Cléo : diagnostic des erreurs en Xesar

Anne Rasse

THESE

présentée par

Anne RASSE

pour obtenir le titre de **DOCTEUR**

de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 23 novembre 1988)

Spécialité : *INFORMATIQUE*

CLEO : DIAGNOSTIC DES ERREURS EN XESAR

Thèse soutenue le 29 juin 1990

Composition du jury :

Président : J. Mossière
Examinateurs : P. Azéma
M.C. Gaudel
P.C. Scholl
J. Sifakis

Thèse préparée au sein du Laboratoire de Génie Informatique de Grenoble

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

46 avenue Felix Viallet
38031 GRENOBLE cedex

Tél. : 76.57.45.00

Année universitaire 1989

Président de l'Institut :
Monsieur Georges LESPINARD

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	INPG
BLIMAN Samuel	ENSERG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BONNET Guy	ENSPG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHERUY Arlette	ENSIEG
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COLINET Catherine	ENSEEG
CORNUT Bruno	ENSIEG
COULOMB Jean-Louis	ENSIEG
COUMES André	ENSERG
CROWLEY James	ENSIMAG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPEY Maurice	ENSERG
DEPORTES Jacques	ENSPG
DEROO Daniel	ENSEEG
DESRE Pierre	ENSEEG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSPG
FAUTRELLE Yves	ENSHMG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GENTIL Sylviane	ENSIEG
GREVEN Hélène	IUFA
GUEGUEN Claude	ENSIEG
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOST Rémy	ENSPG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LADET Pierre	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LORET Benjamin	ENSHMG
LOUCHET François	ENSEEG
LUCAZEAU Guy	ENSEEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOHR Roger	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PA ULEAU Yves	ENSEEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
PIC Etienne	ENSERG
PLATEAU Brigitte	ENSIMAG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
REINISCH Raymond	ENSPG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrièle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SERMET Pierre	ENSERG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VINCENT Henri	ENSPG
ZADWORNY François	ENSERG

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
COURNIL M.
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
LACHENAL D.
LADET Pierre
LATOMBE Claudine
LE HUY H.
LE GORREC Bernard
MADAR Roland
MEUNIER G.
MULLER Jean
NGUYEN TRONG Bernadette
NIEZ J.J.
PASTUREL Alain
PLA Fernand
ROGNON J.P.
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri
YAVARI A.R.

Chercheurs du C.N.R.S

DIRECTEURS DE RECHERCHE CLASSE 0

LANDEAU	Ioan
NAYROLLES	Bernard

Directeurs de recherche 1ère Classe

ANSARA Ibrahim
CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
KRAKOWIAK Sacha
LEPROVOST Christian
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ARMAND Michel
AUDIER Marc
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
CHATILLON Chritiant
CLERMONT Jean-Robert
COURTOIS Bernard
DAVID René
DION Jean-Michel
DRIOLE Jean
DURAND Robert
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GARNIER Marcel
GUELIN Pierre
JOUD Jean-Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MEUNIER Jacques
PEUZIN Jean-Claude
PIAU Monique
RENOUARD Dominique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
VENNEREAU Pierre
WACK Bernard
YONNET Jean-Paul

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

E.N.S.E.E.G
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.M.G
ROWE Alain

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

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIEB Maurice
VINCENDON Marc

Laboratoires extérieurs :

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Situation particulière

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J..Claude	Détachement	21/10/1989
ENSHMG	PIERRARD	J.Marie	Détachement	30/04/1989
ENSIMAG	VEILLON	Gérard	Détachement	30/09/1990
ENSIMAG	VERJUS	J.Pierre	Détachement	30/09/1989
ENSPG	BLOCH	Daniel	Recteur à c/	21/12/1988

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/1989
ENSHMG	BOUVARD	Maurice	30/09/1991
ENSEEG	PARIAUD	J.Charles	30/09/1991

Je tiens à remercier

Monsieur Jacques Mossière, Professeur à l'INPG, pour l'honneur qu'il me fait en présidant le jury de cette thèse.

Monsieur Pierre Azéma, Directeur de Recherches au CNRS, et Madame Marie-Claude Gaudel, Professeur à l'Université de Paris XI, qui ont bien voulu juger mon travail. L'accueil qu'ils m'ont réservé et leurs suggestions m'ont été précieux.

Monsieur Pierre-Claude Scholl, Professeur à l'Université Joseph Fourier, dont j'apprécie particulièrement qu'il ait accepté de participer au jury.

Monsieur Joseph Sifakis, Directeur de Recherches au CNRS, pour ses conseils et le soutien constant qu'il m'a apporté durant la réalisation de cette thèse, ainsi que pour la confiance jamais démentie qu'il a eue en l'aboutissement de ce travail.

Que soient remerciés également tous les membres du groupe Spectre pour leurs encouragements et leur amitié. Je remercie en particulier

Ahmed Bouajjani, Susanne Graf, et Jean-Claude Fernandez pour les discussions fructueuses que nous avons eues à l'occasion de leur lecture des premières versions du manuscrit,

Hubert Garavel, Pierre Berlioux et Carlos Rodriguez qui ont eu la patience de relire et de commenter des parties de cette thèse,

Alain Kerbrat pour sa collaboration à la réalisation de Cléo et les nombreuses améliorations qu'il y a apportées,

Michel Lévy, qui a accepté avec le sourire ma mainmise sur notre espace vital commun.

Je remercie enfin Philippe et tous mes amis qui ont supporté avec indulgence mon humeur souvent difficile...

à KL et KC...

Notations

A et B sont des ensembles finis :

- 2^A est l'ensemble des parties de A.
- $A \setminus B$ est le complémentaire de B dans A.
- $Card(A)$ est le nombre d'éléments de A.

Ensembles de séquences.

- A^* est l'ensemble des séquences ou suites finies d'éléments de A.
- ε désigne la séquence vide.
- A^ω est l'ensemble des séquences infinies d'éléments de A
- $A^\infty = A^* \cup A^\omega$.
- l'opération de concaténation dans A^∞ est notée "."
- si $s \in A^* \setminus \{\varepsilon\}$, $(s)^\omega = s.s.s....$
- A^c est l'ensemble des séquences infinies cycliques d'éléments de A :

$$A^c = \{s_1.(s_2)^\omega \,/\, s_1 \in A^* \text{ et } s_2 \in A^* \setminus \{\varepsilon\}\}$$

- L'opération de concaténation est étendue aux ensembles de séquences de façon à être distributive sur l'opération d'union :

si $U,U' \subset A^\infty$, on note :

$$U.U' = \{s.s' \,/\, s \in U \cap A^*,\ s' \in U'\} \cup (U \cap A^\omega)$$

$$U^1 = U \text{ et pour } i \geq 2,\ U^i = U^{i-1}.U$$

$$U^* = \{\varepsilon\} \cup U^1 \cup U^2 \ldots$$

$$U^\omega = \{s_0.s_1.s_2 \ldots \,/\, \forall i \in \mathbb{N},\ s_i \in U \text{ et } s_i \neq \varepsilon\}$$

Expressions régulières et ω-régulières sur un ensemble fini A.

Une *expression régulière* sur A est une expression du langage :

$$U ::= \{a\} \mid \varnothing \mid (U) \mid U \cup U \mid U.U \mid U^*$$

et une expression ω-régulière sur A est une expression du langage :

$$U ::= \{a\} \mid \varnothing \mid (U) \mid U \cup U \mid U.U \mid U^* \mid U^\omega$$

où a est un élément de A.

On dit qu'un ensemble U est régulier s'il peut s'exprimer par une expression régulière, qu'il est ω-régulier s'il peut s'exprimer par une expression ω-régulière. En particulier, $\{\varepsilon\}$ et A^∞ sont des ensembles ω-réguliers car :

$\{\varepsilon\} = \varnothing^*$, et $A^\infty = A^* \cup A^\omega$

On identifie généralement un ensemble ω-régulier et l'expression ω-régulière qui le représente, et on emploie une notation allégée pour les expressions ω-régulières :

$\varnothing^*$ est noté ε,

$\{a\}$ est noté a.

Séquences d'états.

Q est un ensemble dont les éléments sont appelés *états*.

Si $s \in Q^* \setminus \{\varepsilon\}$ et $s = q_1 \ldots q_i \ldots q_n$:

- q_i est un *état* ou *élément* de s, il est noté $s(i)$.
- q_1 est appelé *état initial* ou *origine* de s, il est noté *premier*(s).
- q_n est appelé *dernier état* de s, il est noté *dernier*(s).
- un *état intermédiaire* de s est un état de s qui n'est pas le dernier.

Si $s \in Q^\omega$ et $s = q_1 \ldots q_i \ldots$:

- q_i est un *état* ou *élément* de s, il est noté $s(i)$
- q_1 est appelé *état initial* ou *origine* de s, il est noté *premier*(s).

Si $s \in Q^* \cup Q^\omega$:

- *états* $(s) = \{q \in Q \,/\, \exists i : s(i) = q\}$

Si $s \in Q^*$:

- *longueur*$(\varepsilon) = 0$
- *longueur*$(q_1 \ldots q_i \ldots q_n) = n$

Si $s \in Q^\omega$:

- *longueur*$(s) = \omega$ où ω est tel que $\forall n \in \mathbb{N}$, $\omega > n$.

Une séquence s' est une *sous-suite* de s s'il existe des séquences $s_1, \ldots s_n$ et $s'_1, \ldots s'_n$ telles que :

$s' = s'_1.s'_2 \ldots s'_n$ et $s = s'_1.s_1.s'_2.s_2 \ldots s'_n.s_n$

Une sous-suite s' de s est un *préfixe* de s si il existe une séquence s'' telle que $s = s'.s''$;
ce préfixe est *strict* si $s'' \neq \varepsilon$.

Une sous-suite s' de s est un *suffixe* de s si il existe une séquence s'' telle que $s = s''.s'$; ce suffixe est *strict* si $s'' \neq \varepsilon$.
Une sous-suite s' de s est un *circuit* de s (figure 0) si il existe deux séquences s_1 et s_2 telles que $s = s_1.s'.s_2$ et $dernier(s') = dernier(s_1)$.

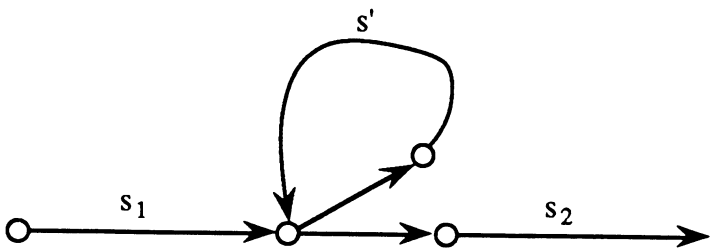


figure 0. s' est un circuit de $s_1.s'.s_2$.

Si $s = s_1.s'.s_2$ et si s' est un circuit de s, alors on dit que $s_1.s_2$ est la séquence s *privée* du circuit s'.

Formules.

$P_1, \ldots, P_n$ étant des prédicats de base, les formules sont engendrées par la grammaire :

$f ::= f \vee f \mid g$
$g ::= g \wedge g \mid h$
$h ::= (f) \mid \neg h \mid pot[f]h \mid some[f]h \mid al[f]h \mid inev[f]h \mid P_1 \mid \ldots \mid P_n$

Nous allégerons l'écriture des formules en employant l'écriture abrégée '*op f* ' pour les formules de la forme '*op*[*vrai*]*f*' où *op* est un des opérateurs *pot*, *some*, *al* ou *inev*.
La notation *fb* désigne les *formules de base*, c'est-à-dire les formules équivalentes à un prédicat de base, à la négation d'un prédicat de base, ou à une formule de la forme $al[f_1]f_2$, ou $inev[f_1]f_2$.

Table des matières

Introduction

L'utilisation de systèmes parallèles en général et de systèmes réactifs en particulier est de plus en plus répandue. Un système parallèle est composé de plusieurs processus fonctionnant simultanément en coopération. Un système réactif est un système parallèle destiné à maintenir en permanence une interaction avec son environnement, comme par exemple les protocoles de communication et les systèmes d'exploitation. Des systèmes réactifs sont présents dans la plupart des systèmes informatiques, dans lesquels leur fiabilité est d'une importance primordiale. D'autre part leur complexité croissante a rendu nécessaire le développement d'outils favorisant leur conception et leur réalisation : il s'agit en particulier d'outils permettant de vérifier leur fonctionnement et d'en corriger les erreurs.

Vérification et aide à la mise au point.

Vérifier consiste à comparer un système à ses spécifications en établissant entre eux une relation de satisfaction. Le système est réalisé par un programme écrit dans un langage exécutable et ses spécifications décrivent les services que l'on attend du système. Si le programme est conforme à ses spécifications, il est correct. Inversement, s'il ne satisfait pas ses spécifications, il est incorrect relativement à ses spécifications. Dans ce dernier cas, le programme doit être corrigé. Le problème est alors le suivant : sachant qu'il y a une erreur dans le programme, quelle est-elle ? L'expérience prouve qu'en l'absence d'informations complémentaires il est extrêmement difficile, voire impossible, de corriger un programme. Ainsi, s'il est satisfaisant de savoir vérifier des programmes corrects, le résultat de la vérification est par contre insuffisant dans le cas les programmes incorrects.
Pour pallier cette faiblesse, il convient d'adjoindre aux vérificateurs des outils d'aide à la mise au point qui permettront à l'utilisateur de corriger le programme. Leur fonction est de fournir un diagnostic d'erreur qui explique en quoi le programme est incorrect.

Un outil d'aide à la mise au point est en général conçu comme un complément indispensable d'un outil de vérification ; les phases de vérification et d'aide à la mise au point ne sont pas toujours dissociées, et d'une manière générale, les techniques d'aide à la mise au point sont tributaires de celles employées pour la vérification : aussi est-il difficile d'aborder le problème de l'aide à la mise au point indépendamment de celui de la vérification. Dans le cas des systèmes parallèles, les outils d'aide à la mise au point sont encore à l'heure actuelle du domaine expérimental : quelques aspects en sont étudiés au cours du chapitre 1.

Il existe principalement deux approches au problème de la vérification. Dans la première, les propriétés du programme sont analysées à partir de sa syntaxe. L'analyse est effectuée par un système déductif, c'est-à-dire un ensemble de règles d'inférence qui décrivent la relation attendue entre le programme et ses spécifications : il s'agit donc d'établir une preuve dans ce système déductif. Du fait de leur non déterminisme, cette méthode est difficilement automatisable dans le cas des systèmes parallèles, l'utilisateur étant amené à guider lui-même la déduction au moment de certains choix. Dans la seconde approche, on réalise une modélisation des comportements du système. La spécification définit une classe de modèles, et la vérification est le test d'appartenance du modèle du programme à cette classe (relation de satisfaction, ou d'équivalence modulo un certain critère).

Vérification des systèmes parallèles finis.

D'une manière générale, la vérification se heurte à des problèmes d'indécidabilité. Le comportement du programme dépend de la valeur des données dont le domaine peut être infini : le programme a dans ce cas une infinité d'états. Cependant, l'étude des systèmes parallèles a montré que leur complexité provient du contrôle, et non pas des données : dans le cas des protocoles de communication, par exemple, les données sont manipulées sans tenir compte de leur valeur, et seulement en fonction d'informations de contrôle qui ont un nombre fini de valeurs. Cela signifie qu'il est possible de se libérer du problème des données en limitant les valeurs de ces dernières à un domaine fini, sans rien perdre par rapport au problème considéré : il est alors possible de modéliser les principaux aspects du comportement de ces systèmes par un nombre fini d'états. Parmi les outils permettant de vérifier de tels systèmes, citons XESAR [QS82], [RRSV87], EMC [CES83] et MEC [Ar89].

Usuellement, l'ensemble des comportements d'un système parallèle est représenté par un *système de transition* ou graphe orienté. Chaque séquence de ce système de transition représente un entrelacement possible des exécutions des différents composants du système parallèle. Une exécution du système est donc modélisée par une séquence de ce graphe

appelée *séquence d'exécution*, et son comportement global à partir d'un état est l'arbre obtenu en développant le graphe à partir de cet état : cet arbre est appelé *arbre d'exécution.*
Les spécifications décrivent des propriétés du programme soit en termes de séquences d'exécution, soit en termes d'arbres d'exécution. Pour exprimer de telles propriétés, des logiques temporelles particulières ont été définies [Pn77], [BMP81]. On distingue essentiellement deux classes de logiques temporelles : les logiques du temps linéaire et les logiques du temps arborescent. Les premières décrivent la manière dont les événements se produisent le long de chaque séquence d'exécution : des modèles pour de telles logiques sont les séquences d'exécution d'un programme. Les secondes décrivent des propriétés des états et de l'ensemble des séquences d'exécution qui en sont issues : les modèles pour ces logiques sont des arbres d'exécution.

Notre travail se situe dans le cadre de XESAR, dont nous donnons ci-dessous un bref aperçu :

XESAR

XESAR (figure 1) permet la vérification de systèmes parallèles non pas sur une implantation réelle, mais sur une description du système par un programme dont les exécutions modélisent les comportements du système. Le programme est écrit en Estelle/R, langage dérivé du langage Estelle, proposé par l'ISO pour la description de protocoles ([ISO85]). Le programme est analysé de façon à produire le graphe d'états. La logique utilisée pour décrire les spécifications du programme est une logique du temps arborescent : *CTL* [CES83]. On en donne une description au chapitre 2.
L'analyse d'un programme en Estelle/R se décompose en trois phases :

- compilation et configuration du programme en tâches.
- génération : le fonctionnement de chaque tâche est représenté par un automate séquentiel étendu. Les états sont des vecteurs de variables, et les transitions sont étiquetées par des séquences d'instructions. Un réseau de Petri interprété est obtenu à partir de ces automates en couplant les transitions des automates correspondant aux communications entre les tâches.
- génération du graphe d'états : le réseau de Petri obtenu lors de la phase précédente est entièrement simulé, en tenant compte des valeurs des variables. Le résultat de cette simulation est un graphe dont les états sont des vecteurs de valeurs des variables, et les transitions des séquences d'instructions sur les variables, ou des instructions de communication entre les différentes tâches.

Vérifier qu'un programme satisfait une spécification représentée par une formule *f* revient à vérifier que tous les arbres d'exécution du programme satisfont *f*. Un arbre étant déterminé par sa racine, on dira qu'un état du programme satisfait *f* si l'arbre d'exécution à partir de cet état satisfait *f*. La procédure de vérification a été décrite dans [Ro88] ; elle consiste à évaluer l'ensemble des états qui satisfont *f*, en considérant les opérateurs de *CTL* comme des fonctions sur les ensembles d'états. Nous reviendrons au paragraphe 1.3.3. sur cette procédure.

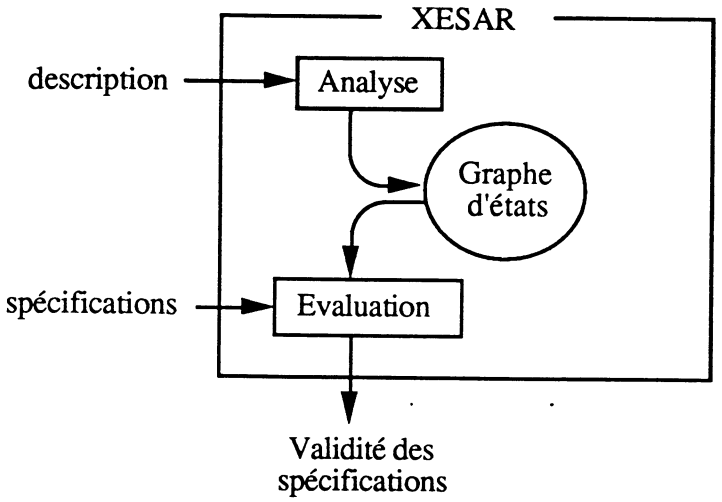


figure 1. XESAR

XESAR fournit pour chaque formule représentant une spécification un résultat de l'un des deux types suivants :

- la formule est vraie pour tous les états du graphe, ce qui signifie que la spécification correspondante est vérifiée,
- la formule est fausse pour certains états du graphe, ce qui signifie qu'une des spécifications n'est pas vérifiée par le programme.

Dans ce dernier cas, la seule information complémentaire que peut obtenir l'utilisateur est le numéro des états qui ne vérifient pas la formule : cette information ne permet en aucun cas de retrouver dans le texte du programme la cause de l'erreur. D'autre part, la taille du graphe d'états rend impraticable son analyse par l'usager.

Il était donc d'une importance cruciale d'élaborer un outil ayant pour fonction de fournir un diagnostic d'erreur en termes aussi proches que possible de la description du système : en s'appuyant sur l'évaluation des formules et en connaissant le graphe d'états, il est possible de rechercher la cause de l'erreur dans la structure de ce graphe. C'est pour répondre à ce besoin que l'idée d'un logiciel de calcul de diagnostics d'erreur a vu le jour, pour compléter la partie "vérification" de XESAR. La conception et la réalisation d'un tel logiciel, Cléo, fait l'objet de ce travail.

Notion de diagnostic.

Usuellement, le terme *diagnostic* est employé dans le sens de diagnostic médical. Il désigne "l'action de déterminer une maladie d'après ses symptômes" (Petit Robert), ou le résultat de cette action. Cette notion peut être étendue à d'autres domaines : d'une façon générale, un diagnostic est une "hypothèse tirée de signes" (idem). Un symptôme est "ce qui manifeste ou révèle" (idem). Il est le résultat de *l'observation* d'un sujet (un malade, un programme...), c'est-à-dire qu'il est la projection de l'état du sujet dans un domaine observable :

- pour un malade, un symptôme est un malaise qu'il ressent,
- dans le cas qui nous intéresse, le sujet est un programme, et le symptôme est le fait qu'une spécification du programme n'est pas vérifiée.

Ayant constaté un symptôme (d'une maladie ou d'une erreur), on souhaite généralement le faire disparaître, c'est-à-dire trouver le remède à ce défaut.
La constatation d'un symptôme n'est pas suffisante pour déterminer un remède adapté. Il est nécessaire d'approfondir la connaissance du sujet afin d'établir la bonne manière de remédier au défaut constaté : c'est la fonction du diagnostic (figure 2).

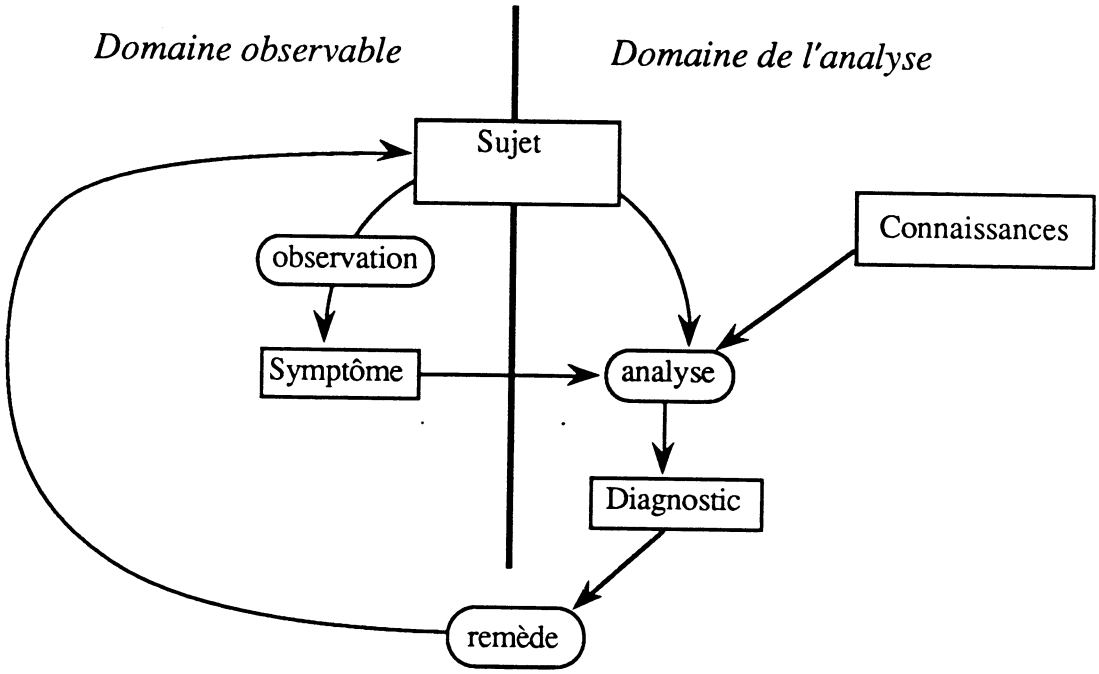


figure 2

Si le symptôme est du domaine observable, le diagnostic est du domaine du spécialiste, ou de *l'analyse*. Tous deux sont des propriétés du sujet, et le diagnostic est la détermination de la cause du symptôme :

- déterminer la maladie qui est la cause du malaise,
- mettre en évidence les propriétés du programme (c'est-à-dire de ses séquences d'exécution) qui sont la cause pour laquelle une spécification n'est pas vérifiée.

Le diagnostic est établi à partir :

- du symptôme observé,
- de l'analyse du sujet,
- de connaissances permettant de guider l'analyse et d'en tirer des conclusions.

En ce qui nous concerne (figure 3), les deux derniers points seront :

- l'analyse du système de transitions qui modélise les comportements du programme,
- la connaissance de la sémantique du langage de spécifications pour le programme.

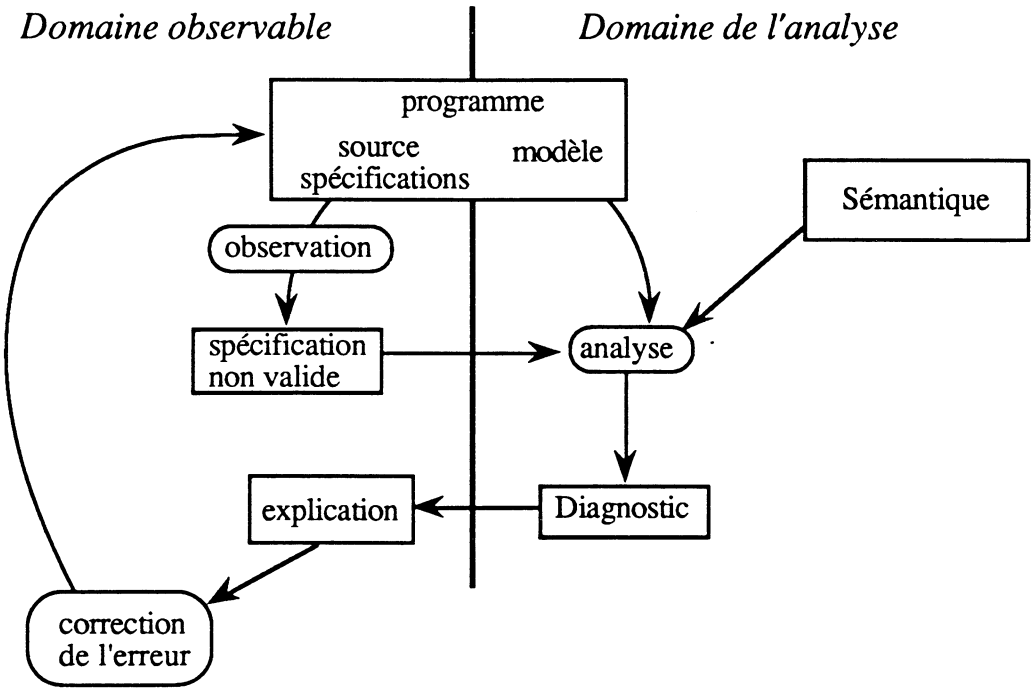


figure 3

La fonction du diagnostic est de donner des indications sur le remède à employer : il s'adresse donc à celui qui est amené à déterminer ce remède. Dans le cas du malade, c'est le médecin qui établit le diagnostic et qui propose le remède. Par contre, dans le cas du programme, le diagnostic est fourni par Cléo, et c'est l'utilisateur qui doit corriger l'erreur. Pour que le diagnostic soit accessible à l'utilisateur, il doit être projeté dans le domaine de l'observation : dans le cas de Cléo, le résultat de cette "projection" est appelé *explication*.

Diagnostics et explications en Cléo.

Les spécifications non valides pour lesquelles Cléo fournit diagnostic et explication sont des spécifications usuelles des systèmes répartis : celles qui expriment des propriétés de sûreté et de vivacité. Les premières signifient qu'une erreur ne peut jamais se produire, les secondes qu'il se produira inévitablement un événement souhaité. Pour être conforme à des spécifications de ce type, le programme doit être tel que toutes ses exécutions possibles (c'est-à-dire toutes les séquences du modèle) ont la propriété souhaitée. Le cas contraire se traduit par l'existence d'une séquence (au moins) conduisant à une erreur, ou le long de laquelle l'événement souhaité ne se produit pas. Un diagnostic est alors un exemple de séquence d'exécution dont l'existence prouve que le programme n'est pas correct.

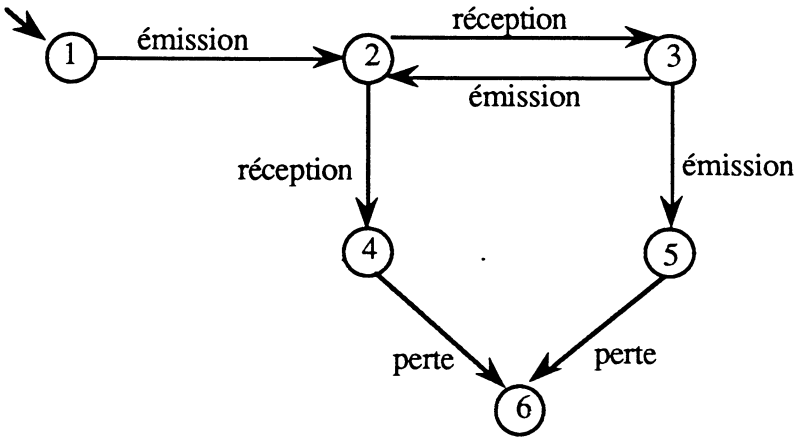


figure 4

Exemple.

Le graphe représenté par la figure 4 modélise un système dont une spécification est : "le système ne se bloque jamais". Le système ne satisfait pas cette spécification, un diagnostic en est par exemple l'existence de la séquence : 1.2.4.6, car dans l'état 6, le système ne eput plus évoluer.

L'explication a pour but de montrer à l'utilisateur, à l'aide du diagnostic, pourquoi la formule considérée n'est pas valide. Nous nous attacherons donc à ce que le diagnostic et l'explication aient les qualités pédagogiques suivantes :

- minimalité du diagnostic.

Les propriétés représentées par le diagnostic sont toutes nécessaires pour prouver qu'il y a une erreur ; par exemple "la batterie est à plat et les pneus sont usés" n'est pas une

explication minimale du fait que "la voiture ne démarre pas". Ce critère a été pris en compte pour les explications et définit un sous-ensemble de celles-ci, les explications minimales.

Exemple : le diagnostic ci-dessus est minimal. Un diagnostic non minimal serait l'existence de la séquence 1.2.3.2.4.6.

• lisibilité de l'explication.

L'explication doit être exprimée sous une forme accessible à l'utilisateur, c'est-à-dire en termes du domaine observable. Le diagnostic est obtenu sous la forme de séquences d'exécution, mais l'explication est donnée en termes de suites d'actions du programme.

Par exemple, pour le système ci-dessus, la suite d'actions correspondant à la séquence 1.2.4.6 est : *émission réception perte*.

Pour contribuer à une meilleure lisibilité, nous avons permis une certaine souplesse dans l'expression des explications : celle-ci peut être obtenue avec différents niveaux de précision : niveau de détail, niveau d'abstraction, focalisation sur un ou certains éléments. Une explication est exprimée en termes de séquences d'actions du programme et de sous-formules de la spécification satisfaites par les états de ces séquences. Les niveaux de précision concernent par conséquent les formules ou les actions.

• raffinement.

Pour des spécifications qui s'expriment par des formules complexes, on sera amené à raffiner une explication : les propriétés des états de la séquence nécessitent à leur tour une explication. Le mode "pas à pas" de Cléo (décrit au chapitre 4) permet à l'utilisateur d'obtenir une explication d'une formule complexe étape par étape : la formule de plus haut niveau lui est expliquée en fonction de ses sous-formules, puis, à sa demande, lui sont fournies les explications des sous-formules.

• abstraction.

Généralement, la taille du graphe d'états rend les explications "brutes", mêmes minimales, inexploitables par l'utilisateur. On donne donc à celui-ci la possibilité de définir l'ensemble des transitions qu'il juge significatives. Les explications lui sont alors données sous une forme simplifiée, dans laquelle seules apparaîtront les transitions de son choix.

Pour ce dernier point, une méthode consiste à déterminer d'abord l'explication, puis de la simplifier à la demande de l'utilisateur. Nous avons adopté une stratégie différente : connaissant les transitions jugées significatives par l'utilisateur, l'explication est calculée directement sous une forme réduite : ceci nous a conduit à étudier quelles simplifications du

modèle, compatibles avec la réduction des explications, pouvaient être effectuées préalablement à leur calcul. Nous avons défini pour chaque formule f, une relation d'équivalence entre modèles : si deux modèles sont équivalents, les ensembles d'explications réduites de f dans ces deux modèles sont les mêmes. Intuitivement, si deux modèles sont équivalents, ils satisfont f pour les mêmes raisons.

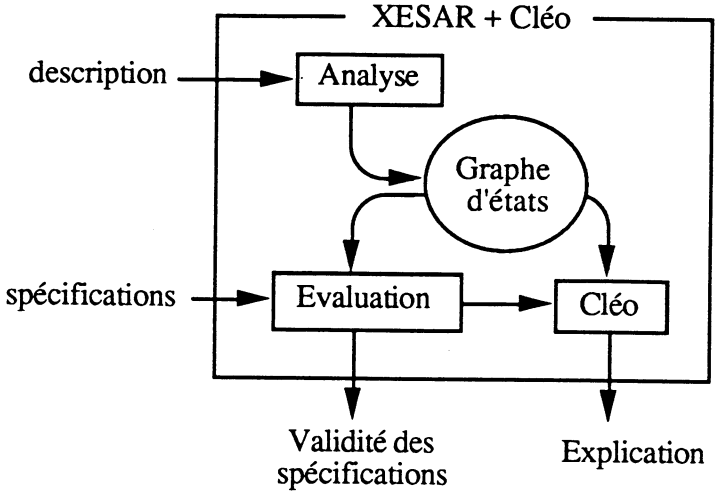


figure 5. XESAR et Cléo

Cléo s'intègre dans le système XESAR (figure 5) : par conséquent certains choix de représentations ont été faits de manière à assurer la cohérence avec XESAR.

Relation entre le diagnostic et le test.

Le système Cléo a été conçu et réalisé pour l'élaboration de diagnostics d'erreur. Cependant, le problème du diagnostic est très voisin de celui de la génération de séquences de tests guidée par les propriétés, pour les implantations réelles de systèmes :
La mise au point d'un programme Estelle/R décrivant un système de processus communicants est une première étape vers la réalisation d'un tel système. On dispose alors d'une description Estelle/R correcte relativement aux spécifications. Dans une seconde étape, le modèle Estelle/R devient la spécification, et il s'agit de tester la conformité de l'implantation réelle avec cette description.
L'implantation peut être considérée comme une boîte noire dont on ne peut observer que les interactions avec l'environnement, c'est-à-dire les entrées/sorties. La procédure de test de l'implantation consiste à comparer son comportement observable avec la description de référence. La comparaison s'effectue selon un critère de correction permettant de décider si une exécution du système est correcte.

Un test est une exécution du système : dans le cas des systèmes déterministes, un test est défini par la valeur initiale des données, dans le cas des systèmes parallèles, il est défini par une séquence d'entrées. Le domaine des données ou des entrées étant généralement infini, le système a un nombre infini d'exécutions possibles : il n'est donc pas possible de tester chacune d'elles. Le problème du test réside dans le choix d'un ensemble fini de tests (ou jeu de tests) permettant d'assurer de façon fiable, que si chacun de ces tests est correct, le système testé est une implantation correcte de la description.
Un test est défini à partir d'une séquence du modèle : à chacune de ces séquences est associée une séquence d'entrées/sorties ; le test est l'exécution du système pour la suite des entrées, appelée *séquence de test*. Un test est correct s'il définit la même suite d'entrées/sorties que la séquence du modèle.
Un jeu de test est donc défini à l'aide d'un ensemble de séquences du modèle. Il est choisi en fonction du critère de conformité que l'on veut tester. Un critère usuel est le critère de couverture de toutes les transitions du modèle : la méthode de sélection de jeux de tests associée est appelée *transition tour* [SB84].
Un autre critère peut être une propriété : si le modèle abstrait a telle propriété à cause de telles séquences d'exécution, tester ces séquences sur l'implantation donne une bonne indication sur le fait qu'elle satisfait ou non cette propriété.
Ainsi, le choix de séquences de test guidées par une propriété (représentée par une formule f) est un problème analogue à celui du calcul du diagnostic d'erreur (pour des spécifications exprimées par une formule $\neg f$) : il s'agit dans les deux cas de rechercher dans le graphe du programme, les séquences d'exécution qui ont pour conséquence que le modèle satisfait f.
Par conséquent, bien que la vocation initiale de Cléo soit le diagnostic, une autre application possible des principes présentés dans ce mémoire concerne la génération de séquences de tests.

Organisation du document

• Le chapitre 1 est un rapide tour d'horizon de l'état de l'art en matière d'aide à la mise au point des programmes parallèles afin de situer XESAR par rapport à d'autres approches. Ce chapitre peut être lu indépendamment des autres.

• Dans le chapitre 2, le langage de spécification, c'est-à-dire la logique *CTL*, est décrit. Le but de ce chapitre est de définir ce que nous appelons les explications d'une formule (dont la négation représente une spécification). Lorsque la non validité de cette spécification se traduit par l'existence de certains comportements du programme, on associe à la formule un ensemble de séquences appelées *séquences explicatives* : chacune d'elles permet de prouver

que la spécification n'est pas valide. Les explications d'une formule seront construites à l'aide de ces séquences et des explications de ses sous-formules. Une explication est obtenue par dérivation d'une assertion dans un système de règles de réécriture.

• On définit dans le chapitre 3 une forme réduite des explications en fonction d'un critère d'observation, afin d'alléger le travail d'interprétation de l'utilisateur. Le critère d'observation tient compte des transitions que l'utilisateur désire voir apparaître le long des séquences explicatives, ainsi que de la formule à expliquer. Une relation d'équivalence entre les modèles, qui préserve l'ensemble des explications réduites selon un critère d'observation donné, est définie. On donne une méthode de calcul d'une forme normale pour cette équivalence : les explications réduites pourront être calculées à partir d'un modèle sous forme normale.

• Le chapitre 4 est consacré à la description de la réalisation du système Cléo. Cléo est fondée sur l'exploitation d'automates extraits du graphe du programme, et qui acceptent un ensemble de séquences explicatives. A l'aide de ces automates préalablement réduits selon un équivalence de traces, Cléo calcule soit l'ensemble des séquences qu'ils acceptent, soit une séquence particulière parmi celles-ci. Ce dernier mode de fonctionnement permet d'obtenir une explication "pas à pas" au moyen d'une suite d'automates : chaque pas correspond à l'application d'une règle du système de réécriture décrit dans le chapitre 2.

• Un exemple d'utilisation de Cléo est présenté dans le chapitre 5. L'exemple choisi est une description du protocole du bit alterné.

Chapitre 1

Aide à la mise au point.

Au cours de ce chapitre, nous passons en revue quelques techniques d'aide à la mise au point. Deux grandes approches sont envisagées : la première est basée sur l'analyse du flot de contrôle (paragraphe 1.2), et la seconde sur l'analyse du graphe d'exécution (paragraphe 1.3) dans laquelle nous nous situons. On ne peut pas parler d'aide à la mise au point sans parler de vérification : la plupart des outils d'aide à la mise au point comprennent un module de vérification, et l'élaboration du diagnostic s'appuie le plus souvent sur des informations mémorisées lors de la vérification.

L'analyse du flot de contrôle permet de vérifier certaines propriétés du programme au moment de la compilation, indépendamment de toute exécution. Ces propriétés sont des propriétés de la structure de contrôle, et non des spécifications de fonctionnement.
L'analyse du graphe d'exécution permet de vérifier des propriétés des exécutions du programme : propriétés des états, des séquences d'exécutions ou de la structure d'exécution. Le premier cas correspond à une généralisation des outils d'aide à la mise au point traditionnels interactifs : l'utilisateur est amené à procéder lui même à certaines vérifications, l'outil d'aide à la mise au point lui proposant des fonctionnalités d'exploration de son programme. Lorsque les spécifications sont des propriétés des séquences d'exécution, il est classique de les exprimer par des formules du temps linéaire ; le diagnostic d'erreur peut être alors élaboré parallèlement à la phase de vérification. Dans le dernier cas, dont XESAR et Cléo sont un cas particulier, les spécifications sont exprimées par des formules du temps arborescent ; des techniques de vérification efficaces sont plus faciles à réaliser que dans le cas linéaire, inversement, le diagnostic d'erreur ne peut pas être élaboré pendant cette phase, et nécessite un traitement a posteriori spécifique.

1.1. Présentation du problème

1.1.1. Mise au point.

La notion d'erreur de programmation est définie par rapport à un *critère* qui caractérise les programmes corrects. Les critères de correction peuvent s'exprimer par des propriétés syntaxiques ou par des propriétés des exécutions du programme auxquelles nous réserverons le nom de spécifications. Une erreur est *détectée* par comparaison du programme au critère de correction. Ayant constaté une défaillance, il est nécessaire, afin de pouvoir la corriger, d'en déterminer la cause, c'est-à-dire de *localiser* le défaut qui l'a provoquée.
On constate que les phases de détection et de localisation sont indissociables de celle de correction proprement dite. Dorénavant, nous appelerons *mise au point* l'ensemble de ces opérations.

1.1.2. Outils d'aide à la mise au point.

Les outils d'aide à la mise au point ont pour but d'aider l'utilisateur dans les phases de détection et de localisation des erreurs.

Une aide à la détection peut être apportée par des outils interactifs qui permettent à l'utilisateur d'ausculter le programme afin de décider lui même de sa conformité au critère de correction. Elle peut être apportée également par des outils qui connaissent les critères en fonction desquels le programme est analysé, et par conséquent détectent automatiquement les erreurs : il s'agit d'outils de vérification.
Si n'ayant pas détecté d'erreur lors de la vérification, on peut assurer que le programme est correct, on parle de *vérification complète*, sinon, de *vérification partielle.*

Pour localiser une erreur, il faut :
- établir la correspondance entre la manifestation de l'erreur et le texte du programme,
- déterminer la cause de l'erreur au niveau des instructions du programme.

En effet, lorsque l'erreur est détectée par analyse des exécutions du programme, on peut assez précisément définir à quel moment de cette exécution l'erreur s'est manifestée, c'est-à-dire le point de divergence entre le comportement attendu et le comportement effectif ; cependant, il n'est pas possible de donner des spécifications suffisamment fines pour que ce point de divergence coïncide avec la cause de l'erreur : l'erreur a été provoquée en général, à

un moment antérieur de l'exécution (figure 1.1) ; l'intervalle de temps (d'exécution) entre la cause et l'effet est appelé *latence d'erreur*.

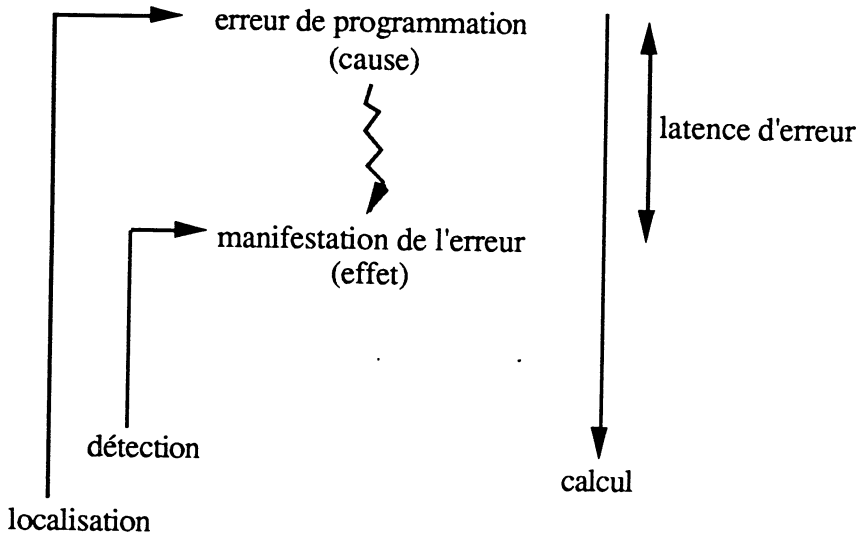


figure 1.1.

Déterminer l'origine d'une erreur est une tâche ardue qui ne peut être effectuée de manière précise que par l'auteur du programme, et bien évidemment, le degré de difficulté de cette tâche varie d'une erreur à l'autre. Cependant, une aide relative peut être apportée dans cette phase par des informations résultant de la phase de détection : l'analyse du programme effectuée lors de cette phase peut retenir des caractéristiques du programme qui ne sont pas conformes au critère. Ces informations constituent une *explication* ou un *diagnostic* de l'erreur : leur examen par l'utilisateur permet de guider sa localisation.

Nous appellerons *outil d'aide à la mise au point*, ou, l'anglicisme étant plus usuellement employé, *debugger*, tout dispositif qui favorise la détection et (éventuellement) la localisation des erreurs de programmation.

1.1.3. Niveaux de vérification

La vérification de la correction du programme est effectuée par analyse des propriétés du programme. Cette analyse est effectuée soit :

- au niveau du texte source,
- au niveau d'une représentation interne résultant de sa compilation,
- au niveau de ses comportements lors de l'exécution (figure 1.2)

Dire qu'un programme est caractérisé par l'une ou l'autre de ces formes n'est pas tout à fait exact : sur une machine donnée, plusieurs textes sources différents peuvent donner le même

code compilé, et donc définir les mêmes exécutions, et le même ensemble d'exécutions peut être obtenu à partir de codes différents. Inversement, le même texte source, sur des machines différentes peut donner des comportements différents. Nous dirons donc qu'un programme est un ensemble de comportements, mais que ce programme peut être représenté par un code source ou compilé. De même, les critères de correction d'un programme sont des propriétés de son comportement, mais certaines erreurs peuvent être détectées par analyse du texte du programme ou de la forme intermédiaire.

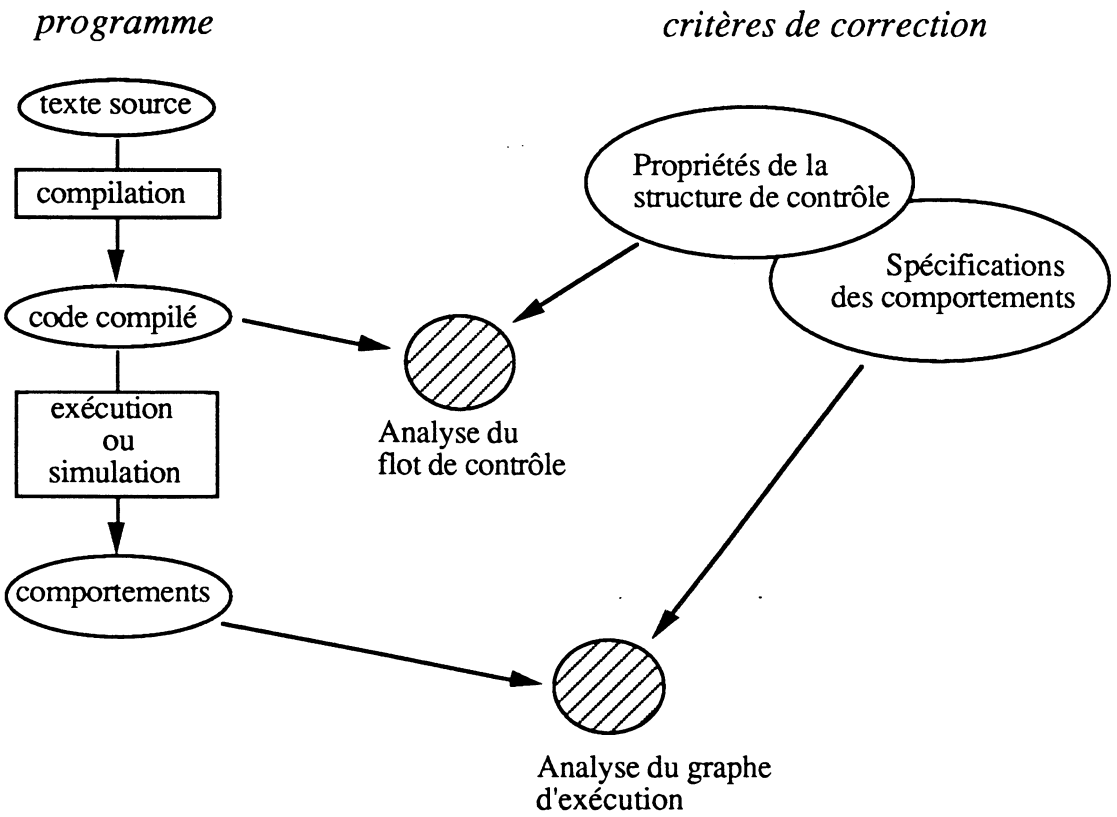


figure 1.2. Niveaux de vérification

La finesse des différents niveaux d'analyse n'est pas comparable. Le soin de l'analyse du texte source étant laissé à l'auteur du programme, nous distinguerons :

- l'*analyse du flot de contrôle* fondée uniquement sur des informations issues de la compilation ; les critères de correction sont des propriétés de la structure de contrôle, et la vérification de la conformité du programme à ces critères est généralement partielle.
- l'*analyse du graphe d'exécution* qui utilise des informations résultant de l'exécution ou de la simulation de l'exécution du programme ; les critères de correction ou *spécifications* sont des propriétés des comportements du programme lors de l'exécution, et la vérification peut être partielle ou complète.

Nous ne considérerons pas ici les outils de preuve formelle ([Ho72], [St85]), qui commencent seulement à être disponibles et dont l'efficacité dépend souvent des compétences de l'utilisateur. Leur mise en œuvre devient vite impraticable pour de gros programmes : l'utilisateur devant constamment intervenir pour guider l'outil dans ses déductions, le temps nécessaire à l'élaboration de la preuve peut être inacceptable ; d'autre part, si la taille du programme est importante, les expressions manipulées deviennent trop complexes, et l'utilisateur ne peut plus jouer son rôle de guide.

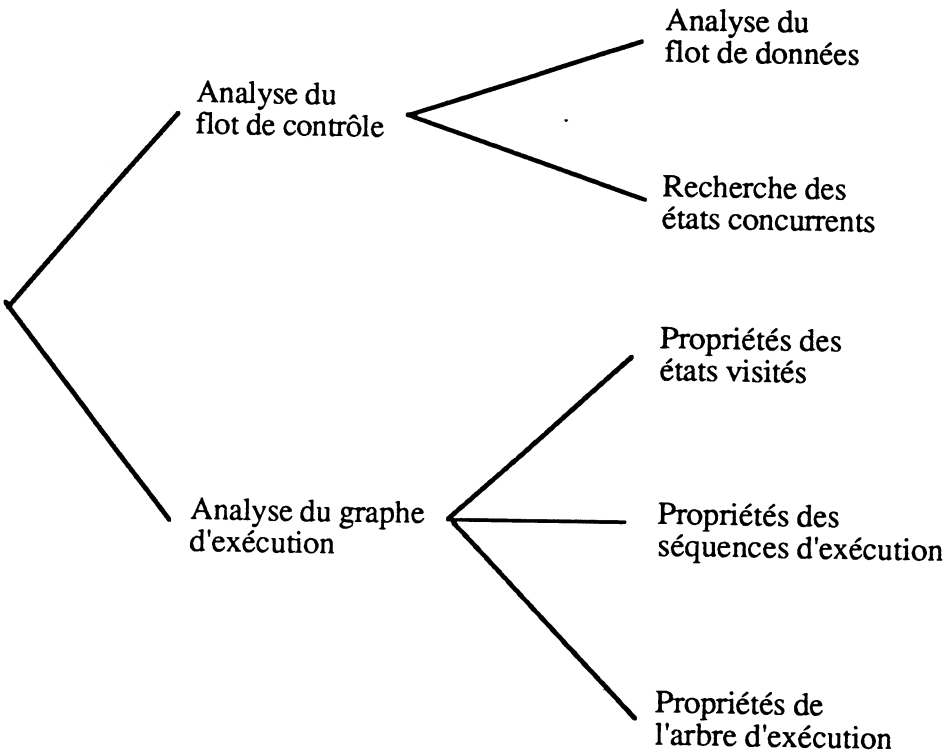


figure 1.3. Représentation schématique de la classification adoptée.

Les techniques d'analyse du flot de contrôle sont généralement coûteuses et ne permettent la détection que d'une classe d'erreurs limitée. Les critères de corrections envisagés par l'analyse du graphe d'exécution ne sont pas comparables puisqu'ils s'expriment directement en termes de spécifications de comportements.

