathbf{print}\ (2)]\!]\,\eta\varepsilon\kappa\omega =$ par interprétation de "**;**"
$[\![\mathbf{print}\ (1)]\!]\,\eta\varepsilon($
$\quad\lambda a_1.\lambda\omega_1.\,[\![\mathbf{print}\ (2)]\!]\,\eta\varepsilon($
$\quad\quad\lambda a_2.\lambda\omega_2.\kappa(a_2)(\omega_2)$
$\quad)(\omega_1)$
$)(\omega) =$ par interprétation de "**print**"
$(\lambda v.\lambda\omega'.($ et de K
$\quad(\lambda a_1.\lambda\omega_1.\,[\![\mathbf{print}\ (2)]\!]\,\eta\varepsilon($
$\quad\quad\lambda a_2.\lambda\omega_2.\kappa(a_2)(\omega_2)$
$\quad)(\omega_1))(nop)(append(\omega', v))$
$))(1)(\omega) =$

$$
\begin{array}{ll}
[\![\textbf{print }(2)]\!]\,\eta\varepsilon(& \\
\quad \lambda a_2.\lambda\omega_2.\kappa(a_2)(\omega_2) & \\
)(\mathit{append}(\omega,1)) = & \text{par interprétation de “}\textbf{print}\text{”} \\
\lambda v.\lambda\omega'.(& \text{et de } K \\
\quad (\lambda a_2.\lambda\omega_2.\kappa(a_2)(\omega_2))(\mathit{nop})(\mathit{append}(\omega',v)) & \\
)(2)(\mathit{append}(\omega,1)) = & \\
(\lambda a_2.\lambda\omega_2.\kappa(a_2)(\omega_2))(\mathit{nop})(\mathit{append}(\mathit{append}(\omega,1),2)) = & \\
\kappa(\mathit{nop})(\mathit{append}(\mathit{append}(\omega,1),2)) &
\end{array}
$$

La valeur de l'expression est égale à *nop* et les valeurs 1 et 2 ont été rajoutées au fichier de sortie qui est passé à la continuation du programme. ■

Tuple

$$
\begin{array}{lll}
[\![(E_0,\ldots,E_n)]\!]\,\eta\varepsilon\kappa\omega & \stackrel{\mathrm{d}}{=} & [\![E_0]\!]\,\eta\varepsilon(\\
& & \quad \lambda v_0.\lambda\omega_0.\,[\![E_1]\!]\,\eta\varepsilon(\\
& & \quad\quad \ldots \\
& & \quad\quad \lambda v_{n-1}.\lambda\omega_{n-1}.\,[\![E_n]\!]\,\eta\varepsilon(\\
& & \quad\quad\quad \lambda v_n.\lambda\omega_n.\kappa((v_0,\ldots,v_n))(\omega_n) \\
& & \quad\quad)(\omega_{n-1}) \\
& & \quad\quad \ldots \\
& & \quad)(\omega_0) \\
& &)(\omega)
\end{array}
$$

Ensemble

$$
\begin{array}{lll}
[\![\{E_1,\ldots,E_n\}]\!]\,\eta\varepsilon\kappa\omega & \stackrel{\mathrm{d}}{=} & [\![E_1]\!]\,\eta\varepsilon(\\
& & \quad \lambda v_1.\lambda\omega_1.\,[\![E_2]\!]\,\eta\varepsilon(\\
& & \quad\quad \ldots \\
& & \quad\quad \lambda v_{n-1}.\lambda\omega_{n-1}.\,[\![E_n]\!]\,\eta\varepsilon(\\
& & \quad\quad\quad \lambda v_n.\lambda\omega_n.\kappa(\{v_1,\ldots,v_n\})(\omega_n) \\
& & \quad\quad)(\omega_{n-1}) \\
& & \quad\quad \ldots \\
& & \quad)(\omega_1) \\
& &)(\omega)
\end{array}
$$

Sous-domaine

$$
\begin{array}{lll}
[\![\{E_1\ \ldots\ E_2\}]\!]\,\eta\varepsilon\kappa\omega & \stackrel{\mathrm{d}}{=} & [\![E_1]\!]\,\eta\varepsilon(\\
& & \quad \lambda v_1.\lambda\omega_1.\,[\![E_2]\!]\,\eta\varepsilon(\\
& & \quad\quad \lambda v_2.\lambda\omega_2.\kappa(\mathit{domain}(v_1,v_2))(\omega_2) \\
& & \quad)(\omega_1) \\
& &)(\omega)
\end{array}
$$

où $\mathit{domain}(v_1, v_2)$ représente le sous-domaine du type des expressions E_1 et E_2 (qui est supposé énumérable — voir la section 2.6.2) contenant les valeurs v telles que $v_1 \leq v \leq v_2$.

Appel de fonction

$$
\begin{array}{l}
[\![F\ \mathbf{(}E_1\mathbf{,}\ldots\mathbf{,}E_n\mathbf{)}]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} [\![E_1]\!]\,\eta\varepsilon(\\
\quad \lambda v_1.\lambda\omega_1.\,[\![E_2]\!]\,\eta\varepsilon(\\
\quad\quad \ldots \\
\quad\quad \lambda v_{n-1}.\lambda\omega_{n-1}.\,[\![E_n]\!]\,\eta\varepsilon(\\
\quad\quad\quad \lambda v_n.\lambda\omega_n.(\eta(F))(v_1,\ldots,v_n)(\kappa)(\omega_n) \\
\quad\quad)(\omega_{n-1}) \\
\quad\quad \ldots \\
\quad)(\omega_1) \\
)(\omega)
\end{array}
$$

Variable XTL

$$[\![x]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \kappa(\varepsilon(x))(\omega)$$

Variable BCG

$$[\![E\ \mathbf{.}\ x]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} [\![E]\!]\,\eta\varepsilon(\lambda s.\lambda\omega'.\kappa((val_S(s))(x))(\omega'))(\omega)$$

où $s \in S$.

Expressions "let"

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l}
\mathbf{let}\ x_0\mathbf{:}T_0\mathbf{:=}E_0\mathbf{,}\ldots\mathbf{,}x_n\mathbf{:}T_n\mathbf{:=}E_n\ \mathbf{in} \\
\quad E \\
\mathbf{endlet}
\end{array}\right]\!\!\right]\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \\
[\![E_0]\!]\,\eta\varepsilon(\\
\quad \lambda v_0.\lambda\omega_0.\,[\![E_1]\!]\,\eta\varepsilon(\\
\quad\quad \ldots \\
\quad\quad \lambda v_{n-1}.\lambda\omega_{n-1}.\,[\![E_n]\!]\,\eta\varepsilon(\\
\quad\quad\quad \lambda v_n.\lambda\omega_n.\,[\![E]\!]\,\eta(\varepsilon \oslash [v_0/x_0,\ldots,v_n/x_n])(\kappa)(\omega_n) \\
\quad\quad)(\omega_{n-1}) \\
\quad\quad \ldots \\
\quad)(\omega_0) \\
)(\omega)
\end{array}
$$

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l}
\mathbf{let}\ \mathbf{(}x_0^0\mathbf{:}T_0^0\mathbf{,}\ldots\mathbf{,}x_0^{n_0}\mathbf{:}T_0^{n_0}\mathbf{):=}E_0\mathbf{,}\ldots\mathbf{,(}x_m^0\mathbf{:}T_m^0\mathbf{,}\ldots\mathbf{,}x_m^{n_m}\mathbf{:}T_m^{n_m}\mathbf{):=}E_m\ \mathbf{in} \\
\quad E \\
\mathbf{endlet}
\end{array}\right]\!\!\right]\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=} \\
[\![E_0]\!]\,\eta\varepsilon(\\
\quad \lambda v_0.\lambda\omega_0.\,[\![E_1]\!]\,\eta\varepsilon(\\
\quad\quad \ldots \\
\quad\quad \lambda v_{m-1}.\lambda\omega_{m-1}.\,[\![E_m]\!]\,\eta\varepsilon(\\
\quad\quad\quad \lambda v_m.\lambda\omega_m.\,[\![E]\!]\,\eta(\varepsilon \oslash [(v_0)_0/x_0^0,\ldots,(v_0)_{n_0}/x_0^{n_0},\ldots,(v_m)_0/x_m^0,\ldots,(v_m)_{n_m}/x_m^{n_m}])(\kappa)(\omega_m) \\
\quad\quad)(\omega_{m-1}) \\
\quad\quad \ldots \\
\quad)(\omega_0) \\
)(\omega)
\end{array}
$$

Expression "case"

Quelques notions auxiliaires sont nécessaires afin de simplifier la présentation. Etant donné une construction "$P_0 \mid \ldots \mid P_n$" et une valeur v, nous définissons la fonction sémantique $[\![P_0 \mid \ldots \mid P_n]\!] v$, qui modélise l'évaluation séquentielle des filtres $P_1, \ldots, P_n$ sur v :

$$
\begin{array}{rl}
[\![P_0 \mid \ldots \mid P_n]\!]\, v \stackrel{\mathrm{d}}{=} & \mathit{if}\ \exists i \in [0,n].\, (([\![P_i]\!]\, v)_1 = \mathbf{tt}\ \wedge\ \forall j \in [0, i-1].([\![P_j]\!]\, v)_1 = \mathbf{ff})\ \mathit{then} \\
& \quad (\mathbf{tt}, ([\![P_i]\!]\, v)_2) \\
& \mathit{else} \\
& \quad (\mathbf{ff}, [\ ]) \\
& \mathit{endif}
\end{array}
$$

Pour chaque expression "**case**" ayant m branches (éventuellement suivies par une branche "**otherwise**") et pour chaque continuation κ, environnement de fonctions η et environnement de variables ε, nous définissons la continuation κ_{exit}, modélisant les activités effectuées à la "sortie" de l'expression "**case**" si les premières m branches ont échoué :

$$
\kappa_{exit} \stackrel{\mathrm{d}}{=} \begin{cases} \lambda v.\lambda\omega.[\![E'_{m+1}]\!]\eta\varepsilon\kappa\omega & \text{si la clause "\textbf{otherwise}" est présente} \\ \lambda v.\lambda\omega.(\mathit{nop}, \mathit{append}(\omega, \texttt{"unexpected case"})) & \text{sinon} \end{cases}
$$

Lorsque la clause "**otherwise**" est absente et les premières m branches échouent, l'exécution du programme XTL doit être arrêtée. Ceci est modélisé de manière naturelle au moyen d'une continuation qui renvoie la valeur *nop* et imprime un message d'erreur approprié sur le fichier de sortie.

