Spécification et validation de systèmes en Xesar

Carlos Rodriguez

THESE

présentée par

Carlos RODRIGUEZ

pour obtenir le titre de **DOCTEUR**
de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(arrêté ministériel du 5 juillet 1984)
Spécialité : *INFORMATIQUE*

SPECIFICATION ET VALIDATION DE SYSTEMES EN XESAR

Thèse soutenue le 27 mai 1988
Composition du jury :

Président : J.P. Verjus
Examinateurs : A. Arnold
E. Hehner
J. Sifakis
J. Voiron
P. Wolper

Thèse préparée au sein du Laboratoire de Génie Informatique

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD

Année 1988

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G

BOREL Joseph

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

ROWE Alain

E.N.S.I.M.A.G

COURTIN Jacques

E.F.P.

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Remerciements

Je remercie mon responsable de recherche, Monsieur Joseph SIFAKIS, pour m'avoir accuelli au sein de son équipe et pour l'impulsion qu'il a donnée à ces travaux par ces critiques et suggestions. Je le remercie également pour sa participation active dans la conception de l'outil XESAR.

Je tiens à remercier :

- M. André ARNOLD, Professeur à l'Université de Bordeaux I
- M. Eric HEHNER, Professeur associé à l'ENSIMAG
- M. Jean-Pierre VERJUS, Directeur de recherche au CNRS
- M. Jacques VOIRON, Maître de Conférences à l'Université Joseph Fourier
- M. P. WOLPER, Professeur à l'Université de Liège

d'avoir accepté de participer à ce jury de thèse. Je remercie particulièrement Monsieur J.P. VERJUS d'avoir accepté d'en être le président, MM. A. ARNOLD et P. WOLPER d'en être les rapporteurs et M. J. VOIRON pour sa participation active dans la conception et la réalisation de XESAR.

Enfin, ce travail n'aurait pu être réalisé sans l'environnement particulièrement favorable dans lequel il s'est déroulé : je tiens à remercier tous les paticipants, actifs ou passifs, au projet GOUTER.

Table des Matières

1 Introduction. 1

I Etude du langage de spécification. 7

2 Le Mu-calcul arborescent. 9

2.1 Syntaxe. 9

2.2 Sémantique. 10

2.3 Relation entre le Mu-calcul et les logiques arborescentes. . . . 13

2.4 Exemple de spécification en XESAR. 14

2.4.1 Les prédicats de base en XESAR. 14

2.4.2 Sémantique des prédicats de base. 15

2.4.3 Spécification d'un protocole de fenêtre glissante. . . . 15

2.5 Conclusion. 17

3 Les graphes de sûreté. 19

3.1 Introduction. 19

3.2 La relation d'implantation sûre. 22

3.3 Les réductions de graphes. 28

3.4 Des réductions optimales de graphes. 31

3.5 La relation d'implantation sûre et les graphes de sûreté. . . . 34

3.6 Caractérisation logique de $\sqsubseteq$. 36

3.7 Application des graphes de sûreté. 40

3.7.1 Les graphes associés aux formules not-to-unless. . . . 41

3.7.2 Le comptage d'occurrences d'actions. 46

3.8 Exemples de spécification. 48
3.8.1 Le protocole de Stenning. 48
3.8.2 Un protocole de transport. 51
3.9 Conclusion. 53

II Réalisation du vérificateur de XESAR. 55

4 Présentation générale de XESAR. 57
4.1 Le langage ESTELLE/R. 57
4.2 Organisation de XESAR. 59
4.3 Transformation du programme ESTELLE/R vers une forme intermédiaire. 62
4.3.1 La génération des automates. 62
4.3.2 La composition des automates. 64
4.3.3 Le modèle sémantique de ESTELLE/R 64

5 La génération de graphes d'états en XESAR. 67
5.1 Le *type* des transitions. 68
5.2 Le graphe d'états. 69
5.3 Mise en œuvre du simulateur. 70
5.4 Entrées-Sorties. 72
5.4.1 Entrées 72
5.4.2 Sorties. 73
5.5 Structures de données. 73
5.6 L'algorithme détaillé de génération des graphes. 75
5.7 Conclusion 76

6 L'évaluateur de formules de XESAR. 79
6.1 Décomposition fonctionnelle. 79
6.2 Entrées et sorties : 82
6.2.1 Entrées : 82
6.2.2 Sorties : 84
6.3 La transformation *after*. 84
6.4 Structures de données. 87

6.4.1 L'ensemble des arcs du graphe. 87
6.4.2 Le tableau des ensembles d'actions. 87
6.4.3 Les registres-prédicats. 88
6.4.4 Tableau des valeurs des expressions booléennes. 89
6.4.5 Le tableau d'images. 89
6.5 La syntaxe abstraite des formules. 89
6.6 La représentation interne des formules. 93
6.7 L'évaluation des formules. 95
6.8 Algorithmes : 96
6.8.1 Les formules *sink* et *init*. 97
6.8.2 Les formules *enable*(*ens*) et *after*(*ens*). 98
6.8.3 Les formules $pre(f)$ et $\widetilde{pre}(f)$. 106
6.8.4 Les formules $<B>f$ et $B.f$. 107
6.8.5 Les formules $pot[f_1]f_2$, $al[f_1]f_2$ et $inev[f_1]f_2$. 109
6.8.6 Les formules $B_1.f_1 + \ldots + B_n.f_n$. 111
6.8.7 L'évaluation des formules de point fixe. 112
6.8.8 L'évaluation de systèmes d'équations. 115

7 Utilisation de XESAR. 117

8 Conclusions. 123

A Syntaxe concrète d'ESTELLE/R. 131

B Description du protocole de Stenning. 141

C Algorithmes des procédures utilisées par la transformation *after*. 147

D Syntaxe abstraite des synonymes et des expressions booléennes. 151

E Descriptions des menus de XESAR. 155

Chapitre 1

Introduction.

L'utilisation répandue des systèmes distribués a conduit à un accroissement de leur complexité. Face à cette complexité, il a été nécessaire de développer des méthodes de validation permettant la construction de systèmes corrects et fiables.

La validation consiste à comparer un système à ses spécifications. Les deux méthodes de validation les plus utilisées sont :

- La validation par simulation qui teste la conformité aux spécifications pour certaines séquences de fonctionnement significatives.
- La vérification formelle qui consiste à comparer une description du système à des propriétés de son comportement (de l'ensemble de ses séquences de fonctionnement).

La vérification formelle a été considérée d'un intérêt pratique limité puisque, dans le cas général, la description de toutes les séquences de fonctionnement d'un système peut être impossible. Pour l'utiliser, il est nécessaire de se limiter à des applications qui, bien qu'étant complexes du point de vue conception et programmation, peuvent être décrites en utilisant un nombre fini d'états. Dans ce travail, nous nous intéressons à une de ces applications, les protocoles de communication, et nous avons choisi la vérification formelle pour les valider.

Lorsque l'on vérifie, on compare deux descriptions d'un système. Ces descriptions peuvent être données dans des formalismes de types différents ou peuvent ne pas être du même niveau. Par convention, on appelle spécification la description la plus abstraite et implantation l'autre description. Pour vérifier la conformité d'une implantation I à sa spécification S, on définit une relation $\mathcal{R}$. $I \mathcal{R} S$ signifie que I satisfait la spécification S.

Les formalismes les plus utilisés pour décrire les systèmes sont ceux

auxquels on peut associer une sémantique basée sur les systèmes de transitions. Les langages de programmation sont des formalismes de ce type dans la mesure où ils sont définis pour une machine abstraite qui les interprète.

La description des systèmes en utilisant un de ces formalismes donne pour la vérification deux approches :

L'approche du langage unique : la spécification et l'implantation sont exprimées en utilisant le même langage. En général le langage choisi est de type algébrique et la relation $\mathcal{R}$ est une relation d'équivalence entre les termes de l'algèbre. Déterminer si un terme implantation I est conforme à un terme spécification S revient à décider si I est équivalent à S.

L'approche des deux langages : la spécification est exprimée dans un langage différent de celui utilisé pour l' implantation. Le langage de spécification est plus abstrait et décrit ce que l'on attend du système. En général, le langage utilisé est un langage de formules logiques et la relation $\mathcal{R}$ utilisée pour comparer les descriptions et les spécifications est une relation de satisfaction $\models$. Dans ce cas, si I est conforme à S on écrit $I \models S$.

Ces deux approches sont complémentaires. Des techniques de vérification basées sur le calcul d'équivalences entre termes se trouvent dans : [Mil80], [Mil83], [BK84], [AB84], [BHR84], [Sor87]. Des utilisations de spécifications logiques de programmes décrits par des langages algorithmiques se trouvent dans : [Pnu77], [Abr79], [Lam83], [QS83], [AMP83], [CE85].

Ce travail s'inscrit dans le cadre du projet CESAR. Ce projet a comme but le développement d'outils de vérification de systèmes distribués en utilisant l'approche des deux langages. D'une façon générale, le fonctionnement de ces outils (voir figure 1.1) est le suivant : le programme décrivant le système est traduit en une forme intermédiaire. A partir de cette forme intermédiaire et par simulation exhaustive, on génère un graphe d'états, modèle de la logique. Ensuite, les formules composant la spécification sont évaluées sur le modèle pour décider si l'implantation est conforme à la spécification.

A présent, il existe deux outils développés en suivant ce principe, et un troisième en cours de réalisation, différenciés principalement par le langage de description.

Le premier outil, appelé QUASAR([Sch83]), utilise un langage inspiré de CSP ([Hoa78]) où un programme est un ensemble de tâches communicantes et chaque tâche est un ensemble des commandes gardées. Le langage de spécification est la logique CTL ([Sch83]).

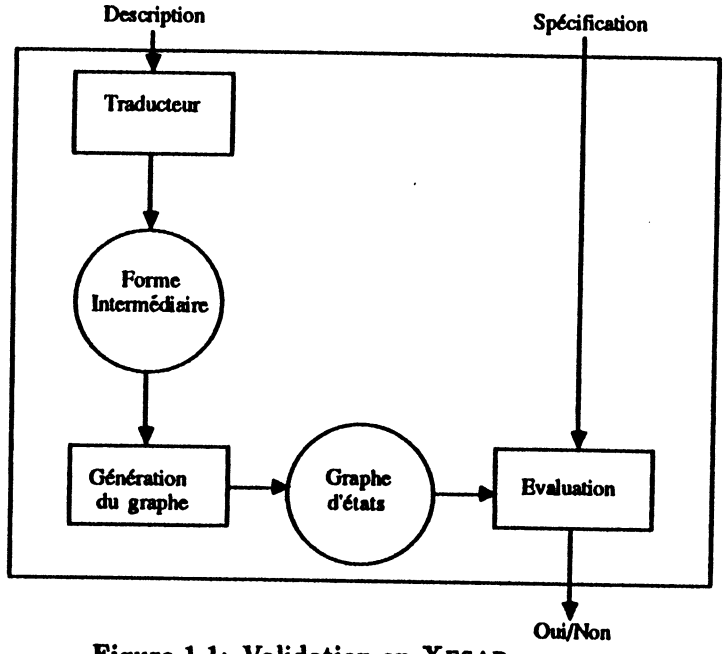

Figure 1.1: Validation en XESAR.

Le deuxième outil, XESAR([RRSV87b]), est un outil de vérification des protocoles de communication. Il a été développé dans le cadre d'un contrat avec le Centre National d'Etudes de Télécommunications (CNET). En XESAR, les descriptions sont faites en ESTELLE/R, un langage dérivé de ESTELLE ([Est85]), langage proposé par l'ISO pour la description de protocoles. En ESTELLE/R chaque tâche est un automate séquentiel communicant, dont les transitions sont décrites par un sur-ensemble du langage PASCAL. Le langage de spécification de XESAR est décrit dans ce mémoire. Il est basé sur le Mu-calcul arborescent STL ([GS86]) et il a été conçu pour rendre plus facile les spécifications des propriétés.

Nous avons participé à la conception et mise en œuvre de XESAR. Dans le cadre de ce projet, notre travail a consisté, d'une part en l'implantation de l'évaluateur de formules et d'autre part, en l'étude des problèmes que pose l'utilisation d'un tel outil. Ces problèmes sont de deux types : l'expression des propriétés attendues d'un système et la méthode de vérification de ces propriétés.

Expression des propriétés.

Le choix d'un langage approprié pour l'expression des spécifications des protocoles est d'une importance cruciale pour l'utilisation de XESAR. L'expérience avec les logiques temporelles utilisées dans QUASAR, montre qu'elles mènent à des descriptions difficiles à formuler et à interpréter. En effet, certaines situations que l'on peut caractériser aisement par des com-

portements (systèmes de transitions), sont difficiles à décrire par des logiques temporelles.

Pour cette raison, en XESAR nous proposons comme langage de spécification un Mu-calcul propositionnel obtenu par extension du langage de termes d'un calcul de processus. Ce Mu-calcul a été étudié dans [GS86] et présente l'avantage de pouvoir exprimer aussi bien les opérateurs de la logique temporelle arborescente que les comportements réguliers.

Notre objectif a été l'adaptation de ce langage à la spécification de protocoles de communication. Nous avons constaté que la spécification du service d'un protocole peut s'effectuer en utilisant un nombre très limité de classes de propriétés ; par exemple, des propriétés exprimant la préservation de l'ordre des messages, des relations de causalité, des contraintes temporelles, etc. Par ailleurs, parmi les propriétés caractérisant le service d'un protocole, seules les propriétés de sûreté ([Lam83],[Pnu77]) sont valides pour les descriptions dont on dispose habituellement. En effet, celles-ci modélisent les pertes de message par des choix non-déterministes, ce qui dans l'absence d'hypothèse d'équité rend les propriétés de vivacité non-valides.

Pour cette raison, nous nous sommes plus particulièrement intéressés à l'expression et à la vérification des propriétés de sûreté. Nous avons défini une relation d'équivalence $\approx$ sur les systèmes de transitions qui préserve les propriétés de sûreté. Ceci signifie que la classe d'un système modulo cette relation représente exactement les systèmes satisfaisant les mêmes propriétés de sûreté. L'utilisation de cette relation d'équivalence permet de simplifier les modèles de la logique tout en préservant la relation de satisfaction pour les propriétés de sûreté.

Vérification des propriétés.

La principale limitation des outils comme XESAR est liée à la complexité du modèle généré et à la complexité des algorithmes de vérification. Nous avons étudié les différents choix de représentation du modèle et les choix d'implantation des algorithmes de vérification.

En ce qui concerne la représentation du modèle, nous avons étudié différentes organisations de données et des méthodes d'accès associées pour minimiser le temps de sa génération. En ce qui concerne les algorithmes de vérification, nous avons implanté aussi bien les algorithmes généraux pour l'évaluation des points fixes que des algorithmes adaptés au calcul de point fixes particuliers qui caractérisent les opérateurs de la logique temporelle arborescente. Ceci a été fait dans un but d'efficacité puisque ces derniers algorithmes sont d'une complexité linéaire par rapport à la taille du modèle.

Organisation de ce document.

La première partie de ce travail, intitulée *Etude d'un langage de spécification*, comporte deux chapitres :

Dans le chapitre 2, un langage de spécification de XESAR est décrit. On donne un exemple d'utilisation en spécifiant un protocole de fenêtre glissante.

Dans le chapitre 3 la relation d'implantation sûre entre graphes d'états est définie. Elle nous permet d'exprimer des ensembles des propriétés de sûreté sans recourir au langage logique.

La deuxième partie, *Réalisation du vérificateur de* XESAR, contient les chapitres suivants :

Le chapitre 4 contient une présentation générale de XESAR : on décrit succinctement la transformation du programme ESTELLE/R en un modèle sur lequel sont évaluées les formules qui composent la spécification.

Les chapitres 5, 6 et 7 se réfèrent à la mise en œuvre de XESAR. Le chapitre 5 décrit d'une manière sommaire l'implantation du générateur des graphes d'états (simulateur). Le chapitre 6 décrit la réalisation de l'évaluateur de formules. On détaille les structures de données choisies et les algorithmes implantés. Le chapitre 7 contient des exemples d'utilisation de XESAR.

Partie I

Etude du langage de spécification.

Chapitre 2

Le Mu-calcul arborescent.

Le langage de spécification de XESAR est le langage des formules logiques généré à partir d'un ensemble de prédicats de base $\mathcal{P}$ en utilisant des opérateurs du Mu-calcul arborescent STL ([GS86]). Dans ce chapitre, nous présentons sa syntaxe, sa sémantique et un exemple de spécification.

2.1 Syntaxe.

Soit E un ensemble de noms d'actions, $\mathcal{P}$ un ensemble de variables propositionnelles et $\mathcal{X}$ un ensemble de variables. Le langage des formules $\mathcal{F}$ est le langage des formules généré par les règles :

$$f ::= \top \mid p \mid x \mid f_1 \vee f_2 \mid \neg f \mid init \mid after(B) \mid B.f \mid B_1.f_1 + B_2.f_2 \mid \mu x.f(x)$$

où x est un élément de $\mathcal{X}$, f , f_1 et f_2 sont des formules, B, B_1 et B_2 sont des sous-ensembles de E, p est une variable propositionnelle et $\mu x.f(x)$ est telle que toute occurrence de x dans $f(x)$ est sous un nombre pair de négations.

On utilise les abréviations suivantes :

- $\bot = \neg\top$
- $f_1 \wedge f_2 = \neg(\neg f_1 \vee \neg f_2)$
- $f_1 \supset f_2 = \neg f_1 \vee f_2$
- $f_1 \equiv f_2 = f_1 \supset f_2 \wedge f_2 \supset f_1$
- $<B>f = B.f + E.\top$
- $[B]f = \neg<B>\neg f$

- $enable(B) = \langle B \rangle \top$
- $sink = \neg \langle E \rangle \top$
- $\nu x.f(x) = \neg \mu x. \neg f(\neg x)$

2.2 Sémantique.

Définitions

Définition 1 *Un graphe d'états est une structure* $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ *où :*

- $E = \{a_1, \ldots, a_m\}$ *est un ensemble d'actions.*
- $Q = \{q_1, \ldots, q_n\}$ *est un ensemble d'états*
- $\{\xrightarrow{a}\}_{a \in E}$ *est la relation de transition entre les états de l'ensemble Q. Chaque* $\xrightarrow{a}$ *est un sous-ensemble de* $Q \times Q$. *On note* $q \xrightarrow{a} q'$ *si et seulement si* $(q, q') \in \xrightarrow{a}$.
- i_G *est l'état initial de G.*

Pour des raisons de lisibilité, certains algorithmes présentés dans la suite utilisent la notation R_a pour $\xrightarrow{a}$ et R_E pour $\{\xrightarrow{a}\}_{a \in E}$.

Définition 2 *Soit* $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ *un graphe d'états. Les fonctions* $act : Q \to 2^E$ *et* $suc : Q \times E \to 2^Q$ *sont définies par :*

$$act(q) = \{a \in E | \exists q' \in Q,\ q \xrightarrow{a} q'\}$$

$$suc(q, a) = \{q' | q \xrightarrow{a} q'\}$$

L'ensemble $act(q)$ contient les actions exécutables à partir de l'état q. L'ensemble $suc(q, a)$ contient les états accessibles à partir de q par l'exécution de l'action a.

Définition 3 *Soient* $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ *un graphe d'états,* $\alpha = a_1, \ldots, a_n$ *et* $\gamma = q_0, \ldots, q_n$ *des séquences d'actions et d'états, respectivement. On note :*

$$q_0 \xrightarrow[\gamma]{\alpha} q_n$$

le fait que $\forall\, 0 \leq i < n, q_i \xrightarrow{a_{i+1}} q_{i+1}$

$q_0 \xrightarrow{\alpha} q_n$ *signifie qu'il existe une séquence* γ *telle que* $q_0 \xrightarrow[\gamma]{\alpha} q_n$

$q_0 \xrightarrow[\gamma]{} q_n$ *signifie qu'il existe une séquence* α *telle que* $q_0 \xrightarrow[\gamma]{\alpha} q_n$

Définition 4 *Soient* $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ *un graphe d'états et* q *un état de* Q. *Une exécution à partir de* q *est un couple de séquences infinies* (α, γ),

$$\alpha = a_1 \ldots a_i \ldots \text{ et } \gamma = q_0 q_1 \ldots q_i \ldots$$

ou un couple de séquences finies (α, γ),

$$\alpha = a_1 \ldots a_n \text{ et } \gamma = q_0 \ldots q_n$$

telles que $q_0 = q, \forall\, i \geq 0, q_i \xrightarrow{a_{i+1}} q_{i+1}$ *et si* γ *et* α *sont finies, alors* $act(q_n) = \emptyset$. *L'ensemble d'exécutions à partir de* q *est noté* $EX(q)$.

Si une exécution (α, γ) est telle que

$$\alpha = a_1 \ldots a_i, \ldots \text{ et } \gamma = q_0 q_1 \ldots q_i \ldots$$

on note γ_i l'etat q_i, $k \geq 0$ et α_i l'action a_i.

Interprétation intuitive des éléments de la logique.

Les modèles pour notre logique sont des graphes d'états. Chaque formule f du langage $\mathcal{F}$ représente un ensemble d'états, appelé l'ensemble caractéristique de f. On dit qu'un état satisfait f si et seulement si, il appartient à son ensemble caractéristique. Intuitivement, l'ensemble caractéristique associé à une formule est défini comme suit :

- A la formule $\top$ est associé l'ensemble contenant tous les états du graphe.
- A la formule $\bot$ est associé l'ensemble vide.
- Aux formules $\neg f$, $f \vee g$ et $f \wedge g$ sont associés respectivement, le complémentaire, l'union et l'intersection des ensembles caractéristiques de leurs arguments.
- Les formules $[B]f, \langle B \rangle f$ et $B.f$ expriment le fait que des états caractérisés par f peuvent ou doivent être atteints après l'exécution d'une transition.

- A la formule $<B>f$ est associé l'ensemble des états à partir desquels il est possible d'atteindre un état satisfaisant f en exécutant une action contenue dans l'ensemble B.
- A la formule $[B]f$ est associé l'ensemble des états à partir desquels toutes les transitions étiquetées par des éléments de B mènent à des états satisfaisant f.
- A la formule $B.f$ est associé l'ensemble des états à partir desquels il existe des transitions étiquetées par des éléments de B qui mènent à des états satisfaisant f, et il n'existe que de telles transitions.

- A la formule $B_1.f_1 + B_2.f_2$ est associé l'ensemble des états à partir desquels il existe un choix non-déterministe entre $B_1.f_1$ et $B_2.f_2$, c'est-à-dire des états à partir desquels :
  - il existe des transitions étiquetées par des éléments de B_1 qui mènent à des états satisfaisant f_1.
  - il existe des transitions étiquetées par des éléments de B_2 qui mènent à des états satisfaisant f_2.
  - toute transition est étiquetée par des éléments de B_1 ou B_2 et mène à un état satisfaisant f_1 ou f_2, respectivement.
- Les opérateurs μ et ν sont, respectivement, les opérateurs de plus petit et plus grand point fixe.

Les modèles.

On appelle *modèle* un graphe d'états $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ plus une fonction d'interprétation $\mathcal{I} : \mathcal{P} \to 2^Q$ qui associe à chaque variable propositionnelle un ensemble d'états. Soit f une formule de $\mathcal{F}$ avec variables libres $\vec{X} = \{x_1, \ldots, x_n\}$. On suppose que le graphe d'états est tel que :

$$\forall q, q', q'' \in Q, \forall a, b \in E, q' \xrightarrow{a} q \wedge q'' \xrightarrow{b} q \implies a = b$$

c'est-à-dire que tous les arcs qui mènent à un ètat portent la même action. La sémantique de f est définie par une fonction $|\ |^G : \mathcal{F}x(2^Q)^n \to 2^Q$. On écrit $f(\vec{X})$ pour indiquer que toutes les variables libres de f appartiennent à $\vec{X}$. Une valuation $\vec{V} = (v_1, .., v_n)$ est une affectation de sous-ensembles de Q aux variables libres $\vec{X}$. On définit la valeur de la fonction $|\ |^G$ appliquée à f et $\vec{V}$:

Sémantique des opérateurs

- $|\top|^G(\vec{V}) = Q$

- $|p|^G(\vec{V}) = \mathcal{I}(p)$
- $|x_i|^G(\vec{V}) = v_i$
- $|f \vee g|^G(\vec{V}) = |f|^G(\vec{V}) \cup |g|^G(\vec{V})$
- $|\neg f|^G(\vec{V}) = Q - |f|^G(\vec{V})$
- $|init|^G = \{i_G\}$
- $|after(B)|^G = \{q \mid \exists a, \exists q', q' \xrightarrow{a} q \wedge \forall q', q' \xrightarrow{a} q \implies a \in B\}$
- $|B_1 f_1 + B_2 f_2|^G(\vec{V}) = \{q \mid (act(q) \cap B_1 \neq \emptyset)$ **et** $(act(q) \cap B_2 \neq \emptyset)$ **et** $\forall\, a, q \xrightarrow{a} q' \implies (a \in B_1$ **et** $q' \in |f_1|^G(\vec{V}))$ **ou** $(a \in B_2$ **et** $q' \in |f_2|^G(\vec{V}))\}$
- $|\mu x.f(x)|^G(\vec{V}) = \cup\{Q' \subseteq Q \mid |f(Q')|^G(\vec{V}) \supseteq Q'\}$

2.3 Relation entre le Mu-calcul et les logiques arborescentes.

Les logiques arborescentes ([CES83],[QS83]) contiennent des modalités temporelles du type $pot[f]g$, $al[f]g$, $inev[f]g$ et $fair[f]g$, dont l'interprétation intuitive est :

- $pot[f]g$ représente les états à partir desquels il est possible d'avoir f jusqu'à ce que g devienne vrai.
- $al[f]g$ représente les états à partir desquels, pour toute évolution γ, le fait que f soit vrai sur un préfixe γ' de γ implique que g est vrai dans tout état q tel que $\gamma'.q$ est aussi un prefixe de γ.
- $inev[f]g$ représente les états à partir desquels, pour toute évolution possible, f est satisfaite jusqu'à ce que g devienne vrai.
- $fair[f]g$ a la même interprétation que $inev[f]g$. La différence entre les deux formules vient du fait que $fair[f]g$ ne considère que des évolutions *équitables* ([QS83]).

Ces logiques contiennent aussi des opérateurs qui caractérisent les ensembles de prédécesseurs de l'ensemble d'états qui satisfont une formule f. Ces opérateurs sont notés pre et $\widetilde{pre}$ et leur interprétation intuitive est :

- $pre(f)$ représente l'ensemble des états à partir desquels il est possible d'atteindre un état satisfaisant f par l'exécution d'une transition.

- $\widetilde{pre}(f)$ représente l'ensemble des états à partir desquels toute transition mène à un état qui satisfait f.

Si l'on utilise des opérateurs du Mu-calcul, ces opérateurs sont exprimés de la façon suivante (l'ensemble E est l'ensemble d'actions du modèle) :

- $pre(f) = <E>f$
- $pot[f]g = \mu x.(g \vee (f \wedge pre(x)))$
- $inev[f]g = \mu x.(g \vee (f \wedge \widetilde{pre}(x) \wedge pre(x)))$

Les opérateurs $\widetilde{pre}$ et al sont les duaux des opérateurs pre et pot, respectivement. Ils s'expriment donc par :

- $\widetilde{pre}(f) = \neg pre(\neg f)$
- $al[f]g = \neg pot[f]\neg g$

L'opérateur *fair* est exprimé en termes des autres opérateurs temporels par l'équation ([QS83]) :

$$fair[f]g = al[not\ g](pot[f](g))$$

2.4 Exemple de spécification en XESAR.

2.4.1 Les prédicats de base en XESAR.

En XESAR les variables propositionnelles sont appelées *prédicats de base*. Elles servent à faire la liaison entre les éléments de la logique et le programme ESTELLE/R. L'ensemble des prédicats de base en XESAR contient :

1. Les expressions booléennes sur des variables du programme.
2. Le prédicat $after(B)$, B étant un sous-ensemble de E.
3. Le prédicat *init*.

Interprétation intuitive des prédicats de base.

- Une expression booléenne permet la caractérisation des états du modèle où l'expression s'évalue à vrai.
- Le prédicat $enable(B)$ caractérise les états à partir desquels il existe des transitions qui portent des actions de l'ensemble B.

- Le prédicat *after*(B) caractérise les états q tels que toute transition qui mène à q porte des actions de l'ensemble B et il existe au moins une transition qui arrive à q (q est accessible).
- Le prédicat *init* caractérise l'état initial du graphe.
- Le prédicats *sink* caractérise les états ne possédant pas de successeur, c'est-à-dire qu'aucune transition n'est exécutable à partir de ces états. Ces états sont appelés états *puits*.

2.4.2 Sémantique des prédicats de base.

On définit les valeurs de la fonction d'interprétation $\mathcal{I}$ sur le graphe $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$.

- $\mathcal{I}(expression) = \{q \mid \text{expression est vraie en } q\}$
- $\mathcal{I}(enable(B)) = \{q \mid \exists a \in act(q),\ a \in B\}$
- $\mathcal{I}(after(B)) = \{q \mid \exists q' \in Q, \exists a \in B, q' \xrightarrow{a} q \text{ et } \forall\ q' \in Q, q' \xrightarrow{a} q \implies a \in B\}$
- $\mathcal{I}(init) = \{\text{état initial du système}\}$
- $\mathcal{I}(sink) = \{q \mid act(q) = \emptyset\}$

2.4.3 Spécification d'un protocole de fenêtre glissante.

Dans cette section on montre une spécification partielle du protocole de fenêtre glissante ([Ste76]). Une spécification complète est donnée dans [RRSV87a]. Ce protocole supporte des flux unidirectionnels de données sur des lignes qui peuvent perdre, réordonner ou multiplier les messages. La bonne réception d'un message est suivie de l'émission d'un acquittement.

On considère 4 processus dans la description du protocole (voir figure 2.1) :

L'émetteur. L'émetteur prend des messages de la couche supérieure et leur associe un numéro de séquence. Cette action est appelée in_i, où i est le numéro de séquence modulo un entier k.

Le récepteur. Le récepteur accepte des messages en provenance des lignes. Si un message a un numéro de séquence conforme à celui attendu, le message est passé à la couche supérieure. Cette action est dénotée par out_i, i étant le numéro de séquence du message.

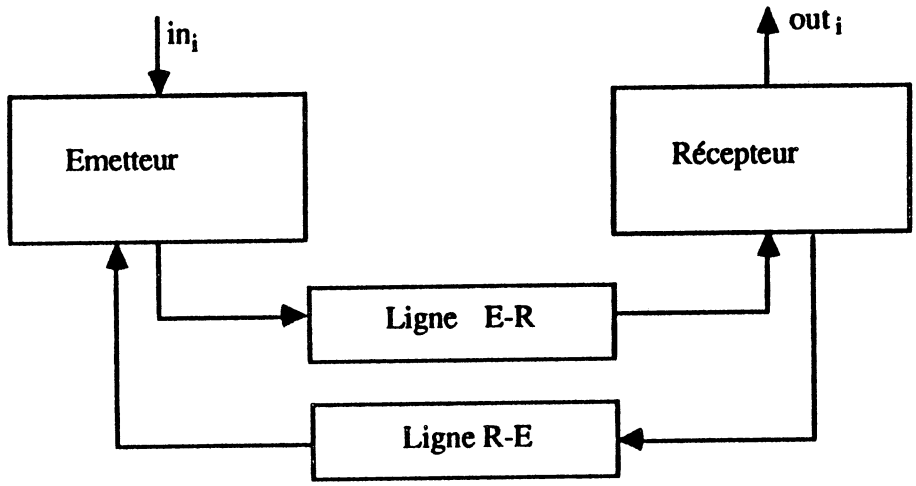

Figure 2.1: Modélisation du protocole de Stenning.

Les lignes. Les lignes sont deux processus qui relient l'émetteur au récepteur et le récepteur à l'émetteur. Elles peuvent perdre, dupliquer ou réordonner les messages.

Les propriétés que l'on décrit sont :

1. Des propriétés exprimant l'alternance des actions in_i et out_i, i entre 0 et $k-1$.

   Ces propriétés sont exprimées par les trois ensembles de formules suivants :

   (a) $after(in_i) \supset \neg\, pot[\neg\, after(out_i)](enable(in_i) \wedge \neg after(out_i))$: si un message étiqueté par i est transmis, il n'est pas possible de transmettre un autre message i à moins qu'un message i n'ait été délivré.

   (b) $after(out_i) \supset \neg\, pot[\neg\, after(in_i)](enable(out_i) \wedge \neg after(in_i))$: si un message étiqueté par i est délivré, il n'est pas possible de délivrer un autre message i à moins qu'un message i n'ait été accepté.

   (c) $init \supset \neg\, pot[\neg\, after(in_i)](enable(out_i \wedge \neg\, after(in_i))$: initialement il n'est pas possible de délivrer un message i avant qu'un message i n'ait été accepté.

2. Les acceptations des messages par l'émetteur et leur passage par le récepteur à la couche supérieure, sont faits en respectant leurs numéros de séquence. Cette propriété est exprimée par quatre ensembles de formules :

(a) $\forall\, 0 \leq n < k, n \neq 1$

$$after(in_i) \supset \neg\, pot[\neg after(in_{(i+1) \bmod k})]$$
$$(enable(in_{(i+n) \bmod k}) \wedge \neg after(in_{(i+1) \bmod k})) :$$

l'acceptation d'un message i ne peut être suivie que du passage du message $(i+1)$ mod k.

(b) $\forall\, 0 \leq n < k, n \neq 1$

$$after(out_i) \supset \neg\, pot[\neg after(out_{(i+1) \bmod k})]$$
$$(enable(out_{(i+n) \bmod k}) \wedge \neg after(out_{(i+1) \bmod k})) :$$

le passage d'un message i à la couche supérieure ne peut être suivi que du passage du message $(i+1)$ mod k.

(c) $\forall\, 0 < n < k$

$$init \supset \neg\, pot[\neg\, after(in_0)](enable(in_n \wedge \neg\, after(in_0)) :$$

initialement il n'est pas possible d'accepter un message avec un numéro de séquence différent de 0.

(d) $\forall\, 0 < n \leq k-1$

$$init \supset \neg\, pot[\neg\, after(out_0)](enable(out_n \wedge \neg\, after(out_0))$$

c'est-à-dire initialement il n'est pas possible de délivrer un message avec un numéro de séquence différent de 0.

2.5 Conclusion.

On a présenté le langage de formules de XESAR et on l'a utilisé pour faire la spécification de service du protocole de Stenning.

La spécification donnée ne comporte que des propriétés de sûreté qui décrivent des relations de causalité entre les exécutions de certains actions. Si l'on ne prend pas d'hypothèse d'équité, la spécification donnée est suffisante pour décrire le service puisque la non-fiabilité des lignes de transmission cause la non-validité des propriétés de vivacité.

Dans [RRSV87a] on a défini la macro-notation **not-to-unless** pour exprimer les propriétes de sûreté montrées précédemment. La formule :

not a **to** b **unless** c

signifie qu'à partir d'un état satisfaisant a il n'est pas possible d'atteindre un état satisfaisant b sans visiter des états qui satisfont c.

Avec cette macro-notation, exprimer le fait que tout passage d'un message i du récepteur à la couche supérieure ne peut être suivi que du passage du message $(i+1)$ mod k, est fait par l'ensemble de formules :

$$\forall\, 0 \leq n \leq k,\ n \neq 1$$

not $after(out_i)$ **to** $enable(out_{(i+n) \bmod k})$ **unless** $after(out_{(i+1) \bmod k})$

Si l'on utilise des éléments du langage logique défini dans ce chapitre, not a **to** b **unless** c **est équivalent à la formule :**

$$a \supset \neg\, pot[\neg c](b \wedge \neg c)$$

L'utilisation de cette macro-notation simplifie la formulation des propriétés de sûreté. Dans le prochain chapitre on propose une nouvelle approche pour exprimer ce genre de propriétés qui contribue à rendre plus facile la conception des spécifications.

Chapitre 3

Les graphes de sûreté.