Cependant, certains outils d'analyse du graphe d'exécution (comme les debuggers interactifs) interagissent avec le comportement du programme, par conséquent, il n'est pas garanti que le fonctionnement du programme en présence de ces outils soit le même qu'en leur absence : l'outil d'aide à la mise au point peut par exemple ralentir certaines actions, influer sur les choix non déterministes. Ce phénomène est appelé "probe effect".

Par contre, les outils d'analyse du flot de contrôle ont à leur actif de ne pas interagir avec le comportement du programme, et donc de fournir des résultats fiables : dans les cas où le

"probe effect" rend impraticable l'analyse des exécutions, ce sont les seules techniques d'aide à la mise au point envisageables.

Toute synthèse passe par une tentative de classification (figure 1.3). La plupart des méthodes proposées sont liées à un type d'application particulier ou à une architecture particulière : il est clair que les propriétés attendues d'un programme de calcul scientifique et d'un système réactif ne sont pas de même nature, et qu'il est difficilement possible de comparer les méthodes qui permettent de vérifier ces propriétés ; de même, la complexité de l'analyse d'un système distribué est inhérente à sa nature répartie : les méthodes d'analyse sont confrontés à des problèmes spécifiques à ces applications.

Ce tour d'horizon est largement inspiré de deux articles de synthèse sur le debugging des programmes parallèles de McDowell ([McDH88]) et Leu ([Le89]).

1.2. Analyse du flot de contrôle.

Il s'agit d'analyser la structure de contrôle et de déduire de cette analyse l'existence d'erreurs potentielles ou *anomalies.*

- Critère de correction.
L'analyse du flot de contrôle permet de détecter des erreurs d'usage des données et de synchronisation. Autrement dit, le critère de correction auquel cette analyse se réfère est l'absence de telles anomalies.
Parmi les anomalies d'usage des données on trouve, comme pour les programmes séquentiels, des erreurs telles que la redéfinition d'une variable ou la référence à une variable non initialisée, et des erreurs propres aux programmes parallèles telles que la mise à jour concurrente d'une même variable par différents processus. Parmi les erreurs de synchronisation, on trouve les attentes infinies ou les blocages du contrôle.

- Manifestation des anomalies au niveau de la structure de contrôle.
Dans le cas des programmes séquentiels, la structure de contrôle est l'organigramme du programme, c'est-à-dire un graphe dont les états sont les états de contrôle du programme et les transitions les instructions. Les comportements possibles du programme sont projetés sur les séquences de l'organigramme.
Dans le cas des programmes résultant de la mise en parallèle de plusieurs processus séquentiels, la structure de contrôle est obtenue en composant les structures de contrôle de

tous les processus, en tenant compte des instructions de synchronisation éventuelles. La structure obtenue peut être représentée par un réseau de Petri dont les états atteignables sont caractérisés par l'état du contrôle et des valeurs des différents processus. Le modèle sous-jacent est l'ensemble des exécutions potentielles du programme obtenues en développant la structure de contrôle. Des anomalies se manifestent par :

- l'existence de sous-séquences illégales d'opérations le long d'une séquence d'exécution potentielle,
- du fait du non déterminisme des programmes parallèles, l'existence d'incohérences dans l'ensemble de ces séquences, c'est-à-dire la possibilité d'exécuter des instructions dans un ordre quelconque alors que le résultat du programme dépend de l'ordre dans lequel elles sont exécutées.

- Localisation de l'erreur.

Le diagnostic éventuel est un rapport d'anomalie incluant la nature de l'erreur, et sa localisation dans le graphe de contrôle. La correspondance entre ce graphe et le texte du programme est suffisamment étroite pour que ce diagnostic soit directement interprétable par l'utilisateur.

Nous envisagerons deux approches. La première consiste à adapter aux programmes parallèles des techniques d'analyse de flot de données employées par les compilateurs pour l'optimisation des programmes. La seconde est fondée sur le caractère non déterministe des programmes parallèles. Le comportement du programme est modélisé par une relation d'ordre entre les dates d'exécution des instructions d'interaction (comme l'accès aux variables partagées, l'échanges de messages). Il est indispensable que bien que non déterministe, le comportement du programme ne dépende que de ses interactions avec l'environnement, et non pas des interactions des processus entre eux. Il convient donc de chercher si tous les entrelacements d'instructions possibles sont cohérents par rapport à l'ordre d'occurrence des interactions : par exemple s'il est possible à un moment donné que deux processus modifient la même variable, il y a une incohérence si l'ordre relatif de ces modifications n'est pas déterminé ; dans le cas où un processus reçoit des messages de plusieurs autres dans un ordre quelconque, il convient de vérifier que le comportement ultérieur du programme est indépendant de l'ordre d'arrivée de ces messages. En recherchant quelles instructions du programme peuvent s'exécuter en parallèle, on sait si le comportement du programme dépend de l'ordre d'exécution de ces instructions.

1.2.1. Analyse du flot de données

Les techniques d'analyse du flot de données consistent à rechercher des sous-suites illégales (non nécessairement contiguës) dans la suite d'opérations constituant les exécutions potentielles du programme. Le modèle considéré est donc la projection (ou historique) de ces exécutions sur un domaine d'événements significatifs ; considérons par exemple les événements pris en compte dans [TO80] :

- pour chaque variable, les événements significatifs sont les moments où sa valeur devient indéfinie, où elle est initialisée, et où elle est référencée.

- pour un processus, les événements significatifs sont les moments où il est devient inactif, où il est lancé, et où un autre processus attend qu'il se termine.

La présence de sous-suites illégales d'opérations peut être détectée sans que soit développée la structure de contrôle : par exemple, le graphe de contrôle du programme étant complété par des arcs correspondant aux opérations de synchronisation, l'algorithme présenté par Taylor et Osterweil ([TO80]) consiste à calculer, à chaque état, des historiques possibles concernant les variables et les processus, en fonction des événements significatifs situés en amont. Ainsi, par exemple, la référence à une variable non initialisée est détectée par la présence d'une sous-suite d'événements *indéfinie-référencée* pour cette variable, l'attente d'un processus inactif par la sous-suite *inactif-attente* pour ce processus.

Une approche légèrement différente est envisagée dans [BDER79]. A chaque état sont associés des ensembles d'états qui peuvent ou doivent être atteints avant, en parallèle ou après l'état considéré. De ces ensembles d'états sont déduites des propriétés des exécutions potentielles jusqu'à l'état considéré.

Dans les deux cas, les anomalies d'usage des données et de synchronisation détectées sont les suivantes :

- pour l'usage des données : la référence à une variable non initialisée, redéfinition d'une variable, variable jamais référencée, modification concurrente à un autre accès à la même variable (ayant donc pour conséquence qu'à un moment donné, sa valeur est indéterminée),
- pour les synchronisations : attente infinie d'un processus inactif, réentrance ou fin prématurée d'un processus (par exemple lorsque le processus appelant s'arrête).

Un des défauts majeurs de ce type d'analyse est de n'offrir à l'utilisateur qu'un diagnostic en termes d'anomalies, et non pas d'erreurs certaines. Le fait que de nombreuses erreurs potentielles correspondent à des états de contrôle inatteignables n'est pas détecté par l'analyse de flot de données classique. En effet, pour une même donnée, une modification et une

référence concurrentes ne correspondent à une erreur réelle que si pour deux exécutions réelles du programme, elles s'exécutent dans un ordre différent. Il est montré dans [CS89] que la complexité de ce problème, en n'utilisant que les informations présentes dans le graphe de contrôle est exponentielle par rapport au nombre d'états de ce graphe. Une approximation de la solution en temps polynomial y est présentée.

1.2.2. Recherche des instructions concurrentes

La recherche des instructions concurrentes d'accès à une variable partagée peut être effectuée par l'analyse de flot de données. Elle peut aussi constituer à elle seule une méthode d'analyse de la structure du programme.

Dans [BDER79], les instructions concurrentes sont déterminées directement à partir du graphe de contrôle. A cette fin, les ensembles d'états *concurrents* à un état donné – états dont la date d'occurrence peut être aussi bien antérieure que postérieure à celle de l'état considéré – et *toujours concurrents* à un état donné – états concurrents dont une occurrence appartient à toute séquence d'exécution qui contient l'état considéré – sont calculés. Le calcul de ces ensembles est effectué au moyen d'algorithmes similaires à ceux employés pour l'analyse de flot de données.

Cependant, lorsqu'il est possible de déterminer les états atteignables du contrôle du programme, l'examen des successeurs possibles de ces états permet de détecter de manière simple les anomalies d'accès concurrent à une même variable. Il s'agit en effet par exemple de détecter les états de contrôle dans lesquels deux processus peuvent accéder à une variable partagée, l'un de ces accès étant une modification.
Si le principe de cette méthode d'analyse est très simple, sa mise en œuvre est confrontée à un problème de coût. En effet, le nombre des états atteignables, est dans les cas défavorables, égal au produit du nombre d'états de chaque processus.
Une méthode de réduction est proposée dans [McD88]. Tout d'abord, le nombre des états générés peut être réduit en limitant l'état du contrôle de chaque processus à l'instruction de synchronisation la plus récente (envoi de message, lancement ou arrêt d'un processus...). Une transition représente pour un processus donné, l'exécution d'une séquence terminée par une instruction de synchronisation. A chaque transition est associé l'ensemble des variables qui sont modifiées et référencées au cours de son exécution. De plus, dans le cas où le système est composé de la mise en parallèle de processus identiques, des réductions importantes peuvent être effectuées par des regroupements d'états isomorphes (dans lesquels les tâches sont indifférenciées).

1.2.3. Combinaison avec d'autres techniques.

Devant l'explosion combinatoire du nombre d'états explorés par l'analyse des états atteignables de la structure de contrôle, Young et Taylor ([YT88]) proposent de l'employer conjointement à l'exécution symbolique du programme. La coopération des deux techniques permet de réduire l'espace de travail de chacune d'elles. Le principe est le suivant : une analyse d'une fraction de la structure de contrôle développée repère les états dont il est certain qu'ils ne conduisent pas à une erreur ; les autres, c'est-à-dire : ceux où une anomalie est détectée, et ceux dont les successeurs n'ont pas été explorés, peuvent éventuellement mener à une erreur. Dans la partie du graphe restreinte aux ancêtres de ces états, l'exécution symbolique permet de tronquer les branches dont il est certain qu'elles ne sont jamais atteintes, par exemple lorsqu'elles sont issues d'états de contrôle où une condition est évaluée à faux. L'analyse reprend alors à partir des états prometteurs d'erreur qui n'ont pas été éliminés par l'exécution symbolique. Outre le fait qu'elle réduit le champ d'investigation de l'analyse, l'avantage de cette méthode est d'éliminer un grand nombre de rapports d'anomalies qui ne correspondent pas à des erreurs réelles. Cependant, l'exécution symbolique est coûteuse et l'évaluation à faux des conditions est presque toujours indécidable.

1.3. Analyse du graphe d'exécution.

L'analyse des exécutions du programme permet de comparer une modélisation de comportements réels du programme aux comportements corrects. Le critère de correction est donné par la spécification qui est une propriété caractérisant une classe de comportements de programme. Les comportements du programme étudié doivent appartenir à cette classe.

- Modélisation des comportements du programme.

Pour les programmes qui s'exécutent sur un seul processeur, un comportement ou une exécution est la séquence des instructions et des états de la mémoire (c'est-à-dire la valeur des variables) après l'exécution de chaque instruction. Ces séquences sont appelées *séquences d'exécution.* On modélise généralement l'ensemble des exécutions d'un programme parallèle par un graphe dont les chemins sont les séquences d'exécution. En développant ce graphe à partir d'un de ses états, on obtient un arbre appelé *arbre d'exécution.*

Dans les cas des systèmes distribués où il n'est pas possible d'établir un ordre strict entre les dates d'exécution de toutes les instructions, le comportement est modélisé par une relation d'ordre partiel établie par les interactions entre les processus : si ces interactions concernent l'accès à une mémoire partagée, la séquence des accès est totalement ordonnée ; les dates d'exécution des interactions concernant l'échange de messages sont totalement ordonnées pour chaque couple de processus permettant ainsi d'établir un ordre partiel dans l'ensemble des interactions. On peut aussi modéliser le comportement des programmes distribués par des expressions d'événements qui sont des termes formés à partir des instructions au moyen d'opérateurs tels que le séquencement et l'exécution parallèle. Les méthodes de debugging de ces programmes sont basés sur la reconnaissance des expressions d'événements : nous en avons considéré quelques aspects au cours du paragraphe 1.3.1.2.

- Spécifications.

Les spécifications peuvent être des propriétés :

- des états visités (ou des événements qui se produisent) lors d'une exécution du programme,
- des séquences d'exécution du programme,
- des arbres d'exécution du programme.

Différents formalismes permettent de définir les séquences et les arbres d'exécution corrects.

- Il est usuel de définir les séquences d'exécution correctes d'un système parallèle par une formule de logique temporelle linéaire, par exemple *PTL* [Pn77] : les modèles pour une telle formule sont des séquences, une séquence d'exécution est donc correcte si elle satisfait la formule. Les formules de logique linéaire peuvent être traduites par un automate de Büchi [Bü62] fini qui accepte la même classe de séquences. On peut également décrire directement ces spécifications par un automate : une séquence d'exécution est correcte si elle est acceptée par l'automate.
- Pour décrire des classes d'arbres d'exécution, on emploie généralement une logique temporelle arborescente, par exemple *CTL* [CES83] ou *CTL** [EH83] : les modèles des formules arborescentes sont des arbres, un arbre d'exécution est correct si il satisfait la formule qui représente la spécification.

  La correspondance entre les formules temporelles arborescentes et les automates d'arbres existe, mais l'utilisation de ces derniers est moins intuitive et entraîne une mise en œuvre trop complexe pour être réalisable dans la pratique.

Il est évident que les méthodes de vérification dépendent de la nature des objets dont les spécifications expriment les propriétés :

- les propriétés des états visités lors d'une exécution du programme sont indépendantes de l'ordre d'occurrence de ces états lors de l'exécution ; la vérification de telles propriétés ne nécessite pas de mémoriser d'historique du comportement antérieur.
- les propriétés des séquences d'exécution dépendent de l'ordre d'exécution des instructions : leur vérification nécessite d'analyser chaque séquence dans son ensemble ; par contre, chaque exécution peut être vérifiée indépendamment.
- les propriétés des arbres d'exécution tiennent compte du branchement aux états de leurs modèles : leur vérification nécessite une analyse du graphe du programme.

Nous avons donc adopté une classification des techniques d'analyse du graphe d'exécution en fonction de la nature des spécifications dont elles permettent la vérification. La vérification ne pourra être complète que s'il est possible de modéliser tous les comportements du programme par une structure finie.

1.3.1. Vérification des propriétés des états visités lors d'une exécution.

Les spécifications sont constituées par un ensemble d'assertions sur l'état de la mémoire à certains moments de l'exécution du programme. Elles expriment par exemple le maintien d'une cohérence entre les valeurs des variables : propriétés invariantes, état d'une structure de données, ou un lien logique entre un résultat calculé et les données. La conformité de l'exécution du programme aux spécifications est déduite de la vérification de la validité des assertions aux instants considérés. Chaque assertion est vérifiée indépendamment des autres. Une erreur est signalée dès qu'une des vérifications détecte une assertion non valide. L'origine de l'erreur est localisée entre le point où le désaccord avec les spécifications a été détecté et le point de vérification précédent, et le diagnostic est complété par la propriété qui a été invalidée (figure 1.4).
La démarche liée à la vérification de ce type de spécifications est parfois appelée debugging cyclique. Schématiquement, cette démarche est la suivante :

on définit un point d'arrêt dans le programme,
l'exécution du programme est lancé,
le programme s'arrête lorsque son exécution rencontre le point d'arrêt,
l'état du programme est examiné,
en fonction du résultat de cet examen, un point d'arrêt est défini en amont et le programme est réexécuté, ou en aval, et le programme continue son exécution.

Ainsi, la cause de l'erreur est cernée par examens successifs.

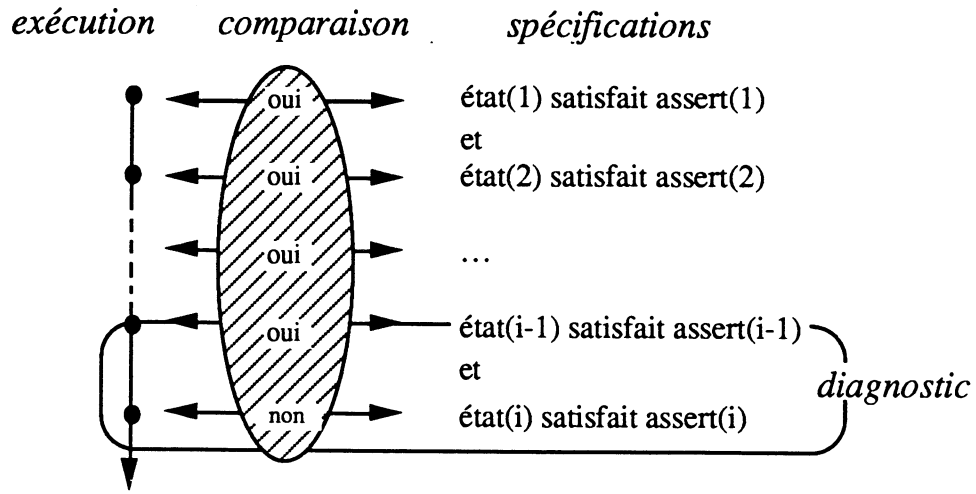


figure 1.4. Vérification des propriétés des états visités.

Des outils associés à cette démarche sont les debuggers traditionnels qui permettent la mise au point des programmes déterministes séquentiels. Nous montrons de quelle manière ces outils peuvent être adaptés pour la mise au point des programmes parallèles ou non déterministes.

1.3.1.1. Debugging traditionnel des programmes déterministes séquentiels

Il s'agit d'outils interactifs qui offrent les possibilités suivantes :

- le contrôle de l'exécution du programme par
  - la définition de points d'arrêt,
  - la reprise de l'exécution,
  - l'exécution pas à pas,
  - l'examen de traces ;

- l'examen et la modification de l'état du programme
  - c'est-à-dire essentiellement des valeurs des variables.

Un point d'arrêt est défini par :

- l'exécution d'une instruction particulière du programme,
- l'évaluation à vrai d'une condition sur les variables du programme.

Ainsi, si une spécification du programme est qu'une propriété est invariante au cours de l'exécution du programme, elle peut être vérifiée en définissant un point d'arrêt lorsque cette propriété devient fausse.

1.3.1.2. Debugging des programmes parallèles ou non déterministes

Une manière simple d'adapter ces techniques aux systèmes parallèles consiste à associer un debugger de ce type à chaque processus parallèle et à coordonner leur fonctionnement de façon centralisée. Cette méthode a le mérite d'être à l'heure actuelle à la base des seuls outils d'aide à la mise au point des programmes parallèles actuellement dans le commerce : c'est le cas par exemple de *dbxtool* [AM86].

Pour les implantations distribuées des programmes parallèles (c'est-à-dire sur des processeurs multiples), la principale difficulté est la définition d'un état global du système : elle concerne par conséquent la reprise d'exécution, la définition des points d'arrêt, et l'examen de l'état du programme. Nous présentons rapidement ci-dessous quelques solutions pour mettre en œuvre ces fonctionnalités. Un "pas d'exécution", et par conséquent l'exécution pas à pas n'ont généralement pas de sens pour les programmes distribués.

- Reprise d'exécution.
L'arrêt (provoqué par une commande ou la rencontre d'un point d'arrêt) peut concerner soit un seul processus ou un ensemble de processus déterminé, soit l'ensemble des processus. Dans le premier cas, la poursuite de l'exécution des autres processus est conditionnée par le fait qu'elle n'interagit pas avec les processus stoppés : ils peuvent donc poursuivre leur exécution jusqu'à ce qu'ils rencontrent une instruction d'interaction ou dépendante du temps (tel qu'un time-out). Ceci garantit une reprise de l'exécution à partir d'un état cohérent du système. Dans le deuxième cas, la définition d'un état global cohérent est particulièrement nécessaire si les processus s'exécutent sur des processeurs distribués : l'arrêt de tous les processus ne peut pas être simultané mais doit se produire le plus tôt possible tout en ayant le minimum d'impact sur la reprise de l'exécution ultérieure éventuelle.

- Définition des points d'arrêt.
Le domaine des expressions ou des prédicats qui interviennent dans les conditions qui définissent les points d'arrêt est beaucoup plus large dans le cas des programmes parallèles que dans celui des programmes séquentiels. En particulier si le système est distribué, il est difficile de vérifier des conditions portant sur plusieurs processus : à cause des délais de communication, il peut apparaître que la condition globale est remplie alors qu'à aucun moment elle ne l'a été, et inversement. Plus généralement dans de tels systèmes, les points d'arrêt sont définis en fonction d'événements qui regroupent des notions très variables telles que l'échange de messages, le lancement ou l'arrêt d'un processus, l'accès à la mémoire commune ou des conditions définies par l'utilisateur. L'évaluation des conditions d'arrêt suppose un contrôle global du temps afin de connaître les dates relatives d'occurrence des

états ou des transitions rencontrés par les différents processus. On trouve dans [HW88] une étude de la définition de l'état global d'un système distribué, et des points d'arrêts globaux en fonction d'événements globaux.

Les dispositifs de reconnaissance d'expressions d'événements complexes permettent la définition et la vérification de spécifications très fines. Considérons par exemple le cas de EDL (Event Description Language [BW83] et [B89]) : ce langage permet de décrire des spécifications par des expressions d'événements formées à partir d'événements élémentaires. Ces expressions constituent elles-mêmes de nouveaux événements en fonction desquels sont formées des expressions de plus haut niveau. On obtient ainsi une description hiérarchique des spécifications. Une technique de filtrage des événements permet de ne pas considérer ceux qui n'interviennent pas dans la constitution des expressions d'événements, c'est-à-dire qui ne sont pas pertinents. L'algorithme de reconnaissance d'événements est fondé sur des techniques d'analyse syntaxique : il compare le flot d'événements générés par le système et la description EDL du comportement attendu. Lorsque la comparaison échoue, un diagnostic est fourni indiquant le moment et la cause de cet échec.

Dans [BDV86], une méthode assez proche est donnée pour la mise au point de programmes distribués écrits dans un langage dérivé de *CSP*. A chaque processus est associé une expression d'événements décrivant sa spécification : le programme pourra s'exécuter tant qu'il y a correspondance entre ces événements et le comportement du programme. Les événements sont des interactions et des assertions sur les variables du processus. A chaque processus P est associé un processus de debugging connaissant sa spécification. Ce processus peut communiquer avec les processus de debugging associées aux ancêtres de P dont la spécification contrôle certaines interactions relatives à P. L'algorithme de reconnaissance est fondé sur cette hiérarchie : les assertions sont vérifiées au niveau local, et lorsque survient une interaction, son occurrence est communiquée à tous les debuggers qui la contrôlent : chacun d'eux vérifie qu'elle n'entraîne pas d'incohérence avec la spécification. Si une erreur est détectée au niveau d'un processus, il est immédiatement stoppé ainsi que tous ses descendants, et le contrôle est rendu à l'utilisateur.

- Examen de l'état du programme.

Une approche originale est envisagée dans [MC89]. Plutôt que de considérer que l'information qui intéresse l'utilisateur au moment où il stoppe le programme est l'état global du système, on lui donne la possibilité d'examiner des informations conservées depuis un point antérieur à l'arrêt. Le programme est découpé en blocs séquentiels (typiquement, un bloc est une procédure) éventuellement entrelacés. Au début de l'exécution de chaque bloc

sont mémorisées les valeurs des variables qui y sont lues, à la fin, de celles qui y sont modifiées. A partir de la structure de contrôle et de ces informations, il est alors possible de produire des traces à la demande de l'utilisateur.

Un des défauts majeurs des debuggers interactifs est que, du fait du non déterminisme des systèmes parallèles, leur comportement dans des conditions identiques ne peut pas être reproduit : si après examen de l'état du système au cours de son exécution, l'utilisateur détecte une erreur et désire en cerner la cause par une nouvelle exécution, il est possible que l'erreur ne se produise plus lors de cette nouvelle exécution. Une solution à ce problème est apportée par les techniques dites de "replay". Le principe général est le suivant : des informations sur la résolution des choix non déterministes sont mémorisées lors d'une exécution du programme, puis celui-ci est réexécuté de manière identique. Le contrôle par l'utilisateur de l'exécution du programme se fait donc sur la réexécution par la pose de points d'arrêt, l'examen de l'état des processus ... Le programme peut être "rejoué" autant de fois que le nécessitera la détection de l'erreur.
La mise en œuvre de cette technique est fortement tributaire de la quantité d'informations mémorisées lors de l'exécution initiale du programme. Les recherches dans ce domaine ont pour but de réduire cette quantité d'information. Ainsi, lorsque l'exécution des instructions d'interaction (échange de message, accès aux variables partagées) déterminent entièrement l'exécution des autres instructions, l'exécution du programme est entièrement décrite par l'ordre d'occurrence des instructions d'interaction : de ce point de vue, pour que deux exécutions du programme soient équivalentes, il suffit que l'ordre dans lequel se produisent les interactions soit le même, et ne nécessite pas que l'entrelacement de l'exécution de toutes les instructions des différents processus soit le même. Cette constatation est à l'origine de méthodes ayant pour but la réexécution du programme de manière équivalente (au sens donné ci-dessus).
Le système *Instant Replay* ([LM87]) est caractéristique de cette approche, reprise de manière similaire dans [Fo89]. Toutes les interactions peuvent être représentées par des opérations d'accès à la mémoire commune. Puisque le comportement d'un processus entre deux interactions est entièrement déterminé, la valeur affectée à une variable (ou le contenu du message émis) l'est également. Ainsi, seules sont pertinentes, et donc conservées, les informations sur les références et les modifications concernant un objet, c'est-à-dire :

- le nombre d'accès à chaque version de la valeur d'un objet,
- le numéro de la version de l'objet associé à chaque référence ou modification.

Cette méthode s'applique bien à la réexécution de l'ensemble du programme. En revanche, pour un programme constitué par exemple de la mise en parallèle de plusieurs processus

identiques, il est plus intéressant de ne réexécuter qu'un seul de ces processus ([GGK84], [PL89]). Cela nécessite bien sûr la mémorisation d'une quantité plus grande d'information afin de pouvoir simuler le reste du programme.

1.3.2. Vérification des propriétés des séquences d'exécution.

Nous nous intéressons dans ce paragraphe au cas où une exécution du programme est modélisée par une séquence d'états (caractérisés par l'état de la mémoire, c'est-à-dire des variables du programme) et de transitions (étiquetée par une instruction), et où les spécifications sont des propriétés qui ne concernent pas seulement chacun des états du programme mais toute la séquence d'instructions exécutée. De telles propriétés ne peuvent pas être vérifiées par des examens ponctuels, car elles dépendent en général du comportement passé ou à venir.
Considérons par exemple, la spécification *Spec* suivante d'un système composé d'un processus émetteur et d'un processus récepteur

Spec :

- chacune des émissions est suivie de la réception correspondante,
- il ne peut pas y avoir de nouvelle émission avant la réception du message précédent,
- le système commence par une émission.

Cette spécification peut être exprimée par la propriété : lors de la séquence d'exécution, après une émission initiale, les émissions et les réceptions ont lieu alternativement. La vérification de la spécification consiste à s'assurer que la séquence d'exécution vérifie cette propriété. Dans ce cas précis, il est simple de déterminer le point de désaccord du comportement avec les spécifications. La séquence d'exécution jusqu'au point de désaccord constitue le diagnostic de l'erreur.

- Expression des spécifications.

Une propriété exprimée par une formule de logique temporelle peut être traduite par un automate fini qui accepte les séquences qui satisfont cette propriété [VWS83]. Une propriété peut caractériser des séquences finies et/ou des séquences infinies. L'automate associé reconnaît donc des séquences finies ou infinies ; soit $S = (Q, \rightarrow, I, F, R, V_\lambda)$ un tel automate :
Q est un ensemble fini d'états, I, F, et R sont des sous-ensembles de Q appelés respectivement états initiaux, finaux, et de répétition, V_λ est une ensemble d'étiquettes, $\rightarrow$ est un sous-ensemble de $Q \times V_\lambda \times Q$ appelé relation de transition. L'ensemble des séquences acceptées par S est le sous-ensemble $L(S)$ de ${V_\lambda}^* \cup {V_\lambda}^\omega$ tel que :

si $w \in V_\lambda{}^*$, $w \in L(S)$ si et seulement si w est la trace d'une séquence de l'automate, qui va de I à F,
si $w \in V_\lambda{}^\omega$, $w \in L(S)$ si et seulement si w est la trace d'une séquence de S issue de I et qui passe infiniment souvent par un état de R.

Les automates de Büchi [Bü62] correspondent au cas où $F = \emptyset$. Par extension, nous appellerons automate de Büchi tout automate tel que S.

La classe de propriétés exprimables par un automate de Büchi inclut strictement la classe de celles que l'on peut exprimer dans la plupart des logiques temporelles : la souplesse d'expression des spécifications est donc un avantage des méthodes de vérification de propriétés exprimées par des automates.
Contrairement aux automates finis réguliers, il n'est pas toujours possible de rendre déterministe un automate de Büchi. Les propriétés exprimées par des automates non-déterministes nécessitent des techniques de vérification spécifiques que nous ne considérerons pas ici. L'automate de Büchi associé aux spécifications sera donc considéré comme étant déterministe.

Les spécifications définissent l'ensemble des séquences correctes modulo une certaine abstraction : on s'intéresse généralement à la trace de ces séquences sur un ensemble d'événements ou d'instructions significatives. L'ensemble des étiquettes V_λ d'un automate qui représente les spécifications est composé de l'ensemble V de ces événements significatifs et d'un symbole (λ) qui est associé aux transitions non significatives. Les séquences d'exécution sont modélisées par un élément de $V_\lambda{}^* \cup V_\lambda{}^\omega$ qui est une abstraction de la séquence d'instructions qu'elles représentent : toute instruction qui n'appartient pas à V est représentée par λ. On note $trace(s)$ l'abstraction de la séquence d'exécution s. Si les spécifications sont définies par l'automate S, la séquence s est correcte si et seulement si $trace(s) \in L(S)$.

Par exemple, la spécification *Spec* est décrite par l'automate représenté sur la figure 1.5 :

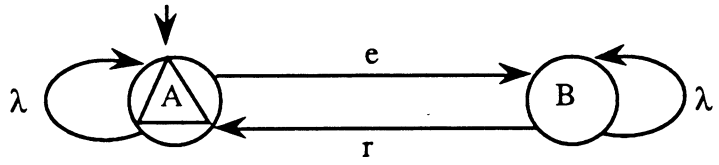


figure 1.5. Automate associé à la spécification Spec

A est l'état initial, c'est un état de répétition, et c'est le seul état final ; on s'intéresse seulement aux émissions et aux réceptions notées respectivement e et r ; si on note λ toute

action qui n'est ni e ni r, une séquence d'exécution est correcte si sa trace appartient à l'expression ω-régulière :

$$(\lambda^*.e.\lambda^*.r)^\omega$$

Les méthodes de vérification liées à l'expression des propriétés par de tels automates ont une base théorique très simple : il s'agit de décider si les exécutions du programmes sont acceptées par l'automate décrivant la spécification.
Considérons deux approches. Dans la première, on dispose d'une modélisation du programme par un système de transitions fini, obtenu par simulation (il est donc fini) : la vérification s'effectuera en analysant ce modèle, et pourra être complète. Dans la seconde, le modèle n'est pas construit (sa taille pouvant rendre impraticable la première approche) : la vérification s'effectue "à la volée" en même temps que le programme est exécuté ou simulé. Dans ce dernier cas, si le nombre d'états du programme est infini, la vérification ne peut être que partielle.

1) Le système de transitions qui modélise le programme peut être considéré comme un automate de Büchi P qui accepte l'ensemble $L(P)$ des traces sur V de ses séquences d'exécution : tous les états sont des états de répétition, et les états puits des états finaux. Si $L(S)$ est l'ensemble des séquences acceptées par S, le programme est correct si $L(P)$ est contenu dans $L(S)$.
Des méthodes de vérification liées à cette approches sont basées sur l'étude de produit d'automates. Une première classe de méthodes est fondée sur l'équivalence :

$$(L(P) \subset L(S)) \Leftrightarrow (L(P) \setminus L(S) = \emptyset)$$

en considérant que $L(P) \setminus L(S)$ est l'ensemble des séquences acceptées par le produit des automates P et S', où $L(S')$ est le complémentaire de $L(S)$. Cette méthode est adéquate lorsque les spécifications sont définies par une formule f : l'automate S' est l'automate acceptant l'ensemble des séquences qui satisfont $\neg f$ ([VW86]).

Une deuxième classe de méthodes est fondée sur l'équivalence :

$$(L(P) \subset L(S)) \Leftrightarrow (L(P) \cap L(S) = L(P))$$

Une méthode de ce type est suggérée dans [CGK87]. Nous en décrivons le principe :
Connaissant P et S, on peut construire un automate $P \times S$ qui accepte l'ensemble $L(P) \cap L(S)$. Les états de $P \times S$ sont les éléments du produit cartésien des états de P et de S, les états initiaux (resp. finaux, de répétition) sont les éléments du produit cartésien des états initiaux (resp. finaux, de répétition) de P et de S. Les transitions relient les états (q_1, r_1) et (q_2, r_2) tels qu'on passe de q_1 à q_2 dans P et de r_1 à r_2 dans S par la même instruction.

La connaissance de cet automate permet de décider si $L(P) \cap L(S) = L(P)$. En effet, $L(P) \cap L(S) = L(P)$ si et seulement si $P \times S$ est tel que :

- pour toute séquence sans circuit $(q_0,r_0)(q_1,r_1)(q_2,r_2)\ldots(q_n,r_n)$ si q_n est un état final de P, r_n est un état final de S.
- pour toute séquence $(q_0,r_0)(q_1,r_1)(q_2,r_2)\ldots(q_n,r_n)$ telle que $(q_j,r_j) = (q_n,r_n)$ et $(q_j,r_j)\ldots(q_n,r_n)$ est un circuit élémentaire, au moins un des r_i $(j \leq i < n)$ est un état de répétition de S.

Une méthode de construction de $P \times S$ consiste à parcourir en profondeur les séquences de cet automate, en mémorisant la séquence courante : si (q,r) est l'état courant de $P \times S$, et si pour un successeur q_1 de q, il existe un successeur r_1 de r tels que les transitions (q,q_1) de P et (r,r_1) de S sont étiquetées par la même instruction, alors (q_1,r_1) est un état de $P \times S$, et $((q,r), (q_1,r_1))$ est une transition de $P \times S$; si (q_1,r_1) n'existe pas encore dans $P \times S$, il devient le nouvel état courant, sinon, l'état courant devient le premier état (q',r') rencontré en "remontant" la séquence courante, tel que tous les successeurs de r' n'ont pas encore été explorés. Si au cours de cette construction de $P \times S$, on atteint un état final de P sans avoir atteint simultanément un état final de S, ou si on détecte un circuit le long duquel aucun état de répétition de S n'est rencontré, c'est qu'un des comportements possibles du programme est incorrect : une séquence d'accès à l'état concerné, ou une séquence d'accès au circuit considéré, suivie de la répétition infinie de ce circuit constituent un tel comportement. Un des intérêt de cette méthode est de permettre d'obtenir immédiatement de cette façon une explication de la cause de l'erreur.

Par exemple, soit un programme correspondant à la spécification du système émetteur / récepteur citée précédemment ; le graphe de ce programme est représenté sur la figure 1.6. En calculant le produit du graphe par S on détecte que la séquence 1.2.3.4.5 aboutit à une erreur.

Si le graphe construit est mémorisé, chaque transition est parcourue une et une seule fois (ou moins en cas d'erreur). Il reste ensuite à vérifier que tous les circuits élémentaires comportent au moins un état de répétition : un problème équivalent est de vérifier que le graphe de l'automate produit privé des états de répétition est sans circuit. Si le coût de construction de chaque transition est constant, le coût global est donc de même ordre que la taille de l'automate produit.
Cependant, en particulier si l'espace mémoire disponible ne permet pas de mémoriser la totalité de l'automate produit, on peut malgré tout le parcourir exhaustivement en ne conservant que la séquence d'accès à l'état courant, c'est-à-dire avec au plus un nombre

d'états de l'ordre du diamètre de cet automate. La contrepartie est que puisque les états parcourus n'ont pas été mémorisés, le nombre de transitions parcourues, donc le temps nécessaire, peut être une fonction exponentielle de la taille du graphe du programme.

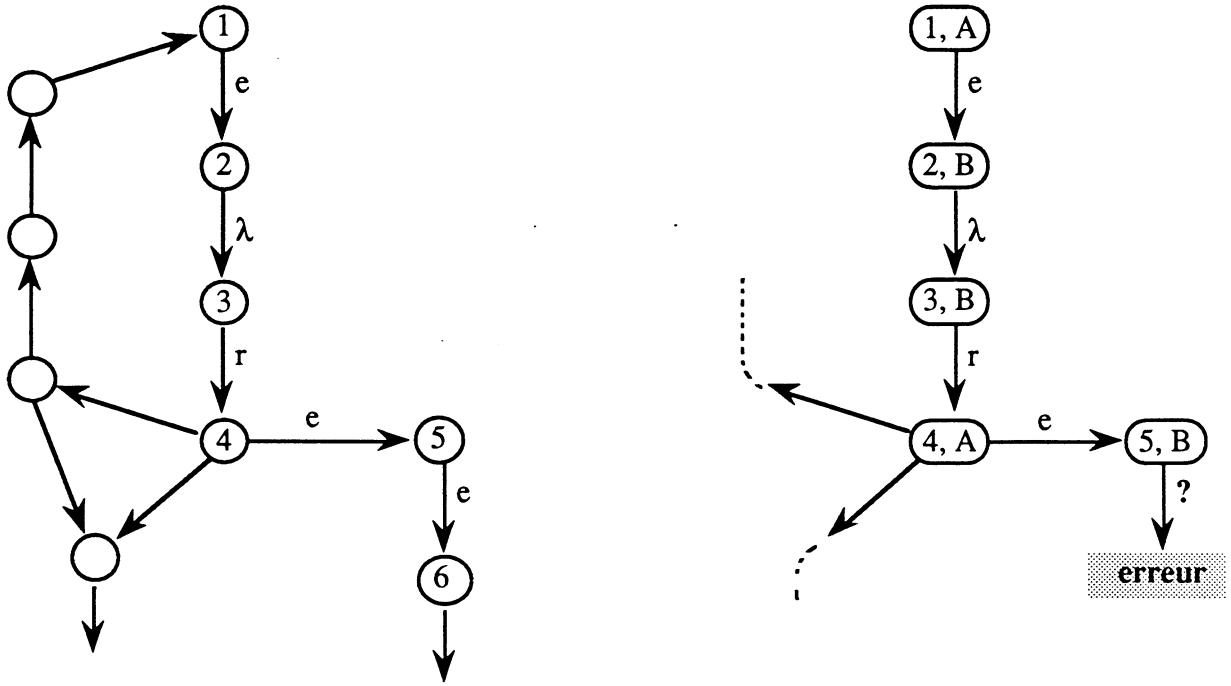


figure 1.6.

2) Si le graphe du programme est fini, remarquons qu'il n'est pas nécessaire de connaître *P a priori* pour mettre en œuvre la méthode précédente : il suffit de pouvoir parcourir le graphe du programme en profondeur, par exemple par une simulation exhaustive qui est la méthode habituelle de construction de ce graphe. L'adaptation est immédiate.

Sinon, seule une partie du graphe peut être testée, en simulant quelques exécutions du programme et en vérifiant qu'aucune erreur ne se produit pendant le temps d'observation. Il s'agit en particulier de la méthode des observateurs employée dans le système VEDA ([Gr89]).

Le principe est le suivant :

Si s est la séquence d'exécution à vérifier, un algorithme classique consiste à parcourir simultanément la séquence s et les transitions de l'automate. On peut alors conclure dans les cas suivants :

- le dernier état de s a été atteint en même temps qu'un état non final de S :

  $\rightarrow$ s n'est pas correcte

- un état final de S a été atteint en même temps que le dernier état de s :

→ s est correcte

- le dernier état de s n'est pas atteint, mais la transition suivante de s n'est pas applicable à partir de l'état courant de S :

→ s n'est pas correcte

- si on a atteint deux occurrences successives d'un état de répétition de l'automate, et que les états q_1 et q_2 correspondants de s sont identiques, un comportement correct est celui qui répèterait indéfiniment à partir de q_1 la séquence comprise entre q_1 et q_2.

Si le dernier état de s n'est pas atteint sans que le parcours des transitions de S ne soit bloqué, on ne peut pas conclure : c'est en particulier le cas si s est infinie.

1.3.3. Vérification des propriétés des arbres d'exécution.

Les propriétés auxquelles on s'intéresse ici concernent non seulement l'ordre dans lequel surviennent certains événements le long des séquences d'exécution, mais aussi des propriétés telles que le fait que toutes les exécutions, ou au moins une exécution, vérifient une certaine propriété. Une telle spécification pour un système émetteur / récepteur peut être par exemple que toute émission est inévitablement suivie d'une réception. On exprime généralement ce type de propriétés par des formules de logique temporelle arborescente, et nous dirons qu'un arbre d'exécution est correct relativement à une spécification exprimée par une formule f, si sa racine (qui est un état du graphe du programme) satisfait f.

Des approches similaires sont utilisées dans les systèmes EMC, MEC et XESAR pour la vérification des propriétés des arbres d'exécution des programmes qui peuvent être représentés par un graphe fini. On peut adapter ces méthodes pour des simulations partielles du programme : la vérification est alors partielle.
Considérons en particulier le cas de XESAR. Les propriétés sont exprimées par des formules de la logique *CTL* dont la syntaxe et la sémantique sont décrites au paragraphe 2.2. Pour vérifier que le programme est conforme à ses spécifications, le système évalue sur le graphe l'ensemble des états qui satisfont la formule décrivant les spécifications : le programme est correct si tous ses états satisfont cette formule. La procédure d'évaluation est décrite dans [Ro88].
L'évaluation d'une formule est déduite des résultats de celles de ces sous-formules : à chaque opérateur de la logique est associée une opération permettant d'obtenir l'ensemble des états qui satisfont une formule en fonction des ensembles de ceux qui satisfont ses sous-formules.

Les formules atomiques sont évaluées en fonction des valeurs des variables associées aux états et des transitions qui leurs sont adjacentes. Aux opérateurs du calcul propositionnel sont associées des opérations élémentaires : à la négation correspond la complémentation, à la disjonction correspond l'union. Les opérateurs modaux sont associés à des plus petits points fixes de fonctions monotones et continues : par exemple, si on note $|f|$ l'ensemble des modèles de f, et $pre(|f|)$ l'ensemble de leurs prédécesseurs, l'ensemble des états qui satisfont la formule $f = pot[f_1]f_2$ (voir paragraphe 2.1) est le plus petit point fixe de l'équation :

$X = |f_2| \cup (|f_1| \cap pre(X))$.

Connaissant les résultats $|f_1|$ et $|f_2|$ de l'évaluation des fils de f, celui de l'évaluation de f peut donc être calculé par approximation successive en parcourant par niveaux le graphe à partir de $|f_2|$ en suivant le sens inverse des transitions :

$|f| := |f_2| ; Y := \varnothing ; Z := pre(|f|) ;$
tant que $Y \neq Z$ faire :
$|f| := |f| \cup (|f_1| \cap Z) ; Y := Z ; Z := pre(|f|)$

Par conséquent, un algorithme d'évaluation d'une formule f consiste à évaluer successivement ses sous-formules depuis les plus imbriquées jusqu'à f. Si le coût de l'accès aux prédécesseurs d'un état est constant, le coût de l'évaluation d'une formule de taille m sur un graphe de taille n est de l'ordre de $m \times n$.

- Diagnostic.

Les spécifications exprimant des propriétés d'une structure arborescente, le diagnostic d'erreur doit mettre en évidence les caractéristiques de cette structure qui sont erronées : il s'agit donc de donner non plus une simple trace d'exécution, mais un fragment de l'arbre d'exécution. En contrepartie de leur faible coût, les méthodes de vérification telles que celle décrite ci-dessus ne permettent pas d'obtenir directement ce diagnostic.

Conclusion.

Nous avons étudié différentes méthodes d'aide à la mise au point des programmes : nous avons distingué l'analyse d'une représentation du programme résultant de la compilation, qui permet de corriger les anomalies de la structure de contrôle, et l'analyse des exécutions du programme qui concernent les erreurs détectées aù cours de l'exécution. Les techniques mises en œuvre et la portée de ces méthodes ne sont pas comparables ; leur caractère

complémentaire a conduit à concevoir des outils mixtes combinant l'analyse du flot de contrôle et l'exécution symbolique.
Le système XESAR est basé sur l'analyse du graphe de contrôle. Ce graphe est entièrement généré ce qui autorise une vérification complète. Les différentes techniques envisagées dans ce cadre se distinguent par l'expression des spécifications. L'avantage des spécifications exprimées par des formules de logique linéaires ou par des automates est de permettre d'obtenir un diagnostic d'erreur au cours de la phase de détection. Les méthodes de vérification des propriétés exprimées dans une logique arborescente sont simples à mettre en œuvre. En contrepartie, le diagnostic d'erreur ne peut être obtenu au cours de cette phase. Or celui-ci est d'autant plus nécessaire pour l'utilisateur que les propriétés des arbres d'exécution sont plus difficiles à appréhender et moins intuitives que celles des séquences d'exécution. Nous définissons dans le chapitre suivant ce que peut être un diagnostic de la non satisfaction de propriétés exprimées par des formules *CTL*, et comment il peut être obtenu.

Chapitre 2

Explication des formules

Introduction

XESAR est un outil de vérification de systèmes parallèles communicants finis dont les spécifications sont exprimées par des formules *CTL*. Si une spécification n'est pas vérifiée, c'est-à-dire s'il y a une erreur, il est indispensable pour l'utilisateur d'en connaître la cause. Pour cela, la démarche habituelle consiste à déterminer de quelle manière l'erreur se produit : "comment est-il possible que telle spécification ne soit pas vérifiée ?".
Considérons une spécification, exprimée par une formule du temps arborescent : l'ensemble des modèles de cette formule est un ensemble d'arbres d'exécution. Or dans le graphe d'exécution du programme, un arbre d'exécution est complètement déterminé par sa racine. On associera donc simplement à une formule l'ensemble des racines de ses modèles dans le graphe, et on dira que les états de cet ensemble satisfont la formule considérée. Si une spécification représentée par la formule f, n'est pas vérifiée par le programme, il existe un ou plusieurs états du programme qui invalident f, c'est-à-dire qui satisfont la formule $g = \neg f$. Notre but est d'expliquer pourquoi f n'est pas valide en expliquant pourquoi un tel état, disons q, satisfait g : ceci se traduit généralement par l'existence dans l'arbre d'exécution issu de q, de certaines séquences d'exécution qui expliquent *comment* q satisfait g. Afin d'exprimer, à l'intention de l'utilisateur, des caractéristiques de ces séquences qui expliquent *pourquoi* telle spécification n'est pas vérifiée, les états de ces séquences sont étiquetées par les sous-formules de g qu'ils satisfont : en se référant à la sémantique des formules, l'utilisateur est alors en situation de comprendre les raisons pour lesquelles q satisfait g. En première approximation, une explication de ce que q satisfait g est donc une séquence d'exécution issue de q (dont l'existence prouve que q satisfait g), et dont les états sont

étiquetés par les sous-formules de g qu'ils satisfont. De même qu'on a expliqué pourquoi q satisfait g, il faut également montrer pourquoi ces états satisfont les sous-formules de g correspondantes : on est donc amené à associer aux états de la séquence de nouvelles séquences d'exécution. Aux sous-formules de g dont la satisfaction ne s'explique pas par l'existence de séquences d'exécutions particulières (telles que les prédicats sur les valeurs des variables du programme), on associera simplement une assertion traduisant le fait qu'elles sont satisfaites. La structure obtenue est appelée *explication* de "q satisfait g " : elle est constituée d'un sous-arbre de l'arbre d'exécution issu de q, dont les états sont étiquetés par des sous-formules de g.

Nous illustrerons les notions introduites au cours des chapitres 2 et 3 à l'aide de l'exemple suivant, choisi pour sa simplicité, et pour le grand nombre d'erreurs qu'il contient. Un exemple plus réaliste est présenté au cours du chapitre 5.

Exemple.
Un processus émetteur E et un processus récepteur R communiquent au moyen des canaux $L1$ et $L2$. Le canal $L1$ transmet à R les messages (mess) émis par E, le canal $L2$ transmet à E les accusés de réception (ack) émis par R. La figure 2.1 représente les automates associés à chaque composant.
Les canaux peuvent perdre les messages sans les transmettre, et le canal $L2$ peut se trouver hors service. Si R n'a pas reçu de nouveau message au bout d'un certain temps, il réémet son accusé de réception. Nous appellerons E/R ce système.

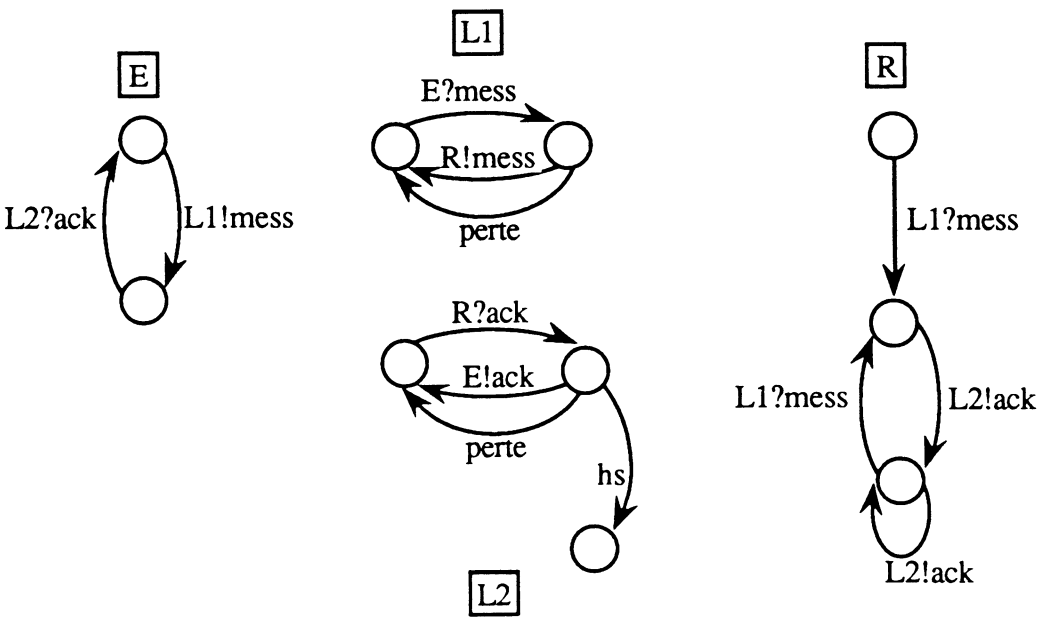


figure 2.1. Le système E/R.

Ce chapitre est organisé de la manière suivante :

• La syntaxe et la sémantique du langage des spécifications sont décrites dans le paragraphe 2.1.

• Le paragraphe 2.2 est consacré à la méthode de génération des explications, par récurrence sur la structure de la formule. A cet effet, les formules sont écrites sous une forme ad hoc, appelée *forme disjonctive*, qui permet cette récurrence. On étudie ensuite, pour les formules dont la satisfaction est prouvée par l'existence de certaines séquences d'exécution, l'ensemble de ces séquences, ou *séquences explicatives* : cet ensemble est muni d'une relation d'ordre, permettant de définir un sous-ensemble de *séquences explicatives minimales*.

• La définition des explications est donnée au paragraphe 2.3. Les explications sont construites par réécriture dans un système de règles défini à partir de séquences explicatives.

2.1. Langage des spécifications

Définitions.

Un *système de transitions étiqueté* (STE) est un quadruplet $(Q, \rightarrow, q_{init}, A)$ où :

Q est un ensemble d'états,

A un ensemble d'étiquettes,

$\rightarrow$ est un sous-ensemble de $Q \times A \times Q$ appelé *relation de transition*,

q_{init} est un élément de Q.

On dit que les éléments de A *étiquettent* les transitions. Pour tout α appartenant à A, on note $\rightarrow^{\alpha}$ l'ensemble $\{(q_1, q_2) / (q_1, \alpha, q_2) \in \rightarrow\}$; si $\alpha \neq \beta$, les ensembles $\rightarrow^{\alpha}$ et $\rightarrow^{\beta}$ sont disjoints.

On note : $q_1 \rightarrow^{\alpha} q_2$ si et seulement si $(q_1, q_2) \in \rightarrow^{\alpha}$.

Si $w = \alpha_1 \ldots \alpha_n$ appartient à A^*, on note $q_0 \rightarrow^{w} q_n$ si et seulement si $\exists\, q_1, q_2, \ldots q_{n-1}$ et $\forall\ 1 \leq i \leq n,\ q_{i-1} \rightarrow^{\alpha_i} q_i$.