Utilisant les notations ci-dessus, la sémantique de l'expression "**case**" est définie comme suit :

$$
\left[\!\!\left[\begin{array}{l}
\textbf{case}\ E_0\ \textbf{in} \\
\quad P_1^0 \mid \ldots \mid P_1^{n_1}\ [\textbf{where}\ E_1]\ \text{->}\ E'_1 \\
\quad \ldots \\
\mid P_m^0 \mid \ldots \mid P_m^{n_m}\ [\textbf{where}\ E_m]\ \text{->}\ E'_m \\
[\mid \textbf{otherwise}\ \text{->}\ E'_{m+1}] \\
\textbf{endcase}
\end{array}\right]\!\!\right] \eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=}
$$

$$
\begin{array}{l}
[\![E_0]\!]\,\eta\varepsilon(\\
\quad \lambda v_0.\lambda\omega_0.\mathit{let}\ (b_1, \varepsilon_1) := [\![P_1^0 \mid \ldots \mid P_1^{n_1}]\!] v_0\ \mathit{in} \\
\qquad\qquad [\![E_1]\!]\,\eta(\varepsilon \oslash \varepsilon_1)(\\
\qquad\qquad\quad \lambda b'_1.\lambda\omega_1.\mathit{if}\ b_1\ \wedge\ b'_1\ \mathit{then} \\
\qquad\qquad\qquad\qquad [\![E'_1]\!]\,\eta(\varepsilon \oslash \varepsilon_1)(\kappa)(\omega_1) \\
\qquad\qquad\qquad \mathit{else} \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\quad \mathit{let}\ (b_m, \varepsilon_m) := [\![P_m^0 \mid \ldots \mid P_m^{n_m}]\!] v_0\ \mathit{in} \\
\qquad\qquad\qquad\qquad [\![E_m]\!]\,\eta(\varepsilon \oslash \varepsilon_m)(\\
\qquad\qquad\qquad\qquad\quad \lambda b'_m.\lambda\omega_m.\mathit{if}\ b_m\ \wedge\ b'_m\ \mathit{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![E'_m]\!]\,\eta(\varepsilon \oslash \varepsilon_m)(\kappa)(\omega_m) \\
\qquad\qquad\qquad\qquad\qquad \mathit{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa_{exit}(v_0)(\omega_m) \\
\qquad\qquad\qquad\qquad\qquad \mathit{endif} \\
\qquad\qquad\qquad\qquad)(\omega_{m-1}) \\
\qquad\qquad\qquad\quad \mathit{endlet} \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad \mathit{endif} \\
\qquad\qquad)(\omega_0) \\
\qquad\quad \mathit{endlet} \\
)(\omega)
\end{array}
$$

Expression "case action"

Afin d'obtenir une sémantique déterministe pour les expressions XTL (c'est-à-dire, de garantir que l'évaluation d'une expression ne produise qu'un seul résultat), il est nécessaire d'assurer que toutes les formules α contenues dans une expression "**case action**" ne produisent, lors de leur évaluation, qu'au plus un seul environnement initialisant les variables exportées. Nous formulons cette condition au moyen de deux prédicats $d_{tt}, d_{ff} : AForm \rightarrow \mathbf{Bool}$, définis inductivement dans la table B.3. Intuitivement, si $d_{tt}(\alpha)$ est vrai, chaque fois que α est satisfaite par une action a dans le contexte d'un environnement ε, elle renvoie un ensemble ne contenant qu'au plus un seul environnement qui initialise les variables exportées par α avec des valeurs extraites de a. Si $d_{ff}(\alpha)$ est vrai, la même propriété est assurée lorsque α n'est pas satisfaite par a.

α	$d_{tt}(\alpha)$	$d_{ff}(\alpha)$
$O_0 \dots O_m [\mathbf{\dots}] O_{m+1} \dots O_{m+n}$ [**where** E]	**tt**	**tt**
true	**tt**	**tt**
false	**tt**	**tt**
not α_1	$d_{ff}(\alpha_1)$	$d_{tt}(\alpha_1)$
α_1 **or** α_2	$v_{tt}(\alpha_1) \cap v_{tt}(\alpha_2) = \emptyset$	$d_{ff}(\alpha_1) \ \wedge \ d_{ff}(\alpha_2)$
α_1 **and** α_2	$d_{tt}(\alpha_1) \ \wedge \ d_{tt}(\alpha_2)$	$v_{ff}(\alpha_1) \cap v_{ff}(\alpha_2) = \emptyset$

Table B.3: Conditions assurant le déterminisme des formules sur actions

Ces conditions suffisantes pour le déterminisme des formules α sont formellement exprimées par le lemme suivant, qui peut être facilement montré par induction structurelle sur α.

Lemme B-1
Les propriétés suivantes sont vérifiées pour tout $\alpha \in AForm$, $a \in A$ et $\varepsilon \in \mathbf{DEnv}$:

$$\text{i) } (d_{tt}(\alpha) \ \wedge \ (\llbracket \alpha \rrbracket \varepsilon a)_1 = \mathbf{tt}) \Rightarrow |(\llbracket \alpha \rrbracket \varepsilon a)_2| \leq 1$$
$$\text{ii) } (d_{ff}(\alpha) \ \wedge \ (\llbracket \alpha \rrbracket \varepsilon a)_1 = \mathbf{ff}) \Rightarrow |(\llbracket \alpha \rrbracket \varepsilon a)_2| \leq 1.$$

■

Remarque B-4
Conformément à la sémantique des formules XTL sur actions (voir la section 3.5.2), une formule α évaluée sur une action a dans le contexte d'un environnement ε peut renvoyer un ensemble d'environnements $(\llbracket \alpha \rrbracket \varepsilon a)_2$ vide. Cette situation, qui signifie que α n'a pas été satisfaite par a et ε, doit être prise en compte dans la définition sémantique des expressions "**case action**". ■

Nous supposons que toutes les formules α contenues dans les expressions "**case action**" sont déterministes. Pour simplifier la présentation, nous introduisons les notations auxiliaires suivantes.

Pour chaque expression "**case action**" ayant m branches (éventuellement suivies par une branche "**otherwise**"), pour chaque $1 \leq i \leq m$ et pour chaque continuation κ, environnement de fonctions η et environnement de variables ε, nous définissons une continuation κ_i, modélisant l'exécution de la i-ème branche de l'expression "**case action**" :

$$
\begin{array}{rl}
\kappa_i \stackrel{\mathrm{d}}{=} & \lambda a.\lambda\omega.let\ (b_i, \mathcal{E}_i) := [\![\alpha_i]\!]\,\varepsilon a\ in \\
& \quad if\ b_i\ \wedge\ \mathcal{E}_i = \{\varepsilon_i\}\ then \\
& \quad\quad [\![E_i]\!]\,\eta(\varepsilon \oslash \varepsilon_i)(\\
& \quad\quad\quad \lambda b'_i.\lambda\omega_i.if\ b'_i\ then \\
& \quad\quad\quad\quad\quad [\![E'_i]\!]\,\eta(\varepsilon \oslash \varepsilon_i)\kappa\omega_i \\
& \quad\quad\quad\quad else \\
& \quad\quad\quad\quad\quad \kappa_{i+1}(a)(\omega_i) \\
& \quad\quad\quad\quad endif \\
& \quad\quad)(\omega) \\
& \quad else \\
& \quad\quad \kappa_{i+1}(a)(\omega) \\
& \quad endif \\
& endlet
\end{array}
$$

La continuation κ_{m+1} est définie de la manière suivante :

$$
\kappa_{m+1} \stackrel{\mathrm{d}}{=} \begin{cases} \lambda a.\lambda\omega.[\![E'_{m+1}]\!]\eta\varepsilon\kappa\omega & \text{si la clause “}\textbf{otherwise}\text{” est présente} \\ \lambda a.\lambda\omega.(nop, append(\omega, \texttt{"unexpected case"})) & \text{sinon} \end{cases}
$$

Utilisant les notations ci-dessus, la sémantique des expressions “**case action**” est définie comme suit :

$$
\left[\!\!\left[\begin{array}{l} \textbf{case action } E_0 \textbf{ in} \\ \quad \alpha_1\ \ [\textbf{where } E_1]\ \ \text{–>}\ E'_1 \\ \quad \ldots \\ \ \ |\ \alpha_m\ [\textbf{where } E_m]\ \text{–>}\ E'_m \\ \ [|\ \textbf{otherwise}\ \text{–>}\ E'_{m+1}] \\ \textbf{endcase} \end{array} \right]\!\!\right] \eta\varepsilon\kappa\omega \ \stackrel{\mathrm{d}}{=}\ [\![E_0]\!]\,\eta\varepsilon(\lambda a_0.\lambda\omega_0.\kappa_0(a_0)(\omega_0))(\omega)
$$

Expression “if”

Comme nous avons mentionné à la section 2.5.1, les expressions “**if**” ne sont pas des constructions XTL primitives. Afin de simplifier la présentation, nous définissons la sémantique de ces expressions par traduction en termes d'expressions “**case**” :

$$
\left[\!\!\left[\begin{array}{l} \textbf{if } E_0 \textbf{ then } E'_0 \\ \quad \textbf{elsif } E_1 \textbf{ then } E'_1 \\ \quad \ldots \\ \quad \textbf{elsif } E_n \textbf{ then } E'_n \\ \ [\textbf{else } E'_{n+1}] \\ \textbf{endif} \end{array} \right]\!\!\right] \eta\varepsilon\kappa\omega \ \stackrel{\mathrm{d}}{=}\ \left[\!\!\left[\begin{array}{l} \textbf{case } E_0 \textbf{ in} \\ \quad \textbf{true}\ \text{–>}\ E'_0 \\ \quad |\ \textbf{otherwise}\ \text{–>} \\ \quad\quad \textbf{case } E_1 \textbf{ in} \\ \quad\quad\quad \textbf{true}\ \text{–>}\ E'_1 \\ \quad\quad\quad |\ \textbf{otherwise}\ \text{–>} \\ \quad\quad\quad \ldots \\ \quad\quad\quad\quad \textbf{case } E_n \textbf{ in} \\ \quad\quad\quad\quad\quad \textbf{true}\ \text{–>}\ E'_n \\ \quad\quad\quad\quad\quad [|\ \textbf{otherwise}\ \text{–>}\ E'_{n+1}] \\ \quad\quad\quad\quad \textbf{endcase} \\ \quad\quad\quad \ldots \\ \quad\quad \textbf{endcase} \\ \textbf{endcase} \end{array} \right]\!\!\right] \eta\varepsilon\kappa\omega
$$

Expression “assert”

$$
\left[\!\!\left[\begin{array}{l}\textbf{assert } E_0,\ldots,E_n \textbf{ in}\\ \quad E\\ \textbf{endassert}\end{array}\right]\!\!\right]\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=}
$$

$$
\begin{array}{l}
[\![E_0]\!]\,\eta\varepsilon(\\
\quad \lambda b_0.\lambda\omega_0.\mathit{if}\ b_0\ \mathit{then}\\
\qquad\qquad [\![E_1]\!]\,\eta\varepsilon(\\
\qquad\qquad\quad \ldots\\
\qquad\qquad\quad \lambda b_n.\lambda\omega_n.\mathit{if}\ b_n\ \mathit{then}\\
\qquad\qquad\qquad\qquad [\![E]\!]\,\eta\varepsilon(\kappa)(\omega_n)\\
\qquad\qquad\qquad\quad \mathit{else}\\
\qquad\qquad\qquad\qquad (\mathit{nop}, \mathit{append}(\omega_n, \texttt{"assertion failed"}))\\
\qquad\qquad\qquad\quad \mathit{endif}\\
\qquad\qquad\quad \ldots\\
\qquad\qquad)(\omega_0)\\
\qquad\quad \mathit{else}\\
\qquad\qquad (\mathit{nop}, \mathit{append}(\omega_0, \texttt{"assertion failed"}))\\
\qquad\quad \mathit{endif}\\
)(\omega)
\end{array}
$$

L’effet d’une assertion non vérifiée est modélisé, de manière naturelle, à l’aide d’une continuation qui arrête le programme en imprimant un message d’erreur approprié sur le fichier de sortie.