3.1 Introduction.

Les graphes d'états sont un des formalismes les plus utilisés pour spécifier des systèmes parallèles. Ils sont basés sur des concepts simples, donc faciles à utiliser : les concepts d'état et de transition. De plus, ils ont une représentation graphique, ce qui dans certains cas permet d'exprimer d'une manière claire et compréhensible les propriétés d'un système. Par exemple, considérons la spécification du protocole de Stenning présentée dans la section 2.4.3. Une des propriétés énoncées est (on prend $k = 4$) :

> L'acceptation d'un message i ne peut être suivie que de l'acceptation du message $(i + 1) \bmod 4$.

Si l'on utilise la macro-notation présentée dans le chapitre 2, cette propriété est exprimée par l'ensemble de formules suivant :

1. Formules qui expriment le fait qu'après l'acceptation du message 0 il n'est pas possible d'accepter un message 0 s'il n'y a pas eu d'acceptation précédente des messages 1, 2, et 3 :

   not $after(in_0)$ **to** $enable(in_0)$ **unless** $after(in_1)$

   not $after(in_0)$ **to** $enable(in_0)$ **unless** $after(in_2)$

   not $after(in_0)$ **to** $enable(in_0)$ **unless** $after(in_3)$

2. Formules qui expriment le fait qu'après l'acceptation du message 1 il n'est pas possible d'accepter un message 1 s'il n'y a pas eu d'acceptation précédente des messages 0, 2, et 3 :

not *after*(in_1) **to** *enable*(in_1) **unless** *after*(in_0)

not *after*(in_1) **to** *enable*(in_1) **unless** *after*(in_2)

not *after*(in_1) **to** *enable*(in_1) **unless** *after*(in_3)

3. **Formules qui expriment le fait qu'après l'acceptation du message 2 il n'est pas possible d'accepter un message 2 s'il n'y a pas eu d'acceptation précédente des messages 0, 1, et 3 :**

   not *after*(in_2) **to** *enable*(in_2) **unless** *after*(in_0)

   not *after*(in_2) **to** *enable*(in_2) **unless** *after*(in_1)

   not *after*(in_2) **to** *enable*(in_2) **unless** *after*(in_3)

4. **Formules qui expriment le fait qu'après l'acceptation du message 3 il n'est pas possible d'accepter un message 3 s'il n'y a pas eu d'acceptation précédente des messages 0, 1, et 2 :**

   not *after*(in_3) **to** *enable*(in_3) **unless** *after*(in_0)

   not *after*(in_3) **to** *enable*(in_3) **unless** *after*(in_1)

   not *after*(in_3) **to** *enable*(in_3) **unless** *after*(in_2)

La construction de cet ensemble n'est pas une tâche facile, même pour des utilisateurs ayant une large expérience des logiques.

Avec un graphe d'états cette propriété pourrait être spécifiée par le graphe de la figure 3.1. Dans ce graphe on restreint l'ensemble d'actions à $\{in_0, in_1, in_2, in_3\}$, c'est-à-dire que toutes les autres actions du programme sont des actions *non-visibles*.

On représente ainsi le fonctionnement du système à un certain niveau d'abstraction en décrivant seulement l'effet d'actions significatives pour une propriété. La sémantique donnée à ces graphes diffère de la sémantique usuelle des graphes d'états. Pour expliquer ces différences, considérons la transition $\{e_1 \xrightarrow{in_1} e_2\}$. L'interprétation de ce prédicat dans un graphe ordinaire est la conjonction des deux propriétés suivantes :

P1. Si à partir de e_1 on exécute in_1, on atteint l'état e_2. De plus, on ne peut exécuter que cette action à partir de e_1.

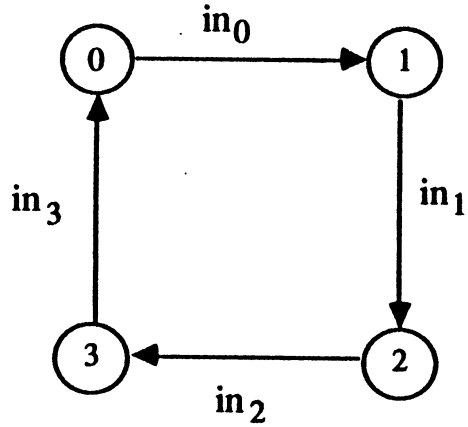

Figure 3.1: Les messages acceptés par l'émetteur respectent l'ordre introduit par leurs numéros de séquence.

P2. L'état e_1 n'est pas un état puits.

La propriété P1 est une propriéte de sûreté, tandis que la conjonction de P1 et P2 donne une propriété de vivacité.

Les abstractions utilisées permettent de considérer chaque état du graphe comme un macro-état contenant des états à partir desquels l'exécution d'une action non-visible mène à un élément dans le même macro-état. Ceci signifie que s'il existe des chemins d'exécution cycliques ne contenant que des actions non-visibles, il est possible de ne jamais sortir d'un macro-état. Dans le cas précis du protocole de Stenning, l'utilisation de lignes non-fiables rend probable l'existence de chemins d'exécution où il existe un nombre infini de pertes de messages. Il est donc possible qu'à partir d'un état dans e_1, il existe des chemins ne contenant pas d'exécution de in_1. Par conséquent, la propriété P2 peut ne pas être satisfaite par des implantations correctes du protocole.

Les graphes que l'on propose pour spécifier des propriétés des systèmes ont une sémantique moins forte : on considère que les transitions sont caractérisées pour la propriété P1 uniquement. Ces graphes sont appelés *graphes de sûreté* puisqu'ils ne spécifient que ce type de propriétés.

Les comparaisons entre graphes d'états.

Si la description d'un système et sa spécification sont données sous la forme de graphes d'états, on peut établir des comparaisons entre ces graphes par des relations d'équivalence. Plusieurs relations d'équivalence peuvent être utilisées pour comparer des graphes : l'équivalence de traces, les modèles de refus, les modèles d'acceptation et l' équivalence par bisimulation.

L'équivalence de traces est définie comme l'égalité des langages de traces d'actions des graphes. Les modèles de refus ([BHR84]) et les modèles d'acceptation ([Gra86]) sont des ensembles de paires (α, B) où α est une séquence finie d'éléments de l'ensemble d'actions A et B est un sous-ensemble fini de A. Si (α, B) est une paire d'un modèle de refus, α est une séquence d'actions exécutable et elle conduit à un état à partir duquel *aucune* action de B n'est exécutable. Si (α, B) est une paire d'un modèle d'acceptation, α est une séquence d'actions exécutable et elle conduit à un état à partir duquel les actions de B sont exécutables. L'équivalence par bisimulation est la plus grande relation $\sim$ telle que, pour toute paire d'états q_1 et q_2 d'un graphe $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$:

$$q_1 \sim q_2 \iff$$

$$q_1 \xrightarrow{a} q_1' \implies \exists q_2', q_2 \xrightarrow{a} q_2' \text{ et } q_1' \sim q_2'$$

$$\text{et } q_2 \xrightarrow{a} q_2' \implies \exists q_1', q_1 \xrightarrow{a} q_1' \text{ et } q_1' \sim q_2'$$

L'équivalence par bisimulation est la plus forte : deux états équivalents sous cette relation le sont aussi sous les autres relations d'équivalence. La relation la moins forte est l'équivalence de traces, c'est-à-dire qu'elle contient toutes les autres relations.

La relation de comparaison choisie définit la sémantique du langage de spécification puisque la signification d'un graphe est la famille de graphes qui lui sont équivalents. Un critère important pour son choix est le type de propriétés que la relation préserve.

Dans ce chapitre on définit l'*équivalence de sûreté* qui caractérise la notion de sûreté dérivée des graphes que l'on propose. Cette équivalence est basée sur la notion d'*implantation sûre* qui nous permet de comparer des graphes et de décider si un graphe de description est conforme aux propriétés exprimées par un *graphe de sûreté*.

3.2 La relation d'implantation sûre.

Considérons deux états q_1 et q_2 d'un graphe G. On dit que q_1 est une implantation sûre de q_2 si toute évolution à partir de q_1 est contenue dans l'ensemble des évolutions de q_2. Si l'on prend comme visibles toutes les actions du graphe, ceci signifie que pour tout état q_1' accessible à partir de q_1 par l'exécution de l'action a, il existe un état q_2' accessible à partir de q_2 par l'exécution de a, tel que q_1' est une implantation sûre de q_2'.

Dans cette section on définit formellement cette relation, notée $\sqsubseteq$, et qui porte sur les états d'un graphe $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$. Un paramètre

important dans cette définition est l'ensemble d'actions A, dont les éléments sont appelés *actions visibles*. Cet ensemble doit être contenu dans E. Les actions de l'ensemble $E - A$ sont appelées *actions non-visibles*. Par la suite, le symbole τ est utilisé pour représenter une action non-visible quelconque.

La définition de la relation d'implantation sûre est basée sur la définition suivante :

Définition 5 *Soit $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ un graphe d'états et A un ensemble d'actions visibles tel que $A \subseteq E$. Les relations $\xrightarrow{\tau^*}$ et $\xrightarrow{\tau^* a}$ sont telles que :*

1. $\xrightarrow{\tau^*} = \bigcup_{i \in N} \xrightarrow{\tau^i}$

$$\xrightarrow{\tau^0} = \{(q, q) \mid q \in Q\}$$

$$\xrightarrow{\tau^{i+1}} = \{(q, q') \mid \exists q'', (q, q'') \in \xrightarrow{\tau^i} \text{ et } q'' \xrightarrow{\tau} q'\}, \qquad i \geq 0$$

2. $\xrightarrow{\tau^* a} = \{(q, q') \mid \exists q'', (q, q'') \in \xrightarrow{\tau^*} \text{ et } q'' \xrightarrow{a} q'\}$

Définition 6 (Implantation sûre) *Soient $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ et $A \subseteq E$ un ensemble d'actions visibles. La relation $\sqsubseteq$ est la plus grande relation $\sqsubseteq \subseteq Q \times Q$ telle que $q_1 \sqsubseteq q_2$ si et seulement si :*

$$\forall q_1' \in Q, \forall a \in A,\ q_1 \xrightarrow{\tau^* a} q_1' \implies \exists q_2',\ q_2 \xrightarrow{\tau^* a} q_2' \text{ et } q_1' \sqsubseteq q_2'$$

et

$$\forall q_1' \in Q,\ q_1 \xrightarrow{\tau^*} q_1' \implies q_1' \sqsubseteq q_2$$

On peut remarquer que si un état n'a pas de successeur, il est une implantation sûre de n'importe quel état. Ceci est également vrai pour tout état à partir duquel il est impossible d'exécuter des séquences d'actions comportant des actions visibles, c'est-à-dire les états q tels que :

$$\forall q',\ q \xrightarrow{\alpha} q' \implies \forall i \in \mathcal{N}, \alpha_i = \tau$$

Exemple 1 *Dans la figure 3.2, si l'ensemble d'actions visibles est $\{a, b\}$, alors les états 1 des graphes de la colonne de gauche sont des implantations sûres des états 8 des graphes placés en colonne droite. Si l'on considère le premier couple de graphes, on observe que :*

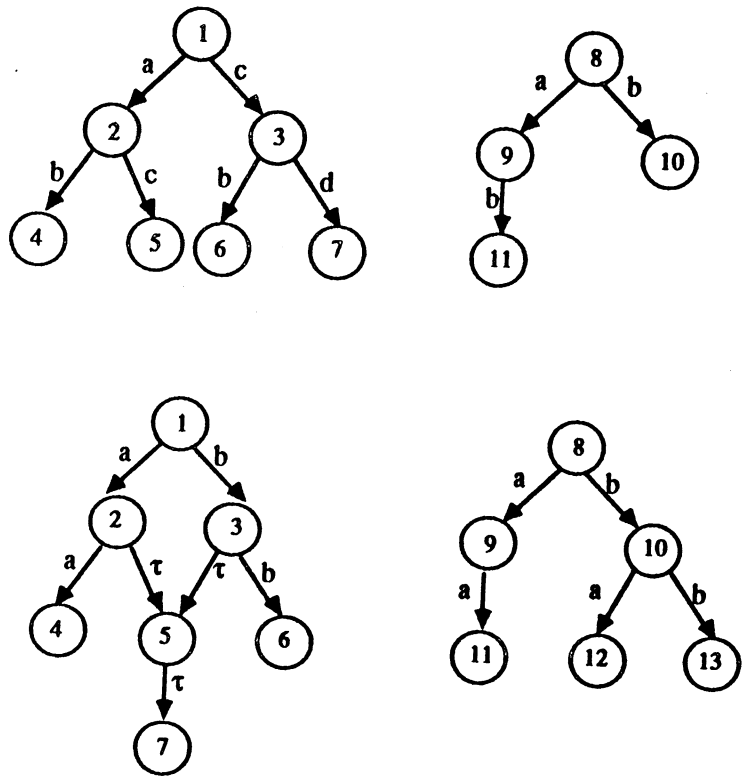

Figure 3.2: Exemples d'implantations sûres.

$1 \sqsubseteq 8$ *parce que* $1 \xrightarrow{a} 2,8 \xrightarrow{a} 9$ *et* $2 \sqsubseteq 9$
$1 \xrightarrow{\tau} 3$ *et* $3 \sqsubseteq 8$

$2 \sqsubseteq 9$ *parce que* $2 \xrightarrow{b} 4,9 \xrightarrow{b} 11$ *et* $4 \sqsubseteq 11$
$2 \xrightarrow{\tau} 5$ *et* $5 \sqsubseteq 9$

$3 \sqsubseteq 8$ *parce que* $3 \xrightarrow{b} 6,8 \xrightarrow{b} 10$ *et* $6 \sqsubseteq 10$
$3 \xrightarrow{\tau} 7$ *et* $7 \sqsubseteq 8$

Trivialement, $4 \sqsubseteq 11, 5 \sqsubseteq 9, 6 \sqsubseteq 10$ *et* $7 \sqsubseteq 10$ *puisque* $4, 5, 6$ *et* 7 *sont des états puits.*

On note $q_1 \approx q_2$ si et seulement si $q_1 \sqsubseteq q_2$ et $q_2 \sqsubseteq q_1$. La relation $\approx$ est appelée *équivalence de sûreté.*

La relation $\sqsubseteq$ peut être calculée comme le plus grand point fixe de la suite $\sqsubseteq^0, \sqsubseteq^1, \ldots$ où :

$$\sqsubseteq^0 = Q \times Q$$

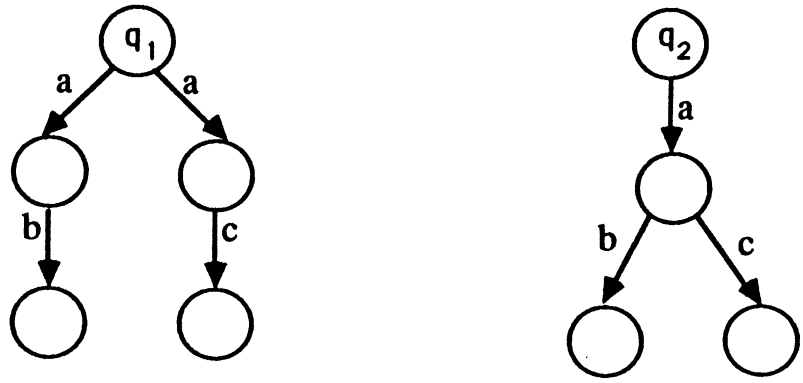

Figure 3.3: Les langages de traces de q_1 et q_2 sont égaux et $q_1 \not\approx q_2$.

$$i > 0,\ q_1 \sqsubseteq^{i+1} q_2 \iff$$

$$\forall q_1' \in Q, \forall a \in A\ ,\ q \xrightarrow{\tau^* a} q' \implies \exists q_2',\ q_2 \xrightarrow{\tau^* a} q_2' \text{ et } q_1' \sqsubseteq^i q_2'$$

et

$$\forall q_1' \in Q, q_1 \xrightarrow{\tau^*} q_1' \implies q_1' \sqsubseteq^i q_2$$

La figure 3.3 montre deux graphes qui ont les mêmes ensembles de traces mais qui ne sont pas équivalents par la relation $\approx$.

La figure 3.4 montre deux graphes équivalents par $\approx$ mais qui ne sont pas équivalents sous les équivalences de refus et d'acceptation. La figure 3.5 montre deux graphes équivalents sous les équivalences de refus et d'acceptation mais qui ne sont pas équivalents sous l'équivalence de sûreté.

Définition 7 *Soit $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ un graphe et $q \in Q$. L'ensemble $L(q)$ (le langage de q) est défini de la manière suivante :*

$$L(q) = \cup_{i \geq 0} L^i(q)$$

où

$$L^0(q) = \{\epsilon\}\ \ \epsilon \textit{ est l'élément neutre de la concaténation de séquences}$$

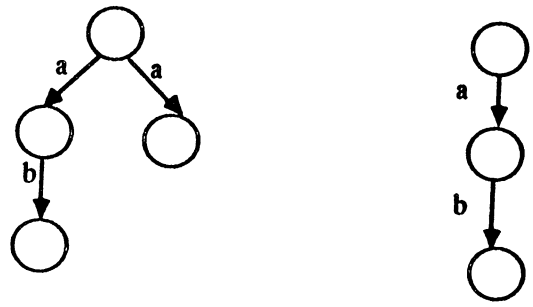

Figure 3.4:

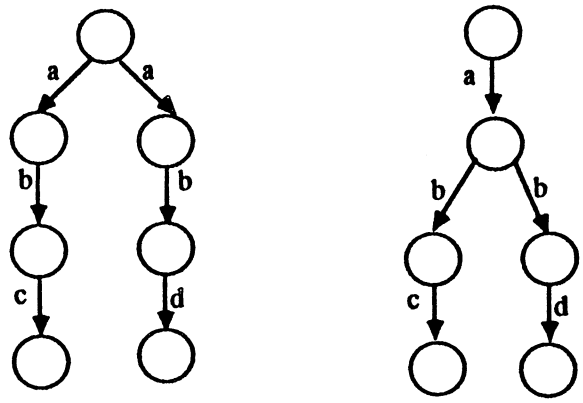

Figure 3.5:

$$L^{i+1}(q) = \{a.\alpha \mid \exists q', q \xrightarrow{\tau^* a} q' \wedge \alpha \in L^i(q')\}, \quad i > 0$$

La proposition 1 montre que la relation d'implantation sûre préserve l'inclusion des langages, c'est-à-dire que la relation $\sqsubseteq$ est contenue dans la relation d'inclusion de traces.

Proposition 1 *Soit* $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ *un graphe d'états,* $A \subseteq E$ *un ensemble d'actions visibles et* q_1, q_2 *des éléments de Q. Alors*

$$q_1 \sqsubseteq q_2 \implies L(q_1) \subseteq L(q_2)$$

Preuve

On prouve que :

$$\forall q_1, q_2,\ q_1 \sqsubseteq q_2 \implies \forall i \geq 0\ L^i(q_1) \subseteq L^i(q_2)$$

- $i = 0$. Evident

- On suppose que :

$$\forall q_1, q_2,\ q_1 \sqsubseteq q_2 \implies k \geq 0\ L^k(q_1) \subseteq L^k(q_2)$$

- Soit $\beta = a_1 \ldots a_{k+1}$ appartenant à $L^{k+1}(q_1)$

$$\implies \exists \beta' = a_2 \ldots a_{k+1}, \exists q_1', q_1 \xrightarrow{\tau^* a_1} q_1' \text{ et } \beta' \in L^k(q_1')$$

$$q_1 \sqsubseteq q_2 \implies \exists q_2', q_2 \xrightarrow{\tau^* a_1} q_2' \text{ et } q_1' \sqsubseteq q_2'$$

$$\implies a_2 \ldots a_{k+1} \in L^k(q_2') \text{ par hypothèse inductive}$$

$$\implies \beta \in L^{k+1}(q_1) \text{ par définition de L(q)}$$

■

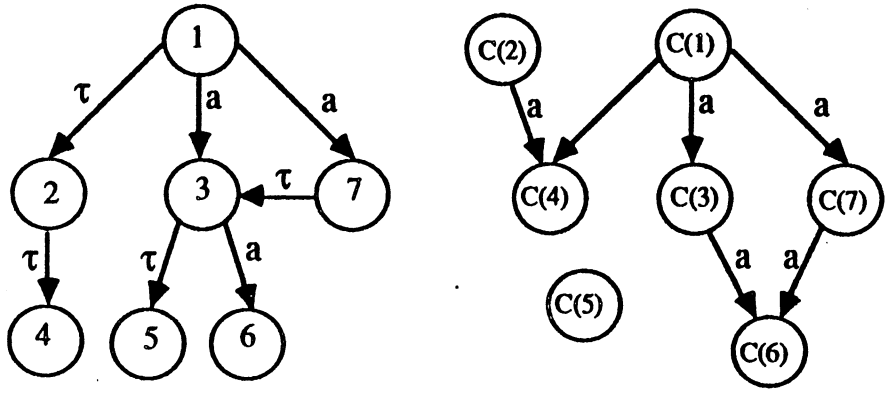

Figure 3.6: Le graphe G et son graphe réduit

3.3 Les réductions de graphes.

Dans cette section on s'intéresse à la définition de transformations sur les graphes qui produisent des graphes ne contenant pas d'action non-visible et préservant la relation d'implantation sûre, c'est-à-dire des graphes équivalents, par la relation $\approx$, aux graphes originaux.

Les états de ces versions réduites sont des ensembles d'états du graphe initial. Les éléments de ces ensembles sont accessibles à partir d'un état q par l'exécution d'actions non-visibles. Ces ensembles sont définis de la manière suivante :

Définition 8 *Soit $G = (Q, E, \{\xrightarrow{a}\}_{a\in E}, i_G)$ un graphe. Pour tout état q de Q, on écrit*

$$c(q) = \{q' \mid q \xrightarrow{\tau^*} q'\}$$

Le graphe réduit $G_c = (Q', A, \{\xrightarrow{a}_c\}_{a\in A})$ est tel que $Q' = \{c(q)\}_{q\in Q}$ et sa relation de transition $\{\xrightarrow{a}_c\}_{a\in A}$ est définie par la règle :

$$\frac{q \xrightarrow{\tau^* a} q'}{c(q) \xrightarrow{a}_c c(q')}$$

Exemple 2 *La figure 3.6 montre le resultat de l'application de la règle de construction du graphe réduit. Dans cet exemple, les états $c(q)$ sont égaux à :*

$$c(1) = \{1, 2\} \quad c(2) = \{2\} \quad c(3) = \{3, 5\}$$

$$c(4) = \{4\} \quad c(5) = \{5\} \quad c(6) = \{6\}$$
$$c(7) = \{3, 5, 7\}$$

Lemme 1 *Si $q' \in c(q)$ alors $c(q') \sqsubseteq c(q)$.*

Preuve

$$c(q') \xrightarrow{a}_c c(q'_1) \implies q' \xrightarrow{\tau^* a} q'_1$$
$$\implies q \xrightarrow{\tau^* a} q'_1 \text{ définition de } c(q)$$
$$\implies c(q) \xrightarrow{a}_c c(q'_1)$$
$$c(q'_1) \sqsubseteq c(q'_1) \implies c(q') \sqsubseteq c(q)$$

■

Proposition 2 *Soit $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ un graphe et $G_c = (\{c(q)\}_{q \in Q}, A, \{\xrightarrow{a}_c\}_{a \in A})$ son graphe réduit. Alors, pour tout état $q \in Q$, $q \approx c(q)$.*

Preuve

1. Preuve de $q \sqsubseteq c(q)$ par induction sur le calcul de $\sqsubseteq$.
   (a) $\forall q \in Q, q \sqsubseteq^0 c(q)$ est trivialement vrai.
   (b) On suppose que pour tout état q, $q \sqsubseteq^k c(q)$
   - $q \xrightarrow{\tau^* a} q' \implies c(q) \xrightarrow{a} c(q')$ par définition. Par hypothèse d'induction on a $q' \sqsubseteq^k c(q')$.
   - $q \xrightarrow{\tau^*} q' \implies c(q') \sqsubseteq c(q)$. Par hypothèse d'induction on a $q' \sqsubseteq^k c(q')$.

   La conjonction des deux points précédents donne $q \sqsubseteq^{k+1} c(q)$.
2. Preuve de $c(q) \sqsubseteq q$ par induction sur le calcul de $\sqsubseteq$.
   (a) $\forall c(q), c(q) \sqsubseteq^0 q$ est trivialement vrai.
   (b) On suppose que pour tout $c(q)$, $c(q) \sqsubseteq^k q$
   $c(q) \xrightarrow{a} c(q') \implies \exists q', q \xrightarrow{\tau^* a} q'$. Par hypothèse d'induction, on a $c(q') \sqsubseteq^k q'$.
   Ceci donne $c(q) \sqsubseteq^{k+1} q$.

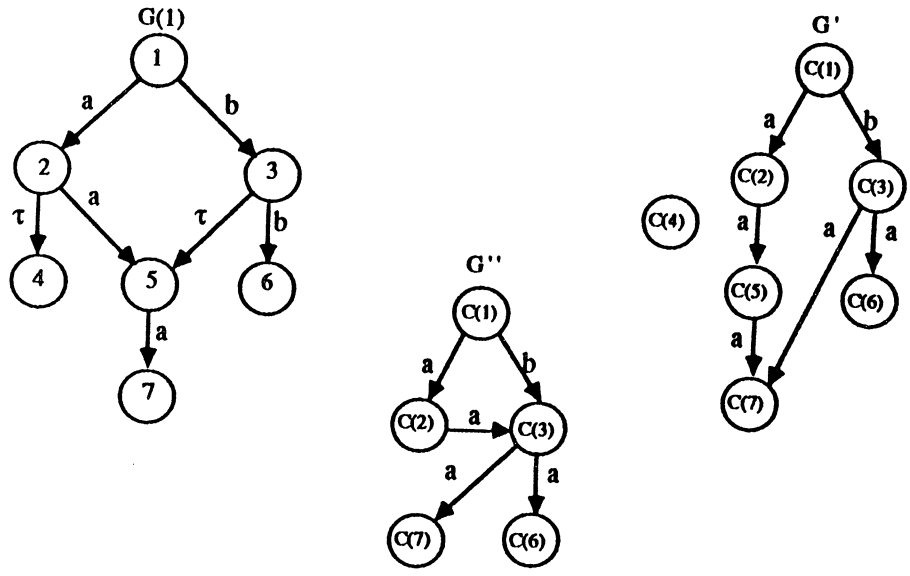

Figure 3.7:

■

Définition 9 *Soit $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ un graphe d'états. On appelle $G(q)$ le sous-graphe contenant tous les successeurs de l'état q.*

Définition 10 *Soient $G(q)$ et $G(q')$ les graphes des successeurs des états q et q'. On écrit $G(q) \sqsubseteq G(q')$ si $q \sqsubseteq q'$.*

Lemme 2 *Soit $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$ un graphe et q_1, q_2, deux de ses états. Si $G(q_1)$ est une implantation sûre de $G(q_2)$, alors, pour tout état q_1' accessible à partir de q_1, il existe q_2' accessible à partir de q_2 tel que $q_1' \sqsubseteq q_2'$.*

Preuve
Directe à partir de la définition de $\sqsubseteq$.

■

La construction du graphe réduit $G_c = (\{c(q)\}_{q \in Q}, A, \{\xrightarrow{a}_c\}_{a \in A})$ peut générer des états c et c' tels que $c \sqsubseteq c'$. L'exemple 3 montre que d'une façon systématique, il n'est pas possible de remplacer un état c du graphe réduit par un état c' qui le majore tout en gardant l'équivalence avec le graphe original.

Exemple 3 *La figure 3.7 montre le graphe réduit G' du graphe $G(1)$. Les états $c(q)$ contiennent les éléments :*

$$c(1) = \{1\} \quad c(2) = \{2, 4\} \quad c(3) = \{3, 5\}$$

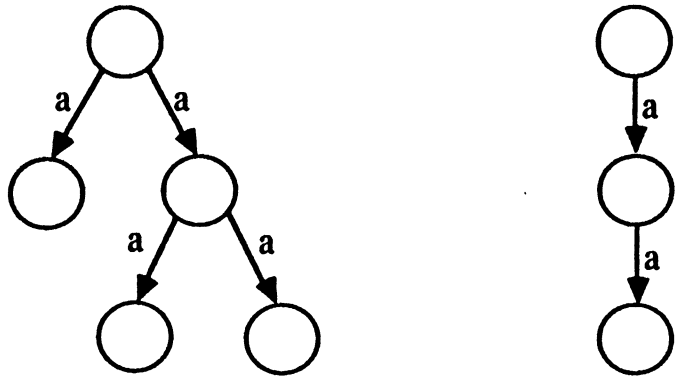

Figure 3.8: Elimination des arcs redondants.

$c(4) = \{4\}$ $c(5) = \{5\}$ $c(6) = \{6\}$
$c(7) = \{7\}$

Si l'on remplace la classe $c(5)$ *par la classe* $c(3)$ *(*5 *appartient à* $c(3)$*), le graphe obtenu* (G'') *n'est pas équivalent à* $G(1)$*.*

Dans la prochaine section, on présente des conditions permettant le remplacement des états du graphe réduit par des états qui les majorent.

3.4 Des réductions optimales de graphes.

La construction du graphe G_c n'est pas optimale au sens où l'application de la règle donnée pour calculer la relation de transition $\{\xrightarrow{a}_c\}_{a\in A}$ génère des états *inaccessibles* et des arcs *redondants*.

Définition 11 *Soit* $G = (Q, E, \{\xrightarrow{a}\}_{a\in E}, i_G)$ *un graphe d'états. Un élément* q *de* Q *est dit* inaccessible *si et seulement si il n'existe pas d'état* q' *tel que* q *est accessible à partir de* q' *par l'exécution d'une transition.*

Définition 12 *Soit* $G = (Q, E, \{\xrightarrow{a}\}_{a\in E}, i_G)$ *un graphe d'états. Un arc* $q \xrightarrow{a} q'$ *est dit* redondant *si et seulement si il existe un arc* $q \xrightarrow{a} q''$ *tel que* $q' \neq q''$ *et* $q' \sqsubseteq q''$*.*

Il est clair que l'élimination des arcs redondants génère des graphes équivalents aux graphes originaux, c'est-à-dire que le graphe $G'(q)$ obtenu à partir de $G(q)$ par élimination de ces arcs est tel que $G'(q) \approx G(q)$.

Exemple 4 *Dans la figure 3.8, le graphe après l'élimination des arcs redondants est équivalent au graphe original.*

Le graphe réduit de la figure 3.6 contient les états $c(2)$ et $c(5)$ qui ne sont pas accessibles à partir de l'état 1 et aussi l'arc $c(1) \xrightarrow{a} c(3)$ qui est redondant puisque $c(1) \xrightarrow{a} c(7)$ et $c(3) \sqsubseteq c(7)$.

L'application de la procédure décrite produit à partir d'un graphe $G(q_0)$ un graphe réduit G'_c ne possédant d'état inaccessible ni d'arc redondant. De plus, ce graphe est équivalent au graphe $G(q_0)$. La procédure calcule les ensembles d'états Q'_c du graphe réduit plus la famille d'ensembles $\{R_a\}_{a \in A}$. A partir de ces ensembles les relations de transition $\{\xrightarrow{a}\}_{a \in A}$ sont construites par la règle :

$$q \xrightarrow{a} q' \Longleftrightarrow (q, q') \in R_a$$

La procédure de génération de Q'_c et des R_a est :

1. $Q'_c \longleftarrow \{c(q_0)\}$
   $\forall\, a \in A, R_a \leftarrow \{\}$
   ATraiter$\leftarrow \{c(q_0)\}$
   /*ATraiter garde les états du graphe réduit dont on n'a pas encore généré les successeurs*/
2. **tantque** ATraiter$\neq \{\}$ **faire**
   (a) Soit $c(q')$ un élément de ATraiter.
   (b) ATraiter$\leftarrow$ ATraiter $-\{c(q')\}$
   (c) $\forall\, a \in A$
      i. succ$\leftarrow \{q_1 \mid q' \xrightarrow{\tau^* a} q_1\}$
      ii. $\forall\, q_1 \in$ succ
         Si $\exists q_2 \in$ succ ,$q_1 \neq q_2$ et $q_1 \sqsubseteq q_2$ alors
         succ$\leftarrow$succ$-\{q_1\}$
      iii. $\forall\, q_1 \in$ *succ*
         A. $R_a \leftarrow R_a \cup \{(c(q'), c(q_1))\}$
         B. ATraiter$\leftarrow$ATraiter $\cup \{c(q_1)\}$
         C. $Q'_c \leftarrow Q'_c \cup \{c(q_1)\}$
3. Fin

Exemple 5 *La figure 3.9 montre le résultat de l'application de la procédure précédente au graphe de la figure 3.6.*

Proposition 3 *Soit $G(q_0)$ et $G_c(c(q_0)) = (Q'_c, A, \{\xrightarrow{a}_{c'}\}_{a \in A})$, le graphe obtenu par application de la procédure décrite ci-dessus. Alors, pour tout état q de Q tel que $c(q) \in Q'_c$, $q \approx c(q)$*

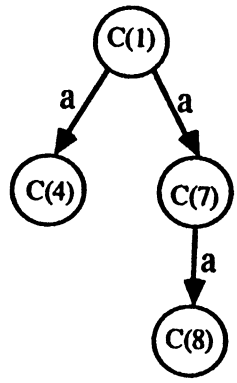

Figure 3.9: Graphe réduit de $G(1)$

Preuve

1. On prouve que $\forall q \in Q$ tel que $c(q) \in Q'_c$, $q \sqsubseteq c(q)$, par induction sur le calcul de $\sqsubseteq$.

   (a) $q \sqsubseteq^0 c(q)$ est trivialement vrai.

   (b) On suppose que $\forall\, q \in Q$ tel que $c(q) \in Q'_c, q \sqsubseteq^k c(q)$.

   Si $q \xrightarrow{\tau^* a} q'$, d'après les règles de construction de G'_c on a un des cas suivants :

   i. $c(q) \xrightarrow{a}_c c(q')$

   ii. $\exists q'', q' \in c(q'')$ et $c(q) \xrightarrow{a}_c c(q'')$

   Si $c(q) \xrightarrow{a}_c c(q')$ alors

   $$q \sqsubseteq^k c(q') \text{ par hypothèse inductive}$$

   Si $c(q) \xrightarrow{a}_c c(q'')$ alors

   $$q \xrightarrow{\tau^* a} q''$$

   et par hypothèse inductive $q'' \sqsubseteq^k c(q'')$

   $$\Longrightarrow q \sqsubseteq^{k+1} c(q)$$

2. On prouve que $c(q) \sqsubseteq q$ par induction sur le calcul de $\sqsubseteq$.

   (a) $c(q) \sqsubseteq^0 q$ est trivialement vrai.

   (b) On suppose que $\forall\, q \in Q, c(q) \sqsubseteq^k q$.

   $$c(q) \xrightarrow{a}_c c(q') \Longrightarrow q \xrightarrow{\tau^* a} q'$$
   $$\Longrightarrow c(q') \sqsubseteq^k q'$$
   $$\Longrightarrow c'(q) \sqsubseteq^{k+1} q$$

3.5 La relation d'implantation sûre et les graphes de sûreté.

Lorsque l'on utilise des graphes d'états pour spécifier des ensembles de propriétés, on suppose que toutes les actions de ces graphes sont visibles. On appelle *graphe de sûreté* tout graphe $G(q_0)$ qui ne contient que des actions visibles.

Soit q_s un état d'un graphe de sûreté $G(q_0)$ tel que :

$act(q_s) = \{a_1, \ldots, a_n\}$ et

$\forall 1 \leq i \leq n, suc(q_s, a_i) = \{q_1^i, \ldots, q_{j_i}^i\}$

On introduit une nouvelle relation $\leq$, telle que $q \leq q_s$ signifie :

- Toute exécution de a_i à partir de q mène à un état q' qui est une implantation sûre d'au moins un des états q_1^i à $q_{j_i}^i$.
- Il n'est pas possible d'exécuter à partir de q des actions appartenant à $A - act(q_s)$.
- Si à partir de q on exécute une action non-visible on arrive à un état implantation sûre de q_s.

On remarque la similitude de la relation $\leq$ avec la relation d'implantation sûre. La différence fondamentale entre les deux relations vient du fait que la relation $\leq$ est établie entre deux états q et q_s en regardant uniquement les états accessibles par l'exécution d'une transition.