Si (q_1, α, q_2) est une transition, on dit que q_2 est un *successeur* de q_1 et que q_1 est un *prédécesseur* de q_2. On note $succ(q)$ l'ensemble des successeurs de q, et $pred(q)$ l'ensemble de ses prédécesseurs.

q' est *accessible* à partir de q si et seulement si $\exists w \in A^*$ tel que $q \rightarrow^{w} q'$.

Un *programme* est représenté par un STE $(Q, \rightarrow, q_{init}, Act_{\lambda})$ où :

- Q est un sous-ensemble du produit cartésien des domaines des valeurs des variables du programme ; les programmes que l'on considère sont tels que leurs variables ne peuvent prendre qu'un nombre fini de valeurs : Q est donc fini.

- les éléments de Act sont des identificateurs du programme qui représentent des séquences d'instructions : les éléments de Act sont appelés *actions* ; le symbole λ n'appartient pas à Act, il étiquette les transitions auxquelles n'est associée aucune action du programme ; on note Act_λ l'ensemble $Act \cup \{\lambda\}$.
- q_{init} est appelé *état initial* du programme ; tout élément de Q est accessible à partir de q_{init}.

Une *séquence d'exécution* est une suite non vide d'états du programme s telle que :

$\forall i \in [1, longueur(s) - 1], s(i+1) \in succ(s(i))$

Elle est *issue* de q si $s(1) = q$.

Elle est *maximale*, si :

s est infinie

ou s est finie et son dernier élément n'a pas de successeur.

On note $Ex(q)$ l'ensemble des séquences d'exécution issues de q, et $Mex(q)$ celles d'entre elles qui sont maximales.

Un *arbre d'exécution* est un arbre dont les sommets sont étiquetés par des éléments de Q, et tel que si un sommet étiqueté par q_2 est un fils d'un sommet étiqueté q_1, alors q_2 est un successeur de q_1 dans le programme.

2.1.1. Syntaxe

L'ensemble $\mathcal{F}$ des formules logiques *CTL* est engendré à partir d'un ensemble fini $\mathcal{P}$ de *prédicats de base*, en utilisant des opérateurs du calcul propositionnel et des opérateurs temporels.

Les éléments de $\mathcal{P}$ sont :

- des expressions booléennes sur les variables du programme : par exemple, "$x = y$",
- des prédicats de la forme $enable(\alpha)$ et $after(\alpha)$ où α est un élément de Act,
- le prédicat *vrai*,
- le prédicat *init*
- le prédicat *puits* .

Si $\mathcal{P} = \{P_1,\ldots,P_n\}$, $\mathcal{F}$ est engendré par la grammaire :

$f ::= f \vee f \mid g$

$g ::= g \wedge g \mid h$

$h ::= (f) \mid \neg h \mid pot[f]h \mid some[f]h \mid al[f]h \mid inev[f]h \mid P_1 \mid \ldots \mid P_n$

2.1.2. Sémantique

Aux états du programme on associe par une fonction notée Π appelée interprétation l'ensemble des prédicats de base qu'ils satisfont :

$\forall\, q \in Q$,

- si P est une expression booléenne, $P \in \Pi(q)$ si et seulement si P est vrai pour les valeurs des variables à l'état q
- $enable(\alpha) \in \Pi(q)$ si et seulement si $\exists\, q' \in Q$ et $q \rightarrow^{\alpha} q'$
- $after(\alpha) \in \Pi(q)$ si et seulement si[1] $\forall\, q' \in pred(q),\ q' \rightarrow^{\alpha} q$
- $vrai \in \Pi(q)$
- $init \in \Pi(q)$ si et seulement si $q = q_{init}$
- $puits \in \Pi(q)$ si et seulement si $succ(q) = \varnothing$.

Etant donné un programme $(Q,\ \rightarrow, q_{init}, Act_{\lambda})$, un ensemble de prédicats $\mathcal{P}$ et une fonction Π, les modèles de *CTL* sont les arbres d'exécution du programme. On représente l'arbre d'exécution issu de q (ou modèle d'état initial q) par la structure $(Q, \rightarrow, q, Act_{\lambda}, \mathcal{P}, \Pi)$.

La sémantique des formules est définie par la relation de satisfaction $\models$ entre éléments de Q et éléments de $\mathcal{F}$: $(Q, \rightarrow, q, Act_{\lambda}, \mathcal{P}, \Pi)$ est un *modèle* de f si et seulement si $q \models f$. Pour tout état q, la relation de satisfaction est définie de la manière suivante :

$\forall\, P \in \mathcal{P}$:

$q \models P \quad \Leftrightarrow \quad P \in \Pi(q)$

$\forall\, f, f_1, f_2 \in \mathcal{F}$:

$q \models f_1 \vee f_2 \quad \Leftrightarrow \quad (q \models f_1$ ou $q \models f_2)$

$q \models f_1 \wedge f_2 \quad \Leftrightarrow \quad (q \models f_1$ et $q \models f_2)$

$q \models \neg f \quad \Leftrightarrow \quad q \not\models f$

$q \models (f) \quad \Leftrightarrow \quad q \models f$

$q \models pot[f_1]f_2 \quad \Leftrightarrow \quad (\exists s \in Ex(q)$, $\exists k \geq 1$ tels que : $(\forall i,\ 1 \leq i < k \Rightarrow s(i) \models f_1)$ et $s(k) \models f_2)$

$q \models some[f_1]f_2 \quad \Leftrightarrow \quad (\exists s \in Ex(q),\ \forall k \geq 1 : (\forall i,\ 1 \leq i < k \Rightarrow s(i) \models f_1) \Rightarrow\ s(k) \models f_2)$

---

[1] Un état satisfait $after(\alpha)$ si et seulement si il est atteint par exécution de α et de α seulement. Le choix de cette sémantique permet d'exprimer par une formule l'inévitabilité de l'occurrence d'une action : l'action α est inévitable si et seulement si le prédicat $after(\alpha)$ est inévitable. Pour évaluer $after(\alpha)$, il est donc nécessaire de distinguer les états atteints par α et les états atteints par des transitions différentes. Or, lors de la génération du STE, les états caractérisés par les mêmes valeurs des variables du programme mais atteints par des transitions différentes sont identifiés, dans le but de réduire l'explosion combinatoire du nombre d'états. Pour évaluer le prédicat $after(\alpha)$, on est amené à modifier localement le STE en duplicant les états atteints par α et par une transition différente de α. Pour plus de détails sur cette transformation, on peut se reporter à [Ro88].

$q \models inev[f_1]f_2 \quad \Leftrightarrow (\forall s \in Mex(q), \exists k \geq 1$ tel que : $(\forall i, 1 \leq i < k \Rightarrow s(i) \models f_1)$ et $s(k) \models f_2)$

$q \models al[f_1]f_2 \quad \Leftrightarrow (\forall s \in Mex(q), \forall k \geq 1 : (\forall i, 1 \leq i < k \Rightarrow s(i) \models f_1) \Rightarrow s(k) \models f_2)$

Les opérateurs temporels peuvent s'interpréter intuitivement de la manière suivante :

- *pot* (possible) : $q \models pot[f_1]f_2$ signifie qu'il est *possible* d'atteindre à partir de q, sous la condition f_1, un état satisfaisant f_2.
- *inev* (inévitable) : $q \models inev[f_1]f_2$ signifie qu'il est *inévitable* d'atteindre à partir de q, sous la condition f_1, un état satisfaisant f_2.

al (always) et *some* sont les opérateurs duaux des précédents. $q \models al[f_1]f_2$ signifie que f_2 reste toujours vraie sous la condition f_1. La signification de $q \models some[f_1]f_2$ est moins intuitive : nous y reviendrons en détail au paragraphe 2.2.4.1.

On appelle *assertions* les éléments de $\models$, et pour toute formule f, on note $|f|$ l'ensemble $\{q \,/\, q \models f\}$.
Une formule f est *satisfaisable* si et seulement si il existe un modèle de f. Une formule est *valide* si et seulement si tous les modèles sont des modèles de f.

Les formules f et g sont *équivalentes* si et seulement si, quel que soit $\mathcal{P}$ et Π, tout modèle de f est un modèle de g et tout modèle de g est un modèle de f. Si f est g sont équivalentes, on note $f \Leftrightarrow g$. On déduit en particulier de la définition de la relation de satisfaction les équivalences suivantes :

$pot[f]g \quad \Leftrightarrow \neg al[f]\neg g$

$some[f]g \Leftrightarrow \neg inev[f]\neg g$.

Dans la suite, nous utiliserons les abréviations suivantes :

$f \Rightarrow g = \neg f \vee g$

$op\, f = op[vrai]f$, $(op = al, inev, pot, some)$.

2.2. Méthode de génération des explications

2.2.1. Exemple.

Le système *E/R* est modélisé par un STE obtenu en composant les automates représentés sur la figure 2.1. Les noms d'actions sont indiqués de manière abrégée, et ont la signification suivante (rappelons que les communications se font par rendez-vous) :

e_em : envoi d'un message de *E* à *L*1 (et réception de ce message par *L*1)
e_rec : réception par *E* de l'accusé de réception émis par *L*2
r_em : envoi d'un accusé de réception de *R* à *L*2
r_rec : réception par *R* d'un message émis par *L*1
*p*1 (resp. *p*2) : perte d'un message par *L*1 (resp. *L*2)
*hs*2 : mise hors service de *L*2.

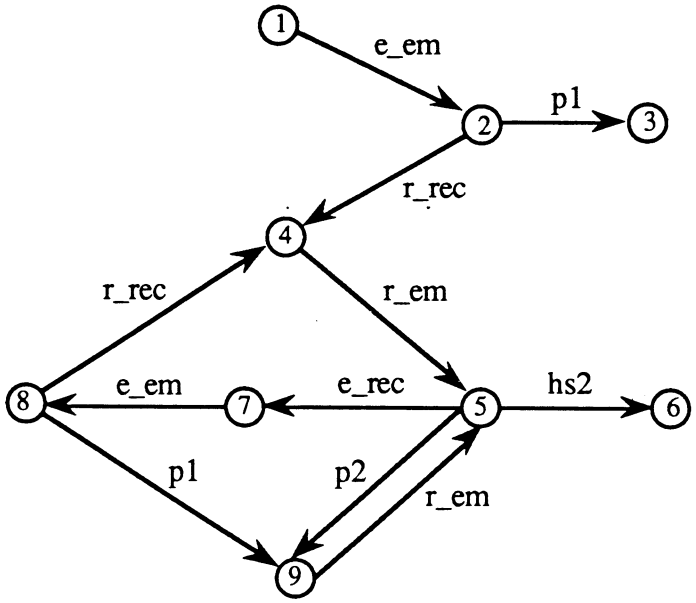


figure 2.2. Modèle généré pour le système E/R.

Soit la spécification suivante : "après toute émission d'un message par *E*, il est inévitable que *R* reçoive un message". Cette spécification n'est pas remplie par le système, en effet :

*après émission par E du premier message, si L*1 *perd ce message, le système ne peut plus évoluer dans un état différent (donc il n'est pas inévitable que R reçoive le message).*

Traduisons ceci formellement :

- la spécification peut s'exprimer par la formule *f* :

$after(e\_em) \Rightarrow inev\ after(r\_rec)$.

- *f* n'est pas valide, car tous les états ne satisfont pas cette formule : $2 \not\models f$ et $8 \not\models f$. Pour expliquer pourquoi *f* n'est pas valide, on explique par exemple l'assertion $2 \models \neg f$.

$\neg f$ peut s'exprimer par la conjonction de deux formules : $after(e\_em) \wedge some\ \neg after(r\_rec)$. Ainsi, $2 \models \neg f$ parce que $2 \models after(e\_em)$ et $2 \models some\ \neg after(r\_rec)$.

- L'explication de $2 \models after(e\_em)$ est triviale : l'état 2 est atteint après émission d'un message par *E*.

- L'explication de $2 \models some\ \neg after(r\_rec)$ est construite à partir d'un exemple de séquence d'exécution issue de l'état 2, le long de laquelle *R* ne reçoit jamais de message : la séquence d'exécution maximale 2.3.

En conclusion, une explication de $2 \models \neg f$ est la suivante :

$2 \models \neg f$ parce que

$2 \models after(e\_em)$

et

$2 \models some\ \neg after(r\_rec)$.

($2 \models after(e\_em)$ parce que 2 est atteint après émission d'un message par E)

$2 \models some\ \neg after(r\_rec)$ parce que

$2 \rightarrow^{p1} 3$

et

$2 \models \neg after(r\_rec)$

et

$3 \models \neg after(r\_rec)$ et $3 \models puits$

($2 \models \neg after(r\_rec)$ parce que 2 n'est pas atteint après réception d'un message par R)

($3 \models \neg after(r\_rec)$ parce que 3 n'est pas atteint après réception d'un message par R)

($3 \models puits$ parce que 3 n'a pas de successeur).

2.2.2. Principe

Il s'agit d'associer une explication à l'assertion $q \models f$: or, d'une manière générale, l'état q satisfait la formule f parce que des séquences d'exécution issues de q conduisent à des états qui vérifient des sous-formules de f. Ainsi, la structure des formules joue un rôle prépondérant dans la construction des explications : l'explication d'une formule sera définie par récurrence à partir des explications de ses sous-formules. On est ainsi amené à distinguer les *formules de base* auxquelles des explications peuvent être associées directement, et les autres formules, pour lesquelles une décomposition récurrente est nécessaire à l'obtention d'une explication. Devant utiliser la structure des formules, nous définissons un sous-ensemble de celles-ci, les formules sous forme disjonctive : la syntaxe en est plus simple, et pour toute formule, il existe une formule équivalente sous forme disjonctive.

2.2.2.1. Formules de base, formules temporelles

Les propriétés exprimées par un prédicat de base concernent un état, et dans le cas des prédicats de la forme $enable(\alpha)$ et $after(\alpha)$, les transitions adjacentes à cet état. Ces propriétés sont indépendantes des exécutions possibles du programme à partir de cet état : les assertions $q \models P$, où P est un prédicat de base, ne nécessitent donc pas d'explication, autrement dit l'assertion $q \models P$ est sa propre explication.

Les formules $al[f_1]f_2$ et $inev[f_1]f_2$ sont satisfaites par un état q parce que toutes les séquences d'exécution issues de q ont certaines propriétés. Il n'est pas réaliste de vouloir expliquer la satisfaction d'une formule en exhibant une telle information que sa taille rend inutilisable pour un usager. Ne pouvant pas donner une explication complète de $q \models al[f_1]f_2$ ni de $q \models inev[f_1]f_2$, on pourrait, à l'aide d'une ou de quelques séquences d'exécution arbitraires, tenter d'en donner tout au plus une illustration. Ceci est de nature différente de ce qui nous intéresse ici. On ne fournit donc pas d'explication de la satisfaction de $al[f_1]f_2$ et de $inev[f_1]f_2$: les assertions $q \models al[f_1]f_2$ et $q \models inev[f_1]f_2$ seront leur propre explication.

Par définition, les *formules de base* sont : les prédicats de base et les formules équivalentes à une formule de la forme $al[f_1]f_2$ et $inev[f_1]f_2$. Par convention, la notation *fb* désignera une formule de base quelconque. On réservera le terme *formule temporelle* aux autres formules, qui sont de la forme $pot[f_1]f_2$ ou $some[f_1]f_2$ (ou équivalentes).

Justification du choix des formules de base.

En nous limitant à l'explication de la satisfaisabilité, en termes de séquences d'exécution, des formules $pot[f_1]f_2$ ou $some[f_1]f_2$, nous ne réduisons pas l'intérêt de notre démarche : nous traitons ainsi tous les cas intéressants (où il est utile d'obtenir une explication) et uniquement ceux-ci (figure 2.3). En effet :

- dans l'approche diagnostic où a été conçue Cléo, soit f une formule *non valide* exprimant les spécifications :

  - si f est de la forme $al[f_1]f_2$ ou $inev[f_1]f_2$, un diagnostic de la satisfaction de $\neg f$ est fourni. La plupart des propriétés usuellement exprimées dans les spécifications le sont au moyen de formules de la forme $al[f_1]f_2$ et $inev[f_1]f_2$: en effet, il est généralement admis que toute spécification d'un système est exprimable en termes de propriétés de sûreté et de vivacité. Intuitivement, une propriété de sûreté signifie qu'une certaine erreur (c'est-à-dire une action non prévue sous certaines conditions) ne peut jamais se produire, ce qui s'exprime par la formule $\neg pot$(erreur), c'est-à-dire $al(\neg$erreur) ; une propriété de vivacité signifie qu'il se produira inévitablement un événement souhaité, ce qui s'exprime par la formule *inev*(événement). La non validité de ces propriétés s'exprime par une formule de type *pot* ou *some*. L'explication de la non validité de telles spécifications consiste à montrer des séquences d'exécution qui conduisent à une erreur dans le premier cas, où le long desquelles l'événement souhaité ne se produit jamais dans le second, et à exprimer pourquoi ces séquences ont cette propriété.
  - si f est de la forme $pot[f_1]f_2$ (ou $some[f_1]f_2$), un diagnostic serait formé de toutes les séquences d'exécution issues d'un état qui satisfait $\neg f$: un tel diagnostic est pratiquement

inexploitable. Pour de telles spécifications, des diagnostics significatifs peuvent être obtenus en précisant les spécifications : ainsi, s'il n'est pas possible qu'un événement se produise, on cherchera plutôt à savoir pourquoi l'occurrence de cet événement n'est pas inévitable dans les conditions où il devrait normalement se produire.

• selon l'approche génération de séquences de tests, considérons une formule *f*, *valide*, exprimant une propriété que l'on veut vérifier :

- si *f* est de la forme $al[f_1]f_2$ ou $inev[f_1]f_2$, *f* est vraie sur le modèle parce que toutes ses séquences d'exécution ont une certaine propriété. Par conséquent, pour une telle propriété, toutes les séquences sont des séquences de test.
- si *f* est de la forme $pot[f_1]f_2$ (ou $some[f_1]f_2$), le diagnostic fourni par Cléo permet de vérifier l'existence et de construire des séquences qui expliquent pourquoi *f* est valide pour le modèle.

Spécifications	Non Valides	Valides
al inev	**Diagnostic** (= Explication de pot ou de some)	Toutes les séquences sont des séquences de test.
pot some	Pas de diagnostic intéressant	**Séquences de test** (= Explication de pot ou de some)

figure 2.3

2.2.2.2. Forme disjonctive d'une formule.

Une formule sous forme disjonctive est de la forme $\bigvee_i \bigwedge_j (f_{i,j})$ où les $f_{i,j}$ sont des formules de base, ou des formules temporelles dont les sous-formules sont sous forme disjonctive. L'ensemble $\mathcal{D}$ des formules sous forme disjonctive est le sous-ensemble de $\mathcal{F}$ engendré par la grammaire :

$f ::= d \mid e$
$d ::= d \vee e \mid e \vee e$
$e ::= c \mid g$
$c ::= c \wedge g \mid g \wedge g$
$g ::= pot[f]\ k \mid some[f]\ k \mid al[f]\ k \mid inev[f]\ k \mid h \mid \neg h$

$k ::= (d) \mid (c) \mid g$
$h ::= P_1 \mid \ldots \mid P_n$

A toute formule f de $\mathcal{F}$ on associe une formule équivalente $fd(f)$ de $\mathcal{D}$ appelée forme disjonctive de f. La transformation permettant de passer de f à $fd(f)$ est une adaptation de la transformation classique de mise sous forme normale disjonctive des expressions du calcul propositionnel. $fd(f)$ est obtenue :

1) en propageant les opérateurs de négation au niveau des prédicats de base,
2) puis en distribuant les conjonctions sur les disjonctions,
3) et enfin, en simplifiant l'écriture de la formule en ne conservant que les parenthèses nécessaires à l'évaluation d'opérateurs avant celle d'opérateurs plus prioritaires.

Exemple.
Les P_i, i=1...5 appartenant à $\mathcal{P}$, pour la formule :

$f = \neg(\neg pot[\neg(P_1 \vee P_2)](al[vrai]P_3) \vee \neg(P_4 \vee P_5))$

on obtient

- après propagation des opérateurs de négation au niveau des prédicats de base :

$(pot[(\neg P_1 \wedge \neg P_2)](al[vrai]P_3) \wedge (P_4 \vee P_5))$

- après distribution des opérateurs de conjonction sur les opérateurs de disjonction :

$((pot[(\neg P_1 \wedge \neg P_2)](al[vrai]P_3) \wedge P_4) \vee (pot[(\neg P_1 \wedge \neg P_2)](al[vrai]P_3) \wedge P_5))$

- en éliminant les parenthèses inutiles :

$fd(f) = pot[\neg P_1 \wedge \neg P_2]al[vrai]P_3 \wedge P_4 \vee pot[\neg P_1 \wedge \neg P_2]al[vrai]P_3 \wedge P_5$

Propriété 2.1. $\forall f \in \mathcal{F}, fd(f) \in \mathcal{D}$ et $fd(f) \Leftrightarrow f$

Nous admettrons que si deux formules sont équivalentes, toute explication de la satisfaisabilité de l'une est une explication de la satisfaisabilité de l'autre. On appelle désormais formule une classe d'équivalence de formules, et on choisit comme représentant de la classe, une formule sous forme disjonctive, sachant qu'il en existe toujours au moins une.

Pour définir les explications des formules sous forme disjonctive, on raisonne par récurrence sur la structure des formules :

- on explique qu'un état q satisfait une formule de la forme $\bigvee_i \bigwedge_j (f_{i,j})$ en montrant que pour un certain i, cet état satisfait les $f_{i,j}$,
- on explique qu'un état q satisfait une formule temporelle f au moyen d'une séquence d'exécution issue de q, le long de laquelle sont satisfaites des sous-formules de f.

- on explique qu'un état q satisfait une formule de base fb au moyen de l'assertion $q \models fb$.

C'est donc à partir des séquences permettant d'expliquer la satisfaisabilité des formules temporelles que seront construites les explications. La suite du paragraphe 2.2 est consacré à ces séquences, que nous nommons séquences explicatives : dans les paragraphes 2.2.3 et 2.2.4, nous considérons les ensembles de séquences explicatives d'une formule temporelle de la forme $pot[f_1]f_2$ et $some[f_1]f_2$. Dans chacun de ces ensembles, nous définissons une relation d'ordre partiel : cette relation d'ordre est liée au nombre de circuits élémentaires du graphe apparaissant dans les séquences ; de plus, pour les séquences finies, elle fait intervenir leur longueur. On caractérise alors un sous-ensemble fini de séquences minimales, tel que pour toute séquence s montrant que la formule est satisfaite, il existe dans ce sous-ensemble une séquence comparable à s permettant aussi de montrer que la formule est satisfaite.
On donne ensuite au paragraphe 2.3 la définition des explications et leur construction à partir des séquences explicatives des formules temporelles.

2.2.3. Séquences explicatives de l'assertion $q \models pot[f_1]f_2$

2.2.3.1. Définition et propriétés

Par définition de l'opérateur *pot*, un état q satisfait $pot[f_1]f_2$ si et seulement si il existe une séquence d'exécution $q_1 \ldots q_i \ldots$ et un entier $k>0$ tels que $q_1=q$ et $(i<k \Rightarrow q_i \models f_1)$ et $q_k \models f_2$ (figure 2.4).

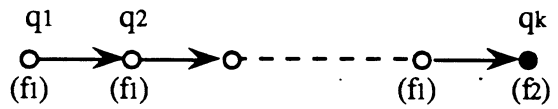


figure 2.4. $q_1 \models pot[f_1]f_2$.

Soit $S(q \models pot[f_1]f_2)$ l'ensemble des préfixes de telles séquences dont les états intermédiaires satisfont f_1 et dont le dernier état satisfait f_2 (on rappelle qu'un état intermédiaire d'une suite finie désigne n'importe quel état de cette suite sauf le dernier) :

$$S(q \models pot[f_1]f_2) = \{q_1 \ldots q_k \in Ex(q) \,/ (1 \leq i < k \Rightarrow q_i \models f_1) \text{ et } q_k \models f_2\}.$$

Un élément quelconque de $S(q \models pot[f_1]f_2)$ montre comment q satisfait $pot[f_1]f_2$. On exprimera ceci plus succinctement : les éléments de $S(q \models pot[f_1]f_2)$ permettent d'atteindre un but f_2 sous la condition f_1.

Il n'y a pas d'autre exigence sur les états intermédiaires : en particulier, la formule f_2 peut être satisfaite avant le dernier état de la séquence, et il est possible de passer plusieurs fois par le même état. Cependant, si la formule f_2 est satisfaite par un état intermédiaire, cela signifie que le but est déjà atteint : la fin de la séquence n'apporte pas d'argument utile à la preuve de la satisfaction de $pot[f_1]f_2$. De même, si deux états de la séquence sont identiques, il n'est pas nécessaire de considérer les états situés entre les deux occurrences pour démontrer la possibilité d'atteindre f_2. Rappelons que par définition, une séquence s' est un circuit d'une séquence s s'il existe deux séquences s_1 et s_2 telles que $s = s_1.s'.s_2$ et $dernier(s') = dernier(s_1)$.

Exemples.

- Si les états 1 à 6 satisfont f_1, et si l'état 7 satisfait f_2, la séquence 1.2.3.4.5.6.7 appartient à $S(1 \models pot[f_1]f_2)$.

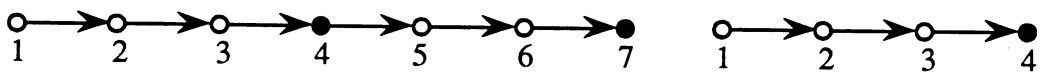


figure 2.5

Si de plus l'état 4 satisfait f_2, la séquence 1.2.3.4 suffit à prouver que l'état 1 satisfait $pot[f_1]f_2$: le fait que le suffixe 5.6.7 de la séquence 1.2.3.4.5.6.7 permette d'atteindre le but f_2 sous la condition f_1 est une propriété complémentaire, dont on considérera qu'elle n'apporte rien à l'explication de $1 \models pot[f_1]f_2$.

- Si 1.2.3.4.8.9.4.5.6.7 appartient à $S(1 \models pot[f_1]f_2)$, alors, puisque 4 est un successeur de 3 et un prédécesseur de 5, 1.2.3.4.5.6.7 est aussi le préfixe d'une séquence d'exécution, et suffit à prouver que 1 satisfait $pot[f_1]f_2$.

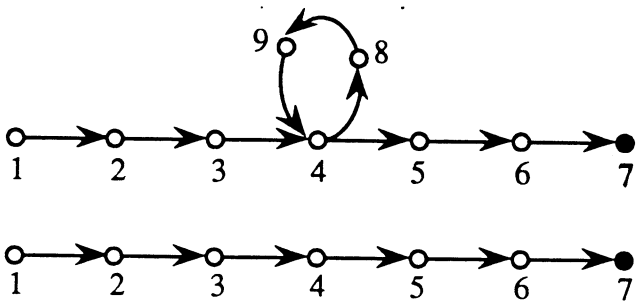


figure 2.6

La présence du circuit 8.9.4 de la séquence 1.2.3.4.8.9.4.5.6.7 n'apporte aucun argument à la preuve de $1 \models pot[f_1]f_2$.

- Considérons le système *E/R* (figure 2.2). Soit la spécification suivante : "après que *E* ait émis un message, *E* ne peut pas recevoir d'accusé de réception sans que *R* ait reçu un message", traduite par la formule :

$after(e\_em) \Rightarrow \neg pot[\neg after(r\_rec)]\ enable(e\_rec)$.

(cette formule est la traduction *CTL* de la méta-formule *not e_em to e_rec unless r_rec*)
Cette formule n'est pas valide car un état au moins, l'état 8, satisfait sa négation : $after(e\_em) \wedge pot[\neg after(r\_rec)]\ enable(e\_rec)$.
On a donc en particulier : $8 \models pot[\neg after(r\_rec)]\ enable(e\_rec)$. La séquence 8.9.5.9.5 appartient à $S(8 \models pot[\neg after(r\_rec)]\ enable(e\_rec))$: en effet, les états 8, 9, 5 et 9 satisfont $\neg after(r\_rec)$, et l'état 5 satisfait $enable(e\_rec)$. Cependant, la séquence 8.9.5 suffit à prouver que 8 satisfait $pot[\neg after(r\_rec)]\ enable(e\_rec)$.

Définition. L'ensemble $S(q \models pot[f_1]f_2)$ est appelé ensemble des séquences explicatives de l'assertion $q \models pot[f_1]f_2$.

Propriétés 2.2. $S(q \models pot[f_1]f_2)$ est un ensemble non vide si et seulement si $q \models pot[f_1]f_2$. Ses éléments sont des séquences finies telles que
- tout préfixe d'un élément de $S(q \models pot[f_1]f_2)$ dont le dernier état satisfait f_2 appartient à $S(q \models pot[f_1]f_2)$:

$(q_1 \ldots q_k \in S(q \models pot[f_1]f_2)$ et $\exists\ i<k$ tel que $q_i \models f_2)$
$\Rightarrow q_1 \ldots q_i \in S(q \models pot[f_1]f_2)$.

- toute séquence obtenue à partir d'un élément de $S(q \models pot[f_1]f_2)$ par suppression d'un circuit appartient à $S(q \models pot[f_1]f_2)$:

$(q_1 \ldots q_k \in S(q \models pot[f_1]f_2)$ et $\exists\ i<j\leq k$ tel que $q_i = q_j)$
$\Rightarrow q_1 \ldots q_i.q_{j+1} \ldots q_k \in S(q \models pot[f_1]f_2)$.

2.2.3.2. Séquences d'accès à un but

Certaines séquences de $S(q \models pot[f_1]f_2)$ ne comportent pas de circuit et sont telles qu'aucun de leurs préfixes stricts n'appartient à $S(q \models pot[f_1]f_2)$, c'est-à-dire telles que le dernier état de ces séquences est le seul qui satisfasse f_2. Comme nous l'avons illustré dans les exemples du paragraphe précédent, ce sont des séquences particulières que nous allons caractériser et étudier.
On dit qu'une séquence finie *s* est une *séquence d'accès* à un ensemble d'états *B*, le but, si le dernier état de *s* appartient à *B* et aucun état intermédiaire de *s* n'appartient à *B*.
Une séquence d'accès à *B* est acyclique si elle est sans circuit. L'ensemble des séquences acycliques d'accès à *B* est noté *saa*(*B*) :

Définition. Si $B \subset Q$,

$saa(B) = \{q_1....q_k \in Q^* / \forall i, j \leq k, (q_i=q_j \Rightarrow i=j)$ et $(q_i \in B \Leftrightarrow i=k)$ et $q_k \in B\}$.

Nous ne nous intéresserons dans la suite qu'à celles des séquences d'accès qui sont acycliques : par abus de langage, nous les appellerons simplement séquences d'accès.

Propriété 2.3. Si Q est fini, $saa(B)$ est un ensemble fini.

Démonstration : immédiate.

Si $S(q \models pot[f_1]f_2)$ n'est pas vide, il contient, d'après les propriétés 2.2, des séquences d'accès à l'ensemble des états qui satisfont f_2. Nous allons définir une relation d'ordre partiel sur $S(q \models pot[f_1]f_2)$ tel que :

- ces séquences sont les éléments minimaux de $S(q \models pot[f_1]f_2)$ pour cette relation d'ordre,
- chaque élément de $S(q \models pot[f_1]f_2)$ a (au moins) un minorant minimal.

2.2.3.3. Relation d'ordre dans Q^*

Définition. La relation $\leq_{fi}$ est la fermeture transitive de la relation $\leq_{fi}^0$ définie sur Q^* par :
$\forall\ s_1, s_2 \in Q^*,\ s_1 \leq_{fi}^0 s_2 \Leftrightarrow$

$(\exists s' \in Q^* / s_2 = s_1.s')$

ou

$(\exists s'_1, s'_2, s'_3 \in Q^* \setminus \{\varepsilon\} / dernier(s'_1) = dernier(s'_2)$ et $s_2 = s'_1.s'_2.s'_3$ et $s_1 = s'_1.s'_3)$.

Autrement dit, $s_1 \leq_{fi}^0 s_2$ si et seulement si s_1 est un préfixe de s_2, ou s_1 est obtenue à partir de s_2 par suppression d'un circuit.

Propriété 2.4. La relation $\leq_{fi}$ est une relation d'ordre partiel dans Q^*.

Démonstration

$\leq_{fi}$ est évidemment réflexive et transitive, vérifions qu'elle est antisymétrique.

Soit s et s' appartenant à Q^* telles que : $s \leq_{fi} s'$ et $s' \leq_{fi} s$.

On a : $s \leq_{fi} s' \Rightarrow (\exists s_0, \ldots, s_n \in Q^*$ tels que $s_0 = s$, $s_n = s'$, et $(0 \leq i < n \Rightarrow s_i \leq_{fi}^0 s_{i+1}))$. De même, $s' \leq_{fi} s \Rightarrow (\exists s'_0, \ldots, s'_m \in Q^*$ tels que $s'_0 = s'$, $s'_m = s$, et $(0 \leq i < m \Rightarrow s'_i \leq_{fi}^0 s'_{i+1}))$.

Or, $s_i \leq_{fi}^0 s_{i+1} \Rightarrow longueur(s_i) \leq longueur(s_{i+1})$ et

$s'_i \leq_{fi}^0 s'_{i+1} \Rightarrow longueur(s'_i) \leq longueur(s'_{i+1})$.

Par conséquent : $longueur(s) = longueur(s')$, donc, $\forall i, j$, $longueur(s_i) = longueur(s_j)$

Enfin, $(s_i \leq_{fi}^0 s_{i+1}$ et $longueur(s_i) = longueur(s_{i+1})) \Rightarrow s_i = s_{i+1}$,

donc $s = s'$.

$\therefore$

Propriété 2.5. $\forall\ s, s' \in Q^*,\ s \leq_{fi} s' \Rightarrow longueur(s) \leq longueur(s')$.

2.2.3.4. Séquences explicatives minimales de l'assertion $q \models pot[f_1]f_2$

Définition. Soit $S^0(q \models pot[f_1]f_2)$ l'ensemble des séquences explicatives de $q \models pot[f_1]f_2$ qui sont séquences d'accès à l'ensemble des états satisfaisant f_2, soit :

$$S^0(q \models pot[f_1]f_2) = S(q \models pot[f_1]f_2) \cap saa(|f_2|).$$

Propriétés 2.6.

- $S^0(q \models pot[f_1]f_2)$ est un ensemble fini.
- les éléments de $S^0(q \models pot[f_1]f_2)$ sont les éléments minimaux de $S(q \models pot[f_1]f_2)$ pour la relation d'ordre $\leq_{fi}$:

$$s \in S^0(q \models pot[f_1]f_2) \Leftrightarrow$$
$$[s \in S(q \models pot[f_1]f_2) \text{ et } (\forall\ s' \in S(q \models pot[f_1]f_2) : (s' \leq_{fi} s \Rightarrow s' = s))]$$

- chaque élément de $S(q \models pot[f_1]f_2)$ a un minorant minimal dans $S^0(q \models pot[f_1]f_2)$:

$$\forall\ s \in S(q \models pot[f_1]f_2) : \exists\ s' \in S^0(q \models pot[f_1]f_2) \text{ tel que } s' \leq_{fi} s.$$

Démonstration.

- Q étant fini, $saa(|f_2|)$ est un ensemble fini, par conséquent, il en est de même de $S^0(q \models pot[f_1]f_2)$.
- Que les éléments de $S^0(q \models pot[f_1]f_2)$ soient minimaux découle de leur définition : ils ne comportent pas de circuit, et aucun de leurs préfixes stricts n'appartient à $S(q \models pot[f_1]f_2)$.

Réciproquement, un élément minimal s de $S(q \models pot[f_1]f_2)$ ne comporte pas de circuit, et aucun de ses préfixes stricts n'appartient à $S(q \models pot[f_1]f_2)$. Autrement dit :

s est sans circuit,

et $\{p \in états(s) / p \models f_2\} = \{dernier(s)\}$,

c'est-à-dire $s \in saa(|f_2|)$, donc $s \in S^0(q \models pot[f_1]f_2)$.

- Prouvons que tout élément de $S(q \models pot[f_1]f_2)$ a au moins un minorant dans $S^0(q \models pot[f_1]f_2)$:

  - les éléments minimaux sont leur propre minorant
  - à un élément non minimal on peut associer une suite finie décroissante d'éléments distincts dont le dernier est minimal : d'après les propriétés de $S(q \models pot[f_1]f_2)$, tout élément non minimal possède un suffixe de longueur non nulle ou un circuit, tel qu'on obtient un élément de $S(q \models pot[f_1]f_2)$ en "extrayant" ce suffixe ou ce circuit, c'est-à-dire qu'une séquence non minimale possède au moins un minorant de longueur strictement

inférieure. On peut donc construire une suite de séquences de longueurs strictement décroissantes telles que chaque séquence est un minorant de la précédente : une telle suite est donc finie ; le dernier élément est une séquence sans circuit et dont aucun préfixe strict n'est dans $S(q \models pot[f_1]f_2)$, c'est-à-dire qu'il appartient à $S^0(q \models pot[f_1]f_2)$.

$\therefore$

Exemple.

Considérons le STE représenté par la figure 2.7 :

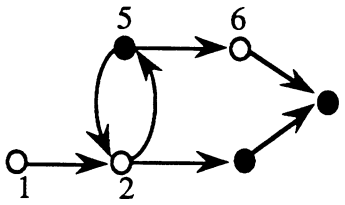


figure 2.7

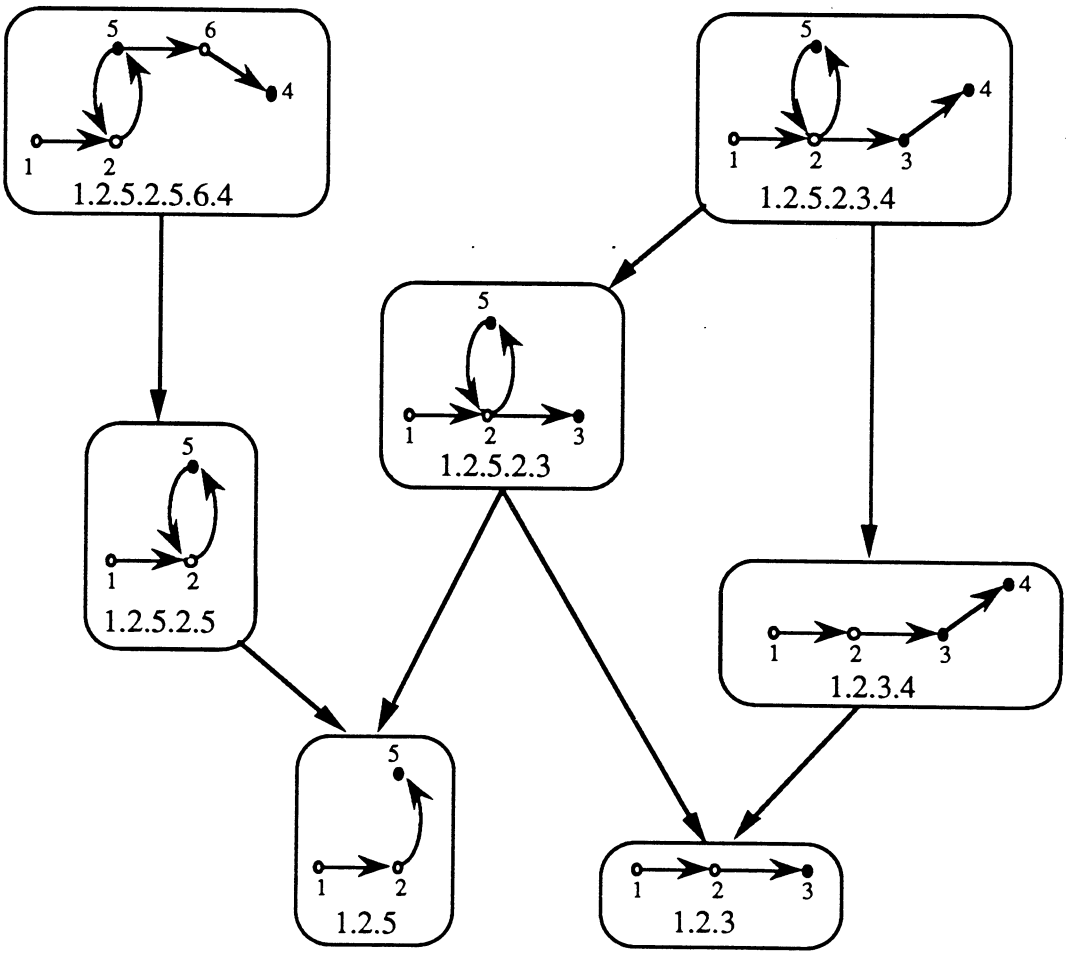


figure 2.8

Deux formules f_1 et f_2 sont satisfaites respectivement par tous les états et par les états 3, 4 et 5. L'état 1 satisfait $pot[f_1]f_2$. La figure 2.8 est une représentation d'un sous-ensemble de $S(1 \models pot[f_1]f_2)$ et de la relation d'ordre. Les flèches indiquent la décroissance stricte de relations $\leq_{fi}^0$. $S(1 \models pot[f_1]f_2)$ a deux éléments minimaux : 1.2.5 et 1.2.3. Chaque élément de $S(1 \models pot[f_1]f_2)$ est plus grand que l'un au moins de ces éléments.

2.2.4. Séquences explicatives de l'assertion $q \models some[f_1]f_2$

2.2.4.1. Définition

La formule $some[f_1]f_2$ est équivalente à la négation de la formule $inev[f_1]\neg f_2$. Considérons pour commencer un état q qui satisfait $inev[f_1]\neg f_2$: alors le long de toute séquence d'exécution maximale $q_1 \ldots q_i \ldots$ telle que $q_1 = q$, il existe $k \geq 0$ tel que $(i<k \Rightarrow q_i \models f_1)$ et $q_k \models \neg f_2$ (figure 2.9).

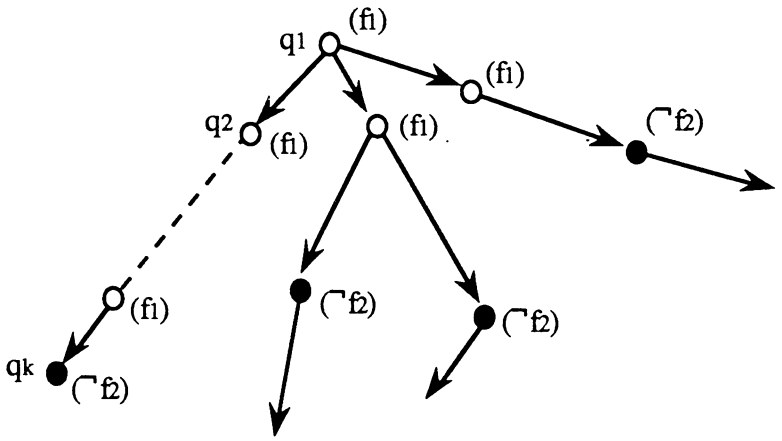


figure 2.9. $q_1 \models inev\ [f_1]\ \neg f_2$.

Autrement dit, quelle que soit la séquence d'exécution suivie, le but $\neg f_2$ sera atteint en un nombre fini de pas, la condition f_1 étant satisfaite par tous les états intermédiaires : comme son nom l'indique, cette propriété exprime l'inévitabilité d'atteindre $\neg f_2$ sous la condition f_1. Au contraire, si un état q satisfait $some[f_1]f_2$, c'est-à-dire ne satisfait pas $inev[f_1]\neg f_2$, c'est que l'une des séquences d'exécution à partir de q ne permet pas d'atteindre $\neg f_2$ sous la condition f_1 en un temps fini. C'est-à-dire : il est possible d'atteindre un état où f_1 n'est pas satisfaite avant d'avoir atteint $\neg f_2$, ou il est possible qu'une exécution se termine sans atteindre $\neg f_2$, ou encore, il est possible que le programme s'exécute pendant un temps infini sans jamais atteindre $\neg f_2$. Aux deux premières éventualités correspond l'existence de séquences d'exécutions finies, à la dernière, l'existence d'une séquence d'exécution infinie.

Propriété 2.7. $q \models some[f_1]f_2 \Leftrightarrow$

$\exists\, s \in Ex(q)$ telle que :

(1) $s = q_1 \ldots q_k$, $k \geq 1$ et $(i<k \Rightarrow q_i \models f_1 \wedge f_2)$ et $q_k \models \neg f_1 \wedge f_2$

ou

(2) $s = q_1 \ldots q_k$, $k \geq 1$ et $(i<k \Rightarrow q_i \models f_1 \wedge f_2)$ et $q_k \models f_1 \wedge f_2 \wedge puits$

ou

(3) $s = q_1 \ldots q_k (q_{k+1} \ldots q_n)^\omega$, $n>k \geq 1$ et $(1 \leq i \leq n \Rightarrow q_i \models f_1 \wedge f_2)$ et $q_n = q_k$.

Ces trois cas sont illustrés par les figures 2.10, 2.11 et 2.12.

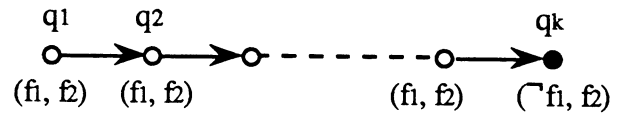


figure 2.10

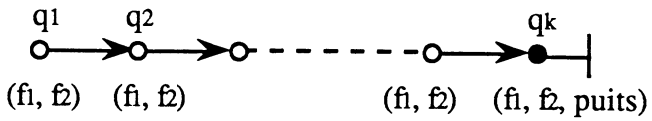


figure 2.11

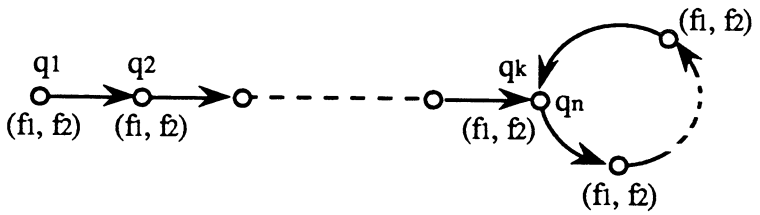


figure 2.12

Démonstration.

$q \models some[f_1]f_2 \Leftrightarrow$ non $(q \models inev[f_1]\neg f_2)$

$\Leftrightarrow$ non $(\forall\, q_1 \ldots q_i \ldots \in Mex(q), \exists k \geq 1 \,/\, (i<k \Rightarrow q_i \models f_1)$ et $q_k \models \neg f_2)$

$\Leftrightarrow \exists\, s = q_1 \ldots q_i \ldots \in Mex(q) \,/\, \forall k \geq 1, (\exists i,\ 1 \leq i<k$ et $q_i \models \neg f_1)$ ou $q_k \models f_2$

Soit (e_k) la propriété : $(\exists i,\ 1 \leq i<k$ et $q_i \models \neg f_1)$ ou $q_k \models f_2$

c'est-à-dire $(e_1) \Leftrightarrow q_1 \models f_2$

$(e_2) \Leftrightarrow q_1 \models \neg f_1$ ou $q_2 \models f_2$

...

$(e_k) \Leftrightarrow q_1 \models \neg f_1$ ou $q_2 \models \neg f_1$ ... ou $q_{k-1} \models \neg f_1$ ou $q_k \models f_2$

On a donc : $q \models some[f_1]f_2 \Leftrightarrow \exists\, s = q_1 \ldots \in Mex(q) \,/\, \forall k \geq 1 : (e_k)$

Soit (E) la propriété : $\exists\, s = q_1 \ldots q_i \ldots \in Mex(q) \,/\, \forall k \geq 1 : (e_k)$

Envisageons les trois cas suivants

(c_1) : $\exists j / q_j \models \neg f_1$

(c_2) : $\forall j, q_j \models f_1$ et s est finie

(c_3) : $\forall j, q_j \models f_1$ et s est infinie

Alors, $E \Leftrightarrow$ (E et (c_1) ou E et (c_2) ou E et (c_3)). Notons (E_i) la propriété (E et (c_i)) pour i=1, 2, 3.

Si (c_1), soit h le plus petit indice tel que $q_h \models \neg f_1$.

$\forall\, i<h$, ($q_i \models f_1$ et (e_i)) $\Leftrightarrow$ $\forall\, i<h$, ($q_i \models f_1$ et $q_i \models f_2$)

$\forall\, i<h$, ($q_i \models f_1$) et $(e_h) \Leftrightarrow q_h \models f_2$.

Donc $(E_1) \Leftrightarrow \exists\, q_1 \ldots q_h \in Ex(q)$ $(h \geq 1)$ telle que ($i<h \Rightarrow q_i \models f_1 \wedge f_2$) et $q_h \models \neg f_1 \wedge f_2$ (figure 2.10).

Si (c_2), soit h la longueur de s.

$\forall\, i \leq h$, ($q_i \models f_1$ et (e_i)) $\Leftrightarrow$ $\forall\, i \leq h$, ($q_i \models f_1$ et $q_i \models f_2$)

$s \in Ex(q)$ et s est finie $\Leftrightarrow q_h \models puits$

Donc $(E_2) \Leftrightarrow \exists\, q_1 \ldots q_h \in Ex(q)$ $(h \geq 1)$ telle que ($i<h \Rightarrow q_i \models f_1 \wedge f_2$) et $q_h \models f_1 \wedge f_2 \wedge puits$ (figure (2.11).

Si (c_3), on a : $\forall\, i$, ($q_i \models f_1$ et (e_i)) $\Leftrightarrow$ $\forall\, i$, ($q_i \models f_1$ et $q_i \models f_2$)

Donc, $\exists\, s = q_1 \ldots q_i \ldots \in Ex(q)$ infinie telle que $\forall i \geq 1$ $q_i \models f_1 \wedge f_2$

Or s est une séquence infinie d'un ensemble fini : elle passe donc au moins deux fois par un même état : $\exists k<n$ tel que $q_k = q_n$. Dans ce cas, la séquence $q_1 \ldots q_k (q_{k+1} \ldots q_n)^{\omega}$ est une séquence d'exécution infinie issue de q telle que $\forall i$=1…n, $q_i \models f_1 \wedge f_2$.

Donc $(E_3) \Leftrightarrow \exists\, q_1 \ldots q_k (q_{k+1} \ldots q_n)^{\omega}$ $(1 \leq k<n)$ telle que $q_1 = q$, $q_k = q_n$ et ($i \geq 1 \Rightarrow q_i \models f_1 \wedge f_2$) (figure 2.12).

$\therefore$

Soit $S_{fi}(q \models some[f_1]f_2)$ l'ensemble des séquences telles que (1) ou (2) :

$S_{fi}(q \models some[f_1]f_2) =$

$\{q_1 \ldots q_k \in Ex(q)$ /($1 \leq i<k \Rightarrow q_i \models f_1 \wedge f_2$) et ($q_k \models \neg f_1 \wedge f_2$ ou $q_k \models f_1 \wedge f_2 \wedge puits$)$\}$,

et soit $S_{in}(q \models some[f_1]f_2)$ l'ensemble des séquences telles que (3) :

$S_{in}(q \models some[f_1]f_2) = \{q_1 \ldots q_k.(q_{k+1} \ldots q_n)^{\omega} \in Ex(q) / q_k = q_n$ et ($i \geq 1 \Rightarrow q_i \models f_1 \wedge f_2$)$\}$

Soit $S(q \models some[f_1]f_2)$ l'union de ces deux ensembles.

Définition. L'ensemble $S(q\models some[f_1]f_2)$ est appelé ensemble des séquences explicatives de l'assertion $q\models some[f_1]f_2$.

Les éléments de $S_{fi}(q\models some[f_1]f_2)$ sont des séquences finies. Les éléments de $S_{in}(q\models some[f_1]f_2)$ sont des séquences infinies cycliques. On note Q^c l'ensemble des séquences infinies cycliques d'éléments de Q : $Q^c = \{s_1.(s_2)^\omega / s_1 \in Q^* \text{ et } s_2 \in Q^* \setminus \{\varepsilon\}\}$.

Remarque. $q\models some[f_1]f_2 \Leftrightarrow S(q\models some[f_1]f_2) \neq \varnothing$. Cependant, pour certains graphes, et certains états, il est possible que la preuve de $q\models some[f_1]f_2$ ne puisse se faire qu'à l'aide de séquences finies, ou qu'à l'aide de séquences infinies : on peut avoir $q\models some[f_1]f_2$ et $S_{fi}(q\models some[f_1]f_2) = \varnothing$ ou $S_{in}(q\models some[f_1]f_2) = \varnothing$, mais pas $S_{fi}(q\models some[f_1]f_2) = \varnothing$ et $S_{in}(q\models some[f_1]f_2) = \varnothing$.

Exemple.