Exemple B-3
L’expression “**assert ff in** E **endassert**” est évaluée dans le contexte de η, ε, κ et ω comme suit :

$$
\begin{array}{ll}
[\![\textbf{assert ff in } E \textbf{ endassert}]\!]\,\eta\varepsilon\kappa\omega = & \text{par interprétation de “}\textbf{assert}\text{”}\\
[\![\textbf{ff}]\!]\,\eta\varepsilon(& \\
\quad \lambda b_0.\lambda\omega_0.\mathit{if}\ b_0\ \mathit{then} & \\
\qquad\qquad [\![E]\!]\,\eta\varepsilon\kappa\omega_0 & \\
\qquad\quad \mathit{else} & \\
\qquad\qquad (\mathit{nop}, \mathit{append}(\omega_0, \texttt{"assertion failed"})) & \\
\qquad\quad \mathit{endif} & \\
)(\omega) = & \text{par interprétation de “}\textbf{false}\text{”}\\
\lambda b_0.\lambda\omega_0.\mathit{if}\ b_0\ \mathit{then} & \\
\qquad\quad [\![E]\!]\,\eta\varepsilon\kappa\omega_0 & \\
\qquad \mathit{else} & \\
\qquad\quad (\mathit{nop}, \mathit{append}(\omega_0, \texttt{"assertion failed"})) & \\
\qquad \mathit{endif} & \\
)(\textbf{ff})(\omega) = & \\
\mathit{if}\ \textbf{ff}\ \mathit{then}\ [\![E]\!]\,\eta\varepsilon\kappa\omega & \\
\mathit{else}\ (\mathit{nop}, \mathit{append}(\omega, \texttt{"assertion failed"})) & \\
\mathit{endif} = & \\
(\mathit{nop}, \mathit{append}(\omega, \texttt{"assertion failed"})) &
\end{array}
$$

■

Expressions "loop" et "continue"

$$\left[\!\!\left[\begin{array}{l}\textbf{loop } (x_0{:}T_0{:=}E_0,\ldots,x_n{:}T_n{:=}E_n) \textbf{ : } RT \textbf{ in}\\ \quad E\\ \textbf{endloop}\end{array}\right]\!\!\right]\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=}$$

$$\begin{array}{l}
[\![E_0]\!]\,\eta\varepsilon(\\
\quad \lambda v_0.\lambda\omega_0.\,[\![E_1]\!]\,\eta\varepsilon(\\
\quad\quad \ldots\\
\quad\quad \lambda v_{n-1}.\lambda\omega_{n-1}.\,[\![E_n]\!]\,\eta\varepsilon(\\
\quad\quad\quad \lambda v_n.\lambda\omega_n.(f(v_0,\ldots,v_n))(\kappa)(\omega_n)\\
\quad\quad)(\omega_{n-1})\\
\quad\quad \ldots\\
\quad)(\omega_0)\\
)(\omega)
\end{array}$$

où $f \stackrel{\mathrm{d}}{=} \mathit{lfp}\ \lambda f'.(\lambda v_0{:}T_0,\ldots,v_n{:}T_n.\,[\![E]\!]\,(\eta \oslash [f'/\texttt{loop}])(\varepsilon \oslash [v_0/x_0,\ldots,v_n/x_n]))$ et `loop` est un identificateur spécial de fonction, différent de tous les autres identificateurs utilisés dans le programme.

Remarque B-5

Le plus petit point fixe $\mathit{lfp} : (D \to D) \to D$ ci-dessus est défini sur le domaine $D \stackrel{\mathrm{d}}{=} T_0 \times \cdots \times T_n \to$ **ECont** $\to$ **OutFile** $\to$ **Val** $\times$ **OutFile**. Ce domaine ayant une structure d'ordre partiel complet (induite par le fait que le produit **Val** $\times$ **OutFile** est un ordre partiel complet) et la fonctionnelle $\lambda f'.(\lambda v_0{:}T_0,\ldots,v_n{:}T_n.\,[\![E]\!]\,(\eta \oslash [f'/\texttt{loop}])(\varepsilon \oslash [v_0/x_0,\ldots,v_n/x_n]))$ étant continue (car produite par composition de fonctions continues, à l'aide de λ-abstraction et d'application), ceci assure l'existence et l'unicité du plus petit point fixe. ■

$$\begin{array}{l}
[\![\textbf{continue } (E_0,\ldots,E_n)]\!]\,\eta\varepsilon\kappa\omega \stackrel{\mathrm{d}}{=}\\
\quad [\![E_0]\!]\,\eta\varepsilon(\\
\quad\quad \lambda v_0.\lambda\omega_0.\,[\![E_1]\!]\,\eta\varepsilon(\\
\quad\quad\quad \ldots\\
\quad\quad\quad \lambda v_{n-1}.\lambda\omega_{n-1}.\,[\![E_n]\!]\,\eta\varepsilon(\\
\quad\quad\quad\quad \lambda v_n.\lambda\omega_n.(\eta(\texttt{loop}))(v_0,\ldots,v_n)(\kappa)(\omega_n)\\
\quad\quad\quad)(\omega_{n-1})\\
\quad\quad\quad \ldots\\
\quad\quad)(\omega_0)\\
\quad)(\omega)
\end{array}$$

Remarque B-6

La définition sémantique ci-dessus utilise un seul identificateur spécial `loop`. Ceci est correct, car chaque occurrence d'une expression "**continue**" est associée avec la plus proche expression "**loop**" qui la contient (en effet, les "sauts" entre les expressions "**loop**" imbriquées sont interdits). ■

Expression "for"

De la même manière que pour l'expression "**loop**", la sémantique de l'expression "**for**" peut être définie au moyen d'opérateurs de point fixe. Néanmoins, afin de simplifier la présentation, nous avons préféré une définition sémantique par traduction en termes d'autres expressions XTL.

Etant donné que tout type T associé à une variable d'itération doit être énumérable, nous utiliserons dans les définitions sémantiques les fonctions $\mathit{init}_T : \to T$, $\mathit{inc}_T : T \to T$ et $\mathit{end}_T : T \to$ `boolean` associées à ces types (voir la section 2.6.2). Nous supposerons également l'existence, pour chaque type T, de la fonction $member_T : T \times 2^T \rightarrow \mathbf{Bool}$, qui teste si une valeur de type T appartient à un sous-ensemble d'éléments de T.

La sémantique des expressions "**for**" est définie par étapes successives. D'abord, nous traduisons une expression "**for**" générale, ayant $m+1$ variables d'itération, en termes d'une expression "**for**" ayant m variables d'itération.

$$\left[\!\!\left[\begin{array}{l} \mathbf{for}\ x'_0{:}T'_0[\mathbf{among}\ E'_0],\ldots,x'_m{:}T'_m[\mathbf{among}\ E'_m] \\ \quad \mathbf{var}\ x_0{:}T_0{:=}E_0,\ldots,x_n{:}T_n{:=}E_n \\ \ \ [\mathbf{where}\ E''_1] \\ \ \ [\mathbf{while}\ \ E''_2] \\ \quad \mathbf{in}\ E \\ \ \ [\mathbf{result}\ E'] \\ \mathbf{endfor} \end{array}\right]\!\!\right]\eta\varepsilon\kappa\omega \ \stackrel{\mathrm{d}}{=}$$

$$\left[\!\!\left[\begin{array}{l} [\mathbf{let}\ (x_0{:}T_0,\ldots,x_n{:}T_n){:=}] \\ \quad \mathbf{loop}\ (x'_0{:}T'_0{:=}init_{T'_0}(),x_0{:}T_0{:=}E_0,\ldots,x_n{:}T_n{:=}E_n)\ \mathbf{:}\ (T_0,\ldots,T_n)\ \mathbf{in} \\ \qquad \mathbf{if}\ end_{T'_0}(x'_0)\ \mathbf{then} \\ \qquad\quad (x_0,\ldots,x_n) \\ \qquad \left[\begin{array}{l} \mathbf{elsif}\ not(member_{T'_0}(x'_0,E'_0))\ \mathbf{then} \\ \quad \mathbf{continue}\ (inc_{T'_0}(x'_0),x_0,\ldots,x_n) \end{array}\right] \\ \qquad \mathbf{else} \\ \qquad\quad \mathbf{let}\ (x''_0{:}T_0,\ldots,x''_n{:}T_n)\mathbf{:=} \\ \qquad\qquad \mathbf{for}\ x'_1{:}T'_1[\mathbf{among}\ E'_1],\ldots,x'_m{:}T'_m[\mathbf{among}\ E'_m] \\ \qquad\qquad\quad \mathbf{var}\ x_0{:}T_0{:=}x_0,\ldots,x_n{:}T_n{:=}x_n \\ \qquad\qquad\ \ [\mathbf{where}\ E''_1] \\ \qquad\qquad\ \ [\mathbf{while}\ \ E''_2] \\ \qquad\qquad\quad \mathbf{in}\ E \\ \qquad\qquad \mathbf{endfor} \\ \qquad\quad \mathbf{in} \\ \qquad\qquad \mathbf{continue}\ (inc_{T'_0}(x'_0),x''_0,\ldots,x''_n) \\ \qquad\quad \mathbf{endlet} \\ \qquad \mathbf{endif} \\ \quad \mathbf{endloop} \\ \left[\begin{array}{l} \mathbf{in} \\ \quad E' \\ \mathbf{endlet} \end{array}\right] \end{array}\right]\!\!\right]\eta\varepsilon\kappa\omega$$

Utilisant l'identité ci-dessus de manière répétitive, toute expression "**for**" ayant plusieurs variables d'itération peut être réduite à une expression "**for**" avec une seule variable d'itération. Cette expression "**for**" est ensuite traduite en termes d'une expression "**loop**" de la manière suivante :

$$\left[\!\!\left[\begin{array}{l} \mathbf{for}\ x'{:}T'[\mathbf{among}\ E'] \\ \quad \mathbf{var}\ x_0{:}T_0{:=}E_0,\ldots,x_n{:}T_n{:=}E_n \\ \ \ [\mathbf{where}\ E''_1] \\ \ \ [\mathbf{while}\ \ E''_2] \\ \quad \mathbf{in}\ E \\ \ \ [\mathbf{result}\ E''_3] \\ \mathbf{endfor} \end{array}\right]\!\!\right]\eta\varepsilon\kappa\omega \ \stackrel{\mathrm{d}}{=}$$

$$\left[\!\!\left[\begin{array}{l}
[\textbf{let } (x_0{:}T_0,\ldots,x_n{:}T_n){:=}] \\
\quad \textbf{loop } (x'{:}T'{:=}init_{T'}(),x_0{:}T_0{:=}E_0,\ldots,x_n{:}T_n{:=}E_n) : (T_0,\ldots,T_n) \textbf{ in} \\
\quad\quad \textbf{if } end_{T'}(x') \; [or \; not(E_2'')] \textbf{ then} \\
\quad\quad\quad (x_0,\ldots,x_n) \\
\quad\quad \left[\begin{array}{l} \textbf{elsif } not(E_1'' \; [and \; member_{T'}(x',E')]) \textbf{ then} \\ \quad \textbf{continue } (inc_{T'}(x'),x_0,\ldots,x_n) \end{array}\right] \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{let } (x_0''{:}T_0,\ldots,x_n''{:}T_n){:=}E \textbf{ in} \\
\quad\quad\quad\quad \textbf{continue } (inc_{T'}(x'),x_0',\ldots,x_n') \\
\quad\quad\quad \textbf{endlet} \\
\quad\quad \textbf{endif} \\
\quad \textbf{endloop} \\
\left[\begin{array}{l} \textbf{in} \\ \quad E_3'' \\ \textbf{endlet} \end{array}\right]
\end{array}\right]\!\!\right] \eta\varepsilon\kappa\omega$$

Les itérateurs, les ensembles définis en compréhension et les quantificateurs sont des cas particuliers d'expressions "**for**". Leurs traductions respectives ont été données aux sections 2.6.3, 2.6.4 et 2.6.5.