Si l'on formalise la description de la relation $\leq$, on obtient la définition suivante :

Définition 13 *Soient q et q_0 des états, $G(q_0) = (Q_{q_0}, A, \{\xrightarrow{a}\}_{a \in A})$ un graphe de sûreté et q_s un état de Q_{q_0}. La relation $\leq$ est définie comme la plus grande relation telle que :*

$q \leq q_s \iff$

$$q \xrightarrow{a} q' \implies \exists q'_s, q_s \xrightarrow{a} q'_s \text{ et } q' \leq q'_s$$

$$\text{et } q \xrightarrow{\tau} q' \implies q' \leq q_s$$

La relation $\leq$ est calculée comme le plus grand point fixe de la suite $\leq^0, \leq^1, \ldots$ où :

$\leq^0 = Q \times Q_s$

$i > 0$, $q \leq^{i+1} q_s \iff$

$\forall a \in A$, $q \xrightarrow{a} q' \implies \exists q'_s$, $q_s \xrightarrow{a} q'_s$ et $q' \leq^i q'_s$

et

$q \xrightarrow{\tau} q' \implies q' \leq^i q_s$

Si l'on compare deux éléments q, q_s tels que q_s appartient à un graphe de sûreté $G(q_0)$, la relation d'implantation sûre peut être exprimée de la façon suivante :

$q \sqsubseteq q_s$ si et seulement si :

$$q \xrightarrow{\tau^* a} q' \implies \exists q'_s \text{ , } q_s \xrightarrow{a} q'_s \text{ et } q' \sqsubseteq q'_s$$

et

$$q \xrightarrow{\tau^*} q' \implies q' \sqsubseteq q_s$$

La proposition 4 montre que si l'on ne considère que des graphes de sûreté à droite du symbole $\sqsubseteq$, alors les relations $\leq$ et $\sqsubseteq$ sont équivalentes.

Proposition 4 *Soient q un état et q_s un état d'un graphe de sûreté. Alors*

$q \leq q_s \iff q \sqsubseteq q_s$

Preuve

- On montre que $\sqsubseteq \implies \leq^i$, pout tout $i \geq 0$.
  - $\forall q, q_s \; q \sqsubseteq q_s \implies q \leq^0 q_s$.
  - On suppose que $\forall q, q_s \; q \sqsubseteq q_s \implies q \leq^k q_s$, $k \geq 0$.
  - $q \sqsubseteq q_s \implies$
    - $\forall a \in A, q \xrightarrow{a} q' \implies \exists q'_s, q_s \xrightarrow{a} q'_s$ et $q' \sqsubseteq q'_s$
      $\implies q' \leq^k q'_s$ par hypothèse inductive
    - $q \xrightarrow{\tau} q' \implies q' \sqsubseteq q_s$
      $\implies q' \leq^k q_s$ par hypothèse inductive

  $\implies q \leq^{k+1} q_s$

- On montre que $\leq \Longrightarrow \sqsubseteq^i$, pout tout $i \geq 0$.
  - $\forall q, q_s \; q \leq q_s \Longrightarrow q \sqsubseteq^0 q_s$.
  - On suppose que $\forall q, q_s \; q \leq q_s \Longrightarrow q \sqsubseteq^k q_s$, $k \geq 0$.
    - $q \xrightarrow{\tau^*} q' \Longrightarrow \exists q^0 \ldots q^m, q^0 = q, q^m = q'$ et $q^j \xrightarrow{\tau} q^{j+1}$
      $q \leq q_s \Longrightarrow \forall j > 0 \; q^j \leq q_s$
      $\Longrightarrow q^1 \sqsubseteq^k q_s$ par hypothèse inductive
    - $q \xrightarrow{\tau^* a} q' \Longrightarrow \exists q'', q \xrightarrow{\tau^*} q''$ et $q'' \xrightarrow{a} q'$
      $q \leq q_s \Longrightarrow q'' \leq q_s$
      $\Longrightarrow \exists q'_s, \; q_s \xrightarrow{a} q'_s$ et $q' \leq q'_s$
      $\Longrightarrow q' \sqsubseteq^k q'_s$ par hypothèse inductive

  $\Longrightarrow q \sqsubseteq^{k+1} q_s$

Il faut remarquer que pour tout graphe de sûreté G ayant A comme ensemble d'actions visibles, le graphe $G_c(q)$ est tel que si $G(q) \sqsubseteq G(q_s)$, alors $G_c(q) \sqsubseteq G(q_s)$, c'est-à-dire que $G_c(q)$ est le plus petit graphe de sûreté qui décrit les propriétés de sûreté relatives à A satisfaites par $G(q)$.

3.6 Caractérisation logique de $\sqsubseteq$.

De la sémantique que l'on a donnée aux graphes de sûreté on peut déduire des propriétés qui doivent être satisfaites par les implantations du système. Dans les sections précédentes, déterminer si ces propriétés étaient satisfaites par un graphe d'implantation, se faisait en calculant la relation $\sqsubseteq$. Cette section contient une autre caractérisation des graphes d'états. On leur associe un ensemble de formules du Mu-calcul défini dans le chapitre 2. Un état q est une implantation sûre d'un état q_s appartenant à un graphe de sûreté $G(q_0)$, si et seulement si $q \models f_s$, où f_s est la formule associée à q_s. $G(q)$ est une implantation sûre de $G(q_0)$ si $q \models f_{q_0}$.

Soit q_s un état d'un graphe de sûreté $G(q_0)$ tel que :

$act(q_s) = \{a_1, \ldots, a_n\}$ et

$\forall 1 \leq i \leq n, suc(q_s, a_i) = \{q^i_1, \ldots, q^i_{j_i}\}$

alors un état q satisfait la formule f_s générée à partir de q_s si et seulement si :

- toute exécution de a_i à partir de q mène à un état qui satisfait la disjontion des formules des états q^i_1 à $q^i_{j_i}$.

- Il n'est pas possible d'exécuter à partir de q des actions appartenant à $A - act(q_s)$.
- Si à partir de q on exécute des actions de $E - A$, on arrive à des états satisfaisant f_s.

Plus formellement :

Définition 14 *Soit $G(q_0)$ un graphe de sûreté et $Q = \{q_0, \ldots, q_n\}$ son ensemble d'états. Soit $X = \{x_0, \ldots, x_n\}$ un ensemble de variables. A chaque état q_i de l'ensemble Q on associe la formule :*

$$f_i(x_0, \ldots, x_n) = \bigwedge_{a \in act(q_i)} [a]\Big(\bigvee_{q_k \in suc(q_i, a)} x_k\Big) \wedge \bigwedge_{a \in A - act(q_i)} [a]\bot \wedge \bigwedge_{a \notin A} [a]x_i$$

Exemple 6 *L'ensemble de formules associé aux états du graphe de la figure 3.1 est :*

$$A = \{in_0, in_1, in_2, in_3\}$$

$$\begin{aligned}
f_0(x_0, x_1, x_2, x_3) &= [in_0]x_1 \wedge [in_1, in_2, in_3]\bot \wedge \bigwedge_{a \notin A} [a]x_0 \\
f_1(x_0, x_1, x_2, x_3) &= [in_1]x_2 \wedge [in_0, in_2, in_3]\bot \wedge \bigwedge_{a \notin A} [a]x_1 \\
f_2(x_0, x_1, x_2, x_3) &= [in_2]x_3 \wedge [in_0, in_1, in_3]\bot \wedge \bigwedge_{a \notin A} [a]x_2 \\
f_3(x_0, x_1, x_2, x_3) &= [in_3]x_0 \wedge [in_0, in_1, in_2]\bot \wedge \bigwedge_{a \notin A} [a]x_3
\end{aligned}$$

Soit $\preceq$ un ordre total sur l'ensemble $\{0, 1, \ldots, n\}$ et $i_0, \ldots, i_n$ l'ordre induit par $\preceq$. Pour évaluer la formule associée à l'état q_{i_n}, on définit la fonction ϕ qui lui fait correspondre une formule du Mu-calcul. $\phi(q_{i_n})$ peut s'exprimer de la manière suivante :

La séquence de formules f_j^k, $k = 1, 2, \ldots, n$, est définie par :

$$\{f_j^k = f_j^{k-1}(\nu x_{i_{k-1}}.f_{i_{k-1}}^{k-1}/x_{i_{k-1}})\}_{i_k \preceq j}$$

avec $\{f_j^0 = f_j\}_{i_0 \preceq j}$ et où $f(g/x)$ représente la formule obtenue à partir de f par substitution de x à g.

On obtient ainsi $\phi(q_{i_n}) = \nu x_{i_n}.f_{i_n}^n$.

Proposition 5 $(\phi(q_i))_{i=0}^n$ *est la plus grande solution du système d'équations :*

$$\{x_i = f_i(x_1, \ldots, x_n)\}_{i=0}^n$$

Exemple 7 *Le système d'équations associé au graphe de la figure 3.1 est :*

$$\begin{aligned}
x_0 &= [in_0]x_1 \wedge [in_1, in_2, in_3]\bot \wedge \bigwedge_{a \notin A} [a]x_0 \\
x_1 &= [in_1]x_2 \wedge [in_0, in_2, in_3]\bot \wedge \bigwedge_{a \notin A} [a]x_1 \\
x_2 &= [in_2]x_3 \wedge [in_0, in_1, in_3]\bot \wedge \bigwedge_{a \notin A} [a]x_2 \\
x_3 &= [in_3]x_0 \wedge [in_0, in_1, in_2]\bot \wedge \bigwedge_{a \notin A} [a]x_3
\end{aligned}$$

La proposition 6 montre que pour déterminer si un graphe est une implantation sûre d'un graphe de sûreté, les deux méthodes présentées (le calcul de la relation $\sqsubseteq$ et l'évaluation du système d'équations déduit du graphe) sont équivalentes.

Proposition 6 $q \sqsubseteq q_s \iff q \models \phi(q_s)$

Preuve
Il suffit de prouver que :

$$\forall k \geq 0\ ,\ q \sqsubseteq^k q_s \iff q \models f_s^k$$

où $f_i^0 = \top$, $\forall\, 0 \leq i \leq n$ et $f_i^{k+1} = f_i(f_0^k, \ldots, f_n^k)$, étant donné que :

$$\phi(q_s) = \bigwedge_{k \in \mathcal{N}} f_s^k \quad \text{et} \quad \sqsubseteq = \bigwedge_{k \in \mathcal{N}} \sqsubseteq^k$$

La preuve est faite par induction sur k.

- Pour $k = 0$ la proposition est vraie.
- On suppose que pour $k \geq 0$:

$$q \sqsubseteq^k q_s \iff q \models f_s^k$$

- On prouve que $q \sqsubseteq^{k+1} q_s \implies q \models f_s^{k+1}$

- $q \sqsubseteq^{k+1} q_s \Longrightarrow$

  $\forall\, a \in A, q \xrightarrow{a} q' \Longrightarrow \exists q_j, q_s \xrightarrow{a} q_j$ **et** $q' \sqsubseteq^k q_j$

  $\Longrightarrow q' \models f_j^k$ **par hypothèse inductive**

  $\forall\, a \in A, q \xrightarrow{a} q' \Longrightarrow \exists q_j, q_s \xrightarrow{a} q_j$ **et** $q' \models f_j^k$

  $\Longrightarrow q \models \bigwedge_{a \in act(q_s)} [a] \bigvee_{q_j \in suc(q_s, a)} f_j^k$

- $q \sqsubseteq^{k+1} q_s \Longrightarrow$

  $q \xrightarrow{\tau} q' \Longrightarrow q' \sqsubseteq^k q_s$

  $\Longrightarrow q' \models f_s^k$ **par hypothèse inductive**

  $\Longrightarrow q \models [\tau] f_s^k$

- $q \sqsubseteq^{k+1} q_s \Longrightarrow \forall a \in A - act(q_s), a \notin act(q)$

  $\Longrightarrow q \models \bigwedge_{a \in A - act(q_s)} [a]\bot$

$\Longrightarrow q \models f_s^{k+1}$

- **On prouve** $q \models f_s^{k+1} \Longrightarrow q \sqsubseteq^{k+1} q_s$

  - $q \models f_s^{k+1} \Longrightarrow q \models \bigwedge_{a \in act(q_s)} [a] \bigvee_{q_j \in suc(a, q_s)} f_j^k \wedge \bigwedge_{a \in A - act(q_s)} \bot$

    $\Longrightarrow (\forall a \in A, q \xrightarrow{a} q' \Longrightarrow \exists q_j, q_s \xrightarrow{a} q_j$ **et** $q' \models f_j^k)$

    $\Longrightarrow (\forall a \in A, q \xrightarrow{a} q' \Longrightarrow \exists q_j, q_s \xrightarrow{a} q_j$ **et** $q' \sqsubseteq^k q_j)$

  - $q \models f_s^{k+1} \Longrightarrow q \models [\tau] f_s^k$

    $\Longrightarrow (\forall\, q', q \xrightarrow{\tau} q' \Longrightarrow q' \models f_s^k$

    $\Longrightarrow q' \sqsubseteq^k q_s$

  $\Longrightarrow q \sqsubseteq^{k+1} q_s$

Exemple 8 *Considérons le premier couple de graphes de la figure 3.2. L'ensemble de formules associé à* $G(8)$ *est :*

$$
\begin{aligned}
f_8 &= [a]x_9 \wedge [b]x_{10} \wedge [\tau]x_8 \\
f_9 &= [b]x_{11} \wedge [a]\bot \wedge [\tau]x_9 \\
f_{10} &= [a,b]\bot \wedge [\tau]x_{10} \\
f_{11} &= [a,b]\bot \wedge [\tau]x_{11}
\end{aligned}
$$

Les formules $\phi(q)$, pour tout état q en $G(8)$ sont :

$$\begin{aligned}
\phi(8) &= \nu x_8.([a]\nu x_9.([b]\nu x_{11}.([a,b]\bot \wedge [\tau]x_{11}) \wedge [a]\bot \wedge [\tau]x_9) \\
&\quad \wedge [b]\nu x_{10}.([a,b]\bot \wedge [\tau]x_{10} \wedge [c]x_8)) \\
\phi(9) &= \nu x_9.([b]\nu x_{11}.([a,b]\bot \wedge [\tau]x_{11}) \wedge [a]\bot \wedge [\tau]x_9) \\
\phi(10) &= \nu x_{10}.([a,b]\bot \wedge [\tau]x_{10}) \\
\phi(11) &= \nu x_{11}.([a,b]\bot \wedge [\tau]x_{11})
\end{aligned}$$

Les ensembles caractéristiques $|\ \phi(q)\ |^{G(1)}$ associés aux formules $\phi(q)$ sont :

$$\begin{aligned}
|\phi(8)|^{G(1)} &= \{1,2,3,4,5,6,7\} \\
|\phi(9)|^{G(1)} &= \{2,3,4,5,6,7\} \\
|\phi(10)|^{G(1)} &= \{4,5,6,7\} \\
|\phi(11)|^{G(1)} &= \{4,5,6,7\}
\end{aligned}$$

3.7 Application des graphes de sûreté.

Considérons des formules de la forme :

not (*after*(a) **to** *enable*(b))**unless** *after*(c)

Ces propriétés expriment le fait qu'après l'exécution de a, il n'est pas possible d'exécuter b à moins que c ait été exécuté. Dans [RRSV87a] on associe à cette macro-notation la formule suivante :

$$after(a) \supset \neg\, pot[\neg\, after(c)](enable(b) \ \wedge \ \neg\, after(c)) \qquad (3.1)$$

Dans cette section on présente une autre caractérisation de la macro-notation **not-to-unless** en utilisant des graphes de sûreté. Basées sur cette nouvelle représentation, on introduit des macro-notations permettant l'expression des propriétés telles que :

- après la dernière exécution de a toute exécution de b doit être précédée d'un minimum de n occurrences de c.
- après la dernière exécution de a toute exécution de b doit être précédée d'un maximum de n occurrences de c.

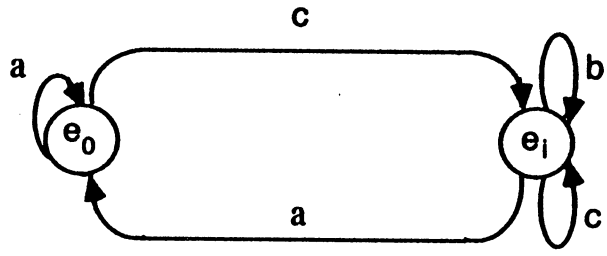

Figure 3.10:

3.7.1 Les graphes associés aux formules not-to-unless.

Pour exprimer la sémantique des formules équivalentes à la formule 3.1, on propose le graphe montré dans la figure 3.10. Il a les caractéristiques suivantes :

- A partir de l'état initial e_i, toutes les actions visibles (l'ensemble $\{a, b, c\}$) sont exécutables. L'exécution des actions b et c ne change pas d'état et l'exécution de a mène à l'état e_0.
- A partir de l'état e_0, les exécutions de a mènent à e_0, les exécutions de c mènent à l'état initial et les exécutions de b sont interdites.

On remarque que les états du modèle qui satisfont le prédicat $after(a)$ sont contenus dans e_0 et qu'à partir de cet état, toute exécution de b est nécessairement précédée par une (ou plus) exécution(s) de c.

Pour montrer l'équivalence de cette représentation graphique et de l'interprétation donnée précédement, considérons d'abord, d'une façon plus détaillée, la formule associée aux macros **not-to-unless** :

Les formules associées à not-to-unless :

Dans la section 2.3, la formule $pot[g]f$ est présentée comme la plus petite solution de l'équation :

$$x = f \vee \ (g \wedge pre(x))$$

En utilisant cette équivalence, la sémantique des formules **not-to-unless** est aussi exprimée comme le plus petit point fixe de l'équation :

$$x = (\neg after(c) \wedge enable(b)) \vee (\neg after(c) \wedge pre(x))$$

Si l'on utilise le fait que $pre(g) \equiv \langle E \rangle g$ et $enable(A) \equiv \langle A \rangle \top$, où E est l'ensemble des actions du modèle et A un de ses sous-ensembles, alors :

$$x = (\neg after(c) \wedge \langle b \rangle \top) \vee (\neg after(c) \wedge \langle E \rangle x)$$

et cette équation est équivalente à :

$$x = \neg after(c) \wedge (\langle b \rangle \top \vee \langle E \rangle x)$$

En complémentant x on obtient :

$$\neg x = after(c) \vee ([b]\bot \wedge [E]\neg x)$$

Donc, la formule

$$after(a) \supset \neg pot[\neg after(c)](enable(b) \wedge \neg after(c))$$

est équivalente à :

$$after(a) \supset y$$

y étant le plus grand point fixe de :

$$y = after(c) \vee ([b]\bot \wedge [E]y) \qquad (3.2)$$

Les formules associées aux graphes de la figure 3.10 :

En utilisant les définitions de la section 3.6, on obtient pour ces graphes les équations suivantes :

$$x_0 = [a, \tau]x_0 \wedge [c]x_i \wedge [b]\bot$$

$$x_i = [a]x_0 \wedge [b, c, \tau]x_i$$

Equivalence des deux interprétations :

Pour prouver l'équivalence de deux interprétations de la macro-notation not-to-unless, on procède de la façon suivante :

- On montre d'abord que le système d'équations associé au graphe de la figure 3.10 est équivalent au système d'équations du graphe de la figure 3.11, plus la condition $after(a) \supset w_0$, w_0 étant le plus grand point fixe de l'équation associée à l'état e'_0.

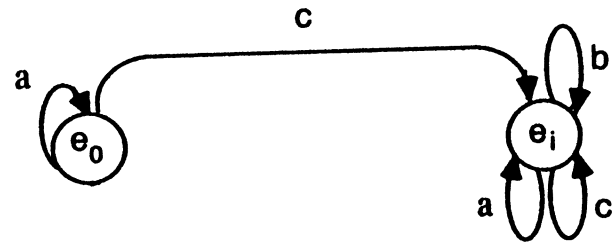

Figure 3.11:

- **Ensuite, on montre l'équivalence :**

$$after(a) \supset w_0 \equiv after(a) \supset y$$

où y est le plus grand point fixe de l'équation 3.2.

La proposition 7 montre que le système d'équations dérivé du graphe de la figure 3.10 est équivalent au système dérivé du graphe de la figure 3.11 avec la condition $after(a) \supset w_0$.

Proposition 7 ***Le système d'équations :***

$$x_0 = [a, \tau]x_0 \wedge [c]x_i \wedge [b]\bot$$

$$x_i = [a]x_0 \wedge [b, c, \tau]x_i$$

est équivalent au système :

$$w_0 = [a, \tau]w_0 \wedge [c]w_1 \wedge [b]\bot$$

$$w_1 = [E]w_1$$

plus la condition $after(a) \supset w_0$

Preuve

Etant donné que les plus grands point fixes de x_0, x_i, w_0 et w_1 peuvent être calculés comme les points fixes des séquences x_0^j, x_i^j, w_0^j et w_1^j, où :

$$x_0^0 = \top \quad x_i^0 = \top \quad w_0^0 = \top \quad w_i^0 = \top$$

et pour j supérieur ou égal 0 :

$$\begin{aligned} x_0^{j+1} &= [a,\tau]x_0^j \wedge [c]x_i^j \wedge [b]\bot \\ x_i^{j+1} &= [a]x_0^j \wedge [b,c,\tau]x_i^j \\ w_0^{j+1} &= [a,\tau]w_0^j \wedge [c]w_1^j \wedge [b]\bot \\ w_1^{j+1} &= [E]w_1^j \end{aligned}$$

il est suffisant de montrer que $\forall\, j \geq 0$

$$\begin{aligned} x_0^j &\equiv w_0^j \\ x_i^j &\equiv w_1^j \end{aligned}$$

- $j = 0$, évident.
- On suppose que :

$$\begin{aligned} x_0^k &\equiv w_0^k \\ x_i^k &\equiv w_1^k \end{aligned}$$

-

$$\begin{aligned} w_0^{k+1} &= [a,\tau]w_0^k \wedge [c]w_1^k \wedge [b]\bot \\ &= [a,\tau]x_0^k \wedge [c]x_i^k \wedge [b]\bot \\ &= x_0^{k+1} \end{aligned}$$

$$\begin{aligned} w_1^{k+1} &= [a]w_1^k \wedge [b,c,\tau]w_1^k \\ &= [a]w_0^k \wedge [b,c,\tau]w_1^k \text{ parce que } \mathit{after}(a) \supset w_0 \\ &= [a]x_0^k \wedge [b,c,\tau]x_i^k \\ &= x_i^{k+1} \end{aligned}$$

■

Il est facile de voir que le plus petit point fixe de l'équation $w_1 = [E]w_1$ est $\top$, et en conséquence, le système correspondant au graphe de la figure 3.11 se réduit à l'équation :

$$w_0 = [a,\tau]w_0 \wedge [c]\top \wedge [b]\bot \tag{3.3}$$

La preuve de :

$$after(a) \supset w_0 \equiv after(a) \supset y$$

où w_0 et y dénotent les points fixes des équations 3.3 et 3.2 respectivement, est basée sur les résultats suivants :

1. Soit $B \subseteq E$ et $a \in B$. Alors

$$[B](after(a) \vee f) \equiv [a]after(a) \wedge [B - \{a\}]f$$

Preuve
$(\Longrightarrow)$
$q \models [B](after(a) \vee f) \Longrightarrow$
$(q \xrightarrow{a} q' \Longrightarrow q' \models after(a)$ **et**
$q \xrightarrow{a'} q', a' \in B, a' \neq a \Longrightarrow q' \models f)$
$\Longrightarrow q \models [a]after(a) \wedge [B - \{a\}]f$
$(\Longleftarrow)$
$q \models [a]after(a) \wedge [B - \{a\}]f \Longrightarrow$
$(q \xrightarrow{a} q' \Longrightarrow q' \models after(a)$ **et**
$q \xrightarrow{a'} q', a' \in B, a' \neq a \Longrightarrow q' \models f)$
$\Longrightarrow (q \xrightarrow{a''} q', a'' \in B \Longrightarrow q' \models after(a) \vee f)$
$\Longrightarrow q \models [B](after(a) \vee f)$

2. $[a, c, \tau]y \equiv [c]\top \wedge [a, \tau]y$

Preuve

$$\begin{aligned}
[a, c, \tau]y &\equiv [a, c, \tau](after(c) \vee [b]\bot \wedge [E]y) \text{ définition de } y \\
&\equiv [c]after(c) \wedge [a, \tau]([b]\bot \wedge [a, c, \tau]y) \\
&\equiv [c]\top \wedge [a, \tau]([b]\bot \wedge [a, c, \tau]y) \\
&\equiv [c]\top \wedge [a, \tau](after(c) \vee [b]\bot \wedge [E]y) \\
&\equiv [c]\top \wedge [a, \tau]y
\end{aligned}$$

3. $\forall i \geq 0, [a, c, \tau]y^i \equiv [a, \tau]w_0^i \wedge [c]\top$

Preuve

$i = 0$, évident

On suppose :

$$[a, c, \tau]y^k \equiv [a, \tau]w_0^k \wedge [c]\top$$

On prouve :

$$[a,c,\tau]y^{k+1} \equiv [a,\tau]w_0^{k+1} \wedge [c]\top$$

$$\begin{aligned}
[a,c,\tau]y^{k+1} &\equiv [c]\top \wedge [a,\tau]y^{k+1} \\
&\equiv [c]\top \wedge [a,\tau](after(c) \vee [b]\bot \wedge [E]y^k) \\
&\equiv [c]\top \wedge [a,\tau]([b]\bot \wedge [a,c,\tau]y^k) \\
&\equiv [c]\top \wedge [a,\tau]([b]\bot \wedge [a,\tau]w_0^k \wedge [c]\top) \\
&\equiv [c]\top \wedge [a,\tau]w_0^{k+1}
\end{aligned}$$

Proposition 8
$after(a) \supset y \equiv after(a) \supset w_0$

Preuve

$$\begin{aligned}
after(a) \supset y &\equiv after(a) \supset after(c) \vee [b]\bot \wedge [E]y \\
&\equiv after(a) \supset [b]\bot \wedge [a,\tau]z \wedge [c]\top \\
&\equiv after(a) \supset z
\end{aligned}$$

■

3.7.2 Le comptage d'occurrences d'actions.

Les macro-notations $after(a) \xrightarrow{c \geq n} enable(b)$.

Une généralisation naturelle des propriétés exprimées par les macros not-to-unless est l'expression de propriétés où l'exécution de l'action b après l'exécution de l'action a, n'est admise que si l'action c a éte exécutée au moins n fois. Dans le graphe de la figure 3.10, correspondant aux formules not-to-unless, n égal à 1. Dans le cas général, les graphes exprimant propriétés suivent le schéma montré dans la figure 3.12. Dans ce graphe, on peut donner aux états l'interprétation suivante :

- L'état e_i représente l'état initial du système. A partir de cet état, toutes les actions sont exécutables et seulement l'exécution de a provoque un changement d'état.
- Les états e_j $(0 \leq j \leq n)$ contiennent des états du modèle atteints à partir de la dernière exécution de a par des chemins comportant, hormis l'exécution d'actions non-visibles, j exécutions de c. En particulier, e_0 contient les états satisfaisant $after(a)$.

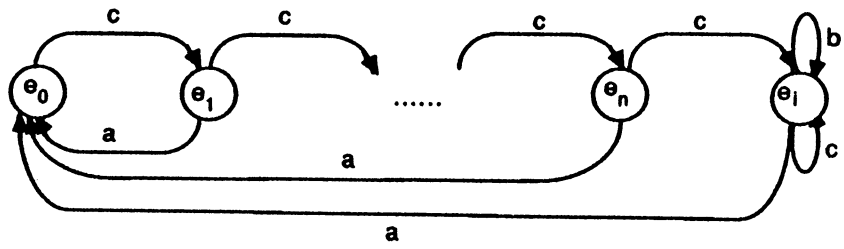

Figure 3.12:

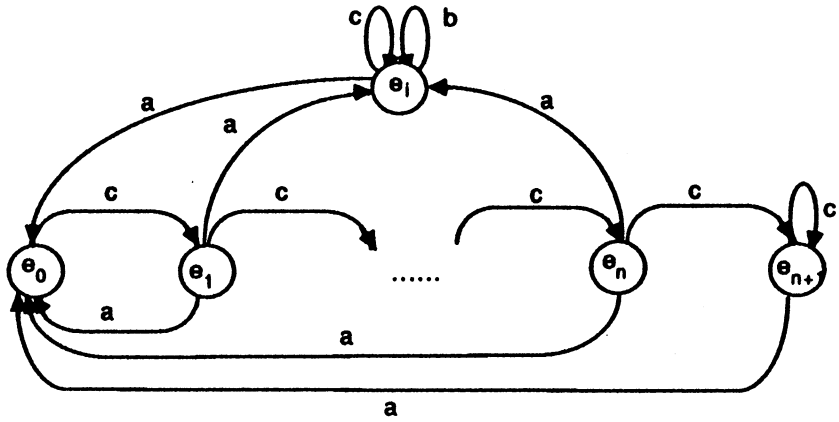

Figure 3.13:

Pour exprimer les propriétés dérivées de ces graphes, on propose la macro-notation :

$after(a) \xrightarrow{c>n} enable(b)$

Dans la section 3.8.1 on montre des exemples d'utilisation de ces formules.

Les macro-notations $after(a) \xrightarrow{c \leq n} enable(b)$.

Avec ces macro-notations on exprime des propriétés telles que :

> après la dernière exécution de a toute exécution de b doit être précédée d'un maximum de n occurrences de c.

Si l'on utilise des graphes de sûreté, on associe à ces macros des graphes suivant le schéma de la figure 3.13. Dans ces graphes on distingue :

- L'état initial e_i, à partir duquel toutes les actions sont exécutables et uniquement l'exécution de a provoque un changement d'état.
- Les e_j $(0 \leq j \leq n+1)$, contenant des états du modèle atteints à partir de la dernière exécution de a par des chemins comportant j exécutions de c. Dans les e_j toute exécution de a mène à e_0. Pour $j \leq n$, toute exécution de b mène à l'état initial. A partir de e_{n+1}, les exécutions de b sont interdites.

Dans la section 3.8.1 on montre des exemples d'utilisation de ces formules.

3.8 Exemples de spécification.

3.8.1 Le protocole de Stenning.

Dans la section 2.4.3 on a donné une description succincte du protocole de Stenning. Elle ne comportait que des propriétés relatives à la spécification de services du protocole, c'est-à-dire la manière dont le protocole doit se comporter par rapport aux couches supérieures.

Cette section contient la description de ces propriétés en utilisant des graphes de sûreté, et la description de quelques propriétés dites *internes* : des propriétés qui caractérisent le comportement des programmes décrivant le protocole. Elles servent à vérifier si le programme est une implantation correcte du protocole de fenêtre glissante.

Avant de décrire ces propriétés, on complète la description intuitive du protocole :

L'émetteur : L'émetteur possède une fenêtre dans laquelle sont contenus les messages transmis qui n'ont pas encore été acquittés. Ils sont placés à partir de la case *lowest_unacked* jusqu'à la case

$$(lowest\_unacked + number\_in\_transit - 1) \bmod k$$

incluse. L'entier *lowest_unacked* représente le numéro de séquence le plus bas des messages dans la fenêtre et *number_in_transit* leur nombre. La taille de la fenêtre de l'émetteur est déterminée par la constante TWS, inférieure à k.

Après la transmission d'un message (action tm_i), un compteur de temps est déclenché pour ce message. S'il atteint TOE unités, les compteurs de tous les messages placés dans la fenêtre sont arrêtés et ceux-ci sont tous retransmis.

Si un acquittement est reçu (actions ra_i) et son numéro de séquence i est dans la fenêtre, alors on donne à *lowest_unacked* la valeur $(i+1) \bmod k$ et les compteurs de temps des autres messages dans la fenêtre sont mis à 0.

Le récepteur : Le récepteur maintient une variable entière, appelée *next_required*, qui indique le numéro de séquence (modulo k) du prochain message attendu, c'est-à-dire le numéro de séquence qui suit le dernier message acquitté. Il maintient aussi une fenêtre de réception dont la taille RWS est inférieure à k.

Lorsqu'il reçoit un message (action re_i) ayant un numéro de séquence i égal à *next_required*, il passe à la couche supérieure tous les messages jusqu'au premier message dans la fenêtre qui n'a pas encore été reçu, et il envoie un acquittement du dernier message livré (action ta_m, m numéro de séquence du dernier message livré). Si le message reçu a un numéro de séquence différent de *next_required* et est dans la fenêtre, il est gardé et un acquittement avec un numéro de séquence égal à $(next\_required - 1) \bmod k$ est envoyé.

Les lignes : Aux lignes sont associés des délais DTM et dtm qui représentent, respectivement, les temps maximal et minimal nécessaires pour transmettre un message ou un acquittement.

Spécifications du service :

1. Pour chaque message accepté par l'émetteur auquel on donne le numéro de séquence i (action in_i), il a y eu un message avec numéro de séquence i délivré par le récepteur (action out_i), et inversement, pour chaque message numéroté par i délivré par le récepteur, il existe un message i accepté par l'émetteur. De plus, la première acceptation d'un message i est faite avant le premier passage d'un message i à la couche supérieure.

   Pour k égal à 4, cette propriété est exprimée par les graphes de la figure 3.14.

2. L'acceptation des messages et leur passage à la couche supérieure sont faits en respectant leur numéro de séquence.

   Pour k égal à 4, cette propriété est exprimée par les graphes de la figure 3.15.

Spécifications internes :

Dans les spécifications internes on utilise les prédicats suivants :

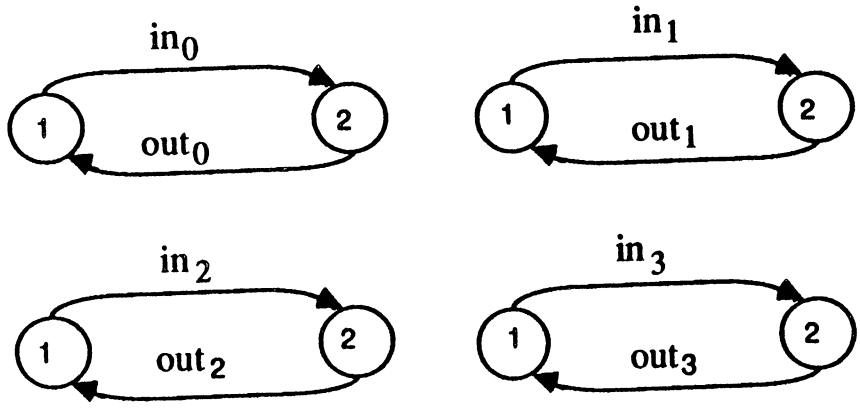

Figure 3.14:

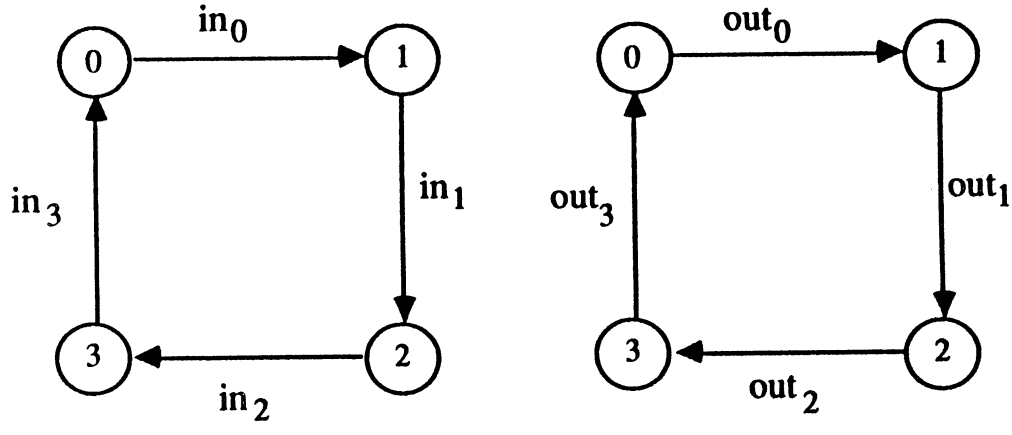

Figure 3.15:

- **in_window**$(i, b, w, k) =$
  $((b + w + \leq k) \implies (b \leq i)$ **et** $(i < b + w))$ **et** $(not\,(b + w) \leq k) \implies$
  $(b \leq i)$ **ou** $(i < b + w - k))$
  Ce prédicat est vrai si i **appartient à l'intervalle cyclique modulo** k **qui commence par** b **et qui a** w **comme largeur.**

- **in_transit**(i)

  Ce prédicat est vrai si le message i **a été transmis et n'a pas encore été acquitté.**

La spécification interne contient les formules suivantes :

1. $al($**in_transit**$(i) \implies$
   in_window$(i, lowest\_unacked, number\_in\_transit, k))$:
   il est toujours vrai que les numéros de séquence des messages transmis et qui n'ont pas encore été acquittés sont dans la fenêtre.

2. $al(enable(mess_i) \implies$
   in_window$(i, lowest\_unacked, number\_in\_transit, k))$:
   il est toujours vrai que si l'on est capable d'accepter un message i**, alors** i **est dans la fenêtre.**

3. $al(0 \leq number\_in\_transit \leq TWS)$:
   il est toujours vrai que le nombre de messages envoyés dont les acquittements n'on pas été reçus, est borné par TWS.

4. $after(tm_i) \xrightarrow{\chi \leq DTM} enable(re_i)$:
   si après la transmission d'un message i **on est capable de recevoir un message** i**, alors le nombre d'exécutions de l'action** χ **est inférieur à** DTM. **En** XESAR χ **est le nom de l'action qui dénote la progression du temps.**

5. $after(tm_i) \xrightarrow{\chi \geq dtm} enable(re_i)$:
   si après la transmission d'un message i**, on est capable de recevoir un message** i**, alors le nombre d'unités de temps écoulées entre les occurrences des deux actions est supérieur à** dtm.

Le chapitre 7 montre l'utilisation de XESAR **pour vérifier si la description du protocole de Stenning contenue dans l'annexe B, vérifie la spécification donnée dans cette section.**

3.8.2 Un protocole de transport.

Dans cette partie on donne la spécification d'un protocole de communication correspondant à la couche 4 du schéma de normalisation ISO ([Zim80]),

appelée couche de transport. Le but de cette couche est d'offrir une abstraction du réseau de communication aux couches supérieures. Pour réaliser ces fonctions, la couche 4 fait appel à un ensemble de fonctions offertes par la couche inférieure (réseau).

Le protocole qui nous intéresse a deux interlocuteurs communiquant à travers un réseau :

- L'*initiateur*, qui décide d'établir la communication.
- L'*accepteur*, qui est appelé et accepte la communication.

A partir d'un état où il n'y a pas de connexion établie, l'initiateur appelle l'accepteur en lui envoyant une *demande de connexion* (ConReq). Le réseau transmet à l'accepteur une *indication de connexion* (ConInd) et ce dernier émet une *réponse de connexion* (ConResp). Le réseau tramsmet à l'initiateur une *confirmation de connexion* (ConConf) et la communication est établie. Ensuite, l'initiateur peut commencer à transmettre des messages vers l'accepteur. Pour ce faire, il envoie à travers le réseau une *demande de données* (DataReq) et le réseau transmet à l'accepteur une *indication de données* (DataInd).

A tout moment la communication peut être interrompue par l'initiateur. Dans ce cas, il émet une *demande de déconnexion* (DiscReq). Le réseau transmet à l'accepteur une *indication de déconnexion* (DiscInd) et le système revient à l'état initial lorsque l'initiateur reçoit du réseau une *confirmation de déconnexion* (DiscConf).

Propriétes de sûreté.

Le graphe de la figure 3.16 spécifie des propriétés de sûreté qui doivent être vérifiées par toute description du protocole de transport.

Il nous permet de spécifier des propriétés de sûreté telles que :

- Toute réponse de connexion est précédée d'une demande de connexion.
- Toute confirmation de connexion est précédée d'une réponse de connexion.
- Toute demande de données est précédée d'une confirmation de connexion et entre les occurrences des deux actions il n'y a pas de demande de déconnexion.
- Toute confirmation de déconnexion est précédée d'une indication de déconnexion.

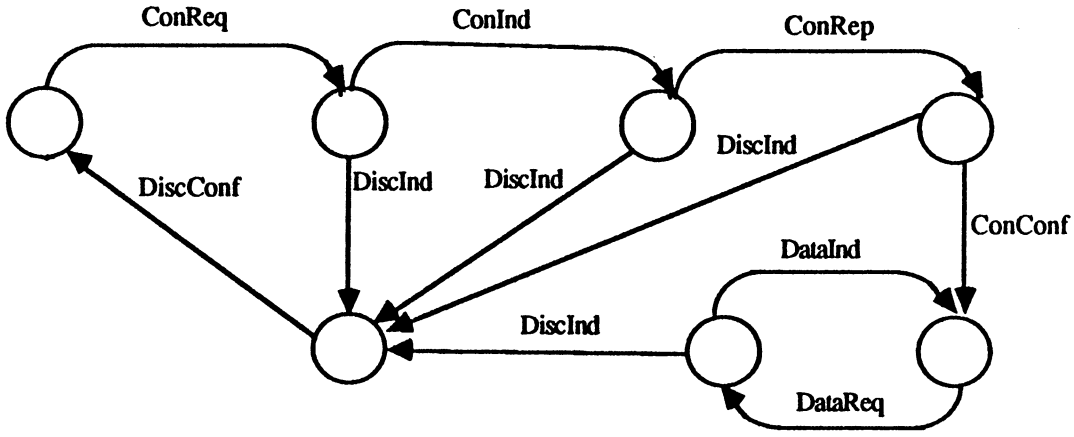

Figure 3.16:

3.9 Conclusion.

On a présenté la relation d'implantation sûre qui nous permet de comparer deux graphes d'états et de décider si l'un, appelé graphe d'implantation, est conforme aux propriétés dérivées de l'autre, appelé graphe de sûreté.

Il faut remarquer que les notions de graphes d'implantation et de graphes de sûreté sont relatives : si $G(q) \sqsubseteq G(s)$ et $G(s) \sqsubseteq G(s')$, alors le graphe $G(s)$, implantation du graphe $G(s')$ peut aussi être pris comme une spécification du graphe $G(q)$. Ces caractéristiques de la relation $\sqsubseteq$ la rendent très utile pour la vérification de systèmes développés par abstractions successives. Dans ce cas, on peut utiliser les descriptions les plus abstraites comme des spécifications des versions plus raffinées.

L'approche présentée pour spécifier et valider des propriétés en utilisant des graphes, est différente des techniques de spécification et vérification des systèmes proposées en [PM87] et [AF85]. Ces articles préconisent la transformation des formules de logiques temporelles linéaires en automates finis. Ensuite, vérifier si un programme satisfait les propriétés exprimées par les formules consiste à déterminer si l'ensemble de séquences d'exécution qu'il génère est *accepté* par l'automate. Une autre différence par rapport à notre méthode, est que chaque état d'un automate dérivé d'une formule caractérise des ensembles d'états d'un programme en utilisant des prédicats sur les valeurs de variables. Avec l'approche proposée dans ce chapitre, on confond des états dans une classe en regardant les langages d'actions visibles générés à partir de ces états. Il est évident que cette dernière approche facilite la définition de plusieurs niveaux d'abstraction des spécifications. De

plus, notre méthode peut être enrichie pour prendre en compte des prédicats sur les valeurs des variables. Pour ce faire, on associe à chaque état q_s du graphe de sûreté un prédicat p_{q_s}. La formule qui correspond à cet état est :

$$f_{q_s}(x_0, \ldots, x_n) = p_{q_s} \wedge$$

$$\left(\bigwedge_{a \in act(q_i)} [a] \left(\bigvee_{q_k \in suc(q_i, a)} x_k \right) \wedge \bigwedge_{a \in act(q_i)} [a] \bot \wedge \bigwedge_{a \notin A} [a] x_i \right)$$

Dans les systèmes présentés précédemment, le prédicat p_{q_s} était égal à $\top$, pour tout état du graphe de sûreté.

Dans la spécification de services du protocole de Stenning, on peut remarquer que les actions apparaissant dans les graphes de sûreté forment l'ensemble

$$\{in_0, in_1, in_2, in_3, out_0, out_1, out_2, out_3\}$$

Si l'on applique la procédure de réduction de la section 3.4, on obtient un graphe d'implantation ne comportant que des actions de cet ensemble et qui préserve la relation de implantation sûre. On peut donc évaluer les systèmes d'équations des graphes de sûreté sur ce graphe réduit. Donc, une méthode à suivre pour évaluer les systèmes d'équations d'un ensemble de graphes de sûreté est :

1. Réduire le graphe d'implantation modulo l'ensemble des actions contenues dans les graphes de sûreté de la spécification.
2. Evaluer les systèmes d'équations sur le graphe réduit.

Partie II

Réalisation du vérificateur de Xesar.

Chapitre 4

Présentation générale de Xesar.

XESAR est un outil de validation qui suit l'approche des deux langages. Le langage de description choisi est ESTELLE/R, un dérivé du langage ESTELLE. Son langage de spécification est le langage de formules défini dans le chapitre 2.

Dans la première partie de ce chapitre on fait une présentation sommaire du langage de description en montrant les principes généraux de ESTELLE/R. La deuxième partie du chapitre donne des détails sur la traduction des programmes de description vers des graphes d'états, modèles du langage de spécification.

4.1 Le langage ESTELLE/R.

ESTELLE/R est une variante du langage ESTELLE ([Est85]). ESTELLE a été conçu pour la spécification des protocoles et de leurs services sous la forme d'un système de composantes séquentielles, structurées de manière hiérarchique. Les composantes du système communiquent via des liens bidirectionnels établis entre des ports, appelés *points d'interaction*. En ESTELLE, à chaque point d'interaction est associée une file où sont placés les messages envoyés à ce point.

ESTELLE/R garde les mêmes principes pour l'organisation du programme de description : il est décrit comme un système d'automates étendus, c'est-à-dire des automates tels que des actions, décrites par des blocs d'instructions, sont associées aux transitions. La différence principale avec ESTELLE est l'utilisation du *rendez-vous* pour réaliser les communications.

name *nom_du_processus*

parameter *liste_de_paramètres*

interactions *liste_de_noms_point_interaction*

variables *liste_de_variables_locales*

states *liste_de_noms_états*

init *transition_initiale*

transitions *transitions*

Figure 4.1: Grammaire abstraite d'un processus.

La figure 4.1 montre une syntaxe abstraite d'une composante du système définie en ESTELLE/R. On distingue les éléments suivants :

- Son nom.
- Une liste de paramètres passés au processus au moment de sa création.
- La liste des points d'interaction du processus. A travers chaque point, le processus peut envoyer (ou recevoir) des ensembles de valeurs. Chaque ensemble est appelé *signal*.
- Une liste de déclarations de variables locales au processus.
- Une liste de noms d'états.
- Une transition d'initialisation qui désigne l'état initial parmi les éléments de la liste d'états. Cette transition est de la forme :

  to *nom_état bloc_instructions_init*

  bloc_instructions_init est un bloc d'instructions PASCAL contenant des affectations aux variables locales qui fixent leur valeur initiale.

- La liste de transitions ; chaque transition suit le schéma suivant :

  from *étiquette état_1* **to** *état_2 cond bloc_instructions*

étiquette **est un nom associé à la transition référençable depuis le langage de spécification.**

cond **conditionne l'exécution de la transition. Elle peut être composée des éléments suivants :**

- **Des conditions sur les valeurs des variables locales. Elle sont exprimées par la construction :**

  provided *exp_bool*

- **La réception d'un message, exprimée par la construction :**

  when *nom_point_interaction liste_variables*

- **Une condition temporelle, exprimée par :**

  delay(m, n)

  Intuitivement, ce type de condition indique que l'exécution de cette transition doit être retardée d'au minimun m unités de temps et qu'elle doit être exécutée avant n unités de temps.

La partie *bloc_instructions* **du corps d'une transition est un bloc d'instructions** PASCAL **qui peut être enrichi des constructions :**

output *nom_point_interaction liste_expressions*
nextstate *nom_état*

L'instruction output émet la liste de valeurs spécifiée à travers le point d'interaction. L'exécution de l'instruction nextstate mène le processus à l'état indiqué.

4.2 Organisation de XESAR.

La vérification des programmes en XESAR **est réalisée en cinq phases. Les quatre premières effectuent la transformation du programme** ESTELLE/R **d'entrée vers un graphe d'états. La cinquième prend les spécifications données par l'utilisateur et les évalue sur le graphe.**

L'architecture générale de XESAR **est montrée dans la figure 4.2. Les fonctions de chaque phase sont :**

Compilation.

XESAR **a été conçu pour être intégré dans un environnement de programmation contenant un compilateur du langage** ESTELLE/R**. Pour cette raison, en** XESAR**, les programmes d'entrée sont supposés syntactiquement corrects. Cette phase réalise un filtrage syntaxique ayant pour but la génération d'une représentation interne du programme.**

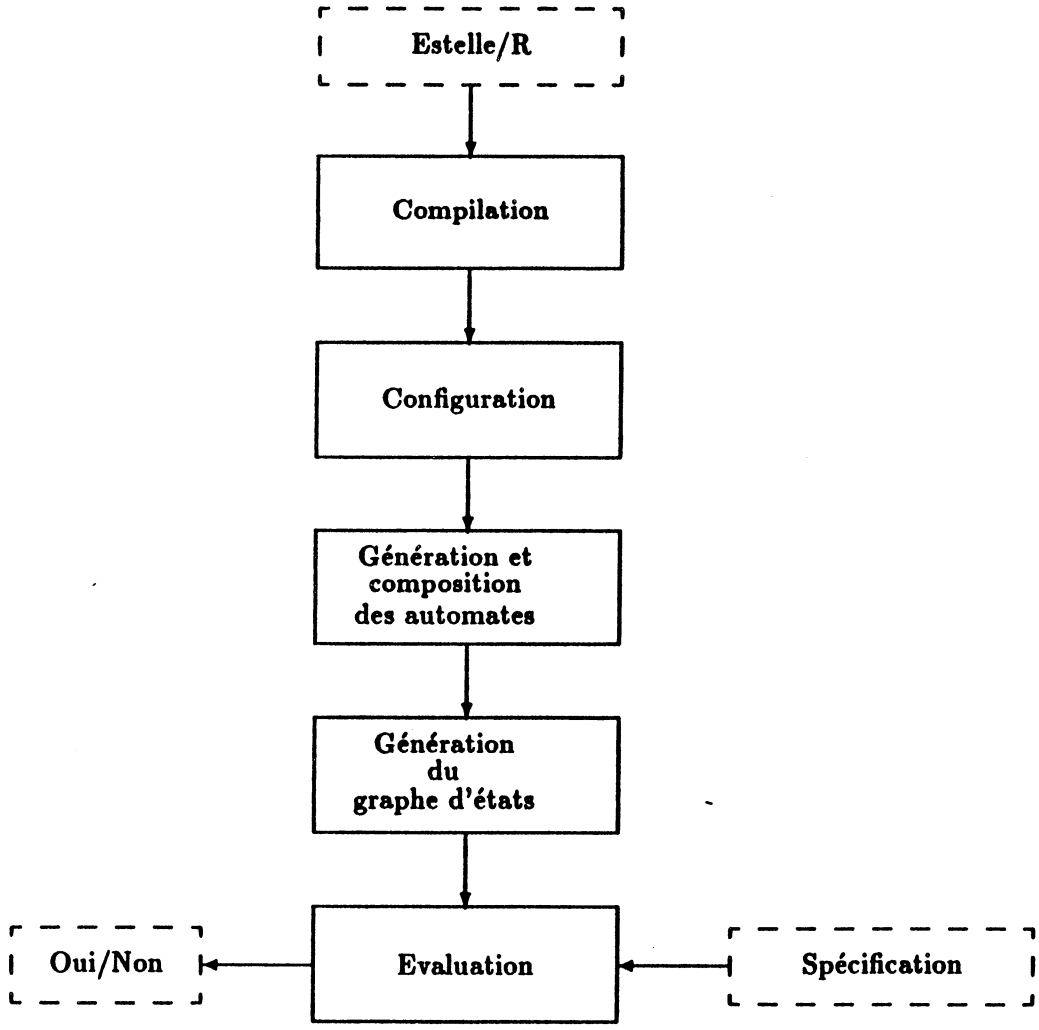

Figure 4.2: Organisation de XESAR.

Configuration.

Cette phase exécute les parties d'initialisation des corps de modules du programme ESTELLE/R pour créer les instances et pour établir les liens entre les points d'interaction. Les instances non-vides (contenant au moins une transition) sont appelées tâches. Les informations nécessaires pour la création de ces instances sont tirées de la représentation interne générée par la phase de compilation.

Génération et composition des automates.

Cette phase construit un automate fonctionnellement équivalent au programme ESTELLE/R. Ceci est fait en deux étapes : d'abord, chaque tâche est traduite en un automate séquentiel ; ensuite, l'ensemble d'automates séquentiels est composé en couplant les instructions d'entrée-sortie correspondant aux communications entre les automates séquentiels.

Génération du graphe d'états.

Cette phase réalise la simulation exhaustive de l'automate composé pour générer un graphe d'états. Chaque état de ce graphe est déterminé par une combinaison de valeurs de l'ensemble de variables de l'automate composé.

L'évaluation de formules.

Cette phase reçoit la spécification donnée par l'utilisateur. Pour chaque formule, une représentation interne est générée et ensuite évaluée sur le graphe d'états. Comme résultat de cette évaluation, l'utilisateur obtient des diagnostics indiquant si sa description est conforme à la spécification.

Réalisation de XESAR.

La réalisation de XESAR a été divisée en deux parties. La première comportait la mise en œuvre des phases de compilation, configuration et génération et composition des automates. Cette partie a été réalisée par J.L. Richier et est décrite dans [Ricre]. Les phases de génération du graphe d'états et d'évaluation des formules ont été réalisées par l'auteur de ce document. Les chapitres 5 et 6 contiennent des descriptions des mises en œuvre de ces phases. Dans la section 4.3 on schématise le processus de génération, à partir du programme ESTELLE/R, d'une représentation intermédiaire qui sera l'entrée de la phase de génération des graphes d'états.

4.3 Transformation du programme ESTELLE/R vers une forme intermédiaire.

A partir du système de processus défini par le programme ESTELLE/R, XESAR génère un ensemble d'automates séquentiels communicants qui sont ensuite composés pour construire un seul automate. La simulation exhaustive de cet automate composé produit le graphe d'états. Dans cette section on décrit brièvement la génération des automates et la façon dont ils sont composés.

4.3.1 La génération des automates.

Chaque automate généré par XESAR est composé d'un ensemble d'états $\mathcal{E}$, d'un ensemble de transitions $\mathcal{T}$ et d'une relation de transition $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{T} \times \mathcal{E}$. Chaque transition est une paire (c, e, a) où c est une condition qui détermine si la transition peut être exécutée, e décrit l'effet de son exécution sur l'état actuel du système et a est le nom de l'action associée à la transition.

On considère ci-dessous une partie d'un programme ESTELLE/R qui contient la description des transitions à partir d'un état nommé A. On montre l'ensemble des états et des transitions de l'automate produits par XESAR à partir de A.

- Pour la transition :

  from A **to** B **when** *p1 x* **provided** *cond*
  begin *bloc1*; **output** *p2 e*; *bloc2*; **end**;

  XESAR génère l'automate de la figure 4.3. On remarque dans cet automate la génération d'un état intermédiaire à partir duquel il n'existe qu'une transition possible, conditionnée par l'émission de l'expression e à travers le point d'interaction $p2$.

- Pour l'ensemble de transitions :

  from A **to** B **delay**(m, n) **provided** *c1* **do** *bloc1*
  from A **to** D **when** *pi x* **provided** *c2* **do** *bloc2*

  XESAR génère l'automate de la figure 4.4. Pour chaque transition contenant la clause **delay**, on introduit un compteur qui compte les unités de temps écoulées depuis l'arrivée à l'état d'origine de cette transition. Sur la figure on observe que chaque fois que l'on atteint l'état A, ce compteur est remis à 0. A partir de cet état, il y a trois transitions possibles :

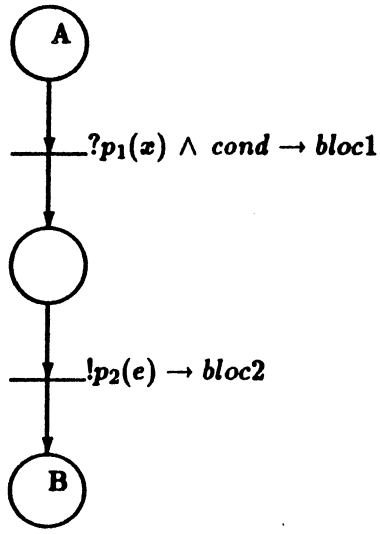

Figure 4.3:

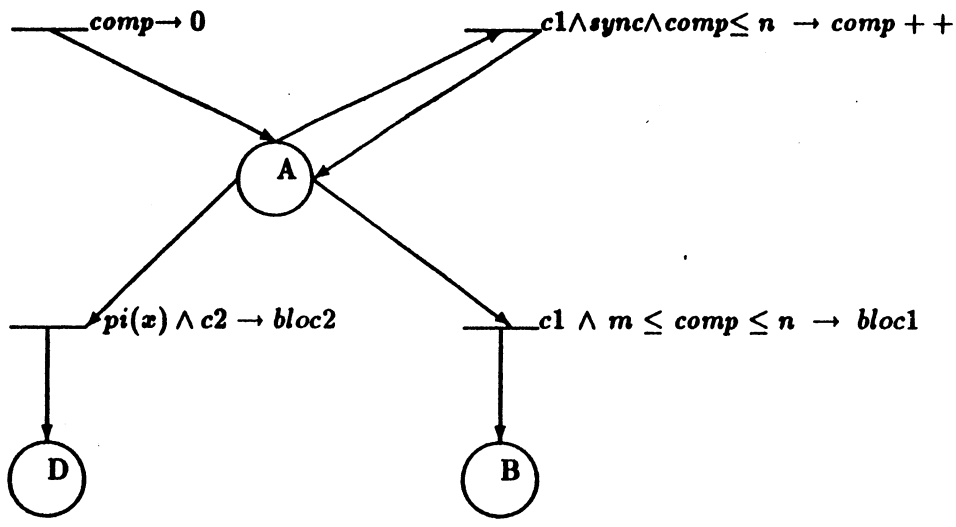

Figure 4.4:

- La transition de l'état A à l'état A, qui sert à faire le comptage des progressions du temps. Ces progressions se font si *cond1* est vrai, si le compteur est inférieur ou égal à n et si la condition *sync* est vraie. *sync* est vraie lorsque tous les processus du système sont prêts à réaliser une transition synchrone. Ceci signifie que chacun des participants doit être dans un état à partir duquel il existe des transitions synchrones dont la condition spécifiée par la clause **provided** est vraie.
- La transition de A à B, exécutable si le compteur du délai est dans l'intervalle $[n, m]$ et c1 est vrai.
- La transition de A à D, exécutable s'il y a une réception par le point *pi* et *cond2* est vrai.

4.3.2 La composition des automates.

La composition des automates consiste à coupler les transitions correspondant aux paires émission-réception par un même point d'interaction. Ces paires sont transformées en transitions ne comportant que des affectations des expressions énumérées dans l'instruction **output** aux variables spécifiées dans la clause **when**.

A titre d'exemple, considérons la figure 4.5. La composition des deux automates ne modifie que les transitions où participe le point d'interaction *pi*. Dans l'automate composé, il existe une seule transition avec deux entrées et deux sorties, ayant comme condition *cond* où on a substitué e à x ; l'action de cette transition est la concatenation de *a1* et *a2* précédée de l'affectation à x des valeurs de e.

4.3.3 Le modèle sémantique de ESTELLE/R

Au langage ESTELLE/R, on a associé un modèle sémantique basé sur l'algèbre des processus temporisés définie dans [RSV86]. La principale caractéristique de cette algèbre est la partition de l'ensemble d'actions en deux classes : les actions *synchrones* et les actions *asynchrones*. Une action synchrone ne peut être exécutée que si toutes les composantes du système participent à son exécution. Les actions asynchrones n'ont pas besoin de la participation de toutes les composantes.

En ESTELLE/R à chaque transition est associée un ensemble d'actions. Les actions relatives aux transitions internes à un processus ou relatives aux communications, sont des actions asynchrones. En revanche, les actions associées aux transitions qui marquent la progression du temps sont des transitions synchrones. Ceci signifie que l'avancement du temps est réalisé

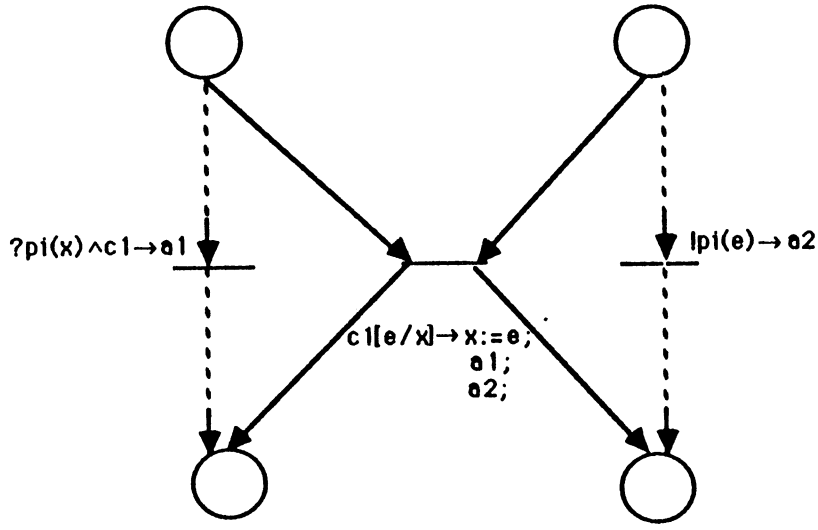

Figure 4.5: Composition des automates.

au même moment par toutes les composantes du système et uniquement si toutes peuvent le faire.

Chapitre 5

La génération de graphes d'états en Xesar.

La phase de génération de graphes a pour but la construction d'un graphe d'états à partir de l'automate composé produit par la phase de composition des automates.

L'automate d'entrée est une structure :

$$\mathcal{A} = (T, V, \Sigma, Tr)$$

où :

- T est l'ensemble de tâches qui constituent le système de processus communicants.
- V est l'ensemble de variables d'états. Il est l'union des ensembles V_t contenant les variables d'une tâche t et les compteurs générés par les constructions **delay**.
- Σ est l'union des ensembles Σ_t contenant les états d'une tâche t.
- Tr est l'ensemble de transitions de $\mathcal{A}$. Chaque transition tr de Tr est de la forme :

  $$tr = (type, R, C, \vec{A}, L)$$

  où :

  - *type* est le type de la transition : échange, synchrone ou interne.
  - R est un ensemble de couples d'états. Un élément (e, e') de R est tel que ses deux composantes appartiennent à la même tâche

t. Les premières composantes de ces couples sont appelés *états d'entrée* de *tr* et les deuxièmes *états de sortie*. Intuitivement, si les composantes de (e, e') appartiennent à *t*, l'exécution de *tr* change l'état courant de *t* de *e* à e'.

- C est une condition sur les valeurs des variables d'état.
- $\vec{A}$ est une liste d'affectations de la forme $v_i \leftarrow a_i(\vec{V})$, où les v_i sont des éléments de V et les a_i sont des fonctions du même type que v_i et prenant leurs arguments dans l'ensemble des valeurs des variables d'états.
- L est l'ensemble des noms des actions associées à *tr*. L'ensemble d'actions de l'automate est composé des étiquettes associées aux transitions du programme ESTELLE/R et des *échanges*. En ESTELLE/R chaque point d'interaction peut émettre (ou recevoir) plusieurs ensembles de valeurs. Chaque ensemble de valeurs est appelé *signal* et chaque paire *point d'interaction-signal* est appelée *échange*. Donc, on a un nom d'action pour chaque signal d'un *point d'interaction*. Le nom d'une telle action est le nom du point d'interaction suivi du nom du signal.

5.1 Le *type* des transitions.

Les transitions de l'automate composé peuvent être classifiées en trois types :

Les transitions d'échange. Ces transitions correspondent au couplage des actions d'émission et de réception du programme ESTELLE/R. Comme le montre la figure 4.3, l'ensemble R associé à ces transitions comporte deux couples d'états.

La condition d'une transition d'échange porte sur les valeurs des variables des deux tâches qui interviennent dans la communication. Les fonctions du vecteur $\vec{A}$ modifient uniquement des variables appartenant à ces tâches. L'ensemble L comporte des noms d'actions de t_i et t_j, plus le nom du point d'interaction utilisé pour la communication.

Les transitions synchrones. Les transitions synchrones représentent la progression du temps (par exemple, la transition de l'état A à l'état A dans la figure 4.4, est une transition synchrone). Pour chaque transition du programme ESTELLE/R comportant la clause **delay**, une transition synchrone est générée. L'ensemble R d'une transition synchrone contient un couple d'états identiques. Sa condition est la conjonction d'une condition locale et une condition globale :

- la condition locale est la conjonction de la condition de la clause provided de la transition ESTELLE/R et d'un test de la valeur du compteur associé
- la condition globale (représentée par *sync* en dans la section 4.3) qui n'est vérifiée que si toutes les tâches peuvent exécuter des transitions synchrones.

Les transformations a_i de $\vec{A}$ ne modifient que les compteurs des délais et la liste d'actions contient une action χ qui dénote la progression du temps.

Les transitions internes. Les transitions internes correspondent aux opérations réalisées par une tâche t sans le concours des autres éléments du système. L'ensemble R contient un seul couple d'états de t.

La condition d'une transition de ce type porte uniquement sur l'espace des valeurs des variables de la tâche t. $\vec{A}$ ne contient qu'un élément : une fonction dont les arguments sont dans V_t et qui peut modifier les éléments de cet ensemble de variables.

5.2 Le graphe d'états.

Le graphe d'états résultant, $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$, est produit par la simulation exhaustive de l'automate $\mathcal{A}$. Chaque élément q de Q est de la forme :

$$q = (M, U)$$

où $M = (\alpha_1, \ldots, \alpha_n)$ et $U = (u_1, \ldots, u_m)$, n et m étant respectivement, le nombre de tâches et de variables de l'ensemble V.

M représente *le marquage* de q. C'est une liste de n entiers $\alpha_1, \ldots, \alpha_n$ tels que α_i est le numéro d'un état de la i-ième tâche. Si $q = (M, U)$ avec $M = (\alpha_1, \ldots, \alpha_n)$, alors l'état numéroté par α_i est marqué en q.

U est l'ensemble des valeurs de variables de q.

Deux états du graphe, $q = (M, U)$ et $q' = (M', U')$ de Q sont égaux si et seulement si :

$$\forall\ 1 \leq i \leq n,\ \alpha_i = \alpha'_i \quad \text{et}$$

$$\forall\ 1 \leq i \leq m,\ u_i = u'_i$$

A partir de l'automate $\mathcal{A}$, l'ensemble des états Q et l'ensemble des relations R_a du graphe sont générés par l'algorithme suivant :

L'algorithme général de génération des graphes :

1. $e \leftarrow$ **Init()**
   /* Init est une fonction qui génère l'état initial du graphe */
2. $Q \leftarrow \{e\}$
3. ATraiter$\leftarrow \{e\}$
4. $\forall a \in E$
   $R_a \leftarrow \{\}$
5. **tantque** ATraiter $\neq \emptyset$ **faire**
   (a) Soit $e \in$ATraiter
   ATraiter$\leftarrow$ ATraiter$-\{e\}$
   (b) $\forall\, tr \in Tr,\ tr = (type, R, C, \vec{A}, L)$
   i. Si $c(e)$ alors $e' \leftarrow f(e)$
   /* c **est une fonction booléenne qui évalue la condition** C **de** tr **sur les valeurs de** e**.** $f(e)$ **est une fonction qui génère un nouvel état après l'application des transformations associées à** tr **sur les valeurs de l'état** e ***/**
   ii. $\forall a \in L$
   $R_a \leftarrow R_a \cup \{(e, e')\}$
   iii. Si $e' \notin Q$ alors
   $Q \leftarrow Q \cup \{e'\}$
   ATraiter$\leftarrow$ATraiter$\cup\{e'\}$
6. Fin.

5.3 Mise en œuvre du simulateur.

A la différence des autres phases de XESAR, dont les fonctions sont réalisées par des programmes écrits en langage C, la génération du graphe est effectuée par un programme PASCAL. Ce choix est dû au fait que les variables d'états sont définies par des constructions PASCAL. L'utilisation du langage C aurait impliqué la traduction de ces contructions en constructions C et cette traduction est hautement dépendante des implantations de ces deux langages.

Le choix de PASCAL conditionne la façon dont le simulateur reçoit l'automate à simuler :

- Les éléments qui définissent les valeurs d'un état, c'est-à-dire le marquage et l'ensemble de variables, sont reçus sous la forme d'une construction *record* de PASCAL appelée *TypeEtat*.

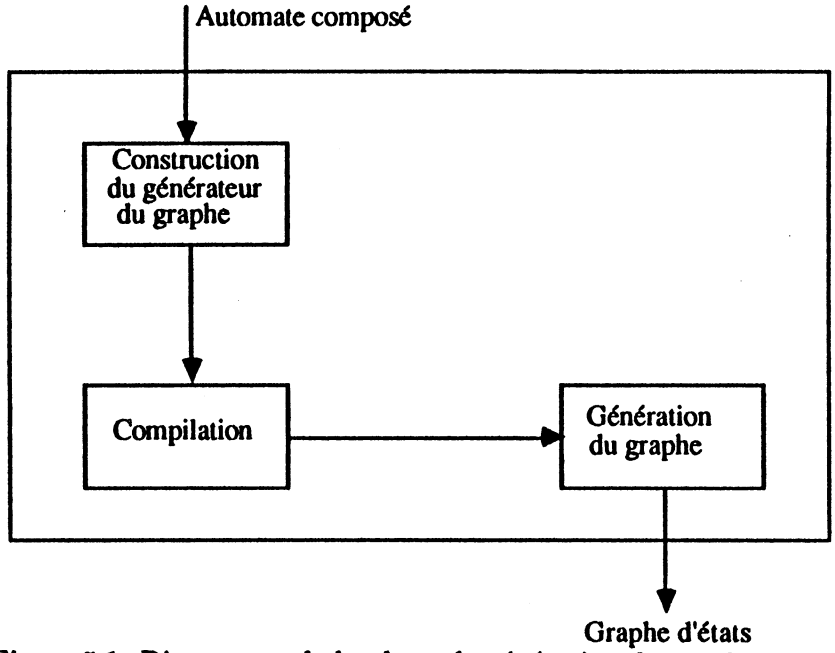

Figure 5.1: Diagramme de la phase de génération du graphe.

- **Les parties C et $\vec{A}$ associées à une transition sont intégrées dans une procédure PASCAL avec une construction conditionnelle *if* qui teste une condition booléenne équivalente à C. Si la condition est vraie, la procédure exécute un bloc d'instructions réalisant les transformations décrites par $\vec{A}$.**

Ceci implique que pour simuler il faut d'abord *construire* un programme PASCAL, appelé *simulateur*, en composant :

1. **Une partie générée par la phase de composition des automates, contenant les définitions des constantes, types et variables nécessaires pour la définition du type *TypeEtat*. De plus, cette partie comporte les procédures condition-action associées à chaque transition, une procédure d'initialisation qui produit l'état initial du graphe et une fonction qui compare les valeurs de deux états.**

2. **Une partie *constante* qui exécute à partir de chaque état toutes les transitions possibles.**

Le programme produit par cette composition est compilé et le résultat de son exécution est le graphe d'états. Une vision schématique de cette phase est montrée en figure 5.1.

5.4 Entrées-Sorties.

5.4.1 Entrées

La phase de génération du graphe a comme entrées :

- Quatre fichiers contenant des structures PASCAL. Les trois premiers contiennent des définitions de constantes, types et variables, et le quatrième l'ensemble des procédures nécessaires pour la construction du simulateur.
- Un fichier texte contenant le *squelette* de l'automate : des informations qui permettent la détermination des ensembles T, Σ_t et les champs *type*, R et L des transitions. Les autres éléments de l'automate proviennent des quatre fichiers PASCAL.

Le fichier texte représente deux fonctions bijectives ; la première, f_1, associe à chaque tâche un ensemble d'états ; la deuxième, f_2, donne pour chaque état de l'automate un ensemble de transitions. Les images par f_2 sont telles que si e et e' sont deux états de l'automate composé, alors $f_2(e) \cap f_2(e') = \emptyset$, c'est-à-dire on peut associer chaque transition à un seul état. Pour faire ceci, on associe les transitions d'échange aux états des tâches réceptrices.

En utilisant ces conventions, on peut associer à chaque état de l'automate composé un numéro entre 1 et $|\Sigma|$. De manière analogue, à chaque transition de cet automate on associe un numéro de l'intervalle $[1..|Tr|]$. Les numérotations sont telles que tous les états d'une tâche et toutes les transitions d'un état ont des numéros consécutifs.

Le fichier qui contient le squelette de l'automate est divisé en quatre zones : Entête, Tâches, Etats et Transitions, qui contiennent les informations suivantes :

Entête : Contient trois valeurs entières : le nombre de tâches n, la cardinalité de l'ensemble d'états et la cardinalité de l'ensemble de transitions.

Tâches : Spécifie, pour chaque tâche t, le numéro de son premier état et la cardinalité de son ensemble d'états Σ_t.

Etats : Donne, pour chaque état p, quatre entiers : le numéro de sa première transition, le nombre de transitions internes, le nombre de transitions d'échange et le nombre de transitions synchrones.

Transitions : Spécifie, pour chaque transition, les données suivantes :

- Son type : i pour les transitions internes, e pour les transitions d'échange et s pour les transitions synchrones.
- Une liste de couples (place-entrée,place-sortie) dont les éléments appartiennent à une même tâche. Si le type de la transition est *i* ou *s*, il existe un seul couple. Dans le cas de transitions d'échange le nombre de couples est 2.
- Une liste d'entiers qui contient les numéros des actions associées à la transition et, dans le cas de transitions d'échange, le numéro du point d'interaction utilisé pour la communication.

5.4.2 Sorties.

Le simulateur produit trois fichiers :

1. Un fichier contenant la description du graphe avec des paramètres tels que : son nombre d'états, son nombre d'arcs et la cardinalité de son ensemble d'actions ; ce fichier contient également le nombre d'états et le nombre de transitions de l'automate composé.

2. Un fichier qui contient l'ensemble des états du graphe. Chaque état du graphe est un enregistrement PASCAL du type *TypeEtat*.

3. Un fichier qui contient l'ensemble des arcs du graphe. Chaque arc est composé de trois valeurs entières e_d, tr et e_a qui représentent le numéro de son état de départ, le numéro de la transition exécutée et le numéro de l'état d'arrivée, respectivement.

5.5 Structures de données.

Représentation du squelette de l'automate.

Le squelette de l'automate est représenté par les trois structures suivantes :

TâchesEtats :

Cette structure garde, pour chaque tâche, le numéro de son premier état et le nombre d'états qu'elle contient. Elle est utilisée pour déterminer la tâche à laquelle appartient un état de l'automate composé et pour vérifier si dans les marquages, l'état associé à une tâche appartient à cette tâche.

EtatsTrans

Cette structure garde, pour chaque état de l'automate composé, quatre valeurs entières : le numéro de la première transition sortant de cet état et le nombre de transitions internes, d'échange et synchrones qui lui sont associées.

Transitions

Cette structure contient, pour chaque transition de l'automate composé, son type, ses couples (place-entrée,place-sortie) et la liste des numéros des actions qui lui sont associées.

Les états et les arcs.

Les ensembles des états et des arcs du graphe sont stockés dans des tableaux. Pour l'ensemble d'états, le type de chaque élément est *TypeEtat* et chaque arc est un triplet contenant trois valeurs entières.

Organisation de l'ensemble d'états.

Lors de la simulation exhaustive il est nécessaire de vérifier si chaque état généré appartient déjà à l'ensemble des états du graphe. Ceci oblige à comparer les valeurs de e avec les valeurs de *tous* les autres éléments de l'ensemble d'états Q.

Il est donc important de définir une partition de Q tel que l'on puisse associer facilement à chaque état un élément de cette partition. De cette manière, déterminer si un état appartient à Q se limite à déterminer s'il appartient au sous-ensemble d'états qu'on lui associe.

La technique la plus utilisée pour faire ces partitions d'ensembles est l'adressage dispersé. Dans notre cas, la fonction d'adressage dispersé est appliquée aux valeurs d'un état et le résultat est un pointeur vers une liste d'états. Ces listes sont organisées sous la forme de *B-arbres*.

La figure 5.2 montre un exemple de cette structure.

La représentation de l'ensemble d'arcs pose moins de problèmes parce que le simulateur n'utilise pas les arcs déjà générés. Ceci nous permet de stocker l'ensemble des arc dans un fichier.

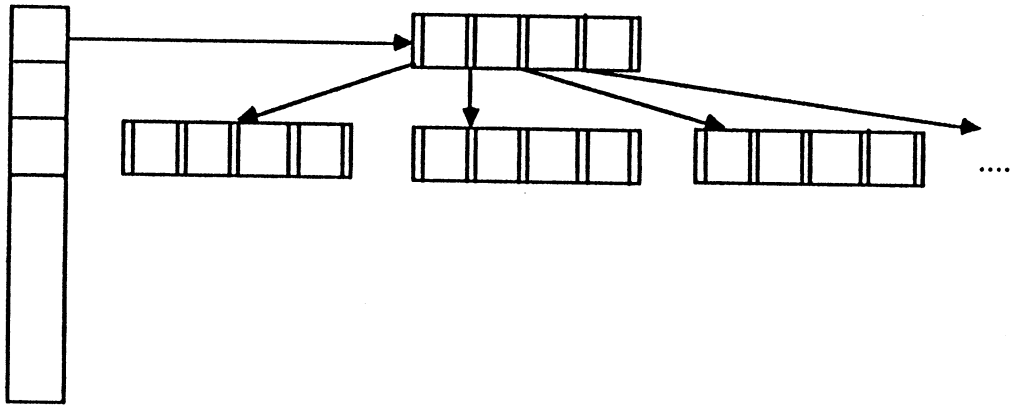

Figure 5.2: La représentation des états.

5.6 L'algorithme détaillé de génération des graphes.

Dans l'algorithme que l'on montre ci-dessous, le pointeur *Courant* indique l'indice de l'état dont on génère les successeurs. Ceci permet de déterminer l'ensemble ATraiter comme les états du tableau placés entre *Courant* et l'indice du dernier état inséré.

Définition 15 *Une transition tr de l'automate* $\mathcal{A} = (T, V, \Sigma, Tr)$ *est dite* tirable *à partir de l'état q si et seulement si :*

- *tr est de type interne ou échange, les places d'entrée de tr sont marquées en q et la condition C de tr est vraie en q.*
- *tr est de type synchrone, toutes les places marquées dans q sont l'origine de transitions synchrones et les conditions de toutes ces transitions sont vraies en q.*

L'algorithme détaillé de génération des graphes est :

Procédure Géneration_de_graphes

1. *e* ← **Init()**
   /* e est une variable de type TypeEtat. **Init** génère l'état initial du système */

2. *Courant* ← 1
3. NombreEtats ← 1
4. PutVar(e,*Courant*)
   /* PutVar insère e dans le tableau d'états et met à jour le B-arbre correspondant*/
5. **tantque** *Courant* ≤ NombreEtats **faire**
   (a) GetVar(e,*Courant*)
   /* GetVar cherche dans l'ensemble d'états celui qui a comme index *Courant* et rend en e les valeurs de ses variables */
   (b) **Pour toute** transition tr **faire**
      i. *exécutée* ← CondAct(tr, e)
      /* CondAct est une fonction booléenne qui rend vrai si la transition tr est tirable à partir de e. Dans le cas affirmatif, elle rend dans la variable globale e' les valeurs de l'état produit par les transformations associées à tr */
      ii. Si *exécutée* alors
      Si NouvelEtat(e') alors
      /* La fonction NouvelEtat détermine si e' appartient à l'ensemble d'états */
      NombreEtats ← NombreEtats + 1
      PutVar(e',NombreEtats)
      EcrireArc(e, tr, e')
   (c) *Courant* ← *Courant* + 1
6. Fin.

Cet algorithme peut être paramétré par :

- La fonction d'adressage dispersé qui détermine le B-arbre dans lequel on cherche un état.
- Les paramètres du B-arbre.
- La localisation des valeurs des états : mémoire ou disque. Si l'on prend l'option disque, la version de PASCAL utilisée doit permettre l'utilisation de fichiers à accès direct.

5.7 Conclusion

Cette implantation de la phase de génération des graphes d'états rempli les objectifs fixés. Les perfomances obtenues sont satisfaisantes. A titre

d'exemple, considérons le protocole de Stenning. Le graphe généré à partir de la description de l'annexe B possède 9144 états et 18892 arcs. L'exécution du programme PASCAL qui produit ce graphe prend 54 secondes sur un SUN3/50.

Un aspect important pour l'obtention de ces perfomances est la structure de données choisie pour organiser les collisions de la fonction d'adressage dispersé. La version actuelle de XESAR utilise la somme modulo 29 des octets qui constituent les valeurs d'un état (marquage plus valeurs de variables). Chaque B-arbre a un maximun de 20 éléments.

Pour le protocole de Stenning, la simulation exhaustive conduit à la génération de 18893 états, dont uniquement 9144 sont différents. La structure de données choisie limite le nombre de comparaisons à 127783, ce qui donne en moyenne 6,8 comparaisons par état du graphe.

Si l'on prend l'option qui garde l'ensemble d'états en mémoire, le simulateur alloue des blocs de taille *Lg* au fur et à mesure qu'il en a besoin. Ceci peut conduire à un épuisement de l'espace disponible et, en conséquence, à une erreur système qui arrête l'exécution. La plupart des versions de PASCAL ne permettent pas la récupération de ces erreurs. Il serait donc souhaitable de trouver des moyens d'intercepter ces signaux d'erreurs et ainsi de donner au simulateur la possibilité de continuer a générer le graphe en stockant les nouveaux états sur disque.

Chapitre 6

L'évaluateur de formules de Xesar.

L'évaluateur permet de déterminer si un graphe d'états est un modèle d'un ensemble de formules du langage de spécification. Le graphe modèle est fourni par la phase de génération de graphes de XESAR et la spécification est donnée par l'utilisateur.

La liaison entre les formules et le programme est effectuée via les prédicats de base. Ils servent à référencer deux types d'entités du programme ESTELLE/R : les variables d'états et les actions. L'ensemble des variables d'état contient toutes les variables définies au niveau principal des tâches. L'ensemble des actions contient l'ensemble des étiquettes du programme ESTELLE/R et l'ensemble des *échanges* utilisés dans les communications.

Avant de décrire le contenu de ce chapitre, on présente les sous-modules qui composent l'évaluateur.

6.1 Décomposition fonctionnelle.

Hormis l'évaluation des formules, la phase d'évaluation réalise plusieurs autres fonctions : l'analyse syntaxique des formules, l'association entre noms externes et internes, etc. De plus, des fonctions complémentaires dérivées des choix de réalisation et des facilités offertes à l'utilisateur. La figure 6.1 montre la façon dont interagissent les sous-modules de l'évaluateur. Ces sous-modules sont :

Le traitement des définitions de synonymes. Dans l'évaluateur toute référence aux entités du programme doit se faire en utilisant leur noms complets : si X est le nom d'une action, ou variable d'état, son nom complet est construit en préfixant par les noms de tous les blocs qui

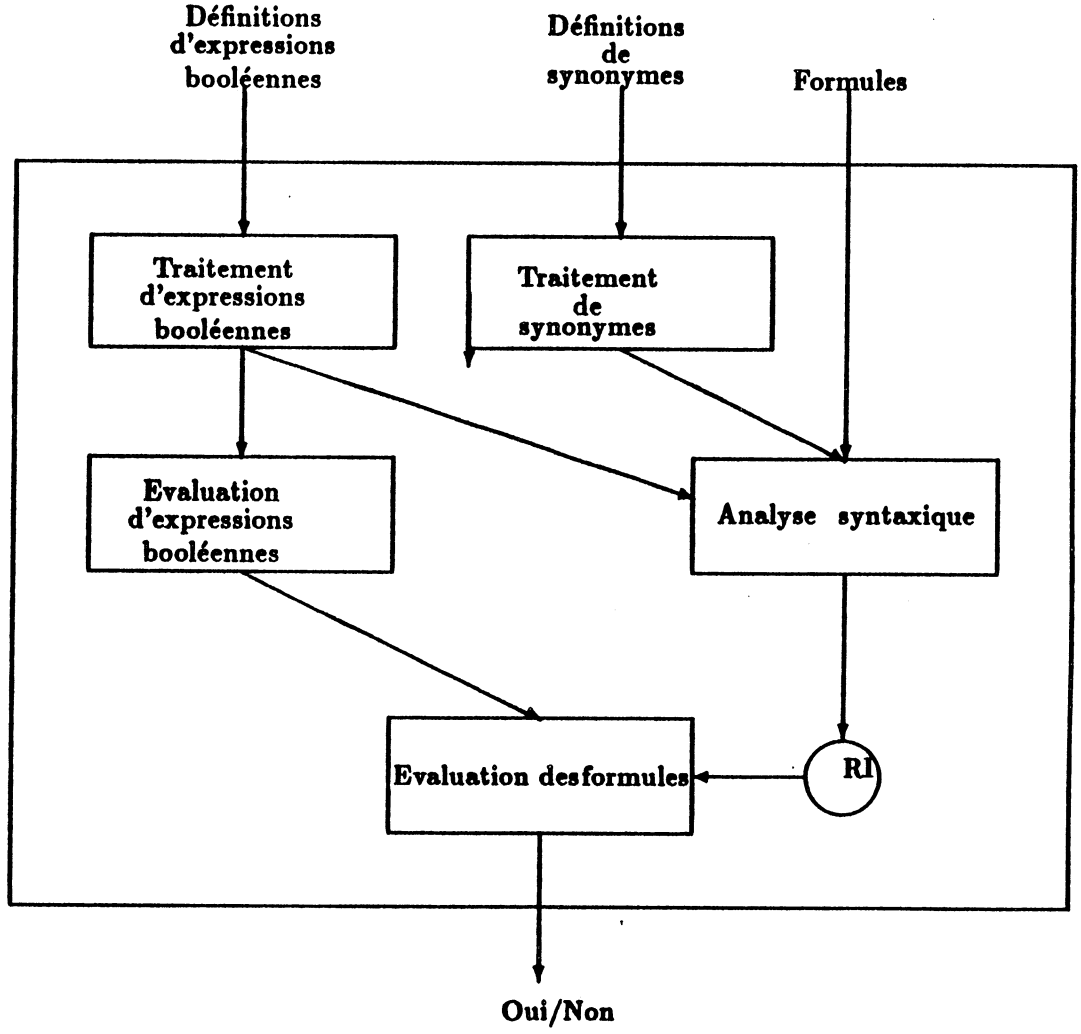

Figure 6.1: Organisation de la phase d'évaluation.

englobent sa définition. Ceci est nécessaire pour enlever les ambiguïtés dans les références à des entités de même nom mais définies dans des tâches différentes.

Pour simplifier cette tâche, l'évaluateur admet la définition de *synonymes*. Un synonyme est un identificateur qui designe un nom complet ou un préfixe d'un nom complet. Par exemple :

la == *t.lowest_unacked*
c1 == *t.message*

sont des définitions de synonymes. Ces définitions permettent d'écrire dans la spécification *la* à la place de *t.lowest_unacked* et *c1.val* à la place de *t.message.val*.

Le sous-module de traitement de synonymes prend les définitions proposées par l'utilisateur, fait leur analyse syntaxique et détermine l'entité du programme ESTELLE/R qu'elles designent. Dans l'annexe D on montre la syntaxe abstraite des définitions de synonymes.

Le traitement des expressions booléennes. L'ensemble des prédicats de base a comme éléments :

- Les expressions booléennes sur les valeurs des variables d'états.
- Les prédicats sur des actions du programme ESTELLE/R : *enable*(B) et *after*(B), où B est un sous-ensemble de l'ensemble d'actions.

Les prédicats *enable* et *after* sont évalués en utilisant l'ensemble des arcs du graphe. L'évaluation des expressions booléennes utilise les valeurs de variables d'états. Ces valeurs ont été stockées sous la forme d'enregistrements PASCAL du type *TypeEtat* et pour y accéder, on a besoin de générer un programme PASCAL qui les lise.

Pour simplifier le processus d'évaluation d'une formule, on a décidé de séparer l'évaluation des expressions booléennes des évaluations des autres éléments de la logique. Ceci implique que les formules qui comportent ce genre de prédicats de base ne sont pas admises directement par l'évaluateur. Par exemple, la formule :

$$inev\,(X > 0)$$

où X est le nom complet d'une variable d'état, doit être proposée à l'évaluateur de la façon suivante :

1. On définit un prédicat qui a comme corps l'expression booléenne souhaitée et à laquelle on associe un nom :

$$sup == X > 0$$

2. Après l'évaluation de ce prédicat, c'est-à-dire après la génération, compilation et exécution du programme PASCAL qui lui correspond, la formule

   inev sup

   peut être évaluée.

Le sous-module de définition d'expressions booléennes prend les définitions proposées par l'utilisateur, fait leur analyse syntaxique et génère une représentation interne.

Le sous-module d'évaluation d'expressions booléennes produit à partir de ces représentations internes un programme PASCAL dont l'exécution calcule leurs ensembles caractéristiques.

L'analyse et l'évaluation de formules. Ces sous-modules de la phase d'évaluation font l'analyse syntaxique des formules pour construire leurs représentations internes. Ces représentations internes sont ensuite évaluées pour déterminer la validité des formules.

La suite de ce chapitre contient :

1. Une description des entrées et sorties de l'évaluateur.
2. Une explication sommaire de la transformation *after*. Ceci est nécessaire puisque cette transformation a beaucoup d'influence sur les choix des structures de données.
3. La description des structures de données utilisées.
4. La syntaxe abstraite des formules.
5. La représentation interne des formules.
6. L'évaluation de ces représentations internes.

6.2 Entrées et sorties :

6.2.1 Entrées :

Les entrées de cette phase sont :

- Le *descripteur du graphe* généré par la phase de génération des graphes d'états.

- Les ensembles des états et des arcs du graphe.
- Les ensembles des actions associées à chaque transition.
- La *représentation interne d'associations* (RIA), constituée d'un arbre et d'un ensemble de tables. Elle est produite par la phase de composition des automates et permet à l'évaluateur de faire la liaison entre les noms complets de variables ou d'actions et leurs noms internes.
- Les formules qui constituent la spécification du programme.

Le descripteur du graphe est un fichier qui contient des paramètres utilisés pour déterminer les tailles des structures de données de l'évaluateur. Ces paramètres sont :

- Le nombre d'états du graphe.
- Le nombre d'arcs du graphe.
- Le nombre d'étiquettes du programme ESTELLE/R.
- Le nombre d'actions du programme ESTELLE/R.
- Le nombre de transitions de l'automate composé.

L'ensemble d'états est un fichier contenant les enregistrements du type *TypeEtat* qui représentent les valeurs des états.

L'ensemble des arcs du graphe est contenu dans un fichier où chaque arc est représenté par trois valeurs entières : le numéro de l'état de départ, le numéro de la transition exécutée et le numéro de l'état d'arrivée.

Les ensembles des actions de chaque transition proviennent d'un fichier généré par la phase de composition des automates.

La représentation interne d'associations.

La RIA est constituée d'un arbre et d'un ensemble de tables construits à partir de l'automate composé. Elle contient toutes les informations nécessaires pour associer le nom externe des actions et des variables à leur nom interne.

L'arbre de la RIA est un résumé de la représentation interne de l'automate composé. Cet arbre garde uniquement le schéma des définitions des entités du programme ESTELLE/R.

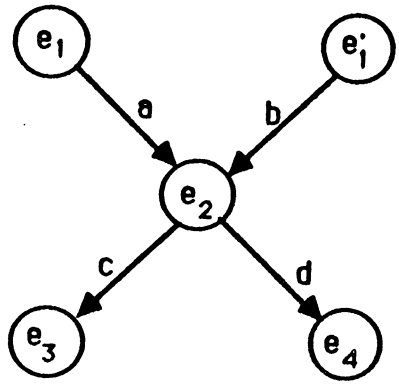

Figure 6.2:

6.2.2 Sorties :

Pour chaque formule de la spécification, l'évaluateur détermine sa *validité*. Une formule est dite *valide* si tous les états du graphe la satisfont.

6.3 La transformation *after*.

Lors de la génération du graphe d'états, deux états $e = (M, U)$ et $e' = (M', U')$ sont égaux si et seulement si $M = M'$ et $U = U'$. Ceci pose des problèmes de compatibilité avec la logique choisie au moment de l'évaluation du prédicat de base qui utilisent l'opérateur *after*. On détaille ces problèmes :

Si a est une action, *after*(a) est vrai dans tous les états du graphe qui sont accessibles *juste après* l'exécution de a et qui sont uniquement accessibles après l'exécution de cette action. Dans le graphe de la figure 6.2, on observe que l'état e_2 est accessible à partir de e_1 par l'exécution de a. Néanmoins, on ne peut pas affirmer que e_2 satisfait *after*(a) puisqu'il est accessible à partir de e'_1 par l'exécution de b.

Une solution à ce problème est le renforcement du critère d'égalité entre états : deux états e_1 et e_2 sont égaux si et seulement si, leurs représentations sont égales et les actions exécutées pour produire e_1 sont les mêmes que celles qui ont été exécutées pour produire e_2. Cette solution est impraticable puisqu'elle conduit à une explosion du nombre d'états.

En XESAR on a voulu donner à l'utilisateur la possibilité de concevoir sa spécification même après la génération du graphe d'états. Le critère d'égalité d'états est celui décrit au début de cette section et, en conséquence, il est nécessaire de faire des modifications locales à la structure du graphe pour assurer un traitement correct des prédicats construits en utilisant l'opérateur

after. Ces modifications sont appelées *les tranformations after*. Voici la description des pas à suivre pour la transformation *after*(a) du graphe $G = (Q, E, \{\xrightarrow{a}\}_{a \in E}, i_G)$:

Procédure transformation *after*(a)

1. $solution \leftarrow \emptyset$
2. Pour tout état accessible e_1 de Q tel qu'il existe e_2 en Q et $(e_2, e_1) \in R_a$.
   (a) Soit $pred_a(e_1) = \{e_2 | (e_2, e_1) \in R_a\}$
   /* $pred_a(e_1)$ contient les états possédant des arcs étiquetés par a qui mènent à e_1 */
   (b) Si $(\cup_{a' \in E} pred_{a'}(e_1) \neq \{\})$ et $(pred_a(e_1) \neq \{\})$ alors
      i. $e' \leftarrow$ **nouvel_etat**(e_1)
      /***nouvel_état** est une fonction qui crée un nouvel état avec les mêmes valeurs que e_1*/
      ii. $Q \leftarrow Q \cup \{e'\}$
      iii. $solution \leftarrow solution \cup \{e'\}$
      iv. $R_a \leftarrow R_a \cup \{(e_2, e') | e_2 \in pred_a(e_1)\}$
      v. $R_a \leftarrow R_a - \{(e_2, e_1) | e_2 \in pred_a(e_1)\}$
      vi. $\forall b \in E \; R_b \leftarrow R_b \cup \{(e', e'') | (e_1, e'') \in R_b\}$
3. Fin.

Les états qui satisfont le prédicat *after*(a) sont les éléments de l'ensemble *solution*.

Exemple 9 *La figure 6.3 montre le résultat de l'application de la transformation after(a) sur le graphe de la figure 6.2.*

L'algorithme de transformation a les conséquences suivantes :

- Le pas numéro 2(b)ii *rajoute* un état au graphe. Si e' est un état créé en répliquant les valeurs de un état e_1, e' est dit *image* de e_1.
- Les pas 2(b)iv et 2(b)v *éliminent* les arcs de la forme $e_2 \xrightarrow{a} e_1$ et rajoutent des arcs $e_2 \xrightarrow{a} e'$.
- Le pas 2(b)vi *rajoute*, pour chaque arc $e_1 \xrightarrow{a} e_2$ un arc $e' \xrightarrow{a} e_2$.

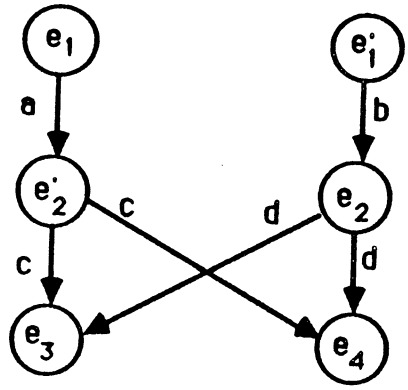

Figure 6.3: Evaluation du prédicat *after*(*a*).

Ceci implique que la taille du graphe peut augmenter pendant la phase d'évaluation.

Un des choix de réalisation de XESAR est de ne pas réaliser des allocations de mémoire pendant l'exécution de ses programmes. Ceci implique qu'au début de chaque phase on reserve une quantité d'espace estimée suffisante pour satisfaire tous les besoins du module. Si cette estimation s'avère en dessous des nécessités réelles, le programme est arrêté et l'utilisateur doit refaire ses estimations.

Dans le cas particulier de l'évaluateur, l'utilisateur donne des pourcentages maximaux de croissance des nombres d'arcs et d'états. Par exemple, si le graphe a M arcs et N états et l'utilisateur prévoit une croissance maximale de p_m pour-cent pour le nombre d'arcs et de p_n pour-cent pour le nombre d'états, alors les structures de données sont définies pour contenir un maximun de

$$MAX\_ETATS = N + \left\lceil \frac{N * p_n}{100} \right\rceil$$

états et un maximun d'arcs égal à :

$$MAX\_ARCS = M + \left\lceil \frac{M * p_m}{100} \right\rceil$$

6.4 Structures de données.

6.4.1 L'ensemble des arcs du graphe.

L'ensemble des arcs est représenté par deux tableaux, l'un appelé *arcs* et l'autre appelé *premier_arc*. Le premier contient *Nombre_arcs* couples, *Nombre_arcs* étant la cardinalité de l'ensemble des arcs. Les premières composantes des éléments de ce tableau sont des numéros de transition et les deuxièmes des numéros d'état. Elles correspondent aux transitions et états de départ des arcs du graphe. Le tableau *premier_arc* donne, pour chaque état *e*, l'index du premier élément du tableau *arcs* qui a *e* comme état d'arrivée. L'intervalle [*premier_arc*(q),*premier_arc*(q+1) -1] donne les numéros des entrées du tableau *arcs* contenant les numéros des transitions et les numéros des état de départ des arcs arrivant à l'état *q*. Dans la figure 6.4 on montre un exemple de cette structure. On remarque que :

- Il n'y a pas d'arc arrivant à 1.
- Les arcs arrivant à 2 sont dans les entrées dont les numéros sont dans l'intervalle [*premier_arc*(2),*premier_arc*(3) -1]=[1,3] (arcs $1 \xrightarrow{a} 2$, $3 \xrightarrow{b} 2$ et $4 \xrightarrow{c} 2$).
- Les arcs arrivant à 3 sont dans les entrées dont les numéros sont dans l'intervalle [*premier_arc*(3),*premier_arc*(4) -1]=[4,5] (arcs $4 \xrightarrow{a} 3$ et $1 \xrightarrow{b} 2$).
- Les arcs arrivant à 4 sont dans les entrées dont les numéros sont dans l'intervalle [*premier_arc*(4),*premier_arc*(5) -1]=[6,7] (arcs $1 \xrightarrow{c} 4$ et $3 \xrightarrow{b} 4$).

Cette représentation des arcs permet de faire des économies de mémoire par rapport à la structure de données qui garde les trois composantes de chaque arc. Un aspect important c'est le fait que les algorithmes d'évaluation des opérateurs de la logique sur cette structure ont une complexité équivalente à la complexité des algorithmes sur la structure plate.

6.4.2 Le tableau des ensembles d'actions.

Chaque arc du graphe comporte le numéro de la transition de l'automate composé qui a été exécutée pour le générer. Lors de l'évaluation des formules, l'utilisateur peut faire des références aux actions associées à cette transition.

Dans l'évaluateur on représente ces ensembles par des listes de bits. Ceci est possible parce que l'on a donné aux actions des noms internes qui sont

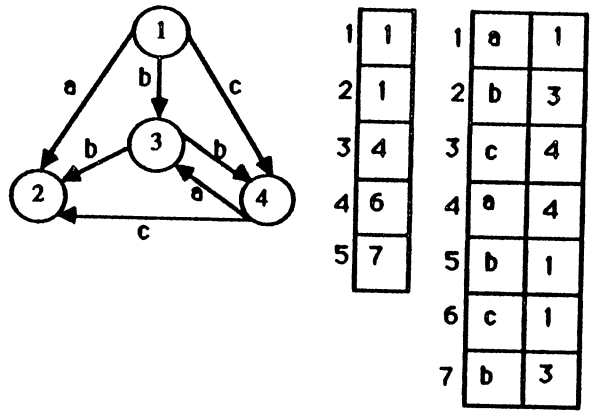

Figure 6.4: Représentation de l'ensemble des arcs du graphe.

des entiers entre 1 et la cardinalité de l'ensemble. L'appartenance de l'action i à un ensemble est donc vérifiée en regardant si la valeur du ième bit est 1.

6.4.3 Les registres-prédicats.

Pendant l'évaluation d'une formule, l'évaluateur peut produire des résultats intermédiaires. Par exemple, pour la formule :

$$pot[f]\,g$$

on évalue d'abord les formules f et g. Ces évaluations donnent les ensembles caractéristiques des deux formules qui sont ensuite utilisés pour évaluer l'opérateur temporel *pot*.

Les résultats intermédiaires sont gardés dans des structures que l'on appelle des *registres-prédicats*. Chaque registre contient les champs suivants :

Status indique si le registre est *Inactive* (n'est pas utilisé) ou *Active* (est déjà utilisé).

Valeur est une liste de bits qui représente l'ensemble caractéristique du prédicat.

Longueur est une valeur entière qui indique le nombre d'états qu'avait le graphe après l'évaluation de la sous-formule dont l'ensemble caractéristique est gardé dans ce registre. Ce champ est utilisé pour déterminer si les registres-prédicats arguments d'un opérateur de la logique, ont le même nombre d'éléments. Par exemple, soit la formule :

$$enable(a) \wedge after(b)$$

Si le nombre des états au début de l'évaluation de *enable*(*a*) est n, après son évaluation l'ensemble caratéristique aura n éléments significatifs. L'évaluation de *after*(*b*) peut occasionner l'augmentation du nombre d'états et en conséquence son ensemble caractéristique aura n' états.

Avant l'évaluation de la conjonction il est nécessaire de faire une *normalisation* des ensembles caractéristiques. Pour ce faire, on fixe à n' la longueur de l'ensemble associé à *enable*(*a*) et ensuite, pour tous les nouveaux états (tous les états entre $n+1$ et n') on réplique la valeur de l'état dont il est l'image ; par exemple, si l'état e est une image de l'état e' et e' appartient à l'ensemble caractéristique de *enable*(*a*), alors e appartient aussi a cet ensemble.

6.4.4 Tableau des valeurs des expressions booléennes.

L'exécution du programme PASCAL correspondant à une expression booléenne donne comme résultat une liste de bits qui représente l'ensemble caractéristique de l'expression. Cet ensemble est lu par l'évaluateur et stocké dans un tableau d'ensembles. Le nombre des ensembles de ce tableau est spécifié par l'utilisateur.

6.4.5 Le tableau d'images.

Le tableau d'images contient, pour chaque état créé par les transformations *after*, le numéro de l'état dont il est la réplique.

6.5 La syntaxe abstraite des formules.

L'ensemble de formules admises par l'évaluateur est généré par les règles suivantes :

```
formule_specif :        declaration_formule
                        | corps_formule
```

```
declaration_formule :   ident  "==" corps_formule

corps_formule :         formule
                        | systeme

formule :               formule "=>" formule
                        | formule "<=>" formule
                        | formule "<>" formule
                              /* "<>" est l'ope'rateur xor*/
                        | formule "and" formule
                        | formule "or" formule
                        | formule_plus
                        | "not" formule
                        | op_temp arg_temp formule
                        | "<" l_label ">" formule
                        | formule_simple

formule_simple :        basic_predicate
                        | "pre"  "(" formule ")"
                        | "pretilda"  "(" formule ")"
                        | "true"
                        | "false"
                        | "(" formule ")"
                        | form_point_fixe

formule_plus :          formule_rond { "+" formule_rond }*

formule_rond :          "{" l_label "}" formule

l_label :               label { "," label }*

arg_temp :              { "[" formule "]" }

op_temp :               "pot"   | "al"  | "inev"  | "some"  | "fair"

label :                 alias_symbole_etiq
                        | "tau"
                        | "chi"

form_point_fixe :       op_pf ident "." formule_simple

op_pf :                 "gfp" | "lfp"

def_systeme :           "systeme" "var" l_var ";" l_eqn

l_var :                 ident { "," ident}*

l_eqn :                 eqn { eqn }*
```

```
eqn :          ident signe formule_simple

signe :        "=>" | "<= "
```

Remarques :

- La priorité des opérateurs est, en ordre croissant :
  - =>, <=>, <> (opérateurs non-associatifs)
  - and, or
  - + (uniquement permis entre des termes préfixés par {})
  - <>, {}, not, pot, al, some, inev, fair
  - pre, pretilda
- `chi` est l'action qui représente la progression du temps.
- `tau` est l'action fictive portée par les transitions internes sans étiquette.
- `alias_symbole_etiq` est un `alias_symbol` correspondant à une étiquette XESAR.
- `gfp` et `lfp` correspondent respectivement, aux opérateurs de plus grand et plus petit point fixe.
- Parmi les facilités offertes à l'utilisateur pour l'écriture des spécifications, l'évaluateur donne la possibilité d'associer un nom à une formule. La syntaxe de ces *définitions* de formules est :

  $$ident == f$$

  où *ident* est un identificateur et f est un élément de $\mathcal{F}$.

  Pour évaluer une formule, l'utilisateur a deux possibilités : soit il propose à l'évaluateur la formule f et il obtient comme réponse un message indiquant la validité de la formule, soit il propose une définition $ident == f$; dans ce cas il obtient aussi le message qui indique la validité de f et, de plus, l'évaluateur crée une entrée dans la table de symboles où il garde l'ensemble caractéristique de f sous le nom *ident*. Ce nom peut être utilisé dans des autres formules.
- Dans la syntaxe on remarque la possibilité de définir des systèmes d'équations de point fixe. Des systèmes de ce type ont été introduits en [Dic85] et implantés dans le système MEC. En XESAR ils sont

nécessaires pour pouvoir évaluer les systèmes dérivés des graphes de sûreté.

Les systèmes admis par XESAR sont de la forme :

$$\begin{array}{ll} \textbf{var } x_0, x_1, \ldots, x_n & \\ \quad x_0 & \texttt{signe}_0\ f_0(x_0, \ldots, x_n) \\ \quad \vdots & \quad \vdots \\ \quad x_n & \texttt{signe}_n\ f_n(x_0, \ldots, x_n) \end{array}$$

où **signe**$_i$ est => ou <= pour indiquer que x_i est le plus petit (respectivement, le plus grand) point fixe de $f_i(x_0, \ldots, x_n)$.

Les systèmes admis par XESAR doivent satisfaire les conditions suivantes :

- Si **signe**$_i$ est égal à => et **signe**$_j$ est égal à => alors, toute occurrence de x_j dans f_i doit être sous un nombre *pair* de négations ; si **signe**$_j$ est égal à <=, toute occurrence x_j dans f_i doit être sous un nombre *impair* de négations.
- Si **signe**$_i$ est égal à <= et **signe**$_j$ est égal à <= alors, toute occurrence de x_j dans f_i doit être sous un nombre *pair* de négations ; si **signe**$_j$ est égal à =>, toute occurrence x_j dans f_j doit être sous un nombre *impair* de négations.

Syntaxe concrète des predicats de base.

L'ensemble des prédicats de base admis par XESAR est :

```
basic_predicate:        "after" "(" l_label ")"
                      | "enable" "(" l_label ")"
                      | "init"
                      | "sink"
                      |  ident
```

Remarque :

ident représente une expression booléenne ou une formule nommée.

6.6 La représentation interne des formules.

L'analyse syntaxique des formules produit une représentation interne sous la forme d'un arbre. Chaque nœud de cet arbre a un type et à chaque type de nœud sont associées deux types d'informations : ses fils, si le nœud n'est pas un nœud terminal, et ses attributs, qui sont des informations générales du nœud (par exemple, si le nœud correspond à la définition d'un canal, un attribut est le pointeur vers la chaîne de caractères qui donne son nom externe).

Pour définir formellement les représentations internes, on prend les conventions suivantes :

- Une *classe* est un ensemble de nœuds dont un élément peut apparaître en partie droite de la définition d'un nœud. Les noms de classes commencent par une lettre minuscule.
- Les noms de nœuds commencent par une majuscule. Le nœuds sont constitués de trois zones. La première contient son nom et est placée sur la partie gauche de la ligne. A droite du nom et séparée par le symbole "⇒", on trouve sa liste d'attributs. A droite aussi et précédées du symbole "→", on peut trouver les informations suivantes :
  1. La chaîne "()" qui signifie que le nœud est une feuille sans information.
  2. La chaîne "implanté comme un symbole". Dans ce cas, le nœud est aussi une feuille correspondant à la définition d'un élément de la table de symboles.
  3. Un nom de nœud suivi de la chaîne "* ..." qui signifie que le nœud est de type liste.
  4. Une liste de noms de nœuds séparés par des blancs. Cette liste est la liste des fils du nœud.
- Le caractère "*" devant la définition d'un nœud signifie qu'il s'agit d'un nœud terminal.
- Le caractère "R" devant la définition d'un nœud signifie qu'il s'agit d'une racine possible de l'arbre.

Description des nœuds :

```
R Actpred        →Opevl          L_Etiq
                 /* Prédicats sur les actions*/
```

```
R Binaire       →form          OperatBin     form
                /* Formules avec deux arguments*/

  Etiq          ⇒Nom_interne
                /* Une étiquette */
  L_Etiq        →Etiq *...
                /* Liste d'étiquettes */
  L_Form_Next   →Nextime *...
                /* Liste de formules associées à l'opérateur plus */

  L_Var_Evl     →Var_Evl *...
                /* Liste de variables d'un système d'équations*/

R Nextime       →Opevl         form          L_Etiq
                /* Opérateurs dits de nextime*/
R OperOparam    ⇒type
                /* Opérateurs sink et init*/
  Opevl         ⇒type
                /*Opérateurs μ ν enable after */

  OperatBin     ⇒type
                /* Opérateurs pot inev al ∧ ∨ ≡ ⟹  */

  Opplus        →L_Form_Next
                /* Opérateur plus */

R Parenthese    →form

R Pointfixe     ⇒Pt_var
                →Opevl         form
                /*Opevl est μ ou ν */

R Refform       ⇒Num_reg
                /* Référence à une formule */

R Refpred       ⇒Num_reg
                /* Référence à une expression booléenne */

  Refvarevl     ⇒Num_reg
                /* Référence à une variable (nœud permis uniquement
                dans les formules de point fixe)*/

R Systeme       →L_Var_Evl     L_form
```

/* Définition d'un système d'équations */

Var_Evl →implanté comme un symbole
/* Variable d'un système ou d'un opérateur de point fixe */

Les classes :

form ::=Parenthese Pointfixe Nextime Oper0param Systeme
Opplus Binaire Actpred Refform Refpred

6.7 L'évaluation des formules.

Pour évaluer une formule, on parcourt sa représentation interne en évaluant chaque nœud de l'arbre. Ces évaluations sont réalisées par la fonction *eval_arbre* qui reçoit un paramètre de type représentation interne de formule et rend le numéro d'un registre-prédicat contenant l'ensemble caractéristique de la formule. Le corps de cette formule est :

fonction *eval_arbre*(f)

1. **case** *type*(f) **of**
   /* *type* est une fonction qui rend le type du nœud */

   Binaire : (a) *reg1*← *eval_arbre*(*fils*(f,1))
   /* *fils*(f,i) rend le i-ième fils de f */
   (b) *reg2*← *eval_arbre*(*fils*(f,3))
   (c) *op*← *fils*(f,2)
   (d) Retourner *eval_binaire*(*op*,*reg1*,*reg2*)

   Unaire : (a) *reg*← *eval_arbre*(*fils*(f,2))
   (b) *op*← *fils*(f,1)
   (c) Retourner *eval_unaire*(*op*,*reg*)

   Oper0param : (a) *op*← *fils*(f,1)
   (b) Retourner *eval_0param*(*op*)

   Nextime : (a) *reg*← *eval_arbre*(*fils*(f,2))
   (b) *op*← *fils*(f,1)
   (c) *ens*← *ens_actions*(*fils*(f,3))
   /* *ens_actions* rend un ensemble contenant les éléments de la liste d'actions */
   (d) Retourner *eval_nextime*(*op*,*reg*,*ens*)

   Actpred : (a) *op*← *fils*(f,1)

(b) *ens*← *ens_actions*(*fils*(f, 2))

(c) Retourner *eval_actpred*(*op*,*reg*,*ens*)

Opplus : (a) *liste_ens*← *l_ens_actions*(*fils*(f, 1))
/*_l_ens_actions_ construit une liste avec les ensembles d'actions des n arguments de l'opérateur */

(b) *liste_reg*← *l_ens_form*(*fils*(f, 1))
/*_l_ens_form_ construit une liste contenant les n ensembles caractéristiques des formules préfixées par les ensembles d'actions */

(c) Retourner *plus*(*liste_reg*,*liste_ens*)

Refvarevl :

Refform :

Refpred : Retourner *valeur_attribut(f,1)*
/* Le premier attribut de ces nœuds est le numéro du registre-predicat où se trouve l'ensemble caractéristique de l'élément référencé */

Paranthese : Retourner *eval_arbre*(*fils*(f,1))

Pointfixe : Retourner *eval_point_fixe*(f)

Système : Retourner *eval_systeme*(f)

2. Fin.

Dans la fonction *eval_binaire* on appelle une des fonctions *and*, *or*, *implic*, *pot*, *inev*, *al* ou *fair* si l'opérateur est de type $\wedge, \vee, \supset$, *pot*, *inev*, *al* ou *fair*, respectivement.

Dans la fonction *eval_unaire* on appelle une des fonctions *not*, *pre* ou $\widetilde{pre}$ si l'opérateur est de type *not*, *pre* ou $\widetilde{pre}$, respectivement.

Dans la fonction *eval_0param* on appelle une des fonctions *sink* ou *init* si l'opérateur est *sink* ou *init*, respectivement.

Dans la fonctions *eval_nextime* on appelle une des fonction *los* ou *prefixage* si la formule est de type $<A>f$ ou $A.f$, respectivement.

Dans la fonction *eval_actpred* on appelle une des fonctions *enable* ou *after* si l'opérateur est de type *enable* ou *after*, respectivement.

6.8 Algorithmes :

Dans cette section on montre les algorithmes qui évaluent les formules de la logique. Chaque algorithme agit sur le graphe qui a *Nombre_Etats* états et *Nombre_Arcs* arcs. On utilise les fonctions suivantes :

- *new_pred*() : rend le numéro d'un registre-prédicat libre.
- *release_pred*(n_reg) : libère (met à *inactive*) le registre-prédicat numéro n_reg.
- *valeur*(n_reg) : représente l'ensemble d'états du registre-prédicat n_reg.
- *premier_arc*(*e*) : donne l'index premier arc qui arrive à l'état *e*.
- *dernier_arc*(*e*) : donne l'index du dernier arc qui arrive à l'état *e*.
- *actions*(*tr*) : donne l'ensemble d'actions de la transition *tr*.
- *Depart*(*j*) : donne l'état de départ de l'arc *j*.
- *Trans*(*j*) : donne le numéro de la transition de l'arc *j*.

6.8.1 Les formules *sink* et *init*.

sink

La formule *sink* est vraie dans tous les états qui n'ont pas de successeur. L'algorithme qui l'évalue est :

fonction *sink*()

1. *solution*← *new_pred*()
2. *valeur*(*solution*) ← *vrai*
3. $\forall 1 \leq e \leq$ *Nombre_Etats*
   (a) $\forall$ *premier_arc*(*e*) $\leq j \leq$ *dernier_arc*(*e*)
   valeur(*solution*) ← *valeur*(*solution*) − {*Depart*(*j*)}
4. Retourner *solution*
5. Fin

Cette fonction visite une fois chaque état et une fois chaque arc du graphe. Sa complexité est donc $O($*Nombre_Etats* + *Nombre_Arcs*$)$.

La formule *init*

La formule *init* est vraie dans l'état initial et dans tous les états images de l'état initial. Elle est évaluée par l'algorithme :

fonction *init*()

1. *solution*← *new_pred*()

2. *valeur*(*solution*) ← {1}
   /* L'état initial est l'état numéro 1 */

3. ∀ *Nombre_Etats_Init* < *e* ≤ *Nombre_Etats*
   /* *Nombre_Etats_Init* est le nombre des états initiaux du graphe */
   Si *image*(*e*) = 1 alors
   valeur(*solution*) ← *valeur*(*solution*) ∪ {*e*}

4. Retourner *solution*.

5. Fin.

Cette fonction visite une fois chaque nouvel état créé par des transformations *after*. Le nombre maximal d'images d'un état est *Nombre_Actions* − 1, et en conséquence, la complexité de cet algorithme est $O(Nombre\_Etats * Nombre\_Actions)$.

6.8.2 Les formules *enable*(*ens*) et *after*(*ens*).

La formule *enable*(*ens*)

enable(*ens*) est vrai dans tous les états à partir desquels il existe des arcs qui portent des actions contenues dans *ens*. Voici l'algorithme qui évalue ce type de formules :

fonction *enable*(ens)

1. *solution*← *new_pred*()

2. *valeur*(*solution*) ← *faux*

3. ∀ 1 ≤ *e* ≤ *Nombre_Etats*

   (a) ∀ 1 ≤ *j* ≤ *Nombre_Arcs*
   Si *actions*(*Trans*(*j*)) ∩ *B* ≠ ∅ alors
   valeur(*solution*) ← *valeur*(*solution*) ∪ {*Depart*(*j*)}

4. Retourner *solution*.

5. Fin

Cette fonction visite une fois chaque arc du graphe. Ceci implique que sa complexité est $O(Nombre\_Arcs)$.

La formule *after(ens)*

Dans la section 6.3 on a vu que l'évaluation des prédicats *after* peut modifier la structure du graphe. En particulier, le nombre d'arcs du graphe peut augmenter par l'adjonction de nouveaux états. La génération de ces nouveaux éléments doit se faire de façon à ce que la structure qui garde les arcs reste triée par état d'arrivée.

L'évaluation du prédicat *after*(B), B étant un sous-ensemble de l'ensemble d'actions, peut être schématisée de la façon suivante :

Pas 1. Soit Q l'ensemble d'états avant l'évaluation du prédicat *after*(B). Pour chaque état e :

1. Déterminer s'il est nécessaire d'appliquer la transformation *after* sur cet état, c'est-à-dire déterminer si parmi les arcs qui arrivent à e il existe des arcs portant des actions de B et des arcs ne portant pas des actions de B.
2. Générer un nouvel état e'. Pour chaque arc (e'', t, e) tel que t ne contient pas d'actions de B, créer un nouvel arc (e'', t, e') et éliminer l'arc (e'', t, e).

Pas 2. Pour chaque état e' créé lors de l'exécution du pas précédent, générer les arcs qui l'ont comme origine : si e' est l'image de l'état e, pour chaque arc (e, t, e'') générer un arc (e', t, e'').

Dans la section 6.4 on a décrit la stucture de données utilisée pour stocker les arcs du graphe. La figure 6.5 montre un exemple de cette structure.

Chaque nouvel état généré par le Pas 1 a comme numéro *Nombre_Etats* + 1. Cela signifie que les arcs qui y arrivent doivent être placés *en dessous* du dernier arc du graphe. L'état des structures qui gardent les arcs après l'exécution de cette première partie de l'évaluation du prédicat *after* est montré en figure 6.6.

Par la suite, on appelle la zone occupée par les arcs initiaux *première zone*. La zone à partir du *n_initial_arcs* jusqu'à *Nombre_arcs* est appelée *deuxième zone*. Comme conséquence de l'élimination des arcs dans la première zone, il existe des cases vides. Avant de faire la génération des arcs qui sortent des nouveaux états, on réalise un compactage de la première zone de façon à produire une structure similaire à celle illustrée en figure 6.7.

La génération des arcs qui sortent des nouveaux états est décomposée en deux parties : la génération des arcs des nouveaux états aux nouveaux états et la génération des arcs des nouveaux états aux anciens états. L'emplacement de nouveaux arcs est conditionné par les paramètres suivants :

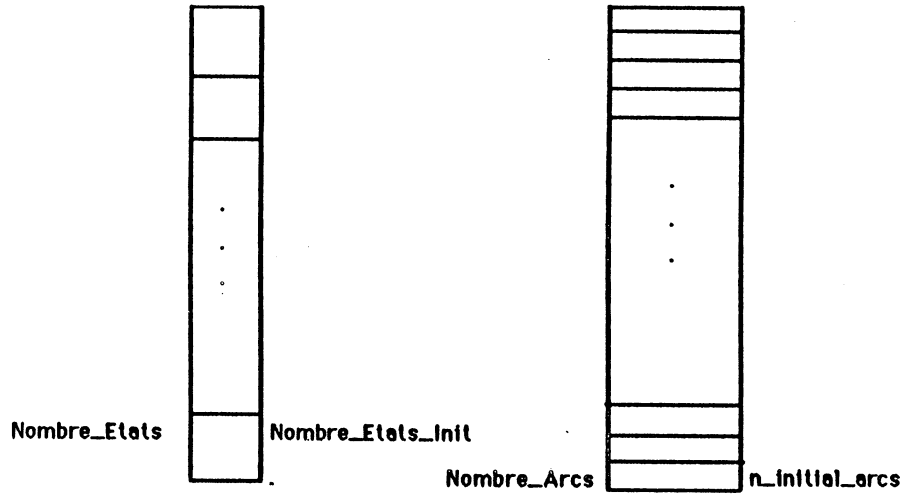

Figure 6.5:

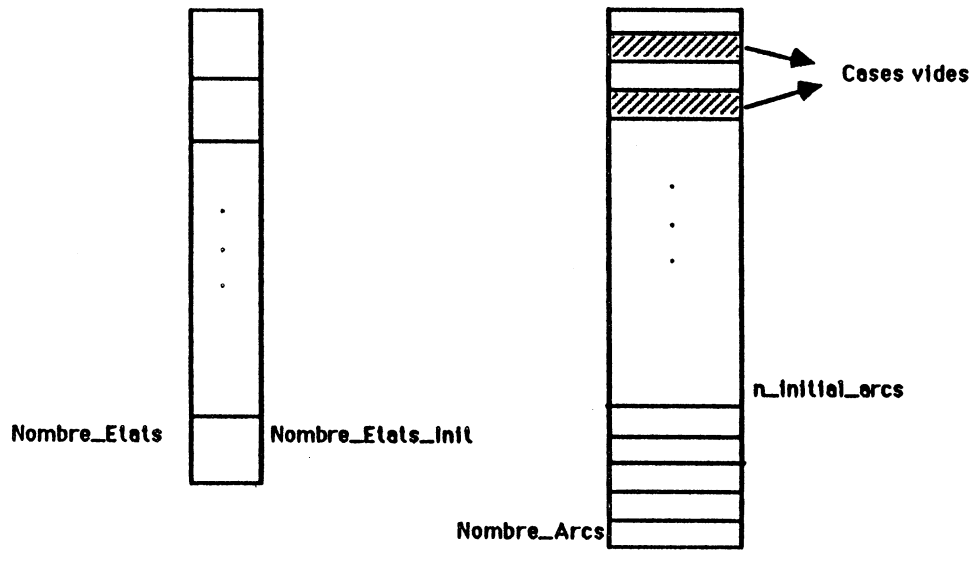

Figure 6.6:

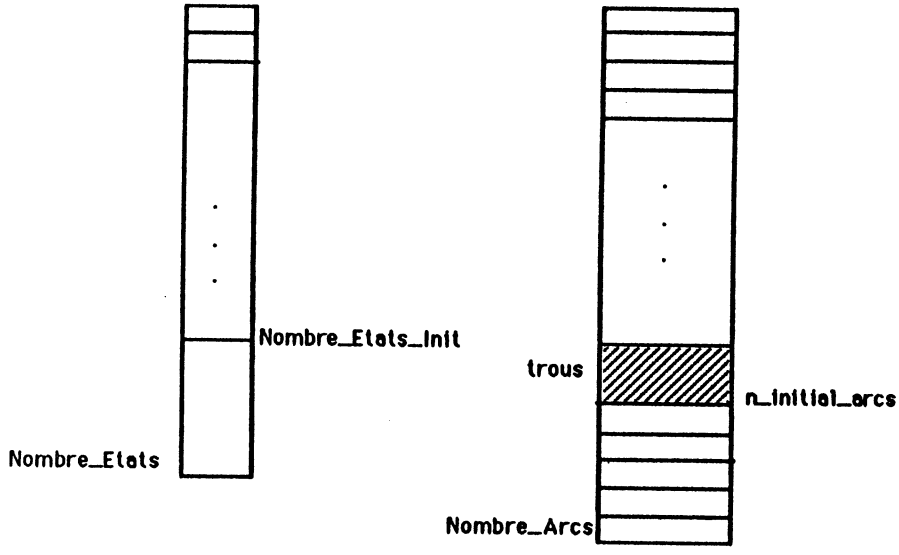

Figure 6.7:

nouveaux_arcs : nombre total d'arcs qui sortent des nouveaux états.

trous : nombre de cases vides à la fin de la première zone.

nouveaux_arcs' : nombre de nouveaux arcs sortant de nouveaux états et arrivant aux nouveaux états.

Après l'exécution du pas 1, la taille du tableau des arcs est *n_initial_arcs* plus le nombre d'arcs de la deuxième zone (ce nombre est égal à *trous*). La somme de ces deux paramètres est égale à *Nombre_Arcs*. La façon dont on fait la génération des arcs des nouveaux états aux nouveaux états dépend des valeurs des paramètres *nouveaux_arcs* et *trous* :

- Si *nouveaux_arcs* est supérieur à *trous*, alors il est nécessaire d'augmenter la taille du tableau de *nouveaux_arcs* − *trous* cases. On obtient le tableau des arcs montré en figure 6.8.

  La génération des arcs ayant comme origine un nouvel état commence à partir de la partie inférieure de la deuxième zone (case signalée par * dans la dernière figure). On copie chaque arc dans la case vide la plus proche de la limite inférieure de la zone allouée (initialement, on utilise la case indiquée par →). Si l'état d'origine de l'arc est un état dupliqué, alors on génère aussi un arc sortant de l'état image et préservant l'état d'arrivée et la transition de l'arc traité. Cet arc est placé juste *en dessus* du dernier arc copié. Cette procédure a deux conditions d'arrêt :

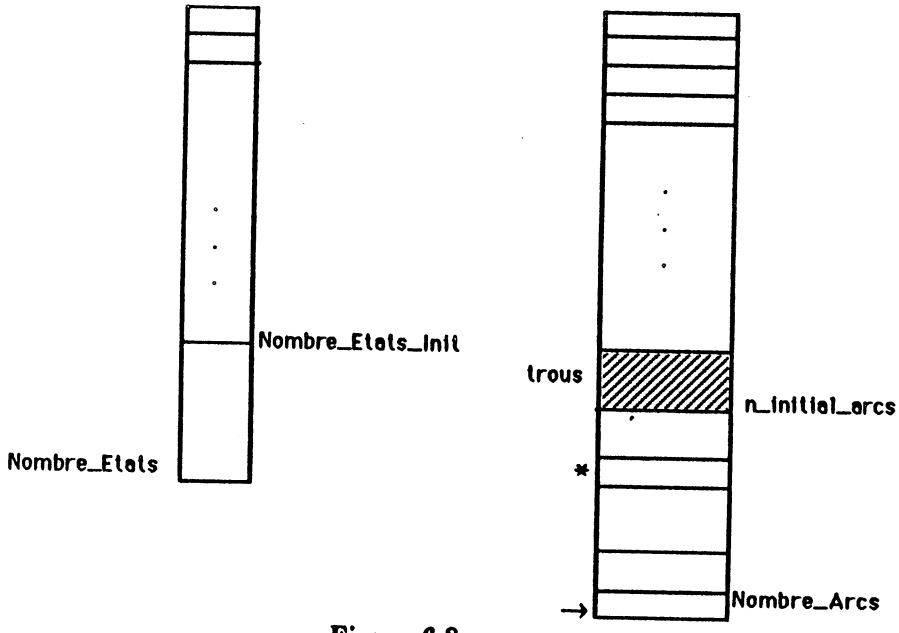

Figure 6.8:

1. On atteint le premier arc de la deuxième zone du tableau. Graphiquement, cette situation est illustrée en figure 6.9.
2. On épuise les cases allouées en bas de la deuxième zone. Ceci signifie que le nombre d'arcs qui arrivent aux nouveaux états (*nouveaux_arcs'*) est supérieur au nombre d'arcs qui arrivent aux anciens états. En conséquence, la génération des arcs des nouveaux états aux nouveaux états recommence à partir du premier arc de la deuxième zone en déplaçant les arcs vers les cases vides placées entre le deux zones. Pendant ce déplacement, chaque arc traité peut engendre un nouvel arc si l'état de départ est un état dupliqué. A la fin de cette procédure, on a la configuration montrée en figure 6.10.

- Si le nombre de nouveaux arcs est inférieur au nombre de trous, on applique la procédure décrite ci-dessus dans le cas où *nouveaux_arcs'* est supérieur au nombre d'arcs arrivant aux ancien états. Le résultat de son application rend un tableau des arcs similaire à celui montré en figure 6.11.

La génération des arcs des nouveaux états aux anciens états est réalisée en parcourant la première zone à partir de son dernier arc. Chaque arc visité est recopié dans la dernière case libre avant la deuxième zone ; si son état de départ est un état dupliqué, on génère un nouvel arc ayant comme origine l'image de cet état. Après l'application de cette dernière procédure, on arrive à un tableau sans cases vides et correctement trié.

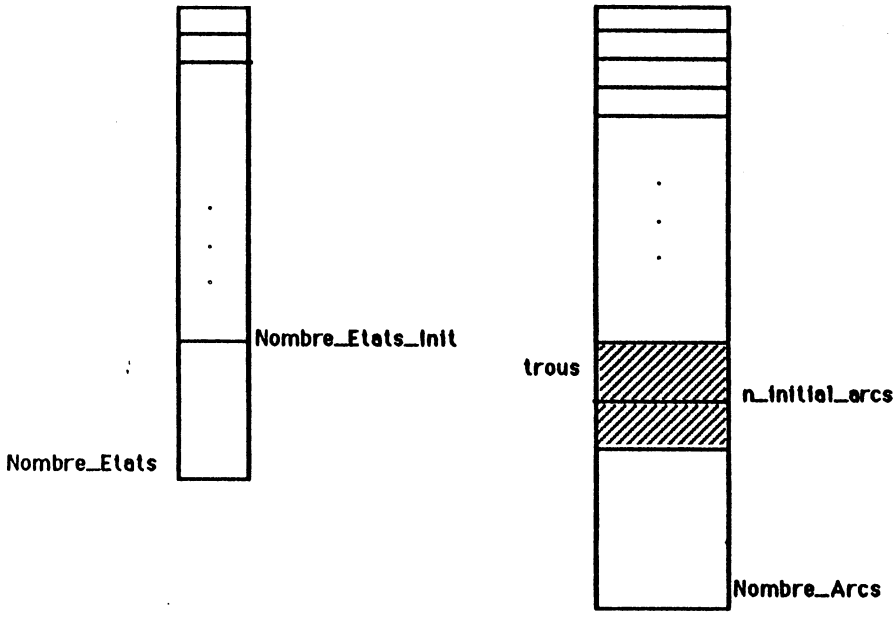

Figure 6.9:

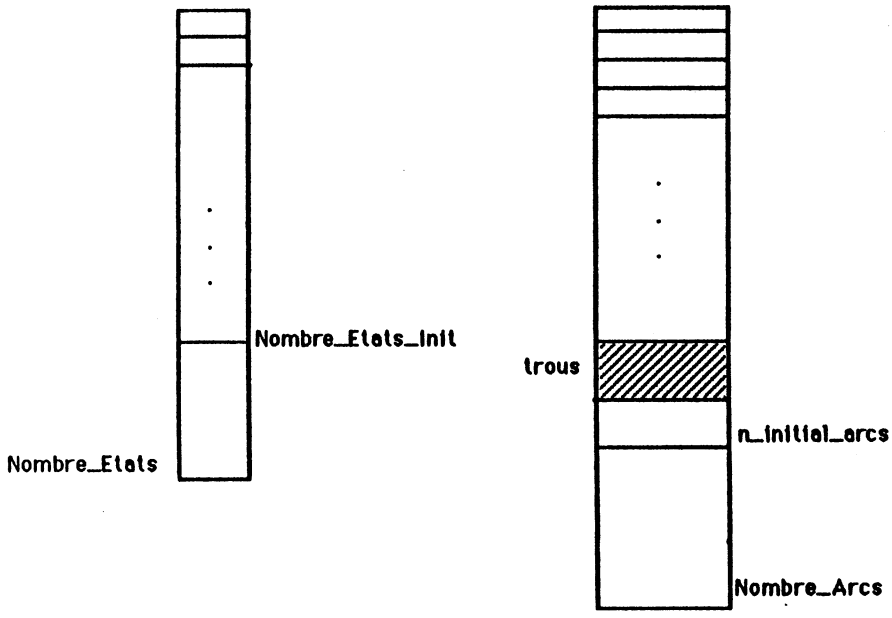

Figure 6.10:

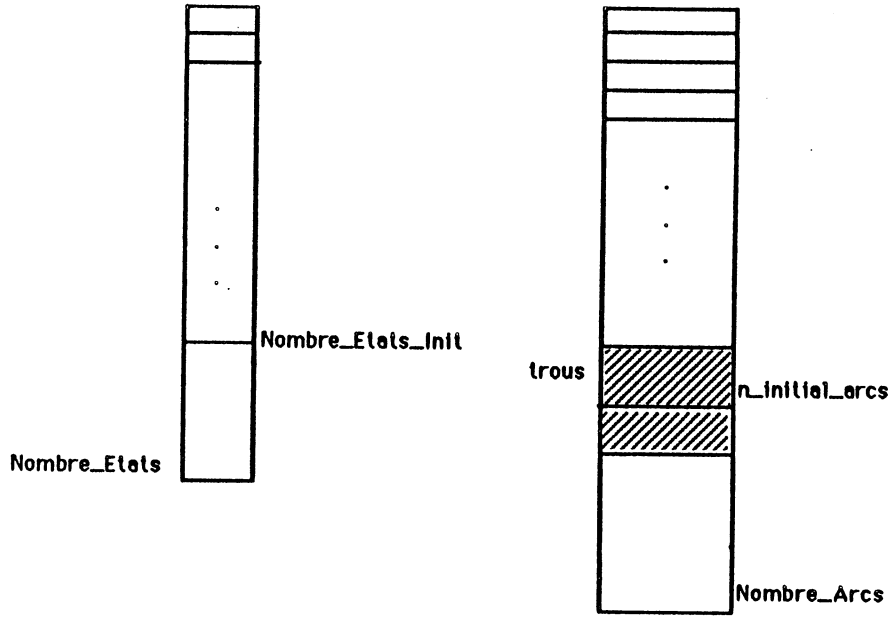

Figure 6.11:

Voici l'algorithme qui réalise l'évaluation du prédicat *after*(*ens*).

fonction *after*(*ens*)

1. *solution* ← *new_pred*()
2. *valeur*(*solution*) ← ∅
3. *netats*←*Nombre_Etats*
   /**netats* garde le nombre d'état au début de l'évaluation de *after*(*ens*)*/
4. *narcs*←*Nombre_Arcs*
   /**narcs* garde le nombre d'arcs au début de l'évaluation de *after*(*ens*)*/
5. *trous*← 0
6. *ptarc*← 1
   /**ptarc* garde le numéro du prochain arc à traiter dans le tableau arcs*/
7. $\forall 1 \leq e \leq$ *MAX_ETATS*
   pre_image(*e*)← 0
   /* Le tableau *pre_image* garde, pour chaque nouvel état, le numéro de l'état dont il est l'image */
8. $\forall 1 \leq e \leq$ *netats*
   Si **actions_differentes**(*e*, *ens*, *EGALES*) alors

(a) $ne \leftarrow$ **nouvel_etat()**
/* **nouvel_etat()** rend le numéro d'un nouvel état */

(b) ***pre_image***$(ne) \leftarrow e$

(c) **deplacer_effacer**(***ptarc***, ***trous***, $e, ne,$ ***ens***)
/* **deplacer_effacer** crée des nouveaux arcs dans la deuxième zone, élimine des arcs dans la première et fait des compactages locaux */

(d) ***nouveaux_arcs*** $\leftarrow$ ***nouveaux_arcs*** $+$ ***nombre_succ***(e)

(e) ***solution*** $\leftarrow$ ***solution*** $\cup \{e\}$

Sinon Si ***EGALES*** alors
/* ***EGALES*** indique si tous les arcs qui arrivent à e portent des actions de l'ensemble ***ens*** */

(a) **deplace**(***ptarc***,e)
/*__deplace__ fait des compactages locaux dans le tableau des arcs*/

(b) ***solution*** $\leftarrow$ ***solution*** $\cup \{e\}$

9. ***nouveaux_arcs***$'$ $\leftarrow$ **aux_n_etats**(***premier_arc***$(ne+1)$, ***Nombre_Arcs***)
/*__aux_n_etats__ calcule le nombre des nouveaux arcs qui arrivent aux nouveaux états */

10. ***diff*** $\leftarrow$ ***nouveaux_arcs*** $-$ ***trous***

11. Si ***diff*** ≥ 0 alors

(a) **grow_down**(***diff***$, ne+1$)
/* **grow_down** alloue ***diff*** nouveaux arcs supplémentaires et génère des nouveaux arcs dans la deuxième zone*/

(b) Si ***nouveaux_arcs***$' >$ ***diff*** alors
grow_up(***nouveaux_arcs***$' - diff, ne+1$)
/* **grow_up** génère de nouveaux arcs à partir de la première casse occupée de la deuxième zone en prenant des espaces vides de la première zone*/

Sinon /* Si ***diff*** > 0 */
grow_up(***nouveaux_arcs***$' -$ ***diff***$, ne+1$)
/* génère les arcs des nouveaux états aux anciens états */

12. **grow_first_zone**($ne+1$, ***nouveaux_arcs***, ***nouveaux_arcs***$'$, ***ptarc*** -1)

13. Retourner ***solution.***

14. Fin

Les algorithmes des fonctions utilisées dans la fonction *after* sont décrits dans l'annexe C.

On a comparé l'évaluation des opérateurs *after* en utilisant les procédures décrites ici à leur évaluation en utilisant une structure de données qui gardait les trois composantes de chaque arc.

Sur cette structure, l'évaluation du prédicat *after*(A) se fait de la façon suivante :

1. Pour chaque état e du graphe, si parmi les arcs qui arrivent à e il existe des arcs portant des actions de A et des arcs ne portant pas d'actions de A, alors, remplacer chaque arc (e'', t, e) tel que t ne contient pas des actions de A, par l'arc (e'', t, e'), e' un nouvel état. Contrairement à la méthode décrite précédemment, l'exécution de ce pas ne produit pas de case vide dans le tableau des arcs.

2. Pour chaque état e' créé lors de l'exécution du pas 1, générer les arcs qui l'ont comme origine. Ces nouveaux arcs sont placés après le dernier arc du graphe initial.

3. On fait appel à la fonction *qsort* fournie par le système UNIX hôte qui effectue un tri rapide de l'ensemble des arcs du graphe.

On a pris un ensemble de 10 prédicats *after* de la spécification du protocole de Stenning et l'on a mesuré le temps que prennait son évaluation en utilisant les deux versions de l'évaluateur. Le temps d'exécution de la version que l'on vient de décrire était presque le double du temps pris par la version présentée au début de cette section.

6.8.3 Les formules $pre(f)$ et $\widetilde{pre}(f)$.

Les formules $pre(f)$

$pre(f)$ est vrai dans tous les états qui ont au moins un arc qui mène à un état contenu dans l'ensemble caractéristique de f. Dans l'algorithme montré ci-dessous cet ensemble est dans le registre-prédicat *reg*.

fonction *pre*(*reg*)

1. *solution*$\leftarrow$ *new_pred*()

2. *valeur*(*solution*)$\leftarrow$ *faux*

3. $\forall\ 1 \leq e \leq$ *Nombre_Etats*
   Si $e \in$ *valeur*(*reg*) alors

(a) $\forall$ *premier_arc*$(e) \leq j \leq$ *dernier_arc*(e)
valeur(*solution*) $\leftarrow$ *valeur*(*solution*) $\cup$ *Depart*(j)

4. Retourner *solution*

5. Fin

Les formules $\widetilde{pre}(f)$

$\widetilde{pre}(f)$ est vrai dans tous les états à partir desquels tous les arcs mènent à des états contenus dans l'ensemble caractéristique de f. Dans l'algorithme montré ci-dessous, cet ensemble est dans le registre-prédicat *reg*.

fonction $\widetilde{pre}$(*reg*)

1. *solution*$\leftarrow$ *new_pred*()

2. *valeur*(*solution*) $\leftarrow$ *vrai*

3. $\forall$ $1 \leq e \leq$ *Nombre_Etats*
Si $e \notin$ *valeur*(*solution*) alors

   (a) $\forall$*premier_arc*$(e) \leq j \leq$ *dernier_arc*(e)
   valeur(*solution*) $\leftarrow$ *valeur*(*solution*) $-$ {*Depart*(j)}

4. Retourner *solution*

5. Fin

Les deux fonctions décrites ci-dessus, visitent, dans le cas le plus défavorable, une fois chaque état et une fois chaque arc du graphe. Leur complexité peut être bornée par $O($*Nombre_Etats* + *Nombre_Arcs*$)$.

L'algorithme de l'opérateur *not* n'est pas décrit. Son évaluation est faite en utilisant des opérations ensemblistes classiques.

6.8.4 Les formules $<B>f$ et $B.f$.

La formule $<B>f$.

Les formules $<B>f$ est vraie dans tous les états ayant au moins un arc qui porte une étiquette contenue dans B et qui mène à un état dans l'ensemble caractéristique de f. L'algorithme montré ci-dessous reçoit comme paramètres un ensemble d'actions *ens* et un ensemble caractéristique de formules contenu dans le registre-prédicat *reg*.

fonction *los*(*ens*,*reg*)

1. *solution*← *new_pred*()

2. *valeur*(*solution*) ← *faux*

3. $\forall\ 1 \leq e \leq$ *Nombre_Etats*
   Si $e \in$ *valeur*(*reg*) alors

   (a) $\forall$ *premier_arc*(*e*) $\leq j \leq$ *dernier_arc*(*e*)
   Si *ens* $\cap$ *actions*(*Trans*(*j*)) $\neq \emptyset$ alors
   solution← *solution* $\cup$ {*Depart*(*j*)}

4. Retourne *solution*.

5. Fin.

La formule *B.f*.

La formule *B.f* est vraie dans tous les états qui ne sont pas des états puits et dont tous les arcs portent des actions contenues dans *B* et mènent à des états dans l'ensemble caractéristique de *f*. L'algorithme montré ci-dessous, reçoit comme paramètres un ensemble d'actions *ens* et un registre-prédicat contenant un ensemble caractéristique de formules.

fonction *prefixage*(*ens*,*reg*)

1. *solution*← *new_pred*()

2. *valeur*(*solution*) ← *vrai*

3. *valeur*(*solution*)← *valeur*(*solution*) − *valeur*(*sink*)
   /* Les états puits ne satisfont pas cet opérateur.*/

4. $\forall\ 1 \leq e \leq$ *Nombre_Etats*

   (a) $\forall$ *premier_arc*(*e*) $\leq j \leq$ *dernier_arc*(*e*)
   Si ($e \notin$ *valeur*(*reg*)) et (*ens* $\cap$ *actions*(*Trans*(*j*)) $= \emptyset$) alors
   valeur(*solution*) ← *valeur*(*solution*) − {*Depart*(*j*)}

5. Retourner *solution*.

6. Fin

Les deux fonctions décrites ci-dessus visitent, dans le cas le plus défavorable, une fois chaque état et une fois chaque arc du graphe. Leur complexité peut être bornée par $O(Nombre\_Etats + Nombre\_Arcs)$.

6.8.5 Les formules $pot[f_1]f_2$, $al[f_1]f_2$ et $inev[f_1]f_2$.

Les formules $pot[f_1]f_2$

La fonction qui évalue la formule $pot[f_1]f_2$, *pot(reg1, reg2)*, suppose l'ensemble caractéristique de la condition f_1 stocké dans le registre *reg1* et l'ensemble caractéristique de f_2 dans le registre *reg2*. Elle rend un ensemble caractéristique qui contient les états de *valeur(reg2)* et les états à partir desquels il existe des chemins d'exécution dont les composants appartiennent à *valeur(reg1)* jusqu'à ce que l'on atteigne un état dans *valeur(reg2)*. L'ensemble caractéristique rendu par cette fonction peut être calculé par l'application exhaustive des règles suivantes :

$$\vdash valeur(solution) \leftarrow reg2$$

$$\frac{q \in Q, \exists q' \in Q, q \rightarrow q' \text{ et } q' \in valeur(solution)}{valeur(solution) \leftarrow valeur(solution) \cup \{q\}}$$

L'algorithme qui traduit ces règles est :

fonction *pot(reg1,reg2)*

1. *solution*← *new_pred*()
2. *valeur(solution)*← *valeur(reg2)*
3. $\forall\ 1 \leq e \leq$ *Nombre_Etats*
   Si $e \in$ *valeur(reg2)* alors
   (a) *init_pile*()
   (b) *empiler(e)*
   (c) *prof_pot(reg1,solution)*
4. Retourner *solution*.
5. Fin.

Procédure *prof_pot(reg1,solution)*

1. **tantque** $\neg$ *pile_vide*() **faire**
   (a) $e \leftarrow$ *desempiler*()
   (b) $\forall premier\_arc(e) \leq j \leq dernier\_arc(e)$
   Si *Depart(j)* $\notin$ *valeur(solution)* et *Depart(j)* $\in$ *valeur(reg1)* alors
   i. *solution*← *solution* $\cup$ *Depart(j)*

ii. *empiler*(*Depart*(*j*))

2. Fin

La fonction *pot* utilise une variante du parcours en profondeur des graphes. Ce genre de parcours a une complexité $O(Nombre\_Etats + Nombre\_Arcs)$.

Les formules $inev[f_1]f_2$.

La fonction qui évalue les formules $inev[f_1]f_2$ suppose l'ensemble caractéristique de la condition f_1 stocké dans le registre *reg1* et l'ensemble caractéristique de f_2 dans le registre *reg2*. Elle rend un ensemble caractéristique contenant tous les états qui satisfont *reg2* ou tous les états tels que sur tous ses chemins d'exécution les états appartiennent à *valeur*(*reg1*) jusqu'à ce que l'on atteigne un état de *valeur*(*reg2*).

inev(*reg1*,*reg2*) peut être évalué par l'application exhaustive des règles suivantes :

$$\vdash valeur(solution) \leftarrow valeur(reg2)$$

$$\frac{q \in Q, \forall q' \in Q, q \rightarrow q', q' \in valeur(solution)}{solution \leftarrow solution \cup \{q'\}}$$

L'algorithme dérivé de ces règles utilise un tableau appelé *succ* qui contient, pour chaque état, le nombre d'états qu'il peut atteindre par l'exécution d'une transition. Le corps de l'algorithme est :

fonction *inev*(*reg1*,*reg2*)

1. *solution*← *new_pred*()
2. *valeur*(*solution*)← *valeur*(*reg2*)
3. $\forall 1 \leq e \leq Nombre\_Etats$
   (a) Si $e \in valeur(reg2)$ alors
      i. *init_pile*()
      ii. *empiler*(*e*)
      iii. *prof_inv*(*reg2*,*solution*)
4. Retourner *solution*.
5. Fin

Procédure *prof_inv*(*reg2*,*solution*)

1. **tantque** $\neg$*pile_vide*() **faire**
   (a) $e \leftarrow$ *desempiler*()
   (b) $\forall$*premier_arc*(e) $\leq j \leq$ *dernier_arc*(e)
      i. Si (*Depart*(j) $\notin$ *valeur*(*solution*))
         $\wedge$ (*Depart*(j) $\in$ *valeur*(*reg2*))) alors
         succ(*Depart*(j)) $\leftarrow$ *succ*(*Depart*(j) $-$ 1)
      ii. Si *succ*(*Depart*(e)) = 0 alors
         A. *valeur*(*solution*)$\leftarrow$ *valeur*(*solution*) $\cup$ {*Depart*(e)}
         B. *empiler*(e)
2. Fin.

La fonction *inev* utilise une variante du parcours en profondeur des graphes. Ce genre de parcours a une complexité O(*Nombre_Etats* + *Nombre_Arcs*). Cet ordre ne peut être atteint que par l'utilisation du tableau additionnel *succ*. Si o est le nombre d'octets nécessaires pour représenter le numéro d'un état, l'espace occupé par ce tableau est *Nombre_Etats* $\times$ o.

Les formules $al[f_1]f_2$ sont évaluées en utilisant le fait que les opérateurs *al* et *pot* sont duaux.

Les algorithmes des opérateurs binaires $\wedge, \vee, \supset$ et $\equiv$ sont des implantations directes des opérations ensemblistes classiques.

6.8.6 Les formules $B_1.f_1 + \ldots + B_n.f_n$.

Dans le chapitre 2 on a défini l'opérateur binaire +. Dans [GS86] il est montré que cet opérateur est associatif. En XESAR on utilise ce résultat pour permettre à l'utilisateur l'expression de formules comportant n termes de la forme $B_i.f_i$. Si *liste_ens* est une liste de n ensembles d'actions et *liste_reg* est une liste de n numéros de registres, *eval_plus*(*liste_ens*,*liste_reg*) est vrai dans tous les états e qui satisfont les deux conditions suivantes :

1. A partir de e il y a des arcs qui portent des actions en ens_i et qui mènent à des états de reg_i, pour tout ensemble ens_i et tout registre reg_i.
2. Tous les arcs qui sortent de e portent des actions de l'un des ens_i et mènent à des états contenus dans *valeur*(reg_1)

L'algorithme qui évalue ces formules est :

fonction *plus*(*liste_ens*,*liste_reg*)

1. *solution*$\leftarrow$ *new_pred*()

2. *valeur*(*solution*)← *vrai*

3. $\forall\ 1 \leq i \leq n$
   valeur(*solution*)← *valeur*(*solution*) $\cap$ *valeur*(*los*(ens_i, reg_1))
   /**solution* contient tous les états qui satisfont la première condition*/

4. $\forall 1 \leq e \leq$ *Nombre_Etats*

   (a) $\forall$ *premier_arc*(*e*) $\leq j \leq$ *dernier_arc*(*e*)
   Si *Depart*(*e*) $\in$ *valeur*(*solution*) alors
   Si $\nexists i, (e \in reg_i)$ et (*actions*(*Trans*(*j*)) $\cap$ $ens_i \neq \emptyset$) alors
   valeur(*solution*) ← *valeur*(*solution*) $- \{e\}$

5. Retourner *solution*.

6. Fin

6.8.7 L'évaluation des formules de point fixe.

L'évaluation des formules qui comportent des opérateurs de point fixe implique plusieurs passes sur la représentation interne de la formule.

Le premier algorithme que l'on présente est basé sur cette variante du théorème de Knaster-Tarski :

Théorème 1 *Soit* $F : 2^Q \rightarrow 2^Q$ *une fonctionnelle monotone. Alors :*

- *Si F est* $\cup$*-continu alors* $\mu x.F(x) = \cup_i F^i(\text{faux})$
- *Si F est* $\cap$*-continu alors* $\nu x.F(x) = \cap_i F^i(\text{vrai})$

La première version de la fonction *eval_point_fixe* est :

fonction *eval_point_fixe*(*op*,*f*)

1. *solution*← *new_pred*()
2. *reg*← *new_pred*()
3. Si *op* = μ alors
   (a) *valeur*(*solution*)← *faux*
   (b) **répéter**
      i. *valeur*(*reg*)← *valeur*(*solution*)
      ii. *reg*← *eval_arbre*(*f*)
      iii. *valeur*(*reg*) ← *valeur*(*reg*) $\cup$ *valeur*(*solution*)
      jusqu'à *valeur*(*reg*) = *valeur*(*solution*)

4. Si $op = \nu$ alors

   (a) *valeur(solution)*$\leftarrow$ *vrai*

   (b) **répéter**

      i. *valeur(reg)*$\leftarrow$ *valeur(solution)*

      ii. *reg*$\leftarrow$ *eval_arbre(f)*

      iii. *valeur(reg)* $\leftarrow$ *valeur(reg)* $\cap$ *valeur(solution)*

      jusqu'à *valeur(reg)* = *valeur(solution)*

5. Retourner *reg*.

6. Fin

Cet algorithme peut être amélioré grace à la version suivante du théorème 1 :

Théorème 2 *Si* $f : 2^Q \rightarrow 2^Q$ *est* $\cup$*-continu, alors* $\mu x.f(x) = \cup_i f^i(x_0)$ *pour tout ensemble initial* $x_0 \subseteq \mu x.f(x)$. *Symétriquement, si* $f(x)$ *est* $\cap$*-continu, alors* $\nu x.f(x) = \cap_i f^i(x_0)$ *pour tout ensemble initial* $x_0 \supseteq \nu x.f(x)$.

Considérons l'exemple qui suit :

> Soit $f = \mu x_1.f_1(x_1)$ et f_1 contenant la sous-formule $\mu x_2.f_2(x_1, x_2)$. Pour évaluer f, on calcule successivement $f_1(faux), f_1^2(faux)$ jusqu'à stabilisation. Pour calculer $f_1(faux)$ il est nécessaire de calculer :
>
> $$f_2(faux, faux), f_2^2(faux, faux)$$
>
> jusqu'à stabilisation.

Dans l'exemple précédent, il faut calculer $f_1(faux), f_1^2(faux), \ldots$ et pour chaque i on calcule $\mu x_2.f_2(f_1^{i-1}(faux), faux)$.

Si l'on utilise le fait que $f_1^{i-1}(faux) \subseteq f_1^i(faux)$ et $f_2(x_1, x_2)$ est monotone en x_1, alors :

$$\mu x_2.f_2(f^{i-1}(faux), x_2) \subseteq \mu x_2.f_2(f_1^i(faux), x_2))$$

On peut donc appliquer le théorème 2 et commencer le calcul de $\mu x_2.f_2(f_1^i(faux), x_2)$ avec x_2 égal à $\mu x_2.f_2(f_1^{i-1}(faux), x_2)$.

Ci-dessous on montre un algorithme qui met en œuvre ce résultat. Il a été proposé dans [EL86]. On considère l'évaluation de la formule f.

eval_arbre(op,f')

1. Si $op = \mu$ alors
   Si la plus petite σ-sous-formule de f qui contient f' est de type ν, alors

   (a) $valeur(reg) \leftarrow faux$
   /* reg est le registre associé à x */

   (b) Pour toute sous-formule non-fermée μx_j de f' qui n'est pas contenue dans une ν-formule :
   $reg_j \leftarrow faux$

   (c) **répéter**

   i. $valeur(reg') \leftarrow valeur(reg)$

   ii. $reg' \leftarrow eval\_arbre(f')$

   iii. $valeur(reg) \leftarrow valeur(reg) \cup valeur(solution)$
   jusqu'à $valeur(reg) = valeur(reg')$

2. Si $op = \nu$ alors
   Si la plus petite σ-sous-formule de f qui contient f' est de type μ, alors

   (a) $valeur(reg) \leftarrow faux$

   (b) Pour toute sous-formule non-fermée νx_j de f' qui n'est pas contenue dans une ν-formule :
   $reg_j = faux$

   (c) **répéter**

   i. $valeur(reg') \leftarrow valeur(reg)$

   ii. $reg' \leftarrow eval\_arbre(f')$

   iii. $valeur(reg) \leftarrow valeur(reg) \cap valeur(solution)$
   jusqu'à $valeur(reg) = valeur(reg')$

   Retourner reg.

3. Fin

L'évaluation d'une formule f de point fixe a une complexité proportionnelle à $Nombre\_Arcs + Nombre\_Etats^{A(f)+1}$, où $A(f)$ est le *degré maximal d'alternances* de f. Une manière de calculer ce paramètre est :

1. On construit l'arbre syntaxique de f. On suppose f en forme normale positive (aucune variable n'est quantifiée deux fois et toute négation est appliquée sur un prédicat de base).

2. On écrit les chemins des nœuds σx aux nœuds occurrences de x en ne prenant en compte que les occurrences de σ-opérateurs et variables. Si $\sigma = \mu$, le chemin est de la forme :

$$\underbrace{\mu \ldots \mu}_{1}\ \underbrace{\nu \ldots \nu}_{2} \cdots \underbrace{\gamma \ldots \gamma}_{m}$$

ou $\gamma = \nu$ si m est pair ou $\gamma = \mu$ si m est impair.
Si $\sigma = \nu$, le chemin est de la forme :

$$\underbrace{\nu \ldots \nu}_{1}\ \underbrace{\mu \ldots \mu}_{2} \cdots \underbrace{\gamma \ldots \gamma}_{m}$$

ou $\gamma = \mu$ si m est pair ou $\gamma = \nu$ si m est impair.

Le degré d'alternance de ce chemin est m. Le degré maximal d'alternances du graphe est le maximun des degrés d'alternance des chemins du graphe.

La preuve de ce résultat est donnée dans [EL86].

6.8.8 L'évaluation de systèmes d'équations.

A partir d'un système d'équations :

$$\Gamma = \{(x_1, \mathtt{signe}_i, f_1(x_1, \ldots, x_m)), \ldots, (x_m, \mathtt{signe}_i, p_m, f_m(x_1, \ldots, x_m))\}$$

on peut définir une fonctionnelle F :

$$F(\vec{x}) = (f_1(\vec{x}), \ldots, f_m(\vec{x})), \text{ où } \vec{x} = (x_1, \ldots, x_m)$$

La solution de ce système est calculée, d'après le théorème de Knaster-Tarski comme :

$$\lim_{n \to \infty} (\sigma_1^n, \ldots, \sigma_m^n)$$

où :

$$\sigma_i^n = \begin{cases} \cup_n f_i^n(\vec{x}) \text{ , si } \mathtt{signe}_i = => \\ \cap_n f_i^n(\vec{x}) \text{ , si } \mathtt{signe}_i = <= \end{cases}$$

Si l'on implante directement ce resultat, on calcule chaque σ_i^n comme :

$$\sigma_i^n = f_i(\sigma_1^{n-1}, \ldots, \sigma_m^{n-1}) \text{ , pour } n > 0$$

$$\sigma_i^0 = \begin{cases} faux \text{ , si } \mathtt{signe}_i = => \\ vrai \text{ , si } \mathtt{signe}_i = <= \end{cases}$$

L'algorithme que l'on montre ci-dessous est une amélioration évidente de cette méthode : chaque σ_i^n est calculé par l'équation

$$\sigma_i^n = f_i(\sigma_1^n, \ldots, \sigma_{i-1}^n, \sigma_i^{n-1}, \ldots, \sigma_m^{n-1}) \text{ pour } n > 0$$

L'algorithme présenté a deux arguments : la liste de variables du système et la liste d'équations. On nomme x_i et f_i les i-èmes composantes des listes de variables et de formules, respectivement.

fonction *eval_systeme*(*l_var*,*l_form*)

1. $\forall 1 \leq i \leq n$
   Si $type(f_i) = \mu$ alors $valeur(reg_i) \leftarrow faux$
   Si $type(f_i) = \nu$ alors $valeur(reg_i) \leftarrow vrai$
   /* reg_i est le registre associé à la variable i */
2. $reg' \leftarrow new\_pred()$
3. **répéter**
   (a) *changements*← **faux**
   (b) $\forall 1 \leq i \leq n$
      i. $reg' \leftarrow eval\_arbre(f_i)$
      ii. Si $valeur(reg') \neq valeur(reg_i)$ alors
         A. *changements*← **vrai**
         B. Si $type(f_i) = \mu$ alors
            $valeur(reg_i) \leftarrow valeur(reg_i) \cup valeur(reg')$
         C. Si $type(f_i) = \nu$ alors
            $valeur(reg_i) \leftarrow valeur(reg_i) \cap valeur(reg')$

   jusqu'à *changements*=**faux**
4. Fin.

Dans le cas le plus défavorable, *eval_système* a une complexité proportionnelle à $Nombre\_Etats * \Sigma_{i=1}^{m} C(f_i)$, où m est le nombre d'équations et $C(f_i)$ est la complexité d'évaluation de la i-ème formule. Dans le cas général $C(f_i)$ est borné par $Nombre\_Arcs + Nombre\_Etats^{A(f_i)+1}$, $A(f_i)$ le degré maximal d'alternance de f_i. Pour les systèmes d'équations qui ne comportent pas d'opérateur de point fixe, en particulier les systèmes des graphes de sûreté, $C(f_i)$ est borné par $Nombre\_Etats + Nombre\_Arcs$.

Chapitre 7

Utilisation de Xesar.

Ce chapitre contient une description sommaire de l'utilisation de l'outil XESAR. Une description plus complète est faite dans le document [RRSV87b].

XESAR interagit avec ses utilisateurs à travers deux menus. Le premier, appelé *menu principal* contient les options qui exécutent les phases de XESAR. Le second, appelé *menu d'évaluation*, est activé lorsque l'utilisateur désire évaluer les formules de sa spécification sur le graphe d'états. L'annexe E contient une description de ces menus. Dans ce chapitre on montre la vérification de quelques propriétés de la description du protocole de Stenning contenue dans l'annexe B.

Validation du protocole de Stenning.

Pour valider ce protocole, on génère d'abord le graphe d'états. Ceci est fait en utilisant l'option adéquate du menu principal. Ensuite, on entre dans la phase d'évaluation en appelant le menu d'évaluation. L'option 11 de ce menu commence une session d'évaluation. Les formules peuvent être proposées à l'évaluateur après le prompt **Formula>**.

Si l'on veut vérifier si la description *se bloque*, c'est-à-dire si dans le graphe il existe des états à partir desquelles aucune évolution n'est possible, on propose à l'évaluateur la formule *sink* :

Formula> *sink*

et l'on obtient la réponse :

Formula always false

Ce dernier message signifie qu'aucun état ne satisfait la formule et en conséquence, que le graphe ne comporte pas d'état puits.

Considérons la séquence de *définitions* de formules suivante :

enin0 == *enable(t.mess_in0)*
afin0 == *after(t.mess_in0)*
afout0 == *after(t.mess_out0)*

Dans la section 6.4.5 on a décrit le double effet de ces définitions : d'abord, l'évaluateur évalue la formule à droite du symbole == et ensuite il crée une entrée dans la table de symboles où est gardé l'ensemble caractéristique de cette formule sous le nom spécifié.

Si l'on propose à l'évaluateur ces définitions, on obtient :

Formula> *enin0* == *enable(t.mess_in0)*
Invalid formula. Formula false for 8896 states out of 9144

Formula> *afin0* == *after(t.mess_in0)*
Invalid formula. Formula false for 9078 states out of 9338

Formula> *afout0* == *after(t.mess_out0)*
Invalid formula. Formula false for 9312 states out of 9520

Les messages émis par l'évaluateur comportent deux parties. La première affirme que la formule n'est pas une formule valide (une formule est valide si elle est satisfaite par tous les états du graphe). La deuxième montre le nombre d'états du graphe dans lequels la formule est false. On remarque qu'après l'évaluation des formules comportant l'opérateur *after*, le nombre d'états du graphe augmente.

Dans la formule :

afin0 => **not** *pot*[**not** *afout1*] (*enin0* **and not** *afout0*)

on utilise les définitions faites précédemment. Cette formule a été proposée dans la section 2.4.3 pour décrire la propriété suivante du protocole de Stenning :

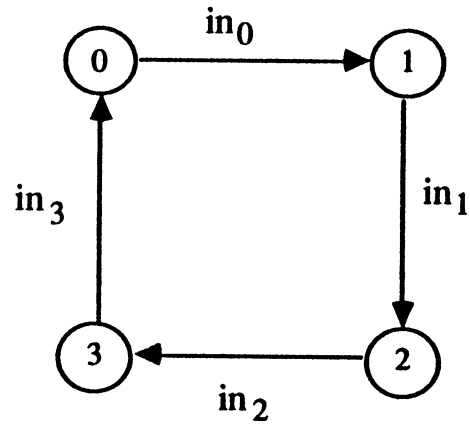

Figure 7.1:

si un message étiqueté par 0 est transmis, il n'est pas possible de transmettre un autre message 0 à moins qu'un message 0 n'ait été délivré.

L'évaluation de cette formule produit le message :

Valid formula

signifiant que la description du protocole satisfait la propriété.

Considérons maintenant l'évaluation de propriétés exprimées en utilisant des graphes. Dans la version actuelle de l'évaluateur, déterminer si un graphe d'implantation *satisfait* les propriétés exprimées par un graphe de sûreté se fait en évaluant sur le graphe d'implantation le système d'équations dérivé du graphe de sûreté. Les systèmes qui correspondent aux spécifications externes données dans la section 3.8.1 sont :

1. Le graphe de la figure 7.1 spécifie que les acceptations de messages se font en respectant les numéros de séquence. On lui associe le système suivant :

   etati== **var :** x_0, x_1, x_2, x_3
   x_0 => **not** < $t.mess\_in_0$ > **not** x_1
   $\wedge$ **not** < $t.mess\_in_1, t.mess\_in_2, t.mess\_in_3$ > **true**
   $\wedge$ **not** < $r.mess\_out_3, r.mess\_out_2, r.mess\_out_1, r.mess\_out_0$,
   t_to_r.min.message,*r_to_t.min.message*,

$t.mt.message, r.mr.message, \mathtt{tau}, \mathtt{chi} > \mathbf{not}\ x_0$

$x_1 \Rightarrow \mathbf{not} < t.mess\_in_1 > \mathbf{not}\ x_2$
$\wedge\ \mathbf{not} < t.mess\_in_0, t.mess\_in_2, t.mess\_in_3 > \mathbf{true}$
$\wedge\ \mathbf{not} < r.mess\_out_3, r.mess\_out_2, r.mess\_out_1, r.mess\_out_0,$
$t\_to\_r.min.message, r\_to\_t.min.message,$
$t.mt.message, r.mr.message, \mathtt{tau}, \mathtt{chi} > \mathbf{not}\ x_1$

$x_2 \Rightarrow \mathbf{not} < t.mess\_in_2 > \mathbf{not}\ x_3$
$\wedge\ \mathbf{not} < t.mess\_in_0, t.mess\_in_1, t.mess\_in_3 > \mathbf{true}$
$\wedge\ \mathbf{not} < r.mess\_out_3, r.mess\_out_2, r.mess\_out_1, r.mess\_out_0,$
$t\_to\_r.min.message, r\_to\_t.min.message,$
$t.mt.message, r.mr.message, \mathtt{tau}, \mathtt{chi} > \mathbf{not}\ x_2$

$x_3 \Rightarrow \mathbf{not} < t.mess\_in_3 > \mathbf{not}\ x_0$
$\wedge\ \mathbf{not} < t.mess\_in_0, t.mess\_in_1, t.mess\_in_2 > \mathbf{true}$
$\wedge\ \mathbf{not} < r.mess\_out_3, r.mess\_out_2, r.mess\_out_1, .mess\_out_0,$
$t\_to\_r.min.message, r\_to\_t.min.message,$
$t.mt.message, r.mr.message, \mathtt{tau}, \mathtt{chi} > \mathbf{not}\ x_3$

L'évaluateur calcule la solution de ce système et garde sous le nom *etati* les états qui appartiennent à l'ensemble représenté par la variable x_0. Cette variable est supposée l'état initial du graphe parce qu'elle occupe la première place dans la liste de variables. Pour déterminer si le graphe produit à partir de la description ESTELLE/R satisfait le graphe de sûreté, on évalue la formule :

$init \Rightarrow etati$

et l'on obtient la reponse :

Valid formula

2. Le graphe de la figure 7.2 exprime le fait que la réception de messages se fait en respectant les numéros de séquence. On lui associe le système suivant :

etato== **var** : x_0, x_1, x_2, x_3
$x_0 \Rightarrow \mathbf{not} < r.mess\_out_0 > \mathbf{not}\ x_1$
$\wedge\ \mathbf{not} < r.mess\_out_1, r.mess\_out_2, r.mess\_out_3 > \mathbf{true}$
$\wedge\ \mathbf{not} < t.mess\_in_3, t.mess\_in_2, t.mess\_in_1, t.mess\_in_0,$
$t\_to\_r.min.message, r\_to\_t.min.message,$
$t.mt.message, r.mr.message, \mathtt{tau}, \mathtt{chi} > \mathbf{not}\ x_0$

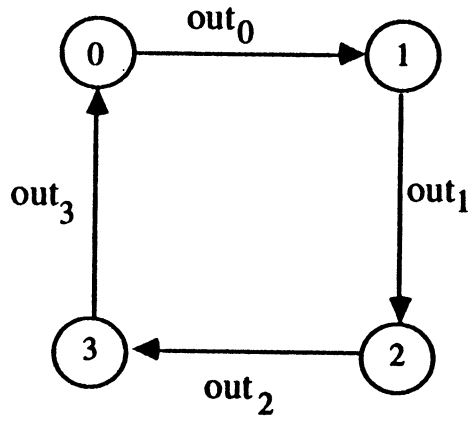

Figure 7.2:

x_1 => (not < *r.mess_out*$_1$ > not x_2
∧ not < *r.mess_out*$_0$, *r.mess_out*$_2$, *r.mess_out*$_3$ > true
∧ not < *t.mess_in*$_3$, *t.mess_in*$_2$, *t.mess_in*$_1$, *t.mess_in*$_0$,
t_to_r.min.message,*r_to_t.min.message*,
t.mt.message,*r.mr.message*,tau,chi> not x_1

x_2 => not < *r.mess_out*$_2$ > not x_3
∧ not < *r.mess_out*$_0$, *r.mess_out*$_1$, *r.mess_out*$_3$ > true
∧ not < *t.mess_in*$_3$, *t.mess_in*$_2$, *t.mess_in*$_1$, *t.mess_in*$_0$,
t_to_r.min.message,*r_to_t.min.message*,
t.mt.message,*r.mr.message*,tau,chi> not x_2

x_3 => not < *r.mess_out*$_3$ > not x_0
∧ not < *r.mess_out*$_0$, *r.mess_out*$_1$, *r.mess_out*$_2$ > true
∧ not < *t.mess_in*$_3$, *t.mess_in*$_2$, *t.mess_in*$_1$, *t.mess_in*$_0$,
t_to_r.min.message,*r_to_t.min.message*,
t.mt.message,*r.mr.message*,tau,chi> not x_3

L'évaluateur calcule la solution de ce système et garde sous le nom *etato* les états qui appartiennent à l'ensemble représenté par la variable x_0. Pour déterminer si le graphe satisfait les propriétés exprimées par le système, on évalue la formule :

init => *etato*

et l'on obtient la réponse indiquant que cette formule est valide.

L'évaluation des systèmes associés aux graphes présentés dans la section 3.8.1 nous permet d'assurer que la description satisfait la spécification donnée.

Chapitre 8

Conclusions.

Le principal objectif de ce travail était la proposition d'un langage de spécification pour XESAR, hautement expressif et facile à utiliser. La logique STL, base du langage que l'on propose, satisfait cette exigence. Le langage proposé permet aussi l'expression d'ensembles de propriétés de sûreté par des graphes d'états auxquels on a donné une sémantique particulière. Par utilisation de ces graphes, on peut simplifier considérablement la spécification de protocoles de communication, principal domaine d'utilisation de XESAR. Nous avons proposé, en donnant la spécification du service du protocole de Stenning, une famille de propriétés de sûreté qui doivent être satisfaites par tout protocole de communication.

Pour décrire les graphes de sûreté, les améliorations suivantes pourraient être envisagées au niveau de l'interface utilisateur :

- Définition d'une syntaxe pour exprimer les graphes de sûreté comme une liste de transitions, chaque transition étant définie par les noms de deux états et un ensemble d'actions. A partir de ces informations, et en considérant comme visible l'ensemble des actions apparaissant dans les transitions, l'évaluateur peut produire le système d'équations associé au graphe.

- Possibilité de représentation graphique. Pour cela on pourra se servir d'une bibliothèque de fonctions utilisée en XESAR pour visualiser les architectures des systèmes décrits en ESTELLE/R.

Les perfomances de l'évaluateur sont bonnes : l'évaluation de l'ensemble de formules montré dans le chapitre 2 est effectuée en moins d'une minute.

La mise en œuvre de la phase de génération des graphes d'états résout un des problèmes majeurs : la diminution du nombre de comparaisons nécessaires pour déterminer si un état appartient déjà à l'ensemble des états du graphe.

Néanmoins, il existe des limitations dues à l'espace mémoire nécessaire pour stocker les états. Dans la version actuelle, les limites sont de l'ordre de 60000 états et 300000 arcs. Ces chiffres sont très en dessous des besoins pour vérifier des protocoles utilisés dans des applications réelles.

Toutefois, ces limitations sont liées à des choix d'implantation pris par des restrictions existantes au début de l'implantation de XESAR. Cet outil a été conçu pour fonctinner sur une SM90 avec un maximun de 4 Mega octets de mémoire. Avec ces paramètres, nos objectifs initiaux visaient des graphes ayant un maximun de 20000 états et 100000 arcs et des temps de réponse acceptables pour un système interactif. Sous ces conditions, on a pris la décision de garder l'ensemble de valeurs des états du graphe en mémoire, ce qui limite le nombre d'états maximaux que peut avoir un graphe, mais qui a permis la génération des graphes dans des temps raissonables. Dans de machines plus performantes et si l'on considère la possibilité de stocker les états sur disque, il est certain que ces limites peuvent être dépassées sans introduire des grands changements.

Des améliorations qui peuvent être prises en compte pour faire des économies d'espace sont :

- Par rapport à la représentation des états.
  - En général, les variables référençables depuis le langage de spécification n'ont de sens que si l'automate composé se trouve dans certains états. On pourrait donc ne garder la valeur d'une variable que si le marquage contient un état de l'automate où cette variable est significative. Dans ce cas, on aura une représentation de l'état de longueur variable.
  - L'utilisateur de XESAR peut, potentiellement, inclure dans sa spécification des références à toutes les variables définies au niveau principal des tâches et toutes les variables associées aux échanges. Il est évident que l'on pourrait éliminer les variables non-référencées dans la spécification et de cette manière diminuer la taille du vecteur de valeurs d'un état.
- Par rapport au graphe d'états.
  - Les automates générés par les phases de génération et composition des automates ne sont pas optimaux. Il est possible d'établir des liens avec des outils comme ALDÉBARAN ([Fer88]) pour faire des réductions des ces automates modulo la bisimulation forte ou l'équivalence observationelle ([Mil80]). Dans ce cas, pour le calcul itératif des bisimulations, au lieu de partir de la relation universelle, il faut predre la partition induite sur les états par la relation de satisfaction pour les prédicats de base, c'est-à-dire

deux états sont 0-équivalents si et seulement si, ils satisfont les mêmes prédicats de base.

- Des études ont été faites pour faire la génération du graphe selon le type de propriétés à vérifier. Jusqu'à présent, on s'est limité aux formules du type $pot[f]g$ et $inev[f]g$. Les résultats montrent que dans la plupart des cas, il est possible d'évaluer ces formules sans garder tout le graphe. La méthode utilisée consiste à transformer l'ensemble des états du graphe en un ensemble de macro-états tels que dans tous les éléments d'un de ces macro-états la formule est vraie ou fausse. Les études se dirigent vers une généralisation de ce procédé pour traiter des formules plus complexes.

Références Bibliographiques

[AB84] D. Austry et G. Boudol. Algèbre de processus et synchronisation. *TCS*, 30, 1984.

[Abr79] K. Abrahamson. Modal logic of concurrent non deterministic programs. Dans *Semantics of Concurrent Computation*, 1979. LNCS, 70.

[AF85] B. Alpern et F.B.Schneider. *Verifying Temporal Properties without using Temporal Logic.* Rapport de Recherche TR 85-723, Cornell University, 1985.

[AMP83] M. Ben Ari, Z. Manna, et A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20, 1983.

[BHR84] S. D. Brookes, C.A.R Hoare, et A.W. Roscoe. Theory of communicating sequential processes. *JACM*, 31(3), 1984.

[BK84] J.A. Bersgstra et J.W. Klop. *CA Complete Inference System for Regular Processes.* Rapport de Recherche CS-R8420, Centre for Mathematics and Computer Science. Department of Computer Science, 1984.

[CE85] E.M. Clarke et E.A. Emerson. Linear and branching time structures in the semantics of reactive structures. Dans *LNCS 194*, 1985. 12th ICALP.

[CES83] E. M. Clarke, E. A. Emerson, et A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. Dans *10th Annual ACM Symp. On Principles of Programming Languages*, 1983.

[Dic85] Anne Dicky. *Une approche algébrique et algorithmique de l'analyse des systèmes de transitions.* PhD thesis, Université de Bordeaux I, 1985.

[EL86] E. A. Emerson et C. L. Lei. Efficient model-checking in fragments of the propositional Mu-calculus. Dans *Symposium on Logic in Computer Science*, 1986.

[Est85] *Estelle: A Formal Description Technique Based on an Extended Transition Model.* ISO, 1985. ISO/TC97/SC21.

[Fer88] J. C. Fernandez. ALDEBARAN, *Vérification de processus communicants.* Thèse, Université de Grenoble, 1988.

[Gra86] S. Graf. A complete inference system for an algebra of regular acceptance models. Dans *Mathematical Foundations of Computer Science*, 1986. LNCS, 233.

[GS86] S. Graf et J. Sifakis. A logic for the description of non deterministic programs and their properties. Dans *Information and Control (68)*, 1986.

[Hoa78] C.A.R Hoare. Communicating sequential processes. *CACM*, 21-8, 1978.

[Lam83] L. Lamport. What good is temporal logic. Dans , IFIP, North Holland, 1983.

[Mil80] R. Milner. A calculus of communication systems. Dans *LNCS 92*, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25, 1983.

[PM87] A. Pnueli et Z. Manna. Specification and verification of concurrent programs by ∀-automata. Dans *14th ACM Symposium of POPL*, 1987.

[Pnu77] A. Pnueli. The temporal logic of concurrent programs. Dans *LNCS 194*, 1977. 12th ICALP.

[QS83] J.P. Queille et J. Sifakis. Fairness and related properties in transition systems. *Acta Informatica*, 19, 1983.

[Ricre] J. L. Richier. *Conception et réalisation de* XESAR, *un outil de validation de protocoles.* PhD thesis, Université de Grenoble, à paraître.

[RRSV87a] J.L. Richier, C. Rodriguez, J. Sifakis, et J. Voiron. Verification in *Xesar* of the sliding window protocol. Dans *17th International Workshop on Protocol Specification Testing and Verification*, 1987.

[RRSV87b] J.L. Richier, C. Rodriguez, J. Sifakis, et J. Voiron. *Xesar: A Tool for Protocol Validation. User's Guide.* LGI-Imag, 1987.

[RSV86] J.L. Richier, J. Sifakis, et J. Voiron. *ATP: An Algebra for Timed Processes.* Rapport de Recherche Projet Cesar RT-1, LGI-IMAG Grenoble, 1986.

[Sch83] J. Ph. Schwartz. *Quasar, une réalisation du système Cesar.* Thèse de Doctorat-Ingénieur, Université de Grenoble, 1983.

[Sor87] Amelia Soriano. *VENUS — Un outil d'aide à la vérification des systèmes communicants.* PhD thesis, Université de Grenoble, 1987.

[Ste76] N. V. Stenning. A data transfer protocol. Dans *Computer Networks (1)*, 1976.

[Zim80] Zimmermann. Osi reference model. the iso model of architecture for open system interconnection. *IEEE Transaction in Communications*, COM-28, 1980.

Annexe A

Syntaxe concrète d'Estelle/R.

Dans cette annexe on donne la syntaxe concrète de ESTELLE/R et on énumère ses différences avec ESTELLE.

Différences.

Les pricipales différences sont :

- Pour des raisons relatives à la méthode de vérification de XESAR, en ESTELLE/R les constructions suivantes ne sont pas admises :
  - Les instructions dites de structuration : **exist, forone.**
  - Les opérations pour la gestion de *files* et de pointeurs.
  - L'utilisation des instructions **output** et **nexstate** n'est pas permise dans les procédures et fonctions.
- L'association d'étiquettes aux transitions et aux instructions **output** est possible par l'utilisation de *pragmats*. Ces étiquettes peuvent ensuite être référencées depuis le langage de spécification.
- Dans cette version de XESAR, les constructions suivantes ne sont pas admises : *goto*, *label*, *all*, *priority*, *with*.

Syntaxe concrète.

Eléments lexicaux.

La syntaxe générale des programmes respecte les normes du PASCAL standard. Les lettres majuscules et minuscules sont équivalentes.

Les tokens "{", "}", "[" et "]" peuvent être représentés par "(*", "*)", "*)", "(." et ".)", respectivement.

Les commentaires peuvent être délimités par "(*" et "*)" ou par "{" ou "}".

Les *pragmats* utilisés pour définir les étiquettes, sont de la forme :

{: *liste_de_noms* }

Conventions du méta-langage.

- *ident* est un identificateur PASCAL ("_" est accepté et les miniscules et majuscules sont équivalentes)
- *integer* et *real* sont des notations PASCAL pour des constantes entières et réelles.
- *string* est une notation PASCAL pour les chaînes de caractères.
- *const* représente le terminal const.
- { X } représente X ou l'ensemble vide.
- { X }+ représente au moins une occurrence de X.
- { X }* représente zéro ou plus occurrence de X.
- Le commentaires sont délimités par "/*" et "*/".
- Le commentaire "/* ignored */" signifie que cette construction est valide mais sans effet en XESAR.
- Le commentaire "/* forbidden */" signifie que la construction est valide pour ESTELLE mais non-admise en XESAR.

Syntaxe de ESTELLE/R.

```
                /* Un programme est derive du non-terminal program*/

program :               def_de_systeme "."

                /* Pascal */

l_id :                  ident { "," ident }*

                /* constantes */
```

```
part_decl_const :       "const"  { def_de_const ";" }+

def_de_const :          ident "=" const

const :                 { signe } const_nombre_id   |   string

const_nombre_id :       integer   |   real   |   ident

signe :                 "+"   |   "-"

              /* definitions de types*/

part_decl_type :        "type"  { def_de_type ";" }+

def_de_type :           ident "=" dscr_de_type

dscr_de_type :          { term_optionnel } ident
                        | { term_optionnel } nv_type

term_optionnel :                "optional"              /* ignored */

nv_type :               nv_type_ord  |  nv_type_structure
                        |  nv_type_pointeur

type_ord :              nv_type_ord    |   ident

nv_type_ord :           type_enumere   |   type_intervalle

type_enumere :          "(" l_id ")"

type_intervalle :               const ".." const

nv_type_structure :     { "packed" } type_structure_non_tasse

type_structure_non_tasse :      type_tableau   |   type_art
                                | type_ensemble   |   type_fichier

type_tableau :          "array" "[" type_ord { "," type_ord }* "]"
                                "of" dscr_de_type

type_art :              "record" l_de_champs "end"

l_de_champs :           { parts_fixes_var }

parts_fixes_var :               sect_art { ";" sect_art }* { ";" }
                        | { sect_art { ";" sect_art }* ";" }
                         "case"  { ident ":" } ident "of" variante
                            { ";" variante }* { ";" }
```

```
sect_art :              l_id ":" dscr_de_type

variante :              const { "," const }* ":" "(" l_de_champs ")"

type_ensemble :         "set" "of" type_ord

type_fichier :          "file" "of" dscr_de_type
                                /*  forbidden  */

nv_type_pointeur :      "^" ident       /*  forbidden  */

              /* variable definitions*/

part_decl_var :         "var"  { def_de_var ";" }+

dcl_de_var :            l_id ":" dscr_de_type

              /* references aux varibles*/

ref_a_une_var :         ident { fin_ ref_a_une_var }*

fin_ ref_a_une_var :    fin_var_indicee   |   fin_acces_champ
                        |   fin_var_pointee

fin_var_indicee :               "[" expr { "," exp }* "]"

fin_acces_champ :       "." ident

fin_var_pointee :               "^"             /*  forbidden  */

              /* Etiquettes Pascal */

part_decl_etiq :                "label" etiq { "," etiq }* ";"
                                     /*  forbidden  */

              /* Parametres actuels*/

l_de_param_effectif :   { "(" param_effectif
                             { "," param_effectif }* ")" }

param_effectif :                expr { fin_de_write }

fin_de_write :          ":" expr  { ":" expr }          /*  forbidden  */

              /* Parametres formels */

l_de_param_formel :     { "(" param_formel
                          { ";" param_formel }* ")" }
```

```
l_de_param_formel_valeur : { "(" param_valeur
                             { ";" param_valeur }* ")" }

param_formel :          param_valeur | param_var
                         |   param_proc   |   param_fonction

param_valeur :          l_id ":" ident
                        | "const" ...           /*  forbidden  */

param_var :             "var" l_id ":" ident

param_fonction :                { term_pure }  "function" ident
                                l_de_param_formel ":" ident

param_proc :            { term_pure } "procedure"
                         ident  l_de_param_formel

              /* Bloc  Pascal */

bloc :                  { part_decl_etiq } { part_decl_const }
                           { part_decl_type } { part_decl_var }
                           { part_decl_proc_fonc }
                           { nom_trans } part_enonces

nom_trans :             "name" ident ":"        /*  ignored  */

              /* Procedure and Function declarations*/
              /* Declaration de procedures et fonctions */

part_decl_proc_fonc :   { decl_proc_fonc ";" }*

decl_proc_fonc :                { term_pure } "procedure" ident
                                 l_de_param_formel ";"  dir_ou_bloc
                        | { term_pureJ} "function"  ident ";" bloc
                        | { term_pureJ} "function" ident
                         l_de_param_formel ":" ident ";" dir_ou_bloc

term_pure :             "pure"          /*  ignored  */

dir_ou_bloc :           bloc   |   "forward"
                        | "primitive"   /*  forbidden  */

              /* Instructions Pascal*/

part_enonces :          "begin" l_enonces "end"

l_enonces :             enonce { ";" enonce }*

enonce :                        { etiq } fin_enonce
```

```
etiq :                  integer ":"             /*  forbidden  */

fin_enonce :            ref_a_une_var ":=" expr
                        | ident l_de_param_effectif
                        | "goto" integer        /*  forbidden  */
                        | "begin" enonce_compose "end"
                        | "if" expr_booleenne "then" enonce
                          { "else" enonce }
                        | "case" expr "of" elm_de_l_de_cas
                             {";" elm_de_l_de_cas }*{";"} "end"
                        | "repeat" l_enonces "until" expr_booleenne
                        | "while" expr_booleenne "do" enonce
                        | "for" ident ":=" expr vers expr "do" enonce
                        | "with" ref_a_une_var {"," ref_a_une_var }*
                            "do" enonce  /* forbidden */
                        | "all" domaine_tout "do" enonce
                              /* forbidden  */
                        | "nextstate" etat
                        | { etiq_xesar } "output" interaction
                          l_de_param_effectif
                        | "init" sous_bloc "with" comportement
                        | "connect" ref_port_designe "to"
                          ref_port_designe
                        | "attach" ref_port "to" ref_port_designe
                        | "detach" ...          /*  forbidden  */
                        | "disconnect" ...      /*  forbidden  */
                        | "release" ...         /*  forbidden  */
                        | "forone" ...          /*  forbidden  */
                        | ""

elm_de_l_de_cas :       const { "," const }* ":" enonce

vers :                  "to"   |   "downto"

                /* Expressions Pascal */

expr_booleenne :        expr
                        | "exist" ...   /*  forbidden  */

expr :                  { expr_simple op_de_relation } expr_simple

expr_simple :           expr_simple op_additif terme
                        | { signe } terme

terme :                 { terme op_multiplicatif } facteur

facteur :                       ref_a_une_var
                        | integer   |   real   |   string
```

```
                        | "nil"                  /*  forbidden  */
                        | appel_de_fonction
                        | cnstr_ensemble
                        | "(" expr ")"
                        | "not" facteur
                        | "undefined"          /*  forbidden  */

cnstr_ensemble :              "[" { elem_ens { "," elem_ens }* } "]"

elem_ens :              expr { ".." expr }

appel_de_fonction :     ident l_de_param_effectif

op_additif :            "+"   |   "\-"   |   "or"

op_multiplicatif :            "*"   |   "/"   |   "div"
                               |   "mod"   |   "and"

op_de_relation :              "="   |   "<>"   |   "<"   |   ">"
                              |   "<="   |   ">="   |   "in"

              /* Extensions Estelle/R*/

def_de_systeme :              "specification" ident ";"
                               { opt_defaut } def_de_corps "end"

opt_defaut :            "default" ...          /*  forbidden  */

def_de_corps :          part_dcl part_initialisation
                            part_transition part_terminaison

part_dcl :              { dcls }*

dcls :                  part_decl_const
                        | part_decl_type
                        | "channel" def_de_type_de_canal
                        | "module" def_de_type_de_module
                        | "body" def_de_comportement
                        | part_decl_var
                        | "state" def_etat
                        | "stateset" { def_ensemble_etat ";" }+
                        | "use" ...            /*  forbidden  */
                        | decl_proc_fonc ";"

part_initialisation :   { "initialize" trans }*

part_terminaison :      "terminate" ... /*  forbidden  */
```

```
def_de_type_de_canal :   ident fin_de_def_de_type_de_canal

fin_de_def_de_type_de_canal :  "(" l_id ")" ";" { clause_par }+
                          | ";" { def_de_signal ";" }+

clause_par :              "by" l_id ":" { def_de_signal ";" }+
                          | "by" l_id ":" ";"

def_de_signal :           ident l_de_param_formel_valeur


def_de_type_de_module : ident classe { "(" l_param_de_module ")" } ";"
                                                        { var_export }

classe :                        "process"
                          | "activity"            /*  forbidden  */

l_param_de_module :       l_param_interaction
                          | {l_param_interaction ";"} "param"
                            param_valeur {";" param_valeur}*

l_param_interaction :     dcl_de_points_interaction
                          { ";" dcl_de_points_interaction }*

dcl_de_points_interaction : l_id ":" { "array" "["
                              type_ord {"," type_ord_veda }* "]"
                              "of"} ident { roles} { disc_de_com }

roles :                   "(" l_id ")"            /*  ignored  */

disc_de_com :             "common" "queue"        /*  forbidden  */
                          | "individual" "queue"  /*  forbidden  */


var_export :              "export" ...            /*  forbidden  */


def_de_comportement :     ident "for" ident ";" dir_ou_corps_de_comp

dir_ou_corps_de_comp :    "external" ";"          /*  forbidden  */
                          | def_de_corps "end" ";"


def_etat :                ":" "(" l_id ")" ";"


def_ensemble_etat :       ident "=" "[" l_id "]"

                /* transitions */
```

```
part_transition :       { trans_emboitees }*

trans_emboitees :               "trans" { etiq_xesar } trans

trans :                   { cnst_quelquesoit }+
                        | { cnst_avec }+
                        | { cnst_entree }+
                        | { cnst_delai }+
                        | { cnst_depuis }+
                        | { cnst_vers }+
                        | { cnst_pourvu_que }+
                        | { cnst_priorite }+
                        | bloc ";"

cnst_quelquesoit :      "any" dcl_id { ";" dcl_id }* "do" trans

dcl_id :                        ident ":" type_ord

cnst_avec :             "with" ref_a_une_var  { "," ref_a_une_var }*
                        "do" trans    /*  forbidden */

cnst_entree :           "when" interaction { "(" l_id")" } trans

cnst_delai :            "delay" "(" expr { "," expr_max } ")"
                         { etiq_xesar } trans

expr_max :              expr   |   "*"

cnst_depuis :           "from" etats_presents trans

etats_presents :                l_id
                         /*  liste  d'etats  ou d'ensembles d'e'tats */

cnst_vers :             "to" prochain_etats trans

prochain_etats :                etats_presents   |   "same"

cnst_pourvu_que :       "provided" expr_booleenne_ou_autre trans

expr_booleenne_ou_autre : "otherwise"   |   expr_booleenne

cnst_priorite :         "priority" expr trans   /*  forbidden  */


sous_bloc :             ref_a_une_var

ref_port :              ident { "[" expr { ";" expr }* "]" }
```

```
ref_port_designe :      sous_bloc "." ref_port

interaction :           ref_port "." ident

comportement :          ident l_de_param_effectif

etat :                  "same"   |   ident

                /* Etiquettes Xesar */

etiq_xesar :            "{:" l_id "}"
```

Annexe B

Description du protocole de Stenning.

```
specification stenning;
const
    k = 4;      (* all numbers are modulo k *)
    k_minus_1 = 3;
    M = 2;      (* Capacity of the media *)
    TWS = 2;    (* Transmitter Window Size *)
    RWS = 2;    (* Receiver Window Size *)
    TOE = 5;    (* Transmit Time Out *)

channel data;
        message(val: integer);

module transmitter process(tm: data; mt: data);

body transmission for transmitter;
    var
(* the transmitter has got (from the user) the messages numbered
                lowest_unacked .. lowest_unacked+number_entered-1 *)
(* the transmitter has sent (on the line) the messages numbered
                lowest_unacked .. lowest_unacked+number_in_transit-1 *)
        lowest_unacked: 0 .. k_minus_1;
        number_in_transit: 0 .. k_minus_1;
        actual_number: 0 .. k_minus_1;
        number_entered_message: 0 .. k_minus_1;
        in_transit: packed array[0..k_minus_1] of boolean;

(* test if i is in the interval base<= <base+width  (mod k) *)
    function in_window(i, base, width: integer): boolean;
        begin
            if (base+width) <= k then
```

```
                in_window := (base <= i) and (i < base+width)
            else
                in_window := (base <= i) or (i < base+width-k)
        end;

(* set to false in_transit[i] for low<=i<=high (mod k) *)
(*  the interval low..high (mod k) is not empty *)
    procedure cancel_timer(low, high: integer);
        var i: integer;
        begin
            if high < low then high := high + k;
            for i := low to high do in_transit[i mod k]:= false;
        end;

    initialize
        var i: 0..k_minus_1;
        begin
            number_in_transit := 0;
            number_entered_message := 0;
            lowest_unacked := 0;
            actual_number := (lowest_unacked+number_in_transit) mod k;
            for i := 0 to k_minus_1 do in_transit[i] := false;
        end;

    trans {: mess_in1}
        provided (actual_number = 1) and (number_in_transit < TWS) and
                 (number_in_transit >= number_entered_message)
        delay(1,*) begin
            output tm.message( 1 );
            in_transit[1] := true;
            number_in_transit := (number_in_transit+1)mod k;
            number_entered_message := (number_entered_message+1)mod k;
            actual_number := (lowest_unacked+number_in_transit) mod k;
        end;

    trans {: mess_in0}
        provided (actual_number = 0) and (number_in_transit < TWS) and
                 (number_in_transit >= number_entered_message)
        delay(1,*) begin
            output tm.message( 0 );
            in_transit[0] := true;
            number_in_transit := (number_in_transit+1)mod k;
            number_entered_message := (number_entered_message+1)mod k;
            actual_number := (lowest_unacked+number_in_transit) mod k;
        end;

    trans {: mess_in2}
        provided (actual_number = 2) and (number_in_transit < TWS) and
                 (number_in_transit >= number_entered_message)
```

```
        delay(1,*) begin
            output tm.message( 2 );
            in_transit[2] := true;
            number_in_transit := (number_in_transit+1)mod k;
            number_entered_message := (number_entered_message+1)mod k;
            actual_number := (lowest_unacked+number_in_transit) mod k;
        end;

    trans {: mess_in3}
        provided (actual_number = 3) and (number_in_transit < TWS) and
                 (number_in_transit >= number_entered_message)
        delay(1,*) begin
            output tm.message( 3 );
            in_transit[3] := true;
            number_in_transit := (number_in_transit+1)mod k;
            number_entered_message := (number_entered_message+1)mod k;
            actual_number := (lowest_unacked+number_in_transit) mod k;
        end;

    trans (* retransmission of a message *)
        provided (number_in_transit < TWS) and
                 (number_in_transit < number_entered_message)
        delay(1,*) begin
            output tm.message((lowest_unacked+number_in_transit) mod k);
            in_transit[(lowest_unacked+number_in_transit) mod k] := true;
            number_in_transit := (number_in_transit+1) mod k;
            actual_number := (lowest_unacked+number_in_transit) mod k;
        end;

    trans (* time out on a message *)
        any j: 0..k_minus_1 do provided in_transit[j] delay(TOE)
            begin
                cancel_timer(j, (lowest_unacked+number_in_transit-1+k) mod k);
                number_in_transit := (j - lowest_unacked +k) mod k;
                actual_number := (lowest_unacked+number_in_transit) mod k;
            end;

    trans (* acknowledgement *)
        when mt.message(j) begin
            if in_window(j, lowest_unacked, number_in_transit) then begin
                cancel_timer(lowest_unacked, j);
                number_entered_message := (number_entered_message -
                    (j + 1 - lowest_unacked) + k) mod k;
                number_in_transit := (number_in_transit -
                    (j + 1 - lowest_unacked) + k) mod k;
                lowest_unacked := (j+1) mod k;
                actual_number := (lowest_unacked+number_in_transit) mod k;
        end end;
end;
```

```
module receiver process(mr: data; rm: data);

body reception for receiver;
(* test if i is in the interval base<= <base+width (mod k) *)
    function in_window(i, base, width: integer): boolean;
        begin
            if (base+width) <= k then
                in_window := (base <= i) and (i < base+width)
            else
                in_window := (base <= i) or (i < base+width-k)
        end;

    state: (wait, deliver);

    var
        next_required: 0.. k_minus_1; (* first message not received *)
        received: packed array[0..k_minus_1] of boolean;

    initialize to wait
        var i: integer;
        begin
            for i:=0 to k_minus_1 do received[i]:= false;
            next_required:= 0;
        end;

    trans
        from wait when mr.message(i)
            provided in_window(i, next_required, RWS)
                            and not received[i]
                to deliver
                    begin
                        received[i] := true;
                    end;
            provided otherwise  (* the message is out of sequence *)
                to wait
                    begin
                        output rm.message((next_required - 1 + k) mod k)
                    end;
    trans {: mess_out0 }        (* deliver to the user *)
        from deliver provided (received[next_required]
                            and (next_required = 0)) to deliver
            begin
                received[0] := false;
                next_required := 1;
            end;
    trans {: mess_out1 }        (* deliver to the user *)
        from deliver provided (received[next_required]
                            and (next_required = 1)) to deliver
```

```
            begin
                received[1] := false;
                next_required := 2;
            end;
    trans {: mess_out2 }        (* deliver to the user *)
        from deliver provided (received[next_required]
                               and (next_required = 2)) to deliver
            begin
                received[2] := false;
                next_required := 3;
            end;
    trans {: mess_out3 }        (* deliver to the user *)
        from deliver provided (received[next_required]
                               and (next_required = 3)) to deliver
            begin
                received[3] := false;
                next_required := 0;
            end;
    trans       (* end of deliver, send an acknowledge *)
        from deliver provided not received[next_required] to wait
            begin
                output rm.message((next_required - 1 + k) mod k)
            end;
end;

module medium process(min, mout: data);
body line for medium;
    state: (wait, reset);
    var mess: packed array[1..M] of -1..k_minus_1;
    procedure shift;
        var i: integer;
        begin
            for i:= 2 to M do mess[i-1]:= mess[i]
        end;
    initialize to wait
        var i: integer;
        begin
            for i:=1 to M do mess[i]:= -1;
        end;

    trans (* receive a message *)
        from wait to wait when min.message(i) begin shift; mess[M]:= i end;
    trans (* receive a message and lose it *)
        from wait to wait when min.message(i) begin shift; mess[M]:= -1 end;
    trans (* try to send any message *)
        from wait to reset any j: 1..M do provided mess[j]<>-1 delay(1)
           begin output mout.message(mess[j]); mess[j]:= -1 end;
    trans
        from reset to wait begin end;
```

```
end;

var t: transmitter;
    r: receiver;
    t_to_r, r_to_t: medium;
initialize begin
    init t with transmission;
    init r with reception;
    init t_to_r with line;
    init r_to_t with line;
    connect t.tm to t_to_r.min;
    connect t.mt to r_to_t.mout;
    connect r.rm to r_to_t.min;
    connect r.mr to t_to_r.mout;
end;
end.
```

Annexe C

Algorithmes des procédures utilisées par la transformation after.

fonction booléenne **actions_differentes**($e, ens, EGALES$)

1. $oui \leftarrow 0$
2. $non \leftarrow 0$
   /* *oui* et *non* indiquent s'il existe des arcs arrivant à *e* portant (ou ne portant pas) des actions de l'ensemble *ens**/
3. $\forall\ premier\_arc(e) \leq k \leq dernier\_arc(e)$
   (a) Si $actions(Trans(k)) \cap ens = \emptyset$ alors
   $non \leftarrow non + 1$
   Sinon
   $oui \leftarrow oui + 1$
   (b) Si $(non > 0)$ et $(oui > 0)$ alors
   Retourne *vrai*
4. $EGALES \leftarrow non = 0$
5. Retourne *faux*
6. Fin

procédure deplacer_effacer($ptarc, e, ne, ens, trous$)

1. $darc \leftarrow ptarc$
   /* *darc* garde le numéro de la case où l'on place le premier arc qui arrive à *e* */

2. $premier\_arc(ne) \leftarrow Nombre\_Arcs$

3. $\forall premier\_arc(e) \leq j \leq dernier\_arc(e)$
   Si $ens \cap actions(Trans(j)) = \emptyset$ alors

   (a) $narc \leftarrow$ **nouvel_arc()**
   /***nouvel_arc** alloue une nouvelle case dans le tableau des arcs et rend son numéro*/

   (b) $Trans(narc) \leftarrow Trans(j)$

   (c) $Depart(narc) \leftarrow Depart(j)$

   (d) $trous \leftarrow trous + 1$

   Sinon
   /* On déplace l'arc pour préserver la partie *haute* de la première zone sans case vide */

   (a) $Trans(ptarc) \leftarrow Trans(j)$

   (b) $Depart(ptarc) \leftarrow Depart(j)$

   (c) $ptarc \leftarrow ptarc + 1$

4. $premier\_arc(e) \leftarrow darc$

5. Fin

procédure deplace$(ptarc, e)$

1. $arc\_act \leftarrow premier\_arc(e)$

2. Si $ptarc = arc\_act$ alors
   /* Il n'y a pas de case vide en dessus du premier arc qui arrive à e */

   (a) $ptarc \leftarrow dernier\_arc(e) + 1$

   (b) Retourner

3. $premier\_arc(e) \leftarrow ptarc$

4. **tantque** $arc\_act \leq dernier\_arc(e)$ **faire**

   (a) $Trans(ptarc) \leftarrow Trans(arc\_act)$

   (b) $Depart(ptarc) \leftarrow Depart(arc\_act)$

   (c) $ptarc \leftarrow ptarc + 1$

   (d) $arc\_act \leftarrow arc\_act + 1$

5. Fin

procédure grow_down (*taille*,*premier_etat*)

1. *ptarc1* ← *Nombre_Arcs*
   /* *ptarc1* pointe vers le prochain arc à traiter */
2. **alloc_arcs** (*taille*)
   /***alloc_arcs** alloue un bloc de *taille* nouveaux arcs. La valeur de la variable *Nombre_Arcs* est modifiée */
3. *ptarc2* ← *Nombre_Arcs*
   /* *ptarc2* pointe la prochaine case à occuper */
4. *ptetat* ← *Nombre_Etats*
   /* *ptetat* garde le numéro de l'état d'arrivée de l'arc pointé par *ptarc1* */
5. **tantque** $e \geq$ *premier_etat* **faire**
   (a) *Trans*(*ptarc2*) ← *Trans*(*ptarc1*)
   (b) *Depart*(*ptarc2*) ← *Depart*(*ptarc1*)
   (c) Si *pre_image*(*Depart*(*ptarc1*)) $\neq$) 0 alors
   /* *Depart*(*ptarc1*) est un nouvel état */
      i. *ptarc1* ← *ptarc1* − 1
      ii. *Trans*(*ptarc2*) ← *Trans*(*ptarc1*)
      iii. *Depart*(*ptarcr2*) ← *pre_image*(*Depart*(*ptarc1*))
      iv. *taille* ← *taille* − 1
      v. Si *taille* = 0 alors
      Retourner
      /* On a épuisé les nouvelles cases */
   (d) *ptarc1* ← *ptarc1* − 1
   (e) *ptarc2* ← *ptarc2* − 1
6. *premier_arc*(*e*) ← *ptarc1* + 1
7. Fin.

procédure grow_up(*decalage*,*debut*)

1. *ptarc2* ← *premier_arc*(*debut*) − *decalage*
   /**ptarc2* pointe vers la prochaine case à occuper */
2. $\forall$ *debut* $\leq e \leq$ *Nombre_Etats*
   (a) *ptarc1*← *premier_arc*(*e*)
   (b) *premier_arc*(*e*) ← *ptarc2*
   (c) **tantque** *ptarc1* $\leq$ *dernier_arc*(*e*) **faire**

i. *Trans(ptarc2)*← *Trans(ptarc1)*
ii. Si *pre_image(Depart(prarc1))* ≠ 0 alors
A. *ptarc2* ← *ptarc2* + 1
B. *Trans(ptarc2)* ← *Trans(ptarc1)*
C. *Depart(ptarc2)* ← *pre_image(Depart(ptarc1))*
D. Si *ptarc1* = *ptarc2* alors
Retourner
iii. *ptarc1* ← *ptarc1* + 1
iv. *ptarc2* ← *ptarc2* + 1

3. *Nombre_Arcs*← *ptarc1* − 1

4. Fin

procédure grow_first_zone(*netat*,*quantite*,*dernier_arc*)

1. *ptarc2*← *premier_arc*(*netat*) − 1
/* *ptarc2* pointe vers la prochaine case à occuper */

2. *ptarc1*← *dernier_arc*
/* *ptarc1* pointe vers le prochain arc à traiter */

3. *e* ← *netat* − 1
/* *e* est l'état d'arrivée de l'arc pointé par *ptarc1* */

4. **tantque** *e* ≥ 1 **faire**

(a) **tantque** *ptarc1* ≥ *premier_arc*(e) **faire**
i. *Trans(ptarc2)* ← *Trans(ptarc1)*
ii. *Depart(ptarc2)* ← *Depart(ptarc1)*
iii. Si *pre_image(Depart(ptarc1))* ≠ 0 alors
A. *ptarc2* ← *ptarc2* − 1
B. *Trans(ptarc2)* ← *Trans(ptarc1)*
C. *Depart(ptarc2)* ← *pre_image(Depart(ptarc1))*
D. *quantite* ← *quantite* − 1
E. Si *quantite* = 0 alors
Retourner

(b) *ptarc1* ← *ptarc1* − 1

(c) *ptarc2* ← *ptarc2* − 1

5. *premier_arc*(e) ← *ptarc2* + 1

6. Fin.

Annexe D

Syntaxe abstraite des synonymes et des expressions booléennes.

La définition de synonymes suit la syntaxe suivante :

dcl_alias :*ident* "==" *alias*

alias :*alias_process*
| *alias_symbole*
| *alias_interaction*
| *alias_echange*
| *alias_echange_var*

alias_process :*nom_process* { "." *nom_sous_process* }*
| *ident_process* { "." *nom_sous_process* }*

alias_symbole :*ident_symbole*
| { *alias_process* "." } *nom_symbole*

alias_interaction :*ident_interaction*
| { *alias_process* "." } *nom_point_interaction*

alias_echange : *ident_echange*
| { *alias_interaction* "." } *nom_signal*

alias_echange_var:*ident_echange_var*
|{ *alias_echange* "." } *nom_echange_var*

nom :*ident indice_ident* { "." *ident indice_ident* }*

indice_ident :{ "[" *indice_const* { "," *indice_const* }* "]" }

indice_const :*entier* | *chaine* | *alias_symbole_const*
| "*true* " | "*false* "

Dans les règles précédentes on utilise les conventions suivantes :

- ***ident_process*** est un identificateur qui représente un synonyme de type ***alias_process***
- ***ident_symbole*** est un identificateur qui représente un synonyme de type ***alias_symbole.***
- ***ident_interaction*** est un identificateur qui représente un synonyme de type ***alias_interaction.***
- ***ident_echange*** est un identificateur qui représente un alias de type ***alias_echange.***
- ***ident_echange_var*** est un identificateur qui représente un synonyme de type ***alias_echange_var.***
- ***alias_symbol_const*** est un ***alias_symbole*** qui représente une constante du programme ESTELLE/R.
- ***nom_process*** est un nom qui représente le nom d'une tâche de la spécification.
- ***nom_sous_process*** est un nom qui représente le nom d'une sous-tâche du module courant.
- ***nom_symbole*** est un élément qui représente un symbole du processus défini par ***alias_process.***
- ***nom_point_interaction*** est un nom qui représente le nom des instances de point d'interaction du processus considéré.
- ***nom_signal*** est un identificateur qui est le nom d'un signal valide pour ce point d'interaction.
- ***nom_echange_var*** est un identificateur qui est le nom d'une des variables dans la liste associée au signal considéré dans la déclaration du canal.

On peut remarquer les points suivants :

1. **Un synonyme ne peut pas être utilisé pour référencer une partie d'une variable d'état (sous-champ d'une variable de type registre ou élément d'un tableau).**

2. **Le premier identificateur de la définition d'un synonyme peut être un synonyme déjà défini.**

La définition d'expressions booléennes.

La définition d'expressions booléennes suit la syntaxe suivante :

predicate :*ident* "==" *expr_booleenne*

expr_booleenne :*expr_simple* { *op_de_relation expr_simple* }

expr_simple :*terme* { *op_additif terme* }*

terme :*facteur* { *op_multiplicatif facteur* }*

facteur :*integer* | *string* | "*false* " | "*true* "
| "*[*" { *expr* { "," *expr* }* } "*]*"
| "*(*" *expr* "*)*"
| "*not*" *facteur*
| *ref_a_une_var*
| *expr_etat*

expr_etat :"*at*" "*(*" *L_etat* "*)*"

L_etat :*alias_symbole_etat* { "," *alias_symbole_etat* }*

ref_a_une_var :*alias_symbole_var indice_ident* {"." *ident indice_ident*}*
| *alias_echange_var indice_ident* {"." *ident indice_ident*}*

indice_ident :{"*[*" *ident_const* { ","*indice_const*}* "*]*" }

index_const :*integer* | *string* | *alias_symbole_const*
| "*true* " | "*false* "

op_additif :"*+*" | "*-*" | "*or*"

op_multiplicatif:"***" | "*/*" | "*div*" | "*mod*" | "*and*"

op_de_relation :"=" | "<>" | "<" | ">"
| "≥" | "≤" | "*in*"

Dans les règles précédentes on a utilisé les conventions :

- *alias_symbole_etat* est un synonyme dont la valeur est un état ESTELLE/R ou un ensemble d'états ESTELLE/R.
- *alias_symbole_var* est un synonyme dont la valeur est une variable d'état.
- *alias_symbole_const* est un synonyme dont la valeur est une constante du programme ESTELLE/R.

Lors de la définition des expressions booléennes, l'évaluateur suppose corrects les corps des définitions, c'est-à-dire qu'aucune vérification de types n'est faite.

Annexe E

Descriptions des menus de Xesar.

Le menu principal de XESAR.

Le menu principal de XESAR est montré dans la figure E.1. Parmi les options on trouve :

Man XESAR : montre la documentation de XESAR accessible par l'exécution de la commande UNIX *man xesar*.

Compilation : exécute la phase de compilation.

Configuration : exécute la phase de configuration.

Automaton_generation : exécute la phase de génération et composition des automates.

Graph_generation : exécute la phase de génération du graphe d'états.

Chain : exécute les phases de compilation, configuration, génération et composition des automates et génération du graphe d'états.

Evaluation : montre le menu d'évaluation.

Le menu d'évaluation.

La figure E.2 montre le menu d'évaluation. Les options offertes dans ce menu sont :

```
================================== OPTIONS ==============================
 -L NO                                  -S NO
 -I                                     -W 2
 -O 1                                   -N 65534
  >

================================ PARAMETERS ============================
Program name :
================================  XESAR PHASES  ========================

            1. Man xesar
            2. Compilation
            3. Configuration
            4. Automaton generation
            5. Graph generation
            6. Chain: compilation, configuration and generations
            7. Evaluation of formulas
            8. Purge the XRI environment
            9. Purge the XLIS environment
           10. Invoke Shell
           11. Execute a command
           12. Editor
           13. Save and quit

        '^N' or key Next line,     '^P' or key Previous line
     CR execute,  'p' modify param,  'o' modify option,  'q' quit
```

Figure E.1: Menu principal

```
                XESAR EVALUATION PHASE

Program name : bitalt

 0. Exit                        11. Enter Formulas
 1. Help                        12. Read Formulas
 2. Unix command                13. Dump Graph
 3. Set option V

21. Enter Alias                 31. Enter Predicates
22. Read Alias                  32. Read Predicates
23. List Alias                  33. List Predicates
24. Save Alias                  34. Save Predicates
25. Delete Alias                35. Delete Predicates
                                36. Evaluate Predicates
```

Figure E.2: Menu d'évaluation

Commandes Générales :

Exit. Fin de la session d'évaluation.

Help. Donne à l'utilisateur des renseignements sur la phase d'évaluation.

Unix Command. Permet l'exécution d'une commande UNIX.

Set/Reset option V. Change l'état actuel de l'option V (verbeuse). Si cette option est active, l'évaluateur donne des informations supplémentaires lors de l'évaluation des formules : la liste des états qui satisfont (ou que ne satisfont pas) la formule et les changements de la taille du graphe lors de l'évaluation des prédicats *after*.

Commandes pour la gestion des formules :

Enter Formulas. Commence une session d'évaluation. pour évaluer une formule, l'utilisateur attend le prompt **Formula**. Ensuite, il écrit une formule ou une définition de formule et obtient le résultat de son évaluation. Un point tapé au début de la ligne indique la fin de la session.

Read Formulas. Permet à l'utilisateur d'évaluer un ensemble de formules contenu dans un fichier.

Commandes pour la gestion du graphe :

Dump Graph. Ecrit dans un fichier text l'ensemble d'arcs du graphe plus une table qui permet l'association des noms internes des actions à leurs noms externes.

Commandes pour la gestion des synonymes :

Enter Alias. Commence une session de définition de synonymes. Pour définir un synonyme, l'utilisateur attend l'apparition du prompt **DEF_ALIAS**. Ensuite, il introduit sa définition. Un point tapé au début de la ligne indique la fin de la session.

Read Alias. Cette commande permet à l'utilisateur de lire un ensemble de définitions de synonymes contenu dans un fichier.

List Alias. Montre les définitions de synonymes contenues dans la table de symboles.

Save Alias. Sauvegarde dans un fichier les définitions de synonymes contenues dans la table de symboles.

Delete Alias. Commence une session d'effacements de définitions de synonymes. Pour effacer une définition, l'utilisateur doit attendre l'apparition du prompt **DEL_ALIAS**. Ensuite, il introduit le nom du synonyme à effacer. Un point tapé au début de la ligne indique la fin de la session.

Commandes pour la gestion de prédicats :

Enter Predicats. Commence une session de définition de prédicats. Pour définir un prédicat, l'utilisateur attend l'apparition du prompt **DEF_PRED**. Ensuite, il introduit sa définition. Un point tapé au début de la ligne indique la fin de la session.

Read Predicates. Lit dans un fichier un ensemble de définitions de prédicats.

List Predicates. Montre l'ensemble de définitions de prédicats contenues dans la table de symboles.

Write Predicates. Sauvegarde dans un fichier l'ensemble de définitions de prédicats.

Delete Predicates. Commence une session d'effacements de prédicats. Pour effacer un prédicat, l'utilisateur attend le prompt **DEL_PRED**.

Ensuite, il donne le nom du prédicat à effacer. Un point tapé au début de la ligne indique la fin de la session.

Predicate Evaluation. Cette option génère, compile et exécute un programme PASCAL qui évalue les prédicats contenus dans la table de symboles.

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs
. A. ARNOLD, Professeur
. P. WOLPER, Professeur,

Monsieur RODRIGUEZ SALAZAR Carlos

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Informatique"

Fait à Grenoble, le 18 mai 1988

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,