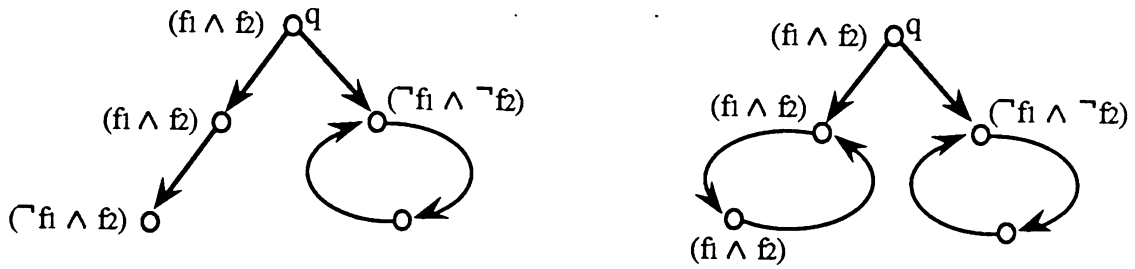


figure 2.13

Dans les modèles représentés sur la figure 2.13, $q\models some[f_1]f_2$. Dans le premier cas, $S_{in}(q\models some[f_1]f_2) = \varnothing$, dans le second, $S_{fi}(q\models some[f_1]f_2) = \varnothing$.

2.2.4.2. Cas des séquences finies

$S_{fi}(q\models some[f_1]f_2) =$

$$\{q_1 \ldots q_k \in Ex(q) / (1\leq i<k \Rightarrow q_i\models f_1\wedge f_2) \text{ et } (q_k\models \neg f_1\wedge f_2 \text{ ou } q_k\models f_1\wedge f_2\wedge puits)\}$$

Comme les séquences permettant de prouver $pot[f_1]f_2$, elles expriment la possibilité d'atteindre un but : un état qui satisfait $\neg f_1\wedge f_2$ ou un état qui satisfait $f_1\wedge f_2\wedge puits$, par une séquence finie issue de q dont les états intermédiaires satisfont $f_1\wedge f_2$. En d'autres termes, il suffit de prouver que q satisfait $pot[f_1\wedge f_2](\neg f_1\wedge f_2)$ ou que q satisfait $pot[f_1\wedge f_2](f_1\wedge f_2\wedge puits)$ pour conclure que q satisfait $some[f_1]f_2$, et il n'y a pas d'autre moyen de le prouver sinon par l'existence d'une séquence infinie :

Propriété 2.8.

$S_{fi}(q\models some[f_1]f_2) = S(q\models pot[f_1\wedge f_2](\neg f_1\wedge f_2)) \cup S(q\models pot[f_1\wedge f_2](f_1\wedge f_2\wedge puits))$.

On en déduit que celles de ces séquences qui sont des séquences d'accès à $|\neg f_1 \wedge f_2|$ ou à $|f_1 \wedge f_2 \wedge puits\ |$ sont les éléments minimaux de $S_{fi}(q \models some[f_1]f_2)$ pour la relation d'ordre $\leq_{fi}$ définie au paragraphe précédent.

Définition.
$S_{fi}^0(q \models some[f_1]f_2) = S_{fi}(q \models some[f_1]f_2) \cap (saa(|\neg f_1 \wedge f_2|) \cup saa(|f_1 \wedge f_2 \wedge puits\ |))$

Remarque. Si s et s' appartiennent à $S_{fi}(q \models some[f_1]f_2)$, et si $s \leq_{fi} s'$ alors s' ne diffère de s que par la présence de circuits. En effet, si un préfixe strict $s = q_1 \ldots q_k$ de $s' = q_1 \ldots q_k . q_{k+1} \ldots q_n$ appartenait à $S_{fi}(q \models some[f_1]f_2)$, cela signifierait que q_k satisfait $\neg f_1$ (q_k ne peut satisfaire *puits* car il a un successeur au moins, q_{k+1}). Ceci est en contradiction avec le fait que s' appartient à $S(q \models some[f_1]f_2)$.

Exemple.
Cas du système *E/R* (figure 2.2). Soit la spécification suivante : "après toute émission d'un message par *E*, il est inévitable que *R* reçoive un message". Cette spécification s'exprime par la formule :

$after(e\_em) \Rightarrow inev\ after(r\_rec)$.

Cette formule n'est pas valide parce que, par exemple, l'état 8 satisfait sa négation :

$after(e\_em) \wedge some\ \neg after(r\_rec)$.

La séquence 8.9.5.9.5.6 appartient à $S_{fi}(8 \models some\ \neg after(r\_rec))$: tous les états de cette séquence satisfont $\neg after(r\_rec)$, et 6 est un état puits ; puisque cette séquence comporte un circuit, la séquence obtenue en extrayant ce circuit est également une séquence explicative de $8 \models some\ \neg after(r\_rec)$: $8.9.5.6 \in S_{fi}(8 \models some\ \neg after(r\_rec))$

Les propriétés de $S_{fi}(q \models some[f_1]f_2)$ et $S_{fi}^0(q \models some[f_1]f_2)$ et de la relation d'ordre $\leq_{fi}$ dans ces ensembles sont analogues à celles de $S(q \models pot[f_1]f_2)$ et de $S_{fi}^0(q \models some[f_1]f_2)$. On les résume ci-dessous :

Propriétés 2.9.
- $S_{fi}(q \models some[f_1]f_2)$ est un ensemble dont les éléments sont des séquences finies. Ces séquences privées d'un circuit appartiennent à $S_{fi}(q \models some[f_1]f_2)$:

  $(q_1 \ldots q_k \in S_{fi}(q \models some[f_1]f_2)$ et $\exists\ i<j \leq k$ tel que $q_i = q_j)$

  $\Rightarrow q_1 \ldots q_i . q_{j+1} \ldots q_k \in S_{fi}(q \models some[f_1]f_2)$.

- $S_{fi}^0(q \models some[f_1]f_2)$ est un ensemble fini.
- les éléments de $S_{fi}^0(q \models some[f_1]f_2)$ sont les éléments minimaux de $S_{fi}(q \models some[f_1]f_2)$ pour la relation d'ordre $\leq_{fi}$:

$s \in S_{fi}^{0}(q \models some[f_1]f_2)$

$\Leftrightarrow [s \in S_{fi}(q \models some[f_1]f_2)$ et $(\forall\, s' \in S_{fi}(q \models some[f_1]f_2) : (s' \leq_{fi} s \Rightarrow s' = s))]$

- chaque élément de $S_{fi}(q \models some[f_1]f_2)$ a au moins un minorant minimal dans $S_{fi}^{0}(q \models some[f_1]f_2)$:

$\forall s \in S_{fi}(q \models some[f_1]f_2), \exists\, s' \in S_{fi}^{0}(q \models some[f_1]f_2)$ tel que $s' \leq_{fi} s$.

2.2.4.3. Cas des séquences infinies

La propriété suivante des séquences explicatives infinies de $q \models some[f_1]f_2$ est utilisée au chapitre 4, elle est indépendante de la suite de ce chapitre : on peut en omettre la lecture dans un premier temps.

La suite de ce paragraphe est organisée comme suit :

- définition d'une écriture canonique pour les séquences infinies cycliques.
- propriétés de $S_{in}(q \models some[f_1]f_2)$, définition d'une relation d'ordre, caractérisation et propriétés des éléments minimaux pour cette relation d'ordre.

Propriété 2.10. $S_{in}(q \models some[f_1]f_2)$ n'est pas en général un ensemble ω-régulier.

Cette propriété est une conséquence de la restriction de $S_{in}(q \models some[f_1]f_2)$ à un ensemble de séquences cycliques, elle n'est pas spécifique à cet ensemble. Nous donnons le principe de sa démonstration sur un exemple.

Supposons que les états du graphe de la figure 2.14 satisfont tous f_1 et f_2. Par conséquent, q satisfait $some[f_1]f_2$.

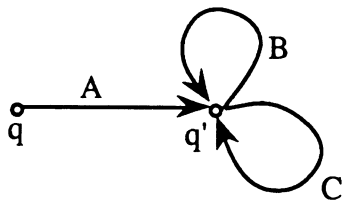


figure 2.14

Si A est l'ensemble des séquences de q à q', et $(B + C)^*$ de q' à lui même, alors $S_{in}(q \models some[f_1]f_2)$ est égal à : $A.(B + C)^*.\{(w)^{\omega} / w \in (B + C)^*\}$. Pour que ce soit un ensemble ω-régulier, il est nécessaire que l'ensemble $D=\{(w)^{\omega} / w \in (B + C)^*\}$ le soit. Si D est ω-régulier, il existe un automate de Büchi qui reconnaît D : nous allons montrer qu'un tel automate a nécessairement un nombre infini d'états.

Démonstration.

Supposons qu'il existe deux mots b et c tels que $b \in B$, $b \notin C$, $c \in C$, $c \notin B$, et qu'il existe un automate de Büchi qui reconnaît D. Considérons les chaînes (w_i) telles que $w_0 = c$, $w_1 = b.c$, ... et pour $i>1$, $w_i = b^i.c$. Alors, $\forall i$, $(w_i)^\omega \in D$: il existe dans l'automate un chemin de trace w_i qui passe un nombre infini de fois par un état de répétition ; autrement dit, puisque l'automate est fini, il existe dans l'automate un circuit contenant au moins un état de répétition, accessible à partir d'un état initial, dont la trace est un nombre entier de fois w_i : notons $U(i)$ l'ensemble des états de ce circuit. Supposons que pour i et j distincts, $U(i)$ et $U(j)$ aient un état commun. Supposons que $U(i)$ soit composé de $m+1$ occurrences de w_i, et $U(j)$ de $n+1$ occurrences de w_j ($n, m \geq 0$). Supposons qu'un état commun à $U(i)$ et $U(j)$ soit accédé dans $U(i)$ à partir de k occurrences de b par un préfixe de w_i ($0 \leq k \leq i$), et dans $U(j)$ à partir de h occurrences de b par un préfixe de w_j ($0 \leq h \leq j$) (figure 2.15).

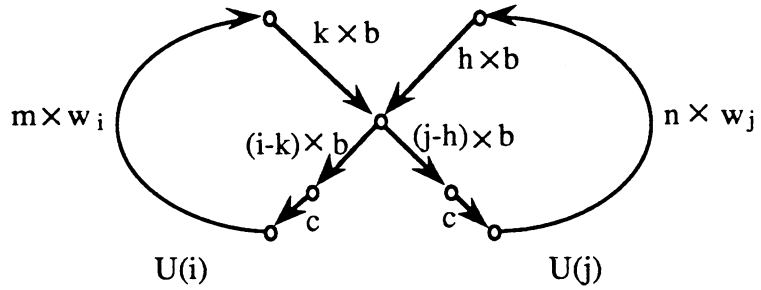


figure 2.15

Notons $u = b^{k+j-h}.c$, $v = b^{h+i-k}.c$, $x = (w_i)^m$, $y = (w_j)^n$. Alors, l'automate accepte une séquence infinie de suffixe : $s = x.u.y.v.x^2.u.y.v.x^3.u.y.v.x^4...$

Notons $k+j-h+1 = p$, $h+i-k+1 = q$, $i+1 = i'$, $j+1 = j'$: on a

$i \neq j \Rightarrow (i' \neq j')$ et $(p \neq i'$ ou $p \neq j')$ et $(q \neq i'$ ou $q \neq j')$.

Si s est cyclique, c'est-à-dire de la forme $s'.(s'')^\omega$, on peut supposer sans perte de généralité que le dernier mot de sa période s'' est c. On montre alors par récurrence sur N que quel que soit N, la longueur de s'' est strictement supérieure à $N \times m \times i' + p + n \times j' + q$.

On en déduit qu'il n'existe pas de période de longueur bornée, c'est-à-dire que si $U(i)$ et $U(j)$ ont un état commun, l'automate accepte des séquences non cycliques.

Ainsi, les ensembles $U(i)$ et $U(j)$ sont non vides et distincts, et en nombre infini : l'automate a donc un nombre infini d'états.

$\therefore$

2.2.4.3.1. Ecriture canonique des séquences cycliques.

Soit s une séquence cyclique, il existe deux séquences finies s_1 et s_2 telles que s_2 est non vide et $s = s_1.(s_2)^\omega$. Pour une même séquence s, les séquences s_1 et s_2 ne sont pas uniques (à chaque séquence correspond une infinité de couples (s_1, s_2)) : si $s = s_1.(s_2)^\omega$, on a par

exemple aussi $s = s_1.(s_2.s_2)^\omega$, $s = s_1.s_2.(s_2)^\omega$, et si $s_2 = s_3.s_4$, $s = s_1.s_3.(s_4.s_3)^\omega$... Parmi toutes les écritures possibles de s, nous en choisissons une à laquelle nous donnons le nom d'écriture canonique : elle est telle que le dernier état de s_1 et de s_2 sont identiques, et les longueurs de s_1 et de s_2 sont les plus petites possibles.

Définition. Si $s \in Q^c$, $s_1.(s_2)^\omega$ est l'*écriture canonique* de s si et seulement si $s = s_1.(s_2)^\omega$ et :

$dernier(s_1) = dernier(s_2)$,
$\forall s'_1, s'_2$ tels que $s = s'_1.(s'_2)^\omega$:
$longueur(s_2) \leq longueur(s'_2)$
et $(dernier(s'_1) = dernier(s'_2) \Rightarrow longueur(s_1) \leq longueur(s'_1))$.

Toute séquence infinie cyclique peut être écrite sous forme canonique, et l'écriture canonique d'une séquence cyclique est unique.

2.2.4.3.2. Propriétés

L'existence d'une séquence de $S_{in}(q \models some[f_1]f_2)$ prouve la satisfaisabilité de $some[f_1]f_2$ parce qu'elle montre la possibilité d'une exécution infinie du programme durant laquelle les formules f_1 et f_2 sont satisfaites. Or dans un graphe fini, l'existence d'une séquence de parcours infinie et l'existence d'un circuit dans le graphe sont équivalentes : si $s_1.(s_2)^\omega \in S_{in}(q \models some[f_1]f_2)$, la séquence $s_1.(s_2)^\omega$ met en évidence l'existence du circuit s_2 dans le graphe du programme.
Supposons que la séquence $s_1.s_2$ contienne un circuit s'_2 différent de s_2, c'est-à-dire que $s_1.s_2 = s'_1.s'_2.s_3$. Alors, la séquence $s'_1.(s'_2)^\omega$ suffit à prouver que q satisfait $some[f_1]f_2$. De plus, selon que s'_2 est un circuit de s_1, de s_2 ou de $s_1.s_2$ (mais ni de s_1, ni de s_2), on peut extraire de $s_1.(s_2)^\omega$ d'autres séquences cycliques qui appartiennent à $S_{in}(q \models some[f_1]f_2)$, comme le montrent les exemples ci-dessous :

Exemples.
- Si, comme sur la graphe de la figure 2.16, la séquence $1.2.3.2.4.5.6.(7.8.6)^\omega$ appartient à $S_{in}(1 \models some[f_1]f_2)$, la présence dans cette séquence du circuit $c' = 3.2$, donc l'existence de la séquence $1.2.(3.2)^\omega$, suffit à prouver que 1 satisfait $some[f_1]f_2$. Il en est de même de la séquence $1.2.3.2.4.5.6.(7.8.6)^\omega$ privée de ce circuit, c'est-à-dire de $1.2.4.5.6.(7.8.6)^\omega$ (figure 2.17).

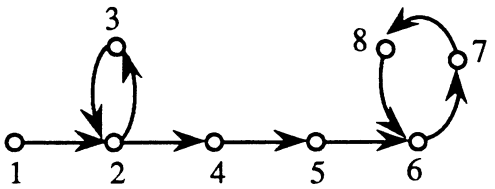


figure 2.16

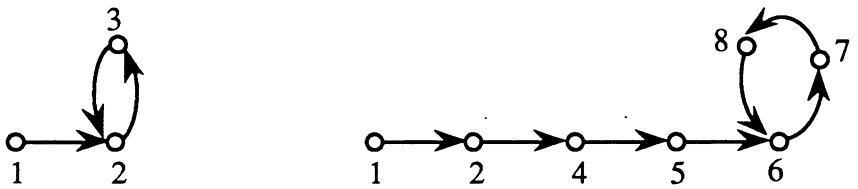


figure 2.17

- Si la séquence 1.2.3.4.5.(6.8.6.7.5)$^{\omega}$ appartient à $S_{in}(1 \models some[f_1]f_2)$ (figure 2.18), alors chacune des séquences 1.2.3.4.5.6.(8.6)$^{\omega}$ et 1.2.3.4.5.(6.7.5)$^{\omega}$ suffit pour prouver que l'état 1 satisfait $some[f_1]f_2$.

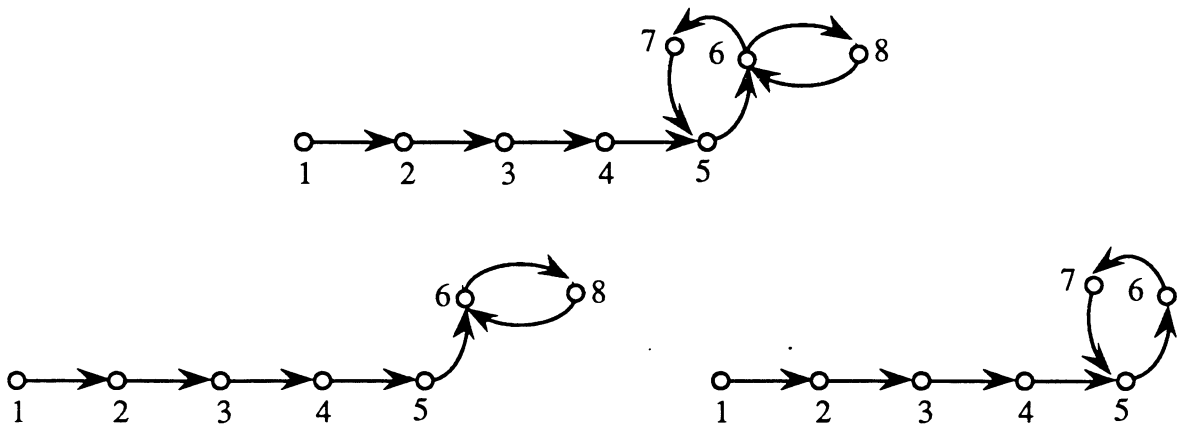


figure 2.18

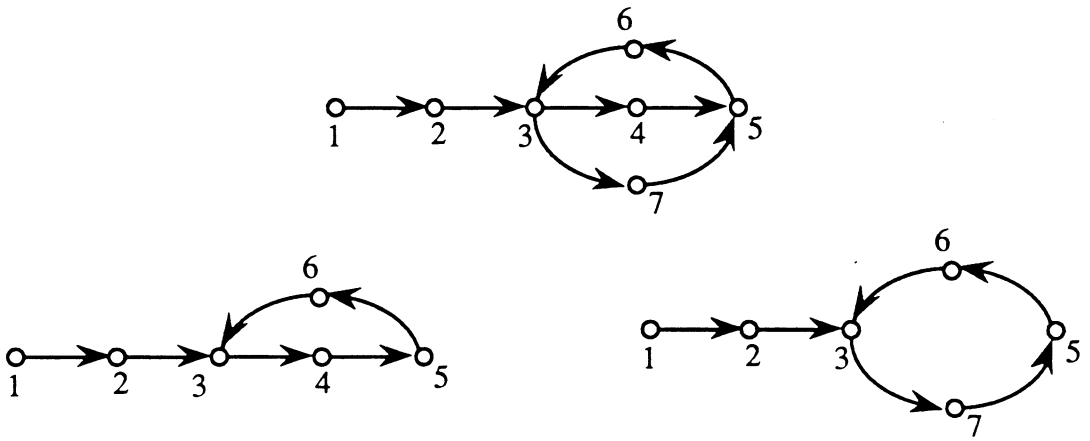


figure 2.19

- Si la séquence 1.2.3.4.5.(6.3.7.5)$^{\omega}$ appartient à $S_{in}(1 \models some[f_1]f_2)$ (figure 2.19), alors l'une ou l'autre des séquences 1.2.3.(4.5.6.3)$^{\omega}$ et 1.2.3.(7.5.6.3)$^{\omega}$ suffit pour prouver que l'état 1 satisfait $some[f_1]f_2$.

- Dans le cas du système *E/R* (figure 2.2), considérons de nouveau l'assertion :

$8 \models some\ \neg after(r\_rec)$.

La séquence 8.9.5.9.5.7.8.9.5.(7.8.9.5.9.5)$^{\omega}$ appartient à $S_{in}(8 \models some\ \neg after(r\_rec))$, mais l'une quelconque des séquences :

8.9.5.(7.8.9.5.9.5)$^{\omega}$, 8.9.5.9.5.(7.8.9.5)$^{\omega}$,... 8.9.(5.9)$^{\omega}$ ou 8.(9.5.7.8)$^{\omega}$

suffit à prouver que l'état 8 satisfait $some\ \neg after(r\_rec)$.

Propriété 2.11.

$\forall\ q_1....q_k.(q_{k+1}...q_n)^{\omega} \in S_{in}(q \models some[f_1]f_2)$ tel que $\exists\ i<j\leq n$ et ($i\neq k$ ou $j \neq n$) et $q_i = q_j$:

$q_1...q_i.(q_{i+1}...q_j)^{\omega} \in S_{in}(q \models some[f_1]f_2)$ et

si $j = n$:

si $i < k$: $q_1...q_i.(q_{i+1}...q_k)^{\omega} \in S_{in}(q \models some[f_1]f_2)$

si $i > k$: $q_1...q_k.(q_{k+1}...q_i)^{\omega} \in S_{in}(q \models some[f_1]f_2)$

si $j \neq n$:

si $j \leq k$: $q_1....q_i.q_{j+1}...q_k.(q_{k+1}...q_n)^{\omega} \in S_{in}(q \models some[f_1]f_2)$

si $k < i$: $q_1...q_k.(q_{k+1}...q_i.q_{j+1}...q_n)^{\omega} \in S_{in}(q \models some[f_1]f_2)$

si $i \leq k < j$: $q_1...q_i.(q_{j+1}...q_n.q_{k+1}...q_j)^{\omega} \in S_{in}(q \models some[f_1]f_2)$

Démonstration.

Si $q_1....q_k.(q_{k+1}...q_n)^{\omega} \in S_{in}(q \models some[f_1]f_2)$

si $j = n$: ($q_k = q_i = q_j$) $\Rightarrow$ ($q_{i+1} \in succ(q_j)$ et $q_{i+1} \in succ(q_k)$ et $q_{k+1} \in succ(q_i)$)

si $j \neq n$: ($q_n = q_k$ et $q_i = q_j$) $\Rightarrow$ ($q_{j+1} \in succ(q_i)$ et $q_{k+1} \in succ(q_n)$).

Par conséquent, chacune des séquences citées appartient à $Ex(q)$ et tous les états de ces séquences satisfont f_1 et f_2.

$\therefore$

2.2.4.3.3. Nombre de circuits d'une séquence cyclique

On appelle nombre de circuits d'une séquence cyclique dont l'écriture canonique est $s_1.(s_2)^{\omega}$ et on note NC(s), le nombre de circuits de la séquence $s_1.s_2$:

Définition. Si $s \in Q^c$, si l'écriture canonique de s est $s_1.(s_2)^{\omega}$ et si $s_1.s_2 = q_1...q_n$, $NC(s) = Card(\{(i,j)\ /\ 1\leq i<j\leq n$ et $q_i = q_j\})$.

Propriétés 2.12.
- Si Q est fini, $\{s \in Q^c / NC(s) = 1\}$ est fini.
- $\forall s, s' \in Q^c$ d'écriture canonique $s_1.(s_2)^\omega$ et $s'_1.(s'_2)^\omega$ respectivement :

$$s_1.s_2 \leq_{fi} s'_1.s'_2 \Rightarrow NC(s) \leq NC(s').$$

Démonstration
- Si $NC(s_1.(s_2)^\omega) = 1$, les séquences s_1 et s_2 ne comportent pas de circuits. Par conséquent, si Q est fini, $Card(\{s \in Q^c / NC(s) = 1\})$ est au plus égal au nombre de couples de séquences finies sans circuits d'un ensemble fini, il est donc fini.
- La deuxième propriété résulte immédiatement de la définition de $\leq_{fi}$.
$\therefore$

De la définition de NC et de la propriété 2.11, on déduit immédiatement la propriété suivante :

Propriété 2.13. $\forall s \in S_{in}(q \models some[f_1]f_2)$:

$$NC(s) > 1 \Rightarrow \exists s' \in S_{in}(q \models some[f_1]f_2) \text{ et } NC(s') < NC(s).$$

Le nombre de circuits de certaines séquences de $S_{in}(q \models some[f_1]f_2)$, s'il n'est pas vide, est donc égal à 1. Nous allons définir une relation d'ordre partiel sur $S_{in}(q \models some[f_1]f_2)$ telle que :
- les séquences n'ayant qu'un seul circuit sont les éléments minimaux de $S(q \models pot[f_1]f_2)$ pour cette relation d'ordre,
- chaque élément de $S_{in}(q \models some[f_1]f_2)$ a un minorant minimal.

2.2.4.3.4. Relation d'ordre dans Q^c

La relation d'ordre que nous définissons permet de comparer les séquences d'écriture canonique $s_1.(s_2)^\omega$ et $s'_1.(s'_2)^\omega$ lorsque les séquences $s_1.s_2$ et $s'_1.s'_2$ sont comparables pour la relation $\leq_{fi}$:

Définition. La relation $\leq_{in}$ est définie sur Q^c par :
$\forall s, s' \in Q^c$ d'écriture canonique $s_1.(s_2)^\omega$ et $s'_1.(s'_2)^\omega$ respectivement,

$$s \leq_{in} s' \Leftrightarrow s_1.s_2 \leq_{fi} s'_1.s'_2.$$

Propriétés 2.14.
- La relation $\leq_{in}$ est une relation d'ordre partiel dans Q^c.

- $\forall\ s, s' \in Q^c$ d'écriture canonique $s_1.(s_2)^\omega$ et $s'_1.(s'_2)^\omega$ respectivement :
  $s \leq_{in} s' \Rightarrow longueur(s_1.s_2) \leq longueur(s'_1.s'_2)$.
- $\forall\ s, s' \in Q^c,\ s \leq_{in} s' \Rightarrow NC(s) \leq NC(s')$

Démonstration : immédiate.

2.2.4.3.5. Séquences explicatives infinies minimales de l'assertion $q \models some[f_1]f_2$.

Définition. Soit $S_{in}^0(q \models some[f_1]f_2) = \{\ s \in S_{in}(q \models some[f_1]f_2) / NC(s) = 1\}$

Propriétés 2.15.

- $S_{in}^0(q \models some[f_1]f_2)$ est un ensemble fini.
- les éléments de $S_{in}^0(q \models some[f_1]f_2)$ sont les éléments minimaux :
  $s \in S_{in}^0(q \models some[f_1]f_2) \Leftrightarrow$
  $[s \in S_{in}(q \models some[f_1]f_2)$ et $(\forall\ s' \in S_{in}(q \models some[f_1]f_2) : (s' \leq_{in} s \Rightarrow s' = s\ ))]$
- chaque élément de $S_{in}(q \models some[f_1]f_2)$ a au moins un minorant minimal dans $S_{in}^0(q \models some[f_1]f_2)$:
  $\forall s \in S_{in}(q \models some[f_1]f_2), \exists\ s' \in S_{in}^0(q \models some[f_1]f_2)$ tel que $s' \leq_{in} s$.

Démonstration

- $S_{in}^0(q \models some[f_1]f_2) \subset \{s \in Q^c / NC(s) = 1\}$, par conséquent puisque Q est fini, c'est un ensemble fini, d'après les propriétés 2.12.
- Chaque élément s de $S_{in}^0(q \models some[f_1]f_2)$ est tel que $NC(s) = 1$. Par conséquent, d'après la propriété 2.13, il est minimal.

Réciproquement :

$\forall s \in S_{in}(q \models some[f_1]f_2), NC(s) > 1 \Rightarrow \exists\ s' \in S_{in}(q \models some[f_1]f_2) / s' \leq_{in} s$ et $s' \neq s$. En effet, si l'écriture canonique de s est $q_1 \ldots q_k.(q_{k+1} \ldots q_n)^\omega$ et $NC(s) > 1$, alors la séquence $q_1 \ldots q_k.q_{k+1} \ldots q_n$ comporte au moins deux circuits, c'est-à-dire un circuit différent de $q_{k+1} \ldots q_n$. Autrement dit, $\exists\ i<j \leq n$ tel que $q_i = q_j$ et $(i \neq k$ ou $j \neq n)$; alors

si $j = n$, si $i < k$, soit $s' = q_1 \ldots q_i.(q_{i+1} \ldots q_k)^\omega$
si $i > k$, soit $s' = q_1 \ldots q_k.(q_{k+1} \ldots q_i)^\omega$
si $j \neq n$, soit $s' = q_1 \ldots q_i.(q_{i+1} \ldots q_j)^\omega$,

dans tous les cas, $s' \leq_{in} s$, $s' \in S_{in}(q \models some[f_1]f_2)$ et $s \neq s'$.

Par conséquent, tout élément s de $S_{in}(q \models some[f_1]f_2)$ tel que $NC(s)$ est strictement supérieur à 1 n'est pas minimal, et donc, l'ensemble des éléments minimaux de $S_{in}(q \models some[f_1]f_2))$ est $S_{in}^0(q \models some[f_1]f_2)$.

- Enfin chaque élément a au moins un minorant minimal : en effet

- les éléments minimaux sont leur propre minorant minimal.
- à un élément non minimal, on peut associer une suite finie décroissante d'éléments distincts dont le dernier est minimal ; soit $s = q_1....q_k.(q_{k+1}...q_n)^\omega$ un élément de $S_{in}(q\models some[f_1]f_2)$ non minimal, c'est-à-dire tel que $NC(s) > 1$: $\exists\ i<j\leq n$ tel que $q_i = q_j$ et $(i \neq k$ ou $j \neq n)$; alors

si $j = n$, si $i < k$, soit $s' = q_1...q_i.(q_{i+1}...q_k)^\omega$

si $i > k$, soit $s' = q_1...q_k.(q_{k+1}...q_i)^\omega$

si $j \neq n$, soit $s' = q_1...q_i.(q_{i+1}...q_j)^\omega$,

dans tous les cas, $s' \in S_{in}(q\models some[f_1]f_2)$, $s' \leq_{in} s$ et $NC(s') < NC(s)$.

On peut donc construire une suite de séquences le long de laquelle la fonction NC est strictement décroissante et telle que chaque séquence est un minorant de la précédente : une telle suite est donc finie ; le dernier élément ne comporte qu'un seul circuit, c'est-à-dire qu'il appartient à $S_{in}^0(q\models some[f_1]f_2)$.

$\therefore$

Remarques.

1) Si s est la séquence d'écriture canonique $q_1...q_k.(q_{k+1}...q_n)^\omega$ et si pour des indices i et j tels que $i\leq k<j\leq n$, $(i \neq k$ ou $j \neq n)$ et $q_i = q_j$ (figure 2.20), alors la séquence $s' = q_1...q_i.(q_{j+1}...q_n.q_{k+1}...q_j)^\omega$ n'est pas comparable à s pour la relation d'ordre $\leq_{in}$, ce n'est donc pas, a fortiori, un minorant de s.

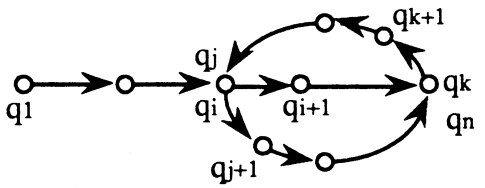


figure 2.20

Ceci vient du choix d'une définition simple pour la relation $\leq_{in}$. Dans un cas semblable, la séquence s a comme minorant la séquence s'' telle que

si $j \neq n$, $s'' = q_1...q_i.(q_{i+1}...q_j)^\omega$, ou si $j = n$, $s'' = q_1...q_i.(q_{i+1}...q_k)^\omega$.

2) Par analogie avec les séquences explicatives minimales finies, on peut caractériser les éléments de $S_{in}^0(q\models some[f_1]f_2)$ par des propriétés d'accès :

$S_{in}^0(q\models some[f_1]f_2)$

$= \{s_1.(s_2)^\omega \in S_{in}(q\models some[f_1]f_2) / s_1 \in saa(états(s_2))$ et $s_2 \in saa(états(s_1))\}$.

Exemple.

Considérons le STE représenté par la figure 2.21 :

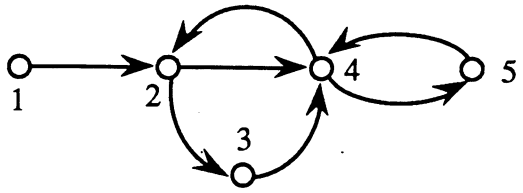


figure 2.21

Des formules f_1 et f_2 sont satisfaites par tous les états. L'état 1 satisfait $some[f_1]f_2$. La figure 2.22 est une représentation de certains éléments de $S_{in}(1 \models some[f_1]f_2)$ et de la relation d'ordre $\leq_{in}$. Les flèches indiquent une décroissance stricte selon la relation $\leq_{in}$, et de la fonction NC. $S_{in}{}^0(1 \models some[f_1]f_2)$ comporte quatre éléments qui mettent chacun en évidence un des circuits élémentaires du graphe : $1.2.(4.2)^\omega$, $1.2.4.(5.4)^\omega$, $1.2.3.4.(5.4)^\omega$ et $1.2.(3.4.2)^\omega$.

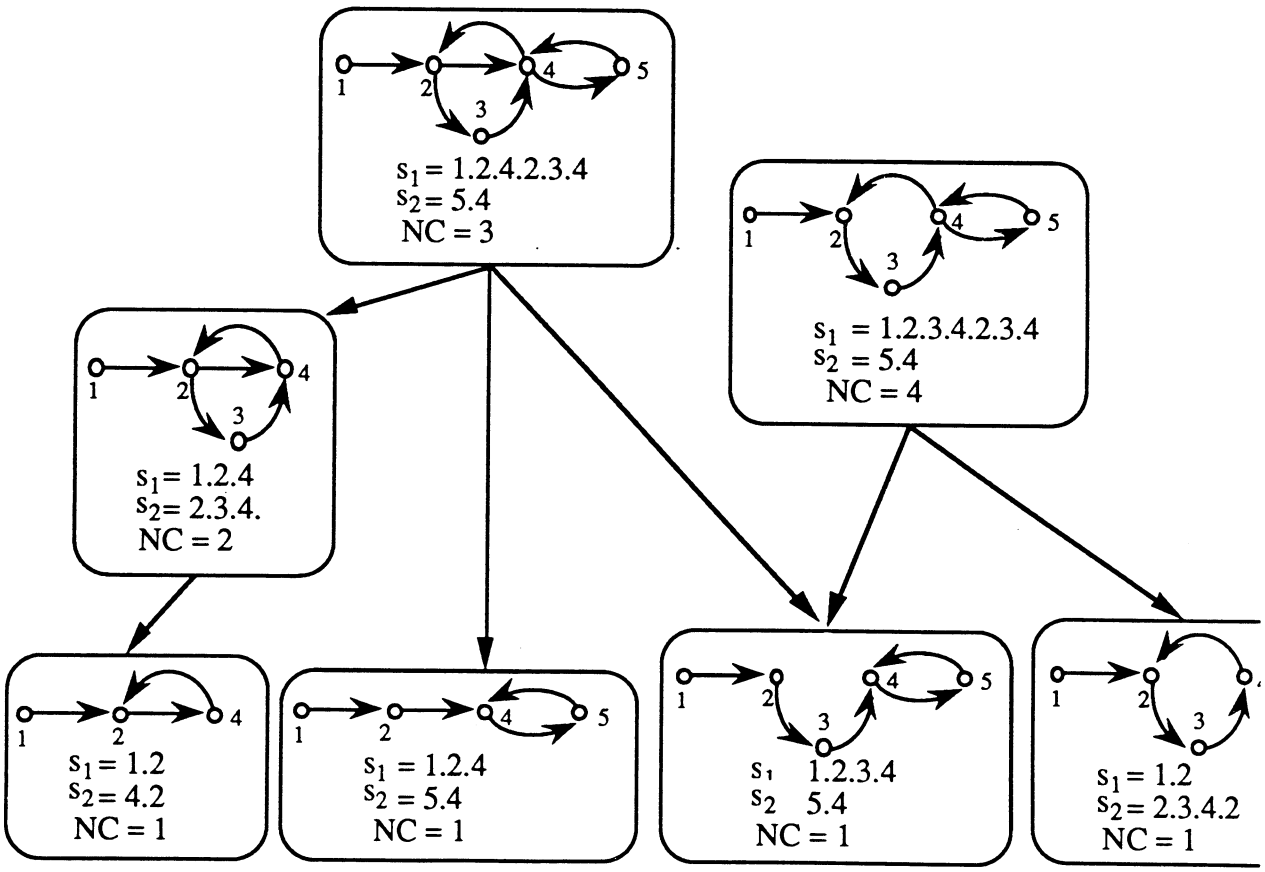


figure 2.22

Conclusion.

Soit $S^0(q \models some[f_1]f_2)$ l'union de $S_{fi}^0(q \models some[f_1]f_2)$ et de $S_{in}^0(q \models some[f_1]f_2)$. Soit $\leq$ la relation d'ordre dans $Q^* \cup Q^c$ égale à $\leq_{fi} \cup \leq_{in}$; pour toute formule temporelle f ($f = pot[f_1]f_2$ ou $f = some[f_1]f_2$), l'ensemble $S(q \models f)$ muni de la relation $\leq$, et l'ensemble $S^0(q \models f)$ sont tels que :

- $S^0(q \models f)$ est un ensemble fini.
- les éléments de $S^0(q \models f)$ sont les éléments minimaux de l'ensemble $S(q \models f)$ muni de $\leq$.
- chaque élément de $S(q \models f)$ a au moins un minorant minimal dans $S^0(q \models f)$.

C'est à partir des séquences explicatives que sont construites les explications. Sachant que si une séquence explicative permet de prouver $q \models f$, alors il existe une séquence explicative minimale qui suffit à le prouver, les explications pourront être construites à partir des séquences explicatives minimales.

2.3. Construction des explications

2.3.1. Notion d'explication

Soit $f = \vee_i \wedge_j (f_{i,j})$ une formule sous forme disjonctive. On explique $q \models f$ en expliquant pour un indice i, $q \models \wedge_j (f_{i,j})$; on explique $q \models \wedge_j (f_{i,j})$ en expliquant pour chaque indice j, $q \models f_{i,j}$; si $f_{i,j}$ est une formule de base, l'explication est l'assertion $q \models f_{i,j}$; si $f_{i,j}$ est une formule temporelle, on explique $q \models f_{i,j}$ au moyen d'une séquence de $S(q \models f_{i,j})$ en associant à chaque état de cette séquence une assertion exprimant la satisfaction de sous-formules de $f_{i,j}$.

Ainsi, la preuve de $q \models f$ est subordonnée à la preuve de la satisfaction de sous-formules de f par les états de séquences d'exécution issues de q. On appelle *explication* de $q \models f$ une structure représentant la preuve complète de $q \models f$: c'est un arbre construit à partir d'une séquence d'exécution (éventuellement réduite à un état), dans laquelle les états sont remplacés par des explications de sous-formules de f. Les feuilles d'une explication sont des explications de formules de base, c'est-à-dire des assertions.

Dans le paragraphe suivant, on définit un langage de termes permettant de représenter les explications. Ensuite, on donne une méthode de construction des explications d'une assertion par un système de règles de réécriture.

2.3.2. Ensemble des explications

Les explications sont construites à partir des assertions, des formules, et des opérateurs ":", "&" , "." et "ω".

L'ensemble $\mathfrak{X}$ des explications est le plus petit ensemble tel que :

- une assertion est une explication :
  $q \models f \in \mathfrak{X}$
- en étiquetant une explication par une formule, on obtient une explication :
  $(f \in \mathfrak{D}, x \in \mathfrak{X}) \Rightarrow (f : (x) \in \mathfrak{X})$
- l'opérateur "&" permet d'"enraciner" des explications :
  $(x_1, x_2, \ldots x_n \in \mathfrak{X}) \Rightarrow (x_1 \,\&\, x_2 \,\&\ldots\&\, x_n \in \mathfrak{X})$
- l'opérateur "." permet de construire des séquences d'explications :
  $(x_1, x_2, \ldots x_n \in \mathfrak{X}) \Rightarrow (x_1 . x_2 \ldots\ldots x_n \in \mathfrak{X})$
- l'opérateur "ω" permet de construire des explications cycliques :
  $(x \in \mathfrak{X}) \Rightarrow ((x)^{\omega} \in \mathfrak{X})$.

2.3.3. Explications d'une assertion

2.3.3.1. Séquence d'assertions associée à une séquence explicative.

Soit f une formule temporelle et s une séquence explicative de $q \models f$. On associe à s et f une séquence d'assertions permettant de prouver la validité de l'assertion $q \models f$. Cette séquence d'assertions est représentée par un élément de $\mathfrak{X}$ noté $expl(s, f)$, défini de la manière suivante :

Définition. Soit f une formule temporelle et s une séquence explicative de $q \models f$.

- si $f = pot[f_1]f_2$ et $s = q_1 \ldots q_n$,
  $expl(s, f) = q_1 \models f_1 \ldots . q_{n-1} \models f_1 . q_n \models f_2$
- si $f = some[f_1]f_2$
  si $s = q_1 \ldots q_n$ et $q_n \models \neg f_1$:
  $expl(s, f) = q_1 \models f_1 \wedge f_2 \ldots . q_{n-1} \models f_1 \wedge f_2 . q_n \models \neg f_1 \wedge f_2$
  si $s = q_1 \ldots q_n$ et $q_n \models f_1$:

$expl(s,f) = q_1 \models f_1 \wedge f_2 \ldots . q_{n-1} \models f_1 \wedge f_2 . q_n \models f_1 \wedge f_2 \wedge puits$

si l'écriture canonique de s (voir paragraphe 2.2.4.3.1) est $q_1 \ldots q_k . (q_{k+1} \ldots q_n)^\omega$:

$expl(s,f) = q_1 \models f_1 \wedge f_2 \ldots . q_k \models f_1 \wedge f_2 . (q_{k+1} \models f_1 \wedge f_2 \ldots . q_n \models f_1 \wedge f_2)^\omega$

On note : $E(q \models f) = \{expl(s,f) / s \in S(q \models f)\}$,

$E^0(q \models f) = \{expl(s,f) / s \in S^0(q \models f)\}$.

2.3.3.2. Génération des explications par réécriture des assertions.

Si s est une séquence explicative de $q \models f$, l'explication $expl(s,f)$ définie ci-dessus est une séquence d'assertions permettant de prouver $q \models f$: c'est une explication de $q \models f$. Autrement dit, si f est une formule temporelle, expliquer $q \models f$ consiste à réécrire cette assertion par une séquence d'assertions $expl(s,f)$. Les assertions de cette séquence portent sur les sous-formules de f. Ceci définit des règles de réécriture des assertions portant sur une formule temporelle. En complétant cet ensemble de règles par des règles de réécriture des assertions portant sur des formules non temporelles, on obtient le système de règles permettant de générer toutes les explications d'une assertion quelconque. Les assertions portant sur les formules de base ne sont pas réécrites : elles sont leur propre explication.

Chaque règle est de la forme :

$(c)\ a \mapsto x$

où c est une condition, a une assertion, et x un élément de $\mathfrak{X}$.

Les règles sont les suivantes :

$(q \models f_i)$	$q \models f_1 \vee \ldots \vee f_n$	$\mapsto q \models f_i$	(D)
$(q \models f_1 \wedge \ldots \wedge f_n)$	$q \models f_1 \wedge \ldots \wedge f_n$	$\mapsto (q \models f_1 \,\&\ldots\&\, q \models f_n)$	(C)
$(e \in E(q \models pot[f_1]f_2))$	$q \models pot[f_1]f_2$	$\mapsto pot[f_1]f_2 : (e)$	(P)
$(e \in E(q \models some[f_1]f_2))$	$q \models some[f_1]f_2$	$\mapsto some[f_1]f_2 : (e)$	(S)

Définition. Si la condition c est vraie, et si y est une explication dont a est un sous-terme, l'*application* de la règle $(c)\ a \mapsto x$ remplace dans y une occurrence de a par x.

Par application successive des règles à une assertion a, on construit une suite d'explications. Nous montrerons au paragraphe suivant que cette suite est toujours finie. On appelle désormais *explication de l'assertion a* le dernier élément de toute suite ainsi construite : c'est une explication à laquelle aucune règle de réécriture ne peut être appliquée.

On note $Expl(M, q \models f)$ l'ensemble des explications de l'assertion $q \models f$ dans le modèle M, ou simplement $Expl(q \models f)$ lorsqu'aucune ambiguïté ne sera possible. Par abus de langage, on appelle *explication de f* toute explication d'une assertion de la forme $q \models f$.

Si f est une formule de base fb, et si $q \models fb$, on a en particulier $Expl(q \models fb) = \{q \models fb\}$.

On définit une fonction racine qui à une explication x associe un élément de Q noté $racine(x)$:

$racine(q \models f) = q$

$racine(f : (x)) = racine(x)$

si $racine(x_1) = racine(x_2) = \ldots = racine(x_n)$, $racine(x_1 \,\&\, x_2 \,\&\ldots\&\, x_n) = racine(x_1)$

sinon, $racine(x_1 \,\&\, x_2 \,\&\ldots\&\, x_n)$ n'est pas défini.

$racine(x_1 x_2 \ldots) = racine(x_1)$.

De la définition des règles de réécriture, on déduit immédiatement que la racine des explications de $q \models f$ est définie et que cette racine est q.

Propriété 2.16. $x \in Expl(q \models f) \Rightarrow racine(x) = q$.

Exemple.

Considérons le cas du système *E/R* (figure 2.2). Considérons une formule f exprimant le fait qu'il est toujours inévitable, après émission d'un message par E, que E reçoive un accusé de réception, à condition que $L1$ ne perde pas de message :

$f = al(after(e\_em) \Rightarrow inev[\neg after(p1)]\ after(e\_rec))$.

Pour expliquer que f n'est pas valide, on explique par exemple l'assertion $1 \models \neg f$:

$1 \models pot(after(e\_em) \wedge some[\neg after(p1)]\ \neg after(r\_rec))$.

Une séquence explicative de cette assertion est la séquence 1.2, la séquence d'assertions associée est $1 \models vrai.\ 2 \models after(e\_em) \wedge some[\neg after(p1)]\ \neg after(r\_rec)$, la règle suivante peut par conséquent être appliquée :

$1 \models pot(after(e\_em) \wedge some[\neg after(p1)]\ \neg after(r\_rec))$

$\mapsto pot(after(e\_em) \wedge some[\neg after(p1)]\ \neg after(r\_rec))$:

$(1 \models vrai.\ 2 \models after(e\_em) \wedge some[\neg after(p1)]\ \neg after(r\_rec))$ (P)

Réécriture d'une assertion portant sur une conjonction :

$2 \models after(e\_em) \wedge some[\neg after(p1)]\ \neg after(r\_rec)$

$\mapsto (2 \models after(e\_em)\ \&\ 2 \models some[\neg after(p1)]\ \neg after(r\_rec))$ (C)

La séquence $2.4.5.(9.5)^{\omega}$ appartient à $S(2 \models some[\neg after(p1)]\ \neg after(r\_rec))$, la séquence d'assertions associée est

$(2 \models (\neg after(p1) \wedge \neg after(r\_rec)).\ 4 \models (\neg after(p1) \wedge \neg after(r\_rec)).$

$5 \models (\neg after(p1) \wedge \neg after(r\_rec)).$

$(9 \models (\neg after(p1) \wedge \neg after(r\_rec)).5 \models (\neg after(p1) \wedge \neg after(r\_rec)))^{\omega}$

par conséquent la règle suivante peut être appliquée :

$2\models$ *some*[$\neg$*after*(*p*1)] $\neg$*after*(*r_rec*)

$\mapsto$ *some*[$\neg$*after*(*p*1)] $\neg$*after*(*r_rec*) :

(2$\models$ ($\neg$*after*(*p*1) $\wedge$ $\neg$*after*(*r_rec*)). 4$\models$ ($\neg$*after*(*p*1) $\wedge$ $\neg$*after*(*r_rec*)).

5$\models$ ($\neg$*after*(*p*1) $\wedge$ $\neg$*after*(*r_rec*)).

(9$\models$ ($\neg$*after*(*p*1) $\wedge$ $\neg$*after*(*r_rec*)). 5$\models$ ($\neg$*after*(*p*1) $\wedge$ $\neg$*after*(*r_rec*)))$^{\omega}$) (S)

Enfin, les assertions $q\models$ ($\neg$*after*(*p*1) $\wedge$ $\neg$*after*(*r_rec*)) pour q = 2, 4, 5, 9 sont réécrites :

$q\models$ ($\neg$*after*(*p*1) $\wedge$ $\neg$*after*(*r_rec*)) $\mapsto$ ($q\models$ ($\neg$*after*(*p*1) & $q\models$ $\neg$*after*(*r_rec*))) (C)

D'où l'explication :

x_1 = *pot*(*after*(*e_em*) $\wedge$ *some*[$\neg$*after*(*p*1)] $\neg$*after*(*r_rec*)) :

(1$\models$*vrai*.

(2$\models$ *after*(*e_em*)

&

some[$\neg$*after*(*p*1)] $\neg$*after*(*r_rec*) :

((2$\models$ ($\neg$*after*(*p*1) & 2$\models$ $\neg$*after*(*r_rec*))). (4$\models$ ($\neg$*after*(*p*1) & 4$\models$ $\neg$*after*(*r_rec*))).

(5$\models$ ($\neg$*after*(*p*1) & 5$\models$ $\neg$*after*(*r_rec*))).

((9$\models$ ($\neg$*after*(*p*1) & 9$\models$ $\neg$*after*(*r_rec*))). (5$\models$ ($\neg$*after*(*p*1) & 5$\models$ $\neg$*after*(*r_rec*))))$^{\omega}$)))

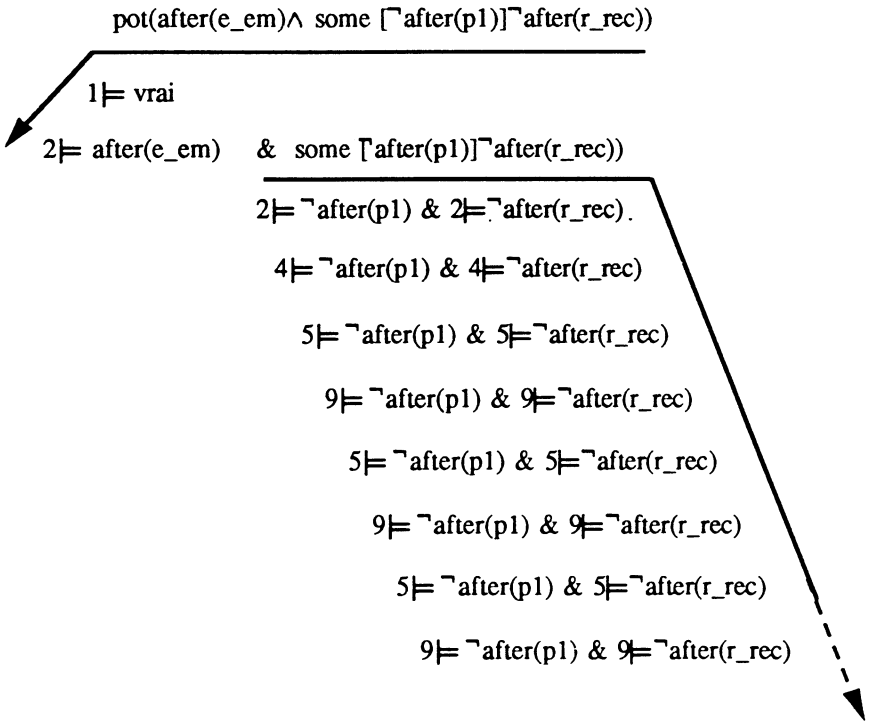


figure 2.23. L'explication x_1.