B.2 Définitions de fonctions

Cette section est consacrée à la sémantique dénotationnelle des définitions de fonctions XTL. La syntaxe abstraite considérée ici comporte quelques différences mineures par rapport à celle présentée à la section 2.1 :

- les définitions de fonctions infixées "**function** _ F _ $(\ldots)$" ont été remplacées par leurs correspondantes préfixées "**function** F $(\ldots)$" ;
- les occurrences du mot-clé "**local**" précédant les définitions de fonctions locales ont été éliminées, car ce mot-clé sert uniquement lors de la liaison statique des fonctions.

Ces deux simplifications n'ont aucune influence sur la sémantique dénotationnelle des définitions de fonctions XTL.

B.2.1 Aspects syntaxiques

Afin de rendre la présentation plus claire, nous considérons directement des listes de définitions de fonctions : ceci rend plus facile la définition sémantique, étant donné le fait que les fonctions XTL peuvent être mutuellement récursives. Le domaine syntaxique associé aux listes de définitions de fonctions XTL est noté *FuncDefList*.

B.2.2 Aspects sémantiques

La sémantique d'une liste de définitions de fonctions XTL est définie par la fonction d'interprétation suivante :

$$[\![.]\!] : \mathit{FuncDefList} \rightarrow \mathbf{FEnv} \rightarrow \mathbf{FEnv}$$

Etant donné une liste de définitions de fonctions $D_1,\ldots,D_n$ et un environnement η (contenant les fonctions prédéfinies XTL et les fonctions BCG), la dénotation $[\![D_1,\ldots,D_n]\!]\,\eta$ renvoie l'environnement de fonctions créé par les définitions $D_1, \ldots, D_n$ dans le contexte de η. Cette fonction sémantique est définie comme suit :

$$\left[\!\!\left[\begin{array}{l}\textbf{function } F_1 \ (x_1^1{:}T_1^1, \ldots, x_1^{m_1}{:}T_1^{m_1}) \ \ : RT_1 \textbf{ is } E_1 \textbf{ endfunc} \\ \ldots \\ \textbf{function } F_n \ (x_n^1{:}T_n^1, \ldots, x_n^{m_n}{:}T_n^{m_n}) : RT_n \textbf{ is } E_n \textbf{ endfunc}\end{array}\right]\!\!\right] \eta \stackrel{\mathrm{d}}{=} [f_1/F_1, \ldots, f_n/F_n]$$

où les fonctions $f_1, \ldots, f_n$ sont définies par l'opérateur de point fixe suivant :

$$\begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} \stackrel{\mathrm{d}}{=} \mathit{lfp}\ \lambda \begin{pmatrix} f_1' \\ \vdots \\ f_n' \end{pmatrix} . \begin{pmatrix} \lambda v_1^1{:}T_1^1, \ldots, v_1^{m_1}{:}T_1^{m_1} . [\![E_1]\!] \, (\eta \oslash [f_1'/F_1, \ldots, f_n'/F_n])[v_1^1/x_1^1, \ldots, v_1^{m_1}/x_1^{m_1}] \\ \vdots \\ \lambda v_n^1{:}T_n^1, \ldots, v_n^{m_n}{:}T_n^{m_n} . [\![E_n]\!] \, (\eta \oslash [f_1'/F_1, \ldots, f_n'/F_n])[v_n^1/x_n^1, \ldots, v_n^{m_n}/x_n^{m_n}] \end{pmatrix}$$

Remarque B-7
Le plus petit point fixe $\mathit{lfp} : (D \to D) \to D$ ci-dessus est défini sur le domaine $D \stackrel{\mathrm{d}}{=} (\textbf{Param} \to \textbf{ECont} \to \textbf{OutFile} \to \textbf{Val} \times \textbf{OutFile})^n$. Le fait que ce domaine ait une structure d'ordre partiel complet (induite par le fait que le produit **Val** $\times$ **OutFile** est un ordre partiel complet) et que la fonctionnelle respective soit continue (étant une composition de fonctions continues, au moyen de λ-abstraction et d'application) assure l'existence et l'unicité du plus petit point fixe. ■

B.3 Programme

Nous concluons cette annexe en définissant la sémantique dénotationnelle des programmes XTL. La syntaxe abstraite utilisée ici est légèrement différente de celle présentée à la section 2.1 : les définitions de formules "**formula**" et les inclusions de bibliothèques "**library**" ne sont plus considérées, car elles ont été expansées syntaxiquement avant de commencer l'analyse sémantique du programme (voir la section 2.13).

B.3.1 Aspects syntaxiques

Le domaine syntaxique associé aux programmes XTL est noté *Program*.

B.3.2 Aspects sémantiques

Quelques notions auxiliaires sont nécessaires. Nous supposons donnée l'interprétation $[\![.]\!] : \mathit{Func} \to \textbf{Param} \to \textbf{Val}$ des fonctions XTL prédéfinies et des fonctions BCG (en pratique, le compilateur XTL dispose de bibliothèques implémentant les fonctions XTL prédéfinies). En utilisant cette interprétation, l'environnement η_0 contenant ces fonctions est défini comme suit :

$$\eta_0(F) \stackrel{\mathrm{d}}{=} \lambda v_1{:}T_1, \ldots, v_n{:}T_n . \lambda\kappa . \lambda\omega . (\kappa([\![F]\!] \, (v_1, \ldots, v_n))(\omega))$$

pour chaque fonction $F : T_1 \times \cdots \times T_n \to T$ prédéfinie en XTL ou définie dans le fichier BCG. L'évaluation d'un appel $F(v_1, \ldots, v_n)$, dans le contexte d'une continuation κ et d'un fichier de sortie ω, produit un appel de κ avec la valeur $([\![F]\!])(v_1, \ldots, v_n)$ en laissant le fichier ω inchangé (car aucune fonction, hormis l'opérateur XTL "**print**", ne peut modifier le fichier de sortie).

Nous introduisons aussi la continuation d'un programme XTL, appelée *result*, qui modélise les actions effectuées après l'exécution du programme :

$$\mathit{result} \stackrel{\mathrm{d}}{=} \lambda v . \lambda\omega . (v, \mathit{append}(\omega, \texttt{"end-of-file"}))$$

Après la terminaison du programme, *result* renvoie la valeur produite par le corps du programme et rajoute le message `"end-of-file"` à la fin du fichier de sortie obtenu.

La sémantique d'un programme XTL est définie par la fonction d'interprétation suivante :

$$[\![.]\!] : \mathit{Program} \rightarrow \mathbf{FEnv} \rightarrow \mathbf{Val} \times \mathbf{OutFile}$$

Etant donné un programme XTL PG et un environnement η_0 (contenant les fonctions XTL prédéfinies et les fonctions BCG), la dénotation $[\![PG]\!]\,\eta_0$ renvoie un tuple contenant deux champs : (1) la valeur produite par l'évaluation du corps de PG dans le contexte de η_0 et (2) le fichier de sortie contenant les valeurs imprimées par PG au cours de son exécution. Cette fonction sémantique est définie de la manière suivante :

$$\left[\!\!\left[\begin{array}{l} [D_0 \ldots D_m] \\ \quad E \\ [\mathbf{where}\ D_{m+1} \ldots D_{m+n}] \end{array} \right]\!\!\right] \eta_0 \stackrel{\mathrm{d}}{=} [\![E]\!]\,(\eta_0 \oslash ([\![D_0 \ldots D_m]\!]\,\eta_0 \oplus [\![D_{m+1} \ldots D_{m+n}]\!]\,\eta_0))[\,]\ \mathit{result}\ \mathit{nil}$$

Le corps E du programme est évalué dans le contexte d'un environnement de fonctions qui étend η_0 avec les fonctions (optionnellement) définies dans $D_0, \ldots, D_m$ et dans $D_{m+1}, \ldots, D_{m+n}$, d'un environnement de variables vide (car il n'existe pas de variables globales), de la continuation *result* et d'un fichier de sortie vide (car aucune valeur n'a été imprimée en sortie avant que l'évaluation de E soit commencée).

Annexe C

Logiques temporelles traduites en XTL

Cette annexe présente les traductions en XTL de plusieurs logiques temporelles arborescentes largement utilisées dans les applications.

Les sections C.1 et C.2 contiennent les traductions en XTL des logiques CTL [CES86] et ACTL [NV90]. Les opérateurs de CTL peuvent être combinés naturellement avec des formules XTL sur états, ce qui autorise l'expression de propriétés portant sur les valeurs au moyen du méta-opérateur "**current**" sur états. Les opérateurs d'ACTL peuvent être combinés avec des formules XTL sur états aussi bien qu'avec des formules XTL sur actions, ce qui permet d'exprimer des propriétés concernant les valeurs contenues dans les actions du STE.

La section C.3 décrit la traduction en XTL d'un fragment "purement arborescent" de la logique modale de μCRL [GvV94]. Les opérateurs de cette logique peuvent être combinés, eux aussi, avec des formules XTL sur états et sur actions.

Finalement, la section C.4 présente les définitions en XTL de plusieurs opérateurs temporels utiles appartenant à la logique LTAC [Que82, QS83] et au calcul de Dicky [Dic86, Arn89, Arn92, ABC94].

C.1 Traduction de la logique CTL

La logique temporelle CTL [CES86] peut être implémentée en XTL en utilisant les opérateurs de point fixe et les mécanismes de définition de formules paramétrées. Les définitions de formules ci-dessous peuvent constituer une bibliothèque `ctl.xtl`, destinée à être incluse et utilisée, au moyen de l'expanseur XTL (voir la section 2.14), dans d'autres programmes XTL.

Les paramètres `P`, `P1` et `P2` dénotent des formules CTL sur états (φ). Le prédicat de base `INIT` caractérise l'état initial du modèle. Les opérateurs booléens contenus dans les formules CTL sont incorporés dans XTL. Les opérateurs `EX (P)`, `AX (P)`, `EU (P1, P2)` et `AU (P1, P2)` représentent respectivement les opérateurs $\mathbf{EX}\varphi$, $\mathbf{AX}\varphi$, $\mathbf{E}[\varphi_1 \ \mathbf{U} \ \varphi_2]$ et $\mathbf{A}[\varphi_1 \ \mathbf{U} \ \varphi_2]$ de CTL. Les opérateurs `EF (P)`, `AF (P)`, `EG (P)` et `AG (P)` définissent les modalités usuelles $\mathbf{EF}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{EG}\varphi$ et $\mathbf{AG}\varphi$.

```
formula INIT () is                                  (* Predicats de base    *)
    (current = init)
endform
```

```
formula EX (P) is                                (* Operateurs modaux    *)
    < true > (P)
endform

formula AX (P) is
    (< true > true and [ true ] (P))
endform

formula EU (P1, P2) is                           (* Operateurs temporels *)
    mu Y . ((P2) or (P1) and EX (Y))
endform

formula AU (P1, P2) is
    mu Y . ((P2) or (P1) and AX (Y))
endform

formula EF (P) is EU (true, P)     endform       (* Operateurs derives   *)

formula AF (P) is AU (true, P)     endform

formula EG (P) is not AF (not (P)) endform

formula AG (P) is not EF (not (P)) endform
```

Les traductions ci-dessus sont basées sur les caractérisations des modalités temporelles de CTL en termes d'opérateurs de point fixe (voir la section 1.2.4). Les opérateurs `EX` et `AX` sont utilisés dans les traductions de `EU` et de `AU` afin d'obtenir des descriptions plus concises.

Les arguments `P`, `P1` et `P2` des opérateurs CTL peuvent être des prédicats de base XTL sur états, exprimés au moyen du méta-opérateur "**current**" sur états.

Exemple C-1
Soit un programme parallèle contenant `n` processus (numérotés de `1` à `n`) qui sont en compétition pour une ressource partagée. La formule CTL suivante exprime le fait qu'un processus `i` qui a demandé l'accès à la ressource l'obtiendra inévitablement au bout d'un temps fini :

```
request (current.processes, i) implies AF (grant (current.processes, i))
```

où la variable d'état `processes` mémorise l'ensemble des processus et les prédicats `request` (*resp.* `grant`), définis dans le programme à vérifier, expriment qu'un processus a demandé (*resp.* a obtenu) l'accès à la ressource. Le méta-opérateur "**current**", combiné avec le mécanisme d'expansion des formules, permet le passage en paramètre des prédicats portant sur l'état courant, autorisant ainsi une description intuitive des propriétés. ■

Le prédicat de base `INIT` (qui ne fait pas partie de la définition originelle de CTL [CES86], étant en fait utilisé dans la logique LTAC [QS83]) permet d'exprimer certaines propriétés temporelles utiles.

Exemple C-2
Le fait que le programme soit réinitialisable peut être exprimé, au moyen de l'opérateur `INIT`, par la formule CTL suivante :

```
AG (EF (INIT))
```

qui spécifie qu'à partir de tout état du programme, il est possible d'atteindre l'état initial. ■

C.2 Traduction de la logique ACTL

De la même manière que pour CTL, les opérateurs de la logique temporelle ACTL [NV90] peuvent être définis comme formules XTL paramétrées par des formules sur actions et sur états. Les définitions de formules ci-dessous peuvent constituer une bibliothèque `actl.xtl`, destinée à être réutilisée dans d'autres spécifications XTL (voir l'exemple 2-73 dans la section 2.15).

Les paramètres `A`, `A1` et `A2` dénotent des formules ACTL sur actions (α). Les paramètres `P`, `P1` et `P2` dénotent des formules ACTL sur états (φ). Les opérateurs booléens contenus dans les formules sur actions ou dans les formules sur états sont incorporés dans XTL. Les opérateurs `EX_A (A, P)`, `AX_A (A, P)`, `EU_A (P1, A, P2)`, `AU_A (P1, A, P2)`, `EU_A_A (P1, A1, A2, P2)` et `AU_A_A (P1, A1, A2, P2)` représentent respectivement les opérateurs $\mathbf{EX}_\alpha\varphi$, $\mathbf{AX}_\alpha\varphi$, $\mathbf{E}[\varphi_{1\,\alpha}\mathbf{U}\ \varphi_2]$, $\mathbf{A}[\varphi_{1\,\alpha}\mathbf{U}\ \varphi_2]$, $\mathbf{E}[\varphi_{1\,\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$ et $\mathbf{A}[\varphi_{1\,\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$ d'ACTL. Outre les modalités dérivées usuelles `EF (P)`, `AF (P)`, `EG (P)` et `AG (P)`, qui représentent respectivement les opérateurs $\mathbf{EF}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{EG}\varphi$ et $\mathbf{AG}\varphi$, nous avons défini aussi les opérateurs correspondants indexés par des formules sur actions, qui permettent l'expression de propriétés utiles : `EF_A (A, P)` dénote $\mathbf{E}[true_\alpha\mathbf{U}\ \varphi]$; `AF_A (A, P)` dénote $\mathbf{A}[true_\alpha\mathbf{U}\ \varphi]$; `EG_A (A, P)` dénote $\neg\mathbf{A}[true_\alpha\mathbf{U}\ \neg\varphi]$; `AG_A (A, P)` dénote $\neg\mathbf{E}[true_\alpha\mathbf{U}\ \neg\varphi]$.

```
formula TAU () is                                  (* Filtres d'actions    *)
    (any where not visible (current))
endform

formula EX_A (A, P) is                             (* Operateurs modaux    *)
    < A > (P)
endform

formula AX_A (A, P) is
    (< true > true and [ not (A) ] false and [ A ] (P))
endform

formula EU_A (P1, A, P2) is                        (* Operateurs temporels *)
    mu Y . ((P2) or (P1) and EX_A (A, Y))
endform

formula AU_A (P1, A, P2) is
    mu Y . ((P2) or (P1) and AX_A (A, Y))
endform

formula EU_A_A (P1, A1, A2, P2) is
    mu Y . ((P1) and (< A2 > (P2) or < A1 > Y))
endform

formula AU_A_A (P1, A1, A2, P2) is
    mu Y . (
        (P1) and < true > true and [ not ((A1) or (A2)) ] false and
        [ not (A1) and (A2) ] (P2) and [ not (A2) ] Y and
        [ (A1) and (A2) ] (Y or (P2))
    )
endform
```

```
formula EF_A (A, P) is EU_A (true, A, P)     endform (* Operateurs derives   *)

formula EF (P)      is EF_A (true, P)        endform

formula AF_A (A, P) is AU_A (true, A, P)     endform

formula AF (P)      is AF_A (true, P)        endform

formula EG_A (A, P) is not AF_A (A, not (P)) endform

formula EG (P)      is EG_A (true, P)        endform

formula AG_A (A, P) is not EF_A (A, not (P)) endform

formula AG (P)      is AG_A (true, P)        endform
```

Les traductions ci-dessus sont basées sur les caractérisations des modalités temporelles d'ACTL en termes d'opérateurs de point fixe (voir la section 1.2.4). Les opérateurs `EX_A` et `AX_A` sont utilisés dans les traductions de `EU_A` et de `AU_A` afin d'obtenir des descriptions plus concises.

Il est possible d'utiliser des formules XTL sur actions comme arguments `A`, `A1` et `A2` des opérateurs ACTL et, par conséquent, de bénéficier des mécanismes de filtrage et d'extraction des valeurs contenues dans les actions du modèle STE. A partir de la sémantique statique de XTL, nous pouvons préciser les règles de visibilité des variables exportées par les formules XTL sur actions passées en paramètre aux opérateurs ACTL (voir table C.1).

OPÉRATEUR ACTL	VARIABLES	
	EXPORTÉES PAR	VISIBLES DANS
$\mathbf{EX}_\alpha\ \varphi$	$v_{tt}(\alpha)$	φ
$\mathbf{AX}_\alpha\ \varphi$	$v_{tt}(\alpha)$	φ
$\mathbf{E}[\varphi_1{}_{\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$	$v_{tt}(\alpha_2)$	φ_2
$\mathbf{A}[\varphi_1{}_{\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$	$v_{tt}(\alpha_2)$	φ_2

Table C.1: Visibilité des variables définies dans les arguments des opérateurs ACTL

Ces règles de visibilité concordent avec la sémantique des opérateurs ACTL (voir la définition 1-9). Pour $\mathbf{EX}_\alpha\ \varphi$ (*resp.* $\mathbf{AX}_\alpha\ \varphi$), les variables définies dans α sont visibles dans φ, puisque φ peut (*resp.* doit) être atteinte après une α-transition. Pour $\mathbf{E}[\varphi_1{}_{\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$ (*resp.* $\mathbf{A}[\varphi_1{}_{\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$), les variables définies dans α_2 sont visibles dans φ_2, puisque φ_2 peut (*resp.* doit) être atteinte après une séquence d'α_1-transitions terminée par une α_2-transition ; en revanche, les variables définies dans α_1 ne sont visibles ni dans φ_1, ni dans φ_2, car la séquence d'α_1-transitions précédant l'α_2-transition finale peut être vide.

Exemple C-3
Une propriété usuelle de vivacité des protocoles de communication est qu'à partir de tout état du protocole, il est possible d'atteindre l'émission d'un message, qui sera obligatoirement suivie par la réception du même message. Utilisant les opérateurs ACTL paramétrés par des formules XTL sur actions, cette propriété s'exprime de la façon suivante :

```
EU_A_A (true, true, SEND ? m : Msg,
    AU_A_A (true, not (SEND any), RECV ! m, true)
)
```

La variable m, définie dans l'argument `SEND ? m : Msg` de l'opérateur `EU_A_A`, est visible dans la formule `AU_A_A`. La formule `not (SEND any)`, passée comme deuxième argument de l'opérateur `AU_A_A`, assure qu'aucun autre message n'a été émis avant la réception de m. ■

Les arguments P, P1 et P2 des opérateurs ACTL peuvent dénoter des formules XTL sur états : ceci permet d'utiliser de manière naturelle les opérateurs ACTL pour exprimer des propriétés portant sur les états (voir la section 1.2.3). En particulier, les opérateurs modaux et temporels de CTL peuvent être exprimés en ACTL de la manière suivante :

```
formula EX (P)       is EX_A (true, P)       endform

formula AX (P)       is AX_A (true, P)       endform

formula EU (P1, P2) is EU_A (P1, true, P2) endform

formula AU (P1, P2) is AU_A (P1, true, P2) endform
```

La bibliothèque `ctl.xtl` présentée à la section C.1 pourrait donc être définie à partir de `actl.xtl`.

C.3 Traduction d'un fragment de la logique modale de μCRL

Cette section contient la traduction en XTL d'un fragment "purement arborescent" de la logique modale de μCRL définie en [GvV94]. Un fragment de cette logique, contenant des formules sur états et sur chemins, a été présenté à la section 1.2.5. Le fragment "purement arborescent" (c'est-à-dire ne contenant pas de formules de chemins) étudié ci-dessous a été obtenu de la même manière que les fragments similaires des logiques CTL* ou ACTL*, c'est-à-dire en imposant que chaque modalité de chemins @ et **U** soit immédiatement préfixée par un quantificateur. Les définitions de formules données ci-dessous peuvent être groupées dans une bibliothèque `mcrl.xtl`, réutilisable dans d'autres programmes XTL.

Les paramètres A dénotent des formules μCRL sur actions (α). Les paramètres P, P1 et P2 dénotent des formules μCRL sur états (φ). Les termes t, les prédicats d'égalité sur les termes et les quantificateurs sur les valeurs sont incorporés en XTL. Les opérateurs `EX (A, P)`, `AX (A, P)`, `EU (P1, P2)` et `AU (P1, P2)` représentent respectivement les opérateurs $\exists @\, \varphi$, $\forall @\, \varphi$, $\exists[\varphi_1 \ \mathbf{U}\ \varphi_2]$ $\forall[\varphi_1 \ \mathbf{U}\ \varphi_2]$ de la logique μCRL. Les opérateurs `Dia (P)` et `Box (P)` implémentent respectivement les modalités dérivées $\Diamond\varphi$ et $\Box\varphi$. Les opérateurs EU et AU ont des définitions identiques aux opérateurs correspondants de CTL. L'opérateur modal EX (*resp.* AX) exprime l'atteignabilité potentielle (*resp.* inévitable) d'un certain état, après avoir traversé une certaine action (éventuellement précédée et suivie par des séquences de τ-transitions).

```
formula TAU () is                                    (* Filtres d'actions    *)
    (any where not visible (current))
endform

formula EX (A, P) is                                 (* Operateurs modaux    *)
    < TAU* . (A) . TAU* > (P)
endform

formula AX (A, P) is
    AU_A_A (true, TAU, (A), AU_A (true, TAU, (P)))
endform
```

```
formula EU (P1, P2) is                                (* Operateurs temporels *)
    mu Y . ((P2) or (P1) and EX (Y))
endform

formula AU (P1, P2) is
    mu Y . ((P2) or (P1) and AX (Y))
endform

formula Dia (P) is EX (true, P)      endform          (* Operateurs derives   *)

formula Box (P) is not Dia (not (P)) endform
```

Par souci de concision, nous avons traduit les opérateurs `EX` et `AX` respectivement au moyen d'expressions régulières XTL et de l'opérateur **A**[.**U**.] d'ACTL (implémenté par l'opérateur `AU_A_A` défini à la section C.2).

De la même manière que pour ACTL, les paramètres `A` des opérateurs de la logique μCRL peuvent contenir des formules XTL sur actions, ce qui permet d'utiliser les mécanismes de filtrage et d'extraction des valeurs contenues dans les actions du modèle STE.

Exemple C-4
En reprenant l'exemple C-3, la propriété d'atteignabilité potentielle de l'émission d'un message, suivie inévitablement par la réception du même message, peut être exprimée au moyen des opérateurs modaux de la logique μCRL comme suit :

```
EX (SEND ? m : Msg, AX (RECV ! m, true))
```

La formule ci-dessus n'utilise pas de quantificateurs explicites sur les valeurs, mais des filtres d'actions : les règles de propagation des variables exportées dans les expressions régulières assurent que la variable `m` définie dans le filtre d'action `SEND ? m : Msg` est visible dans le filtre d'action `RECV ! m`. ■

C.4 Opérateurs particuliers

L'expérience montre qu'en pratique, il est utile de disposer d'opérateurs dérivés permettant une expression concise et intuitive de certaines propriétés. Dans les paragraphes suivants, nous présentons les traductions en XTL de différents opérateurs temporels utiles appartenant à la logique LTAC [Que82, QS83, Rod88] et au calcul de Dicky [Dic86, Arn89, Arn92, ABC94].

Opérateurs de sûreté Un opérateur utile pour exprimer certaines propriétés de sûreté est "**not** φ_1 **to** φ_2 **unless** φ_3", exprimant le fait qu'à partir d'un état satisfaisant φ_1, il n'est pas possible d'atteindre un état satisfaisant φ_2 sans passer par un état satisfaisant φ_3. Cet opérateur, qui a été utilisé en [RRSV87] comme modalité dérivée de LTAC, peut être défini en CTL comme suit :

```
formula NOT_TO_UNLESS (P1, P2, P3) is
    (P1) implies not EU (not (P3), not (P3) and (P2))
endform
```

Exemple C-5
En reprenant l'exemple C-1, le fait qu'un processus `i` ne puisse pas obtenir l'accès à la ressource avant de l'avoir demandé peut être décrit au moyen de l'opérateur "**not** φ_1 **to** φ_2 **unless** φ_3" :

```
NOT_TO_UNLESS (
    INIT,
    grant   (current.processes, i),
    request (current.processes, i)
)
```

Cette propriété porte sur le passé, car elle fait référence aux états atteints *avant* l'état où le processus `i` a gagné la ressource ; néanmoins, elle peut être exprimée comme propriété du futur, en faisant référence à l'état initial du programme au moyen du prédicat `INIT`. ■

L'opérateur "**not** φ_1 **to** φ_2 **unless** φ_3" défini ci-dessus ne permet pas d'exprimer des propriétés sur les actions du programme et, par conséquent, il n'est pas adapté à la vérification des programmes LOTOS. Du fait de son caractère extensible, le langage XTL permet facilement de définir une version de l'opérateur "**not-to-unless**" adaptée aux actions. Cet opérateur, noté "**not** α_1 **to** φ_2 **unless** α_3", signifie qu'après avoir effectué une α_1-action, il n'est pas possible d'atteindre un état satisfaisant φ_2 sans effectuer une α_3-action. Cet opérateur peut être défini en ACTL de la manière suivante :

```
formula NOT_TO_UNLESS (A1, P2, A3) is
    not EX_A (A1, EF_A (not (A3), P2))
endform
```

Bien entendu, les arguments `A1` et `A3` de cet opérateur peuvent dénoter des formules XTL sur actions ; les variables exportées par α_1 sont visibles dans φ_2 et dans α_3.

Exemple C-6
L'exclusion mutuelle entre plusieurs processus (identifiés par des valeurs d'un type énuméré `Pid`) accédant à une ressource partagée signifie qu'une fois qu'un processus `p1` a obtenu l'accès à la ressource (action `OPEN`), il est impossible qu'un autre processus `p2` obtienne l'accès à la ressource avant que `p1` ne l'ait libérée (action `CLOSE`). Cette propriété peut être exprimée à l'aide de l'opérateur "**not** α_1 **to** φ_2 **unless** α_3" comme suit :

```
NOT_TO_UNLESS (
    OPEN ? p1 : Pid,
    EX_A (OPEN ? p2 : Pid, p1 <> p2),
    CLOSE ! p1
)
```

où les variables `p1` et `p2` sont utilisées conformément aux règles de visibilité associées aux opérateurs `NOT_TO_UNLESS` et `EX_A`. ■

Opérateurs d'équité Une construction utile pour décrire certaines propriétés d'équité est l'opérateur **fair** de la logique LTAC [QS83]. Cet opérateur exprime l'atteignabilité inévitable de certains états suivant des *chemins équitables*, c'est-à-dire des chemins π tels que toute formule atteignable infiniment souvent à partir des états de π soit satisfaite infiniment souvent par des états de π. Un état s satisfait $\mathbf{fair}(\varphi)$ ssi tous les chemins équitables issus de s aboutissent, au bout d'un nombre fini de transitions, à des états satisfaisant φ. Cet opérateur peut être exprimé en termes des autres opérateurs de LTAC [QS83], qui est une logique purement arborescente similaire à CTL. En utilisant l'opérateur $\mathbf{all}(\varphi_1, \varphi_2) \stackrel{\mathrm{d}}{=} \neg\mathbf{E}[\varphi_1 \ \mathbf{U} \ \neg\varphi_2]$, c'est-à-dire le dual de $\mathbf{E}[.\mathbf{U}.]$ sur son deuxième argument, l'opérateur $\mathbf{fair}(\varphi)$ peut être défini de la manière suivante [Que82] :

```
formula FAIR (P) is
    ALL (not (P), EF (P))
endform
```

où le paramètre `P` dénote une formule φ sur états.

L'opérateur **fair**(φ) est défini en termes d'états et, par conséquent, il ne permet pas d'exprimer l'atteignabilité équitable de certaines actions. Une version de cet opérateur basée sur actions peut être définie, utilisant les modalités d'ACTL, de la manière suivante :

```
formula FAIR (A) is
    AG_A (not (A), EF_A (A))
endform
```

où le paramètre `A` dénote une formule α sur actions.

Une autre modalité permettant d'exprimer des propriétés d'équité est l'opérateur **loop** défini dans la logique de Dicky. Un état s satisfait **loop**(φ) ssi s fait partie d'un circuit de transitions du modèle passant par un état qui satisfait φ. Dans [Dic86] il est montré que cet opérateur ne peut pas être exprimé au moyen des autres opérateurs du calcul de Dicky (systèmes d'équations d'alternance 1). Grâce au méta-opérateur "**current**" sur états, XTL permet d'exprimer cet opérateur, en utilisant les modalités de CTL, de la manière suivante :

```
formula LOOP (P) is
    let first_state : state := current in
        EF ((P) and EF (EX (current = first_state)))
    endlet
endform
```

où le paramètre `P` dénote une formule φ sur états. L'état courant sur lequel la formule ci-dessus est évaluée est capturé à l'aide du méta-opérateur "**current**" et mémorisé dans la variable `first_state` de type `state`. Le corps de l'opérateur "**let**" exprime le fait qu'il est possible d'atteindre (après zéro ou plusieurs transitions) un état qui satisfait `P` et à partir duquel il est possible de revenir (après une ou plusieurs transitions) à l'état de départ `first_state`. Ceci garantit l'existence d'un circuit passant par l'état `first_state` et par un état satisfaisant `P`. La formule XTL ci-dessus est d'alternance 1, car elle ne contient que des modalités exprimables au moyen d'opérateurs de plus petit point fixe.

Exemple C-7

La modalité d'équité $\mathbf{E}\overset{\infty}{\mathbf{F}}\varphi \overset{\mathrm{d}}{=} \mathbf{EGF}\varphi$ d'ECTL [EC80] peut être exprimée à l'aide de l'opérateur **loop** de la manière suivante :

```
formula EGF (P) is
    EF (LOOP (P))
endform
```

où le paramètre `P` dénote une formule φ sur états. L'opérateur ci-dessus caractérise les états s du modèle à partir desquels il est possible d'atteindre infiniment souvent un état satisfaisant `P`. ■

L'opérateur **loop**(φ) est défini en termes d'états et, par conséquent, il ne permet pas de caractériser le fait qu'une certaine action soit exécutée infiniment souvent. Une version de cet opérateur basée sur actions peut être définie, au moyen des modalités d'ACTL, de la manière suivante :

```
formula LOOP (A) is
    let first_state : state := current in
        EF_A (true, EX_A (A, EF_A (true, current = first_state)))
    endlet
endform
```

où le paramètre `A` dénote une formule α sur actions. La formule ci-dessus exprime le fait qu'un état s fait partie d'un circuit contenant une transition étiquetée par une action satisfaisant `A`. Tout comme la formule XTL définissant **loop**(φ), la formule ci-dessus est d'alternance 1, puisqu'elle ne contient que des modalités définies par des opérateurs de plus petit point fixe.

Bibliographie

[ABC94] A. Arnold, D. Bégay, and P. Crubillé. *Construction and Analysis of Transition Systems with MEC*. World Scientific, 1994.

[AC88] A. Arnold and P. Crubillé. A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems. *Information Processing Letters*, 29:57–66, 1988.

[AFK87] K. R. Apt, N. Francez, and S. Katz. Appraising Fairness in Languages for Distributed Programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages POPL '87 (Münich, West Germany)*, pages 189–198, January 1987.

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings, 1991.

[And92] H. R. Andersen. Model Checking and Boolean Graphs. In *Proceedings of the 4th European Symposium on Programming ESOP '92 (Rennes, France)*, volume 582 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, February 1992. Springer Verlag.

[And94] H. R. Andersen. Model Checking and Boolean Graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.

[Arn89] A. Arnold. *MEC: A System for Constructing and Analysing Transition Systems*. In J. Sifakis, editor, *Automatic verification of finite state systems*, volume 407 of *Lecture Notes in Computer Science*, pages 117–132. 1989.

[Arn92] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, 1992.

[AS85] B. Alpern and F. B. Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The Temporal Logic of Branching Time. *Acta Informatica*, 20:207–226, 1983. An extended abstract appeared in Proceedings of POPL '81 (Williamsburg, VA), pages 164–176.

[BB86] B. Banieqbal and B. Barringer. A Study of an Extended Temporal Language and a Temporal Fixed Point Calculus. Technical Report UMCS-86-10-2, Dept. of Computer Science, University of Manchester, 1986.

[BC96] G. Bhat and R. Cleaveland. Efficient Model Checking via the Equational μ-calculus. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science LICS '96 (New Brunswick, New Jersey)*, Lecture Notes in Computer Science, pages 304–312. IEEE Computer Society Press, July 1996.

[BCG95] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science LICS '95 (San Diego, California)*, pages 388–397. IEEE Computer Society Press, June 1995.

[Bek84] H. Bekić. *Definable Operations in General Algebras, and the Theory of Automata and Flowcharts*. volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer Verlag, Berlin, 1984.

[BGL94] A. Bouali, S. Gnesi, and S. Larosa. The Integration Project for the JACK Environment. *Bulletin of the EATCS*, 54:207–223, October 1994.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[Bra92] J. C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhäuser, Berlin, 1992.

[BRRd96] Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*. Springer Verlag, August 1996.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cho74] Y. Choueka. Theories of Automata on ω-tapes: A Simplified Approach. *Journal of Computer and System Sciences*, 8(2):117–141, April 1974.

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. Faster Model Checking for the Modal Mu-Calculus. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the 4th International Workshop on Computer Aided Verification CAV '92 (Montréal, Canada)*, volume 663 of *Lecture Notes in Computer Science*, pages 410–422, Berlin, June-July 1992. Springer Verlag.

[Cle90] R. Cleaveland. Tableau-Based Model Checking in the Propositional Mu-Calculus. *Acta Informatica*, 27(8):725–747, 1990.

[CLSS96] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: a Development Environment for Concurrent Systems. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th Workshop on Computer Aided Verification CAV '96 (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 398–401. Springer Verlag, August 1996.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench.* In J. Sifakis, editor, *Automatic Verification of Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. 1989.

[CS91a] R. Cleaveland and B. Steffen. Computing Behavioural Relations, Logically. In *Proceedings of the 18th ICALP*, volume 510 of *Lecture Notes in Computer Science*, pages 127–138, Berlin, 1991. Springer Verlag.

[CS91b] R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In K. G. Larsen and A. Skou, editors, *Proceedings of 3rd Workshop on Computer Aided Verification CAV '91 (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58, Berlin, July 1991. Springer Verlag.

[CS93] R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of the 2nd International Conference on Computer Aided Verification CAV '90 (New Brunswick, New Jersey, USA)*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, Berlin, June 1990. Springer Verlag.

[Dam94a] M. Dam. CTL* and ECTL* as Fragments of the Modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.

[Dam94b] M. Dam. Model Checking Mobile Processes (Full version). Research Report RR 94:1, Swedish Institute of Computer Science, Kista, Sweden, 1994.

[Der95] Nachum Dershowitz. 33 Examples of Termination. In Hubert Comon and Jean-Pierre Jouannaud, editors, *Term rewriting*, volume 909 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[Dic86] A. Dicky. An Algebraic and Algorithmic Method for Analysing Transition Systems. *Theoretical Computer Science*, 46(2-3):285–303, 1986.

[dNV90] R. de Nicola and F. Vaandrager. Three Logics for Branching Bisimulation. In *Proceedings of the 5th Symposium in Logic in Computer Science LICS '90 (Philadelphia, USA)*, pages 118–129, Los Alamitos, CA, June 1990. IEEE Computer Society Press.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing Correctness Properties of Parallel Programs using Fixpoints. In *Proceedings of the 7th ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181, Berlin, 1980. Springer Verlag.

[EC82] E. A. Emerson and E. M. Clarke. Using Branching Time Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EH83] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages POPL '83 (Austin, Texas)*, pages 127–140, 1983. Also appeared in Journal of ACM, 33(1):151-178, 1986.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On Model-Checking for Fragments of μ-calculus. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification CAV '93 (Elounda, Greece)*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396, Berlin, June-July 1993. Springer Verlag.

[EL85] E. A. Emerson and C-L. Lei. Modalities for Model Checking: Branching Time Logic Strikes Back. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages POPL '85 (New Orleans)*, pages 84–96, 1985. Also appeared in Science of Computer Programming, 8:275-306, 1987.

[EL86] E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of the 1st LICS*, pages 267–278, 1986.

[Eme83] E. A. Emerson. Alternative Semantics for Temporal Logics. *Theoretical Computer Science*, 26(1-2):121–130, September 1983.

[ES89] E. A. Emerson and J. Srinivasan. *Branching Time Temporal Logic*. In G. Rozenberg J. W. de Bakker, W-P. de Roever, editor, *Linear time, branching time and partial order in logics and models of concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 123–172. 1989.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FL79] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, (18):194–211, 1979.

[Gar89a] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar89b] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[Gar94] Hubert Garavel. Binary Coded Graphs — Definition of the BCG Format (version 1.0). Rapport interne, INRIA Rhône-Alpes, Grenoble, 1994.

[Gar98] H. Garavel. OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proceedings of 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, Lecture Notes in Computer Science, Berlin, March 1998. Springer Verlag.

[GJM+97] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. CADP'97 – Status, Applications and Perspectives. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997.

[GP90] J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. Technical Report CS-R9076, Centrum voor Wiskunde en Informatica, Amsterdam, December 1990.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the Temporal Analysis of Fairness. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages POPL '80 (Las Vegas, Nevada)*, pages 163–173, January 1980.

[GRRV89] Susanne Graf, Jean-Luc Richier, Carlos Rodríguez, and Jacques Voiron. What are the Limits of Model Checking Methods for the Verification of Real Life Protocols? In Joseph Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 275–285. Springer Verlag, June 1989.

[GS86] S. Graf and J. Sifakis. A Logic for the Description of Non-deterministic Programs and Their Properties. *Information and Control*, 68(1–3):254–270, 1986.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[GT93] Hubert Garavel and Philippe Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'93 (Montréal, Canada)*, 1993.

[GvdP93] J. F. Groote and J. C. van de Pol. A Bounded Retransmission Protocol for Large Data Packets. Technical Report Logic Group Preprint Series 100, Utrecht University, October 1993.

[GvV94] J. F. Groote and S. M. F. van Vlijmen. A Modal Logic for μCRL. Technical Report 114, Logic Group Preprint Series, Department of Philosophy, Utrecht University, 1994.

[HHY90] K. Hamaguchi, H. Hiraishi, and S. Yajima. Branching Time Regular Temporal Logic for Model Checking with Linear Time Complexity. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of the 2nd International Conference on Computer Aided Verification CAV '90 (New Brunswick, New Jersey, USA)*, volume 531 of *Lecture Notes in Computer Science*, pages 253–262, Berlin, June 1990. Springer Verlag.

[HL92] M. Hennessy and X. Lin. Symbolic Bisimulations. Report 1/92, School of Cognitive and Computing Sciences, University of Sussex, 1992.

[HL93] M. Hennessy and X. Liu. A Modal Logic for Message Passing Processes. Report 3/93, School of Cognitive and Computing Sciences, University of Sussex, January 1993.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32:137–161, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.

[HSV94] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-Checking a Data Link Protocol. In H. P. Barendregt and T. Nipkow, editors, *Proceedings of the 1st International Workshop "Types for Proofs and Programs," May 1993 (Nijmegen)*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165, Berlin, 1994. Springer Verlag.

[HT87] T. Hafer and W. Thomas. Computation Tree Logic CTL* and Path Quantifiers in the Monadic Theory of the Binary Tree. In *Proceedings of the 14th ICALP (Karlsruhe, Germany)*, volume 267 of *Lecture Notes in Computer Science*, pages 269–279, Berlin, July 1987. Springer Verlag.

[IEE95] IEEE. Standard for a High Performance Serial Bus. IEEE Standard 1394-1995, Institution of Electrical and Electronic Engineers, 1995.

[IS94] A. Ingolfsdottir and B. Steffen. Characteristic Formulae for Processes with Divergence. *Information and Computation*, 110(1):149–163, June 1994.

[ISO88a] ISO/IEC. ESTELLE — A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[ISO88b] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[IT92] ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, Genève, 1992.

[JJ89] C. Jard and T. Jéron. *On-Line Model-Checking for Finite Linear Temporal Logic Specifications.* In J. Sifakis, editor, *Automatic Verification of Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 189–196. 1989.

[JJ91] C. Jard and T. Jéron. Bounded-memory Algorithms for Verification On-the-fly. In K. G. Larsen and A. Skou, editors, *Proceedings of 3rd Workshop on Computer Aided Verification CAV '91 (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 192–202, Berlin, July 1991. Springer Verlag.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* North-Holland, 1952.

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. *Math. Syst. Theory*, 2:127–145, 1968.

[Koz83] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton Series in Computer Science. Princeton University Press, 1994.

[Lam80] L. Lamport. "Sometime" is Sometimes "Not Never". On the Temporal Logic of Programs. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages POPL '80 (Las Vegas, Nevada)*, pages 163–173, January 1980.

[Lam83] L. Lamport. What Good is Temporal Logic? *Information Processing*, 83:657–668, 1983.

[Lar88] K. G. Larsen. Proof Systems for Hennessy-Milner logic with Recursion. In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming CAAP '88 (Nancy, France)*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230, Berlin, March 1988. Springer Verlag.

[Lar92] K. G. Larsen. Efficient Local Correctness Checking. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of 4th International Workshop in Computer Aided Verification CAV '92 (Montréal, Canada)*, volume 663 of *Lecture Notes in Computer Science*, pages 30–43, Berlin, June-July 1992. Springer Verlag.

[Lar94] F. Laroussinie. *Logique temporelle avec passé pour la spécification et la vérification des systèmes réactifs.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1994.

[LBC+94] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An Improved Algorithm for the Evaluation of Fixpoint Expressions. In D. L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification CAV '94 (Stanford, California, USA)*, volume 818 of *Lecture Notes in Computer Science*, pages 338–350, Berlin, June 1994. Springer Verlag.

[LP85] O. Lichtenstein and A. Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages POPL '85 (New Orleans)*, pages 97–107, 1985.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The Glory of the Past. In *Proceedings of the International Conference of Logics of Programs (Brooklyn)*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer Verlag, June 1985.

[Lut97] Bas Luttik. Description and Formal Specification of the Link Layer of P1394. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997. Also available as CWI Technical Report SEN-R9706.

[Mat93] Radu Mateescu. Optimisation de la compilation des types abstraits algébriques du langage LOTOS. Mémoire d'ingénieur, Institut Polytechnique de Bucarest, September 1993.

[Mat96] R. Mateescu. Formal Description and Analysis of a Bounded Retransmission Protocol. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 98–113. University of Maribor, Slovenia, June 1996. Also available as INRIA Research Report RR-2965.

[Mat97] Radu Mateescu. Vérification de systèmes répartis : l'exemple du protocole BRP. *Technique et Science Informatiques*, 16(6):725–751, June 1997.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MP89] Z. Manna and A. Pnueli. *The Anchored Version of the Temporal Framework.* In G. Rozenberg J. W. de Bakker, W-P. de Roever, editor, *Linear time, branching time and partial order in logics and models of concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 201–284. 1989.

[MP90] Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Proceedings of the 9th ACM Symp. on Principles of Distributed Computing*, pages 377–408, Quebec, August 1990.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, volume I: Specification.* Springer-Verlag, 1992.

[MV93] S. Mauw and G. J. Veltink. *Algebraic Specification of Communication Protocols.* Number 36 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[NFGR91] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An Action-Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. In K. G. Larsen and A. Skou, editors, *Proceedings of 3rd Workshop on Computer Aided Verification CAV '91 (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 37–47, Berlin, July 1991. Springer Verlag.

[NV90] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[Pec98] Charles Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. Research Report RR-3416, INRIA, Grenoble, May 1998. CFS.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report FN-19, Daimi, Aarhus, 1981.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Que82] J.-P. Queille. *Le systéme CESAR : description, spécification et analyse des applications reparties.* Thèse de Doctorat, Université Scientifique et Médicale de Grenoble, June 1982.

[Que97] Juan Quemada, editor. Committee Draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG7 Project 1.21.20.2.3, January 1997.

[Ras90] Anne Rasse. *CLEO : diagnostic des erreurs en XESAR.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, June 1990.

[RH96] J. Rathke and M. Hennessy. Local Model Checking for a Value-Based Modal μ-calculus. Report 5/96, School of Cognitive and Computing Sciences, University of Sussex, June 1996.

[Rod88] Carlos Rodríguez. *Spécification et validation de systèmes en XESAR.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, May 1988.

[RRSV87] Jean-Luc Richier, Carlos Rodríguez, Joseph Sifakis, and Jacques Voiron. Verification in XESAR of the Sliding Window Protocol. In Harry Rudin and Colin H. West, editors, *Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification (Zurich).* IFIP, North-Holland, May 1987.

[Ruf94] Renaud Ruffiot. Définition et réalisation d'un atelier logiciel pour l'étude des systèmes de transitions. Mémoire d'ingénieur CNAM, INRIA Rhône-Alpes, Grenoble, December 1994.

[SC85] A. P. Sistla and E. M. Clarke. Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Sch83] J.P. Schwartz. *QUASAR : une réalisation du système CESAR.* Thèse de Docteur-Ingénieur, Université de Grenoble, 1983.

[Sch88] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Wm. C. Brown, 1988.

[Sig94] Mihaela Sighireanu. Implémentation optimisée des types abstraits algébriques du langage LOTOS. Mémoire d'ingénieur de l'Institut Polytechnique de Bucarest, VERIMAG, Grenoble, September 1994.

[SM97] Mihaela Sighireanu and Radu Mateescu. Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997. Full version available as INRIA Research Report RR-3172.

[Sti87] C. Stirling. Modal Logics for Communicating Systems. *Theoretical Computer Science*, 49(2-3):311–347, 1987.

[Sti92] C. Stirling. Modal and Temporal Logics for Processes. Technical Report ECS-LFCS-92-221, LFCS, Dept. of Computer Science, University of Edinburgh, 1992.

[Str82] R. Streett. Propositional Dynamic Logic of Looping and Converse. *Information and Control*, (54):121–141, 1982.

[SVW87] A. P. Sistla, M. Y. Vardi, and P. Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *Theoretical Computer Science*, 49(2-3):217–237, 1987.

[SW91] C. Stirling and D. Walker. Local Model Checking in the Modal Mu-Calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[Tar55] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, (5):285–309, 1955.

[Tho89] W. Thomas. *Computation Tree Logic and regular ω-languages.* In G. Rozenberg J. W. de Bakker, W-P. de Roever, editor, *Linear time, branching time and partial order in logics and models of concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 690–713. 1989.

[Var88] M. Vardi. A Temporal Fixpoint Calculus. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages POPL '88 (San Diego, California)*, pages 250–259, January 1988.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[VL92] B. Vergauwen and J. Lewi. A Linear Algorithm for Solving Fixed-Point Equations on Transition Systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming CAAP '92 (Rennes, France)*, volume 581 of *Lecture Notes in Computer Science*, pages 322–341, Berlin, February 1992. Springer Verlag.

[VL93] B. Vergauwen and J. Lewi. A Linear Local Model-Checking Algorithm for CTL. In E. Best, editor, *Proceedings of CONCUR '93 (Hildesheim, Germany)*, volume 715 of *Lecture Notes in Computer Science*, pages 447–461, Berlin, August 1993. Springer Verlag.

[VL94] B. Vergauwen and J. Lewi. Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems. In S. Abiteboul and E. Shamir, editors, *Proceedings of the 21st ICALP (Vienna)*, volume 820 of *Lecture Notes in Computer Science*, pages 304–315, Berlin, July 1994. Springer Verlag.

[VM94] B. Victor and F. Moller. The Mobility Workbench – A Tool for the π-Calculus. In D. L. Dill, editor, *Proceedings of the 6th Conference on Computer-Aided Verification CAV '94 (Stanford, California, USA)*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440, Berlin, June 1994. Springer Verlag.

[VWL94] B. Vergauwen, J. Wauman, and J. Lewi. Efficient FixPoint Computation. In *Proceedings of the 1st International Static Analysis Symposium SAS '94 (Namur, Belgium)*, volume 864 of *Lecture Notes in Computer Science*, pages 314–328, Berlin, September 1994. Springer Verlag.

[Win91] G. Winskel. A Note on Model Checking the Modal ν-calculus. *Theoretical Computer Science*, 83(1):157–167, June 1991.

[Wol83] P. Wolper. Temporal Logic can be More Expressive. *Information and Control*, 56(1/2):72–99, January-February 1983.

Résumé : La vérification formelle est indispensable pour assurer la fiabilité des applications critiques comme les protocoles de communication et les systèmes répartis. La technique de vérification basée sur les modèles (*model-checking*) consiste à traduire l'application vers un système de transitions étiquetées (STE), sur lequel les propriétés attendues, exprimées en logique temporelle, sont vérifiées à l'aide d'outils appelés évaluateurs (*model-checkers*). Cependant, les logiques temporelles "classiques", définies sur un vocabulaire d'actions atomiques, ne sont pas adaptées aux langages de description comme LOTOS, dont les actions contiennent des valeurs typées.

Cette thèse définit un formalisme appelé XTL (*eXecutable Temporal Language*) qui permet d'exprimer des propriétés temporelles portant sur les données du programme à vérifier. XTL est basé sur une extension du μ-calcul modal avec des variables typées. Les valeurs contenues dans le STE, extraites à l'aide d'opérateurs modaux étendus, peuvent être passées en paramètre aux opérateurs de point fixe ou manipulées à l'aide de constructions d'inspiration fonctionnelle comme "let", "if-then-else", "case", etc. Les propriétés portant sur des séquences d'actions du programme sont décrites succinctement au moyen d'expressions régulières. Des méta-opérateurs spéciaux permettent l'évaluation des formules sur un STE et l'expression de propriétés temporelles non-standard par exploration de la relation de transition.

La sémantique de XTL est formellement définie et des algorithmes efficaces sont proposés pour évaluer des formules temporelles XTL sur des modèles STEs. Un évaluateur XTL est développé et utilisé avec succès pour la validation d'applications industrielles comme le protocole BRP développé par Philips et la couche liaison du bus série IEEE-1394 ("FireWire").

Mots-clés : logique temporelle, LOTOS, mu-calcul, spécification, système de transitions étiquetées, validation, vérification.

Abstract: Formal verification is essential in order to ensure reliability of critical applications like communication protocols and distributed systems. The so-called *model-checking* verification technique consists in translating the application into a Labelled Transition System (LTS) on which the desired properties, expressed in temporal logic, are verified using specialized tools called *model-checkers*. However, the "classical" temporal logics, defined over an alphabet of atomic actions, are not well-adapted for description languages as LOTOS, whose actions contain typed values.

This thesis defines a formalism called XTL (*eXecutable Temporal Language*), which allows to express temporal properties involving the data handled by the program to be verified. XTL is based upon an extension of the modal μ-calculus with typed variables. The values contained in the LTS, extracted using extended modal operators, can be passed as arguments to the fixed point operators or can be combined by means of functional-like constructs as "let", "if-then-else", "case", etc. The properties over action sequences of the program can be described succinctly using regular expressions. Special meta-operators allow to evaluate formulas on an LTS as well as to express non-standard temporal properties by exploring the transition relation.

The semantics of XTL is formally defined, and efficient algorithms are proposed for the evaluation of temporal XTL formulas over LTS models. A model-checker for XTL is developed and successfully used for the validation of industrial applications such as the BRP protocol designed by Philips and the link layer of the IEEE-1394 serial bus ("FireWire").

Key-words: labelled transition system, LOTOS, mu-calculus, specification, temporal logic, validation, verification.