En appliquant d'autres règles à l'assertion $1 \models \neg f$, on aurait pu obtenir par exemple l'explication x_2 :

$x_2 =$ *pot*(*after*(*e_em*) $\wedge$ *some*[$\neg$*after*(*p*1)] $\neg$*after*(*r_rec*)) :
(1$\models$*vrai*.
(2$\models$ *after*(*e_em*)
&
some[$\neg$*after*(*p*1)] $\neg$*after*(*r_rec*) :
((2$\models$ ($\neg$*after*(*p*1) & 2$\models$ $\neg$*after*(*r_rec*))). (4$\models$ ($\neg$*after*(*p*1) & 4$\models$ $\neg$*after*(*r_rec*))).
(5$\models$ ($\neg$*after*(*p*1) & 5$\models$ $\neg$*after*(*r_rec*))). (9$\models$ ($\neg$*after*(*p*1) & 9$\models$ $\neg$*after*(*r_rec*))).
(5$\models$ ($\neg$*after*(*p*1) & 5$\models$ $\neg$*after*(*r_rec*)))).
(6$\models$ ($\neg$*after*(*p*1) & 6$\models$ $\neg$*after*(*r_rec*) & 6$\models$ *puits*))))

Propriétés du système de règles.
Le système de règles possède les propriétés suivantes :

Propriété 2.17.
- Terminaison : si $x \in \mathfrak{X}$, toute suite d'applications de règles à x est finie et conduit à une explication à laquelle aucune règle ne peut être appliquée.
- Régularité des dérivations : si $x \in \mathfrak{X}$ et si les assertions a_i (i=1...n) sont des sous-termes de x, si les x_i (i=1...n) sont des explications des a_i, alors x peut être réécrit en une explication obtenue en substituant dans x les x_i aux a_i.
- Complétude : si $q \models f$, $Expl(q \models f) \neq \varnothing$
- Cohérence : si $q \not\models f$, $Expl(q \models f) = \varnothing$

Démonstration
- terminaison. On définit la complexité $d(f)$ d'une formule f par récurrence sur la structure de f :
$d(fb) = 0$
$d(pot[f_1]f_2) = 1 + max(d(f_1), d(f_2))$
$d(some[f_1]f_2) = 1 + max(d(f_1 \wedge f_2), d(\neg f_1 \wedge f_2), d(f_1 \wedge f_2 \wedge puits))$
$d(f_1 \wedge \ldots \wedge f_n) = 1 + max_{i=1\ldots n}(d(f_i))$
$d(f_1 \vee \ldots \vee f_n) = 1 + max_{i=1\ldots n}(d(f_i))$
Chaque règle $q \models f \mapsto e$ est telle que les assertions qui sont des sous-termes de e sont en nombre fini et portent sur des formules dont la complexité est strictement inférieure à celle de f. La terminaison en découle.

- régularité. L'application d'une règle $a \mapsto x'$ à une explication x dont a est un sous-terme ne dépend pas du contexte de a dans x ; par conséquent, si à x on peut appliquer simultanément les règles $(R_1)\ldots(R_n)$ permettant respectivement de dériver les assertions $a_1\ldots a_n$, le résultat de l'application de ces règles à x ne dépend pas de l'ordre dans lequel elles sont appliquées.

- complétude. Elle résulte immédiatement de la définition du système de règles.

- cohérence. Si $q \not\models fb$, alors $(q, fb) \notin \models$, par conséquent, il n'y a pas d'explication de $q \models fb$. Si f n'est pas une formule de base, on vérifie que, quel que soit f, si $q \not\models f$, aucune explication construite au moyen du système de règles ne peut être une explication de $q \models f$.
$\therefore$

2.3.3.3. Système de règles minimal

Par analogie avec les séquences explicatives minimales, considérons le sous-ensemble du système de règles défini à partir des séquences explicatives minimales des formules temporelles :

$(q \models f_i)$	$q \models f_1 \vee\ldots\vee f_n$	$\mapsto q \models f_i$	(D)
$(q \models f_1 \wedge\ldots\wedge f_n)$	$q \models f_1 \wedge\ldots\wedge f_n$	$\mapsto (q \models f_1 \,\&\ldots\&\, q \models f_n)$	(C)
$(e \in E^0(q \models pot[f_1]f_2))$	$q \models pot[f_1]f_2$	$\mapsto pot[f_1]f_2 : (e)$	(P^0)
$(e \in E^0(q \models some[f_1]f_2))$	$q \models some[f_1]f_2$	$\mapsto some[f_1]f_2 : (e)$	(S^0)

D'après les propriétés des séquences explicatives minimales, il y a pour chaque assertion, un nombre fini de règles minimales, et les séquences d'explications dérivées par des règles minimales sont finies. Le système de règles minimal a les mêmes propriétés que le système de règles général, en particulier il est complet.

Conclusion

La définition des explications minimales nous permet de répondre à une exigence de minimalité pour les explications. La preuve de la non-satisfaction d'une spécification est plus concise et plus claire si elle n'emploie que les arguments strictement nécessaires, ce qui est la cas pour les explications minimales. D'autre part, ces explications sont de taille minimale dans le sens où les séquences explicatives à partir desquelles elles sont construites parcourent un nombre minimal d'états. Il s'avère cependant que cette taille peut être encore trop importante pour permettre une exploitation aisée des explications par l'usager. Au cours de

ce chapitre, nous avons défini les explications à partir des propriétés des séquences d'exécutions, indépendamment des actions portées par les transitions. Or ce sont les actions, et non pas le nom des états des séquences d'exécutions qui intéressent l'utilisateur. Le chapitre suivant est consacré à la simplification des explications en fonction d'une classe d'actions observables.

Chapitre 3

Equivalence explicationnelle

Introduction

Le diagnostic fourni à l'utilisateur par Cléo est une représentation d'une explication. Cette représentation d'une explication x est obtenue en particulier en considérant les séquences d'explications qui sont sous-termes de x. A chacune de ces séquences, on fait correspondre la séquence des racines des explications. Cette séquence est une séquence d'exécution du programme : on lui associe la séquence d'actions qui décrit cette exécution, c'est-à-dire la concaténation des noms d'actions qui étiquettent les transitions de cette séquence.
Dans la pratique, le graphe du programme peut comporter plusieurs milliers d'états, et la taille des séquences d'exécution à partir desquelles sont construites les explications rend ces explications inexploitables par l'utilisateur. C'est pourquoi, afin d'alléger le diagnostic et pour en faciliter l'interprétation, l'utilisateur a la possibilité de filtrer certaines actions : il définit un ensemble d'actions visibles V qui seules apparaîtront dans la forme externe de l'explication. Cette forme externe correspond à la représentation d'une explication réduite : la séquence des racines est une sous-suite d'une séquence d'exécution, la séquence des actions visibles est une sous-suite de la séquence des actions.
Un critère d'observation est un couple (V, f) où V est un ensemble d'actions, et f une formule : si une transition ne porte aucune action de V et si son origine et son extrémité satisfont les mêmes sous-formules de f, cette transition est dite muette. Son existence n'apporte à l'utilisateur aucune information concernant la satisfaction de f qui puisse l'intéresser : il a lui même choisi son critère d'observation (V, f).
Notre but est de supprimer, dans la mesure du possible, les transitions muettes apparaissant dans la représentation des explications, et notre démarche est la suivante (figure 3.1) : plutôt

que d'effectuer des simplifications sur des explications déjà générées, on définit un modèle simplifié, et c'est dans ce nouveau modèle qu'on calcule les explications réduites. Pour un critère d'observation (V,f), cette réduction est effectuée modulo une relation d'équivalence appelée équivalence explicationnelle. Dans des modèles équivalents au sens de cette relation, l'ensemble des explications réduites de f est le même, au nom des états près : autrement dit, si deux modèles sont explicationnellement équivalents, ils satisfont f pour les mêmes raisons.

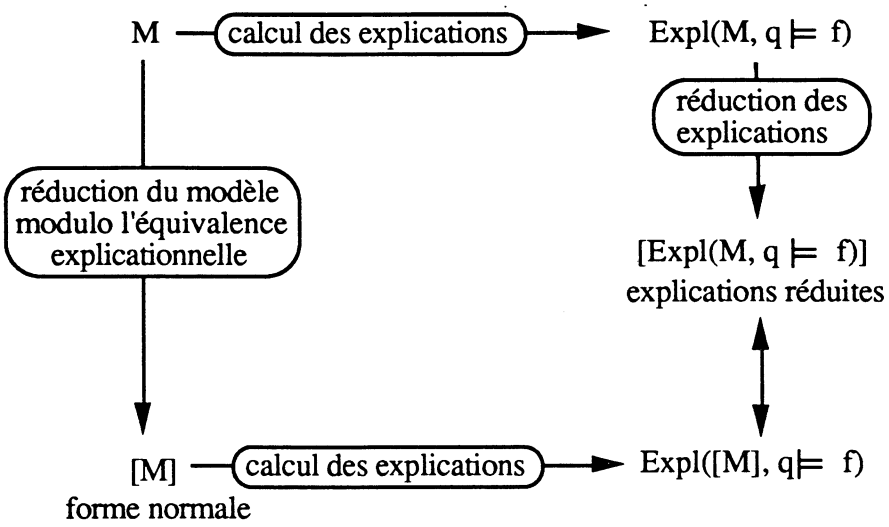


figure 3.1

Exemples.

- Considérons les modèles représentés sur la figure 3.2. Tous les états satisfont une formule g, et les transitions visibles sont (4, 5), (2, 6) et (3, 8), et sont étiquetées par la même action α.

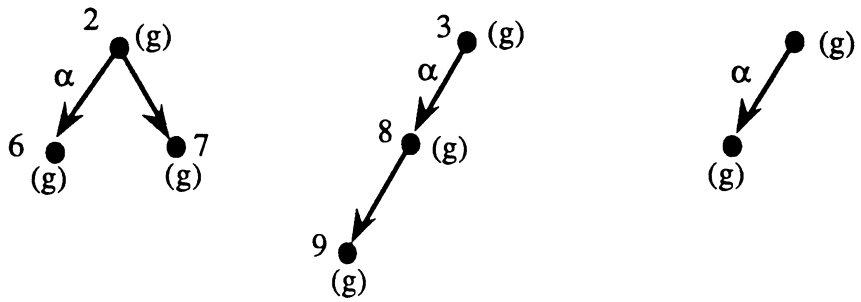


figure 3.2

Les états 1, 2 et 3 satisfont la formule *pot g*, pour les mêmes raisons : à cause de l'existence

- d'une séquence d'exécution finie issue de q (q=1, 2 ou 3), dont la seule transition visible est étiquetée par α, et dont le dernier état satisfait g.

ou

- d'une séquence d'exécution finie issue de q (q=1, 2 ou 3), ne portant aucune action visible, et dont le dernier état satisfait g

Dans cet exemple, les transitions muettes (c'est-à-dire qui ne sont pas visibles) ne sont pas significatives pour l'explication de *pot g* : elles peuvent être supprimées, c'est-à-dire que chacun des modèles de la figure 3.2 pourra être simplifié en un modèle tel que celui de la figure 3.3.

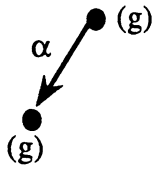


figure 3.3

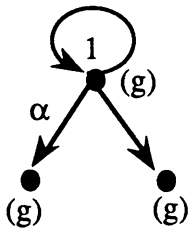


figure 3.4

- Par contre, il est possible que des transitions muettes soient significatives : ce sont celles qu'il n'est pas possible de supprimer sans supprimer un type d'explication de f. Par exemple, pour le modèle de la figure 3.4, on peut expliquer $1 \models$ *some g* de trois manières différentes :

  - au moyen d'une séquence finie maximale d'états qui satisfont g, portant la seule action visible α ;
  - au moyen d'une séquence finie maximale d'états qui satisfont g, ne portant aucune action visible ;
  - au moyen d'une séquence infinie, d'états qui satisfont g, le long de laquelle aucune action n'est visible.

Si on supprime les transitions muettes de ce modèle, les deux derniers types d'explications de *some g* disparaissent : ces transitions sont donc significatives en ce qui concerne à l'explication de cette formule.

Ce chapitre est organisé de la manière suivante :

• Etant donné un modèle et un critère d'observation (V, f), nous définissons dans le paragraphe 3.1 le modèle observé, la forme réduite des explications de f selon ce critère, et une relation d'équivalence entre explications réduites : "les modèles satisfont f pour les mêmes raisons" si les explications réduites de f dans ces modèles sont équivalentes. Le problème est alors le suivant : quelle relation d'équivalence entre modèles, la plus faible possible, a pour conséquence que ces modèles "satisfont f pour les mêmes raisons" ?

• Au cours du paragraphe 3.2, nous montrons que quelques relations d'équivalence précédemment définies ne répondent pas au problème.

• Le paragraphe 3.3. est consacré à l'étude proprement dite de l'équivalence explicationnelle : nous la comparons (selon un critère de finesse) à d'autres relations d'équivalence connues, et nous montrons qu'elle répond bien au problème posé. Elle n'est pas cependant la plus faible relation d'équivalence qui préserverait l'équivalence des explications réduites. Nous montrons l'échec de quelques propositions simples pour l'affaiblir.

Nous définissons une forme normale pour l'équivalence explicationnelle, et nous proposons une suite de transformations permettant de la calculer. Enfin, nous comparons l'ensemble des explications de $q \models f$ calculées dans le modèle sous forme normale et l'ensemble des explications réduites du modèle initial.

3.1. Réduction des explications modulo un critère d'observation.

3.1.1. Critère d'observation.

Définition. Etant donnée une formule f sous forme disjonctive, on note $sf(f)$ l'ensemble des sous-formules (au sens large) de f défini par :

si $f = fb$, $sf(f) = \{f, \neg f\}$.
si $f = pot[f_1]f_2$, $sf(f) = \{f\} \cup sf(f_1) \cup \mathrm{sf}(f_2)$
si $f = some[f_1]f_2$, $sf(f) = \{f\} \cup sf(f_1) \cup sf(f_2) \cup sf(\neg f_1)$
si $f = f_1 \wedge \ldots \wedge f_n$, $sf(f) = \{f\} \cup sf(f_1) \cup \ldots \cup sf(f_n)$
si $f = f_1 \vee \ldots \vee f_n$, $sf(f) = \{f\} \cup sf(f_1) \cup \ldots \cup sf(f_n)$

Remarque. $\neg f_1$ est considéré comme étant une sous-formule de $some[f_1]f_2$ car la satisfaction de cette formule peut être expliquée par celle de $\neg f_1$. Bien que $some[f_1]f_2$ puisse s'expliquer par la satisfaction de la formule *puits*, cette dernière n'est pas considérée comme étant une de ses sous-formules : en effet, lorsque *puits* apparaît dans une explication de $some[f_1]f_2$, elle exprime au même titre qu'une séquence infinie, l'existence d'une séquence d'exécution maximale le long de laquelle f_1 et f_2 sont toujours satisfaites. Ainsi, le fait que le dernier état d'une séquence explicative de $some[f_1]f_2$ satisfait *puits* n'est pas à mettre sur le même plan que le fait que cet état satisfait f_1, f_2 ou $\neg f_1$.

Définition. Un **critère d'observation** pour un modèle $(Q, \rightarrow, q, Act_\lambda, \mathcal{P}, \Pi)$ est un couple $C = (V, f)$ où V est un sous-ensemble de Act, et f une formule. L'ensemble V est tel que $\forall \alpha \in Act$, $(enable(\alpha) \in sf(f)$ ou $after(\alpha) \in sf(f)) \Rightarrow \alpha \in V$.
A la formule f est associée une fonction π_f de Q dans $2^{\mathcal{P}}$ telle que :

$$\forall p \in Q : \pi_f(p) = \{g \in sf(f) / p \models g\}$$

Remarque. Si $q \models f$ et $q' \models f$, et si $\pi_f(q) \neq \pi_f(q')$, q et q' ne satisfont pas les mêmes sous-formules de f. Généralement, les raisons pour lesquelles q et q' satisfont f ne sont pas les mêmes. L'équivalence explicationnelle sera telle que si q et q' sont équivalents, alors $\pi_f(q) = \pi_f(q')$.

3.1.2. Observation d'un modèle selon un critère donné.

Un critère d'observation sert à préciser quelles informations intéressent l'utilisateur, et il est donc utilisé comme filtre, afin d'éliminer les informations qui ne sont pas pertinentes le long des séquences du modèle.
V est un ensemble d'actions, et on note V_λ l'ensemble $V \cup \{\lambda\}$. Rappelons que le symbole λ étiquette les transitions du modèle qui ne portent aucune action du programme.
On désire observer

- les transitions qui sont étiquetées par une action de V,
- les transitions à l'origine et à l'extrémité desquelles la fonction π_f est différente.

Inversement, toute transition qui n'est étiquetée par aucune action de V (c'est-à-dire qui est étiquetée par une action de $(Act_\lambda \setminus V)$) et à l'origine et à l'extrémité de laquelle la fonction π_f associe le même ensemble de formules est réputée inintéressante.
En étiquetant les transitions inintéressantes par un symbole spécial τ n'appartenant pas à Act, on obtient un nouveau modèle, appelé "modèle observé selon le critère $C = (V, f)$".

Définition. Etant donné un modèle $M = (Q, \rightarrow, q, Act_\lambda, \mathcal{P}, \Pi)$, un critère d'observation (V, f), et un symbole τ n'appartenant pas à Act :
- soit $\rightsquigarrow$ le sous-ensemble de $Q \times (V_\lambda \cup \{\tau\}) \times Q$ tel que :
  si $\alpha \in V$, $q_1 \rightsquigarrow^\alpha q_2$ si et seulement si $q_1 \rightarrow^\alpha q_2$,
  $q_1 \rightsquigarrow^\lambda q_2$ si et seulement si $\exists\, \alpha \in (Act_\lambda \setminus V)$ tel que $q_1 \rightarrow^\alpha q_2$ et $\pi_f(q_1) \neq \pi_f(q_2)$.
  $q_1 \rightsquigarrow^\tau q_2$ si et seulement si $\exists\, \alpha \in (Act_\lambda \setminus V)$ tel que $q_1 \rightarrow^\alpha q_2$ et $\pi_f(q_1) = \pi_f(q_2)$
- soit Π_f la relation de Q dans $2^{\mathcal{P}}$ définie par :
  $\forall p \in Q$: $puits \in \Pi_f(p)$ si et seulement si $succ(p) = \varnothing$
  $\forall P \in \mathcal{P} \setminus \{puits\} : \Pi_f(p) = \Pi(p) \cap \pi_f(p)$

Si $M = (Q, \rightarrow, q, Act_\lambda, \mathcal{P}, \Pi)$ est un modèle, le modèle M observé selon C est le modèle $M_C = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi_f)$.

Tout élément de V_λ est appelé action visible, une transition muette est un élément de $\rightsquigarrow^\tau$ et une transition visible est une transition qui n'est pas muette. On appelle boucle τ une transition muette dont l'origine et l'extrémité sont identiques. En particulier toute transition (q_1, λ, q_1) devient une boucle τ dans le modèle observé.

Propriété 3.1. Si $M = (Q, \rightarrow, q, Act_\lambda, \mathcal{P}, \Pi)$ est un modèle, et si $M_C = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi_f)$ est le modèle M observé selon un critère d'observation C :

- $(\exists\, \alpha \in Act_\lambda$ tel que $q_1 \rightarrow^\alpha q_2) \Leftrightarrow (\exists\, \beta \in V_\lambda \cup \{\tau\}$ tel que $q_1 \rightsquigarrow^\beta q_2)$
- M est un modèle de f si et seulement si M_C est un modèle de f.

Démonstration.
Que les relations de transition dans le modèle initial et dans le modèle observé relient les mêmes états découle immédiatement de la définition du modèle observé.
D'autre part, la relation de satisfaction pour une formule f est entièrement définie par la valeur de la fonction d'interprétation pour les prédicats de base qui sont des sous-formules de f, et par la relation de transition (qui détermine l'interprétation du prédicat *puits*). Par conséquent, pour que le modèle observé soit un modèle de f, il faut et il suffit que le modèle initial soit un modèle de f.
Remarquons que cette propriété est indépendante de l'ensemble V.
$\therefore$

D'autre part, le calcul des explications décrit dans le chapitre 2 ne dépend que des relations de transition et de satisfaction. Par conséquent, les explications de $q \models f$ dans le modèle M et dans le modèle M_C sont les mêmes :

Propriété 3.2. $Expl(M, q \models f) = Expl(M_C, q \models f)$.

Exemple.
Considérons le modèle du système *E/R* (figure 3.5). Soit $C = (V, f)$ avec $V = \{e\_em, r\_rec\}$ et $f = after(e\_em) \wedge some\ \neg after(r\_rec)$. Le modèle observé selon C est représenté par la figure 3.6. Dans le modèle initial comme dans le modèle observé, l'état 8 satisfait f, et toute explication de $8 \models f$ dans l'un des modèles est une explication de $8 \models f$ dans l'autre.

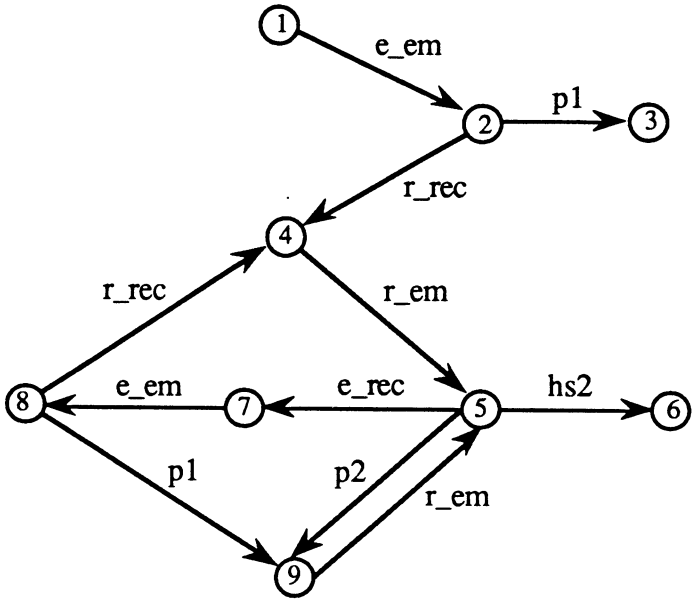


figure 3.5

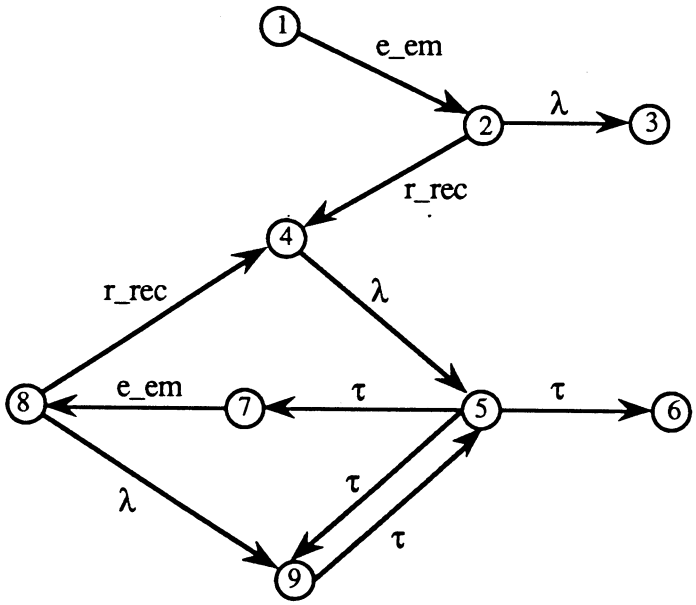


figure 3.6

Notations

Dans toute la suite, nous adopterons les conventions suivantes :

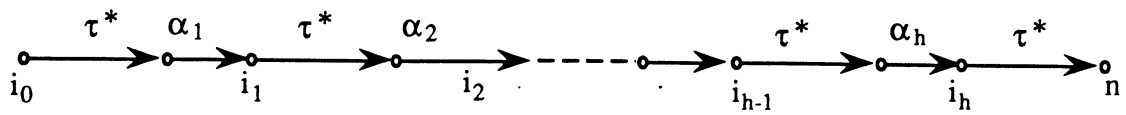


figure 3.7

- (figure 3.7) Si x est une séquence finie d'explications $x_1....x_n$, $q_1...q_n$ désignera la séquence d'exécution correspondante, $\alpha_1...\alpha_h$ la séquence des transitions visibles portées par cette séquence, i_k l'indice de l'extrémité de la transition étiquetée par α_k. On pose $i_0 = 1$. L'indice de l'extrémité de la dernière transition visible de la séquence est i_h.

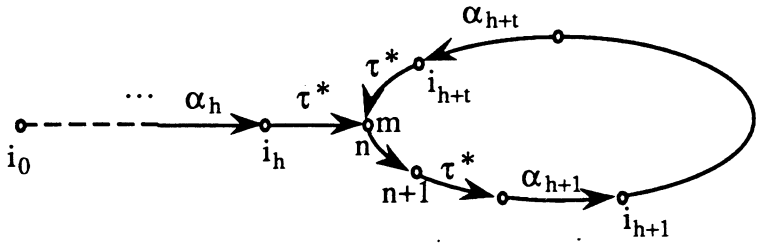


figure 3.8

- (figure 3.8) Si x est une séquence infinie d'explications $x_1.x_2...x_n.(x_{n+1}...x_m)^\omega$, $q_1....q_n.(q_{n+1}...q_m)^\omega$ désignera la séquence d'exécution correspondante, $\alpha_1...\alpha_h$ sont les transitions visibles de la séquence $q_1...q_n$,$\alpha_{h+1}...\alpha_{h+t}$ celles de la séquence $q_n...q_m$, i_k l'indice de l'extrémité de la transition étiquetée par α_k. On pose $i_0 = 1$.
L'indice de l'extrémité de dernière transition visible "avant le circuit" est i_h, celui de la dernière transition visible du circuit, i_{h+t} : toutes les transitions du circuit sont muettes si et seulement si $t = 0$.

3.1.3. Explications réduites.

Nous allons définir au cours de ce paragraphe la forme réduite d'une explication selon un critère d'observation. Par une transformation préalable, on traduit la présence d'états puits ou de circuits de transitions muettes dans les séquences explicatives par des prédicats de convergence et de divergence : cette transformation définit une forme intermédiaire d'une explication appelée forme simplifiée. On définit ensuite une opération de préfixage des explications simplifiées par une suite de transitions muettes, pour enfin définir la forme réduite d'une explication.

3.1.3.1. Prédicats de convergence et de divergence.

Définition. $\forall\, q \in Q, \perp(q) \Leftrightarrow (\exists\, q' \text{ tel que : } q \rightsquigarrow^{\tau^*} q' \text{ et } succ(q') = \varnothing)$
$\forall\, q \in Q, \Delta(q) \Leftrightarrow (\exists\, q' \text{ tel que : } q \rightsquigarrow^{\tau^*} q' \text{ et } q' \rightsquigarrow^{\tau^*\tau} q')$

Le prédicat $\perp$ est appelé prédicat de *convergence*, le prédicat Δ, prédicat de *divergence*. Si $\perp(q)$, on dit que q peut converger, si $\Delta(q)$, que q peut diverger.

$\perp(q)$ signifie qu'il est possible d'atteindre un état puits q' à partir de q par une séquence de transitions muettes ; soit s une telle séquence, et supposons que le programme exécute la séquence s à partir de q : aucune action visible n'est exécutée, et le programme s'arrête au bout d'un temps fini ; du point de vue de l'observation, il n'est pas possible de distinguer les états q et q', ni les états intermédiaires de s, autrement dit, l'état q peut être assimilé pour l'observation, à un état puits.
$\Delta(q)$ signifie qu'il est possible d'atteindre à partir de q, sans exécuter d'action visible, un état q', situé sur un circuit ne comportant que des transitions muettes : autrement dit, qu'il est possible que le programme s'exécute pendant un temps infini sans qu'aucune action visible ne soit jamais observée ; les états de la séquence d'exécution correspondante ne peuvent être distingués les uns des autres.

3.1.3.2. Convergence et divergence dans les explications.

Rappelons que les séquences d'explications qui sont sous-termes d'une explication d'une formule f sont de l'une des formes suivantes :

- $x_1.x_2...x_n$ où $x_1, x_2,..., x_{n-1}$ sont des explications d'une même formule g_1, et x_n une explication d'une formule g_2 : ceci correspond à la réécriture d'une assertion $q \models pot[f_1]f_2$ ou $q \models some\ [f_1]f_2$ avec dans le premier cas $g_1 = f_1$, $g_2 = f_2$, et dans le second $g_1 = f_1 \wedge f_2$, $g_2 = f_1 \wedge f_2 \wedge puits$ ou $g_2 = \neg f_1 \wedge f_2$.

- $x_1.x_2...x_n.(x_{n+1}...x_m)^\omega$ où tous les x_i sont des explications d'une même formule g : ceci correspond à la réécriture d'une assertion $q \models some[f_1]f_2$ avec $g = f_1 \wedge f_2$.

Considérons deux états q_1 et q_2 tels que dans le modèle observé, la transition (q_1, q_2) est muette, et une séquence de deux explications x_1 et x_2 d'une même formule g, dont les racines sont ces états. Les états q_1 et q_2 étant reliés par une transition muette, ils ne sont pas distinguables pour l'utilisateur. Par conséquent, il n'est pas nécessaire que la satisfaction de g lui soit expliquée pour chacun de ces états. Sans perdre d'information utile, on peut assimiler cette séquence de deux explications à l'une quelconque des deux explications qui la constituent : par exemple x_1 (figure 3.9).

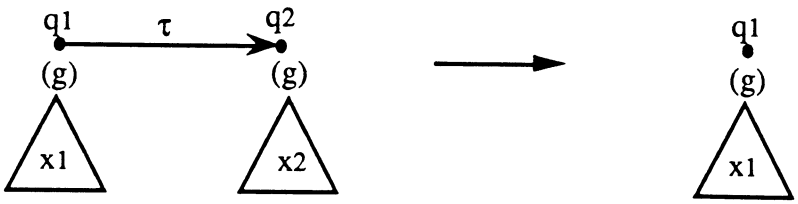


figure 3.9

A partir de cette remarque, nous définissons deux transformations :

(A) (figure 3.10) Supposons que x_1 soit une explication d'une formule g, et x_2 une explication de $g \wedge puits$. Pour l'observateur, q_1 ne peut être distingué de l'état puits q_2 : il suffit à l'utilisateur de savoir pourquoi q_1 satisfait g, et que, après avoir atteint l'état q_1, le programme peut s'arrêter sans exécuter d'action observable ; dans ce cas, on peut assimiler la séquence $x_1.x_2$ à l'explication x_1 à laquelle on adjoindra que q_1 peut converger, ce que nous noterons $(x_1 \,\&\, \bot(q_1))$.

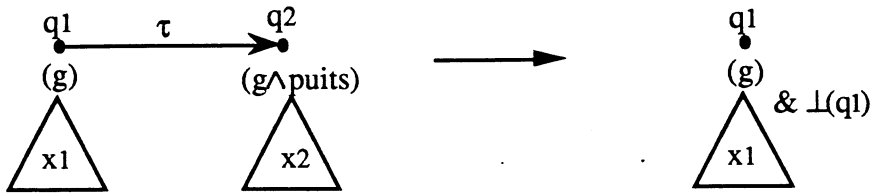


figure 3.10

(B) (figure 3.11) Considérons une séquence $x_1.(x_2 \ldots x_n)^\omega$ d'explications d'une même formule g, où, si q_i est la racine de x_i pour $i = 1 \ldots n$, toutes les transitions (q_i, q_{i+1}) sont muettes. Le long de la séquence d'exécution $q_1.(q_2 \ldots q_n)^\omega$, les q_i sont indistinguables : ayant atteint l'état q_1, il suffit à l'utilisateur de savoir pourquoi q_1 satisfait g, et que le programme peut s'exécuter durant un temps infini sans qu'il puisse observer aucune action ; la séquence d'explications pourra être assimilée à l'explication x_1 à laquelle on adjoindra que l'état q_1 peut diverger, ce que nous noterons $(x_1 \,\&\, \Delta(q_1))$.

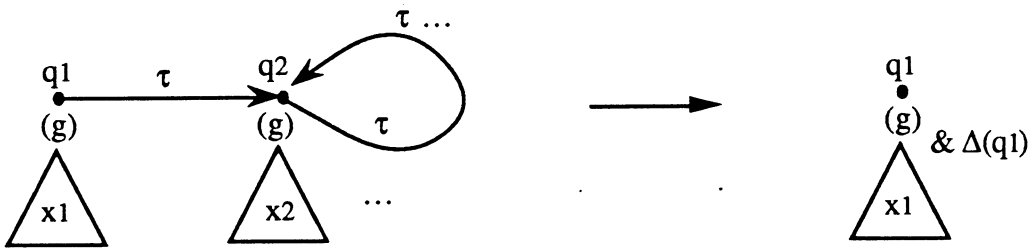


figure 3.11

La généralisation au cas où la séquence $q_1.q_2$ n'est pas une transition muette, mais une suite de transitions muettes de longueur finie quelconque, est immédiate.

Remarque. Les transformations (A) et (B) s'appliquent toujours à une séquence d'explications obtenue à partir d'une séquence explicative de $some[f_1]f_2$; en effet,

- dans le premier cas, $q_2 \models puits$ intervient dans l'explication de f sans que *puits* soit une sous-formule de f (la transition (q_1, q_2) est muette),
- dans le second, la séquence d'explications est infinie.

En appliquant les transformations (A) et (B) à une explication, on obtient une forme simplifiée d'une explication, ayant le même pouvoir explicatif que l'explication initiale. Ces transformations modifient le langage de termes permettant de représenter les explications : en effet, $\perp(q)$ et $\Delta(q)$ sont désormais des explications au même titre que les assertions. Soit $\models_C$ l'ensemble $\models \cup \{\perp(q) / q \in Q\} \cup \{\Delta(q) / q \in Q\}$. L'ensemble des explications simplifiées est le plus petit ensemble $\mathfrak{X}_C$ tel que :

- $\models_C \subset \mathfrak{X}_C$,
- $(f \in \mathfrak{D}, x \in \mathfrak{X}_C) \Rightarrow (f : (x) \in \mathfrak{X}_C)$
- $(x_1, x_2, \ldots x_n \in \mathfrak{X}_C) \Rightarrow (x_1 \,\&\, x_2 \,\&\ldots\&\, x_n \in \mathfrak{X}_C)$
- $(x_1, x_2, \ldots x_n \in \mathfrak{X}_C) \Rightarrow (x_1 . x_2 \ldots\ldots x_n \in \mathfrak{X}_C)$
- $(x \in \mathfrak{X}_C) \Rightarrow ((x)^\omega \in \mathfrak{X}_C)$.

Définition de la forme simplifiée d'une explication.
Etant donné un critère d'observation C, à toute explication x d'une assertion, on associe une explication simplifiée $(x)_C$ de $\mathfrak{X}_C$; la transformation permettant de passer de x à $(x)_C$ est définie par récurrence sur la structure des explications de la manière suivante :

- $(q \models f)_C = q \models f$
- $(f : (x))_C = f : ((x)_C)$
- $(x_1 \,\&\ldots\&\, x_n)_C = (x_1)_C \,\&\ldots\&\, (x_n)_C$
- si $x_n = (x'_n \,\&\, q_n \models \mathit{puits})$,
  $(x_1 \ldots x_n)_C = (x_1)_C \ldots ((x_{i_h})_C \,\&\, \perp(q_{i_h}))$
  sinon,
  $(x_1 \ldots x_n)_C = (x_1)_C \ldots (x_n)_C$
- si $t = 0$,
  $(x_1 x_2 \ldots x_n . (x_{n+1} \ldots x_m)^\omega)_C = (x_1)_C . (x_2)_C \ldots ((x_{i_h})_C \,\&\, \Delta(q_{i_h}))$
  sinon,
  $(x_1 x_2 \ldots x_n . (x_{n+1} \ldots x_m)^\omega)_C = (x_1)_C . (x_2)_C \ldots (x_n)_C . ((x_{n+1})_C \ldots (x_m)_C)^\omega$.

On note $\mathit{Expl}_C(q \models f)$ l'ensemble $\{(x)_C / x \in \mathit{Expl}(q \models f)\}$.

Exemple.
Soit $C = (\{e\_em, r\_rec\}, f)$, où $f = \mathit{after}(e\_em) \wedge \mathit{some}\ \neg \mathit{after}(r\_rec))$. Le modèle du système *E/R* observé selon C est représenté sur la figure 3.6. Dans ce modèle, considérons les explications suivantes de $8 \models f$:

$x_1 = (8 \models \mathit{after}(e\_em)$
&

some ¬*after*(*r_rec*) : (8⊨¬*after*(*r_rec*). 9⊨¬*after*(*r_rec*). 5⊨¬*after*(*r_rec*).
(6⊨¬*after*(*r_rec*) & 6⊨*puits*))

x_2 = (8⊨*after*(*e_em*)
&
some ¬*after*(*r_rec*) : (8⊨¬*after*(*r_rec*). 9⊨¬*after*(*r_rec*).
(5⊨¬*after*(*r_rec*). 9⊨¬*after*(*r_rec*))$^{\omega}$)

Les transitions (9, 5), (5, 9), (5, 6) sont muettes. Les formes simplifiées de ces explications sont :

$(x_1)_C$ = (8⊨*after*(*e_em*)
&
some ¬*after*(*r_rec*) : (8⊨¬*after*(*r_rec*). (9⊨¬*after*(*r_rec*) & ⊥(9)))

$(x_2)_C$ = (8⊨*after*(*e_em*)
&
some ¬*after*(*r_rec*) : (8⊨¬*after*(*r_rec*). (9⊨¬*after*(*r_rec*) & Δ(9))).

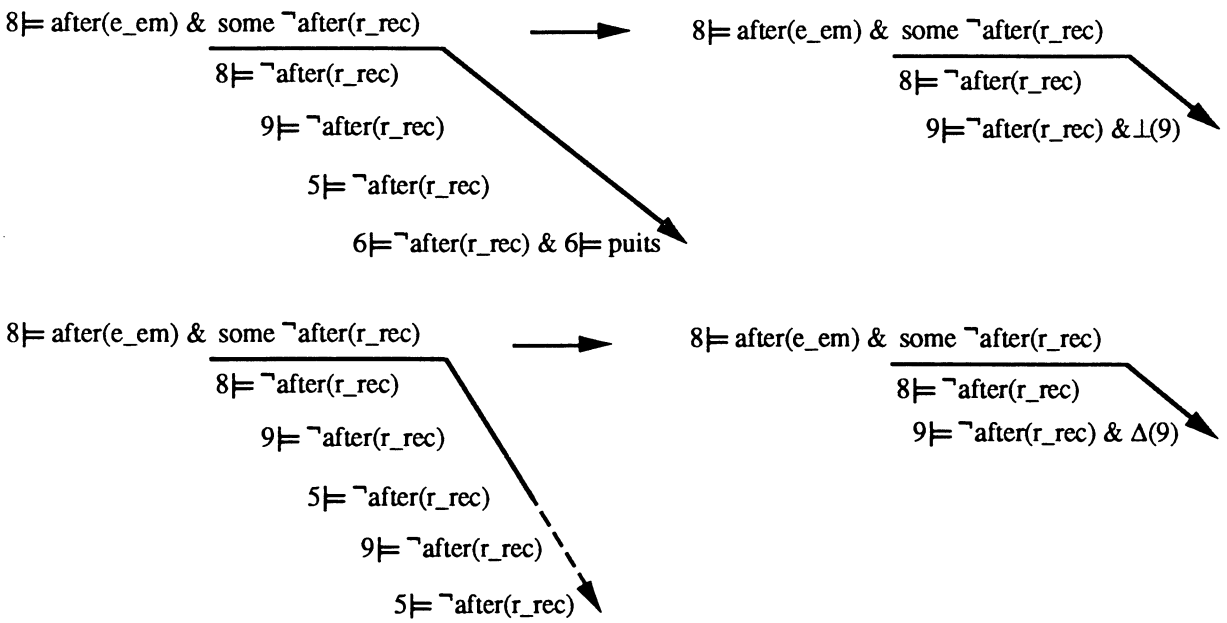


figure 3.12. Simplification de x_1 et de x_2

Avant de définir la forme réduite d'une explication, nous définissons une opération de préfixage des explications simplifiées, par une suite de transitions muettes.

3.1.3.3. Préfixage des explications par des transitions muettes.

Soit $C = (V, f)$. Si $q_1 \rightsquigarrow^{\tau} q_2$ alors

$\forall\, g \in sf(f),\ q_2 \models g \Rightarrow q_1 \models g,$

$\bot(q_2) \Rightarrow \bot(q_1)$

$\Delta(q_2) \Rightarrow \Delta(q_1).$

$\forall\, g = pot[f_1]f_2 \in sf(f),\ \forall g = some[f_1]f_2 \in sf(f),\ s \in S(q_2 \models g) \Rightarrow (q_1.s \in S(q_1 \models g)).$

Toute propriété de q_2 est donc une propriété de q_1, et toute séquence explicative issue de q_2 et préfixée par q_1 est une séquence explicative de la même formule, issue de q_1. En ce qui concerne les explications, il est donc possible d'associer à toute explication simplifiée x de $q_2 \models g$ une explication simplifiée de $q_1 \models g$: l'opération de préfixage de x par la transition muette (q_1, q_2) permet d'obtenir une telle explication.

Exemple.

Considérons le système E/R, et le critère (V, f) avec $V = \{e\_em, r\_rec\}$, et $f = pot[some\ \neg after(r\_rec)]after(r\_rec) \wedge some\ \neg after(e\_em)$. Le modèle observé est identique à celui représenté sur sur la figure 3.6.

Soit x l'explication simplifiée de $5 \models f$ suivante :

$x = pot[some\ \neg after(r\_rec)]after(r\_rec)$:

$(some\ \neg after(r\_rec) : (5 \models \neg after(r\_rec)\ \&\ \bot(5)).$

$(some\ \neg after(r\_rec) : (7 \models \neg after(r\_rec).\ 8 \models \neg after(r\_rec).\ (9 \models \neg after(r\_rec)\ \&\ \Delta(9))).$

$(some\ \neg after(r\_rec) : (8 \models \neg after(r\_rec).\ (9 \models \neg after(r\_rec)\ \&\ \Delta(9))).$

$4 \models after(r\_rec))$

$\&\ some\ \neg after(e\_em) : (5 \models \neg after(e\_em)\ \&\ \bot(5))$

On peut lui associer l'explication simplifiée de $9 \models f$ suivante, en considérant la transition muette (9, 5) :

$y = pot[some\ \neg after(r\_rec)]after(r\_rec)$:

$(some\ \neg after(r\_rec) : (9 \models \neg after(r\_rec)\ \&\ \bot(9)).$

$(some\ \neg after(r\_rec) : (5 \models \neg after(r\_rec))\ \&\ \bot(5)).$

$(some\ \neg after(r\_rec) : (7 \models \neg after(r\_rec).\ 8 \models \neg after(r\_rec).\ (9 \models \neg after(r\_rec)\ \&\ \Delta(9))).$

$(some\ \neg after(r\_rec) : (8 \models \neg after(r\_rec).\ (9 \models \neg after(r\_rec)\ \&\ \Delta(9))).$

$4 \models after(r\_rec))$

$\&\ some\ \neg after(e\_em) : (9 \models \neg after(e\_em)\ \&\ \bot(9))$

L'explication y est obtenue par préfixage de l'explication x par la transition muette (9, 5).

Définition. Soit (q_1, q_2) une transition muette dans le modèle observé, et x une explication simplifiée de $q_2 \models g$. Le résultat du préfixage de l'explication x par la transition (q_1, q_2) est l'explication notée $(q_1, q_2)\triangleleft(x)$ définie ci-dessous :

$(q_1, q_2) \triangleleft (q_2 \models g) = q_1 \models g$
$(q_1, q_2) \triangleleft (\bot(q_2)) = \bot(q_1)$
$(q_1, q_2) \triangleleft (\Delta(q_2)) = \Delta(q_1)$
$(q_1, q_2) \triangleleft (g : (y)) = g : ((q_1, q_2)\triangleleft(y))$
$(q_1, q_2) \triangleleft (x_1 \&\ldots\& x_n) = ((q_1, q_2)\triangleleft(x_1)) \& \ldots \& ((q_1, q_2)\triangleleft(x_n))$
$(q_1, q_2) \triangleleft (x_1.x_2\ldots) = ((q_1, q_2)\triangleleft(x_1)).x_1.x_2\ldots$

On étend cette définition à un nombre quelconque de transitions muettes :

- si x est une explication simplifiée de racine q_1 : $(q_1, q_1)\triangleleft(x) = x$
- si x est une explication simplifiée de racine q_n, et si les transitions (q_i, q_{i+1}) sont muettes, on a :

$$(q_1, q_2, \ldots q_n)\triangleleft(x) = (q_1, q_2)\triangleleft((q_2, q_3)\triangleleft(\ldots(q_{n-1},q_n) \triangleleft (x)\ldots).$$

Propriété 3.3. Etant donné le critère d'observation (V,f), (q_1, q_2) une transition muette, x une explication simplifiée de $q_2 \models g$, alors :

$$(q_1, q_2)\triangleleft(x) \in Expl_C(q_1 \models g).$$

Par conséquent, au préfixage près, si la transition (q_1, q_2) est muette, toute explication simplifiée de $q_2 \models g$ est une explication simplifiée de $q_1 \models g$.

3.1.3.4. Explications réduites

En simplifiant une explication, on fait disparaître un certain nombre de transitions muettes. Les transitions muettes résiduelles seront supprimées dans la forme réduite de l'explication. Examinons donc quelles transitions muettes peuvent subsister dans la forme simplifiée des explications. Les séquences d'explications qui sont les sous-termes d'une explication simplifiée d'une assertion sont de l'une des formes suivantes :

a) $x_1.x_2\ldots x_n$ où $x_1, x_2,\ldots, x_{n-1}$ sont des explications d'une même formule g_1, et

1) x_n est une explication d'une formule g_2 : ceci correspond à la réécriture d'une assertion $q\models pot[f_1]f_2$ ou $q\models some[f_1]f_2$ avec dans le premier cas $g_1 = f_1$, $g_2 = f_2$, et dans le second $g_1 = f_1 \wedge f_2$, et $g_2 = \neg f_1 \wedge f_2$.

ou

2) $x_n = x'_n \& \bot(q_n)$, x'_n étant une explication de g_1 (réécriture de $q\models some[f_1]f_2$ avec $g_1 = f_1 \wedge f_2$).

ou

3) $x_n = x'_n$ & $\Delta(q_n)$, x'_n étant une explication de g_1 (réécriture de $q \models some[f_1]f_2$ avec $g_1 = f_1 \wedge f_2$).

b) $x_1 x_2 \ldots x_n.(x_{n+1} \ldots x_m)^\omega$ où tous les x_i sont des explications d'une même formule g (réécriture de $q \models some[f_1]f_2$ avec $g = f_1 \wedge f_2$).

Dans les cas a2 et a3, la transition (q_{n-1}, q_n) n'est pas muette. Dans le cas b, une au moins des transitions (q_i, q_{i+1}) $(i \geq n)$ n'est pas muette : il n'existe plus de séquence infinie de transitions muettes.

Par conséquent, une transition (q_i, q_{i+1}) peut être muette dans les cas suivants :

- cas a1 :

  x_i et x_{i+1} sont des explications de g_1 et $i+1 < n$,

  ou

  x_i est une explication de g_1, x_{i+1} est une explication de g_2 et $i+1 = n$.

  (Ce dernier cas correspond au cas où la séquence $q_1 \ldots q_n$ n'est pas une séquence explicative minimale)

- cas b : x_i et x_{i+1} sont des explications de g.

Nous avons vu que si x_i et x_{i+1} expliquent la même formule g_1, on pouvait remplacer cette séquence de deux explications par une seule de ces explications, par exemple x_i : il est redondant d'expliquer g_1 pour deux états reliés par une transition muette. Ceci se généralise aisément au cas d'une séquence de longueur quelconque finie k, d'états reliés par des transitions muettes : la séquence correspondante d'explications d'une même formule sera assimilée à l'un de ses éléments (lui même sous-forme réduite). Par exemple cette explication peut être la première de la séquence. Remarquons qu'en appliquant cette transformation dans le cas b, toutes les transitions muettes disparaissent, et que la séquence reste infinie.

Par contre, si x_i est une explication de g_1, x_{i+1} est une explication de g_2, ce choix est à reconsidérer : on ne peut pas choisir indifféremment l'une ou l'autre des deux explications de la séquence si l'on veut qu'une explication réduite représente, au même titre que l'explication initiale, un ensemble de propriétés permettant de prouver qu'une formule est satisfaite. Considérons par exemple une explication de la formule $pot[f_1]f_2$. Soit $x = pot[f_1]f_2$: $(x_1 \ldots . x_n)$. L'explication x_n étant la seule explication de f_2 dans la séquence $x_1 \ldots . x_n$, elle devra apparaître dans la forme réduite de x. D'autre part, considérons l'état q_{i_h}, extrémité de la dernière transition visible de la séquence $q_1 \ldots q_n$: puisque q_n satisfait f_2 qui est une sous-formule de f, q_{i_h} satisfait aussi f_2. A partir de x_n, on peut construire grâce à l'opération de préfixage une explication de f_2 dont la racine est q_{i_h}. C'est cette explication après préfixage et sous forme réduite qui apparaîtra dans la forme réduite de x (voir figure 3.13).

Nous sommes maintenant en mesure de définir la forme réduite d'une explication simplifiée. Nous noterons $[x]$ la forme réduite de x et $[X]$ l'ensemble $\{[x] / x \in X\}$.

Par définition :

(1) $[q \models f] = q \models f$

(2) $[\bot(q)] = \bot(q)$

(3) $[\Delta(q)] = \Delta(q)$

(3) $[f : (x)] = f : ([x])$

(4) $[x_1 \& \ldots \& x_n] = [x_1] \& \ldots \& [x_n]$

(5) $[x_1 \ldots x_n] = [x_{i_0}] \ldots . [x_{i_{h-1}}] . [(q_{i_h} \ldots q_n) \lhd (x_n)]$

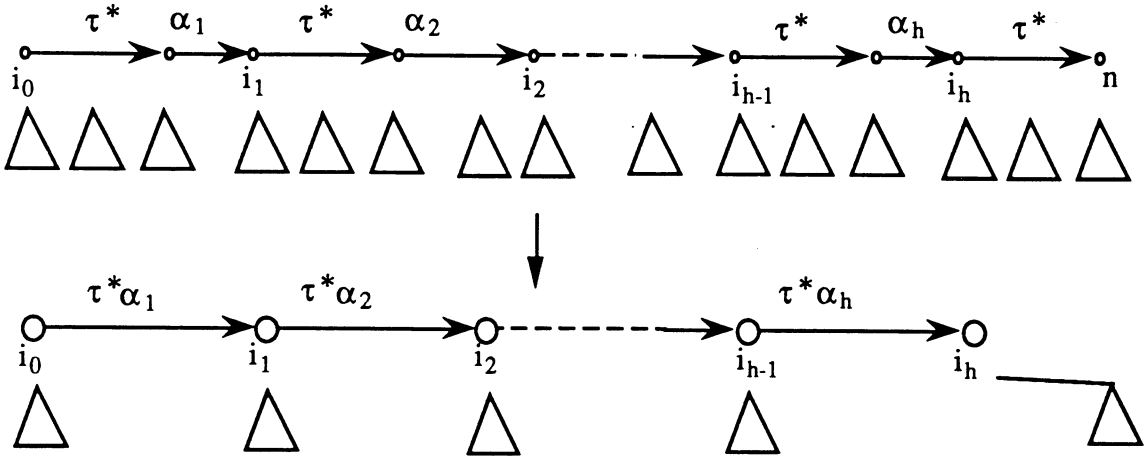


figure 3.13. Réduction d'une séquence d'explications.

En particulier, si toutes les transitions de la séquence sont muettes,

$$[x_1 \ldots x_n] = [(q_1 \ldots q_n) \lhd (x_n)]$$

si aucune transition de la séquence n'est muette,

$$[x_1 \ldots x_n] = [x_1] . [x_2] \ldots [x_n]$$

(6) $[x_1 . x_2 \ldots x_n . (x_{n+1} \ldots x_m)^\omega] = [x_{i_0}] \ldots . [x_{i_h}] . ([x_{i_{h+1}}] \ldots [x_{i_{h+t}}])^\omega$

En particulier, si aucune transition de la séquence n'est muette,

$$[x_1 . x_2 \ldots x_n . (x_{n+1} \ldots x_m)^\omega] = [x_1] . [x_2] \ldots [x_n] . ([x_{n+1}] \ldots [x_m])^\omega$$

Nous appelerons forme réduite d'une explication la forme réduite de l'explication simplifiée qui lui est associée. Par convention, nous noterons $[x] = [(x)_C]$.

La racine d'une explication réduite est toujours définie, et elle est égale à celle de l'explication initiale : $\forall x$, $racine([x]) = racine(x)$. Par contre la séquence des racines d'une séquence d'explications réduite n'est pas toujours une séquence d'exécution.

Notation. $\forall\ \alpha \in V_\lambda$, soit $\Rightarrow^\alpha$ la relation telle que : $q_1 \Rightarrow^\alpha q_2$ si et seulement si $q_1 \rightarrow\!\!\!\!\rightarrow^{\tau^*\alpha} q_2$.

La relation $\Rightarrow^\alpha$ correspond à la τ-fermeture de la relation de transition : si $q_1 \Rightarrow^{\alpha'} q_2$, q_2 est accessible à partir de q_1 par une séquence d'exécution. Des racines successives d'une séquence d'explications réduite sont liées par une relation $\Rightarrow^\alpha$.

Exemple.

Considérons les explications x et y de l'exemple du paragraphe 3.1.3.3. Leurs formes réduites sont les suivantes :

$[x] = pot[some\ \neg after(r\_rec)]after(r\_rec)$:

$(some\ \neg after(r\_rec) : (5 \models \neg after(r\_rec)\ \&\ \bot(5))$.

$(some\ \neg after(r\_rec) : (8 \models \neg after(r\_rec).\ (9 \models \neg after(r\_rec)\ \&\ \Delta(9)))$.

$4 \models after(r\_rec))$

$\&\ some\ \neg after(e\_em) : (5 \models \neg after(e\_em)\ \&\ \bot(5))$

$[y] = pot[some\ \neg after(r\_rec)]after(r\_rec)$:

$(some\ \neg after(r\_rec) : (9 \models \neg after(r\_rec)\ \&\ \bot(9))$

$(some\ \neg after(r\_rec) : (8 \models \neg after(r\_rec).\ (9 \models \neg after(r\_rec)\ \&\ \Delta(9)))$.

$4 \models after(r\_rec))$

$\&\ some\ \neg after(e\_em) : (9 \models \neg after(e\_em)\ \&\ \bot(9))$

Entre les racines des séquences d'explications nous avons les relations :

$5 \Rightarrow^{e\_em} 8 \Rightarrow^{r\_rec} 4$,

$9 \Rightarrow^{e\_em} 8 \Rightarrow^{r\_rec} 4$,

et $8 \Rightarrow^{\lambda} 9$

Les transitions muettes ont disparu.

Dans cet exemple, les explications $[x]$ et $[y]$ ne diffèrent que sur un point : leur racine (5 pour $[x]$, 9 pour $[y]$). Ces explications sont des explications équivalentes.

3.1.4. Explications réduites équivalentes.

Nous définissons une relation d'équivalence entre explications réduites : en particulier, des explications identiques au nom des états près sont équivalentes, comme par exemple les explications [x] et [y] de l'exemple précédent.

Définition. Etant donné un critère d'observation C, soit $\leftrightarrow_C$ la plus faible relation de $[\mathfrak{X}] \times [\mathfrak{X}]$ telle que :

- $q_1 \models f_1 \leftrightarrow_C q_2 \models f_2$ si et seulement si $f_1 \Leftrightarrow f_2$.
- $\perp(q_1) \leftrightarrow_C \perp(q_2)$
- $\Delta(q_1) \leftrightarrow_C \Delta(q_2)$
- $f_1 : (x_1) \leftrightarrow_C f_2 : (x_2)$ si et seulement si $f_1 \Leftrightarrow f_2$ et $x_1 \leftrightarrow_C x_2$.
- $x_1 \,\&\, \ldots \&\, x_n \leftrightarrow_C x'_1 \,\&\, \ldots \,\&\, x'_m$ si et seulement si :

  $\forall i = 1\ldots n, \exists j = 1\ldots m$ tel que $x_i \leftrightarrow_C x'_j$

  et

  $\forall j = 1\ldots m, \exists i = 1\ldots n$ tel que $x_i \leftrightarrow_C x'_j$.
- $x_1.x_2\ldots x_n \leftrightarrow_C x'_1.x'_2\ldots x'_m$ si et seulement si :

  $x_1 \leftrightarrow_C x'_1$

  et

  $\forall\, \alpha \in V_\lambda$:

  $[(racine(x_1) \Rightarrow^\alpha racine(x_2))$

  implique $((racine(x'_1) \Rightarrow^\alpha racine(x'_2))$ et $x_2\ldots x_n \leftrightarrow_C x'_2 \ldots x'_m)$

  et

  $(racine(x'_1) \Rightarrow^\alpha racine(x'_2))$

  implique $((racine(x_1) \Rightarrow^\alpha racine(x_2))$ et $x_2\ldots x_n \leftrightarrow_C x'_2 \ldots x'_m)]$

- $x_1.x_2\ldots x_k.(x_{k+1}\ldots x_n)^\omega \leftrightarrow_C x'_1.x'_2\ldots x'_h.(x'_{h+1}\ldots x'_m)^\omega$ si et seulement si il existe des entiers i, j, p, q tels que $k+1 \le i \le n$, $h+1 \le j \le m$ et :

  $x_1.x_2\ldots x_k.(x_{k+1}\ldots x_n)^p.x_{k+1}\ldots x_i \leftrightarrow_C x'_1.x'_2\ldots x'_h.(x'_{h+1}\ldots x'_m)^q.x'_{h+1}\ldots x'_j$

Propriété 3.4. Si (q_1, q_2) est une transition muette et si x est une explication de racine q_2, alors :

$[(q_1, q_2)\triangleleft(x)] \leftrightarrow_C [x]$

La démonstration de cette propriété est immédiate d'après la définition du préfixage.

Définition. Les ensembles d'explications réduites $[X_1]$ et $[X_2]$ sont équivalents modulo C, et on note $[X_1] \leftrightarrow_C [X_2]$, si et seulement si :

$\forall x_1 \in [X_1], \exists x_2 \in [X_2]$ tel que $x_1 \leftrightarrow_C x_2$

et

$\forall x_2 \in [X_2], \exists x_1 \in [X_1]$ tel que $x_1 \leftrightarrow_C x_2$

Plutôt que de calculer les explications dans un modèle, puis de les réduire, il est naturel de se demander si certaines simplifications ne pourraient pas être effectuées directement au niveau du modèle. Pour cela, nous cherchons s'il existe une relation d'équivalence entre les modèles qui préserve les ensembles d'explications réduites, c'est-à-dire telle que si les modèles $M_C = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi_f)$ et $M'_C = (Q, \rightsquigarrow, q', V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi_f)$ sont équivalents alors $[Expl(M_C, q \models f)] \leftrightarrow_C [Expl(M'_C, q' \models f)]$.

Le but est d'obtenir une forme normale réduite d'un modèle observé afin d'y calculer les explications. Afin que la réduction du modèle soit la plus efficace possible, la relation que nous cherchons à définir doit être la plus faible possible.

3.2. Relations d'équivalence entre modèles observés.

Dans la suite de ce chapitre, le modèle est un modèle observé selon la critère $C = (V, f)$. Les relations d'équivalence entre modèles observés sont définies à partir des relations d'équivalence entre les états de ces modèles. Ainsi, pour toute relation d'équivalence $\approx$, il est indifférent de dire que les modèles $M_C = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi_f)$ et $M'_C = (Q, \rightsquigarrow, q', V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi_f)$ sont équivalents ou de dire que les états q et q' sont équivalents. Dans les deux cas, on emploiera la même notation :

$M_C \approx M'_C$ si et seulement si $q \approx q'$

Des relations d'équivalence entre modèles ont été définies par divers auteurs. Nous adaptons ces définitions aux modèles observés : bien qu'il n'y soit pas fait explicitement référence dans la notation de ces relations d'équivalence, leur définition dépend du critère d'observation. Nous allons dans ce paragraphe étudier leurs propriétés, par rapport aux ensembles d'explications réduites.

3.2.2. Equivalence de sûreté.

L'équivalence de sûreté a été introduite dans [Ro88] ; c'est une équivalence qui préserve les propriétés de sûreté.

Définition. Soit $\sqsubseteq$ la plus faible relation de $Q \times Q$ telle que :

$q \sqsubseteq q' \Leftrightarrow$

(1) $\pi_f(q) = \pi_f(q')$

(2) $\forall \alpha \in V_\lambda$:

$\forall p$: $(q \Rightarrow^\alpha p)$ implique $(\exists p' : q' \Rightarrow^\alpha p'$ et $p \sqsubseteq p')$

L'*équivalence de sûreté* est la relation $\approx_S$ de $Q \times Q$ telle que :

$q \approx_S q' \Leftrightarrow q \sqsubseteq q'$ et $q' \sqsubseteq q$.

Intuitivement, l'équivalence de sûreté signifie que toute séquence d'actions applicable à partir d'un état l'est à partir d'un état équivalent, exception faite des actions muettes. C'est vrai en particulier pour une séquence qui n'est maximale que dans l'un des modèles. Cette relation ne convient pas pour notre problème comme le montre l'exemple ci-dessous.

Exemple.

Considérons les modèles représentés par la figure 3.14.

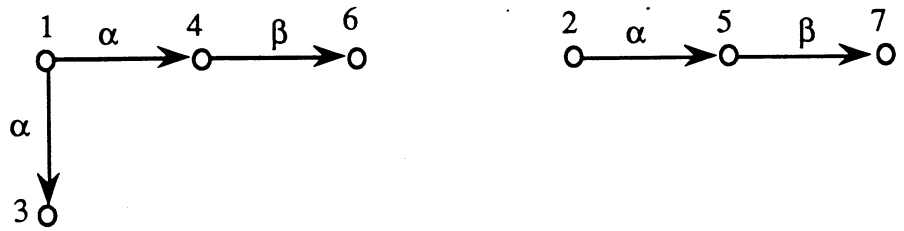


figure 3.14

Tous les états satisfont un prédicat P, et les états 3, 6 et 7 sont des états puits ; soit $f = some\ P$. On a : $1 \approx_S 2$.

L'explication $x_1 = f : (1 \models P.(3 \models P \ \&\ 3 \models puits\ ))$ appartient à $Expl(1 \models f)$, elle est telle que $[x_1] = f : (1 \models P.(3 \models P \ \&\ \bot(3))$; cependant, il n'existe aucune explication x_2 de $Expl(2 \models f)$ dont la forme réduite soit équivalente à x_1 : en effet, $Expl(2 \models f)$ est constitué d'un seul élément, $x_2 = f : (2 \models P\ .\ 5 \models P\ .\ (7 \models P \ \&\ 7 \models puits\ ))$, dont la forme réduite est $[x_2] = f : (2 \models P\ .\ 5 \models P\ .\ (7 \models P \ \&\ \bot(7)))$, et on n'a pas $x_1 \leftrightarrow_C x_2$ (les suites d'actions visibles portées par les séquences 1.3 et 2.5.7 sont différentes).

3.2.3. τ-bisimulation [Fe90]

Définition. La *τ-bisimulation* est la plus faible relation $\approx_\tau$ de $Q \times Q$ telle que :

$q \approx_\tau q' \Leftrightarrow$

(1) $\pi_f(q) = \pi_f(q')$

(2) $\forall \alpha \in V_\lambda$:

$\forall p : (q \Rightarrow^{\alpha} p)$ implique $(\exists p' : q' \Rightarrow^{\alpha} p'$ et $p \approx_{\tau} p')$
et
$\forall p' : (q' \Rightarrow^{\alpha} p')$ implique $(\exists p : q \Rightarrow^{\alpha} p$ et $p \approx_{\tau} p')$

La τ-bisimulation ne permet pas de garantir que les ensembles d'explications réduites sont équivalents dans des modèles équivalents. Il peut exister dans l'un des modèles seulement une explication construite à partir d'une séquence explicative maximale, permettant d'atteindre par des transitions muettes un état puits, ou un circuit de transitions muettes.

Exemple.
Considérons les modèles observés représentés par la figure 3.15.

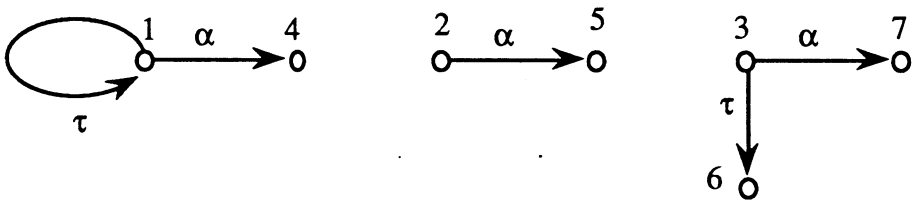


figure 3.15

Tous les états satisfont un prédicat P, et soit $f = some\ P$: on a donc $1 \approx_{\tau} 2 \approx_{\tau} 3$.
L'explication $x_1 = f : (1 \models P)^{\omega}$ appartient à $Expl(1 \models f)$, elle est telle que $[x_1] = x_1$, mais il n'existe aucune explication x_2 de $Expl(2 \models f)$ et aucune explication x_3 de $Expl(3 \models f)$ dont la forme réduite soit équivalente à $[x_1]$.
L'explication $x_3 = f : (3 \models P.\ (6 \models P\ \&\ 6 \models puits\ ))$ appartient à $Expl(3 \models f)$, elle est telle que $[x_3] = f : (3 \models P\ \&\ \perp(3))$, mais il n'existe aucune explication x_1 de $Expl(1 \models f)$ et aucune explication x_2 de $Expl(2 \models f)$ dont la forme réduite soit équivalente à $[x_3]$.

Remarque. La τ-bisimulation est plus forte que l'équivalence de sûreté. Autrement dit, $\forall q, q', q \approx_{\tau} q' \Rightarrow q \approx_S q'$.

3.2.4. Equivalence observationnelle.

L'équivalence observationnelle [Mi80] est fondée sur le fait que les transitions muettes ne sont pas observables. Elle est définie de la manière suivante :

Définition. L'*équivalence observationnelle* est la plus faible relation $\approx_O$ de $Q \times Q$ telle que :

$q \approx_O q' \Leftrightarrow$

(1) $\pi_f(q) = \pi_f(q')$
(2) $\forall \alpha \in V_\lambda$:
$\forall p$: ($q \overset{\alpha}{\rightsquigarrow} p$) implique ($\exists p'$: $q' \overset{\tau^* \alpha \tau^*}{\rightsquigarrow} p'$ et $p \approx_O p'$)
et
$\forall p'$: ($q' \overset{\alpha}{\rightsquigarrow} p'$) implique ($\exists p$: $q \overset{\tau^* \alpha \tau^*}{\rightsquigarrow} p$ et $p \approx_O p'$)
(3) $\forall p$: ($q \overset{\tau}{\rightsquigarrow} p$) implique ($\exists p'$: $q' \overset{\tau^*}{\rightsquigarrow} p'$ et $p \approx_O p'$)
et
$\forall p'$: ($q' \overset{\tau}{\rightsquigarrow} p'$) implique ($\exists p$: $q \overset{\tau^*}{\rightsquigarrow} p$ et $p \approx_O p'$)

De même que pour la τ-bisimulation, des ensembles d'explications réduites peuvent ne pas être équivalents dans des modèles équivalents, lorsque dans l'un des modèle existe une explication construite à partir d'une séquence explicative infinie qui comporte un circuit de transitions muettes.

3.3. Equivalence explicationnelle.

3.3.1. Définition

La raison pour laquelle une relation d'équivalence telle que la τ-bisimulation ne remplit pas les contraintes requises est simple : cette relation ne préserve pas dans des modèles équivalents l'existence d'états puits lorsqu'on y accède par une transition muette, ni l'existence des circuits de transitions muettes ; autrement dit, elle ne préserve pas l'existence de certaines séquences explicatives lorsqu'elles sont des séquences d'exécution maximales. L'équivalence explicationnelle est définie à partir de la τ-bisimulation en ajoutant des contraintes qui exigent dans des modèles équivalents de préserver l'existence des puits et des circuits de τ : ces contraintes sont traduites en termes de prédicats de convergence et de divergence.

Définition. Soit $C = (V, f)$. L'*équivalence explicationnelle* $\approx_E$ est la plus faible relation de $Q \times Q$ telle que :

$q \approx_E q' \Leftrightarrow$
(1) $\pi_f(q) = \pi_f(q')$
(2) $\forall \alpha \in V_\lambda$:
[$\forall p$: ($q \Rightarrow^{\alpha} p$) implique ($\exists p'$: $q' \Rightarrow^{\alpha} p'$ et $p \approx_E p'$)]
et

$[\forall p' : (q' \Rightarrow^{\alpha} p')$ implique $(\exists p : q \Rightarrow^{\alpha} p$ et $p \approx_E p')]$

(3) $\perp(q) \Leftrightarrow \perp(q')$

(4) $\Delta(q) \Leftrightarrow \Delta(q')$

Exemple

Considérons les modèles représentés par la figure 3.16. Le premier est le modèle du système *E/R* observé selon le critère $C = (V, f)$ avec $V = \{e\_em, r\_rec\}, f = pot\ some\ \neg after(r\_rec)$. Le second est un modèle observé selon le même critère.

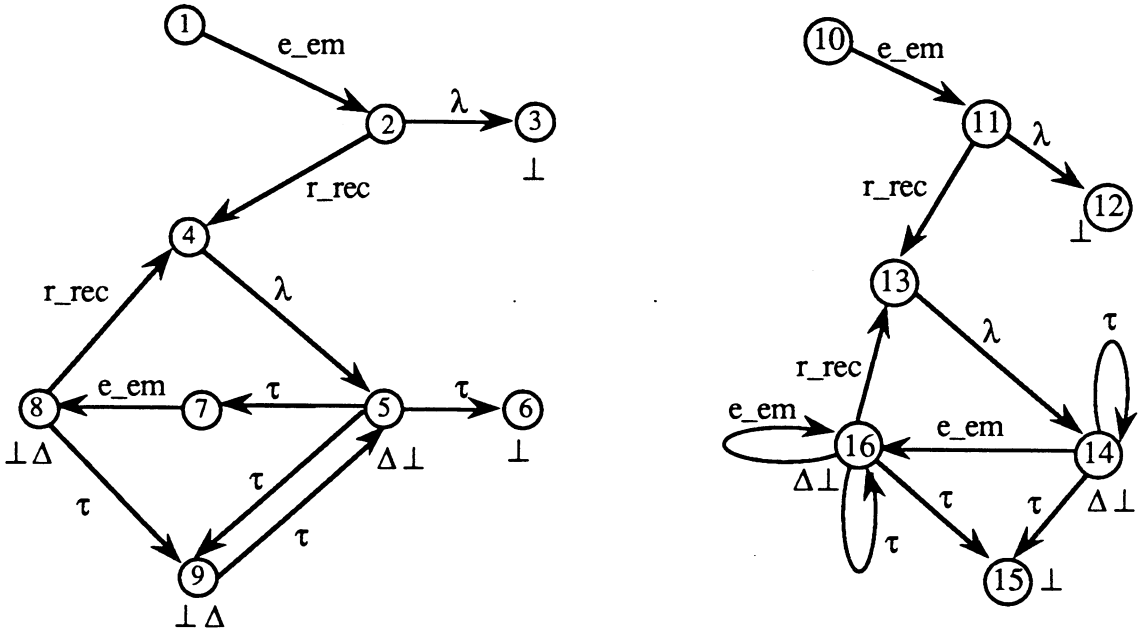


figure 3.16. 1 ≈$_f$ 10

On a : $1 \approx_E 10$. En effet :

$1 \approx_E 10$ parce que	$\pi_f(1) = \pi_f(10)$
	$1 \Rightarrow^{e\_em} 2$, $10 \Rightarrow^{e\_em} 11$, et $2 \approx_E 11$
	$\perp(1) \Leftrightarrow \perp(10)$, $\Delta(1) \Leftrightarrow \Delta(10)$
$2 \approx_E 11$ parce que	$\pi_f(2) = \pi_f(11)$
	$2 \Rightarrow^{\lambda} 3$, $11 \Rightarrow^{\lambda} 12$, et $3 \approx_E 12$
	$2 \Rightarrow^{r\_rec} 4$, $11 \Rightarrow^{r\_rec} 13$, et $4 \approx_E 13$
	$\perp(2) \Leftrightarrow \perp(11)$, $\Delta(2) \Leftrightarrow \Delta(11)$
$3 \approx_E 12$ parce que	$\pi_f(3) = \pi_f(12)$, $\perp(3) \Leftrightarrow \perp(12)$, $\Delta(3) \Leftrightarrow \Delta(12)$
$4 \approx_E 13$ parce que	$\pi_f(9) = \pi_f(18)$
	$4 \Rightarrow^{\lambda} 5$, $13 \Rightarrow^{\lambda} 14$, et $5 \approx_E 14$
	$\perp(4) \Leftrightarrow \perp(13)$, $\Delta(4) \Leftrightarrow \Delta(13)$
$5 \approx_E 14$ parce que	$\pi_f(5) = \pi_f(14)$

$5 \Rightarrow^{e\_em} 8$, $14 \Rightarrow^{e\_em} 16$, et $8 \approx_E 16$
$\perp(5) \Leftrightarrow \perp(14)$, $\Delta(5) \Leftrightarrow \Delta(14)$

$8 \approx_E 16$ parce que $\pi_f(8) = \pi_f(16)$
$8 \Rightarrow^{e\_em} 8$, $16 \Rightarrow^{e\_em} 16$, et $8 \approx_E 16$
$8 \Rightarrow^{r\_rec} 4$, $16 \Rightarrow^{r\_rec} 13$, et $4 \approx_E 13$
$\perp(8) \Leftrightarrow \perp(16)$, $\Delta(8) \Leftrightarrow \Delta(16)$

Dans ces modèles, on vérifie que l'on a : $[Expl(1 \models f)] \leftrightarrow_C [Expl(10 \models f)]$

Propriété 3.5.
$\forall\ C = (V, f),\ \forall q, p \in Q, q \approx_E p \Rightarrow ([Expl(q \models f)] \leftrightarrow_C [Expl(p \models f)])$

Démonstration.
Il suffit de montrer :
$q \approx_E p \Rightarrow\ (\forall x \in Expl(q \models f), \exists\, y \in Expl(p \models f), [x] \leftrightarrow_C [y])$
Notons $P(f)$ cette propriété.

Le principe de la démonstration est une récurrence sur les sous-formules de f, et ne présente pas de difficulté. Aussi nous contenterons nous de traiter les deux cas suivants :
$f = some[f_1]f_2$, $x \in Expl(q \models f)$, et

1) $[x] = f : ([x_{i_0}] \ldots [x_{i_{h-1}}].[x_{i_h} \,\&\, \perp(q_{i_h})])$
2) $[x] = f : ([x_{i_0}] \ldots [x_{i_h}].([x_{i_{h+1}}] \ldots [x_{i_{h+t}}])^{\omega})$

où $\forall\ k, x_{i_k} \in Expl(q_{i_k} \models f_1 \wedge f_2)$.
Les autres cas à considérer sont triviaux ou se traitent de manière analogue au cas 1).

On suppose que f_1 et f_2 satisfont l'hypothèse d'induction : $P(f_1)$ et $P(f_2)$. Il est aisé de vérifier que : ($P(f_1)$ et $P(f_2)$) $\Rightarrow P(f_1 \wedge f_2)$. Nous montrons alors que dans les cas 1) et 2) :

$q \approx_E p \Rightarrow \exists\, y \in Expl(p \models f), [x] \leftrightarrow_C [y]$

Nous nous appuierons sur les remarques suivantes :

• si $g \in sf(f)$, $r \models g$, et $r \rightarrow^{\tau^*} r'$, alors il existe $x' \in Expl(r' \models g)$; par conséquent, si $r \rightarrow^{\tau} r_1 \rightarrow^{\tau} r_2 \rightarrow^{\tau} \ldots \rightarrow^{\tau} r_k$, alors, $\forall\ i = 1 \ldots k, \exists\, x_i \in Expl(r_i \models g)$.

• si $q \approx_E p$, $q_{i_0} = q$, et $q_{i_0} \Rightarrow^{\alpha_1} q_{i_1} \ldots \Rightarrow^{\alpha_{h-1}} q_{i_{h-1}} \Rightarrow^{\alpha_h} q_{i_h}$, alors il existe $p_0, \ldots, p_h$ tels que :

$p_0 = p$,
$p_0 \Rightarrow^{\alpha_1} p_1 \ldots \Rightarrow^{\alpha_{h-1}} p_{h-1} \Rightarrow^{\alpha_h} p_h$.
$\forall\ j = 0 \ldots h, q_{i_j} \approx_E p_j$.

1) $[x] = f : ([x_{i_0}]....[x_{i_{h-1}}].[x_{i_h} \,\&\, \perp(q_{i_h})])$.

On a $q \approx_E p$ et $q \Rightarrow^{\alpha_1} q_{i_1}... \Rightarrow^{\alpha_{h-1}} q_{i_{h-1}} \Rightarrow^{\alpha_h} q_{i_h}$, donc, en considérant les états $p_0, ..., p_h$ de la remarque ci-dessus :

- par hypothèse d'induction, il existe des explications $y_0 ... y_h$ telles que pour $k = 0...h$, $y_k \in Expl(p_k \models f_1 \wedge f_2)$ et $[x_{i_k}] \leftrightarrow_C [y_k]$.
- pour $k = 0...h\text{-}1$, considérons les états intermédiaires de la séquence $p_k...p_{k+1}$: $p_k \rightsquigarrow^{\tau} r_{k_1} \rightsquigarrow^{\tau} r_{k_2} \rightsquigarrow^{\tau} ... \rightsquigarrow^{\tau} r_{k_{m(k)}} \rightsquigarrow^{\alpha_{k+1}} p_{k+1}$; puisque $p_k \models f_1 \wedge f_2$, il existe pour chacun de ces états une explication de f_1 et de f_2, et donc de $f_1 \wedge f_2$.
- $\perp(q_{i_h})$ et $p_h \approx_E q_{i_h} \Rightarrow \perp(p_h)$. Donc il existe une séquence $p_h \rightsquigarrow^{\tau} p_{h+1} ... \rightsquigarrow^{\tau} p_m$ telle que $p_m \models puits$. De plus, pour chacun de ces états, il existe une explication de $f_1 \wedge f_2$.

On en déduit l'existence d'une explication y de $p \models f$ telle que $[y] = [y_0]...([y_h] \,\&\, \perp(p_h))$, et $[x] \leftrightarrow_C [y]$.

2) $[x] = f : ([x_{i_0}]....[x_{i_h}].([x_{i_{h+1}}]...[x_{i_{h+t}}])^{\omega})$.

On a $q \approx_E p$ et $q \Rightarrow^{\alpha_1} q_{i_1}... \Rightarrow^{\alpha_h} q_{i_h} \Rightarrow^{\alpha_{h+1}} q_{i_{h+1}}... \Rightarrow^{\alpha_{h+t}} q_{i_{h+t}} \Rightarrow^{\alpha_{h+1}} q_{i_{h+1}}$. Par conséquent, si $q_{i_0} = q$, il existe des états $p_0, ..., p_h, p_{h+1}{}^1, ..., p_{h+t}{}^1, p_{h+1}{}^2, ..., p_{h+t}{}^2, ..., p_{h+1}{}^m, ..., p_{h+t}{}^m...$ tels que :

$p_0 = p$,

$p_0 \Rightarrow^{\alpha_1} p_1... \Rightarrow^{\alpha_h} p_h \Rightarrow^{\alpha_{h+1}} p_{h+1}{}^1 \Rightarrow ... \Rightarrow^{\alpha_{h+t}} p_{h+t}{}^1 \alpha_{h+1} p_{h+1}{}^2 \Rightarrow ... \Rightarrow^{\alpha_{h+t}} p_{h+t}{}^m...$

$\forall j = 0...h, q_{i_j} \approx_E p_j, \forall j = 1...t, \forall m, q_{i_{h+j}} \approx_E p_{h+j}{}^m$

Donc,

- par hypothèse d'induction, il existe des explications $y_0, ..., y_h, y_{h+1}{}^1, ...y_{h+t}{}^1, y_{h+1}{}^2, ...y_{h+t}{}^2, ..., y_{h+1}{}^m, ...y_{h+t}{}^m...$ de $f_1 \wedge f_2$, dont les racines sont ces états, et telles que :

$\forall j = 0...h, [x_{i_j}] \leftrightarrow_C [y_j], \forall j = 1...t, \forall m, [x_{i_{h+j}}] \leftrightarrow_C [y_{h+j}{}^m]$

- de même que dans le cas précédent, il est possible "d'intercaler" entre ces explications des explications de $f_1 \wedge f_2$ de manière à ce que la suite des racines soit une séquence d'exécution,

D'autre part, le graphe du programme étant fini, il existe deux indices m et n tels que $m < n$ et $p_{h+t}{}^m = p_{h+t}{}^n$.

On en déduit qu'il existe une explication y de $p \models f$ telle que

$[y] = f : ([y_0]....[y_h].[y_{h+1}{}^1]...[y_{h+t}{}^1]...[y_{h+t}{}^m]([y_{h+1}{}^{m+1}]...[y_{h+t}{}^n])^{\omega}$

et on a $[x] \leftrightarrow_C [y]$.

$\therefore$

Conséquence. L'équivalence explicationnelle satisfait donc les contraintes exigées :

$\forall p, q \in Q, q \approx_E p \Rightarrow ([Expl(q \models f)] \leftrightarrow_C [Expl(p \models f)])$

Ainsi, le modèle dans lequel sont élaborées les explications réduites de f peut être choisi parmi des modèles équivalents.

3.3.2. Comparaison avec les autres relations d'équivalence.

Les conditions (1) et (2) de la définition de l'équivalence explicationnelle traduisent la τ-bisimulation, l'équivalence explicationnelle est donc plus forte que cette relation, elle-même plus forte que l'équivalence de sûreté ; on obtient la classification suivante ($\supset$ représente la relation "plus faible que") :

$$\approx_S \supset \approx_\tau \supset \approx_E$$

Par contre, l'équivalence observationnelle et l'équivalence explicationnelle ne sont pas comparables entre elles, comme le montrent les exemples des figures 3.17 et 3.18.

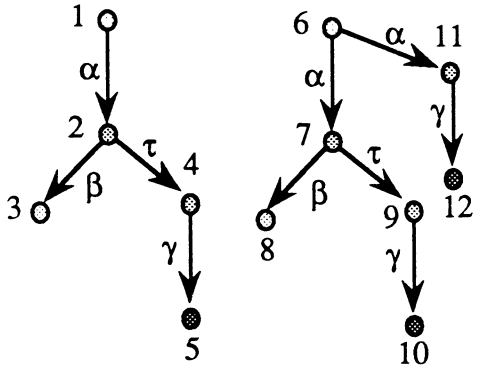


figure 3.17

Si $\pi_f(1) = \pi_f(6)$,
$\pi_f(2) = \pi_f(4) = \pi_f(7) = \pi_f(9) = \pi_f(11)$,
$\pi_f(3) = \pi_f(8)$ et $\pi_f(5) = \pi_f(10) = \pi_f(12)$
on a $1 \approx_O 6$, et $1 \not\approx_E 6$,

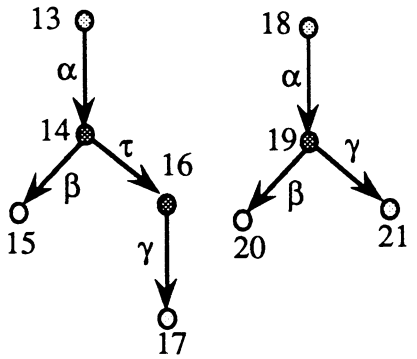


figure 3.18

si $\pi_f(13) = \pi_f(18)$, $\pi_f(14) = \pi_f(16) = \pi_f(19)$,
$\pi_f(15) = \pi_f(20)$, $\pi_f(17) = \pi_f(21)$,
on a $13 \approx_E 18$, et $13 \not\approx_O 18$.

En effet, l'équivalence explicationnelle est issue de la τ-bisimulation : dans le premier exemple, lorsqu'on passe de l'état 6 à l'état 11 par la transition α, il n'est plus possible d'exécuter l'action β, alors que toute exécution de α à partir de l'état 1, conduit à l'état 2, état à partir duquel l'action β peut être exécutée. Inversement, les comportements observables à partir des états 1 et 6 sont les mêmes : ils sont obervationnellement équivalents.
Dans le deuxième exemple, par contre, on peut observer l'action α à partir de l'état 13 et se trouver dans l'état 16 à partir duquel l'action n'est plus possible, alors que ce comportement n'existe pas à partir de l'état 18 : 13 et 18 ne sont pas observationnellement équivalents.

Cependant, ils sont équivalents pour la τ-bisimulation, et par conséquent ici, pour l'équivalence explicationnelle.

Remarque. F. Vernadat [Ve89] propose une variante de l'équivalence observationnelle appelée équivalence de comportement, et qui préserve les possibilités de divergence à partir d'états équivalents. On constate sur les exemples précédents que l'équivalence explicationnelle et l'équivalence de comportement ne sont pas comparables entre elles.

Comparaison avec l'équivalence de "stuttering" [CBG87] et les bisimulations de branchement [DV90].

L'équivalence explicationnelle, l'équivalence de "stuttering" et les bisimulations de branchement sont des relations d'équivalence entre modèles de nature différente. La première est une relation entre modèles observés, la seconde entre structures de Kripke, c'est-à-dire indépendamment des actions qui étiquettent les transitions, et les bisimulations de branchement concernent des systèmes de transitions étiquetés, indépendamment des prédicats de base satisfaits par les états. Leur comparaison nécessite de définir une structure commune assurant la cohérence entre les actions qui étiquettent les transitions et les prédicats satisfaits par les états.

Cette structure commune est un modèle $(Q, \rightarrowtail, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \pi)$ ayant les propriétés suivantes :

- $(q_1 \rightarrowtail q_2$ et $\pi(q_1) = \pi(q_2)) \Leftrightarrow q_1 \rightarrowtail^\tau q_2$
- il existe une bijection ϕ de $2^{\mathcal{P}} \times 2^{\mathcal{P}}$ dans $V_\lambda \cup \{\tau\}$ telle que :

$$q_1 \rightarrowtail^\alpha q_2 \Rightarrow \alpha = \phi(\pi(q_1), \pi(q_2))$$

Les relations d'équivalence définies dans [CBG87] et [DV90] ne sont pas comparables avec l'équivalence explicationnelle car elles ne font pas de distinction entre les notions de convergence et de divergence. Afin de pouvoir établir la comparaison avec l'équivalence explicationnelle, nous proposons une légère modification de la définition de ces équivalences permettant de distinguer les convergences des divergences.

• **Définition.** L'équivalence de "stuttering"est la relation $\approx_B = \bigcap_{i \in \mathbb{N}} (\approx_{Bi})$ de $Q \times Q$ où :

$$q \approx_{B0} q' \Leftrightarrow (1)\ \pi(q) = \pi(q')$$
$$(2)\ \perp(q) \Leftrightarrow \perp(q')$$
$$(3)\ \Delta(q) \Leftrightarrow \Delta(q').$$

et pour $i \geq 0$, $q \approx_{Bi+1} q' \Leftrightarrow$

$[\forall s \in Ex(q) : [\exists\, s' \in Ex(q'), \exists\, s_1, \ldots, s_n, \ldots, \exists\, s'_1, \ldots, s'_n, \ldots$ tels que :

$\forall i, s_i \in Q^* \backslash \{\varepsilon\}, s'_i \in Q^* \backslash \{\varepsilon\}$

et

$s = s_1 \ldots s_n \ldots$, $s' = s'_1 \ldots s'_n \ldots$

et

$\forall k, \forall p \in états(s_k), \forall p' \in états(s'_k) : p \approx_{Bi} p']$

et

$\forall s' \in Ex(q') : [\exists s \in Ex(q), \exists s_1, \ldots, s_n, \ldots, \exists s'_1, \ldots, s'_n, \ldots$ tels que :

$\forall i, s_i \in Q^* \backslash \{\varepsilon\}, s'_i \in Q^* \backslash \{\varepsilon\}$

et

$s = s_1 \ldots s_n \ldots$, $s' = s'_1 \ldots s'_n \ldots$

et

$\forall k, \forall p \in états(s_k), \forall p' \in états(s'_k) : p \approx_{Bi} p']$

"Stuttering" se traduit en français par "bégaiement". Les états q et q' sont équivalents si pour toute séquence s à partir de q, il existe une séquence s' à partir de q' telle que les suites de classes d'équivalences définies par les états de s et de s' sont les mêmes (figure 3.19).

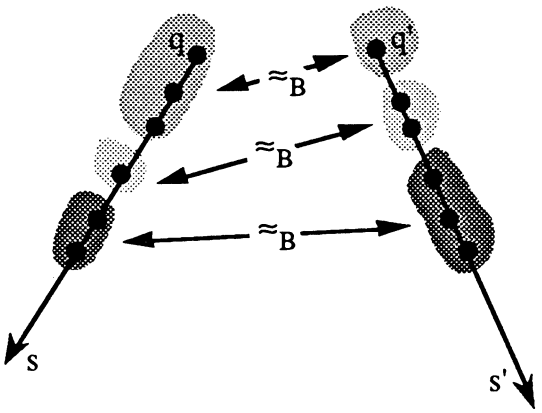


figure 3.19. $q \approx_B q'$

Afin de pouvoir comparer cette relation d'équivalence avec l'équivalence explicationnelle, nous prendrons pour ces deux équivalences la même partition initiale définie par π, et nous donnons une définition de l'équivalence explicationnelle sous la forme de la limite d'une suite de relations :

$\approx_E = \bigcap_{i \in \mathbb{N}} (\approx_{Ei})$ où :

$q \approx_{E0} q' \Leftrightarrow$ (1) $\pi(q) = \pi(q')$

(2) $\perp(q) \Leftrightarrow \perp(q')$

(3) $\Delta(q) \Leftrightarrow \Delta(q')$

pour $i \geq 0$, $q \approx_{Ei+1} q' \Leftrightarrow \forall \alpha \neq \tau$,

$[\forall p : (q \Rightarrow^{\alpha} p) \Rightarrow (\exists p' : q' \Rightarrow^{\alpha} p'$ et $p \approx_{Ei} p')]$

et

$[\forall p' : (q' \Rightarrow^{\alpha} p') \Rightarrow (\exists p : q \Rightarrow^{\alpha} p$ et $p \approx_{Ei} p')]$

On constate sur un exemple que cette variante de l'équivalence explicationnelle n'est pas plus forte que l'équivalence de stuttering :

Exemple.
Considérons les modèles observés représentés sur la figure 3.20. On a $1 \approx_E 5$, mais pas $1 \approx_B 5$:

en effet, les classes d'équivalence de $\approx_{B0}$ sont : {1,2,5,6}, {3,8}, {4,7}

les classes d'équivalence de $\approx_{B1}$ sont : {1,5},{2},{6}, {3,8}, {4,7}

les classes d'équivalence de $\approx_{B2}$ sont : {1}, {5},{2},{6}, {3,8}, {4,7}

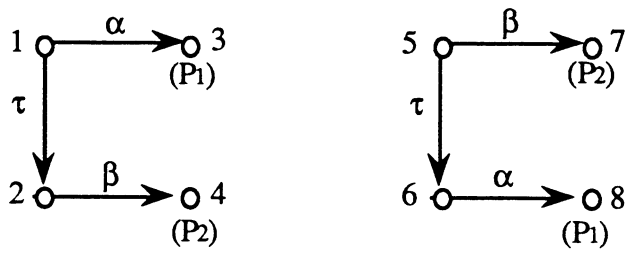


figure 3.20

Propriété 3.6. L'équivalence explicationnelle est plus faible que l'équivalence de stuttering : $\approx_E \supset \approx_B$

Démonstration.
Nous montrons par récurrence sur i la propriété :

$q \approx_{Bi} q' \Rightarrow q \approx_{Ei} q'$.

Si $i = 0$, les deux relations $\approx_{B0}$ et $\approx_{E0}$ sont égales.
Si $i \geq 0$, supposons que l'on ait $\forall\, q, q' \in Q$, $q \approx_{Bi} q' \Rightarrow q \approx_{Ei} q'$, et montrons $q \approx_{Bi+1} q' \Rightarrow q \approx_{Ei+1} q'$.
Soit $\alpha \neq \tau$, et supposons qu'il existe un état r tel que $q \Rightarrow^{\alpha} r$. Il existe une séquence $q_1 \ldots . q_n . q_{n+1}$ telle que : $q_1 = q$, $q_{n+1} = r$, et $\forall i<n$, $\pi(q_i) = \pi(q_{i+1})$, et $\pi(q_n) \neq \pi(q_{n+1})$. Or $q \approx_{Bi+1} q' \Rightarrow \exists\, q'_1 \ldots q'_m . q'_{m+1}$ telle que : $q'_1 = q'$, $q_{n+1} \approx_{Bi} q'_{m+1}$ et $\forall i \leq m$, $\pi(q'_i) = \pi(q_1)$, et $\pi(q'_{m+1}) = \pi(q_{n+1})$. Par conséquent, $\forall i<m$, $q'_i \rightsquigarrow^{\tau} q'_{i+1}$, et $q'_m \rightsquigarrow^{\alpha} q'_{m+1}$. De plus, $q_{n+1} \approx_{Bi} q'_{m+1} \Rightarrow q_{n+1} \approx_{Ei} q'_{m+1}$. On en conclut :
$q \approx_{Bi+1} q' \Rightarrow [q \Rightarrow^{\alpha} r \Rightarrow \exists\, r'$ tel que $q' \Rightarrow^{\alpha} r'$ et $r \approx_{Ei} r']$
c'est-à-dire : $q \approx_{Bi+1} q' \Rightarrow q \approx_{Ei+1} q'$.
$\therefore$

• Dans la définition des bisimulations de branchement proposée dans [DV90], les notions de convergence et de divergence sont confondues : au modèle est ajouté un état "livelock", successeur des états puits et des états situés sur un circuit de transitions muettes. Afin que la comparaison avec l'équivalence explicationnelle ait un sens, nous proposons une variante permettant de distinguer les états qui peuvent converger de ceux qui peuvent diverger, en ajoutant au modèle non pas un seul état mais deux : le premier, $q_\perp$, sera atteint à partir des puits par une transition étiquetée par une action spéciale $\tau_\perp$, le second, q_Δ, à partir des états des circuits de transitions muettes, par une action spéciale τ_Δ.

Définition. Une relation R de $(Q \cup \{q_\perp, q_\Delta\}) \times (Q \cup \{q_\perp, q_\Delta\})$, symétrique, est une bisimulation de branchement si :

$q\ R\ q' \Rightarrow$

(1) $\forall \alpha \in V_\lambda \cup \{\tau_\perp, \tau_\Delta\}$:

$\forall p : (q \overset{\alpha}{\rightsquigarrow} p) \Rightarrow (\exists p', r' : q' \overset{\tau^*}{\rightsquigarrow} r' \overset{\alpha}{\rightsquigarrow} p'$ et $q\ R\ r'$ et $p\ R\ p')$

(2) $\forall p : (q \overset{\tau}{\rightsquigarrow} p) \Rightarrow (p\ R\ q')$

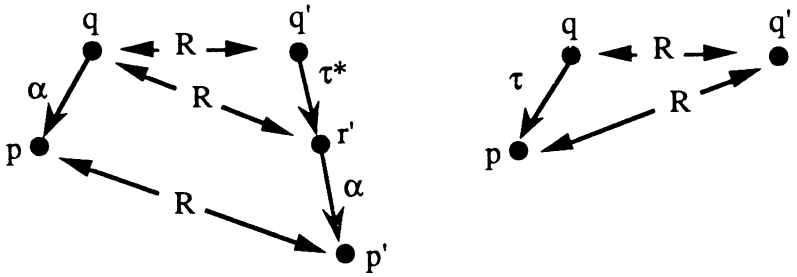


figure 3.21. q R q'.

Par définition, l'équivalence de branchement est la relation $\approx_{BB}$ telle que : $q \approx_{BB} q'$ si et seulement si il existe une bisimulation de branchement R telle que $q\ R\ q'$.

Propriété 3.7. [DV90]. $q \approx_B q' \Leftrightarrow (q \approx_{BB} q'$ et $\pi(q) = \pi(q'))$

Autrement dit, q et q' sont "stuttering"-équivalents si et seulement si il existe une bisimulation de branchement qui relie q et q' et s'ils satisfont le même ensemble de prédicats de base. On en déduit que, de même que l'équivalence de "stuttering", l'équivalence de branchement est plus forte que l'équivalence explicationnelle.

3.3.3. Discussion de quelques variantes de l'équivalence explicationnelle

Etant donné un critère d'observation (V, f), l'équivalence explicationnelle est telle que dans les modèles équivalents, les explications réduites de f sont équivalentes (propriété 3.5). Cependant, la relation que nous avons définie n'est pas la plus faible relation qui remplit cette contrainte. Nous proposons une légère modification de la définition des prédicats de convergence et de divergence liés au critère d'observation qui remédie en partie à ce défaut. La définition des transitions muettes reste cependant contraignante, puisque ne peuvent être reliés par une transition muette que des états qui satisfont les mêmes sous-formules de f. Nous montrons qu'en modifiant cette définition en ne tenant compte que des prédicats de base, les explications réduites ne sont pas nécessairement préservées dans des modèles équivalents. On remarque également qu'une version affaiblie de l'équivalence explicationnelle dans laquelle il n'est pas exigé que des états équivalents satisfassent les mêmes sous-formules de f, mais seulement les mêmes prédicats de base, ne préserve pas la satisfaction des formules *CTL*. En conclusion, il semble difficile de définir la plus faible relation d'équivalence préservant les explications réduites sans calculer préalablement ces explications. Le problème reste donc ouvert.

3.3.3.1. Proposition de modification de la définition des prédicats de convergence et de divergence.

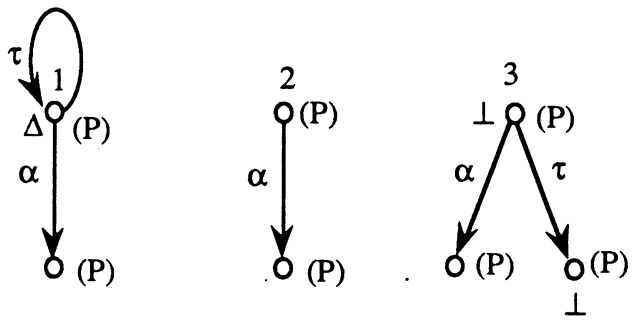


figure 3.22

Considérons les modèles représentés sur la figure 3.22, observés selon un critère $C = (V, f)$ tel que $f = pot\ P$. On a : $[Expl(1 \models f)] \leftrightarrow_C [Expl(2 \models f)] \leftrightarrow_C [Expl(3 \models f)]$, et cependant, ces modèles sont deux à deux non équivalents. En effet, 1 peut diverger mais pas converger, 3 peut converger, mais pas diverger, et 2 ne peut ni converger, ni diverger. Or, puisque $f = pot\ P$, les propriétés de convergence et de divergence n'interviennent dans aucune explication de f. Par contre, si f est de la forme $some[f_1]f_2$, ou, d'une façon plus générale, si

some$[f_1]f_2$ est une sous-formule de f, ces propriétés doivent être considérées dans les explications de f.
Par conséquent, nous pouvons modifier la définition des prédicats $\perp$ et Δ de manière à ce que les propriétés de convergence et de divergence ne soient effectivement discriminantes que lorsqu'elles peuvent apparaître dans les explications de f.

Définition. Les prédicats de convergence et de divergence pour le critère d'observation (V, f) sont définis par :

s'il existe une formule du type *some*$[f_1]f_2$ appartenant à $sf(f)$ alors :

$\forall\, q \in Q, \perp(q) \Leftrightarrow (\exists\, q'$ tel que : $q \rightsquigarrow^{\tau^*} q'$ et $succ(q') = \varnothing)$

$\forall\, q \in Q, \Delta(q) \Leftrightarrow (\exists\, q'$ tel que : $q \rightsquigarrow^{\tau^*} q'$ et $q' \rightsquigarrow^{\tau^*\tau} q')$

sinon :

$\forall\, q \in Q, \perp(q) \Leftrightarrow$ faux et $\Delta(q) \Leftrightarrow$ faux.

On vérifie aisément qu'en remplaçant la précédente définition des prédicats $\perp$et Δ par celle-ci, l'équivalence explicationnelle vérifie toujours la propriété 3.5. Cette modification affaiblit la relation d'équivalence.

Considérons par exemple les modèles représentés par la figure 3.23.

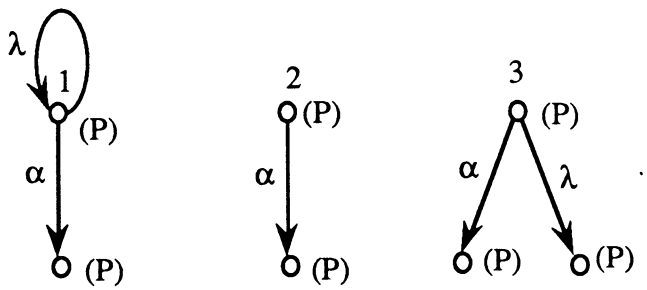


figure 3.23

Supposons que tous les états vérifient un prédicat P, et soit $V = \{\alpha\}$.

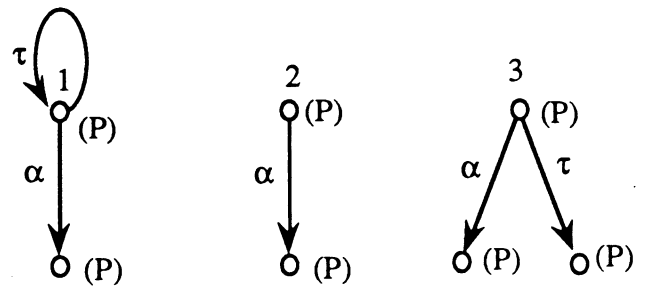


figure 3.24

Soit $C_1 = (V, pot\ P)$, et $C_2 = (V, some\ P)$. Les modèles observés selon C_1 et C_2 sont les mêmes (figure 3.24). Cependant, si la formule est *pot P*, les prédicats $\perp$ et Δ sont vrais pour tous les états, par conséquent, on a pour le critère C_1 :

$1 \approx_E 2 \approx_E 3$

Par contre, si la formule est *some P*, le prédicat $\perp$ est vrai pour l'état 3, mais ni pour 1, ni pour 2, et le prédicat Δ est vrai pour l'état 1, mais ni pour 2, ni pour 3 (figure 3.22). Par conséquent, pour le critère C_2 :

$1 \not\approx_E 2$, $2 \not\approx_E 3$ et $1 \not\approx_E 3$.

Cependant, il apparaît que la modification proposée n'est pas suffisante : en effet, il suffit qu'une formule telle que $some[f_1]f_2$ soit une sous-formule de f pour distinguer des modèles qui ne diffèrent par exemple que par la présence d'un circuit de transitions muettes, même si ce circuit n'intervient dans aucune explication de f. Par conséquent, il est possible d'avoir $[Expl(q \models f)] \leftrightarrow_C [Expl(q' \models f)]$ sans avoir $q \approx_E q'$.

Par exemple si $V = \{\alpha, \beta\}, f = some\ P_1 \wedge pot\ P_2$, et $C = (V, f)$, les modèles observés selon C représentés sur la figure 3.25 ne sont pas équivalents.

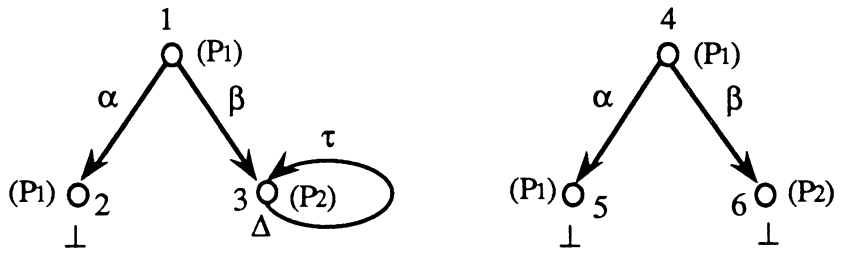


figure 3.25

Cependant, les ensembles d'explications réduites dans ces modèles sont équivalents :

$[Expl(1 \models f)] \leftrightarrow_C [Expl(4 \models f)]$

Le fait que l'état 3 est situé sur un circuit, ou que l'état 6 est un puits n'intervient dans aucune explication de $some\ P_1$: mais pour le savoir, il aurait fallu connaître ces explications a priori !

3.3.3.2. Proposition de modification de la définition des transitions muettes.

Considérons les modèles de la figure 3.26, observés selon le critère $(\{\alpha\}, pot\ P)$:

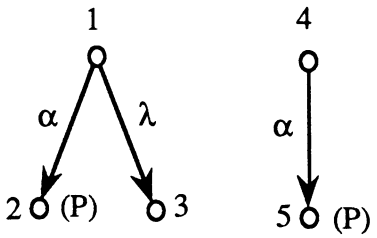


figure 3.26

On a : $[Expl(1 \models pot\ P)] \leftrightarrow_C [Expl(4 \models pot\ P)]$, et cependant : $1 \neq_E 4$.
En effet, $1 \models pot\ P$, $3 \not\models pot\ P$, par conséquent, la transition (1, 3) n'est pas muette. Puisqu'il n'existe pas de transition issue de l'état 4, étiquetée par λ, les états 1 et 4 ne peuvent être équivalents.
La définition des transitions muettes semble donc trop contraignante : dans l'exemple ci-dessus, on souhaiterait que les états 1 et 3 qui satisfont les mêmes prédicats de base soient reliés par une transition muette. Nous proposons donc de modifier la définition de la relation $\rightarrow^\tau$ en conséquence :

Définition. Etant donné $C = (V, f)$, soit π^0_f la fonction telle que :

$$\forall\ q \in Q : \pi^0_f(q) = \{P \in sf(f) \cap \mathcal{P} / q \models P\ \}$$

La relation $\rightarrow^\tau$ est alors la suivante :

$q_1 \rightarrow^\tau q_2$ si et seulement si $\exists\ \alpha \in (Act_\lambda \setminus V)$ tel que $q_1 \rightarrow^\alpha q_2$ et $\pi^0_f(q_1) = \pi^0_f(q_2)$

On constate alors que l'équivalence explicationnelle ne préserve plus les explications réduites. Considérons par exemple les modèles représentés sur la figure 3.27, observés selon le critère $(\{\alpha, \beta\}, f = some[pot\ P_2]\ pot\ P_1)$. On a : $1 \approx_E 4$, $1 \models f$, $4 \models f$, et cependant on n'a pas : $[Expl(1 \models f)] \leftrightarrow_C [Expl(4 \models f)]$. En effet, la séquence 1.2 est une séquence explicative de $1 \models f$: l'état 1 satisfait $pot\ P_2 \wedge pot\ P_1$, et l'état 2 satisfait $\neg pot\ P_2 \wedge pot\ P_1$. Par contre, il n'existe pas de séquence explicative de $4 \models f$ le long de laquelle on observe l'action α : la seule séquence explicative de cette assertion est la séquence constituée de la transition muette (4, 5). On en déduit que si x est une explication de $1 \models f$ construite à partir de la séquence explicative 1.2, il n'existe pas d'explication de $5 \models f$ dont la forme réduite soit équivalente à $[x]$.
Par contre, avec la définition initiale des transitions muettes, la transition (4, 5) est étiquetée par λ, elle est donc visible (figure 3 28) : $pot\ P_2$ est une sous-formule de f satisfaite par l'état 4, mais pas par l'état 5. Le second modèle dans la figure 3.27 n'est donc pas un modèle observé selon le critère $(\{\alpha, \beta\}, f)$.

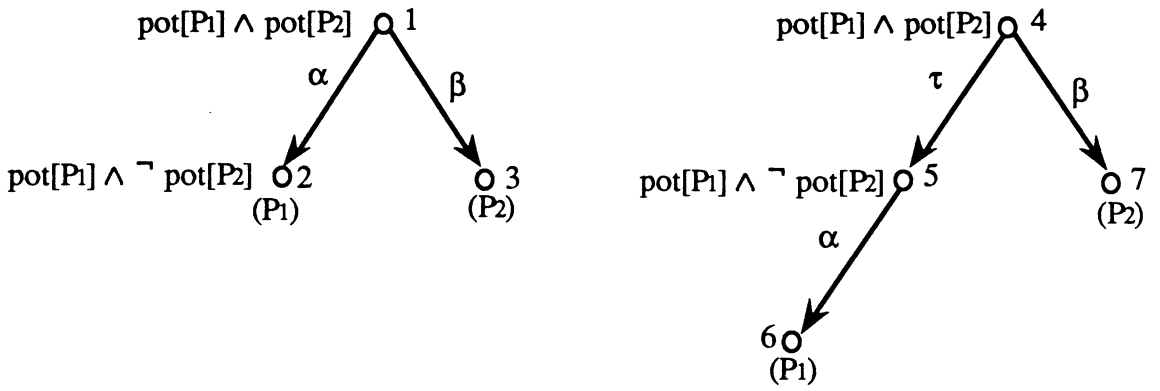


figure 3.27

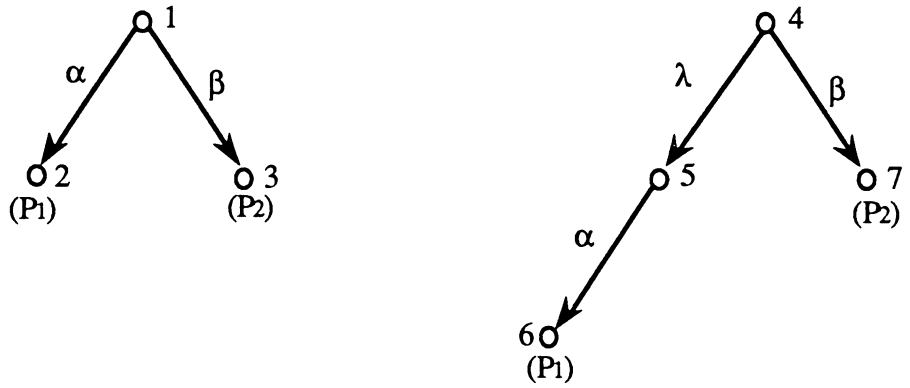


figure 3.28

Une définition plus large des transitions muettes, qui conduirait à la définition d'une relation d'équivalence plus faible que l'équivalence explicationnelle, mais satisfaisant aux mêmes exigences quant aux explications réduites, semble être un problème délicat.

Remarquons enfin que pour avoir : $q \approx_E q'$ implique ($q \models f$ ssi $q' \models f$), il est nécessaire d'exiger que des états équivalents satisfassent les mêmes sous-formules de f. En limitant cette exigence à celles des sous-formules de f qui sont des prédicats de base (c'est-à-dire en replaçant la première contrainte de la définition de la relation d'équivalence par : $\pi^0_f(q) = \pi^0_f(q')$), l'équivalence explicationnelle ne préserve pas la satisfaction de f dans des modèles équivalents (la relation $\leftrightarrow^\tau$ étant celle donnée ci-dessus).
En effet, soit $C = (\{\alpha, \beta\}, f = pot[pot\ P_2]\ P_1)$; pour cette version de l'équivalence explicationnelle, les modèles représentés par la figure 3.29 seraient équivalents, et cependant : $1 \models f$, et $4 \not\models f$.

En conclusion, la relation d'équivalence proposée semble difficile à affaiblir. La seule modification possible est celle de la définition des prédicats de convergence et de divergence.

Nous appellerons désormais équivalence explicationnelle l'équivalence définie en fonction de ces prédicats modifiés.

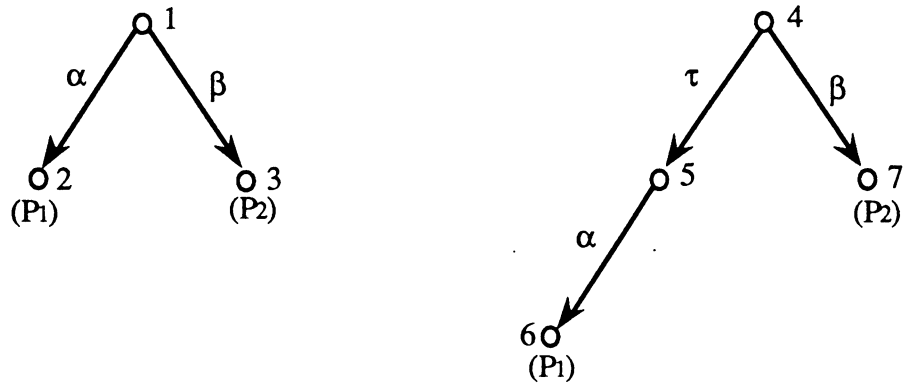


figure 3.29

3.3.4. Forme normale

Définition. Etant donné un critère d'observation C et un modèle observé M_C selon ce critère, soit $[M_C]$ le modèle obtenu à partir de M_C par application successive des transformations (1), (2) et (3) décrites ci-dessous, puis du calcul du quotient du modèle obtenu par la bisimulation forte.

Le modèle quotient est défini de la manière suivante :

Définition. Si $M_C = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$, le modèle quotient par la bisimulation forte du modèle M_C est le modèle $([Q], [\rightsquigarrow], [q], V_\lambda \cup \{\tau\}, \mathcal{P}, [\Pi])$ où :

$[Q]$ est l'ensemble des classes d'équivalence de Q pour la bisimulation forte,

$[q]$ est la classe d'équivalence de q.

$\forall \alpha \in V_\lambda \cup \{\tau\}$, $[q_1]\ [\rightsquigarrow]^\alpha\ [q_2]$ si et seulement si $q_1 \rightsquigarrow^\alpha q_2$

pour tout prédicat P, $P \in [\Pi]([p])$ si et seulement si $P \in \Pi(p)$.

Nous décrivons les transformations (1) (2) et (3) puis nous montrons que $[M_C]$ est une forme normale pour l'équivalence explicationnelle.

Description des transformations.

La méthode est inspirée de celle décrite dans [Fe88] pour l'équivalence observationnelle et dans [Ro88] pour l'équivalence de sûreté. Elle fait intervenir successivement trois transformations sur le modèle : chacune d'elles transforme le modèle M en un modèle noté M'. Les transformations (1) et (2) ont pour but de diminuer le nombre de transitions muettes

du modèle afin de réduire le coût de la τ-fermeture (transformation (3)). Chacune de ces transformations n'est pas nécessairement une fonction, mais le modèle obtenu après les trois transformations est unique à une bisimulation forte près (voir propriété 3.9).

(1) **Elimination des chaînes d'actions muettes.**
Cette transformation élimine les transitions muettes issues des états ayant un seul successeur et qui ne sont pas des boucles τ. Elle est définie formellement par application exhaustive de la règle :

Si $M = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$, $q' \rightsquigarrow^\tau p, p \neq q'$, $Card(succ(q')) = 1$
alors $M' = (Q', \rightsquigarrow', q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$ où
$Q' = Q \setminus \{p\}$
$\forall \alpha \in V_\lambda \cup \{\tau\}$,

$$\rightsquigarrow'^{\alpha} = \rightsquigarrow^{\alpha} \setminus \{(q', p)\} \setminus \{(p, q'') / (p, q'') \in \rightsquigarrow^{\alpha}\} \setminus \{(q'', p) / (q'', p) \in \rightsquigarrow^{\alpha}\}$$
$$\cup \{(q', q'') / (p, q'') \in \rightsquigarrow^{\alpha}\} \cup \{(q'', q') / (q'', p) \in \rightsquigarrow^{\alpha}\}$$

Exemple.
Si $C = (\{e\_em, r\_rec\}, f = pot\ some\ \neg after(r\_rec))$, le modèle du système *E/R* observé selon ce critère est représenté sur la figure 3.16. La figure 3.30 représente ce modèle après la transformation (1).

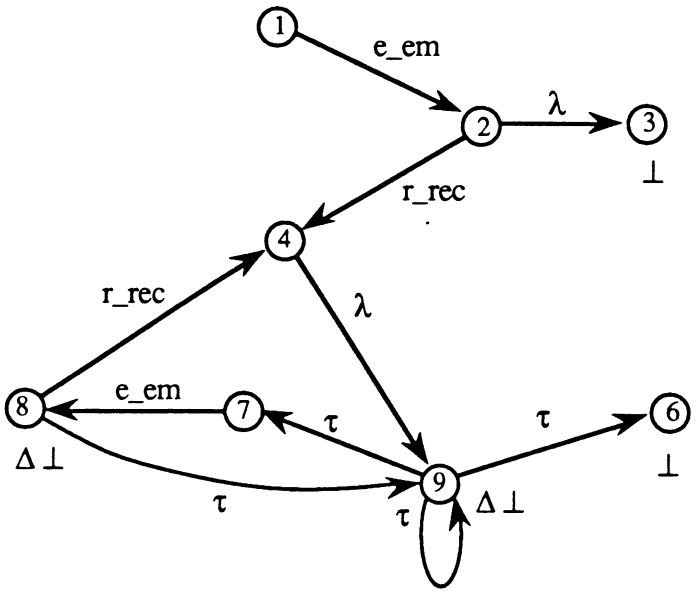


figure 3.30

(2) **Transformation des circuits de transitions muettes en boucles τ.**

Cette transformation ne conserve qu'un seul état parmi ceux d'un circuit de transitions muettes. De cet état sont issues toutes les transitions issues des autres états du circuit et une boucle τ. Les transitions adjacentes aux états éliminés ont pour nouvelle extrémité l'état conservé. Elle est définie formellement par application exhaustive de la règle :

Si $M = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$, $q_i \rightsquigarrow^\tau q_{i+1}$, $i=0\ldots n$, $q_{n+1}= q_0$, $q \notin \{q_1 \ldots q_n\}$
alors $M' = (Q', \rightsquigarrow', q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$ où

$Q' = Q \setminus \{q_1 \ldots q_n\}$

$\forall \alpha \in V$,

$$\rightsquigarrow'^{\alpha} = \rightsquigarrow^{\alpha} \setminus \bigcup_{i=1\ldots n}\{(q_i, q') / (q_i, q') \in \rightsquigarrow^{\alpha}\} \setminus \bigcup_{i=1\ldots n}\{(q'', q_i) / (q'', q_i) \in \rightsquigarrow^{\alpha}\}$$
$$\cup \{(q_0, q') / \exists i=1\ldots n\ (q_i, q') \in \rightsquigarrow^{\alpha}\} \cup \{(q'', q_0) / \exists i=1\ldots n\ (q'', q_i) \in \rightsquigarrow^{\alpha}\}$$

$$\rightsquigarrow'^{\tau} = \rightsquigarrow^{\tau} \setminus \bigcup_{i=1\ldots n}\{(q_i, q') / (q_i, q') \in \rightsquigarrow^{\tau}\} \setminus \bigcup_{i=1\ldots n}\{(q'', q_i) / (q'', q_i) \in \rightsquigarrow^{\tau}\}$$
$$\cup \{(q_0, q') / \exists i=1\ldots n\ (q_i, q') \in \rightsquigarrow^{\tau}\} \cup \{(q'', q_0) / \exists i=1\ldots n\ (q'', q_i) \in \rightsquigarrow^{\tau}\}$$
$$\cup \{(q_0, q_0)\}$$

Remarque. Si q appartient à un circuit de transitions muettes, on peut toujours, modulo une renumérotation des états, supposer $q = q_0$, et donc que l'état q soit l'état conservé au cours de cette transformation.

Exemple.

Le modèle précédent est inchangé par cette transformation : le seul circuit de transitions muettes est une boucle τ).

(3) **τ-fermeture.**

Les transformations (1) et (2) ayant réduit le nombre de transitions muettes, on calcule enfin la τ-fermeture de relation de transition. Cette transformation s'effectue en deux étapes : la première est la τ-fermeture proprement dite, la seconde rétablit les transitions muettes nécessaires à la cohérence des définitions des prédicats Δ et $\perp$.

- Première étape.

On associe aux états où le prédicat $\perp$ est vrai l'ensemble des états puits auxquels ils accèdent par une séquence non vide de transitions muettes :

$$Puits(q') = \{q'' / q' \rightsquigarrow^{\tau^*\tau} q'' \text{ et } Card(succ(q'')) = 0\}.$$

On transforme ensuite le modèle M en un modèle M'_1 :

Si $M = (Q, \rightsquigarrow, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$,
alors $M'_1 = (Q, \rightsquigarrow'_1, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$ où

$\forall \alpha \in V_\lambda \ : \rightarrow'_1{}^{\alpha} = \Rightarrow \alpha$

$\rightarrow'_1{}^{\tau} = \varnothing$

- Deuxième étape.

Le modèle M'_1 est transformé en un modèle M' par application exhaustive de la règle :

Si $M'_1 = (Q, \rightarrow'_1, q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$,

alors $M' = (Q, \rightarrow', q, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$ où :

$\forall \alpha \in V_\lambda \ : \rightarrow'^{\alpha} = \rightarrow'_1{}^{\alpha}$

$\rightarrow'^{\tau} = \{(q',q') / \Delta(q')\} \cup \{(q',q'') / \perp(q'), Card(succ(q'))>0 \text{ et } q'' \in Puits(q')\}$

Exemple.

Après application de la transformation (3), le modèle de la figure 3.30 devient celui représenté sur la figure 3.31.

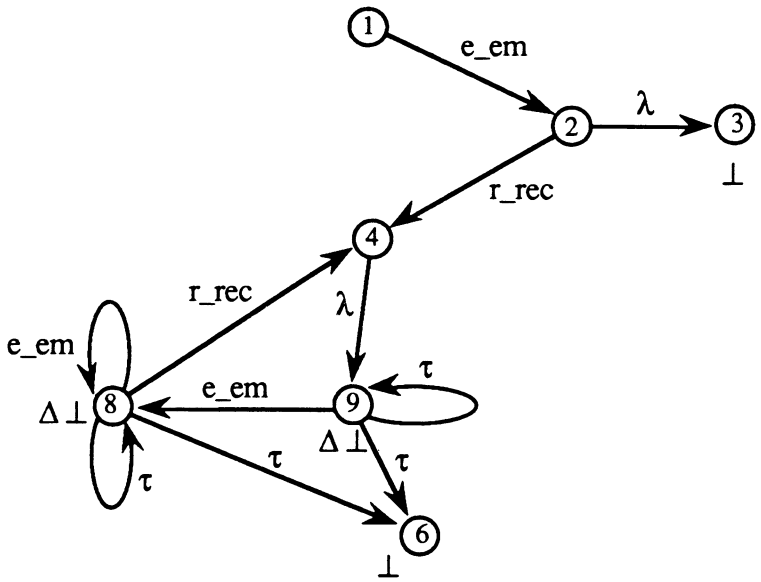


figure 3.31

Propriété 3.8. Soit M' le modèle obtenu à partir du modèle M par application des transformations (1) à (3). Alors,

- $M' \approx_E M$
- les seules transitions muettes dans le modèle M' sont :
  - des boucles τ issues de tous les états p tels que $\Delta(p)$,
  - et des transitions (p, p') telles que : $\perp(p)$, $Card(succ(p')) = 0$, $\exists\, p'' \in succ(p)$ tel que $Card(succ(p'')) \neq 0$ ou $(p, p'') \notin \rightarrow^{\tau}$.

Démonstration : immédiate par construction.

La propriété ci-dessous exprime le fait que les modèles obtenus après les transformations (1) (2) et (3), à partir de modèles équivalents est unique à une bisimulation forte près :

Propriété 3.9. Soient M'_1 et M'_2 les modèles obtenus par application des transformations (1), (2) et (3) aux modèles M_1 et M_2. Alors :

$M_1 \approx_E M_2 \Leftrightarrow M'_1 \sim M'_2$.

La bisimulation forte est la relation $\sim = \bigcap_{i \in \mathbb{N}} (\sim_i)$ de $Q \times Q$ où :

$\sim_0 = \{(q, q') / \pi_f(q) = \pi_f(q')\}$

pour $i \geq 1$, $q \sim_{i+1} q' \Leftrightarrow [\forall \alpha \in V \cup \{\tau\}$:

$\forall p : (q \rightsquigarrow^{\alpha} p) \Rightarrow (\exists p' : q' \rightsquigarrow^{\alpha} p'$ et $p \sim_i p')$

et

$\forall p' : (q' \rightsquigarrow^{\alpha} p') \Rightarrow (\exists p : q \rightsquigarrow^{\alpha} p$ et $p \sim_i p')]$

Démonstration

sens $\Rightarrow$

L'équivalence explicationnelle est préservée au cours des transformations successives : $M_1 \approx_E M'_1$ et $M_2 \approx_E M'_2$. Par conséquent, $M_1 \approx_E M_2 \Rightarrow M'_1 \approx_E M'_2$, il suffit donc de montrer : $M'_1 \approx_E M'_2 \Rightarrow M'_1 \sim M'_2$.

Soit $M'_1 = (Q, \rightsquigarrow, q_1, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$ et $M'_2 = (Q, \rightsquigarrow, q_2, V_\lambda \cup \{\tau\}, \mathcal{P}, \Pi)$.

Nous montrons que dans les modèles M'_1 et M'_2 quels que soient $p_1 \in Q$ et $p_2 \in Q$:

$p_1 \approx_E p_2 \Rightarrow p_1 \sim_0 p_2$

et pour $i \geq 1$, $(p_1 \approx_E p_2 \Rightarrow p_1 \sim_i p_2) \Rightarrow p_1 \sim_{i+1} p_2$

En effet,

- $p_1 \approx_E p_2 \Rightarrow (\pi_f(p_1) = \pi_f(p_2)) \Rightarrow p_1 \sim_0 p_2$.
- Soit $i \geq 1$, supposons que $p_1 \approx_E p_2 \Rightarrow p_1 \sim_i p_2$, et montrons que $p_1 \sim_{i+1} p_2$

S'il existe un état p'_1 tel que $p_1 \rightsquigarrow^{\alpha} p'_1$.

Si $\alpha \neq \tau$, par définition de $\approx_E$, il existe p'_2 tel que $p_2 \rightsquigarrow^{\tau^* \alpha} p'_2$ et $p'_1 \approx_E p'_2$. Or après la transformation (3), on a : $p_2 \rightsquigarrow^{\tau^* \alpha} p'_2 \Rightarrow p_2 \rightsquigarrow^{\alpha} p'_2$. De plus, par hypothèse d'induction $p'_1 \approx_E p'_2 \Rightarrow p'_1 \sim_i p'_2$. On a donc :

$$p_1 \rightsquigarrow^{\alpha} p'_1 \Rightarrow \exists\, p'_2 : p_2 \rightsquigarrow^{\alpha} p'_2 \text{ et } p'_1 \sim_i p'_2$$

Si $\alpha = \tau$, d'après la propriété 3.8, deux cas sont possibles :

si $\perp(p_1)$ est vrai, puisque $p_1 \approx_E p_2$, $\perp(p_2)$ est vrai. Or d'après cette propriété, p_1 est tel qu'au moins un de ses successeurs p''_1 n'est pas un puits accédé par une transition muette : on a $p''_1 = p_1$ et $p_1 \rightsquigarrow^{\tau} p_1$ ou pour $\alpha \neq \tau$, $p_1 \rightsquigarrow^{\alpha} p''_1$; dans le premier cas, on a donc $\Delta(p_1)$, donc $\Delta(p_2)$, par conséquent on a $p_2 \rightsquigarrow^{\tau} p_2$; dans le second cas, il existe un état p''_2 tel

que $p_2 \overset{\alpha}{\rightarrow\!\!\!\!\!\rightarrow} p''_2$. Par conséquent, p_2 est tel que $\perp(p_2)$ est vrai et au moins un des successeurs de p_2 n'est pas un puis accédé par une transition muette : donc il existe un état puits p'_2 tel que $p_2 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p'_2$. Or $(p_1 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p'_1 \Rightarrow \pi_f(p_1) = \pi_f(p'_1))$ et $(p_2 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p'_2 \Rightarrow \pi_f(p_2) = \pi_f(p'_2))$, par conséquent, $\pi_f(p'_1) = \pi_f(p'_2)$, ce qui implique, puisque p'_1 et p'_2 sont des états puits : $p'_1 \approx_E p'_2$; par hypothèse de récurrence $p'_1 \approx_E p'_2 \Rightarrow p'_1 \sim^i p'_2$, on a donc :

$$p_1 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p'_1 \models puits \Rightarrow \exists\, p'_2 : p_2 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p'_2 \text{ et } p'_1 \sim^i p'_2.$$

si $\Delta(p_1)$ est vrai, puisque $p_1 \approx_E p_2$, $\Delta(p_2)$ est vrai, et donc, d'après les propriétés des transitions muettes de M'_2, on a $p_2 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p_2$, donc :

$$p_1 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p'_1 \text{ et } p_1 = p'_1 \Rightarrow \exists\, p'_2 : p_2 \overset{\tau}{\rightarrow\!\!\!\!\!\rightarrow} p'_2 \text{ et } p'_1 \sim_i p'_2$$

On en déduit :

$$\forall\, \alpha \in V_\lambda \cup \{\tau\},\ \forall p'_1 : p_1 \overset{\alpha}{\rightarrow\!\!\!\!\!\rightarrow} p'_1 \Rightarrow (\exists p'_2 \,/\, p_2 \overset{\alpha}{\rightarrow\!\!\!\!\!\rightarrow} p'_2 \text{ et } p'_1 \sim_i p'_2)$$

et par symétrie,

$$\forall\, \alpha \in V_\lambda \cup \{\tau\},\ \forall\, p'_2 : p_2 \overset{\alpha}{\rightarrow\!\!\!\!\!\rightarrow} p'_2 \Rightarrow (\exists p'_1 \,/\, p_1 \overset{\alpha}{\rightarrow\!\!\!\!\!\rightarrow} p'_1 \text{ et } p'_1 \sim_i p'_1)$$

C'est-à-dire que $(p_1 \approx_E p_2 \Rightarrow p_1 \sim_i p_2) \Rightarrow p_1 \sim_{i+1} p_2$.

Donc, $M'_1 \approx_E M'_2 \Rightarrow \forall i, (q_1, q_2) \in \sim_i$, donc : $M'_1 \approx_E M'_2 \Rightarrow (q_1, q_2) \in \sim$,

ou encore : $M'_1 \approx_E M'_2 \Rightarrow M_1 \sim M_2$.

sens $\Leftarrow$

$(M_1 \approx_E M'_1$ et $M_2 \approx_E M'_2) \Rightarrow (M'_1 \approx_E M'_2 \Rightarrow M_1 \approx_E M_2)$. Par conséquent, il suffit de montrer : $M'_1 \sim M'_2 \Rightarrow M'_1 \approx_E M'_2$.

Or, puisque $\sim$ est une équivalence plus forte que $\approx_f$ on a : $M'_1 \sim M'_2 \Rightarrow M'_1 \approx_E M'_2$, ce qui a pour conséquence :

$$M'_1 \sim M'_2 \Rightarrow M'_1 \approx_E M'_2.$$

$\therefore$

On en déduit la propriété suivante :

Propriété 3.10. Une forme normale du modèle M_C pour l'équivalence explicationnelle est $[M_C]$.

Exemple.

Le modèle de la figure 3.31 est sous forme normale.

3.3.5. Comparaison entre les explications réduites et les explications dans la forme normale.

Dans les explications réduites, l'existence de séquences d'exécution muettes maximales est traduite par des termes de la forme $\perp(q)$ ou $\Delta(q)$. Dans leur forme non réduite, ces séquences sont présentes de manière explicite dans l'explication. Aussi comparerons nous les explications réduites dans le modèle M et les explications dans la forme normale $[M]$ sous forme simplifiée. Rappelons que les explications simplifiées sont obtenues après traduction de la présence de séquences muettes maximales en termes de prédicats de convergence ou de divergence (paragraphe 3.1.3.2). Remarquons également que dans la forme normale, le passage à la forme simplifiée d'une explication n'entraîne qu'une transformation mineure : si $\Delta(q)$, la seule séquence muette infinie issue de q est $(q)^\omega$, si $\perp(q)$, la seule séquence muette finie maximale issue de q est réduite à q ou à une seule transition.

Notons $Expl^0(M, q \models f)$ et $Expl^0([M], [q] \models f)$ les ensembles d'explications dans les modèles M et $[M]$ obtenus par application de règles minimales (voir chapitre 2), appelés ensembles d'explications minimales.
Notons $\subset_{\leftrightarrow}$ la relation d'inclusion au sens de l'équivalence $\leftrightarrow_C$:

$X \subset_{\leftrightarrow} X'$ si et seulement si $\exists\, X'' \subset X'$ tel que $X \leftrightarrow_C X''$.

Alors on a :

$[Expl(M, q \models f)] \subset_{\leftrightarrow} Expl_C([M], [q] \models f)$

et

$Expl^0{}_C([M], [q] \models f) \subset_{\leftrightarrow} [Expl^0(M, q \models f)]$

Justifions ces relations :

• Toutes les explications réduites d'un modèle sont des explications simplifiées dans le modèle sous forme normale.
En effet : $[Expl(M, q \models f)] \leftrightarrow_C [Expl([M], [q] \models f)]$. Par conséquent, il suffit de vérifier que toutes les explications réduites dans le modèle sous forme normale sont des explications simplifiées dans ce modèle. Soit $x \in Expl_C([M], [q] \models f))$ telle que $x_C \neq [x]$. Les transformations permettant de passer d'une explication simplifiée à sa forme réduite consistent à éliminer des transitions muettes dans les séquences d'explications. Soit y_C une telle séquence. Les transitions muettes dans le modèle sous forme normale étant des boucles τ et des transitions accédant à un puits, la séquence explicative s des racines des explications de y n'est pas minimale. Par conséquent, il existe une séquence d'explications y', telle que la séquence d'exécution correspondante est une séquence explicative minimale minorant s, telle que $y'_C = [y'] = [y]$. On en déduit qu'il existe une explication x' telle que $x'_C = [x'] = [x]$.

• Toutes les explications simplifiées calculées dans le modèle sous forme normale ne sont pas nécessairement des explications réduites du modèle initial.

Exemple.
Soit $V = \{\alpha\}$, la formule $f = some\ P_1 \wedge pot\ P_2$. Alors, si (V, f) est le critère d'observation, le modèle représenté sur la figure 3.32 est sous forme normale.

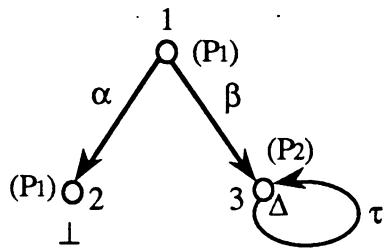


figure 3.32

Cependant, dans ce modèle, les explications simplifiées de la forme :

$some\ P_1 : (1 \models P_1 . (2 \models P_1\ \&\ \bot(2)))$

&

$pot\ P_2 : (1 \models vrai.\ 3 \models vrai....3 \models vrai.3 \models P_2))$

ne sont pas des explications réduites.

Pour avoir cette propriété, il aurait fallu que l'équivalence explicationnelle soit la plus faible possible, ce qui n'est pas le cas. Remarquons que dans l'exemple ci-dessus, l'explication n'est pas une explication minimale, puisque 1.3.3...3 n'est pas une séquence explicative minimale de $1 \models pot\ P_2$. Pour les explications minimales, nous avons par contre la propriété suivante :

• Toutes les explications minimales simplifiées dans la forme normale du modèle sont équivalentes à la forme réduite d'une explication minimale dans tout modèle équivalent.
Soit M le modèle, M' le modèle obtenu par application des transformations (1), (2) et (3), et $[M]$ le quotient de M' par la bisimulation forte.
On a : $[Expl^0(M, q \models f)] \leftrightarrow_C [Expl^0(M', q \models f)]$. En effet, on vérifie aisément que chacune des transformations (1), (2) et (3) laisse inchangé l'ensemble des explications minimales réduites. D'autre part, à toute séquence explicative minimale s de $[q] \models f$ dans $[M]$ correspond une séquence explicative minimale s' de $q \models f$ dans M', telle que :

$longueur(s) = longueur(s')$,

$\forall i \leq longueur(s), s(i) \approx_E s'(i)$

$\forall i$ <*longueur*(s), les transitions ($s(i)$, $s(i+1)$) et ($s'(i)$, $s'(i+1)$) sont étiquetées par le même nom d'action.

On en déduit que toute explication minimale réduite de [M] est équivalente à une explication minimale réduite de M'.

Exemple. Considérons le modèle du système *E/R* observé selon le critère ({*e_em*, *r_rec*}, f = *pot some* ¬*after*(*r_rec*)) et sa forme normale (figure 3.33).

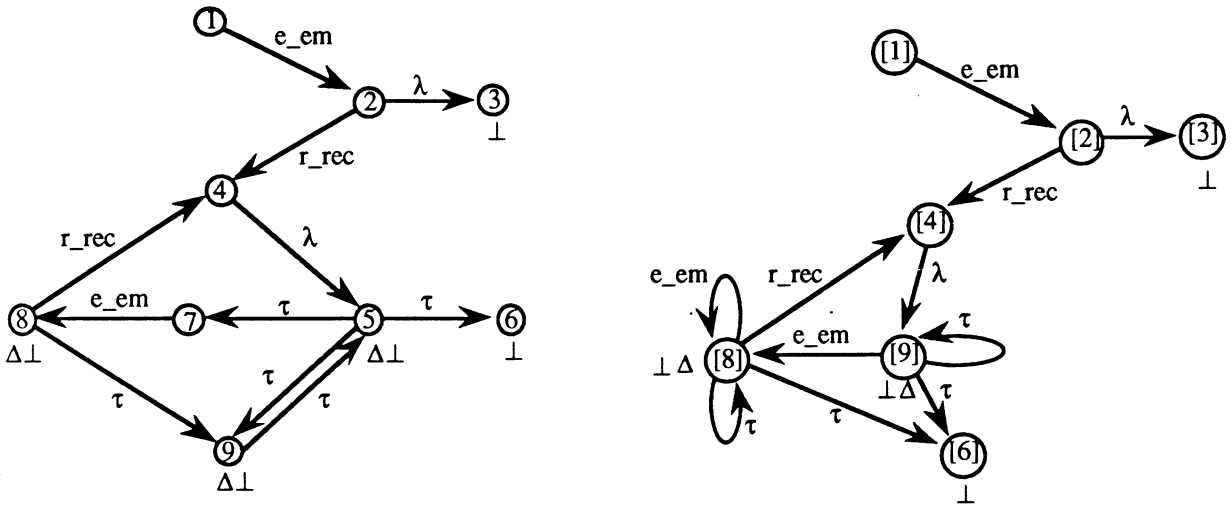


figure 3.33

Dans la forme normale, les explications minimales simplifiées de [4]⊨f sont les suivantes :

x_1 = *pot some* ¬*after*(*r_rec*)) :

([4] ⊨ *vrai. some* ¬*after*(*r_rec*) : (([9] ⊨¬*after*(*r_rec*) & ⊥([9]))

x_2 = *pot some* ¬*after*(*r_rec*)) :

([4] ⊨ *vrai. some* ¬*after*(*r_rec*) : (([9] ⊨¬*after*(*r_rec*) & Δ([9]))

x_3 = *pot some* ¬*after*(*r_rec*)) :

([4] ⊨ *vrai. some* ¬*after*(*r_rec*) : ([9] ⊨¬*after*(*r_rec*).([8] ⊨¬*after*(*r_rec*))$^{\omega}$)

Dans le modèle initial, les formes réduites des explications minimales de 4⊨f sont les suivantes :

y_1 = *pot some* ¬*after*(*r_rec*)) :

(4 ⊨ *vrai. some* ¬*after*(*r_rec*) : ((5 ⊨¬*after*(*r_rec*) & ⊥(5))

y_2 = *pot some* ¬*after*(*r_rec*)) :

(4 ⊨ *vrai. some* ¬*after*(*r_rec*) : ((5 ⊨¬*after*(*r_rec*) & Δ(5))

y_3 = *pot some* ¬*after*(*r_rec*)) :

(4 ⊨ *vrai. some* ¬*after*(*r_rec*) : (9⊨¬*after*(*r_rec*).(8 ⊨¬*after*(*r_rec*))$^{\omega}$)

et on a pour i = 1, 2, 3 : $x_i \leftrightarrow_C y_i$.

• La forme réduite de certaines explications minimales n'est pas équivalente à une explication minimale dans la forme normale du modèle :

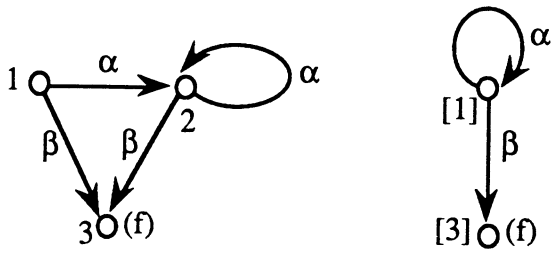


figure 3.34

La séquence 1.2.3 (figure 3.34) est une séquence explicative minimale de $1 \models pot\, f$, il lui correspond une explication $x = pot\, f : (1 \models vrai.\ 2 \models vrai.\ 3 \models f)$ qui est, puisque les transitions (1, 2) et (2, 3) sont visibles, sa propre forme réduite.
Par contre, dans le modèle $[M]$, la seule explication minimale de $[1] \models pot\, f$ est $y = pot\, f : ([1] \models vrai.\ [3] \models f)$, et on n'a pas $[x] \leftrightarrow_C y$.

Dans cet exemple, les modèles sont équivalents à une bisimulation forte près : ce n'est donc pas un défaut de l'équivalence explicationnelle, mais provient de la définition des séquences d'accès à un but (voir paragraphe 2.2.3.2.).

Conclusion.

En considérant les réductions des explications induites par un critère d'observation (V, f), nous avons cherché de quelle manière simplifier le modèle préalablement au calcul des explications, tout en restant compatible avec les explications réduites. Nous avons défini une relation d'équivalence qui préserve les explications réduites de f, et nous nous sommes attachés à ce que cette relation d'équivalence soit la plus faible possible : en particulier, la relation d'équivalence dépend de f, par conséquent, elle est plus faible qu'une relation d'équivalence qui préserverait les explications réduites de toutes les formules.
Cependant, l'équivalence explicationnelle n'est pas la plus faible relation d'équivalence répondant au problème : dans un modèle sous forme normale, certaines explications ne sont pas entièrement réduites. Des tentatives pour affaiblir cette relation d'équivalence tout en préservant ses propriétés échouent : pour y parvenir, il faudrait en effet connaître les explications a priori.

A l'heure actuelle, l'algorithme de calcul de la forme normale d'un modèle n'a pas encore été implanté. Des réductions partielles sont cependant effectuées, qui sont décrites au paragraphe 4.3.

Chapitre 4

Le système Cléo

Ce chapitre est organisé de la manière suivante. Le paragraphe 4.1. est une description générale de Cléo. Cléo est fondée sur l'exploitation d'automates particuliers appelés automates d'explication qui sont introduits dans le paragraphe 4.2. Nous présentons ensuite une méthode de réduction de ces automates (paragraphe 4.3.) qui préserve la trace des séquences acceptées. Les automates d'explication sont utilisés pour calculer soit l'ensemble des séquences qu'ils acceptent (paragraphe 4.5), soit une séquence particulière parmi celles-ci (paragraphe 4.4).

4.1. Les modes de fonctionnement de Cléo

4.1.1. Description générale.

Etant donné un programme dont on a pu constater à l'aide du système XESAR qu'il ne vérifiait pas ses spécifications, le problème demeure d'en comprendre les raisons et la manière d'y remédier. Cléo, système d'aide à la mise au point, a été conçu dans ce but : fournir à l'utilisateur des informations utiles à la formulation d'un diagnostic.
Ces informations sont extraites de la représentation du programme par un graphe d'états. Leur nature et la manière de les représenter doivent être choisies en fonction de leur valeur explicative pour l'utilisateur. Celui-ci est donc invité, avant toute chose, à définir un sous-ensemble des actions du programme, les actions visibles, qui seront les seules à apparaître explicitement dans les explications que le système proposera. Cet ensemble sera noté V par la suite.

D'autre part, l'utilisateur doit préciser, parmi les spécifications, celles auxquelles il s'intéresse : la non-satisfaction de celles-ci est exprimée par une formule de logique temporelle f. Bien sûr, les choix de l'utilisateur pourront se préciser au cours de son étude : des actions visibles trop nombreuses, des spécifications trop complètes conduiront à des explications trop volumineuses pour être utiles, des actions visibles trop rares, des spécifications trop partielles conduiront à des explications trop sommaires.
L'utilisateur dispose alors de deux modes de fonctionnement, dans lesquels lui seront proposées des informations de nature différente correspondant à des méthodes de travail différentes et complémentaires. Nous parlerons de mode "pas à pas" et de mode "trace globale".

Dans le mode "pas à pas", on propose à l'utilisateur de choisir un état particulier q parmi ceux qui satisfont la formule f. Il peut alors suivre "pas à pas" l'explication de l'assertion $q \models f$, c'est-à-dire explorer de proche en proche l'ensemble des règles de réécriture utilisées pour la dérivation de $q \models f$: Cléo propose donc à l'utilisateur de se convaincre de la validité de $q \models f$ (donc de s'expliquer pourquoi $q \models f$), soit par exploration exhaustive de toutes les branches de la dérivation, soit en choisissant ce qui lui paraît le plus significatif, parmi les sous-formules de f et l'ensemble des états du programme qui les satisfont.

Dans le mode "trace globale", Cléo propose à l'utilisateur des informations, non pas sur une assertion $q \models f$ particulière, mais sur l'ensemble des assertions $q \models f$, pour tous les états du modèle et une formule temporelle f de la forme $pot[f_1]f_2$ ou $some[f_1]f_2$. L'information globale considérée est l'ensemble des séquences d'exécution qui expliquent pourquoi $q \models f$ pour tout état q satisfaisant f. Parmi ces séquences, les plus caractéristiques ont été présentées au chapitre 2, sous le terme "séquences explicatives". Toutefois, il est nécessaire de décrire à l'utilisateur ces séquences sous une forme utilisable : on a donc choisi de présenter les traces de ces séquences (c'est-à-dire de les considérer comme des séquences d'actions visibles). Enfin, il est nécessaire de compléter l'ensemble obtenu de manière à pouvoir l'exprimer par une expression ω-régulière (bien sûr, les séquences qui complètent l'ensemble sont elles aussi "explicatives" au sens large du terme). Dans ce mode global, l'utilisateur dispose donc d'une représentation des séquences d'actions qui expliquent pourquoi $q \models f$, pour tous les états du modèle. Le choix du modèle, c'est-à-dire de l'ensemble des états du programme à considérer, se précisera en général au cours de l'utilisation.

4.1.2. Mode "pas à pas".

Il s'agit dans ce mode de fonctionnement d'expliquer comment une assertion $q \models f$ peut être prouvée. Une telle preuve est effectuée par le système par application de règles de réécriture minimales (voir paragraphe 2.3.3.3.). Ce sont les différentes étapes de cette preuve particulière que l'utilisateur peut examiner. Chacune des étapes correspond à la dérivation d'une assertion de la forme $q_c \models f_c$ où f_c est la formule courante. Pour passer d'une étape à une autre, l'utilisateur dispose de trois primitives permettant de modifier la formule courante : sélectionner la première sous-formule de f_c, sélectionner la seconde, reconstruire la formule dont f_c est sous-formule (dans f). La modification de l'état courant q_c se fait ensuite, en le choisissant dans l'ensemble des états satisfaisant la nouvelle formule courante, ensemble qui est énuméré par le système.
Lorsque la dérivation de $q_c \models f_c$ est exprimée sous la forme d'une séquence d'assertions, celle-ci correspond à une séquence d'exécution. Pour en permettre clairement l'explication, on y précise en outre quelle séquence d'actions (visibles) correspondent à cette exécution.
Sachant quelle règle minimale a été appliquée pour prouver q_c, l'utilisateur choisit en partie droite de cette règle quelle est l'assertion dont l'explication l'intéresse en premier chef. Il sélectionne alors la nouvelle formule courante et le nouvel état courant, et obtient la règle de réécriture correspondante. Le système permet ainsi de consulter les éléments de la preuve en tous sens, mais de proche en proche, en parcourant la structure arborescente de la formule f.

La mise en œuvre de ce "pas à pas", dans le cas d'une formule temporelle (*pot* ou *some*) est fondée simplement sur le calcul d'une séquence explicative de $q \models f$. Nous verrons que ceci, la recherche d'un élément de $S^0(q \models f)$, revient à chercher dans le graphe du programme un plus court chemin d'un ensemble d'états Q_1 (ici réduit à q) à un ensemble d'états Q_2, et tel que tous les états rencontrés appartiennent à un ensemble Q_3. Ces ensembles Q_2 et Q_3 sont définis à l'aide de l'automate d'explication associé à la formule f, automate dont la définition et l'utilisation sont l'objet des paragraphes suivants de ce chapitre.

4.1.3. Mode "trace globale".

Dans ce mode de fonctionnement, la formule f concernée est une formule temporelle, de type *pot* ou *some*. L'explication d'une assertion $q \models f$ peut alors s'exprimer en fonction de l'ensemble $S(q \models f)$ des séquences explicatives de $q \models f$. L'information qui, dans ces séquences intéresse l'utilisateur, peut être exprimée par ce que nous appellerons leur trace : la trace d'une séquence d'exécution est la séquence des actions visibles qui en étiquettent les transitions.

Définition. Soit $\rightarrow$ une relation de transition, étiquetée par V_λ, et s une séquence telle que deux éléments successifs sont en relation. La trace de s sera notée *trace*(s).
Soit $s = q_1.q_2\ldots$
si $\forall i \geq 1$, $q_i \rightarrow^\lambda q_{i+1}$, alors $trace(s) = \varepsilon$,
sinon, soit j le plus petit indice tel que $q_j \rightarrow^\alpha q_{j+1}$ et $\alpha \neq \lambda$, et soit $r(s, j)$ la suite $q_{j+1}.q_{j+2}\ldots$, alors, $trace(s) = \alpha.trace(r(s, j))$.

L'ensemble des traces des séquences explicatives des assertions $q \models f$, pour tous les états du modèle satisfaisant f, est une information de type global sur le fonctionnement du programme (ou d'une partie du programme), susceptible de faire apparaître une explication globale du mauvais fonctionnement de ce programme. Toutefois, il faut pour cela exprimer ces traces sous une forme exploitable.

• Lorsque $S(q \models f)$ est composé seulement de séquences finies, les traces forment un langage régulier. Nous le représenterons par une expression régulière notée $T(q \models f)$. Un automate d'états finis reconnaissant ce langage peut être extrait du graphe : les traces sont celles de tous les chemins dans l'automate (qui est fini) qui joignent l'état q à un ensemble d'états, qu'il suffit de définir comme états finals de l'automate.

• Par contre, lorsque $S(q \models f)$ contient des séquences infinies, (c'est-à-dire seulement si $f = some[f_1]f_2$, et si le graphe contient un circuit d'états satisfaisant à la fois f_1 et f_2), l'ensemble des traces n'est pas en général ω-régulier, comme on l'a vu au paragraphe 2.2.4.3 (propriété 2.10). Cependant, on peut compléter cet ensemble de manière à ce qu'il soit ω-régulier, sans altérer sa nature "explicative". Les traces infinies de $S(q \models f)$ ont pour suffixes des séquences cycliques, et nous enrichissons l'ensemble en considérant des suffixes non cycliques. Nous noterons $T(q \models f)$ cet ensemble ω-régulier, qui contient donc les traces des éléments de $S(q \models f)$, ainsi que les traces de ces séquences de suffixe non cyclique. Ces traces complémentaires expliquent, elles aussi, la satisfaisabilité de $some[f_1]f_2$: ce sont des traces de séquences d'états qui satisfont f_1 et f_2.

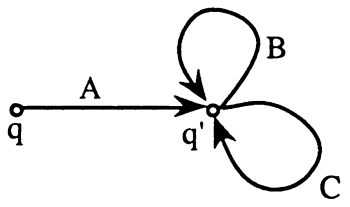


figure 4.1

Par exemple (figure 4.1) si l'ensemble des traces de $S(q \models f)$ est l'ensemble :
$A.(B + C)^*.\{(w)^\omega / w \in (B + C)^*\}$,

$T(q \models f)$ sera l'ensemble $A.(B + C)^{\omega}$.

Définition. Si l'ensemble des traces des éléments de $S(q \models f)$ est de la forme : $\bigcup_j T_j \cup \bigcup_i [A_i.\{(w)^{\omega} / w \in (B_{i,1} +\ldots+ B_{i,m_i})^*\}]$ où les T_j, les A_i, et les $B_{i,k}$ sont des expressions régulières, alors $T(q \models f) = \bigcup_j T_j \cup \bigcup_i [A_i .(B_{i,1} +\ldots+ B_{i,m_i})^{\omega}]$.

Ainsi, nous pouvons représenter les traces infinies par une expression ω-régulière, ce qui nous assure une unité entre les types de représentation. Cette expression ω-régulière, d'autre part, donne une image fidèle et claire de la structure du graphe. Enfin, considérons les traces infinies de $S^0(q \models some[f_1]f_2)$, séquences explicatives minimales : elles forment un ensemble fini de séquences cycliques, qui est donc ω-régulier. Et le calcul d'une expression ω-régulière de cet ensemble par analyse du graphe du programme, est très proche de celui que nous avons à effectuer pour l'ensemble $T(q \models f)$; notre choix s'inscrit donc dans la perspective de l'évolution de Cléo vers l'utilisation des séquences explicatives minimales.

La mise en œuvre du mode "trace globale" consiste à calculer une expression ω-régulière pour $\bigcup_{q \models f}(T(q \models f))$, en construisant un automate de Büchi, l'automate d'explication de f. Cet automate est construit en définissant un sous-graphe du programme, des ensembles d'états initiaux, finaux, et de répétition, en fonction de l'évaluation des sous-formules de f : ce sera l'objet du paragraphe 4.2. Dans les paragraphes 4.3 et 4.5, nous verrons comment réduire cet automate, et comment calculer une expression ω-régulière de l'ensemble des séquences qu'il accepte.
En fait, le calcul de cette expression est effectué en considérant l'ensemble des chemins (dans le graphe du programme) dont l'origine est dans un ensemble Q_1, l'extrémité dans un ensemble Q_2, et ne visitant que des états d'un ensemble Q_3 : nous retrouvons le même type de problème que pour le mode de fonctionnement pas à pas.

4.1.4. Autres modes de fonctionnement possibles.

Comme nous l'avons évoqué dans les deux paragraphes précédents, le système Cléo est articulé autour de la résolution de deux problèmes classiques :

- recherche de l'ensemble des chemins de Q_1 à Q_2, passant par Q_3.
- recherche d'un plus court chemin dans cet ensemble.

Ces problèmes sont résolus dans un graphe préalablement réduit, essentiellement pour diminuer le coût des calculs nécessaires.

D'autres modes de fonctionnement, ou plutôt un éventail plus large d'interrogations plus précises, peuvent facilement être mis en œuvre à partir de ces deux fonctionnalités et de l'évaluateur de formules de XESAR. En effet, Cléo est un prototype dans lequel les possibilités de dialogue avec l'utilisateur n'ont pas été développées. Donnons un bref aperçu des questions auxquelles Cléo permettrait de répondre, moyennant la définition d'une interface avec l'utilisateur :

Considérons une erreur représentée par une formule f,

- le fait qu'il n'y ait jamais d'erreur est représenté par la formule $al\ \neg f$,
- le fait qu'il existe une erreur, par la formule $pot\ f$,
- le fait que l'erreur soit inévitable par la formule $inev\ f$,
- le fait qu'il n'y ait jamais d'erreur sous la contrainte g par la formule $al\ [g]\neg f$.

On peut donc répondre aux interrogations suivantes :

- Y-a-t-il l'erreur f ?

  évaluer $pot\ f$

- Sous la contrainte g, l'erreur f persiste-t-elle ?

  évaluer $pot\ [g]f$

- Est-il possible d'éviter l'erreur (lorsqu'elle existe) ?

  évaluer $pot\ f \wedge some\ \neg f$

Sachant que la formule $pre(f)$ caractérise tous les états dont un successeur satisfait la formule f :

- Jusqu'à quel point l'erreur est-elle évitable ?

  calculer une séquence minimale (ou toutes les séquences) depuis l'état initial (ou depuis un état qui satisfait $pot\ f$) jusqu'à un état vérifiant la formule $some\ \neg f \wedge pre(inev\ f)$

- Comment l'éviter ?

  calculer une séquence minimale (ou toutes les séquences) depuis un état vérifiant $some\ \neg f \wedge pre(inev\ f)$ dont tous les états vérifient $\neg f$

- Comment devient-elle inévitable ?

  calculer une séquence minimale (ou toutes les séquences) depuis les états vérifiant $some\ \neg f$ jusqu'à ceux vérifiant $inev\ f$.

4.2. Automates d'explication

4.2.1. Définition et propriétés

Définition. Un **automate d'explication** est une structure $A = (Q_A, \rightarrow_A, I, F, R, V_\lambda)$ où

- Q_A est un ensemble fini d'états,
- I, F, R sont des sous-ensembles de Q_A dont les éléments sont appelés respectivement états initiaux, états finaux et de répétition,
- $\rightarrow_A$ est un sous ensemble de $Q_A \times V_\lambda \times Q_A$ appelé relation de transition de A.
- Le graphe $(Q_A, \rightarrow_A)$ est tel que si une de ses composantes fortement connexes comporte un état de répétition, alors tous les états de cette composante sont des états de répétition.

Les transitions muettes d'un automate d'explication sont les éléments de la relation $\rightarrow_A^\lambda$.
L'ensemble reconnu par un automate d'explication est un ensemble de traces de séquences de transitions de cet automate (la définition des traces est donnée au paragraphe 4.1.3). Par définition, l'automate d'explication A reconnaît l'ensemble $L(A)$ composé :

- des traces des séquences de I à F,
- des traces des séquences infinies, issues d'un état de I, dont un suffixe ne contient que des états de R.

Autrement dit, $L(A)$ est le sous-ensemble de $(V)^* \cup (V)^\omega$ tel que :

- si $w \in (V)^*$, $w \in L(A)$ si et seulement si il existe une séquence d'états $q_0 \ldots q_n$ telle que :

$$q_0 \in I, q_n \in F, trace(q_0 \ldots q_n) = w,$$

- si $w \in (V)^\omega$, $w \in L(A)$ si et seulement si il existe une séquence d'états $q_0 \ldots q_n \ldots$ telle que :

$$q_0 \in I, \exists\, i \text{ tel que : } \forall j, (j \geq i \Rightarrow q_j \in R), trace(q_0 \ldots q_n \ldots) = w$$

Exemple

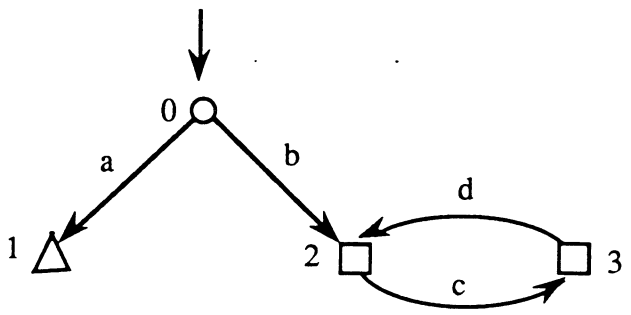


figure 4.2. L'automate d'explication A.

Soit Q_A={0, 1, 2, 3}, V = {a, b, c, d} et A = (Q_A, $\rightarrow_A$, {0}, {1}, {2,3},V_λ) l'automate représenté sur la figure 4.2. Cet automate accepte les traces des séquences de {0} à {1}, c'est-à-dire {a}, et les traces des séquences infinies issues de {0} dont un suffixe ne contient que des états de {2,3}, c'est-à-dire {$b.(c.d)^\omega$}, autrement dit, $L(A)$ = {a} $\cup$ {$b.(c.d)^\omega$}.

Comparaison des automates d'explication et des automates de Büchi

La condition d'acceptation de Büchi diffère de celle des automates d'explication pour les séquences infinies : dans les automates de Büchi, les séquences infinies sont acceptées si et seulement si elles visitent infiniment souvent l'ensemble des états de répétition. Autrement dit, la trace d'une séquence infinie w est acceptée par A , avec la condition de Büchi, si et seulement si il existe une séquence d'états $q_0 \ldots q_n \ldots$ telle que :

$q_0 \in I$, $\{i \,/\, q_i \in R\}$ est infini, $trace(\, q_0 \ldots q_n \ldots) = w$

Remarquons qu'une condition équivalente est la suivante :

$q_0 \in I$, $\exists\, r \in R$ tel que $\{i \,/\, q_i = r\}$ est infini, $trace(\, q_0 \ldots q_n \ldots) = w$

Propriété 4.1. A étant un automate d'explication, si $BL(A)$ est l'ensemble des séquences acceptées par A selon la condition de Büchi, on a : $BL(A) = L(A)$.

Démonstration.

Il est évident que $L(A) \subset BL(A)$.

Réciproquement,

- $BL(A)$ contient les mêmes séquences finies que $L(A)$.
- soit $q_0 \ldots q_n \ldots$ une séquence infinie de l'automate. Le graphe étant fini, le nombre de composantes fortement connexes est fini, donc il existe un rang à partir duquel tous les états appartiennent à la même composante fortement connexe. Si la trace de la séquence appartient à $BL(A)$, cette composante contient au moins un état de répétition, donc elle ne contient que des états de répétition (d'après la définition des automates d'explication). Donc $q_0 \ldots q_n \ldots$ ne contient que des états de répétition à partir d'un certain rang, autrement dit, sa trace appartient à $L(A)$.

$\therefore$

Corollaire. $L(A)$ est un ensemble ω-régulier.

4.2.2. ω-langage accepté par un automate d'explication

Etant donné un automate d'explication A = (Q_A, $\rightarrow_A$, I, F, R, V_λ), on donne ici une caractérisation de $L(A)$.

A tout couple (q_1, q_2) d'éléments de Q_A, on associe l'ensemble régulier des traces des séquences qui vont de l'état q_1 à l'état q_2 :

$W(q_1, q_2) = \{trace(s) / premier(s) = q_1, dernier(s) = q_2\}$

$L(A)$ peut être considéré comme l'ensemble des séquences acceptées avec la condition de Büchi, c'est-à-dire l'ensemble composé :

- des traces des séquences de I à F,
- des traces des séquences de I à un état de répétition r, suivie d'une séquence infinie issue de r et qui passe infiniment souvent par r :

$L(A) = \bigcup_{i\in I, k\in F} W(i, k) \cup \bigcup_{i\in I, r\in R} (W(i, r).(W(r, r))^{\omega})$

4.2.3. Automates d'explication des formules temporelles.

Etant donné un modèle $M = (Q, \rightarrow, q, Act_\lambda, \mathcal{P}, \Pi)$, et un ensemble V d'actions visibles définies par l'utilisateur, on associe à chaque formule temporelle $f = pot[f_1]f_2$ ou $f = some\ [f_1]f_2$ un automate d'explication $A(f) = (Q_A, \rightarrow_A, I, F, R, V_\lambda)$ dont l'ensemble d'états Q_A est une partie de Q, et dont la relation de transition est déduite de $\rightarrow$. Les ensembles Q_A, I, F, et R sont déterminés en fonction de l'évaluation des sous-formules de f.

4.2.3.1. Automate d'explication de $pot[f_1]f_2$

Pour tout q, $S(q \models pot[f_1]f_2)$ est un ensemble de séquences d'exécution finies dont tous les états intermédiaires satisfont f_1, et dont le dernier état satisfait f_2. Chacun des états de ces séquences satisfait en particulier $pot[f_1]f_2$.

L'automate d'explication de $pot[f_1]f_2$ est $A(pot[f_1]f_2) = (Q_A, \rightarrow_A, I, F, R, V_\lambda)$ où :

$Q_A = |pot[f_1]f_2|$,

si $\alpha \in V$, $\rightarrow_A{}^{\alpha} = \rightarrow^{\alpha} \cap (Q_A \times Q_A)$

$\rightarrow_A{}^{\lambda} = \{(q_1, q_2) \in \rightarrow^{\alpha} / \alpha \in (Act_\lambda \setminus V)\} \cap (Q_A \times Q_A)$.

$I = Q_A$,

$F = |f_2|$,

$R = \varnothing$.

4.2.3.2. Automate d'explication de $some\ [f_1]f_2$

Pour tout q, $S(q \models some\ [f_1]f_2)$ est un ensemble de séquences d'exécution du programme dont chaque état satisfait $some\ [f_1]f_2$.

L'automate d'explication de $some\ [f_1]f_2$ est $A(some\ [f_1]f_2) = (Q_A, \rightarrow_A, I, F, R, V_\lambda)$ où :

$Q_A = |some\ [f_1]f_2|$,

si $\alpha \in V$, $\rightarrow_A{}^{\alpha} = \rightarrow^{\alpha} \cap (Q_A \times Q_A)$

$\rightarrow_A{}^{\lambda} = \{(q_1,q_2) \in \rightarrow^{\alpha} / \alpha \in (Act_{\lambda} \setminus V) \} \cap ((Q_A \times Q_A)$.
$I = Q_A$,
$F = |\neg f_1 \wedge f_2| \cup |f_1 \wedge f_2 \wedge puits|$,
$R = |f_1 \wedge f_2|$.

Par construction, les automates d'explication des formules temporelles sont tels que si $f=pot[f_1]f_2$ ou $f=some\ [f_1]f_2$, $L(A(f)) = \cup_{q \models f} (T(q \models f))$.

En effet, tout état d'une séquence explicative de f est un état de $A(f)$, et la relation de transition de $A(f)$ est la restriction de la relation de transition du programme à ces états. Les séquences explicatives finies sont les chemins de I à F ; l'ensemble de leurs traces est donc $\cup_{i \in I, k \in F} W(i, k)$. D'autre part, pour chaque i de I, les traces des séquences explicatives de $i \models f$ sont les éléments de $\cup_{i \in I, r \in R} (W(i, r).\{(w)^{\omega} / w \in W(r, r)\})$. Par conséquent, par définition de $T(i \models f)$, l'ensemble des éléments infinis de cet ensemble est : $\cup_{i \in I, r \in R} (W(i, r).(W(r, r))^{\omega})$. On en déduit que $L(A(f)) = \cup_{q \models f} (T(q \models f))$.

Ainsi, le calcul de $\cup_{q \models f} (T(q \models f))$ consiste à calculer $L(A(f))$ pour l'automate d'explication de f.
D'autre part, pour tout état q de I, un élément de $S^0(q \models f)$ est :

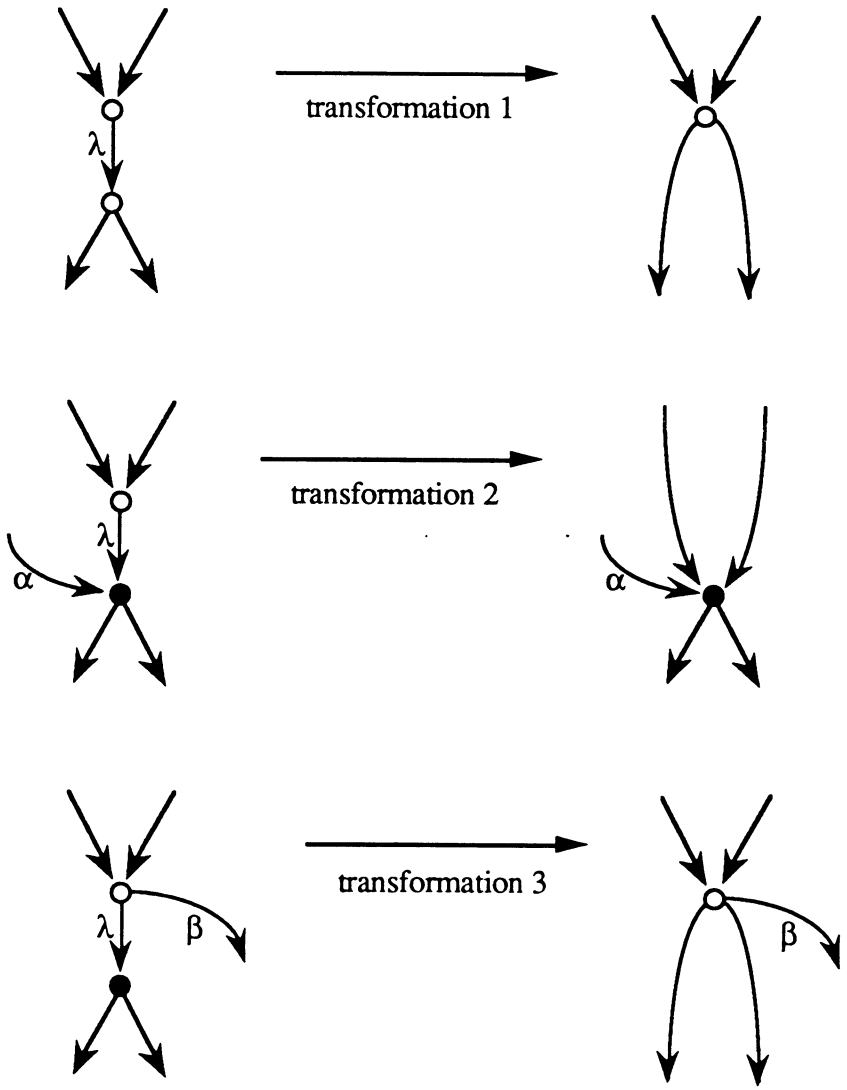
- une séquence issue de q d'accès à un état de F,
- ou une séquence de la forme $s_1.(s_2)^{\omega}$ où s_1 est une séquence issue de q d'accès à R, et s_2 une séquence issue d'un successeur de *dernier*(s_1) et d'accès à *dernier*(s_1).

4.3. Réduction d'un automate d'explication.

Avec Alain Kerbrat, nous avons défini un ensemble de transformations permettant de réduire la taille d'un automate d'explications [Ke89]. Lorsque ces transformations ont été spécifiées, leur but était uniquement de préserver l'ensemble des séquences acceptées par un automate d'explication, ce qui correspond à l'utilisation de ces automates : à chaque sous-formule de la formule courante correspond un automate d'explication déterminé de manière indépendante. Nous n'avions pas encore défini l'équivalence explicationnelle, ni a fortiori la forme normale des modèles pour cette équivalence. La correspondance entre les automates d'explication et les modèles est donnée par une relation permettant d'associer aux états de l'automate les sous-formules de f qu'il satisfait. L'adaptation aux automates des transformations permettant d'obtenir la forme normale d'un modèle pour l'équivalence explicationnelle nous aurait

permis de déduire immédiatement de l'automate de f l'automate de ses sous-formules. Les avantages offerts par cette méthode n'ont pas encore été étudiés et font partie des perspectives de développement de Cléo.

Les transformations qui ont été définies et implémentées substituent chaque fois que cela est possible une transition étiquetée par une action visible α à une suite de transitions portant la suite d'actions $\alpha.\lambda^*\lambda$. Les états devenus inaccessibles ou improductifs sont éliminés. Ainsi, les transformations 1, 2, 3, et 5 font diminuer le nombre d'états et de transitions de l'automate. La transformation 4 crée des transitions supplémentaires : ainsi, il est possible, dans les cas défavorables, que l'automate obtenu comporte plus de transitions que l'automate initial. Nous n'avons jamais rencontré ce cas dans la pratique. Les transformations sont illustrées par la figure 4.3 :



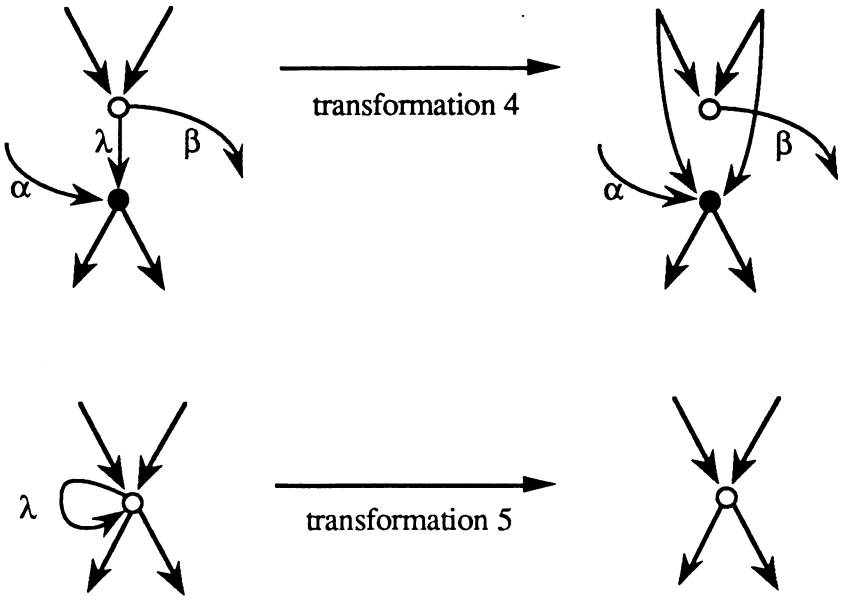


figure 4.3. Réduction d'un automate d'explication.

L'algorithme qui applique ces transformations traite successivement tous les états de l'automate, en ne modifiant pour l'état courant que les transitions incidentes à cet état. Ainsi, aucune possibilité de nouvelle transformation n'est introduite pour un état déjà traité.
La transformation 5 n'est pas appliquée si $f = some[f_1]f_2$. Même si les séquences infinies de transitions muettes n'apparaissent pas au niveau de la trace, on peut retrouver leur existence dans l'automate d'explication réduit : ainsi, les transformations préservent le prédicat de divergence associé aux états situés sur des circuits de transitions muettes. D'autre part, les états finaux de l'automate étant dans la pratique les seuls qui éventuellement vérifient des sous-formules de f différentes de celles satisfaites par les autres états de l'automate, ne sont jamais supprimés par ces transformations. L'automate obtenu après ces transformations est donc tel que les formules qui caractérisent les ensembles d'états initiaux (resp. finaux, de répétition) de $A(f)$ caractérisent également les ensembles d'états initiaux (resp. finaux, de répétition) de l'automate réduit. Cette précaution n'est pas nécessaire pour l'utilisation actuelle des automates d'explication, mais va dans le sens d'une implantation future de transformations préservant l'équivalence explicationnelle.
L'automate réduit est obtenu après ces transformations en calculant son graphe quotient pour la bisimulation forte. Ce calcul est réalisé par le système Aldébaran [Fe88].

4.4. Calcul d'une séquence explicative minimale.

$A(f) = (Q_A, \rightarrow_A, I, F, R, V_\lambda)$, et q un élément de I. Les éléments de $S^0(q \models f)$ sont représentés par des chemins particuliers de $A(f)$:

- Si $f = pot[f_1]f_2$, les éléments de $S^0(q \models f)$ sont les séquences issues de q, d'accès à F, dont les états intermédiaires appartiennent à Q_A.
- Si $f = some[f_1]f_2$, les éléments de $S^0(q \models f)$ sont les séquences issues de q, d'accès à F, dont les états intermédiaires appartiennent à Q_A, et les séquences composées :
  - d'une séquence issue de q, d'accès à R, dont les états intermédiaires appartiennent à Q_A : soit r le dernier état de cette séquence,
  - de la répétition infinie d'une séquence issue d'un successeur de r, d'accès acyclique à r, dont les états intermédiaires appartiennent à R.

Ainsi, dans tous les cas, calculer un élément de $S^0(q \models f)$ se ramène un problème du même type : calcul d'un séquence d'accès unique de Q_1 à Q_2 qui ne passe que par Q_3 où $Q_3 = Q_A$.
Nous n'avons tenu compte d'aucun critère particulier pour choisir une séquence explicative minimale plutôt qu'une autre. En conséquence, la séquence exhibée est la première séquence calculée par l'algorithme : celui-ci est un parcours par niveau de l'automate, la première séquence calculée est donc celle qui visite un plus petit nombre d'états. Cette réalisation a été décrite dans [Ra88].

4.5. Calcul d'un expression ω-régulière représentant $L(A)$.

4.5.1. Introduction.

Comme nous l'avons vu au paragraphe 4.2.2, $L(A)$ s'exprime simplement en fonction d'ensembles réguliers que nous avons notés $W(i, j)$. Nous allons calculer des expressions régulières associées à ces ensembles, et en déduire une expression ω-régulière de $L(A)$.
Le calcul des expressions régulières à partir de l'automate A peut s'effectuer par exemple selon la méthode de Kleene [Kl56]. Cette méthode est une forme particulière de l'élimination de Gauss-Jordan, appliquée à la matrice représentant les transitions de l'automate. En fait, on peut reformuler le problème en termes de calcul d'ensembles de chemins dans un graphe, et de représentation de ces ensembles. Ainsi, nous distinguons :

- un problème de calcul de l'ensemble de chemins d'un graphe faisant partie des problèmes classiques du genre "chemins d'origine donnée", "chemins d'origine et d'extrémité donnée", "$\forall p, q$, chemins d'origine p et d'extrémité q" [Ta81].
- un problème de représentation d'un tel ensemble par une expression, c'est-à-dire sous une forme linéaire.

La première phase, par un algorithme du type de celui de Kleene, a un coût en temps d'exécution de l'ordre de n^3 (n étant le nombre d'états de l'automate), comme nous le verrons en 4.5.2. La seconde phase a un coût qui est, au mieux, de l'ordre du nombre de symboles apparaissant dans l'expression régulière obtenue. Or l'algorithme de Kleene peut calculer des expressions régulières dont la longueur est une fonction exponentielle du nombre d'états de l'automate (et du nombre de symboles du vocabulaire).
Considérons par exemple l'automate représenté sur la figure 4.4, d'état initial 0 et d'état final n, tel que l'ensemble des successeurs de l'état i est l'ensemble $[i+1, n]$:

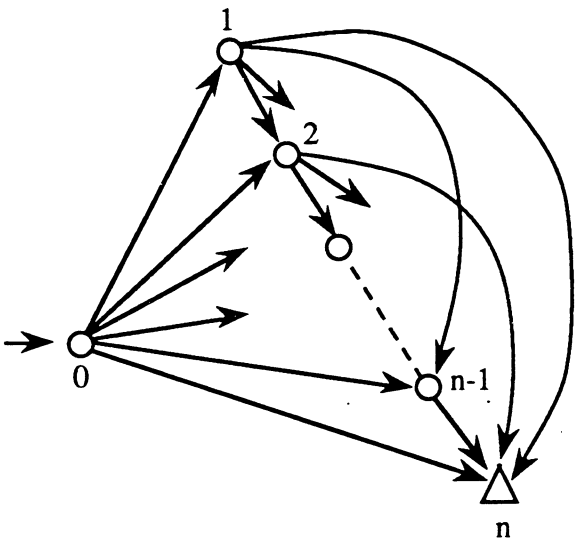


figure 4.4

La longueur de l'expression régulière représentant les chemins de 0 à n calculée par cet algorithme est de l'ordre de 2^n.

La démarche habituelle concernant la complexité des algorithmes nous ferait conclure que la complexité de la première phase, étant polynomiale, ne joue qu'un rôle mineur. Nous devons cependant réexaminer cette question. La complexité de la première phase est de l'ordre de n^3 dans tous les cas. Celle de la seconde phase peut être, dans les cas favorables, de l'ordre de n, ou même indépendante de n.
Or il est clair que nous ne nous intéressons en pratique qu'à ces cas favorables : si la longueur de l'expression obtenue est telle que l'utilisateur ne peut en tirer d'information, il

est inutile de la calculer. C'est donc en réalité la première phase donc la complexité est critique, surtout en ce qui concerne la mémoire nécessaire pour représenter les ensembles de chemins. Nous nous sommes donc attachés à utiliser un algorithme, pour cette phase, qui ne nécessite une taille mémoire de l'ordre de n^3 que dans les cas défavorables. Nous pouvons ainsi traiter des automates de taille significative.

Le système Cléo, dans son état actuel, ne permet que la production d'une expression ω-régulière pour communiquer à l'utilisateur les informations calculées. Nous nous sommes placés dans la perspective du développement d'autres outils de dialogue, appelés à compléter celui-ci, et à le remplacer dans les cas où l'expression ω-régulière se révèle inadaptée.

4.5.2. Passage d'un automate à une expression ω-régulière.

L'expression ω-régulière à calculer représente l'ensemble :

$$L(A) = \bigcup_{i\in I, k\in F} W(i, k) \cup \bigcup_{i\in I, r\in R} W(i, r)(W(r, r))^{\omega}$$

comme nous l'avons vu au paragraphe 4.2.2.

Le calcul de cette expression peut donc se ramener à celui des expressions de :

$W(i, k)$ pour tout état initial i, et pour tout état final k,

$W(i, r)$ pour tout état initial i, et pour tout état de répétition r,

et $W(r, r)$ pour tout état de répétition r.

Ces expressions sont des expressions régulières. L'algorithme classique pour le calcul de ces expressions, à partir de l'automate, est celui de Kleene [Kl56]. Notons 1, 2, ...n les états de l'automate et $Q_k = \{i \,/\, 1 \le i \le k\}$, $Q_0 = \varnothing$. On définit par récurrence un ensemble $C(i, j, k)$ représentant les ensembles de chemins de i à j qui ne visitent que les états d'indice inférieur ou égal à k :

$$C(i, i, 0) = \varepsilon \cup act(i, i)$$

$$C(i, j, 0) = act(i, j) \quad (i \neq j)$$

$$C(i, j, k) = C(i, j, k\text{-}1) \cup C(i, k, k\text{-}1).C(k, k, k\text{-}1)^*.C(k, j, k\text{-}1).$$

où $act(i, j) = \alpha_1 \cup \alpha_2 \cup \ldots \cup \alpha_p$, $\{\alpha_1, \alpha_2, \ldots, \alpha_p\} = \{\alpha \in V_\lambda \,/\, i \rightarrow_A \alpha\, j\}$

Les ensembles cherchés sont de la forme $C(i, j, n)$. Le nombre d'équations à considérer (pour $k \ge 1$) est donc n^3.

Ce schéma d'algorithme s'adapte à différents problèmes sur les ensembles de chemins selon ce que l'on désire calculer. Par exemple, dans le cas du calcul de l'ensemble des plus courts chemins dans un graphe valué, on obtient l'algorithme de Floyd. La complexité de l'algorithme de Floyd est de n^3 en temps d'exécution et n^2 en taille mémoire. En effet, chaque équation est traitée en temps indépendant de n (il s'agit d'un simple calcul de

minimum), chaque résultat peut être mémorisé en taille constante (par exemple une longueur et un numéro de sommet), et il suffit de mémoriser n^2 éléments.
Pour parler de la complexité de l'algorithme, il est donc nécessaire de préciser ce que l'on désire calculer, pour évaluer :

- le temps de résolution d'une équation,
- la taille de mémoire nécessaire par élément.

Les expressions régulières, comme nous l'avons dit, peuvent être de longueur exponentielle. Cependant, le nombre d'expressions différentes intervenant dans les calculs est évidemment inférieur au nombre d'ensembles $C(i, j, k)$, soit n^3. En considérant que chaque expression de cet ensemble est représentée par le quadruplet de ses sous-expressions, chacune est mémorisée en taille constante. Ceci nécessite une taille de mémoire totale en $O(n^3)$. Chaque équation peut alors être traitée en temps constant, d'où un temps total d'exécution en $O(n^3)$.
La structure ainsi définie correspond à la représentation arborescente des expressions, mais avec mise en commun des sous-expressions communes. Ainsi, on n'a pas un arbre (dont la taille serait du même ordre que la longueur de l'expression), mais un graphe sans circuit. Le passage de $C(i, j, n)$ à l'expression régulière est un parcours infixé de l'arbre des chemins (de ce graphe) issus du sommet associé à $C(i, j, n)$. La hauteur de cet arbre est $O(n)$.

Chapitre 5

Exemple d'utilisation de Cléo

5.1. Le protocole du bit alterné

Le protocole du bit alterné est un protocole classique utilisé pour rendre fiable l'échange de messages entre deux processus qui communiquent entre eux par une ligne défectueuse. Une description détaillée en est donnée en [BSW69].
Il est composé de quatre processus : l'émetteur, le récepteur et deux media. Chaque message envoyé par l'émetteur est accompagné alternativement de la valeur 0 ou 1, le récepteur attendant la valeur correspondante pour valider le message. Il renvoit alors un accusé de réception portant la même valeur, pour indiquer à l'émetteur de continuer la transmission de messages. Si l'émetteur reçoit un accusé de réception contenant une valeur de contrôle incorrecte, il renvoit alors le dernier message expédié. Ceci permet dans une certaine mesure de détecter les pertes de messages non signalées par les media.

L'organisation entre ces processus est décrite dans la figure 5.1. *Transmitter* est le nom processus émetteur, *Receiver* du processus récepteur. Le medium *m* sert au transfert de message entre l'émetteur et le récepteur. Le medium *mm* effectue la transmission des accusés de réception. Ils peuvent différer la transmission des messages, les répéter ou les perdre. La signification des noms des actions est la suivante:

. *in* : action de réception d'un message à envoyer par le protocole.
. *out* : action d'envoi du message transmis à une couche supérieure.
. *tm*: transmission du message du *Transmitter* au medium *m*.
. *rm* : transmission d'un accusé de réception du *Receiver* au medium *mm*.
. *lost* : perte du message par un medium.

. $sin.bit_x$: valeur de contrôle, x est égal à 0 ou 1 suivant le cas.

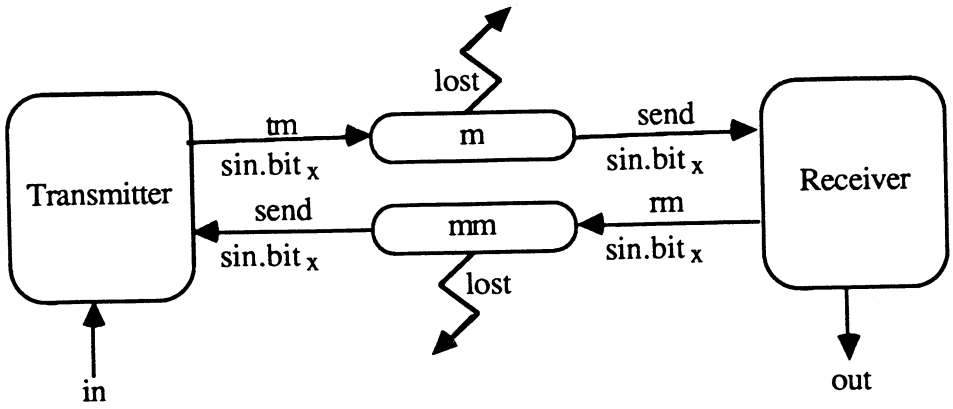


figure 5.1. Le protocole du bit alterné.

Une description en ESTELLE/R est donnée en Annexe B. Dans la description considérée, la perte de message se traduit par la réémission du message par l'émetteur au bout d'un certain délai (time-out), ce qui est un traitement simplifié des pertes de messages.
Lors de la traduction du programme par un graphe d'états, les noms d'actions sont préfixées par le nom du processus qui l'exécute l'action : par exemple, le passage d'un message du *Transmitter* au medium *m* est noté *Transmitter.tm*, la perte d'un message par le medium *mm* est notée *mm.lost*.

5.2. Utilisation de Cléo sur une description de ce protocole

Dans cet exemple, on évalue la formule $\neg$*after*(*transmitter.in*) $\vee$ *inev*(*enable*(transmitter.in)) (*after*(*transmitter.in*) $\Rightarrow$ *inev*(*enable*(*transmitter.in*))) qui exprime que l'émetteur, après avoir reçu un message de l'extérieur, reviendra inévitablement dans un état où il pourra recevoir un nouveau message. Il ne s'agit pas d'une spécification usuelle de ce protocole, et elle n'est pas valide pour la description proposée, qui est par ailleurs correcte. On montre ici de quelle manière Cléo peut être utilisée pour comprendre, à travers des explications de la non-validité de cette formule, la possibilité et la cause de comportements particuliers du protocole.

Cette formule est évaluée à faux pour 3 états sur 63. L'utilisateur fait alors appel à Cléo pour obtenir une explication de la non-validité de la formule. (on trouve en Annexe A la description des menus de XESAR et de Cléo).

Une explication de la non-validité d'une formule est une explication de la satisfaisabilité de sa négation :

```
Formula: after(transmitter.in) and some not enable(transmitter.in)
Formula true for 3 states of 63 states.
```

L'utilisateur désire obtenir une explication particulière. Il peut choisir l'état d'origine de la séquence (parmi ceux où la formule est satisfaite), et de la faire précéder d'une séquence d'accès depuis l'état initial du programme. La touche * permet d'avoir la liste des états où le formule est satisfaite :

```
Which state from (number)? *
2.22.47.
Which state from (number)? 2
With access sequence from init (y/n)? y
```

Résultat fourni par Cléo :

```
ACCESS SEQUENCE :
1-(transmitter.in0)(transmitter.in)->2
     2      |==    after(transmitter.in) and some not enable(transmitter.in)
    BECAUSE :
     2      |==    after(transmitter.in)
AND
     2      |==    some not enable(transmitter.in)
```

Cette explication indique que l'état 2 (état du programme après réception d'un premier signal de l'environnement) vérifie la formule *after(transmitter.in)* $\wedge$ *some* $\neg$*enable(transmitter.in)* car il vérifie les deux opérandes de la conjonction. Ceci correspond à une réécriture de l'assertion 2 $\models$ *after(transmitter.in)* $\wedge$ *some* $\neg$*enable(transmitter.in)*

Pour obtenir une explication de la sous-formule : *some* $\neg$*enable(transmitter.in)*, l'utilisateur modifie la formule courante à l'aide des fonctions de navigation à sa disposition. Cléo se positionne sur le nœud correspondant :

```
Formula :some not enable(transmitter.in)
Formula true for 58 states of 63 states.
```

On est alors en mesure de demander une séquence explicative pour cette sous-formule à partir de l'état 2 :

```
     2      |==    some not enable(transmitter.in)
    BECAUSE :
```

```
    2      |==   some not enable(transmitter.in)
    |
    :      |==   not enable(transmitter.in)
    |
    V
    2      |==   not enable(transmitter.in) WITH CIRCUIT
VIA :      2-3-(transmitter.tm0)(transmitter.tm)(m.sin.bit0)
                                        ->4-(m.lost0)(m.lost)->5-7-9-11-2
```

L'explication fournie indique que l'état 2 fait partie d'un circuit du graphe le long duquel la transition transmitter.in ne peut jamais s'exécuter : il n'est donc pas inévitable que le programme atteigne un tel état après réception du message initial.

Afin de mieux cerner le problème, l'utilisateur décide de focaliser son attention sur les média, et donc de réduire le champ d'observation aux actions qui les concernent. Il redéfinit donc l'ensemble des actions visibles :

```
{transmitter.tm,receiver.rm,m.send,mm.send,m.lost,mm.lost}
```

et demande que seuls les états origine des noms d'actions visibles soient affichés lors d'une explication. La nouvelle séquence explicative est alors la suivante :

```
     2      |==   some not enable(transmitter.in)
   BECAUSE :
     2      |==   some not enable(transmitter.in)
     |
     :      |==   not enable(transmitter.in)
     |
     V
     2      |==   not enable(transmitter.in) WITH CIRCUIT
VIA :2-3-(transmitter.tm)->4-(m.lost)->2
```

Du point de vue du médium *m*, ce circuit correspond à l'exécution infinie d'une transmission du message entre le *Transmitter* et *m*, et de la perte de ce message par ce dernier. L'utilisateur veut maintenant savoir si toutes les explications sont de ce type : il demande donc l'expression ω-régulière représentant toutes les séquences explicatives de *some* ¬*enable*(*transmitter.in*).

Cléo détermine la partie du graphe du programme représentant l'automate d'explication de la formule, et le réduit en un automate équivalent. A titre indicatif, sur cet exemple, les caractéristiques de l'automate avant et après réduction sont les suivantes :

	Automate d'origine :	Automate réduit :
nombre d'états :	58	14
nombre d'arcs :	80	19
transitions muettes :	55	0

L'expression ω-régulière reconnue par l'automate est la suivante (le symbole @ désigne ω):

```
GENERATED REGULAR EXPRESSION
   ({transmitter.tm}{m.lost})@
+
      ({transmitter.tm}{m.lost})*
   .
      {transmitter.tm}
   .
      {m.send}
   .
      (   {receiver.rm}
      .
         {mm.lost}
      .
               ({transmitter.tm}{m.lost})*
            .
               {transmitter.tm}
         +
            ({transmitter.tm}{m.lost})@
      .
         {m.send})@
```

Le premier terme de l'expression régulière correspond au cas de la séquence d'explication précédente, c'est à dire à une suite infinie de pertes de messages par le medium *m*. Le deuxième terme correspond à une transmission réussie du *Transmitter* au *Receiver*, puis à une première perte de l'accusé de réception par le medium *mm*. Puis on rentre dans un cycle

infini de perte de message soit par *m*, soit par *mm*. Les cas d'invalidité de la formule se réduisent donc à des pertes infinies de messages par un des media, c'est à dire par exemple à la coupure de la ligne entre le medium *m* et le *Receiver*.

Conclusion

Bilan

L'objectif de ce travail était la définition et l'élaboration de diagnostics d'erreur en XESAR, les erreurs se manifestant par la non satisfaction des spécifications exprimées dans un langage de formules temporelles arborescentes.
Nous nous situons dans le cadre de l'aide à la mise au point par des techniques d'analyse des comportements du programme. La spécificité de notre approche est de construire d'abord le modèle représentant les exécutions possibles du programme, puis d'en étudier les propriétés.
Le problème de la détection des erreurs étant déjà résolu par ailleurs, nous nous sommes intéressés à l'élaboration de diagnostics. Ne pouvant déduire de la phase de détection toutes les informations permettant d'obtenir un diagnostic, nous avons cherché une méthode pour le calculer a posteriori.
Les résultats sont relatifs à une logique particulière. Cependant l'approche est générale à toutes les logiques dans lesquelles sont exprimables des propriétés de sûreté et de vivacité.

Nous avons montré qu'un diagnostic d'erreur pour de telles spécifications exprimées par une formule f, pouvait être obtenu à l'aide d'un arbre appelé explication de $\neg f$, formé à partir de séquences d'exécutions particulières du programme. Parmi l'ensemble des explications, nous avons montré qu'il était possible d'extraire un sous-ensemble fini d'explications dites minimales qui est suffisant pour exhiber toutes les causes de la non validité de f.
Les explications, inexploitables dans la pratique à cause de leur taille, sont fournies à l'utilisateur sous une forme réduite : les simplifications sont effectuées en fonction de la formule et d'un ensemble d'actions visibles du programme permettant de décrire les séquences d'exécution intervenant dans l'explication.

Nous avons défini une relation d'équivalence entre les modèles d'une même formule f, appelée équivalence explicationnelle, telle que l'ensemble des explications réduites de f est le même dans des modèles équivalents, au nom des états près. Cette relation d'équivalence présente l'originalité de dépendre de la formule : elle est donc plus faible qu'une relation qui préserverait l'ensemble des explications réduites pour toutes les formules.
On donne une méthode de calcul de la forme normale d'un modèle modulo cette relation. C'est dans cette forme normale que sont calculées effectivement les explications.
Il existe un prototype de Cléo. La définition de ses fonctionnalités et la majeure partie de sa réalisation sont antérieures à l'obtention des résultats sur l'équivalence explicationnelle. Par conséquent, une adaptation doit en être faite en fonction des résultats obtenus.
Les méthodes de calcul d'une explication minimale particulière et de l'ensemble des explications d'une assertion ont été présentées. Le calcul d'une explication particulière est une recherche de plus court chemin dans un graphe. Le calcul de l'ensemble des explications est le calcul de l'expression ω-régulière acceptée par un automate de Büchi.
Nous pensons que Cléo apportera une aide réelle à la mise au point des programmes, mais seule une expérimentation approfondie sur des cas réels nous permettra de conclure.

Perspectives

Nous envisageons pour l'avenir d'orienter nos efforts dans trois directions :

1) Chercher s'il est possible d'adapter les résultats à d'autres méthodes de vérification telles que l'évaluation à la volée qui fait l'objet de la nouvelle version de XESAR : nous avons vu en effet que cette méthode d'évaluation permettait d'obtenir directement le diagnostic en cas d'erreur.

2) La recherche d'une relation d'équivalence appropriée s'est avérée plus difficile qu'on ne le pensait. L'équivalence proposée est satisfaisante, mais elle n'est pas la plus faible possible. Par exemple, il suffit qu'une sous-formule de la formule f à expliquer soit de la forme $some[f_1]f_2$ pour que tous les circuits de transitions muettes apparaissent dans la forme normale sous la forme d'une boucle τ, même si ces circuits n'apparaissent dans aucune explication de f. Définir une relation d'équivalence la plus faible possible est un but particulièrement attractif, bien que semblant difficile à atteindre.

3) Amélioration de l'utilisation de Cléo.
Nous envisageons d'améliorer le prototype et de le tester sur un ensemble de cas réels. Les résultats de ces tests doivent nous amener à faire évoluer l'outil existant de façon à le rendre

mieux adapté aux besoins. Cette évolution comprendra entre autre l'amélioration de l'interface avec l'utilisateur, ainsi que de nouvelles fonctionnalités.

- Sans avoir étudié toutes les possibilités qu'offrirait une interface graphique, il est clair que celle-ci nous permettrait de donner une forme plus lisible des diagnostics. Lorsqu'il s'agit de représenter toutes les séquences explicatives d'une formule, la représentation graphique d'un automate serait de beaucoup préférable à l'expression ω-régulière que nous calculons actuellement.
- Un éventail plus large de fonctionnalités peut être offert à partir de la réalisation existante. En effet, la réponse à de nombreuses possibilités d'interrogations autres que celles que nous proposons est la solution d'un des problèmes suivants : évaluation d'une formule sur le graphe du programme, recherche d'un plus court chemin entre deux ensembles d'états, calcul de tous les chemins entre deux ensembles d'états.
- Les séquences explicatives minimales n'ont pas été entièrement exploitées. Nous envisageons par exemple de particulariser, dans le calcul de l'ensemble des séquences explicatives, celui de l'ensemble des séquences explicatives minimales.

Références.

[Ar89] A. Arnold. MEC : a System for Constructing and Analysing Transition Systems. *Proc. Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989.*

[AM86] E. Adams, S.S. Muchnick. Dbxtool : A Window-based symbolic debugger for sun workstations. *Software Practice and Experience, 16(7) : 653-669, July 1986.*

[B89] P.C. Bates. Debugging Heteronogenous Distributed Systems Using Event-based Models of Behavior. *Proc. Workshop on Parallel Distributed Debugging, 11_22, May 1988.*

[BDER79] G. Bristow, C.Drey, B.Edwards, W.Riddle. Anomaly detection in concurrent programs. *Proc. 4th Int. Conf. Software Eng., 1979.*

[BDV86] F. Baiardi, N. DeFrancesco, G. Vaglini. Development of a debugger for a concurent language. *IEEE Trans. on Software Eng. SE_12(4) : 547-553, April 1986.*

[BMP81] M. Ben-Ari, Z. Manna, A. Pnueli. The Temporal Logic of Branching Time. *Proc. 8th Annual ACM Symp. on Principles of Programming Languages, 1981.*

[BSW69] K.A.Barlett, R.A. Scaintlebury, P.T. Wilkinson. A note on reliable transmission over half duplex links. *CACM 12, 1969.*

[Bü62] J.R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. *Proc. Internat. Congr. Logic, Method ans Philos. Sci. 1960, Standford Univeristy Press 1962, pp 1-12.*

[BW83] P.C. Bates, J.C. Wielden. High-level debugging of distributed systems : the behavioral abstraction approach. *Journal of Systems and Software, (3) : 255-264, 1983.*

[CBG87] E.M. Clarke, M.C. Browne, O. Grümberg. Characterizing Kripke Structures in Temporal Logic : *Colloquium on Trees in Algebra and Programming, Pisa, Italy, March 1987.*

[CES83] E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications : A Practical Approach. *Proc. of the 10th ACM Symposium on Principles of Programming Languages, Austin, January 1984, pp 117-141.*

[CGK87] E.M. Clarke, O. Grümberg, R.P. Kurshan. A Synthesis of Two Aproaches for Verifying Finite State Concurrent Systems. *Proceedings of Logic a Botic, USSR, 1989.*

[CS89] D. Callahan, J. Subhlok. Static Analysis of Low-level Synchronization. *Proc. of Workshop on Parallel and Distributed Debugging. SIGPLAN Notice 24(1) : 100-111, May 1988.*

[EH83] E.A. Emerson, J.Y. Halpern. "Sometimes" and "Not Never" Revisited : On Branching Versus Linear Time. *POPL 83.*

[Fe88] J.C. Fernandez. ALDEBARAN : un Système de Vérification par Réduction de Processus Communicants. *Thèse de Doctorat, UJF Grenoble, Mai 988.*

[Fe90] J.C. Fernandez. ALDEBARAN : A Tool for Verification of Communicating Processes. *RTC14, January 1990.*

[Fo89] A.Forin. Debugging Heterogeneous Parallel Systems. *Proc. of Workshop on Parallel and Distributed Debugging, May 1988.*

[Ga89] H. Garavel. Compilation et Vérification de Programmes en LOTOS. *Thèse de Doctorat. U.J.F. Grenoble, Novembre 1989.*

[GGK84] H. Garcia-Molina, F. Germano, W.H. Kohler. Debugging a distributed computing system. *IEEE Trans. on SE, SE-10(2).: 210-219, March 1984.*

[Gr89] R.Groz. Verification de Propriétés Logiques des Protocoles et Systèmes Répartis par Observation de Simulation. *Thèse de Doctorat. Université de Rennes I. Janvier 1989.*

[HW88] D.Haban, W. Weigel. Global Events and Global Breakpoints in Distributed Systems. *Proc. of Hawaï International Conference on System Sciences, 166-175, 1988.*

[Ho72] C. Hoare. Toward a theory of parallel programming. *ed. C.Hoare and R. Perrot, Operating Systems Techniques, Academic Press, 1972.*

[ISO85] ESTELLE. A Formal Description Technique Based on an Extended State Transition Model. *ISO/TC97/SC21. December 1985.*

[Ke89] A. Kerbrat. Interprétation de propriétés de programmes. Application à Cléo. *DEA, U.J.F Grenoble, Juin 1989.*

[Kl56] S.C. Kleene. Representation of Events in Nerve and Finite Automata. *Automata Studies, C.E. Shannon and J.McCarty Eds., Princeton Univbersity Press, Princeton, N.J., 1956, pp 3-40*

[Le89] E. Leu. Techniques de déverminage pour programmes parallèles. *Ecole Polytechnique Fédérale de Lausanne, Département d'Informatique, Laboratoire de Systèmes d'Exploitation. Rapport interne 89/01, Mai 1989.*

[LM87] T.J. LeBlanc, J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. on Computers, C-36(4) : 471-482, April 1987.*

[MC89] B.P. Miler, J.D. Choi. A Mechanism for Efficient Debugging of Parallel Programs. *Proc on Parallel and Distributed Debugging, May 1988.*

[McD88] C.E. McDowell. A Practical Algorithm for Static Analysis of Parallel Programs. *Journal of Parallel and Distributed Computing, 6, 515-536, 1989.*

[McDH88] C.E. McDowell, D.P. Helmbold. Debugging Concurrent Programs. *University of California, Santa Cruz, Computer Research Laboratory. Technical Report 88-21, November 1988.*

[Mi80] R. Milner. A Calculus of communicating systems. *LNCS 92, Springer Verlag, 1980.*

[NV90] R. De Nicola, F. Vaandrager. Three Logics for Branching Bisimulation. *Procs LICS 90.*

[PL89] D.Z. Pan, M.A. Linton. Supporting reverse execution of parallel programs. *Proc. Workshop on Parallel and Distributed Debugging, 124-129, May 1988.*

[Pn77] A. Pnueli. The Temporal Logic of Programs. *Proc 18th Symp; on Foundations of Computer Science. Providence, November 1977, pp 46-57.*

[QS82] J.P. Queille, J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. *International Symposium on Programming, LNCS 137, 1982.*

[Ra88] A. Rasse. Cléo : Interprétation de la non-correction de programmes sur un modèle. *RTC10, Juin 1988.*

[Ro88] C. Rodriguez. Specification et Validation de Systèmes en XESAR. *Thèse de Doctorat, INPG, Grenoble 1988.*

[RRSV87] J.L. Richier, C. Rodriguez, J. Sifakis, J. Voiron. Verification in XESAR of the Sliding Window Protocol. *Proc. IFIP WG 6.1 7th International Conference on Protocol Specification, Testing, and Verification. Noth Holland, Zurich 1987.*

[SB84] B. Sarikaya, G.V Bochman. Synchronisation and Specification Issues in Protocol Testig. *IEEE Trans. on Comm, April 1984, pp 389-395.*

[St85] C. Stirling. A Compositionnal Reformulation of Owicki-Gries's Partial Correctness for a Parallel While Language. *Internal Report, University of Edinburgh, CSR-189-85, August 1985.*

[Ta81] R.E. Tarjan. A Unified Approach to Path Problems. *JACM Vol 28 No 3 July 1981, pp 557-593.*

[TO80] R.N. Taylor, L.J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. on Software Eng. 265-278, May 1980.*

[Ve89] F. Vernadat. Vérification Formelle d'Applications Réparties. Caractérisation Logique d'une Equivalence de Comportement. *Thèse de Doctorat de l'Université P. Sabatier de Toulouse, Novembre 1989.*

[VW86] M. Y. Vardi, P. Wolper. An Automata-Theorethic Approach to Automatic Program Verification. *Proc of the Confernece on Logic in Computer Science, Boston, Mass., June 1986.*

[WVS83] P. Wolper, M.Y. Vardi, A.P. Sistla. Reasoning about Infinite Computation Paths. *Proc 24th IEEE Symp on Foundations of Computer Science, Tuscon, 1983, pp185-194.*

[YT88] M. Young, R.N. Taylor. Combining Static Concurrency Analysis with Symbolic Execution. *IEEE Trans. on Software Eng. SE-14(10) : 1499-1511, October 1988.*

Annexe A : Description des menus

Menu de l'évaluateur.

XESAR EVALUATION PHASE

Program name : bitalt

0. Exit
1. Help
2. Unix command
3. Set option V

11. Enter Formulas
12, Read Formulas
13. Dump Graph
14. Set option I
15. Reduction

21. Enter Alias
22. Read Alias
23. List Alias
24. Save Alias
25. Delete Alias

31. Enter Predicates
32. Read Predicates
33. List Predicates
34. Save Predicates
35. Delete Predicates
36. Evaluate Predicates

Commandes :

11 et 12 : Commandes d'entrée de formules. Les formules sont alors évaluées.

13 : Crréation une image du graphe dans un fichier.

14 : Activation / désactivation de l'appel systématique de Cléo après évaluation.

15 : option en cours de développement.

21 à 25 : Commandes de manipulation des alias. Les alias sont des abbréviations pour des noms d'actions ou de formules, destinés à être utilisés lors de l'évaluation.

31 à 36 : Commandes de manipulation des prédicats définis par l'utilisateur. Ces prédicats sont alors évalués.

Menus de Cléo.

XESAR DIAGNOSTIC PHASE (CLEO)

0. Return to evaluation
1. Help
2. Unix command
3. Dump automaton
4. Options
11. Left sub formula
12. Up formula
13. Right sub formula
14. Explanation sequence
15. Automaton reduction
16. Automaton informations
17. Regular expression

Commandes :

3 : Création de l'image d'un automate dans un fichier.

4 : Appel du menu des options de CLEO.

11 à 13 : Modification de la formule courante.

14 : Mode pas à pas : génération d'une séquence explicvative pour la formule courante.

15 : Réduction de l'automate de la formule courante (si il existe).

16 : Informations sur l'automate de la formule courante (si il existe).

17: Génération d'une expression régulière à partir de l'automate de la formule courante, avec réduction préalable de l'automate.

Menu des options de Cléo.

OPTIONS (CLEO)

0. Return to CLEO
1. Help
2. Unix command
3. Timing : on
10. Echo file : echo
11. Visible actions file : None
12. Visible actions definition
13. Visible actions list
14. Visible states : all

Comandes :

3 : Calcule et affiche le temps d'exécution lors de la réduction d'un automate ou du calcul d'une expression régulière. L'appel de la commande fait basculer l'indicateur. Par défaut, on.

10 : Nom du fichier dans lequel apparaît tous les résultats affichés à l'écran. Par défaut, le nom du fichier est "CLEO.echo".

11 : Nom du fichier contenant l'ensemble des actions visibles. Par défaut, pas de fichier.

12 : Définition 'à la main' de l'ensemble des actions visibles.

13: Liste de l'ensemble des actions définies visibles. Par défaut, cet ensemble contient toutes les actions à l'exception de λ.

14 : indique le mode d'affichage des états dans les séquences d'explication :

. **all** tous les états sont affichés

. **some** seuls les états 'intéressants sont affichés, i.e les états de début et fin de séquence, ainsi que les états à partir desquels on a une transition visible.

. **none** aucun numéro d'état n'est affiché.

Par défaut, l'option all est activée. Chaque appel à cette commande fait changer l'option, dans l'ordre donné ci-dessus.

Annexe B : Description ESTELLE/R du protocole du bit alterné

```
(*      Alternating bit protocol with delays.        *)
(*      Written in Estelle/R (with rendez-vous)      *)
(*      Renaming conventions:                        *)
(*       mr is m.send,  mt is mm.send                *)
(*       lm is m.lost,  lmm is mm.lost               *)

specification bit_altern;

const dt =3;  (* delay for the transmitter to retransmit    *)
    dm =1;    (* transit delay for the two mediums          *)

channel transport;
  bit0; bit1;

module extremity process(sin: transport; sout: transport);

  body transmit for extremity;
    state: (t0,t1,t2,t3,t4,t5);
    initialize to t0 begin end;
    trans
      {: in, in0 } from t0 to t1 delay(0,*) begin end;
    trans
      from t1 to t2 begin {: tm, tm0 } output sout.bit0 end;
    trans
      from t2 to t1 delay(dt) begin end;
              to t1 when sin.bit1 begin end;
              to t3 when sin.bit0 begin end;
    trans
      {: in, in1 } from t3 to t4 delay(0,*) begin end;
    trans
      from t4 to t5 begin {: tm, tm1 } output sout.bit1 end;
```

```
    trans
      from t5 to t4 delay(dt) begin end;
              to t4 when sin.bit0 begin end;
              to t0 when sin.bit1 begin end;
end;

body receive for extremity;
    state: (r0,r1,r2,r3,r4,r5,r6);
    initialize to r0 begin end;
    trans
      from r0 to r4 when sin.bit1 begin end;
              to r1 when sin.bit0 begin end;
    trans
      {: out, out0 } from r1 to r2 begin end;
    trans
      from r2 to r3 begin {: rm, rm0 } output sout.bit0 end;
    trans
      from r3 to r2 when sin.bit0 begin end;
              to r4 when sin.bit1 begin end;
    trans
      {: out, out1 } from r4 to r5 begin end;
    trans
      from r5 to r6 begin {: rm, rm1 } output sout.bit1 end;
    trans
      from r6 to r5 when sin.bit1 begin end;
              to r1 when sin.bit0 begin end;
end;

module medium process(sin: transport; sout: transport);


body B_medium for medium;
    state: (m0,m1,m2);
    initialize to m0 begin end;
    trans
      from m0 to m1 when sin.bit0 begin end;
              to m2 when sin.bit1 begin end;
```

```
        trans {: lost, lost0 }
          from m1 to m0 delay(0,dm) begin end;
        trans
          from m1 to m0 delay(dm) begin {: send, send0 } output sout.bit0 end;
        trans {: lost, lost1 }
          from m2 to m0 delay(0,dm) begin end;
        trans
          from m2 to m0 delay(dm) begin {: send, send1 } output sout.bit1 end;
    end;

var transmitter, receiver: extremity;
    m, mm: medium;

initialize begin
    init transmitter with transmit;
    init receiver with receive;
    init m with B_medium;
    init mm with B_medium;
    connect transmitter.sin to mm.sout;
    connect transmitter.sout to m.sin;
    connect receiver.sin to m.sout;
    connect receiver.sout to mm.sin;
end;
end.
```

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- **Monsieur AZEMA Pierre**
- **Madame GAUDEL Marie-Claude**

Mademoiselle RASSE Anne

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 **12 Juin 1990**

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU