# Vérification de propriétés de programmes flots de données synchrones

Anne-Cecile Glory

**THESE**

présentée par

**GLORY Anne-Cécile**

pour obtenir le titre de **DOCTEUR**
**de L'UNIVERSITE JOSEPH FOURIER - GRENOBLE I**

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

# VERIFICATION DE PROPRIETES DE PROGRAMMES FLOTS DE DONNEES SYNCHRONES

thèse soutenue le 14 Décembre 1989
Composition du jury :

Président : J-P. Verjus
Examinateurs : P. Azéma
J-L. Bergerand
M-C. Gaudel
P. Jorrand
F. Ouabdesselam
J. Sifakis

Thèse préparée au sein du Laboratoire de Génie Informatique

A mes parents,
les pélicans de la fable

Je tiens à remercier

Monsieur Jean-Pierre Verjus, Directeur de recherche au CNRS, Directeur de l'Institut I.M.A.G., de m'avoir fait l'honneur de présider le jury de cette thèse,

Monsieur Pierre Azéma, Directeur de recherche au CNRS, pour avoir accepté de juger ce travail,

Monsieur Jean-Louis Bergerand, de la société MERLIN GERIN, responsable du projet SAGA, pour ses critiques constructives, particulièrement sur les parties de mon manuscrit de thèse liées au contexte industriel. Je le remercie également pour la bonne entente qu'il a su faire régner au sein de son équipe,

Madame Marie-Claude Gaudel, Professeur à l'Université d'Orsay (Paris XI), pour avoir accepté de participer à ce jury,

Monsieur Philippe Jorrand, Directeur de recherche au CNRS, pour avoir accepté de juger ce travail,

Monsieur Farid Ouabdesselam, Maître de Conférences à l'Université Joseph Fourier (Grenoble I), pour avoir dirigé et suivi ce travail. Sa compétence en génie logiciel, alliée à sa connaissance du milieu industriel m'ont permis une bonne approche du problème posé. Sa diligence à se pencher sur mes problèmes ont permis l'aboutissement de cette thèse. Je lui suis reconnaissante de sa gentillesse et de ses encouragements,

Monsieur Joseph Sifakis, Directeur de recherche au CNRS, pour m'avoir accueilli au sein de son équipe, et avoir accepté de participer à ce jury.

Cette thèse ayant été réalisée dans le cadre d'une convention CIFRE, je tiens à remercier aussi tous les participants de cette collaboration, tant dans l'équipe Spécification et Analyse des Systèmes du Laboratoire de Génie Informatique de Grenoble, que dans le département Systèmes et Electronique de Sûreté de MERLIN GERIN. Et plus particulièrement,
dans l'équipe SAS du LGI :

Messieurs Nicolas Halbwachs et Daniel Pilaud, pour leurs idées et l'aide précieuse qu'ils m'ont apportés par leur maîtrise du langage LUSTRE,

Monsieur Ahmed Bouajjani, Mademoiselle Suzanne Graf et Monsieur Pascal Raymond pour leur gentille disponibilité,

et dans le département SES de Merlin Gerin :

Monsieur Patrick Bouteiller, pour sa disponibilité aux problèmes des "thésards CIFRE". Je le remercie d'avoir pris le temps de nous recevoir mensuellement.

Madame Katty Laurent, pour sa collaboration dans les phases préliminaires de mon étude. Merci à elle d'avoir accepté d'être le pionnier du développement en SAGA,

Monsieur Philippe Sanchis, pour avoir contribué au démarrage de cette thèse dans son contexte industriel.

Je remercie encore

tous les collègues et amis du LGI et de Merlin Gerin/SES qui ont rendus mon séjour agréable dans ces deux établissements,

avec une mention spéciale pour Jean-Marie Favre dont l'amitié ne s'est jamais démentie.

les membres du service reprographie pour le soin et l'efficacité qu'ils ont apportés au tirage de cette thèse,

les services aériens, postaux, et la valise diplomatique pour le bon acheminement du courrier en provenance de Jakarta.

# Introduction

Dans le cadre de cette thèse nous nous sommes intéressés à la vérification de systèmes réactifs critiques et temps réel. La vérification de ce type de logiciels est un enjeu important. Dans les domaines comme le nucléaire, la chimie, la défense, le spatial ou les transports, une mauvaise réaction du système informatique peut causer des dégats importants à l'environnement, éventuellement porter atteinte à la santé, et même à la vie, des individus, ou causer la perte de matériel coûteux.

Pour la vérification de systèmes réactifs, deux approches sont généralement envisagées :

- l'approche à **langage unique**, dans laquelle le système et les spécifications sont décrits dans le même formalisme. Si le langage choisi est une algèbre [Mil,80] [BK,85], la vérification consiste à définir une équivalence et à s'assurer que deux termes de l'algèbre sont équivalents.
- l'approche à **deux langages**, dans laquelle le système et ses spécifications sont décrits dans deux formalismes différents. Le langage de spécification est plus abstrait et décrit ce qu'on attend du système. En général ce langage est une logique, et la vérification est réalisée soit par déduction [MC,81] soit par évaluation sur un modèle à l'aide d'une relation de satisfaction [EL,85] [QS,82].

Le travail de cette thèse concerne la vérification de propriétés sur des programmes conçus à l'aide de l'atelier SAGA [BP,88] développé à MERLIN GERIN/SES. Le langage de conception utilisé dans l'atelier SAGA est un langage flots de données synchrone, contrairement à beaucoup de langages dédiés à la programmation de systèmes réactifs, mais similaire en cela, au langage LUSTRE [CPHP,87]. L'atelier SAGA et son langage de conception sont décrits dans le chapitre 1. Nous avons formalisé la sémantique du langage SAGA. Cette sémantique peut être exprimée en LUSTRE. Ainsi, toute application développée au moyen de SAGA peut être traduite en un programme LUSTRE à la fin de la phase de conception. Une description informelle de LUSTRE est fournie au chapitre 1, ainsi que la traduction en LUSTRE des constructions de base du langage SAGA.

Au cours des chapitres suivants, le problème de la vérification est alors étudié sur des programmes LUSTRE. Cela nous permet d'élargir le champ des programmes visés par notre vérification, puisque, s'il y a inclusion stricte du langage SAGA dans LUSTRE, la réciproque n'est pas vraie.

Nous avons choisi de réaliser la vérification de propriété par la méthode d'évaluation sur des modèles ("model checking") qui s'inscrit dans l'approche de vérification à deux langages. Les modèles de programmes LUSTRE que nous avons utilisés pour la vérification, sont, respectivement, un graphe d'états finis, appelé graphe d'états LUSTRE, et l'automate de contrôle généré par le compilateur LUSTRE. Ces deux modèles ont été obtenus par repliage d'un arbre synthétisant toutes les exécutions possibles du programme (arbre des exécutions). Leur différence

réside dans l'abstraction des expressions non booléennes qui est réalisée, avant le repliage, dans le cas de l'automate de contrôle. L'arbre des exécutions, le graphe d'états et l'automate de contrôle ont été formalisés et sont décrits dans le chapitre 2.

Les applications décrites à l'aide de SAGA, sont généralement des systèmes réactifs de sûreté (domaine d'activité de MERLIN GERIN / SES). Nous avons alors cherché à caractériser le type des propriétés qu'il est nécessaire de vérifier pour de telles applications. De l'étude menée sur un certain nombre de systèmes réactifs de sûreté, nous avons déterminé les besoins d'expression pour les contraintes qu'ils devaient vérifier. Les conclusions auxquelles nous sommes arrivés sont exposées au chapitre 3 et ont donné lieu à la définition formelle d'un langage d'expression des propriétés : LEP. Les principales conclusions de ces études sont que les propriétés à vérifier sont généralement des propriétés de sûreté ("safety" [AS,85] et [Lam,84] dans notre cas), et que leur énoncé emploie souvent un opérateur temporel du passé : *au cycle précédent* . La sémantique des formules de LEP a été définie en terme d'arbre des exécutions.

Nous avons ensuite envisagé d'utiliser des logiciels de vérification existants. Dans ces logiciels, le modèle du programme est en général un système de transitions. Le chapitre 4 décrit des tentatives pour trouver un langage de spécification existant, dont l'expressivité serait suffisante pour exprimer toutes les formules de LEP. L'adaptation des logiques temporelles du futur et de leurs extensions, les μ-calculs propositionnels, à l'expression de LEP, y est étudiée à travers deux langages CL[QS,83] et μ-CL. Certains opérateurs de LEP sont exprimables en μ-CL et non en CL, ce qui confirme, s'il en est besoin, que μ-CL est strictement plus expressif que CL qu'il englobe. Notons néanmoins, que les opérateurs de LEP non traduisibles en CL, n'ont pas été utilisés pour formuler des propriétés dans les études de cas, mais ont été ajoutés par la suite afin d'augmenter l'expressivité de LEP. Le souci de la formalisation nous a amené à démontrer la correction de nos traductions. Aucun de ces deux langages ne permet de traduire la construction *au cycle précédent* qui est un opérateur du passé.

L'étude d'une expérience [Rat,88] de validation de programmes LUSTRE à l'aide de l'outil XESAR [RRSV87b] nous a permis de mieux appréhender le problème que pose l'opérateur *au cycle précédent* pour l'évaluation des propriétés sur des graphes d'états. Cet opérateur, comme l'opérateur *pre* de LUSTRE, manipule la notion d'état prédécesseur au cours d'une séquence d'exécution. Le problème provient du fait que sur un graphe, un état q peut avoir plusieurs prédécesseurs par la relation de transition : $\{q' / q' \rightarrow q\}$, alors que dans chaque séquence du graphe passant par q, q n'a qu'un seul prédécesseur.

L'étude de cette expérience a aussi permis de nous rendre compte que les langages utilisés en général pour décrire le modèle du programme et qui sont asynchrones [Est,85], engendrent lors de la génération du graphe d'états, des états superflus pour la vérification de propriétés en LUSTRE. Ceci conduit à une augmentation du temps de vérification et à une perte de place mémoire.

Nous avons alors essayé de définir une méthode qui prenne en compte le synchronisme du programme et la notion de passé du langage d'expression des propriétés.

Nous avons défini le langage MODAL LUSTRE qui s'appuie sur les opérateurs de LUSTRE ( et en particulier l'opérateur du passé *pre* ) et leur associe des modalités temporelles du futur. L'approche de vérification associée est réalisée sur le graphe d'états LUSTRE, ce qui résoud le problème dû à l'asynchronisme. Le problème posé par l'opérateur *pre* est résolu par modification du graphe.

Nous nous sommes ensuite intéressés à la vérification de propriétés suivant une méthode propre à LUSTRE. L'évaluation est réalisée sur un automate de contrôle généré par le compilateur LUSTRE. Par suite de l'abstraction sur les expressions non booléennes qui est réalisée lors de la génération des automates de contrôle, il n'est possible de vérifier par notre méthode que des propriétés de sûreté. Celles-ci sont exprimées dans le langage INVARIANTS LUSTRE, dont les formules sont une composition, à l'aide des opérateurs *et* et *ou* du calcul booléen, de sous-formules de la forme : toujours (E), où E est une expression LUSTRE booléenne.

Dans notre méthode, la propriété n'est plus une entité à part, mais est incorporée au programme sous la forme d'une équation LUSTRE : v = toujours (E), où v est maintenant une sortie du programme. La valeur de l'expression E apparait ainsi dans tout état de l'automate, et l'évaluation de la formule toujours (E) devient triviale. Cette méthode est exposée au chapitre 5.

Le chapitre 6 décrit les différentes expériences de vérification que nous avons menées à l'aide de notre méthode. Il décrit un prototype d'outil de vérification : LESAR, que nous avons réalisé. LESAR met en œuvre notre méthode et évite le problème commun de "l'explosion" du nombre d'états du graphe modèle, en réalisant l'évaluation au cours de la génération des états, évitant ainsi de conserver ces états.

# 1 Les langages SAGA et LUSTRE

SAGA [BP, 88], développé à Merlin Gerin/SES (département Systèmes et Electronique de Sûreté), est un atelier de conception de programmes conçu pour assurer la qualité de logiciels complexes dans le domaine du temps réel. Les logiciels considérés sont de plus des systèmes à haut degré de sûreté, notamment dans les domaines du nucléaire civil et de la défense. Dans le cycle de vie des logiciels retenu par SES pour développer et maintenir ces produits, le processus de développement comprend les étapes suivantes (représentées selon une structure en V) :

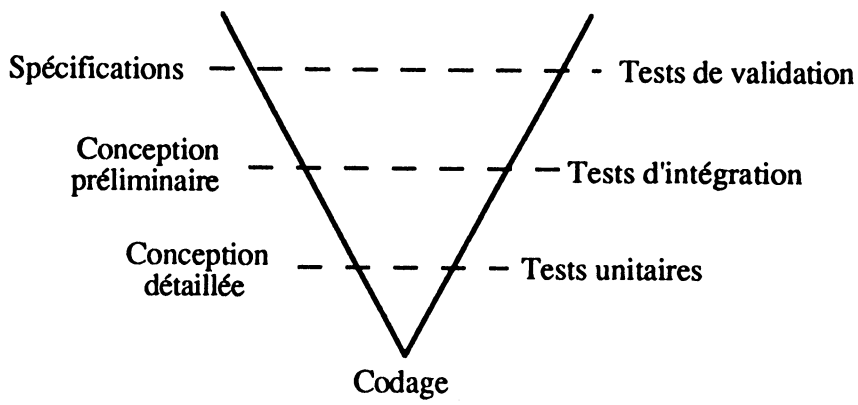



Figure 1.a : processus de développement des logiciels à SES

SAGA couvre dans ce processus les étapes de conceptions, préliminaire et détaillée, et de codage. Il permet de produire les documents dûs à l'issue de chaque phase. Il met en œuvre plusieurs outils : l'un d'eux permet de réaliser la conception, un autre la génération de code à partir de cette conception.

Merlin Gerin/SES désire intégrer à SAGA un outil de validation des logiciels. Cet outil doit permettre l'expression d'une partie des spécifications du système (consignées dans le cahier des charges), notamment les propriétés de sûreté; il doit aussi servir à la vérification de ces propriétés sur la conception obtenue à la fin de la phase de conception détaillée. Cette vérification doit permettre de renforcer la confiance de l'utilisateur dans la qualité du logiciel produit.

Le langage utilisé pour le développement des logiciels dans l'atelier SAGA (phase de conception) est de type **flot de données** et est muni d'une **interprétation synchrone**. Ces deux caractéristiques ont permis la mise en œuvre de principes de validation particuliers.

Ce chapitre décrit le langage SAGA, son environnement de programmation (l'outil SAGA) et, le langage LUSTRE et son compilateur [CPHP, 87]. LUSTRE et SAGA sont fondés sur les mêmes principes; LUSTRE interviendra dans la définition de la sémantique du langage SAGA. Nous verrons au cours des prochains chapitres que le processus de validation s'appuie sur les propriétés de LUSTRE.

Dans une première partie, sont présentés les points communs des langages SAGA et LUSTRE. Viennent ensuite, dans les deux paragraphes suivants, les traits propres à chacun de ces langages et leur environnement de programmation ; une attention particulière est portée à la compilation des programmes LUSTRE. Enfin, une quatrième partie présente la traduction de programmes SAGA en programmes LUSTRE.

## 1.1 POINTS COMMUNS ENTRE SAGA ET LUSTRE : LANGAGES FLOTS DE DONNÉES SYNCHRONES

### 1.1.1 Origine : systèmes réactifs et temps réel

Le terme "**systèmes réactifs**" au sens de [Pnu, 86] désigne des systèmes informatiques qui réagissent à des événements provenant de façon répétitive de leur environnement en produisant eux-mêmes des sorties vers cet environnement : processus de contrôle en temps réel, automatismes de contrôle divers (distributeurs de billets de banque, interfaces clavier/souris), jeux vidéo, ... Cette typologie a été introduite par opposition d'une part aux *systèmes transformationnels* – les programmes classiques, qui disposent de leurs entrées dès leur initialisation, et délivrent leurs résultats lors de leur terminaison – et d'autre part aux *systèmes interactifs* – qui interagissent continûment avec leur environnement, mais à leur vitesse propre (par exemple, les systèmes d'exploitation).

Les particularités des systèmes réactifs sont les suivantes:

- Il s'agit de *systèmes parallèles* : en effet, au moins le parallélisme d'exécution du système et de son environnement doit être considéré. De plus il est généralement naturel de concevoir ces systèmes comme des ensembles d'activités parallèles et coopérantes.

- Ils sont soumis à des *contraintes temporelles* : clairement, la correction d'un tel système ne tient pas seulement à la valeur des résultats (fonction du système), mais également au moment où ces résultats sont émis. Ceci induit, d'une part, la nécessité d'exprimer ces contraintes, et d'autre part celle d'obtenir des performances temporelles suffisantes pour les satisfaire.
- Ce sont des systèmes *déterministes* : la réaction du système à un événement donné est complètement déterminée, ceci par opposition, par exemple, à certains systèmes interactifs dont le comportement peut varier selon des paramètres de charge.

SES s'est spécialisé dans la conception et le développement de systèmes réactifs, et en particulier de systèmes réactifs temps réel. Les systèmes temps réel doivent répondre à leurs entrées en respectant des contraintes de temps fixées par le monde extérieur. Ils possèdent des caractéristiques inhérentes aux propriétés des processus physiques avec lesquels ils interagissent : impossibilité de les interrompre, évolution continuelle, irréversibilité. Il leur est impossible de se synchroniser avec les processus physiques autrement que par le respect de contraintes temporelles.

La notion de temps réel est liée totalement au processus avec lequel le système interagit. Pour les processus physiques lents (certaines réactions chimiques par exemple), les contraintes de temps sont si lâches que tout programme un peu rapide les remplit.

Dans le passé, la programmation des systèmes temps réel s'est effectuée principalement en langage assembleur. Cela présentait l'avantage de gérer finement les temps d'exécution, mais avait par contre l'inconvénient d'offrir au programmeur un très faible niveau d'abstraction et d'interprétation, rendant ainsi le développement et la mise au point des systèmes longs et difficiles.

Ainsi la programmation des système temps réel a évolué vers l'utilisation de langage de haut niveau :

- programmation de tâches en langages évolués classiques, et gestion des tâches via un système d'exploitation (exécutif temps réel);
- programmation dans des langages parallèles généraux (ADA, OCCAM ...).

Ces deux dernières approches méritent d'être commentées:

- L'utilisation d'un exécutif temps réel entraine une séparation complète entre les tâches et leur système de gestion. Il s'ensuit un manque de clarté, puisque le parallélisme et la communication sont complètement cachés au programmeur des tâches. D'autre part, s'il est possible de faire réaliser par l'exécutif une gestion soigneuse des tâches (interruptions, priorités ...) afin de

satisfaire les contraintes temporelles, ces contraintes ne sont pas exprimées dans le programme. Par conséquent, l'analyse et la vérification des programmes sont difficiles.

- Les langages parallèles généraux ont surtout été conçus pour la programmation des systèmes interactifs.

Les mécanismes de base de la plupart d'entre eux proviennent de langages conçus pour la programmation des systèmes d'exploitation: dans ce domaine, on veut décrire des activités qui se déroulent en temps partagé, et peuvent communiquer et se synchroniser entre elles. Les concepts de base sont ceux de processus séquentiels, de communication via une mémoire partagée, de primitives de synchronisation (sémaphores, moniteurs, ...). Pour simplifier la sémantique de ces langages, la communication et la synchronisation ont été unifiées dans des concepts de plus haut niveau: rendez-vous (en CSP ou ADA), file d'attente (dans les langages à flot de donnée [Ka,74], [WA, 85]).

Un des objectifs primordiaux de ces langages est la portabilité des programmes: par suite, le comportement fonctionnel d'un programme doit être indépendant des performances de l'architecture hôte, et un effort important a donc été fait pour rendre ce comportement indépendant du temps d'exécution des processus qui composent le programme. Dans ces conditions, les primitives temporelles introduites dans ces langages (dans ADA, par exemple) ne permettent pas à l'utilisateur de maîtriser totalement les temps d'exécution. Prenons l'exemple des primitives forçant l'attente d'un processus pendant un certain délai. Une instruction "délai 5s" force le processus qui l'exécute à l'instant t, à attendre jusqu'à t+5 secondes. Le problème qui se pose alors est que le programmeur ne peut pas savoir à quel instant un processus atteint une telle instruction. En effet, dans ces langages, le temps est pris en compte de manière asynchrone : quand plusieurs processus sont actifs, la séquence d'exécution est l'un des entrelacements d'instructions compatibles avec l'ordre des instructions dans chacun des processus. Donc, comme l'ont mentionné les auteurs d'ESTEREL [BMR, 83], la séquence "délai 3s; délai 5s" n'est généralement pas équivalente à "délai 8s". Les solutions qui ont été proposées à ce problème dans le cadre asynchrone (priorités,...) sont de très bas niveau.

Enfin, ces langages sont généralement indéterministes, d'une part pour permettre l'expression de l'indéterminisme inhérent aux systèmes d'exploitation, et d'autre part pour assurer l'indépendance temporelle mentionnée plus haut. Notons que le test et la vérification de programmes indéterministes est intrinsèquement plus difficile que celle de programmes déterministes.

Les langages synchrones [Mil,83] ont été introduits pour faciliter la tâche du programmeur en lui donnant des primitives de programmation "idéales" permettant de raisonner comme si le programme réagissait *instantanément* aux événements (**hypothèse de synchronisme**). La question de savoir ce qui se passe lorsqu'un événement externe survient alors que le programme n'a pas terminé le traitement de l'événement précédent, ne se pose alors plus. Chaque événement interne du programme est précisément daté, et le comportement du programme est complètement déterministe, tant du point de vue fonctionnel que temporel.

L'hypothèse de synchronisme se heurte pourtant à une objection sérieuse: elle n'est pas directement implémentable, puisqu'il faudrait disposer d'une machine infiniment rapide! Cette objection n'est cependant pas insurmontable, l'essentiel étant que, pour l'utilisateur final du système, tout se passe "comme si" l'hypothèse était vérifiée. Pour cela, il suffit que le programme s'exécute assez rapidement pour traiter chaque événement externe avant l'occurrence de l'événement suivant. Si l'on parvient à satisfaire cette propriété pour une classe importante de programmes, l'hypothèse de synchronisme n'est pas plus déraisonnable que celle qui consiste à considérer qu'une machine manipule de vrais nombres entiers ou réels.

Le problème est donc transposé au plan des performances du code généré par le compilateur d'un langage synchrone. Or le caractère déterministe des langages synchrones permet d'appliquer aux programmes une technique de transformation vers des automates finis purement séquentiels. Cette technique consiste à synthétiser la structure de contrôle séquentiel du programme, par simulation exhaustive, à la compilation, de toute la communication interne du programme. Elle est utilisée en ESTEREL [BMR,83] (à base de processus séquentiels synchrones), en SIGNAL [LBBG,85] (décrivant des réseaux à flots de données et très proche de LUSTRE), en LTS [BFM,84] (basé sur ML [Mil,84], visant à la description et la simulation du matériel), et enfin en SAGA et en LUSTRE, qui sont fondés sur une interprétation synchrone de LUCID [WA,85].

Le temps maximal de transition (temps maximal de traitement d'un événement) est complètement mesurable sur un processeur donné, le code effectué lors d'une transition étant linéaire (pas de boucle). Pour un programme et une machine donnés, on peut donc vérifier la validité de l'hypothèse de synchronisme.

### 1.1.2 Caractéristique : langages flots de données synchrones

Il existe deux grandes approches de la programmation. L'approche classique, ou impérative, dans laquelle l'utilisateur définit la suite des actions à réaliser par la machine pour

calculer les sorties à partir des entrées. L'approche déclarative, dans laquelle l'utilisateur décrit les relations existant entre les données. Il décrit les fonctions qui, appliquées aux entrées, fourniront les sorties, et non une suite d'actions de la machine qui réalise ces transformations. La relation entre les entrées et les sorties du programme est une composition de fonctions.

La programmation flots de données, dont les langages SAGA et LUSTRE sont des représentants, s'inscrit dans l'approche déclarative. Toute donnée du programme (variable ou expression) est une suite infinie de valeurs. Le programme reçoit donc en entrée des suites infinies de valeurs et produit en sortie des suites infinies de valeurs. Chaque valeur de la suite représentant une sortie possède la même relation (composition de fonctions) vis-à-vis des valeurs des suites d'entrées. On peut imaginer les données comme des flots de valeurs circulant à travers un réseau de fonctions, représentant le programme, activées par la circulation du flot de données qu'elles consomment.

Formellement, dans un langage à flots de données ("functional (or pure) pipeline dataflow"), tout système, tout programme, peut être caractérisé par un système d'équations dont les opérateurs sont continus sur des domaines munis d'un ordre partiel complet. Les données produites par le programme forment le plus petit point fixe du système d'équations équivalent [Ka,74].

Un programme flots de données respecte le principe de **causalité** :
Pour tout entier n, le $n^{ième}$ terme d'une suite représentant une sortie de fonction ne dépend que des termes de rang inférieur ou égal à n, des entrées de cette fonction.

SAGA et LUSTRE sont également des langages synchrones. Dans un langage flots de données **synchrone**, les indices des suites de valeurs ont une interprétation temporelle. On peut considérer qu'instantanément le programme consomme ses entrées, calcule et produit ses sorties. Un programme flots de données synchrone peut alors être vu comme une boucle répétée infiniment dont chaque pas serait synchronisé sur une horloge globale et réalisé en un temps nul.

Dans la réalité, si on appelle temps de cycle la durée (physique) séparant deux instants de l'horloge globale, il suffit que les calculs effectués par le programme en un pas de la boucle aient une durée maximale inférieure au temps de cycle. Alors le temps de ces calculs est négligé et l'hypothèse de synchronisme vérifiée.

Un système réactif à flots de données synchrone est correct vis-à-vis des contraintes temporelles qui lui sont imposées, si son temps de cycle est inférieur à la vitesse d'évolution de l'environnement avec lequel il interagit, et qu'on peut éviter les phénomènes d'interférence entre les entrées, les calculs, les sorties.

Les langages SAGA et LUSTRE permettent aussi de définir d'autres horloges que l'horloge de base du programme, mais toujours synchronisées sur celle-ci.

### 1.1.3 Variables, expressions et horloges

Toute variable ou expression représente d'une part une suite de valeurs et d'autre part une suite d'instants, appelée son horloge. Intuitivement, une variable (ou une expression) possède la $n^{ième}$ valeur de sa suite de valeurs au $n^{ième}$ instant de son horloge. En dehors de ces instants, elle n'a pas de valeur.

Une horloge est aussi une expression comme les autres. Ses valeurs sont booléennes. Les instants d'une horloge sont les instants de sa propre horloge pour lesquels sa valeur vaut vrai.

La figure 1.1.3.a présente une horloge HH, une donnée H de type booléen définie sur l'horloge HH et une variable D d'horloge H. La suite des valeurs de D : $(d_1,d_2,...)$ est synchronisée sur les instants de l'horloge H, de même que la suite des valeurs de H est synchronisée sur les instants de HH.

| instants de HH | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| valeurs de H | true | true | false | true | false | true | false | false |
| instants de H | 1 | 2 | | 3 | | 4 | | |
| valeurs de D | $d_1$ | $d_2$ | | $d_3$ | | $d_4$ | | |

Figure 1.1.3.a : les horloges

Deux données peuvent alors évoluer de façon asynchrone l'une par rapport à l'autre. Par exemple, la donnée D' d'horloge H' définie sur HH (fig. 1.1.3.b) n'est aucunement synchonisée avec la donnée D de la figure 1.1.3.a.

| instants de HH | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| valeurs de H' | true | false | false | false | true | true | false | true |
| instants de H' | 1 | | | | 2 | 3 | | 4 |
| valeurs de D' | $d'_1$ | | | | $d'_2$ | $d'_3$ | | $d'_4$ |

Figure 1.1.3.b : asynchronisme des horloges

L'horloge globale est la seule horloge prédéfinie. Elle est encore appelée **horloge de base**. Ainsi, une variable X sur cette horloge de base possède la $n^{ième}$ valeur de sa suite de valeurs au $n^{ième}$ cycle du programme. Nous verrons comment on peut définir d'autres horloges, qui sont toujours des sous-suites de cette horloge de base. Les langages SAGA et LUSTRE ont chacun leur manière propre de définir ces sous-suites.

### 1.1.4 Equations, opérateurs sur les valeurs

Dans un langage flots de données, tout programme peut être caractérisé par un ensemble d'équations :
si X est une variable et E est une expression, l'équation "X=E" définit X comme :
• la suite de valeurs ($e_0$, $e_1$, ..., $e_n$, ...) de l'expression E
• l'horloge de E.
La variable X possède donc la valeur $e_n$ au $n^{ième}$ instant de l'horloge de E.

Les expressions sont construites à l'aide de variables, de constantes (considérées comme des suites constantes sur l'horloge de base) et d'opérateurs. Les opérateurs usuels (arithmétiques, booléens, conditionnels) sont étendus pour opérer point par point sur les suites. Ils sont appelés opérateurs sur les valeurs (ou opérateurs instantanés ) et ne peuvent être appliqués qu'à des opérandes de même horloge. D'après l'hypothèse de synchronisme, ils sont appliqués aux mêmes indices de suite.
Par exemple, l'expression en LUSTRE

*if* X>Y *then* X-Y *else* Y-X

dénote la suite dont le $n^{ième}$ terme est la valeur absolue de la différence des $n^{ièmes}$ termes de X et de Y ; son horloge est l'horloge commune de X et Y.

Les langages conçus sur des principes équationnels vérifient les deux propriétés suivantes:
• **Principe de substitution** : une équation X=E spécifie une synonymie complète entre la variable X et l'expression E. Dans tout contexte, la variable X peut être remplacée par l'expression E et inversement.
• **Principe de définition** : le contexte de l'utilisation d'une expression E ne peut avoir aucune influence sur le comportement de cette expression. En particulier, aucune information ne peut être inférée sur les entrées.

### 1.1.5 Opérateurs sur les suites

En plus des opérateurs sur les valeurs, SAGA et LUSTRE possèdent des opérateurs, appelés opérateurs sur les suites (ou opérateurs temporels) , qui manipulent effectivement les suites et les horloges. Deux de ces opérateurs leur sont communs :

- L'opérateur *précédent* (*MEMO* en SAGA et *pre* en LUSTRE) retourne la valeur de son argument à l'instant précédent de l'horloge de cet argument.
  Si $(x_0,x_1,...,x_n,...)$ est la suite des valeurs de X, la suite des valeurs de *précédent(X)* est $(nil,x_0,x_1,...,x_{n-1},...)$, où **nil** est une valeur indéfinie (semblable à la valeur d'une variable non initialisée dans les langages impératifs). L'horloge de *précédent(X)* est la même que celle de X.
- L'opérateur *suivi de* (*INIT* en SAGA et –> en LUSTRE) sert à initialiser les variables.
  Si X et Y sont deux variables (ou expressions) de même type et de même horloge, de suites respectives $(x_0, x_1, ..., x_n, ...)$ et $(y_0, y_1, ..., y_n, ...)$ , alors *X suivi de Y* est une expression sur la même horloge que X et Y et dont la suite des valeurs est $(x_0, y_1, y_2, ..., y_n, ...)$. Donc *X suivi de Y* est toujours égal à Y sauf à l'instant initial.

Par exemple, l'équation en LUSTRE

X = 0 –> *pre* (X) + 1;

spécifie que X est initialement nul et est ensuite incrémenté de 1 à chaque cycle du programme. X représente donc la suite des entiers naturels, et du fait du mode d'exécution des programmes flots de données synchrones, est un compteur de cycles.

## 1.2 LE LANGAGE SAGA ET L'ATELIER ASSOCIÉ : PARTICULARITÉS

### 1.2.1 Origine

Le langage et l'atelier SAGA découlent de l'idée d'adapter l'utilisation de diagrammes de type SADT [Rs,77] à l'analyse et la conception de programmes d'automatisme temps réel réalisés à Merlin Gerin/SES.

SADT comprend un langage graphique, "Language of Structured Analysis" (SA), et un ensemble de règles et de méthodes pour utiliser ce langage. SADT sert à modéliser de manière semi-formelle des spécifications d'applications. Un modèle SADT consiste en un ensemble ordonné de diagrammes SA. Chaque diagramme contient des nœuds connectés par des arcs. Les diagrammes de base sont de deux types : les "datagrammes" et les "actigrammes". Sur un actigramme, les nœuds décrivent des activités, et les arcs, le flot des données entre ses activités.

Sur un datagramme au contraire, les nœuds sont des données et les arcs des activités. Les arcs attachés à un nœud décrivent soit une entrée consommée par le nœud, soit une sortie produite par le nœud, soit un contrôle exercé sur le nœud, soit un mécanisme mis en œuvre par le nœud. Chaque sortie d'un nœud doit être connectée en entrée ou comme contrôle d'un autre nœud, ou doit être une sortie vers l'environnement extérieur. Chaque entrée ou contrôle d'un nœud doit provenir d'un autre nœud ou de l'environnement extérieur. Les entrées, sorties ou contrôle d'un nœud peuvent être connectés à des nœud sur d'autres diagrammes du modèle. Dans un actigramme, les entrées/sorties sont des flots de données, les mécanismes sont des moyens de mise en œuvre (mécaniques ou humains), et les contrôles sont des données utilisées, mais non modifiées, par le nœud. Dans un datagramme, les entrées sont les activités produisant la donnée, les sorties sont les activités la consommant, les mécanismes sont les unités utilisées pour stocker les représentations des objets décrits par la donnée, les contrôles sont les conditions d'activation du nœud. Les actigrammes constituant un modèle représentent des traitements qui, par défaut, sont réalisés de manière asynchrone et dont l'ordonnancement est induit par les flots les liant.

La méthode SADT fournit une notation et un ensemble de techniques pour décrire et comprendre des spécifications complexes. Parmi celles-ci, il faut noter celles ayant influencé la création de l'outil SAGA : la décomposition des actigrammes de hauts-niveaux en actigrammes subordonnés, la distinction entre les entrées, les sorties et les contrôles, et les techniques de revues de documents comportant des modèles SADT.

Le langage de SAGA a gardé de l'approche SADT la représentation graphique, la démarche de conception descendante, l'idée d'une circulation des données à l'intérieur de boîtes de traitement, la notion de condition d'activation. Pour être adaptée, d'une part, aux systèmes réactifs d'automatisme, d'autre part, au temps réel, la circulation des données a été formalisée en faisant de SAGA un langage à flots de données muni d'une interprétation synchrone. De plus, afin d'augmenter la qualité de la programmation et de l'adapter aux besoins de la sûreté, les données sont typées et les primitives du langage ont une sémantique claire et simple. Le langage SAGA est associé à un outil qui guide et contraint l'utilisateur dans sa démarche de conception, afin d'assurer un maximum de fiabilité aux logiciels réalisés.

### 1.2.2 Aspect externe

Le langage SAGA est graphique. Un programme SAGA se présente sous la forme d'un réseau de boîtes connectées par des fils. Les boîtes représentent les fonctions, et les fils, les données qui circulent. Une fonction F avec deux entrées et une sortie, est représentée par le schéma suivant :

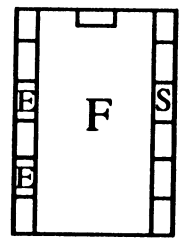



Figure 1.2.2.a : fonction (boîte) SAGA

L'équation $Y = F(X_1, X_2)$ est représentée sur la figure 1.2.2.b :

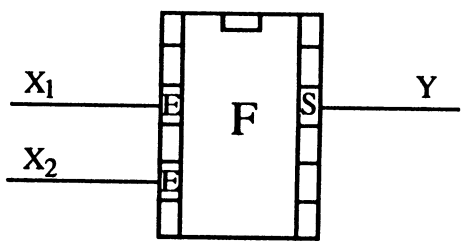
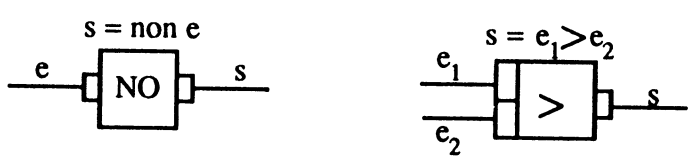



Figure 1.2.2.b : Equation Y = F(X1, X2)

En SAGA, il existe des fonctions prédéfinies appelées opérateurs, ce sont :

• les opérateurs arithmétiques et logiques, unaires et binaires :
Par exemple :




opérateur NON opérateur >

Figure 1.2.2.c : exemple d'opérateurs

• le choix conditionnel s = si cond alors e1 sinon e2 :

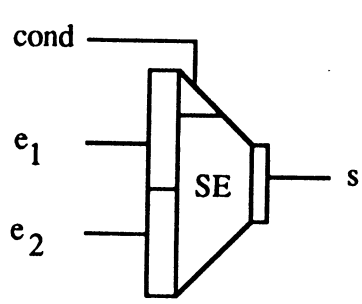



Figure 1.2.2.d : SELECTEUR

• les opérateurs temporels *MEMO* et *INIT* :

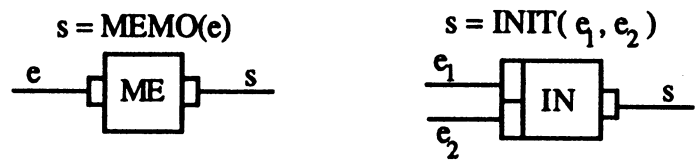



Figure 1.2.2.e : *MEMO* et *INIT*

### 1.2.3 Les opérateurs propres à SAGA

Les fonctions et les opérateurs sont activés par la circulation du flot de données qu'ils consomment. Dans l'interprétation synchrone, si les données arrivent à chaque cycle, les fonctions et les opérateurs sont activés à chaque cycle. Pour que les choses ne se passent pas ainsi, il suffit de filtrer le flot des données à l'entrée d'une fonction ou d'un opérateur : le langage SAGA permet de décrire des données qui sont calculées suivant une condition. Il existe un emplacement particulier sur le dessus des boîtes représentant les fonctions, qui permet de prendre en compte le filtrage : c'est la condition d'activation de la fonction. La construction suivante :

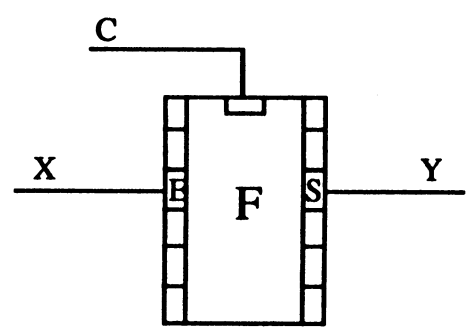



Fig 1.2.3.a : condition d'activation

permet de filtrer l'entrée X selon la condition C (bien entendu, la donnée C doit être de type booléen). La fonction F ne sera exécutée que lorsque la condition C sera vraie. Toute donnée apparaissant dans F est sur l'horloge C. Lorsque la donnée C est fausse, les sorties de F conservent la valeur qu'elles avaient au cycle précédent. Par conséquent, si C est fausse au premier cycle du programme, il y a introduction de valeurs indéfinies. Aussi, l'utilisation de ce type de construction, oblige le concepteur à fournir une valeur d'initialisation des données de sortie de la fonction. Ces valeurs sont utilisées aux premiers cycles tant que la condition C reste fausse, et ne sont pas utilisées sinon.

### 1.2.4 Démarche descendante

La conception d'un programme SAGA est réalisée de manière descendante à partir de sa spécification la plus générale. Elle débute par la définition de l'interface de l'application à l'aide d'un ensemble de commandes de création (cf fig 1.2.4.a et 1.2.4.b). Les entrées/sorties ont un type dont on ignore tout sauf le nom. La suite de la conception va permettre d'affiner progressivement l'application et ses données d'interface pour construire une description la plus complète possible.

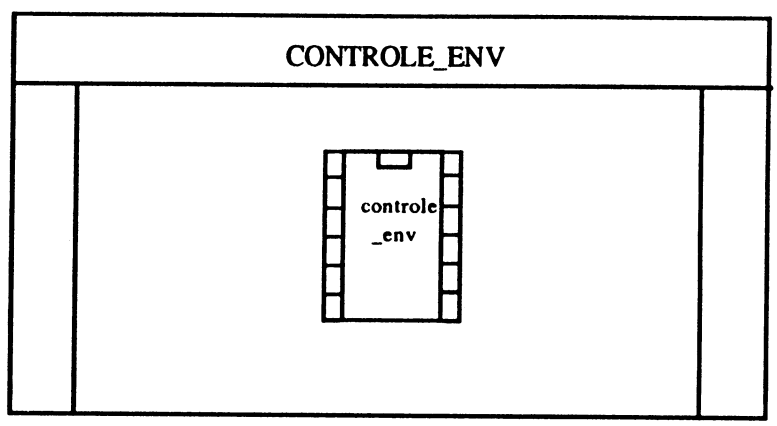



Figure 1.2.4.a : première vue à la création de l'application

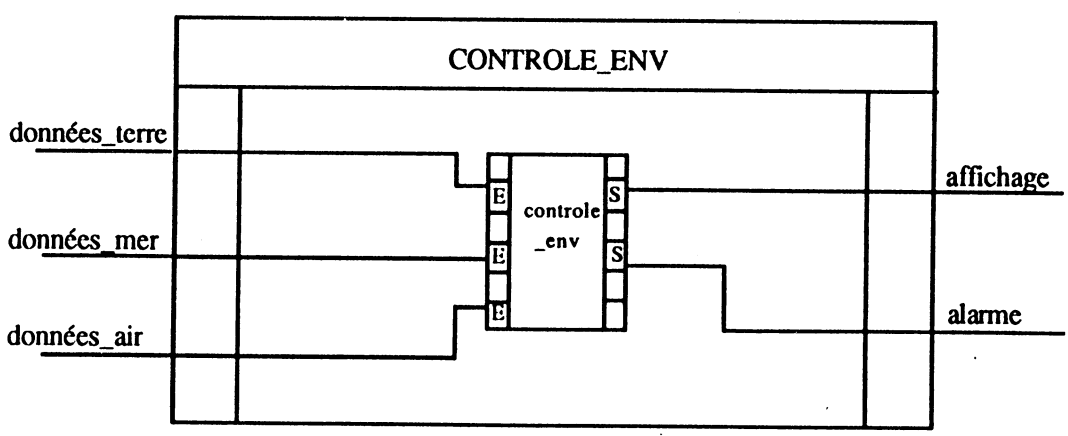



Figure 1.2.4.b : première vue : définition de l'interface de l'application

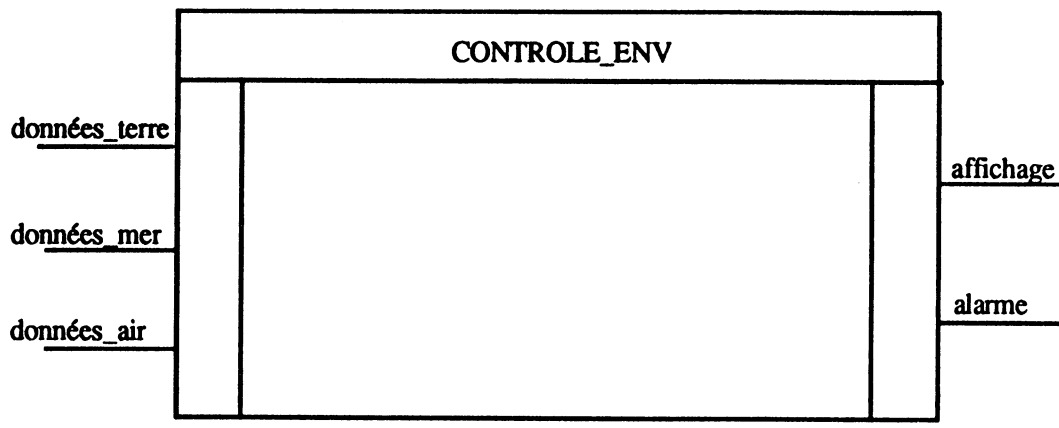



Figure 1.2.4.c : deuxième vue avant l'affinement de l'application

L'affinement est réalisé à l'aide d'un ensemble de commandes. L'interface étant graphique, l'affinement d'une fonction (et en particulier celui de la fonction représentant l'application complète) commence par la création d'une nouvelle vue représentant "l'intérieur" de la fonction, où seules ses entrées et ses sorties sont représentées à l'écran (cf fig 1.2.4.c). Le concepteur doit définir des fonctions et des données pour décrire le comportement de la fonction qu'il affine. Ces fonctions et ces données peuvent être elles-mêmes affinées plus tard. Elles peuvent aussi être déclarées non affinables par le concepteur.

Le comportement de la fonction en cours d'affinement est décrit par un réseau de fonctions et d'opérateurs. Dans un tel réseau, deux fonctions peuvent soit dépendre l'une de l'autre si l'une consomme au moins une des données produites par l'autre, soit être totalement indépendantes l'une de l'autre. Les contraintes de séquencement introduites sont les seules dépendances fonctionnelles.

Une fonction non affinable peut être soit de base (totalement décrite par un réseau d'opérateurs du langage de SAGA), soit décrite dans un autre langage (par exemple C).

Des commandes d'affinement existent également pour opérer sur les données. Seules les données d'entrée ou de sortie de la fonction en cours d'affinement peuvent être détaillées. L'affinement d'une donnée consiste à définir un certain nombre de nouvelles données (nom, type) liées à cette donnée. Dans l'exemple de la figure 1.2.4.c, la première commande pourra être "affiner-donnée" de données-mer. Le concepteur fournit alors les données composant données-mer: données-sous-marines et données-surface, de types respectifs T-sous-marines et T-surface (figure 1.2.4.d).

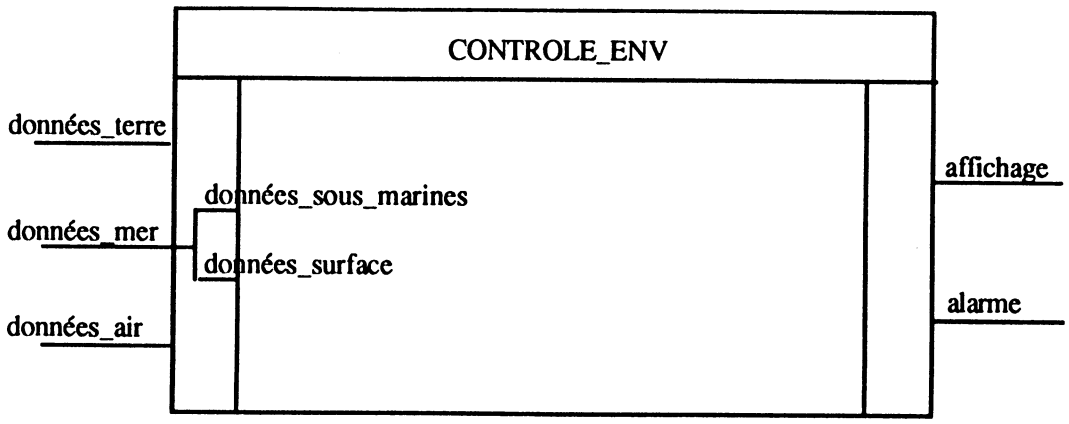



Figure 1.2.4.d : début de l'affinement de l'application : affinement de donnée-mer

De proche en proche, la conception du système va produire :

- un arbre représentant la décomposition de l'application par l'affinement des fonctions en sous-fonctions,
- des arbres représentant la décomposition des données d'interface de l'application par l'affinement des données en sous-données,
- un graphe représentant la dépendance des fonctions par rapport à la production et à la consommation des données.

La conception est terminée lorsque toutes les fonctions déclarées par le concepteur sont à la fois utilisées et complètement définies. Cela correspond à un arbre de décomposition de l'application dont toutes les feuilles sont non affinables.

Cette conception est guidée par l'outil qui ne permet pas de changer de vue (représentation SAGA du premier niveau d'affinement d'une fonction) tant que la vue en cours n'est pas correcte (voir à ce sujet le paragraphe 1.2.5 : vérifications contextuelles) et ne permet l'affinement d'une fonction que si celle-ci a été définie dans une fonction déjà affinée.

### 1.2.5 Vérifications contextuelles

L'outil SAGA réalise un certain nombre de vérifications automatiques. Ce sont des vérifications sur le typage, l'absence de valeur indéfinie et l'absence de blocage.

### 1.2.5.1 Vérification de type

En SAGA, toute donnée ou toute fonction est déclarée obligatoirement avant d'être utilisée. La déclaration d'une donnée comporte notamment l'indication de son type, et celle d'une fonction, le type de son profil (entrées/sorties).

Il y a deux sortes de types : les types de base et les types structurés (ou affinables). Les types de base sont tous les types existant dans le langage cible du générateur de code (actuellement C), ils comprennent en général les types **entier, réel, booléen**. Les types structurés sont caractérisés par leur nom et sont affinés par l'intermédiaire des données. Il n'y a pas d'affinement explicite de type, mais lorsqu'une donnée d de type T est affinée en n données $d_1, \ldots, d_n$ de types respectifs $T_1,\ldots,T_n$, le type T est implicitement affiné en : $T_1 x \ldots x T_n$.

Les données et les fonctions étant typées, une vérification de type peut avoir lieu "en ligne", lors de la connexion d'une donnée à l'entrée ou la sortie d'une fonction.

D'autre part, une même donnée peut avoir plusieurs affinements différents, car elle peut être consommée en entrée par plusieurs fonctions, ou produite et consommée par des fonctions différentes (cf figure 1.2.5.1.a). Par conséquent, l'outil contrôle que les affinements de type correspondants sont cohérents, c'est-à-dire que leurs décompositions en types de base (feuilles de l'arbre d'affinement du type) sont identiques. Pour cela, une vérification est réalisée à chaque nouvel affinement. Dans l'exemple $d_4$ étant de type booléen (condition d'activation), $d_1$ devra être de type booléen.

Aucune conversion de type n'est possible en SAGA. Il peut seulement y avoir déclaration d'identité de deux types de noms différents. L'identité est valable pour toute utilisation de ces types. L'outil ne conserve alors qu'un seul des deux noms dans ses arbres de décomposition.

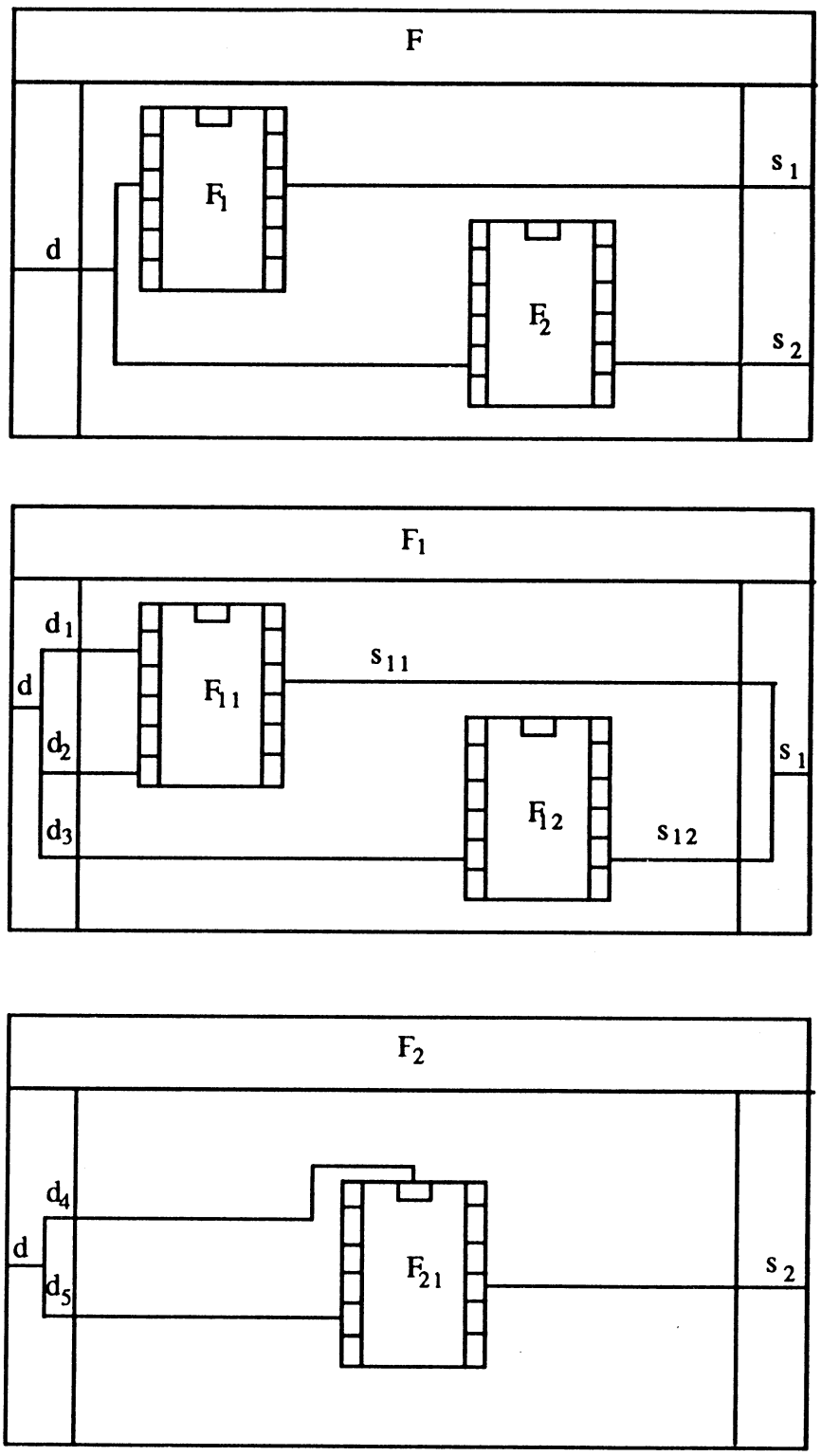



Figure 1.2.5.1.a : affinements mutiples de données

### 1.2.5.2 Contrôle de l'absence de valeur indéfinie

Il consiste à s'assurer qu'aucune valeur indéfinie n'est utilisée pour le calcul d'une sortie. Cette situation peut apparaître dans deux cas particuliers lors de la manipulation des suites de valeurs.

- La première est due au filtrage (voir paragraphe 1.2.3).
- La deuxième peut apparaître lors d'une utilisation de l'opérateur *MEMO*. En effet, l'opérateur *MEMO* donne accès aux valeurs calculées aux cycles précédents. Lors du premier cycle du programme, de telles valeurs sont indéfinies.

Le contrôle exercé par l'outil est le suivant :

- Il oblige le concepteur à définir des valeurs par défaut pour les sorties de toute fonction avec condition d'activation.
- Il vérifie que les données dont la définition utilise des fonctions *MEMO* sont définies par ailleurs au premier cycle c'est-à-dire que l'expression qui les définit contient un opérateur *INIT*. Cette vérification est réalisée "en ligne" lors de la validation de chaque vue. Par conséquent, tout opérateur *MEMO* dans une vue, doit être associé à un opérateur d'initialisation sur la même vue. En effet, l'imbrication de l'initialisation par rapport à la mémorisation, témoigne en général d'une démarche de conception maladroite dans laquelle l'utilisateur n'a pas bien identifié les données en présence et leur circulation. Une vue qui relève d'une telle démarche est rejetée par l'outil.

Cependant cette vérification "en ligne" ne suffit pas et doit être complétée au moment de la génération de code d'une vérification de la cohérence entre les vues (réalisée sur la structure interne représentant la conception, où les appels de fonctions sont remplacés par leur corps). En effet, prenons l'exemple suivant (figures 1.2.5.2 a et b) :

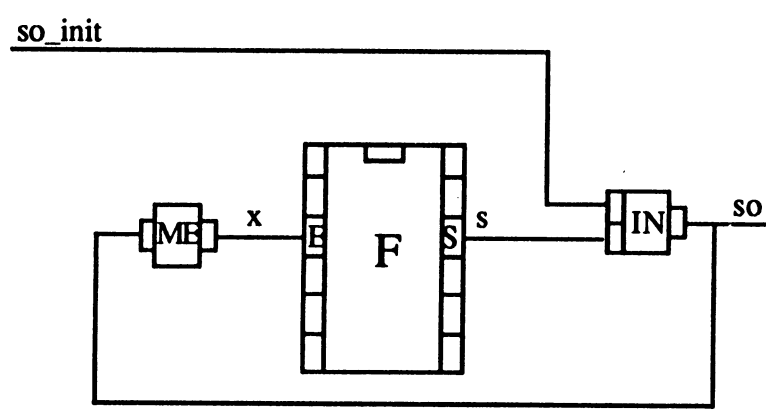



Figure 1.2.5.2.a

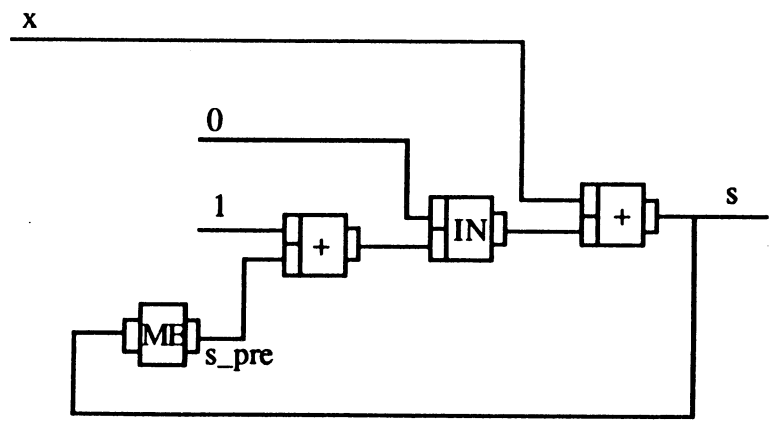



Figure 1.2.5.2.b : Affinement de la fonction F ci-dessus

Les deux vues sont parfaitement correctes, dans le sens où leurs sorties sont toujours définies. Au premier cycle la sortie so (figure 1.2.5.2.a) ne dépend pas de la fonction F, néanmoins celle-ci doit être activée en ce premier cycle. En effet, dans le cas contraire, l'opérateur *INIT* contenu dans l'affinement de F (figure 1.2.5.2.b) n'est pas activé non plus, et les valeurs de la sortie s sont décalées d'un cycle. Mais, si F est activée au premier cycle, cela introduit une valeur indéfinie sur x qui est utilisée pour le calcul de la sortie s de F. Cette sortie s n'est pas utilisée au premier cycle pour le calcul de so (figure 1.2.5.2.a), mais elle est cependant mémorisée pour le second (figure 1.2.5.2.b). Ainsi l'outil, au moment de la génération de code, peut rejeter l'application pour des problèmes de variables non initialisées. Dans le cas évoqué par les figures ci-dessus, la solution que devra apporter le programmeur consiste à ajouter une valeur initale pour x sur la vue 1.2.5.2.a.

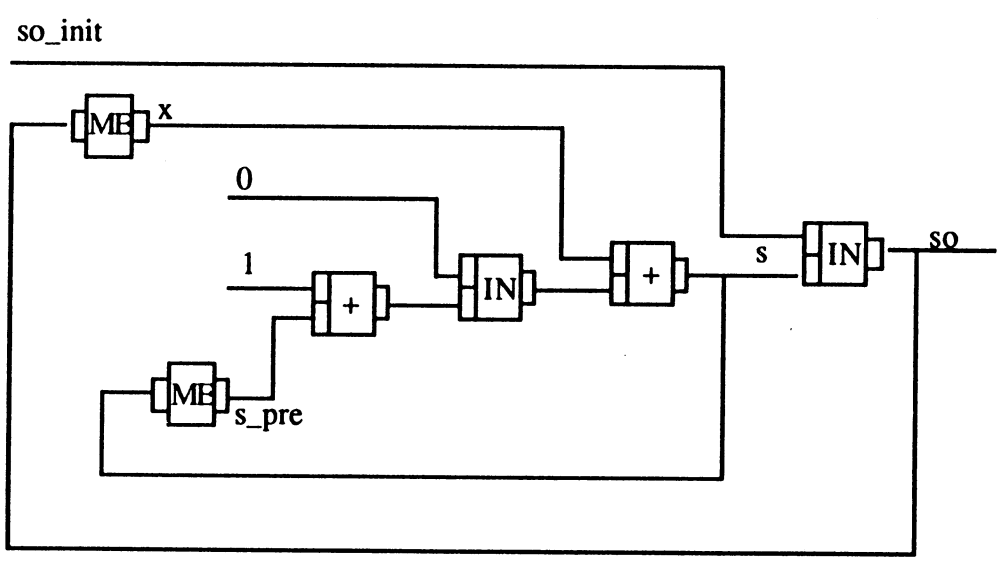



Figure 1.2.5.2.c : Remplacement de F par son corps dans la figure 1.2.5.2.a

### 1.2.5.3 Vérification de l'absence de blocage

Un programme SAGA définit l'ensemble de ses sorties comme la solution d'un système d'équations paramétrées par l'ensemble de ses entrées. Un programme correct vis-à-vis de l'hypothèse de synchronisme admet une solution qui peut être construite pas à pas, et dont le temps de calcul nécessaire à chaque pas est borné. De tels programmes sont dits **sans blocage**, et se caractérisent par le fait qu'aucune variable ne dépend instantanément d'elle-même. Cela signifie qu'on interdit à la compilation des équations de point fixe autres que les équations récurrentes construites avec l'opérateur de retard *MEMO*.

En effet, une équation du type "x = f(x)", où f ne contient pas d'opérateur de retard, n'a pas forcément de solution. De plus, même si elle en a une, on ne peut donner de borne au temps de calcul de cette solution, ce qui viole le principe de synchronisme.

La même démarche de sûreté a été adoptée pour l'absence de blocage que pour l'absence de valeur indéfinie (cf §1.2.5.2), à savoir : préférer contraindre l'utilisateur dans son placement des opérateurs *MEMO* que le voir adopter une démarche de conception maladroite, qui pourrait occasionner des erreurs, et dont la vérification nécessite beaucoup plus d'allers-retours entre vues imbriquées. La vérification de l'absence de blocage est alors réalisée à chaque connexion de donnée.

Par exemple, dans la situation décrite par la figure 1.2.5.3.a (traits pleins), la connexion d'une sortie de G directement sur une entrée de F (tracé en pointillés) est interdit par l'outil de conception. Cette connexion, pour être acceptée, doit comporter nécessairement un opérateur *MEMO*. A moins, que le concepteur ne préfère ajouter l'opérateur *MEMO* sur la connection existante.

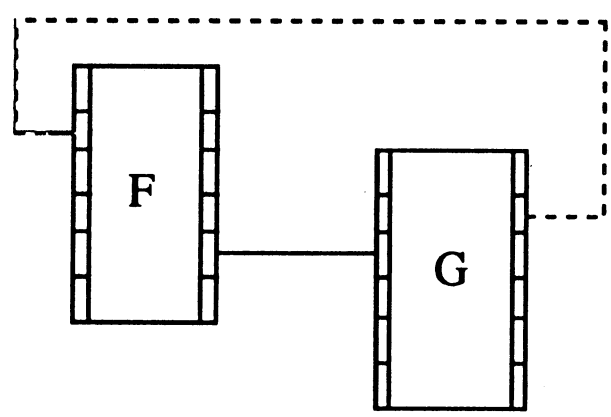



Figure 1.2.5.3.a : vérification de l'absence de blocage lors des connexions

### 1.2.6 L'atelier SAGA

L'atelier SAGA réalise cinq fonctions, chacune étant assurée par un outil. L'outil principal est l'outil de conception que nous avons évoqué dans les paragraphes précédents et qui utilise le langage SAGA.

Remarquons que cet outil permet de reconsidérer des choix de conception : en effet, en plus du mode création (ou mode normal), l'outil peut être utilisé en mode modification. Alors que le mode création ne concerne qu'une fonction à la fois car il est le guide et le garant de la conception descendante, le mode modification peut, lui, affecter plusieurs fonctions. En effet, dans ce mode, une opération sur une fonction peut avoir des conséquences sur tout un ensemble d'autres fonctions dans un arbre d'affinement : par exemple, la suppression d'une donnée d'entrée d'une fonction déjà affinée, entraine des modifications des sous-fonctions consommant cette donnée. L'outil de conception interdira donc de sortir du mode modification tant qu'il restera des modifications à propager.

L'atelier comprend également :

- L'outil de documentation qui permet de mettre en forme l'ensemble des informations données par l'utilisateur lors de la phase de conception.
- L'outil de génération de code qui permet de générer automatiquement le code séquentiel correspondant à la description flots de données faite à l'aide de l'outil de conception.
- L'outil de programmation qui permet au concepteur de coder une fonction dans le langage hôte (actuellement le langage C) en respectant l'interface de cette fonction définie dans l'outil de conception.
- L'outil d'administration qui permet de gérer l'ensemble des applications développées avec l'atelier, de créer de nouvelles applications et de gérer les droits d'accès des différents utilisateurs aux différentes applications.

## 1.3 LE LANGAGE LUSTRE ET SA COMPILATION : PARTICULARITÉS

### 1.3.1 Les opérateurs propres à LUSTRE

Outre les opérateurs -> et *pre*, LUSTRE possède deux opérateurs, sur les suites, qui lui sont propres. Ceux-ci permettent de définir des expressions sur des horloges différentes de

l'horloge de base. Ils répondent aux mêmes motivations que les conditions d'activation des fonctions en SAGA mais n'ont pas le même fonctionnement.

• Si E est une expression et C est une expression booléenne de même horloge que E, *E when C* est une expression dont l'horloge est la suite des instants où C vaut vrai, et dont la suite des valeurs est celle des valeurs prises par E à ces instants.

Les horloges peuvent donc être définies de façon imbriquée : par exemple, la milliseconde peut être représentée par une variable booléenne, vraie à chaque signal d'un quartz, et "seconde" peut être une autre variable booléenne, définie sur l'horloge "milliseconde" (vraie une fois sur mille).

• Si E est une expression d'horloge H, et si H n'est pas l'horloge de base, alors *current(E)* est une expression sur la même horloge que H et dont la valeur à chaque instant est la valeur prise par E au dernier instant où H valait vrai. La table ci-dessous illustre l'effet combiné des opérateurs *when* et *current*.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| E | = ( | $e_0$ | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ | ... |
| C | = ( | true | false | true | true | false | false | true | false | ... |
| X = E when C | = ( | $e_0$ | | $e_2$ | $e_3$ | | | $e_6$ | | ... |
| Y = current(X) | = ( | $e_0$ | $e_0$ | $e_2$ | $e_3$ | $e_3$ | $e_3$ | $e_6$ | $e_6$ | ... |

Fig 1.3.3 : Filtrage et projection

### 1.3.2 Modularité et fonctions externes

#### 1.3.2.1 Tuples

Un tuple est une liste d'expressions LUSTRE. Si $e_1, \ldots, e_n$ sont des expressions de types respectifs $T_1, \ldots, T_n$, alors le tuple $(e_1, \ldots, e_n)$ est de type *tuple* $(T_1, \ldots, T_n)$.

#### 1.3.2.2 Réseaux d'opérateurs et nœuds

Malgré la syntaxe textuelle du langage, tout programme LUSTRE peut être vu comme un réseau d'opérateurs connectés par des fils. Par exemple l'équation *X=0 -> pre(X)+1* décrit le réseau représenté par la figure 1.3.2.1.a.

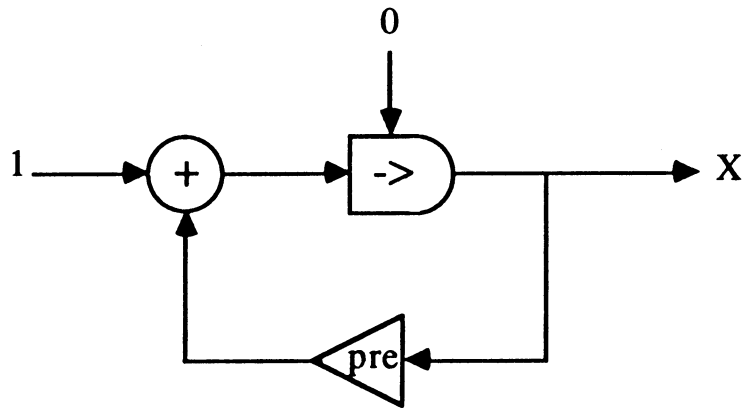



Figure 1.3.2.1.a : réseau d'opérateurs correspondants à l'équation *X = 0 ->pre(X)+1*

Le programmeur peut définir ses propres opérateurs, qui sont appelés des nœuds . Un nœud est un sous-programme LUSTRE. Il opère sur des variables d'entrée, et calcule des variables de sortie et éventuellement des variables locales, au moyen d'un système d'équations. Par exemple, on peut définir un compteur généralisé comme suit :

**node** COUNT (init, incr : **int**; reset : **bool**) **returns** (n : **int**);
**let** -- n prend la valeur init au premier cycle
-- aux autres cycles, sa valeur est init lorsque reset vaut vrai
-- ou est incrémentée de incr lorsque reset vaut faux
n = init -> **if** reset **then** init **else pre**(n) + incr;
**tel**.

et le représenter par le réseau d'opérateurs suivant :

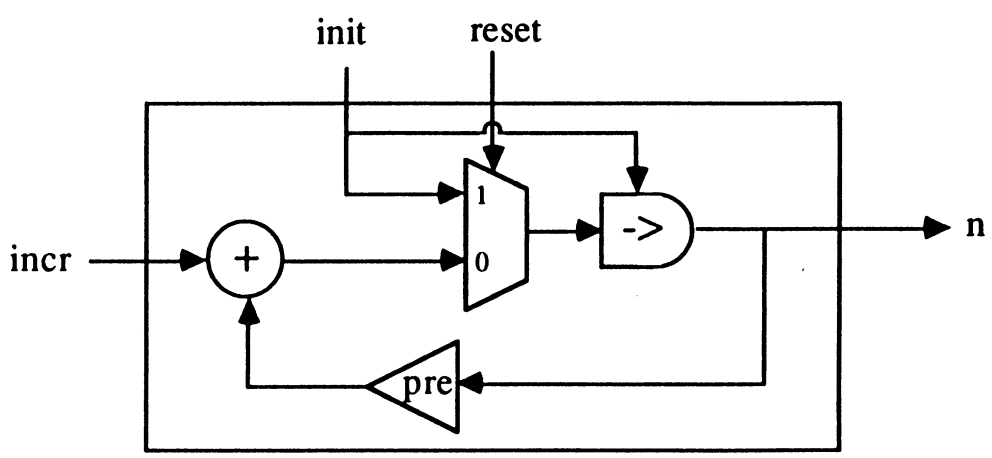



Figure 1.3.2.1.b : réseau d'opérateurs correspondant au nœud COUNT

L'instantiation des nœuds se fait de manière fonctionnelle. Si N est l'identificateur d'un nœud, déclaré avec l'en-tête

**node** N ($i_1$ : $T_1$, $i_2$ : $T_2$, ..., $i_p$ : $T_p$) **returns** ($j_1$ : $Q_1$, $j_2$ : $Q_2$, ..., $j_q$ : $Q_q$)

et si $e_1$,...,$e_p$ sont des expressions de types respectifs $T_1$,..., $T_p$,
alors N($e_1$,...,$e_p$) est une expression de type *tuple* ($Q_1$,..., $Q_q$) dont la $n^{ième}$ valeur est le tuple ($j_{1n}$,...,$j_{qn}$) calculé par le nœud à partir des entrées $e_1$,...,$e_p$.

Pour revenir à l'exemple du compteur, on peut écrire :

pair = COUNT(0,2,**false**);
mod5 = COUNT(0,1, **pre**(mod5=4));

ce qui définit "pair" comme la suite des nombres pairs, et "mod5" comme la suite cyclique des entiers modulo 5.

L'horloge de base d'un nœud est déterminée par l'horloge de ses paramètres d'entrée. Par exemple, l'appel :

COUNT((0,1,**false**) **when** C)

compte chaque fois que C vaut vrai (dans cet exemple, l'opérateur *when* est appliqué au tuple (0,1,false) ; la conditionnelle et les opérateurs sur les suites peuvent être appliqués aux tuples).

Un nœud peut être défini pour des entrées d'horloges différentes. Lorsque l'horloge d'un paramètre n'est pas l'horloge de base, cette horloge doit elle-même être passée en paramètre. Par exemple, l'en-tête de déclaration d'un nœud peut être :

***node N (millisecond : bool; (second : bool) when millisecond) returns*** ...

Ceci signifie que le premier paramètre (dont l'horloge est toujours l'horloge de base du nœud) est l'horloge du second paramètre.

Un nœud peut aussi retourner des résultats d'horloges différentes, à condition que toutes les horloges soient visibles de l'extérieur du nœud.

### 1.3.2.3 Appel de fonctions externes

Le langage LUSTRE, défini au cours des paragraphes précédents (1.1,1.3.3 et 1.3.4), est simple parce qu'il est ciblé. Il sert essentiellement à décrire les aspects temporels des programmes. Par contre, il ne possède pas de mécanismes évolués en ce qui concerne les types de données et les entrées/sorties. De plus, il faut convenir que certains traitements sont plus faciles à écrire dans un

style impératif. Pour ces raisons, il est possible en LUSTRE d'appeler des fonctions externes, écrites dans un autre langage. Ces fonctions sont considérées comme des opérateurs instantanés sur les valeurs, c'est-à-dire, ne réalisant pas de mémorisations internes. On peut aussi définir des types externes, et manipuler des objets de ces types par des fonctions externes.

#### 1.3.2.4 Assertions

Les assertions ont été introduites en LUSTRE (comme en ESTEREL [BMR, 83]) pour exprimer des propriétés connues de l'environnement. Ces propriétés sont utilisées par le compilateur et peuvent avoir une influence sur la taille du code objet produit : savoir, par exemple, que certaines entrées ne sont jamais vraies simultanément, peut permettre la suppression de certains tests dynamiques. On verra comment ces assertions sont utilisées pour la vérification de propriétés.

Il y a deux sortes d'assertions en LUSTRE.

- La première, de syntaxe :

  **assert** *expression_booléenne* ;

  où *expression_booléenne* est une expression LUSTRE quelconque de type booléen, spécifie que l'expression booléenne est invariablement vraie.
- La deuxième spécifie qu'un ensemble d'expressions booléennes sont exclusives, c'est-à-dire que seule l'une d'entre elles peut être vraie à un instant donné. Sa syntaxe est la suivante :

  **assert #**(*liste_exp_bool* );

  où *liste_exp_bool*: ::= *expression_booléenne* | *expression_booléenne* , *liste_exp_bool*..

Nous verrons, sur un exemple, au paragraphe 1.3.3.3.2.2, comment les assertions sont utilisées lors de la génération de code.

### 1.3.3 Compilation

En plus de la vérification syntaxique, le compilateur LUSTRE assure le contrôle de la cohérence des types, l'absence de blocage et la cohérence des horloges.

Dans la phase suivante, le compilateur remplace chaque appel de nœud dans le programme source, par le système d'équations qui lui correspond, afin d'obtenir un programme se réduisant à un simple système d'équations.

Enfin, le compilateur génère le code objet à partir de l'ordonnancement de ces équations. La génération est réalisée de façon simple (code "naïf") ou par synthèse de la structure de contrôle. Le code généré est actuellement une séquence d'instructions C ou Pascal.

Les paragraphes suivants décrivent de manière succinte, les vérifications contextuelles, la mise à plat du programme et la génération de code.

### 1.3.3.1 Vérifications contextuelles

#### 1.3.3.1.1 Vérification de type

Le contrôle des types réalisé par LUSTRE, se situe au niveau :

- des opérateurs : pour chaque opérateur ou fonction est défini le type de ses données d'entrée et de sortie. Les types manipulés dans un programme LUSTRE sont soit prédéfinis : *bool* (booléen), *int* (entier), *real* (réel), soit externes (types C par exemple), soit construits et de forme tuple(...). Certains opérateurs possèdent plusieurs profils : l'addition par exemple, est réalisée sur les entiers : int x int -> int, ou sur les réels : real x real -> real. Le compilateur contrôle que les opérandes de tout nœud, fonction ou opérateur sont en accord avec leur définition.
- des équations : le type déclaré d'une variable doit être le même que celui de l'expression qui la définit (le type d'une expression est déduit du type des variables, des opérateurs, des nœuds et fonctions qui la composent).
- l'interface des nœuds : lors d'un appel de nœud, le type des paramètres effectifs doit correspondre à celui des paramètres formels déclarés dans l'interface du nœud.

En LUSTRE, il n'y a pas de mécanisme de conversion de types.

#### 1.3.3.1.2 Cohérence des horloges

Toute expression LUSTRE possède une horloge (cf § 1.1.3). Pour chaque opérateur, des règles de cohérence entre les horloges de ces opérandes ont été définies afin que le nombre de valeurs du passé nécessaires au calcul d'une sortie soit fini et calculable ( principe de la **Mémoire bornée**). L'une des règles stipule que les opérandes de tout opérateur de base du langage ont forcément la même horloge. Ainsi, il y a un parfait synchronisme dans l'arrivée des valeurs des opérandes. Il n'est donc pas besoin de structures de mémorisation supplémentaires, par rapport à celles nécessaires pour les mémorisations dues aux opérateurs *pre* et *current* , pour réaliser le calcul d'une sortie.

Il faut noter que l'égalité de deux horloges (expressions LUSTRE booléennes) est en fait un problème indécidable. En effet, pour prouver l'égalité de deux horloges, il faudrait pouvoir prouver statiquement des théorèmes portant sur des expressions arithmétiques comme "si x>y alors z=2*u". C'est pourquoi, les opérateurs *when* ne pourront avoir comme opérande droit (horloge) qu'une variable booléenne, et que l'égalité entre horloge sera réduite, pour le

compilateur, à une égalité syntaxique (à une substitution de variables près). Il existe donc des programmes LUSTRE corrects au point de vue des horloges, qui seront rejetés par le compilateur. Néanmoins, grâce au principe de substitution, tout programme LUSTRE correct peut être transformé en un programme accepté par le compilateur (définition d'une variable pour chaque horloge).

Le compilateur LUSTRE vérifie donc pour chaque opérateur utilisé, que les horloges de ses opérandes sont égales. Pour ce faire, il doit être capable de déterminer l'horloge d'une expression LUSTRE quelconque : en fait, il infère cette information à partir de l'horloge des sous-expressions et du type de l'opérateur de plus haut niveau. Les règles d'inférence utilisées ont été énoncées informellement lors de la description de chaque opérateur (voir à ce sujet les paragraphes 1.1.4, 1.1.5 et 1.3.1).

La vérification de la cohérence des horloges est réalisée par pas alternés de vérification et d'inférence. Par exemple, la vérification de la correction vis-à-vis des horloges, de l'équation :

X = (A **or** B) **and** C; où A, B et C sont de même horloge H

est réalisée de la façon suivante :

- vérification pour l'opérateur *or* : ses opérandes sont de même horloge, la sous-expression (A or B) est donc correcte.
- inférence : la sous-expression (A or B) possède la même horloge que ces opérandes, soit H.
- vérification pour l'opérateur *and* : ces deux opérandes sont de même horloge, l'expression (A or B) and C est donc correcte.
- inférence : (A or B) and C est d'horloge H.
- inférence : X est d'horloge H.

#### 1.3.3.1.3 Absence de blocage

Un programme LUSTRE définissant ses sorties comme résultats d'équations sur ses entrées, un blocage dans le calcul des sorties a lieu lorsqu'une variable dépend instantanément d'elle-même. En LUSTRE, à l'inverse de SAGA (cf paragraphe 1.2.5.3), l'absence de blocage est vérifiée lors de la mise à plat du programme. Aucune contrainte n'est imposée au concepteur sur la façon de résoudre un problème de blocage lorsqu'il se pose.

### 1.3.3.2 Mise à plat du programme

On désire obtenir le système d'équations correspondant au programme. Chaque appel de nœud doit alors être remplacé par le système d'équations qui lui correspond.

L'interdiction en LUSTRE d'écrire des programmes récursifs, garantit que toute séquence du graphe de dépendance des appels est finie. Ce graphe définit un ordre $\mathfrak{D}$ sur les nœuds tel qu'on a $n' \mathfrak{D} n$ ssi il existe un chemin sur l'arbre menant de $n'$ à $n$ (le calcul des sorties de $n'$ dépend du calcul des sorties de $n$ ). L'expansion des nœuds (substitution des appels à ces nœuds par les systèmes d'équations correspondants) est réalisée suivant un ordre total compatible avec $\mathfrak{D}$. La mise à plat du programme débute par l'expansion des nœuds n qui ne réalisent dans leur corps, aucun appel à d'autres nœuds : $\{ n / \nexists n', n \mathfrak{D} n' \}$.

Le système d'équations correspondant à un appel de nœud est le corps du nœud (mis à plat) dans lequel toutes les variables en entrée ont été instantiées par les paramètres effectifs utilisés au moment de l'appel.

### 1.3.3.3 Génération de code

#### 1.3.3.3.1 Programme séquentiel simple

On désire obtenir un programme séquentiel écrit dans un langage algorithmique "standard". dont l'exécution correspond à un cycle du programme LUSTRE. La séquence des appels de ce programme séquentiel est synchronisée sur l'horloge de base du programme LUSTRE.

La correspondance "programme lustre"-"programme séquentiel" est assez naturelle. Pour cela, il faut :

- ordonner les équations LUSTRE de manière à ce qu'aucune variable ne soit utilisée avant sa définition,
- remplacer les symboles d'égalité = par des symboles d'affectation (qu'on note := ),
- remplacer les opérateurs spécifiques à LUSTRE par des constructions algorithmiques, par exemple *e->e'* devient *if init then e else e'*, où init est une nouvelle variable qui doit être initialisée à vrai avant l'appel du programme séquentiel et qui est ensuite mise à faux dans le corps de ce programme à la fin de l'appel.
- définir la mémoire locale en introduisant une variable $P_e$ pour chaque expression *pre e*. A la fin de chaque appel, cette variable devra être mise à jour par l'affectation : $P_e := e$.

Nous allons illustrer la génération de code en Pascal par un exemple issu de la gestion des alarmes sur l'application SIREX développée à MERLIN GERIN. Il s'agit de déclencher une alarme lorsque le flux de neutrons dans le cœur du réacteur dépasse un seuil : s1, et de maintenir cette alarme tant que le flux n'est pas redescendu au dessous d'un autre seuil : s2, inférieur au premier.

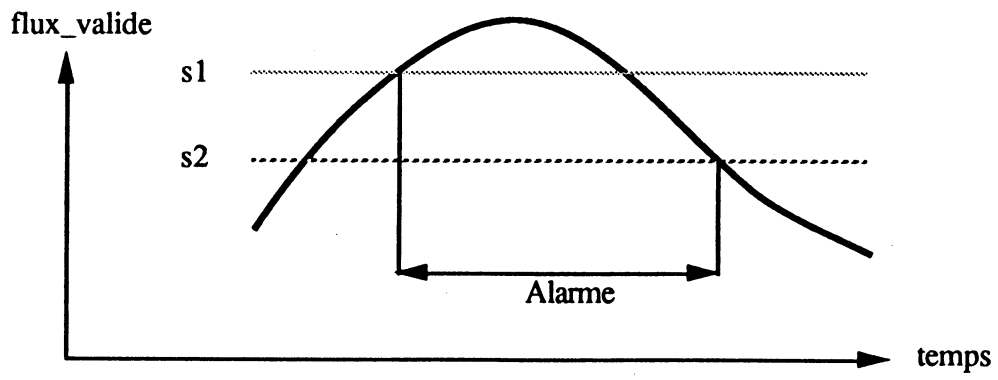



Figure 1.3.3.2.2.1.a : Gestion d'une alarme

Le programme LUSTRE correspondant à notre exemple est le suivant :

```
node ALARME (flux_valide, s1, s2 : real) returns (alarme_s1 : bool);
var montant, descendant : bool;
let
   montant = flux_valide > s1;
   descendant = flux_valide < s2;
   alarme_s1 = AUTOMATE ( montant, montant, descendant);
tel.

node AUTOMATE (état_init, montant, descendant : bool)
returns (état : bool);
let
   état = état_init -> if montant and not pre (état)
                       then true
                       else if descendant and pre (état)
                            then false
                            else pre (état);
tel.
```

La première opération consiste à "expanser" les appels de nœuds pour obtenir un programme plat. Pour cela, on doit en particulier créer des variables intermédiaires et ramener les constantes des nœuds expansés à leurs horloges effectives (ici la seule horloge est l'horloge de base).

```
montant = flux_valide > s1;
descendant = flux_valide < s2;
alarme_s1 = a;
a = montant -> if montant and not Pa
                then true
                else if descendant and Pa
                        then false
                        else Pa;
Pa= pre (a);
```

Pour obtenir un programme séquentiel, il faut donc ordonner les équations, introduire une variable "init_base" qui indique si on est dans l'instant inital de l'horloge "base". On introduit aussi une variable Pa pour mémoriser la valeur de la variable a.

```
procedure ALARME (flux_valide, s1, s2 : real; ref alarme_s1 : bool);
var
   a, Pa : bool;
   init_base : bool := true;
begin
   montant := flux_valide > s1;
   descendant := flux_valide < s2;
   a := if init_base
        then montant
        else if (montant and not Pa)
                then true
                else if (descendant and Pa)
                        then false
                        else Pa;
   alarme_s1 := a;
   Pa := pre (a);
   init_base := false;
end;
```

#### 1.3.3.3.2 Synthèse de la structure de contrôle

##### 1.3.3.3.2.1 Principe

Il est donc assez simple d'obtenir un code séquentiel "naïf" pour un programme LUSTRE. Cependant ce code n'est pas très efficace puisqu'on teste systématiquement si chaque instant est l'instant initial (ce qui correspond dans le code au test : if init_base). Le fonctionnement d'un tel code est schématisé comme suit :

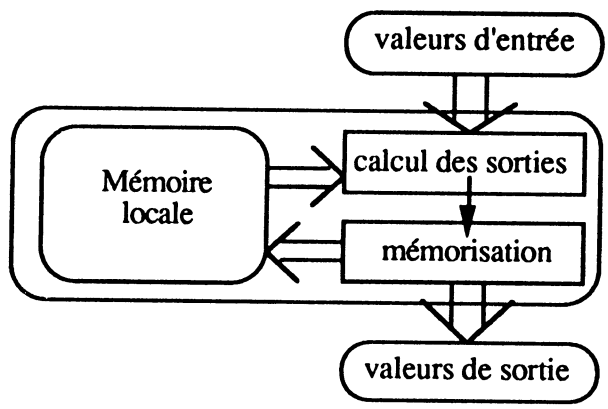



Figure 1.3.3.2.2.1.a : Le fonctionnement naïf d'un nœud LUSTRE.

La mémoire locale comporte toutes les variables correspondant aux expressions *pre* du nœud LUSTRE et aussi les variables init. Dans cette mémoire, on s'intéresse plus particulièrement aux variables booléennes qu'on appelle variables d'état. En effet cette mémoire booléenne a un nombre fini d'états possibles. L'idée est donc d'associer à chacun de ces états un programme simplifié (où toutes les variables d'état sont remplacées par leur valeur). La phase de mémorisation des variables d'état est alors remplacée par la détermination de l'état suivant, auquel sera associé un autre programme simplifié. Lors de la construction de l'automate, le compilateur prend en compte les assertions en ne créant pas les états et les chemins où celles-ci sont fausses.

##### 1.3.3.3.2.2 Exemple

Pour illustrer la génération d'automate dont tout le processus est décrit dans [CPHP,87], nous reprenons l'exemple du programme ALARME.

Les variables d'état de ce programme sont à priori : la variable Pa, et la variable implicite : init_base. Pour générer l'automate, on part de l'état initial $q_0$ où le couple (init_base, Pa) = (true,nil).

Pour ces valeurs, le programme simplifié s'écrit :

```
montant = flux_valide > s1;
descendant = flux_valide < s2;
alarme_s1 = a;
a = montant ;
Pa= pre (a);
```

A partir de $q_0$, on peut accéder, suivant la valeur des entrées aux états $q_1$ : (init_base, Pa) = (false, true) et $q_2$ : (init_base, Pa) = (false, false). Les programmes correspondant à $q_1$ et $q_2$ sont :

```
q1 : montant = flux_valide > s1;
     descendant = flux_valide < s2;
     alarme_s1 = a;
     a = if descendant
           then false
           else true;
     Pa = true -> pre (a);
```

```
q2 : montant = flux_valide > s1;
     descendant = flux_valide < s2;
     alarme_s1 = a;
     a = if montant
           then true
           else false;
     Pa = false -> pre (a);
```

L'automate correspondant est représenté sur la figure 1.3.3.2.2.2.a. Nous appelons **conditions** les expressions dont le compilateur ne peut évaluer la valeur (comparaison entière ou réelle, appel de fonction externe, entrée booléenne) et dont les transitions de l'automate dépendent. Les conditions pour notre exemple sont les variables montant et descendant. Sur la figure, les transitions sont étiquetées par leurs conditions et par le code séquentiel à exécuter (s'il y en a).

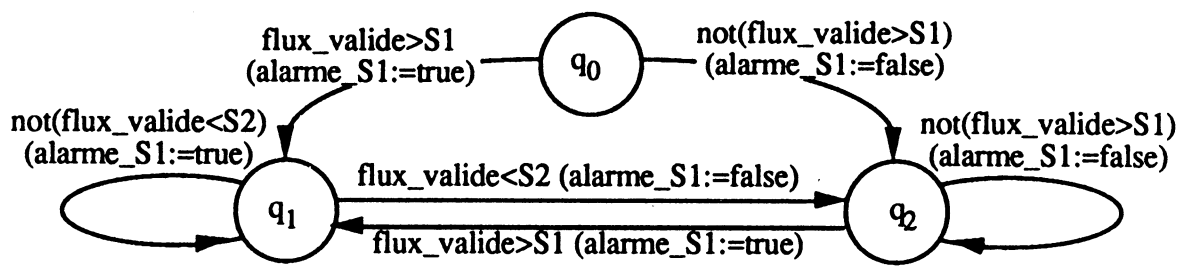



Figure 1.3.3.2.2.2.a : Automate généré pour le programme ALARME.

En ajoutant à la gestion des alarmes décrite par le nœud ALARME ci-dessus, une assertion spécifiant que le flux_valide est toujours inférieur à s1 au premier cycle, sous la forme :

```
assert ( not (flux_valide > s1) -> true);
```

l'automate obtenu devient alors celui de la figure 1.3.3.2.2.2.b.

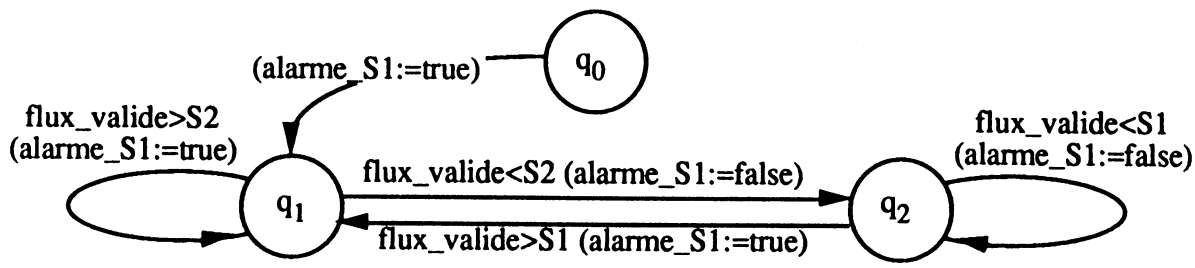



Figure 1.3.3.2.2.2.b : Automate du programme ALARME avec assertion

#### 1.3.3.3.3 Absence de valeur indéfinie

Toute expression LUSTRE doit être définie à tous les cycles de son horloge. C'est ce que vérifie le compilateur lors de la génération de l'automate en calculant au moment de la création de chaque état, toutes les expressions dont l'horloge est à "true". Les valeurs indéfinies peuvent apparaître lors d'une mauvaise utilisation des opérateurs *pre* et *current*.

## 1.4 SÉMANTIQUE DE SAGA EN LUSTRE

Tout programme SAGA (application entièrement affinée) est traduisible en LUSTRE de façon automatique. Nous allons donner dans ce paragraphe les correspondants en LUSTRE des principaux concepts de SAGA : variables, fonctions, mémorisation et initialisation, conditions d'activation, qui permettront de réaliser de telles traductions (pour plus de détails voir [GC, 89]).

### 1.4.1 Variables et fonctions

Une donnée SAGA non affinable est traduisible en une variable LUSTRE. Son type, s'il est différent des types booléen, entier ou réel, qui sont les seuls types prédéfinis en LUSTRE, sera importé. Une donnée affinable ne pourra être prise en compte en LUSTRE, que sous la forme d'un tuple composé des données de l'affinement de base de celle-ci. La traduction d'un programme en cours de conception dont certaines données ne sont pas affinées, semble donc problématique pour l'instant.

Les fonctions SAGA seront naturellement décrites soit par des nœuds LUSTRE, soit par des fonctions externes, selon qu'elles sont affinées en SAGA, ou implémentées dans un autre

langage. Chaque vue SAGA sera donc décrite par un nœud LUSTRE (puisqu'elle correspond à un affinement de fonction) et toute fonction (boîte) apparaissant dans une vue correspond à un appel de nœud ou de fonction externe.

Un traducteur automatique de SAGA vers LUSTRE est en cours d'élaboration à Merlin Gerin/SES. Voici par exemple, la traduction d'une vue, telle que cet outil pourrait la réaliser.

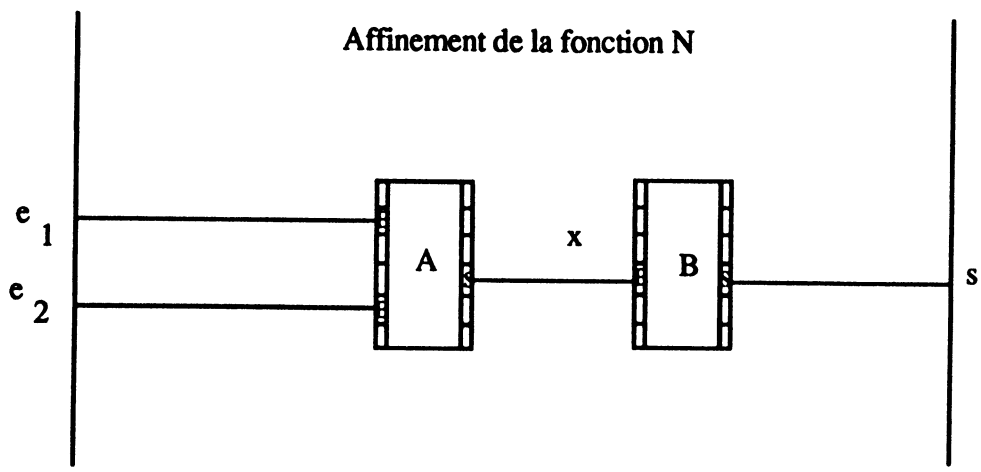



Figure 1.4.1 : Une vue SAGA

Si A est une fonction affinable, et B une fonction non affinable, la traduction en LUSTRE de la figure 1.4.1 sera :

```
node A (e1 : type_e1; e2 : type_e2) returns (x : type_x);
        < corps de A >
function B (x : type_x) returns (s : type_s);
node N (e1 : type_e1; e2 : type_e2) returns (s : type_s);
var x : type_x;
let
        s = B(x);
        x = A (e1, e2);
tel;
```

Tous les nœuds et fonctions sont déclarés au même niveau d'imbrication pour respecter la sémantique de SAGA dans laquelle les fonctions ont une portée globale.

### 1.4.2 Mémorisations et initialisations

L'opérateur *INIT* de SAGA est traduit en LUSTRE par l'opérateur ->, et l'opérateur *MEMO* par l'opérateur *pre* :

INIT (X, Y) = X->Y

MEMO (X) = pre (X)

### 1.4.3 Comportement temporel

Du point de vue temporel, l'hypothèse de base en SAGA est que toute donnée apparaissant dans une vue est définie à chaque cycle de la fonction représentée par la vue. Toute donnée produite par une fonction F avec condition d'activation conserve sa valeur précédente aux cycles où la condition d'activation est fausse et où F n'est pas activée.

On peut alors faire le lien suivant avec les horloges LUSTRE :
L'horloge du nœud représentant une fonction SAGA est la condition d'activation de la fonction, si une telle condition est présente. En l'absence de condition d'activation, l'horloge du nœud est l'horloge de base du nœud englobant.
Aussi, dans la traduction en LUSTRE, faut-il convertir les entrées d'un nœud selon son horloge (filtrage), ainsi que ses sorties selon l'horloge du nœud englobant (projection). La projection prend en compte les valeurs par défaut des sorties lorsque nécessaire.

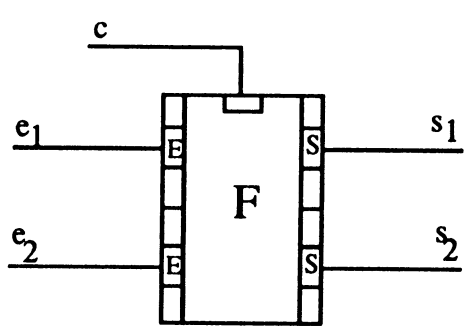



Figure 1.4.3.a : Fonction et condition d'activation

Ce qui donne pour la vue représentée par la Figure 1.4.3.a, avec défaut_s1 et défaut_s2 les deux valeurs données par défaut pour s1 et s2, dans le cas d'une condition d'activation fausse aux premiers cycles :

(s1, s2) = *if* c *then current* (F(e1 *when* c, e2 *when* c))
*else* (défaut_s1, défaut_s2) -> (*pre* s1, *pre* s2)

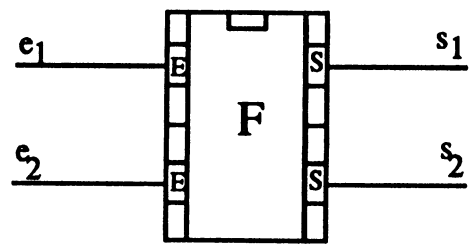



Figure 1.4.3.b : Fonction sans condition d'activation

En l'absence de condition d'activation (cf. Fig.1.3.4.b), la traduction devient simplement (s1, s2) = F ( e1, e2 ).

### 1.4.4 Opérateurs de base (sur les suites)

Les opérateurs de base sont identiques en SAGA et en LUSTRE. Ce sont :

- Les fonctions de calcul arithmétique ( +, -, ... )
- Les fonctions de calcul logique ( *et*, *ou*, *non*, ... en SAGA et *and*, *or*, *not*, ... en LUSTRE)
- Les fonctions de comparaisons ( >, <, =, ... )
- Le sélecteur de SAGA est traduit par le *if...then...else...* de LUSTRE.

### 1.4.5 Exemple de traduction

Nous présentons maintenant un exemple simple, qui nous permet de mieux appréhender la traduction de SAGA en LUSTRE. La fonction décrite est souvent utilisée pour le calcul de booléens :

L'automate à deux états construit une variable booléenne à partir de deux entrées qui peuvent être considérées comme ses fronts montant et descendant (interrupteur) : la variable conserve sa valeur si les entrées "montant" et "descendant" sont toutes deux fausses, la variable passe à vrai lorsque "montant" est vrai et à faux lorsque "descendant" est vrai. Pour traiter le cas où "montant" et "descendant" sont représentés par la même condition (interrupteur par bouton poussoir), il convient de prendre en compte l'état précédent de la variable: "montant" n'occasionnera d'effet que si la variable est fausse et "descendant" que si la variable est vraie. La variable prend au premier cycle la valeur : état_init.

Les figures 1.4.6.a et 1.4.6.b représentent la fonction Automate en SAGA et son affinement.

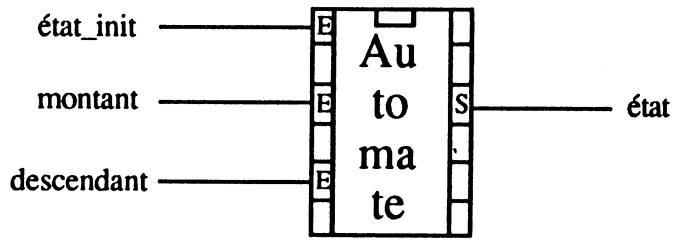



Figure 1.4.6.a : La fonction "Automate"

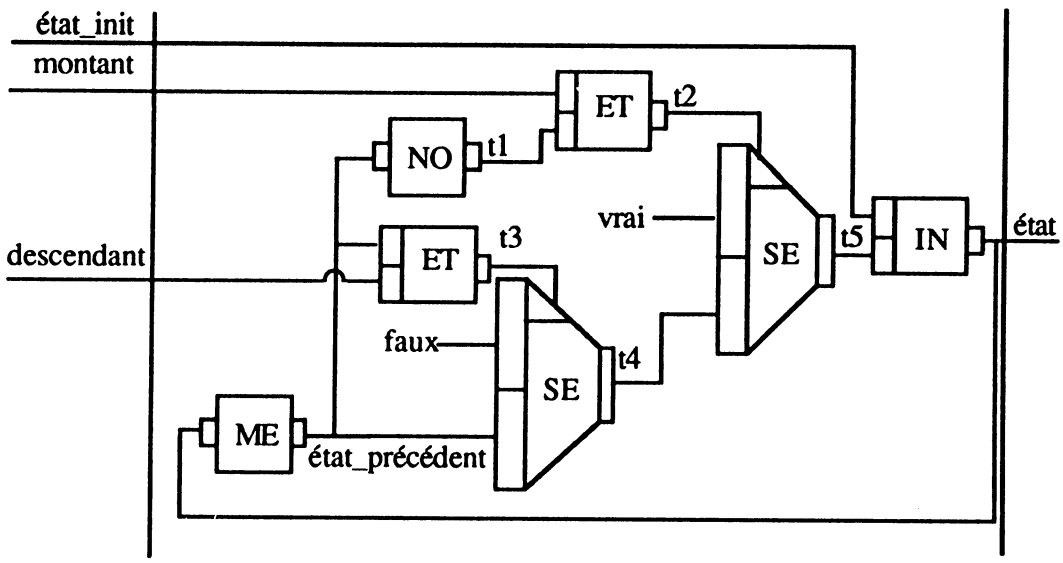



Figure 1.4.6.b : Affinement de la fonction "Automate"

Le programme LUSTRE correspondant, selon une version "traduction automatique", est le suivant :

```
node AUTOMATE (état_init, montant, descendant : bool) returns (état : bool);
var t1, t2, t3, t4, t5 : bool;
let
        état = état_init -> t5;
        t5 = if t2 then true else t4;
        t4 = if t3 then false else état_précédent;
        t3 = état_précédent and descendant;
        t2 = montant and t1;
        t1 = not état_précédent;
        état_précédent = pre(état);
tel.
```

Une version améliorée (lecture facilitée par la réduction du nombre de variables et d'équations) de la traduction est la suivante :

```
node AUTOMATE (état_init, montant, descendant : bool)
        returns (état : bool);
let
        état = état_init  -> if montant and not pre(état)
                                then true
                                else if pre(état) and descendant
                                        then false
                                        else pre(état);
tel.
```

## 1.5 CONCLUSION

La base de la sémantique des langages SAGA et LUSTRE est donc le flot de données muni d'une interprétation synchrone. Ces langages ont tous deux été définis pour le développement de logiciels d'automatisme temps réel.

Leurs différences essentielles portent sur :

- leur représentation : graphique pour SAGA et textuelle pour LUSTRE,
- la manipulation des horloges : condition d'activation en SAGA et opérateurs *when* et *current* en LUSTRE,
- leur fonction dans le cycle de vie des logiciels. Le langage SAGA est un langage de conception; il est associé à une méthode et à un atelier mettant en œuvre cette méthode. LUSTRE est, quant à lui, un langage de programmation, mais dont on verra, dans cette thèse, l'emploi en tant que langage de spécification.

Malgré ces différences, nous avons vu que toute application SAGA en fin de phase de conception peut être traduite en LUSTRE, et qu'un traducteur automatique est en cours de création. Ainsi, nous pourrons par la suite appliquer à tout programme SAGA les outils et les méthodes définies pour des programmes LUSTRE. Notre travail de formalisation sur la vérification de propriétés ne portera dans les chapitres suivants que sur des programmes LUSTRE. Le but étant de l'appliquer, à terme, à des programmes SAGA.

Remarquons que la traduction de LUSTRE en SAGA n'est pas possible car le pouvoir d'expression de LUSTRE est plus grand que celui de SAGA (grâce à sa définition des sous-horloges).

# 2 Sémantique formelle et modèles des programmes LUSTRE

La vérification de programmes consiste à comparer un programme avec ses spécifications. Les formalismes les plus utilisés pour décrire des systèmes réactifs sont ceux auxquels on peut associer une sémantique basée sur les systèmes de transitions. Deux approches principales de la vérification de tels systèmes peuvent être distinguées, suivant le formalisme utilisé pour exprimer les spécifications.

Une des approches consiste à fournir une description des spécifications interprétable sous forme de machine abstraite, présentant ainsi le comportement attendu du programme. Dans ce cas, les spécifications, dites **comportementales**, et le programme peuvent être considérés comme des systèmes de transitions. Ces systèmes sont définis à partir de la sémantique opérationnelle des langages de spécification et de programmation.

La comparaison entre un programme et ses spécifications, est alors réalisée en définissant une relation d'équivalence, ou un préordre d'implémentation, sur les systèmes de transitions, qui induit une relation du même type entre le programme et les spécifications.

Des exemples typiques de spécifications comportementales dans le domaine des systèmes réactifs, sont les algèbres de processus (CCS de [Mil,80], ACP de [BK,85]). Une algèbre de processus est une algèbre de termes munie d'une relation d'équivalence, dont les termes peuvent être interprétés comme des systèmes de transitions.

L'autre approche pour la vérification de programmes, utilise pour les spécifications un formalisme **basé sur les logiques**. A cette fin, une relation $\models$ est définie sur PxF, où P est le langage de description de programme et F la logique constituant le langage de formules. L'assertion $p \models f$ signifie que le programme p satisfait la propriété décrite par f (p est un modèle de f). Des exemples typiques de cette approche sont les logiques de programme ($CTL^*$ de [EH,86], CL de [QS,83]). La vérification qu'un programme satisfait une formule est réalisée en évaluant la formule sur un graphe d'états modélisant le comportement du programme.

Dans les approches considérant un modèle du programme, un langage de description du modèle est utilisé, qui diffère suivant l'approche et les systèmes envisagés. Pour réaliser la vérification de programmes LUSTRE, et donc SAGA, nous avons alors défini différents modèles

de programmes LUSTRE ; nous verrons par la suite (chapitre 4 et 5) comment chacun de ces modèles peut être utilisé pour la vérification.

Le premier modèle que nous présentons est l'**arbre des exécutions** d'un programme LUSTRE. Ce modèle est infini et ne peut donc être utilisé pour une vérification automatique de programmes LUSTRE. C'est pourquoi, nous avons défini deux autres modèles dérivés du premier: le **graphe d'états** et l'**automate de contrôle**. Sous certaines conditions, le graphe d'états peut être fini. L'automate est toujours fini.

Ces trois modèles constituent le cœur du chapitre. Leur présentation nécessite un exposé succint de la sémantique formelle de LUSTRE.

## 2.1 SÉMANTIQUE NATURELLE DE LUSTRE

La sémantique naturelle de LUSTRE [Pla,88] a été donnée par des règles d'inférence structurelles qui déterminent si une suite de mémoires (notées $\sigma$), présentées ci-dessous, est compatible avec un ensemble d'équations LUSTRE.

**Définitions:**

- ID étant l'ensemble des identificateurs de LUSTRE,
  VAL étant le domaine des valeurs associées aux types LUSTRE,
  une **mémoire** $\sigma$ est une fonction partielle de ID dans VAL.
  L'ensemble des mémoires est noté $\Sigma$.
- Une suite de mémoires s'appelle une **histoire**, dénotée $\eta$.
- L'histoire initiale est l'histoire vide, notée <>.
- Le prédicat $\eta.\sigma \vdash$ eqs signifie que, si l'histoire jusqu'à l'instant courant est $\eta$, et si la mémoire à cet instant est $\sigma$, l'histoire $\eta.\sigma$ est cohérente avec le système d'équations eqs. C'est-à-dire, qu' à chaque instant de cette histoire, les valeurs des variables dans la mémoire courante sont les mêmes que celles données par les équations.

**Règles de la sémantique naturelle:**

Dans la suite, il est fait mention explicite des horloges de toutes les expressions apparaissant dans les équations : *e on h* signifie que l'expression e n'est considérée qu'au moment où l'horloge h est vraie.

1. L'histoire initiale est compatible avec toutes les équations:

$$<> \vdash \text{eqs} \qquad (1)$$

2. Aux autres instants, l'histoire est cohérente avec chaque équation:

$$\frac{\eta.\sigma \vdash \text{eq}, \quad \eta.\sigma \vdash \text{eqs}}{\eta.\sigma \vdash \text{eq;eqs}} \qquad (2)$$

3. Equations: Pour chaque équation "id on h = te", la valeur k rendue par l'expression te (exprimé par l'intermédiaire du prédicat $\eta.\sigma \vdash$ te: k défini en 4) doit correspondre à celle de la variable id, au moment où h vaut vrai:

$$\frac{\eta \vdash \text{h: tt}, \ \eta.\sigma \vdash \text{te: k}, \ \sigma(\text{id}) = \text{k}}{\eta.\sigma \vdash \text{id on h = te}} \qquad (3)$$

4. Evaluation des expressions :
On définit le prédicat $\eta \vdash$ te: k, qui signifie que si l'histoire jusqu'à l'instant courant compris, est $\eta$, alors l'expression te est évaluée à k dans cette histoire.

L'horloge de chaque expression étant mentionnée explicitement, nous ne donnons pas la sémantique de l'opérateur *when* qui est devenu redondant. Pour simplifier l'expression, la mention *on h* sera certaines fois omise.

Horloge de base :

$$\eta.\sigma \vdash \text{base: tt} \qquad (4.1)$$

Constantes:

$$\frac{\eta \vdash \text{h: tt}}{\eta.\sigma \vdash \text{k on h: k}} \qquad (4.2)$$

Variables:

$$\frac{\eta \vdash h: tt,\ \sigma(id) = k}{\eta.\sigma \vdash id\ on\ h : k} \quad (4.3)$$

Opérateurs:

$$\frac{\exists\ (1 \leq i \leq p)\ \eta.\sigma \vdash te_i: k_i = nil}{\eta.\sigma \vdash op(te_1, ..., te_p): nil} \quad (4.4)$$

$$\frac{\forall\ (1 \leq i \leq p)\ \eta.\sigma \vdash te_i: k_i \neq nil}{\eta.\sigma \vdash op(te_1, ..., te_p): op\ (k_1, ..., k_p)} \quad (4.5)$$

Opérateur if...then...else :

$$\frac{\eta.\sigma \vdash te: nil}{\eta.\sigma \vdash if\ te\ then\ te_1\ else\ te_2: nil} \quad (4.6)$$

$$\frac{\eta.\sigma \vdash te: tt,\ \eta.\sigma \vdash te_1: k_1}{\eta.\sigma \vdash if\ te\ then\ te_1\ else\ te_2: k_1} \quad (4.7)$$

$$\frac{\eta.\sigma \vdash te: ff,\ \eta.\sigma \vdash te_2: k_2}{\eta.\sigma \vdash if\ te\ then\ te_1\ else\ te_2: k_2} \quad (4.8)$$

Opérateur pre :

L'opérateur pre est traduit en current

$$\frac{\eta \vdash current(id)\ on\ h : k}{\eta.\sigma \vdash pre(id)\ on\ h : k} \quad (4.9)$$

Opérateur current :

$$<> \vdash (\text{current}(id)) \text{ on } h : nil \tag{4.10}$$

$$\frac{\eta.\sigma \vdash h : tt \quad \sigma(id) = k}{\eta.\sigma \vdash (\text{current}(id)) \text{ on } h : k} \tag{4.11}$$

$$\frac{\eta.\sigma \vdash h : k \neq tt \quad \eta \vdash (\text{current}(id)) \text{ on } h : k'}{\eta.\sigma \vdash (\text{current}(id)) \text{ on } h : k'} \tag{4.12}$$

Opérateur ->:

On transforme cet opérateur en définissant une variable fictive:

live = true,

et en s'aidant de l'opérateur unaire is_nil défini par:

$$\frac{\eta.\sigma \vdash te: nil}{\eta.\sigma \vdash is\_nil\ (te): tt} \tag{4.13}$$

$$\frac{\eta.\sigma \vdash te: k \neq nil}{\eta.\sigma \vdash is\_nil\ te: ff} \tag{4.14}$$

On obtient alors l'équivalence: $(e_1 \text{ -> } e_2)$ on h $\Leftrightarrow$ if is_nil (pre(live) on h) then $e_1$ else $e_2$

## 2.2 MODELES

Dans l'approche comportementale comme dans l'approche basée sur des modèles, les systèmes considérés sont en général composés de processus parallèles et communicants. Les processus de base sont des programmes séquentiels. La communication est souvent établie via des canaux ou des variables partagées. Le système global est vu comme l'entrelacement des actions que peuvent réaliser chacun de ces processus.

Voir un programme flots de données synchrone comme un système de processus communicants nécessite une transformation du programme, notamment par l'introduction d'actions fictives et à cause de l'expression asynchrone de la synchronisation des processus (ceci sera illustré

au chapitre 4 quand nous examinerons la possibilité de vérifier un programme LUSTRE avec le logiciel XESAR [RRSV,87b]). C'est pourquoi, nous avons défini des modèles propres aux programmes LUSTRE.

### 2.2.1 Arbre des exécutions

Soit ID l'ensemble des identificateurs LUSTRE et VAL le domaine des valeurs associées aux types LUSTRE (la valeur indéfinie : nil, appartient à VAL), on rappelle que:

- Une **mémoire** $\sigma$ est une fonction partielle de ID -> VAL. On note $\Sigma$ l'ensemble des mémoires. La mémoire vide $\sigma_\emptyset$ associe à toute variable la valeur nil : $\forall v \in ID, \sigma_\emptyset(v) = nil$.
- Une **histoire** $\eta$ est une séquence éventuellement infinie de mémoires.

Soit un programme Pr, on définit les ensembles $H_n$ des histoires de Pr de longueur n par :

- $H_0 = \{< >\}$ où $< >$ est l'histoire vide, c'est-à-dire l'histoire réduite à $\sigma_\emptyset$.
  (rappelons que, selon l'axiome 1 de la sémantique naturelle, $< >$ est la seule histoire compatible avec Pr avant l'exécution de celui-ci )
- $H_{n+1} = \{\eta.\sigma_j / \eta \in H_n, \sigma_j \in \Sigma, \eta.\sigma_j \vdash Pr\}$ où $\vdash$ est le prédicat de la sémantique naturelle de LUSTRE (cf paragraphe 2.1).

**Un arbre des exécutions** de Pr est un couple (A, I) où

*A est un arbre*, c'est-à-dire :

un triplet $(Q, q_{racine}, \rightarrow)$ où

- Q est un ensemble dénombrable d'états,
- $q_{racine} \in Q$, est l'état racine de l'arbre
- $\rightarrow$ une relation binaire sur Q ( $\rightarrow \subseteq$ QxQ) dite relation de transition.
  $\forall (q,q') \in \rightarrow$, q' est dit "successeur" de q et on note $q \rightarrow q'$.
  La relation $\rightarrow$ est telle que :
  1) $\forall q \in Q, q \neq q_{racine} \Rightarrow \exists!\ q' \in Q, q' \rightarrow q$
  (tout état de Q est accessible à partir de $q_{racine}$ et n'a qu'un prédécesseur immédiat)
  2) $\nexists\ q \in Q, q \rightarrow q_{racine}$

On appelle sous-séquence d'exécution (ou portion de l'exécution) partant d'un état $q_0$ toute séquence d'états $ss = q_0, q_1, ..., q_i, q_{i+1}, ...$ telle que $\forall i,\ q_i \rightarrow q_{i+1}$.

**Une séquence d'exécution** partant d'un état $q_0$ est une sous-séquence d'exécution maximale, c'est-à-dire: si cette séquence d'états est finie, son dernier élément n'a pas de successeur.

On appelle EX(q) l'ensemble des séquences d'exécution à partir de q. On note $s_k$ le $(k+1)^{ième}$ élément de la séquence s, s'il existe.

*I est une fonction d'interprétation* qui associe une mémoire à tout état de A :
I : Q -> Σ est telle que :

- $I(q_{racine}) = \sigma_\emptyset$
- $\forall s \in EX(q_{racine}), \forall i \in \mathbb{N}, I(s_0).I(s_1)...I(s_i) \in H_i$
- $\forall i \in \mathbb{N}, \forall \eta \in H_i, \exists s \in EX(q_{racine}), \eta = I(s_0).I(s_1)...I(s_i)$

Intuitivement :

- Chaque état correspond à un instant de l'horloge de base de Pr. L'état $q_{racine}$ correspond à l'instant virtuel "0" pour lequel toutes les variables sont indéfinies. Ses états successeurs correspondent aux valeurs possibles des variables au premier instant de déroulement du programme (instant "1").
- Un cycle du programme correspond au passage d'un état dans un autre suivant la relation →.
- Chaque branche de l'arbre (séquence d'exécution à partir de $q_{racine}$) correspond à une exécution possible du programme :
  soit la séquence d'exécution $s=q_{racine},q_1,\ldots,q_n,\ldots$ , soit l'histoire $\eta=\sigma_\emptyset,\sigma_1,\ldots\sigma_n,\ldots$ telle que $\forall i \in \mathbb{N}$, $I(q_i) = \sigma_i$ (on parlera d'histoire associée à une exécution), et soit x une variable sur l'horloge de base, alors $\sigma_1(x),\ldots\sigma_n(x),\ldots$ est la séquence des valeurs de x pour l'exécution s. En particulier, dans l'état $q_n$, la séquence des valeurs passées de x est $\sigma_1(x),\ldots\sigma_n(x)$, et la séquence des valeurs passées de pre(x) est $\sigma_\emptyset(x),\ldots\sigma_{n-1}(x)$.
- Les programmes LUSTRE sont non déterministes par rapport à leurs entrées puisque c'est l'environnement qui détermine les valeurs de celles-ci. Les entrées étant des variables du programme, elles sont définies dans chaque état par l'intermédiaire de la mémoire associée à cet état. Ainsi, un état de l'arbre des exécutions aura autant d'états successeurs que de combinaisons possibles des entrées, compatibles avec les assertions, au prochain instant de l'horloge de base.

A titre d'exemple, soit le programme LUSTRE constitué du seul nœud Front et calculant le front montant de son entrée :

```
node Front (b:bool) returns (edge:bool);
let
   edge = false → (b and not pre(b));
tel.
```

Un arbre des exécutions du programme Front est le suivant (la notation *nil* représente la valeur indéfinie, la notation*tt*, la valeur vrai, et la notation*ff*, la valeur faux):

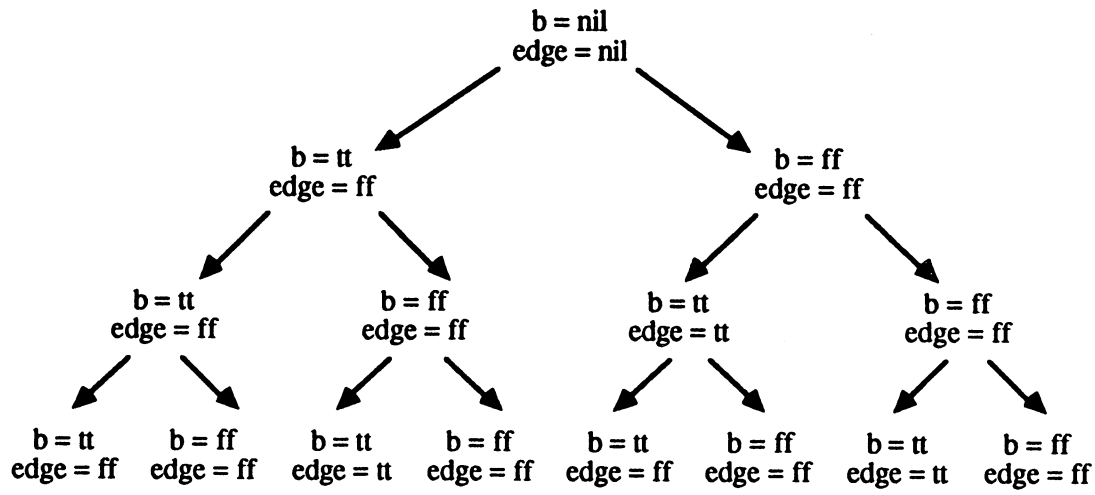



Figure 2.2.1.a : Arbre des exécutions du programme Front

Comparons l'arbre des exécutions d'un programme Pr et l'arbre des exécutions du même programme sous les assertions A (on note $Pr|_A$).

Les assertions sont intégrées au programme Pr suivant la syntaxe exposée au paragraphe 1.3.2.4. Notons qu'un ensemble d'assertions peut toujours être ramené à une seule assertion, puisque l'intersection de deux invariants est un invariant. Dans la suite de cet exposé, A représentera donc la conjonction (*and* en LUSTRE) de toutes les assertions. Notons aussi que l'expression des assertions, comme de toute expression LUSTRE, peut nécessiter la définition de variables intermédiaires à l'aide d'équations, ou la définition de nœuds.

Les assertions étant des parties intégrantes du programme $Pr|_A$, leur valeur apparait dans l'arbre des exécutions. Le domaine de définition des mémoires de l'arbre des exécutions de $Pr|_A$ est alors $D \cup D_A$ où

- D = domaine de définition des mémoires de l'arbre des exécutions de $P_r$,
- $D_A$ = ensemble des variables nécessaires au calcul de A et n'apparaissant pas dans $P_r$ (c.a.d. $D \cap D_A = \emptyset$ ).

Remarquons qu'en général, les assertions portent sur des entrées du programme et que leur description est suffisamment simple pour ne pas nécessiter de définition supplémentaire de variables, auquel cas $D_A=\emptyset$.

Soit l'ensemble $Q|_A \subseteq Q$, où Q est l'ensemble des états de l'arbre de Pr, défini de façon inductive par :

$\forall q \in Q$, $q \in Q|_A$ ssi q est dans l'un des deux cas suivants :

1) $q \not\models A$

2) $\forall q' \in Q$, $q' \rightarrow q \Rightarrow q' \in Q|_A$

L'arbre de Pr $|_A$ est construit à partir de l'arbre de Pr en retranchant :

1) à Q, tous les états de $Q|_A$

2) à $\rightarrow$, tous les couples (q, q') tels que $q' \in Q|_A$

Soit, par exemple, l'assertion A = (b = false $\rightarrow$ not pre(b)) dans $Front|_A$. L'arbre des exécutions de $Front|_A$ est le suivant :

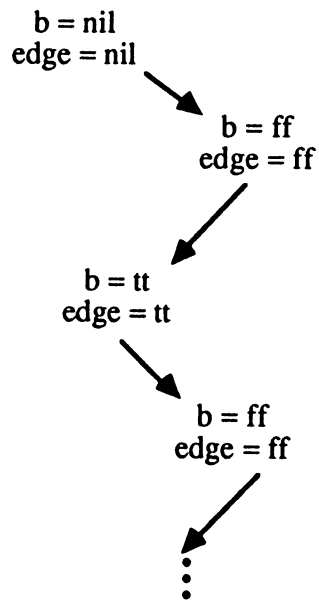



Figure 2.2.1.b : Arbre des exécutions de $Front|_A$

### 2.2.2 Graphe d'états LUSTRE

#### 2.2.2.1 Principe du repliage de l'arbre des exécutions en un graphe d'états

Le graphe d'états est un modèle obtenu par repliage de l'arbre des exécutions du programme. Ce repliage est réalisé suivant une bisimulation définie sur l'arbre. Cette bisimulation s'appuie uniquement sur l'arbre et ne met en jeu aucun critère d'observation (bisimulation forte [Mil,83]). Parmi les bisimulations répondant à la définition que nous allons donner, on choisira la

plus grande afin d'obtenir un nombre minimal d'états (ce qui correspond aux plus larges classes d'équivalence).

Une relation $\rho$ sur QxQ est une bisimulation forte ssi

$\forall\, q_1,q_2 \in Q$

$\rho(q_1,q_2)$ ssi $\forall q'_1 \in Q,\ q_1 \rightarrow q'_1 \Rightarrow \exists q'_2 \in Q,\ q_2 \rightarrow q'_2$ et $\rho(q'_1,q'_2)$

et $\forall q'_2 \in Q,\ q_2 \rightarrow q'_2 \Rightarrow \exists q'_1 \in Q,\ q_1 \rightarrow q'_1$ et $\rho(q'_1,q'_2)$

Soit une relation $\rho$ sur QxQ. On définit la fonctionnelle $\mathfrak{F}$ par

$\mathfrak{F}(\rho) = \{ (q_1,q_2) /$ $\forall q'_1 \in Q,\ q_1 \rightarrow q'_1 \Rightarrow \exists q'_2 \in Q,\ q_2 \rightarrow q'_2$ et $\rho(q'_1,q'_2)$

et $\forall q'_2 \in Q,\ q_2 \rightarrow q'_2 \Rightarrow \exists q'_1 \in Q,\ q_1 \rightarrow q'_1$ et $\rho(q'_1,q'_2)$ $\}$

alors $\rho$ est une bissimulation ssi $\rho \subseteq \mathfrak{F}(\rho)$.

$\mathfrak{F}$ est monotone sur le treillis des relations binaires muni de l'inclusion. Il existe donc un plus grand point fixe pour $\mathfrak{F}$, appelé plus grande bisimulation :

$\sqcup\mathfrak{F} = \cup\{ \rho \mid \rho \subseteq \mathfrak{F}(\rho) \}$

Soit la relation $\approx$ sur QxQ définie par

$\forall\, q_1,q_2 \in Q,\ q_1 \approx q_2$ ssi $I(q_1) = I(q_2)$

On définit la fonctionnelle $\mathfrak{G}$ par

$\mathfrak{G}(\rho) = \{ (q_1,q_2) \in \mathfrak{F}(\rho) \ /\ q_1 \approx q_2 \}$

alors $\rho$ est une bisimulation, contenue dans $\approx$, ssi $\rho \subseteq \mathfrak{G}(\rho)$.

$\mathfrak{G}$ est monotone sur le treillis des relations binaires muni de l'inclusion, grâce à la monotonie de $\mathfrak{F}$. $\mathfrak{G}$ admet donc un plus grand point fixe, appelé plus grande bisimulation contenue dans $\approx$ et noté $\sim$ :

$\sim = \cup \{ \rho \ /\ \rho \subseteq \mathfrak{G}(\rho) \}$

$\sim$ est une relation d'équivalence.

Le repliage de l'arbre des exécutions AE=((Q,$q_{racine}$,$\rightarrow$), I) est réalisé suivant $\sim$. Le graphe d'états G=($\underline{Q}$, $\underline{q}_{racine}$, $\rightarrow$) obtenu est tel que :

- $\underline{Q} = Q/\sim$ Les états de G sont des ensembles d'états de Q.
- $q_{racine} \in \underline{q}_{racine}$
- $\forall\, \underline{q}_1,\underline{q}_2 \in \underline{Q},\ \underline{q}_1 \rightarrow \underline{q}_2$ ssi $\exists q_1 \in \underline{q}_1\ ,\ \exists q_2 \in \underline{q}_2\ ,\ q_1 \rightarrow q_2$

Chaque état de $\underline{Q}$ représente une classe d'équivalence. La mémoire qui lui est associée est la mémoire commune des états de Q dont il représente la classe.

L'interprétation J associée à G est telle que

$\forall \underline{q} \in \underline{Q}$ , $J(\underline{q})=I(q)$, $\forall q \in \underline{q}$

### 2.2.2.2 Repliage

Nous allons exhiber maintenant une relation $\mathfrak{R}$ qui est une bisimulation calculable contenue dans $\approx$ et qui sera dans la pratique la source du repliage de l'arbre en un graphe d'états (de la même façon que ci-dessus). $\mathfrak{R}$ est en général différente de la plus grande bisimulation $\sim$.

Tout d'abord, nous définissons de nouveaux arbres des exécutions, modèles de programmes LUSTRE, par extension du domaine de définition des mémoires associées à chaque état. Puis nous définissons la relation $\mathfrak{R}$ sur les états de cet arbre. Enfin, nous expliquons pourquoi $\mathfrak{R}$ est une bisimulation contenue dans $\approx$.

#### 2.2.2.2.1 Extension des mémoires

Du fait de la définition de la fonction d'interprétation, on a :

$\forall s \in EX(q_{racine}), \forall i \in \mathbb{N}, I(s_0)...I(s_i) \vdash Pr$

Ainsi, en chaque état de l'arbre des exécutions d'un programme Pr, l'histoire depuis l'état racine permet d'évaluer toutes les expressions de Pr.

En particulier, soit EL = pre(E) ou EL = current(E) (où E est une expression), une expression apparaissant dans Pr, alors EL est évaluable dans tout état.

D'après les principes de définition et de substitution (cf. paragraphe 1.1.4), la sémantique du programme Pr n'est pas modifiée lorsqu'on définit dans ce programme une nouvelle variable v par l'équation v = EL, et qu'on remplace toute occurrence de EL par v. Or, cela revient à incorporer dans la mémoire de chaque état, une variable v représentant EL et dont la valeur est en accord avec l'histoire passée, c'est-à-dire telle que :

$\forall s \in EX(q_{racine}), \forall i \in \mathbb{N}, I(s_0)...I(s_i) \vdash EL : val \Rightarrow I(s_0)...I(s_i) \vdash v : val.$

De même, toute expression représentant une horloge définie dans le programme Pr est évaluable dans tout état grâce aux histoires. Il est donc ainsi possible de savoir pour tout état si une horloge a déjà été vraie dans son passé (strict).

La connaissance de ce fait (horloge déjà vraie dans le passé strict) est exprimée par une variable v, par horloge h, incorporée dans la mémoire de chaque état, telle que :

| | | |
|---|---|---|
| v est vraie | ssi | h n'a jamais été vraie dans le passé (instant courant compris) |
| v est fausse | ssi | h a déjà été vraie dans le passé strict (instant courant non compris) |

C'est-à-dire :

$\forall s \in EX(q_{racine}), \exists i \in \mathbb{N}, I(s_0)...I(s_i) \vdash h : true \wedge \forall j<i, I(s_0)...I(s_j) \vdash h : false \Rightarrow$
$\forall j \leq i, I(s_0)...I(s_j) \vdash v : true \wedge \forall j>i, I(s_0)...I(s_j) \vdash v : false$

Reprenons l'exemple du nœud Front défini au paragraphe précédent ; la figure 2.2.2.a représente un deuxième arbre des exécutions de ce nœud, prenant en compte cette fois des variables intermédiaires liées aux initialisations d'horloge ou aux expressions en *pre* et en *current*. La variable intermédiaire init (premier instant de l'horloge de base du programme) est définie par l'équation :

**init = true -> false**

et la variable intermédiaire p est définie par l'équation :

p = **pre**(b)

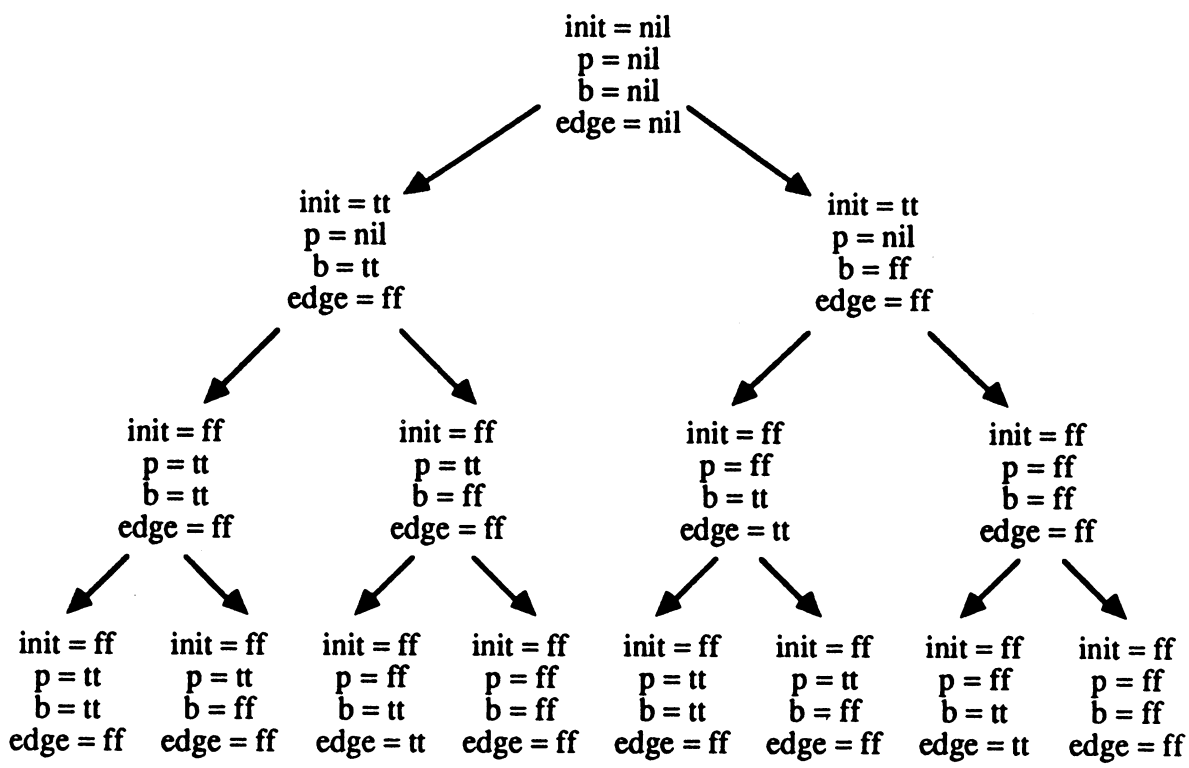



Figure 2.2.2.1.a Arbre des exécutions de Front avec mémoires étendues

Tout programme LUSTRE peut donc être représenté sous la forme d'un arbre des exécutions dont le domaine de définition des mémoires comprend :

- les variables du programme. Soit V l'ensemble de ces variables.
- une variable pour chaque expression pre(E) ou current(E) (avec E expression) apparaissant dans le programme. Soit $V_{pre}$ l'ensemble de ces variables.

• une variable booléenne pour chaque expression $E_1$ -> $E_2$ (avec $E_1$, $E_2$ expressions LUSTRE) apparaissant dans le programme, et dont la valeur vrai correspond au fait que l'horloge de l'expression n'a jamais été vraie dans les états précédents. Soit $V_h$ l'ensemble de ces variables.

#### 2.2.2.2.2 La relation $\mathfrak{R}$

Soit AE un arbre des exécutions avec **mémoires étendues**, du programme Pr : $AE=((Q,q_{racine},\rightarrow), I_E)$. On définit la relation $\mathfrak{R}$ sur QxQ par :

$\forall\ q_1,q_2 \in Q,\ q_1\ \mathfrak{R}\ q_2$ ssi $I_E(q_1) = I_E(q_2)$.

$\mathfrak{R}$ représente l'équivalence des états par rapport à leur mémoire étendue.

#### 2.2.2.2.3 $\mathfrak{R}$ est une bisimulation contenue dans $\approx$

Soit I telle que : $\forall\ q \in Q$, I(q) est la restriction de $I_E(q)$ aux variables n'appartenant pas à $V_{pre} \cup V_h$.

• $\mathfrak{R} \subseteq \approx$

car $\forall\ q_1,q_2 \in Q,\ q_1\ \mathfrak{R}\ q_2 <=> I_E(q_1) = I_E(q_2)$

et $I_E(q_1) = I_E(q_2) => I(q_1) = I(q_2)$

et $I(q_1) = I(q_2) <=> q_1 \approx q_2$

• $\mathfrak{R}$ est une bisimulation

En effet, les ensembles $V_{pre}$ et $V_h$ contiennent toute l'information passée nécessaire au calcul de toutes les variables de la mémoire, puisque :

- seuls les opérateurs *pre*, *current*, -> nécessitent pour leur évaluation des informations qui ne proviennent pas du cycle courant, mais des cycles strictement passés (voir la sémantique naturelle de LUSTRE),
- la valeur de toutes les expressions en *pre* ou en *current* est mémorisée grâce à $V_{pre}$,
- pour toute expression en ->, on est capable de déterminer grâce à $V_h$ si l'état courant correspond ou non au premier instant de son horloge, ce qui permet de l'évaluer (d'après la sémantique naturelle).

Ainsi, à l'aide de ces deux ensembles, il est possible de déterminer dans chaque état et pour chaque vecteur d'entrées, la valeur de toutes les variables de la mémoire.

Or, d'après la sémantique du *pre* et du *current*, tous les successeurs d'un état associent, par l'intermédiaire de la mémoire, la même valeur aux variables liées aux expressions en *pre* ou en *current*.

Ils associent aussi la même valeur aux variables de $V_h$ puisque :

Soit $v \in V_h$, soit $\sigma$ la mémoire de l'état courant et $\sigma'$ la mémoire d'un état successeur quelconque,

$\sigma(v)$ = vrai dans l'état initial

$\sigma(v)$ = vrai $\wedge$ $\sigma(h)$ = vrai => $\sigma'(v)$ = faux

$\sigma(v)$ = vrai $\wedge$ $\sigma(h)$ = faux => $\sigma'(v)$ = vrai

$\sigma(v)$ = faux => $\sigma'(v)$ = faux

Les états successeurs de deux états ayant même mémoire étendue, ont donc mêmes valeurs pour les variables de $V_{pre}$ et $V_h$. D'autre part, tout état a autant de successeurs que de combinaisons possibles des entrées. Tous les états ont donc des successeurs de mêmes entrées. Ainsi, deux états $q_1$ et $q_2$ ayant même mémoire étendue, possèdent chacun un successeur ($q'_1$ pour $q_1$, $q'_2$ pour $q_2$) qui a même mémoire étendue que le successeur de l'autre. Donc :

$\forall q_1,q_2 \in Q,\ I(q_1) \approx I(q_2) =>$

$[\forall q'_1 \in Q,\ q_1 \rightarrow q'_1 => \exists q'_2 \in Q,\ q_2 \rightarrow q'_2 \wedge I(q'_1) \approx I(q'_2)]$

$\wedge\ [\forall q'_2 \in Q,\ q_2 \rightarrow q'_2 => \exists q'_2 \in Q,\ q_1 \rightarrow q'_1 \wedge I(q'_1) \approx I(q'_2)]$

et $\mathfrak{R}$ est une bisimulation.

#### 2.2.2.2.4 Graphe d'états finis

Le graphe d'états d'un programme LUSTRE est fini dans le cas où les variables du programme ont un domaine de définition fini, puisqu'alors, il n'existe qu'un nombre fini de mémoires sur $V \cup V_{pre} \cup V_h$. Le graphe d'états est donc dans ce cas un modèle fini de l'arbre des exécutions du programme.

On voit par exemple ci-dessous, le repliage de l'arbre de la figure 2.2.2.2.1.a en un graphe d'états:

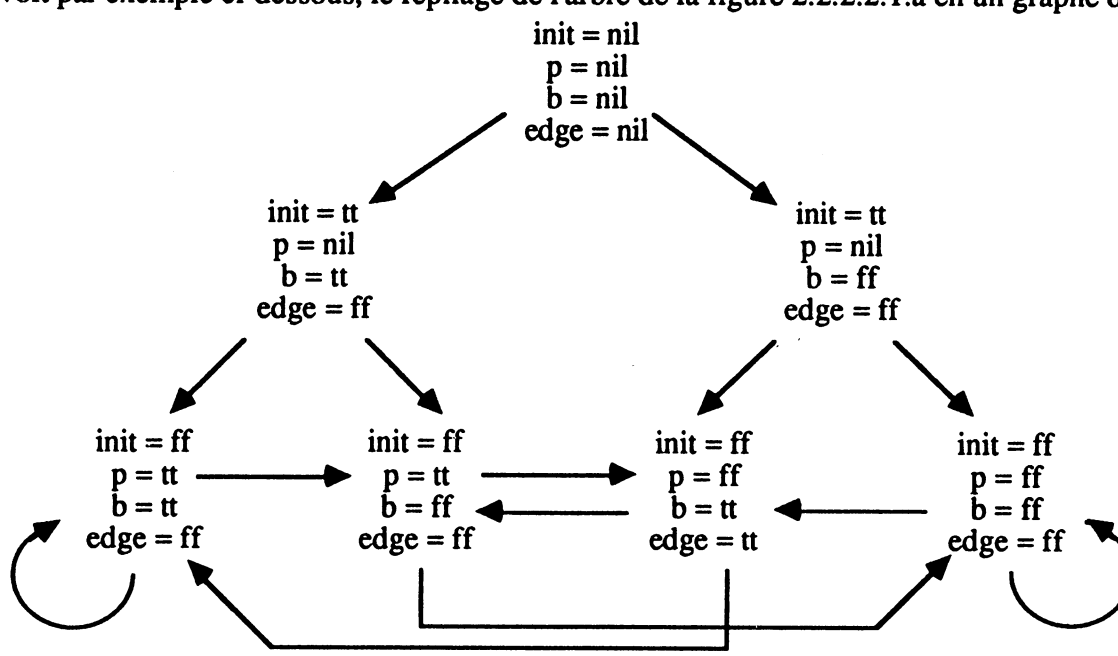



Figure 2.2.2.2.4.a : un graphe d'états SAGA/LUSTRE

Un autre modèle fini d'un programme LUSTRE est l'automate de contrôle. Cet automate est celui utilisé lors de la génération de code par le compilateur LUSTRE (cf paragraphe 1.3.3.2.1). C'est sa structure qui répartit le code à exécuter ; en ce sens, il représente le "contrôle".

### 2.2.3 Automates de contrôle

Comme le graphe d'états, ces automates sont obtenus par repliage d'un arbre des exécutions. La différence provient du fait que l'arbre des exécutions considéré est déjà une abstraction du comportement réel du programme. Différents automates peuvent être obtenus à partir du même programme, selon des options de compilation qui spécifient l'abstraction considérée.

#### 2.2.3.1 Arbre d'interprétation abstraite

On définit la notion d'interprétation abstraite [CC,77] d'un programme LUSTRE de la façon suivante :
Soit ~ une relation d'équivalence sur les mémoires, soit $\Sigma'$ le quotient $\Sigma/\sim$. On note $\alpha$ l'injection canonique de $\Sigma$ dans $\Sigma'$, et $\gamma$ la fonction inverse:

$$\gamma(\sigma') = \{\sigma \mid \alpha(\sigma) = \sigma'\}$$

Soit $H' = \Sigma'^{\infty}$, l'ensemble des suites finies ou infinies d'éléments de $\Sigma'$. On définit de même:

$$\alpha : H \to H' \ , \ \gamma : H' \to 2^H$$
$$\alpha(<>) = \alpha(\sigma_\varnothing) \ , \ \alpha(\eta.\sigma) = \alpha(\eta).\alpha(\sigma)$$
$$\gamma(\eta'.\sigma') = \{\eta.\sigma \mid \eta \in \gamma(\eta') , \sigma \in \gamma(\sigma')\}$$

On définit l'interprétation abstraite d'un programme LUSTRE par rapport à ~ comme suit:
$\eta' \in H'$ est une histoire abstraite du programme $\pi$ ssi
- soit $\eta' = \alpha(\{<>\})$
- soit $\eta' = \eta_1'.\sigma_1'$, et il existe $\eta \in \gamma(\eta_1')$ et $\sigma \in \gamma(\sigma_1')$ tels que $\sigma$ est compatible avec $\pi$ dans $\eta$ : $\eta.\sigma \vdash \pi$.

Un arbre d'interprétation abstraite de $\pi$ est un couple (A,I), où
- A est un arbre $(Q, q_{racine}, \to)$
- I: $Q \to \Sigma'$ est une fonction d'interprétation abstraite, telle que:
  - $I(q_{racine}) = \alpha(\sigma_\varnothing)$
  - $\forall s \in EX(q_{racine})$, $\forall i \in \mathbb{N}$, la suite $I(s_0), I(s_1), \ldots, I(s_i)$ est une histoire abstraite de $\pi$
  - $\forall \eta'$ histoire abstraite de $\pi$, $\exists s \in EX(q_{racine})$, $\exists i \in \mathbb{N}$, $\eta' = I(s_0), I(s_1), \ldots, I(s_i)$

### 2.2.3.2 Repliage de l'arbre d'interprétation abstraite

Nous allons définir la bisimulation $\mathfrak{R}_A$ qui permet de replier l'arbre d'interprétation abstraite en un automate de contrôle générable par le compilateur LUSTRE.
Nous avons vu que les principes de définition et de substitution permettent d'augmenter le domaine de définition des mémoires du programme $\pi$. Nous définissons alors :

- une variable pour chaque expression pre(E) ou current(E) de type **booléen** nécessaire au calcul d'une sortie ou d'une assertion (avec E expression booléenne). (ensemble $V_{pre}$)
- une variable pour chaque expression $E_1$ -> $E_2$ de type **booléen** nécessaire au calcul d'une sortie ou d'une assertion (avec $E_1$, $E_2$ expressions booléennes), et dont la valeur vrai correspond au premier instant de l'horloge de l'expression. (ensemble $V_h$)

Les variables de $V_{pre} \cup V_h$ sont appelées **variables d'état** de l'automate. Elles apparaissent dans la mémoire abstraite.

Cet arbre d'interprétation abstraite ne permet pas de calculer toutes les expressions booléennes du programme $\pi$. En effet, bien que les comparaisons entre réels, ou entiers, soient des expressions booléennes, elles ne sont pas évaluables par suite de l'abstraction réalisée sur leurs opérandes. De même les fonctions externes à valeur booléenne ne sont pas évaluables sur cet arbre.

C'est pourquoi, nous rajoutons dans la mémoire abstraite, des variables correspondant à ces expressions booléennes de base (appelées conditions) qui apparaissent dans le calcul des sorties et des assertions. Ces variables sont prises comme des entrées du programme; ainsi, chaque état a autant de successeurs que de combinaisons possibles des entrées booléennes du programme et de ses conditions booléennes (éclatement des états de l'arbre d'interprétation abstraite en autant d'états que de combinaisons possibles des conditions booléennes). Les sorties booléennes du programme $\pi$ et les assertions sont calculées dans la mémoire abstraite en fonction des entrées booléennes et de ces conditions.

Par cet ajout de conditions en tant qu'entrées, nous ajoutons à l'arbre d'interprétation abstraite des branches ne correspondant à aucune exécution du programme. Par exemple, soit un programme calculant une variable entière x comme sortie d'un compteur modulo 3, et soit la condition x>3 apparaissant dans le calcul de ses sorties. Dans l'arbre des exécutions du programme, cette condition ne sera jamais vérifiée, alors qu'elle le sera dans certains états de l'arbre étendu d'interprétation abstraite.

Les propriétés de vivacité ne pourront pas être vérifiées sur ce nouvel arbre, puisqu'il comporte plus de comportements abstraits que ceux correspondant au programme. Par contre,

toutes les propriétés de sûreté (invariance temporelle) pourront être vérifiées avec le risque d'être invalidées par un comportement n'apparaissant pas réellement dans le programme.

Soit $((Q,q_{racine},\rightarrow), I_E)$ l'arbre étendu d'interprétation abstraite ainsi obtenu. La relation $\Re_A$ est définie sur QxQ par :

$\forall\, q_1,q_2 \in Q,\ q_1 \,\Re_A\, q_2 \quad$ ssi $\quad \forall v \in V_{pre} \cup V_h,\ I_E(q_1)(v) = I_E(q_2)(v)$.

$\Re_A$ est une bisimulation sur l'arbre étendu d'interprétation abstraite de la même façon que $\Re$ aux paragraphes 2.2.2.2.2 et .3 était une bisimulation sur l'arbre des exécutions. En effet, le calcul des expressions booléennes LUSTRE de $\pi$ dépend :

- des entrées booléennes (ensemble $V_e$)
- des conditions booléennes (ensemble $V_c$)
- des variables d'état (ensemble $V_{pre}$ et $V_h$)

qui font partie de la mémoire étendue abstraite.

Dans un état, le calcul des états successeurs (prochaines valeurs des variables de $V_{pre}$ et de $V_h$) correspond au calcul d'une expression booléenne, il dépend donc des entités citées ci-dessus. Or, dans tout état, les variables d'état ont une valeur fixée (qui est la valeur associée à ces variables par les mémoires abstraites repliées en cet état), le calcul des états successeurs est donc une fonction des entrées et des conditions booléennes. Ces entrées et ces conditions étant les mêmes pour tout état du programme, si deux états ont mêmes mémoires sur $V_{pre}$ et $V_h$, alors, pour chaque combinaison des valeurs d'entrée et de condition, ses états auront des successeurs de même mémoire sur $V_{pre} \cup V_h$.

$\Re_A$ est de plus une relation d'équivalence.

L'automate de contrôle est le système de transitions $(\underline{Q}, \underline{q}_{racine}, \rightarrow)$ muni de l'interprétation $\underline{I}$, où :

- $\underline{Q} = Q/\Re_A$
- $q_{racine} \in \underline{q}_{racine}$
- $\forall\, \underline{q}_1,\underline{q}_2 \in \underline{Q},\ \underline{q}_1 \rightarrow \underline{q}_2$ ssi $\exists q_1 \in \underline{q}_1, \exists q_2 \in \underline{q}_2, q_1 \rightarrow q_2$
- $\underline{I} : \underline{Q} \rightarrow (\Sigma)^{2^n}$, où n est le cardinal de $V_e \cup V_c$

  telle que

  $\forall \underline{q} \in \underline{Q},\ \underline{I}(\underline{q}) = \{\sigma \in \Sigma \,/\, \exists q \in \underline{q},\ \sigma = I_E(q)\}$

Ce n'est pas un modèle du programme mais une abstraction finie de l'arbre des exécutions de ce programme.

Remarquons que l'interprétation associe à chaque état, non plus une seule mémoire mais un ensemble de mémoires ayant toutes la même valeur sur $V_{pre} \cup V_h$.

Comme exemple, voyons comment est réalisé le repliage de l'arbre de la figure 2.2.2.1.a (remarquons que, pour ce programme, l'arbre d'interprétation abstraite est identique à l'arbre des exécutions). Si l'on omet la variable init permettant de distinguer l'état initial, l'automate possède une seule variable d'état : pre(b).

Dans l'état initial: état 0, la sortie "edge" est fausse. L'état suivant sera, selon la valeur de b soit "pre(b)=true" soit "pre(b)=false".

Dans l'état correspondant à "pre(b)=true": état 1, la sortie "edge" est égale à "b and false", donc nécessairement fausse. L'état suivant sera, selon la valeur de b, soit "pre(b)=true" soit "pre(b)=false".

Dans l'état correspondant à "pre(b)=false": état 2, la sortie "edge" est égale à "b and true", donc égale à l'entrée. De nouveau, l'état suivant sera "pre(b)=true" ou "pre(b)=false" selon la valeur de b.

Ce qui donne l'automate suivant:

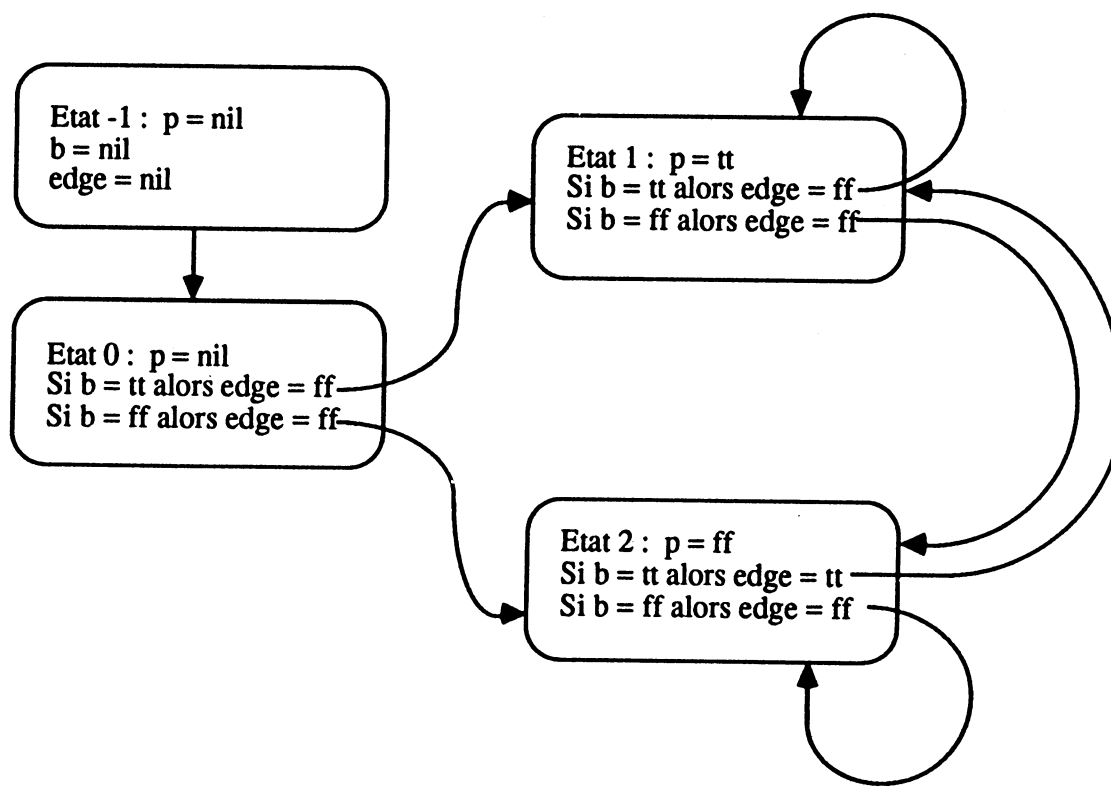



Figure 2.2.3.2.a: repliage de l'arbre

L'état -1 de l'automate correspond à l'instant précédent le premier instant de déroulement du programme. La mémoire de cet état associe la valeur nil à toutes les variables. Cet état ne représente pas un instant de calcul.

Or, l'automate de contrôle a pour vocation d'être le code exécutable correspondant à un programme LUSTRE. L'état -1 est donc contraire à cette vocation. Il peut être supprimé sans perte d'information (figure 2.2.3.2.b), car, d'une part, seul l'état racine est tel que toutes les variables de l'extension mémoire (conditions, horloges, pre et current) sont à nil, et d'autre part, tous les états initiaux de l'arbre des exécutions sont repliés sur le même état : l'état 0.

Ainsi, il est possible, à partir de l'automate généré par le compilateur LUSTRE de retrouver l'automate obtenu par repliage de l'arbre d'interprétation abstraite en rajoutant un état de mémoire $\sigma_{\emptyset}$ et dont le seul successeur est l'état 0.

De plus, il est équivalent de voir le code généré dans chaque état comme faisant partie des transitions (cf figure 2.2.3.2.c).

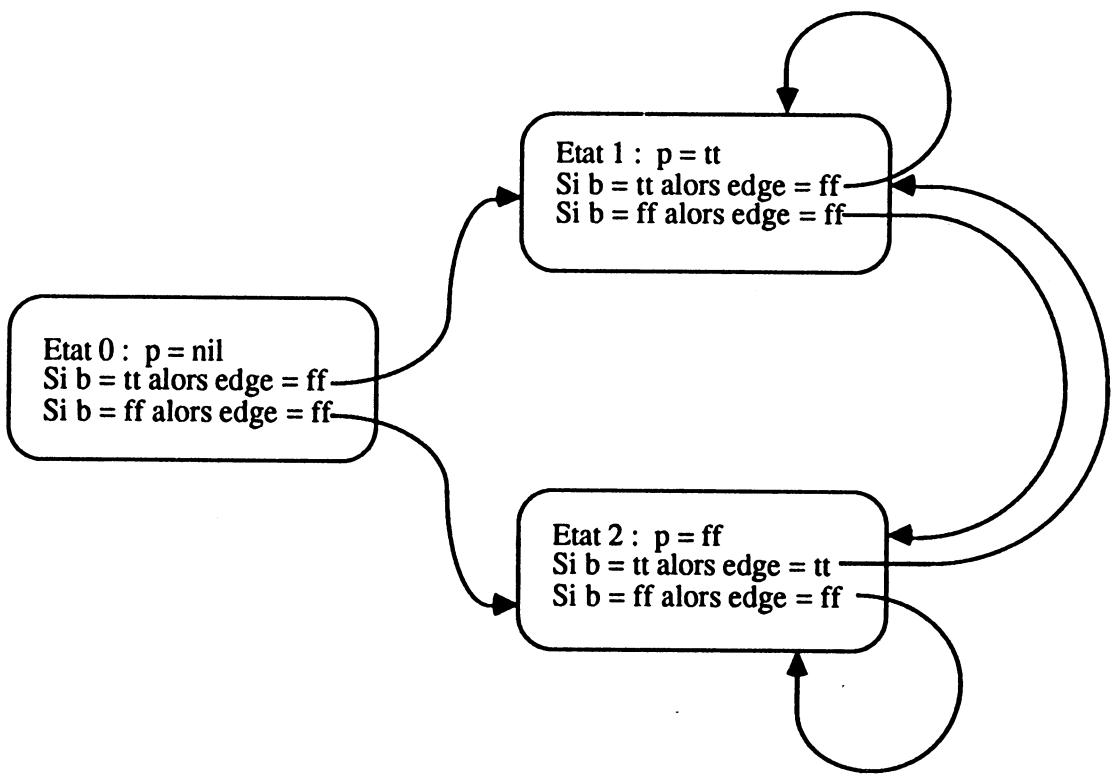



Figure 2.2.3.2.b: un automate de contrôle

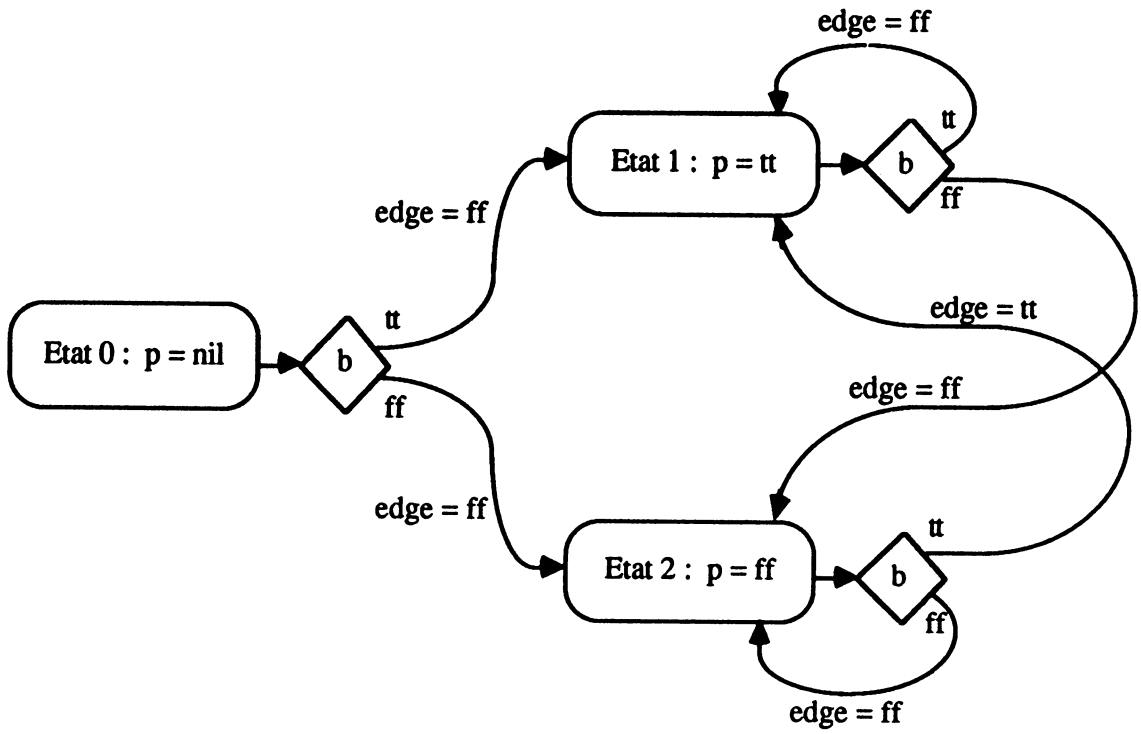



Figure 2.2.3.b : automate de contrôle avec le code sur les transitions

## 2.3 CONCLUSION

Nous avons présenté dans ce chapitre la modélisation de programmes LUSTRE sous forme d'arbres, de graphes d'états ou d'automates de contrôle. L'interprétation de ces structures fait intervenir les notions de mémoire et d'histoire telles qu'elles sont apparues dans la définition de la sémantiques naturelle de LUSTRE.

Le modèle en terme d'arbre va être utilisé dans le chapitre suivant pour l'expression des propriétés. Les deux interprétations des arbres (en tant que formules exprimant une propriété et en tant que programme LUSTRE) vont être liées afin de prouver la validité d'une propriété sur un programme LUSTRE.

Nous verrons dans les chapitres 4 et 5 comment automatiser cette preuve à l'aide des modèles en terme de graphe et d'automate de contrôle.

Rappelons que l'automate de contrôle ne permet pas de vérifier des propriétés de vivacité, mais permet de vérifier les propriétés de sûreté (invariance temporelle).

# 3 Expression des propriétés

Les logiciels conçus à Merlin Gerin/SES appartiennent pour la plupart au domaine de la sûreté : leurs spécifications font donc apparaître certains comportements requis pour garantir la sûreté, comme par exemple le déclenchement d'alarmes sonores, ou l'injection de bore dans le circuit primaire d'un réacteur nucléaire ou encore le lacher des barres de régulation du flux de neutrons dans le cœur d'un réacteur nucléaire, suivant des critères précis. Ces comportements sont considérés comme d'extrême importance puisque leur non respect peut aller jusqu'à entrainer mort d'hommes. Aussi, pour garantir la qualité de tels logiciels, différentes approches sont prises en compte :

- des commissions de spécialistes sont chargés par les gouvernements de créer des normes pour les logiciels touchant à la sûreté (CEI880 dans le domaine du nucléaire en France, GAM_T_17 dans le domaine de la défense française)
- les entreprises marquent une forte tendance à spécifier et vérifier formellement leurs logiciels.

On observe diverses situations : certains s'attachent à définir des méthodes (analyse des situations critiques, objectifs vis-à-vis de l'environnement), alors que d'autres préfèrent vérifier formellement le système, ou encore le simuler. Actuellement cependant, il n'existe que peu de méthodes d'analyse et de vérification de propriétés de sûreté, et les techniques proposées ne sont en général applicables que dans des contextes limités ([Lev,86]). On peut citer :

- **l'analyse des situations critiques par arbre de fautes** (Fault Tree Analysis) [LH,86]. La méthode consiste à définir les états indésirables du système,et à analyser les séquences d'événements (logiciels, matériels, humains...) pouvant conduire à de tels états.
- **la spécification et l'analyse des propriétés temporelles** [JM,86]. Un modèle du système est fourni en termes d'événements et d'actions. Il décrit la dépendance des données et l'ordonnancement des actions à réaliser en réponse aux événements extérieurs. Ce modèle peut ensuite être traduit automatiquement en formules de la logique temporelle RTL. Les propriétés que le système doit vérifier sont des formules de RTL. Pour réaliser la vérification, les formules de RTL spécifiant le système et celles décrivant les propriétés sont traduites par des formules de l'arithmétique de Presburger avec des fonctions entières non interprétées. Seul un sous-ensemble de l'arithmétique de Presburger est utilisé, ce qui permet d'obtenir des procédures de décision pour déterminer si chaque propriété de sûreté est un théorème dérivable de la spécification du système. RTL permet d'exprimer un ordonnancement absolu aussi bien que relatif des événements.

• **la simulation par réseaux de Pétri.** Leveson et Stolzy [LS,86] ont défini des procédures d'analyse et de simulation de réseaux de Pétri admettant des événements parallèles et concurrents. Ces procédures permettent, soit de déterminer les propriétés de sûreté requises (y compris les propriétés temporelles), directement à partir de la description du système, soit d'analyser la fiabilité et la tolérance aux fautes du système, soit encore d'aider à la détection de fautes et de pannes. Les fautes et les pannes peuvent être incorporées dans le réseau de Pétri pour déterminer leurs effets sur le système et en déduire les fonctions à protéger.
La difficulté de la construction des réseaux de Pétri pour des applications de grandeur réelle est cependant un inconvénient majeur à leur utilisation.

La complexité des logiciels considérés (volume important des données à traiter, répartition des traitements, ...) rend quasiment impossible leur vérification formelle complète. En effet, au coût élevé de la vérification s'ajoute la difficulté de réaliser une spécification complète. Par contre, le coût d'une vérification partielle est moindre et est totalement justifié lorsque cette vérification porte sur les contraintes de sûreté à assurer.

Afin de déterminer si des techniques particulières sont nécessaires pour résoudre les problèmes que pose le domaine de la sûreté chez Merlin Gerin/SES, nous avons voulu étudier de façon précise le type d'applications qui y sont développées et les exigences de sûreté qui leur sont associées. Nous avons alors analysé un certain nombre de systèmes réactifs dans ce domaine et, pour chacune de ces applications, nous avons exprimé les contraintes essentielles du comportement désiré. Ces contraintes portent d'une part sur la fonction propre que chaque application doit remplir, d'autre part sur les qualités de sûreté requises. L'énoncé de ces propriétés a d'abord été réalisé en français, puis à l'aide de constructions définies, pour la circonstance, dans une syntaxe proche du français.

Le premier objectif de ce travail a été de mettre en évidence un certain nombre de notations qui permettent d'exprimer les contraintes principales des logiciels. La suite, qui fait l'objet du paragraphe 3.2, a permis d'extraire des diverses variations syntaxiques, exposées dans ce chapitre, un langage formel pour l'expression des propriétés.

Nous proposons donc, dans ce chapitre, d'exposer les conclusions qui ont été tirées de ces études de cas, et notamment comment, à partir des notations utilisées et du sens intuitif qu'on leur prête, nous concevons le langage requis pour l'expression des propriétés. Ce langage est ensuite défini formellement.

## 3.1 ETUDES DE CAS

Notre but est d'étudier le problème de la spécification pour des applications dont la conception est réalisée en SAGA. Afin d'identifier les besoins, nous avons étudié un certain nombre d'applications développées par Merlin Gerin/SES et pour lesquelles un cahier des charges rédigé en français était disponible, ainsi que des applications similaires dans le domaine du temps réel. A partir du cahier des charges de chaque application, nous avons cherché à tirer les contraintes importantes de bon fonctionnement et de sûreté. Nous avons constaté que de telles contraintes sont fortement dépendantes de l'application considérée. De plus, et malheureusement, ces contraintes n'apparaissent pas clairement dans les documents, qui mélangent souvent les niveaux d'abstraction.

### 3.1.1 Méthode de spécification

Nous nous sommes appliqués tout d'abord à considérer chaque programme comme une boîte noire dont seules les sorties seraient apparentes, et à exprimer leurs contraintes de cohérence mutuelle. Puis nous avons pris en compte les entrées de la boîte, et nous avons exprimé pour chaque sortie sa dépendance vis-à-vis des entrées. Nous avons ensuite introduit des variables pour exprimer des relations entre les entrées et les sorties. Ces variables représentent l'état courant du système, elles ont été définies alors que nous considérions toujours l'application comme une boîte noire. Elles n'existent donc pas forcément telles quelles dans le programme développé. Dans une dernière étape, nous avons cherché à envisager le comportement général du programme notamment en terme de propriétés de non blocage. Les propriétés portent en général sur les variables définissant l'état courant du système, et donc en particulier sur les variables internes définies précédemment.

Dans chacune de ces étapes, les propriétés exprimées précisent le comportement du système en indiquant ce qu'il doit faire, tout aussi bien que ce qu'il ne doit pas faire. Les propriétés de sûreté expriment souvent, en effet, les situations à éviter [Lev, 86].

La méthode de spécification employée, en quelque sorte une "spécification dirigée par les sorties", a conduit à exprimer les propriétés en tenant compte du passé (passé des sorties, des entrées, des variables internes), mais nous aurions pu procéder autrement et choisir d'exprimer pour chaque combinaison d'entrées, la valeur des sorties à venir. Nous pensons que le résultat, du point de vue du comportement à vérifier, aurait été strictement identique; seule l'expression aurait été différente. Nous avons néanmoins constaté que, vu la place primordiale qu'occupe la fonction

de surveillance dans les applications de sûreté, les sorties sont en nombre important et couvrent, de façon très satisfaisante pour notre méthode, l'ensemble du problème.

Comme nous le verrons dans le paragraphe suivant, à travers la spécification d'un exemple, certaines propriétés ont fait intervenir des notions temporelles. Il a alors été nécessaire pour les exprimer, d'employer des opérateurs dits temporels, qui permettent d'exprimer l'invariance dans le temps, l'apparition d'un événement passé, la possibilité au cours du futur ou, l'inévitabilité future mais bornée dans le temps.

Nous allons maintenant présenter un exemple de spécification d'application. Les propriétés que doit vérifier cette application, ont été exprimées à l'aide de constructions proches du langage naturel. Ces constructions ont été retrouvées lors de la spécification d'autres applications; elles semblent donc représentatives de la spécification des logiciels de sûreté tels qu'il nous a été donné de les étudier à Merlin Gerin/SES. L'exposé de l'exemple suit les quelques règles présentées plus haut, relatives à la "spécification dirigée par les sorties" : tout d'abord, une présentation générale de l'environnement du logiciel et de ses principales fonctions, puis l'énoncé des sorties du programme à réaliser, et enfin la présentation d'un petit nombre de contraintes à vérifier, suivant la méthode décrite au début de ce paragraphe (la spécification complète de l'exemple se trouve en annexe 1).

### 3.1.2 Exemple

#### 3.1.2.1 Fonctionnement général

L'application considérée est extraite d'une application réelle : SIREX, développée à Merlin Gerin/SES. Elle porte sur l'analyse du flux de neutrons circulant dans le cœur d'un réacteur nucléaire. Sa fonction principale est de surveiller ce flux, et d'émettre des alarmes, voire de déclencher des arrêts d'urgence, lorsque le flux franchit certains seuils.

Pour cela, des capteurs sont disposés tout autour du cœur, à l'extérieur. Il correspondent deux technologies différentes qui permettent de capter des flux d'intensités différentes. Les uns captent des flux de faible intensité : ils constituent le bas niveau de capteurs. Les autres permettent de capter des flux d'intensité moyenne à très élevée : ils constituent le haut niveau de capteurs.

Afin d'avoir une bonne approximation du flux dans le cœur, le logiciel doit gérer l'utilisation de ces capteurs. A chaque instant, il choisit le flux capté sur le bas niveau de capteurs

ou celui capté sur le haut niveau comme représentant du flux réel. Le niveau de capteurs sur lequel est choisi le flux est qualifié de valide.

Le flux est donc comparé à des seuils afin de mettre à jour un certain nombre d'afficheurs, de gérer les capteurs et de déclencher des alarmes ou des arrêts d'urgence. Ces seuils sont établis de telle manière que le flux entre deux mesures, ne peut franchir plus d'un seuil à la fois, que ce soit dans le sens de la montée ou de la descente en puissance du réacteur. Ceci est une propriété importante de l'environnement du logiciel.

Pour simplifier l'exposé de cette étude de cas, nous avons émis trois hypothèses sur l'environnement :

- Lorsque l'ensemble logiciel-électronique de surveillance tombe en panne, le réacteur est arrêté (ce qui aurait lieu si cet ensemble n'était pas en redondance afin d'augmenter la sûreté).
- Chaque fois qu'un arrêt d'urgence est déclenché, l'ensemble de surveillance est arrêté dès qu'il indique que la réaction nucléaire est finie. Il est ensuite relancé en même temps que la réaction.
- Enfin, si les deux chaînes de capteurs ne sont pas en phase (à un $\Delta$ près à définir), l'ensemble de surveillance est considéré en panne.

Les deux premières hypothèses nous conduisent à l'assertion suivante :
"au démarrage de l'application, le flux de neutrons dans le réacteur nucléaire est nul".

Cette assertion ne correspond pas à la réalisation actuelle du logiciel. En effet, afin d'assurer une surveillance continue du réacteur, logiciels et matériels sont utilisés en redondance. Aussi, lorsqu'un système tombe en panne et est réparé, les autres sytèmes et le réacteur continuent à fonctionner. Donc, lors de la réinstallation du système en panne, le flux dans le réacteur n'est pas forcément nul.

Cependant, les hypothèses ci-dessus sont vérifiées dans le cas d'un système de surveillance unique. Elles n'empêchent donc pas d'avoir une vision cohérente d'une application réelle et ont permis d'obtenir une spécification simplifiée pour cet exposé.

#### 3.1.2.2 Les sorties

Les sorties sont de deux types : les unes constituent l'affichage et sont directement observables par l'opérateur, les autres exercent un contrôle sur un autre système et l'opérateur ne peut en voir que les effets.
Les premières affichent les renseignements suivants :

- le bas niveau est valide ou pas (sortie BN-valide),
- le haut niveau est valide ou pas (sortie HN-valide),
- la valeur du flux prélevée sur le bas niveau (sortie Flux-BN),
- la valeur du flux prélevée sur le haut niveau (sortie Flux-HN),
- la valeur de la période (sortie Période),
- le changement de niveau valide du bas niveau au haut niveau est autorisé ou pas (sortie Commut-BH),
- pour chacun des seuils MinBN, Mx1BN, Mx2BN, s'il a été franchi par le flux capté par le bas niveau (sorties PS-MinBN, PS-Mx1BN, PS-Mx2BN),
- pour chacun des seuils MinHN, Mx1HN, Mx2HN, s'il a été franchi par le flux capté par le haut niveau (sorties PS-MinHN, PS-Mx1HN, PS-Mx2HN),
- pour chacun des seuils Mx1PE, Mx2PE, s'il a été franchi par la période (sorties PS-Mx1PE, PS-Mx2PE).

La période rend compte de la vitesse de croissance du flux.
Lorsque le niveau valide est le bas niveau de capteurs, le système peut proposer, sous certaines conditions, de valider le haut niveau. Il ne peut cependant réaliser ce changement qu'après approbation de l'opérateur par le biais d'une commande. C'est ce qui justifie l'affichage de Commut-BH.

Les sorties exerçant un contrôle sont, quant à elles, de trois sortes :
- certaines déclenchent des alarmes sur passage de seuils. Ce sont les sorties : Al-MinBN, Al-Mx1BN, Al-MinHN, Al-Mx1HN, Al-Mx1PE, correspondant aux seuils MinBN, Mx1BN, MinHN, Mx1HN, Mx1PE,
- d'autres déclenchent des arrêts d'urgence sur passage de seuils. Ce sont les sorties AU-Mx2BN, AU-Mx2HN, AU-Mx2PE, correspondant aux seuils Mx2BN, Mx2HN, Mx2PE,
- la dernière conditionne la marche de la haute tension qui active le bas niveau (sortie MarcheHT).

#### 3.1.2.3 Propriétés

Notre démarche de conception pour cette présentation a consisté, partant des sorties, à nous poser deux types de questions :

- sous quelles conditions les sorties sont-elles cohérentes entre elles ?
- quelles sont les conditions requises pour que telle sortie prenne telle valeur ?

Cette dernière question nous a conduits à nous intéresser au comportement interne du système, et par la suite à nous poser une question supplémentaire :

• quelles sont les propriétés générales attendues pour le comportement interne?

Nous allons répondre à chacune de ces questions par l'énoncé d'un certain nombre de propriétés.

Les constructions temporelles utilisées pour cela sont compatibles avec SAGA et LUSTRE, et notamment conservent la notion de cycle propre à ces langages. Un cycle du programme correspond au calcul d'une valeur des variables de sorties. A chaque cycle, de nouvelles valeurs des entrées sont consommées.

Les constructions ayant servi à l'expression des propriétés se veulent de forme proche du français afin de conserver le caractère intuitif des propriétés, qui pourraient alors apparaître telles quelles dans le cahier des charges. Le caractère "naturel" de l'énoncé, nous a conduit à employer des constructions qui peuvent être exprimées en fonction d'autres déjà fournies. Tout au long de l'exposé, diverses variantes syntaxiques sont présentées et les rapports qui existent entre elles explicitement énoncés.

Dans ce paragraphe, chaque notation est introduite par son utilisation dans une propriété. Aussi seules quelques propriétés choisies ont été retenues. L'énoncé complet des 72 propriétés se trouve en annexe 1.

#### 3.1.2.3.1 Cohérence des sorties

Dans ce paragraphe sont énoncés des invariants liant les sorties entre elles.

**Cohérence entre les sorties "niveau valide"**

L'une des chaînes de capteurs : bas niveau ou haut niveau, doit toujours être valide, mais jamais les deux à la fois. En effet, la surveillance du flux doit être continue. Donc, à tout instant, une des chaînes de capteurs doit fournir la valeur représentative du flux. Cette valeur doit être unique, ce qui induit que les deux chaînes ne peuvent être valides à la fois.

On définit un opérateur : *toujours* tel que *toujours* (A) signifie que la propriété A est vraie dans tous les états que peut prendre le système. On transcrit alors la propriété de la façon suivante :

*toujours* (HN-valide *ou* BN-valide)

*et toujours* (*non* (HN-valide *et* BN-valide))

Nous pouvons aussi introduire une autre notation : *jamais* , telle que *jamais*(A) = *toujours* (*non* A), et réécrire la propriété en :

*toujours* (HN-valide *ou* BN-valide)*et jamais* (HN-valide *et* BN-valide) (1)

**Cohérence entre les sorties "passage de seuil" et les autres**

Exemple : on doit observer le passage du seuil MinBN, chaque fois que la valeur de Flux-BN est inférieure à la valeur MinBN, le bas niveau étant valide :

*toujours* (((Flux-BN < MinBN) *et* BN-valide) => PS-MinBN)

On préfèrera introduire une nouvelle notation : ... *chaque fois que*... telle que
A *chaque fois que* B = *toujours* (B=> A), afin d'être plus proche de l'énoncé en français d'une condition suffisante :

PS-MinBN *chaque fois que* ((Flux-BN < MinBN) *et* BN-valide) (2)

Exemple : on ne peut jamais observer le passage du seuil MinBN, sans que le bas niveau soit valide et la valeur de Flux-BN inférieure à la valeur MinBN+hystérésis. Cette propriété pourrait être exprimée à l'aide des opérateurs précédents par :

*toujours* (PS-MinBN => (BN-valide *et* Flux-BN < MinBN+hystérésis))

En introduisant la notation : *jamais*... *sans*... telle que
*jamais* A *sans* B = *toujours* (A=> B), afin d'approcher d'être plus proche de l'énoncé en français d'une condition nécessaire, on peut réécrire la propriété en :

*jamais* PS-MinBN *sans* (BN-valide *et* Flux-BN < MinBN+hystérésis) (10)

### 3.1.2.3.2 Conditions à l'observation

Nous allons répondre maintenant à la question "sous quelles conditions une sortie prend-elle une valeur donnée ?". Pour cela, nous serons amenés à parler des entrées, voire du comportement interne du système. Nous présentons les propriétés (35), (37), (38) et (41) qui nous permettent d'introduire de nouvelles notations.

**Observation de la sortie "Période"**

La période, à un instant donné, est fonction des valeurs courante et précédente du flux mesuré sur le niveau valide à cet instant (remarquons par conséquent que la période ne peut être définie au premier cycle du programme). Pour que la valeur de la période soit significative, il faut que les deux valeurs du flux le soient aussi, et par conséquent que la chaîne valide à l'instant du calcul soit active et qu'elle l'ait été à l'instant précédent. Nous utilisons alors deux notations :

- *au cycle précédent* telle que la propriété *au cycle précédent* (A) est vraie si le cycle courant n'est pas le premier, et si la propriété A était vraie au cycle précédent,

• *initial* telle que *initial* est vrai au premier cycle et faux à tous les autres cycles,
pour exprimer la propriété :

*jamais* (*non initial et* BN-valide) *sans* (*au cycle précédent* BN-actif) (35)

De plus, la période prendra obligatoirement sa valeur par défaut au premier cycle (cycle où la valeur précédente du flux n'est pas définie). On définit *initialement* comme une nouvelle notation pour *initial* => (A), et on écrit :

*initialement* (Période = 0) (37)

**Observation des sorties "niveau valide"**

La première condition est :

Le bas niveau est valide à l'instant initial. Il doit le rester jusqu'au moment où l'opérateur accepte le changement qui lui était proposé par la sortie CommutBH. Pour l'exprimer, nous allons définir une nouvelle construction :

*toujours continuellement... entre... et...*, telle que *toujours continuellement* A *entre* B *et* C est vraie lorsque, quelle que soit l'évolution du système, la propriété A est continuellement vraie entre le moment, inclus, où B est vraie et celui, exclus, où C est vraie.

*toujours continuellement* BN-valide
*entre initial*
*et* (*au cycle précédent* (CommutBH) *et* CommandeOp) (38)

**Observation des sorties "passage de seuil"**

Par exemple : lorsque la valeur de Flux-BN est inférieure au seuil MinBN alors que le bas niveau est valide, ce seuil doit être considéré comme franchi. Il doit le rester tant que la valeur de Flux-BN est inférieure ou égale à MinBN+hystérésis. Pendant tout ce temps, le bas niveau reste valide.

Pour exprimer cette propriété, nous allons définir la construction : *après... et_avant...* par : *après* A *et_avant* B est vraie si le système est passé dans un état vérifiant la propriété A sans être encore passé dans un état vérifiant B, et utiliser l'opérateur logique *équivalent à...* :

*toujours*
(PS-MinBN *équivalent à*
( BN-valide *et*
(*après* (Flux-BN < MinBN *et* BN-actif *et* BN-valide)
*et_avant* (Flux-BN > MinBN+hystérésis)))) (41)

#### 3.1.2.3.3 Propriétés générales de la gestion des capteurs

Toute la gestion des capteurs du bas et du haut niveau a été développée afin d'avoir une meilleure connaissance du flux à l'intérieur du réacteur. La valeur affichée du flux est définie comme suit :

Flux = Si BN-valide alors Flux-BN sinon Flux-HN

On désire que toutes les valeurs prises par le flux (en supposant que les afficheurs de Flux-BN et de Flux-HN le permettent) puissent être captées et affichées, c'est-à-dire :

*toujours* ($\forall$ x $\in$ [0, X], *possible* (Flux = x))

où *possible* (A) signifie que le système peut accéder à un état qui vérifie A.

On en déduit qu'il faut notamment les propriétés suivantes :

(*possible* (HN-actif)) *chaque fois que* BN-actif (62)

(*possible* (BN-actif)) *chaque fois que* HN-actif (63)

(*possible* (HN-valide)) *chaque fois que* BN-valide (64)

(*possible* (BN-valide)) *chaque fois que* HN-valide (65)

REMARQUE :

Dans le chapitre 6, seront exposés des exemples issus de la validation automatique du programme SAGA correspondant à cette application. On remarquera alors que certaines propriétés énoncées ici pour SIREX sont fausses. Certaines erreurs proviennent d'une mauvaise spécification, l'énoncé d'une propriété ne correspondant pas à l'idée intuitive qu'on désirait exprimer. L'outil de validation fournira une aide pour la découverte des erreurs, et ces propriétés seront modifiées en conséquence (cf annexe 1 fin).

### 3.1.3 Conclusion des études de cas

Remarquons qu'il suffit d'un nombre restreint de constructions temporelles pour exprimer les contraintes essentielles de cette application. Ces constructions sont au nombre de dix, et sont les suivantes :

*toujours..., jamais..., jamais... sans..., ... chaque fois que..., initial, initialement..., au cycle précédent..., après... et_avant..., toujours continuellement... entre... et ..., possible...*

De plus, comme nous l'avons exposé, certaines de ces constructions sont exprimables en fonction des autres.

Ces constructions nous ont, en fait, permis d'exprimer les contraintes des autres applications traitées. Celles-ci consistaient, pour l'une, en la commande des barres de régulation à l'intérieur du cœur d'un réacteur nucléaire, pour une deuxième, en la surveillance centralisée des rejets toxiques dans les différents bâtiments d'une centrale nucléaire, et pour la dernière, en la surveillance d'un service d'urgence dans un hôpital.

Remarquons aussi, que les constructions utilisées, à l'exception de *initial*, *initialement*, *au cycle précédent*, *après...et_avant*, et surtout *possible*, sont des invariants temporels, ce qui correspond à des contraintes globales sur tout le comportement du système spécifié. Les propriétés énoncées à l'aide de ces constructions décrivent principalement qu'un comportement indésirable n'a jamais lieu. Ceci est dû au déterminisme que l'on attend de la part d'un système réactif. Ces propriétés sont en général des propriétés de sûreté ("safety properties") au sens de [AS,85].

Par ailleurs, toute propriété temporelle non invariante (marquant la possibilité, l'inévitabilité,...) n'apparaît pas en général dans le cahier des charges. Celui-ci décrit plutôt des comportements précis à mettre en œuvre pour assurer la propriété. Ces comportements sont, quant à eux, descriptibles sous forme de propriétés invariantes.

Un langage d'invariants, qui ne comporte la possibilité de décrire que la modalité temporelle "toujours", ne permet pas d'énoncer des possibilités ou des inévitabilités. Il ne permet donc pas de vérifier que les choix réalisés dans le cahier des charges pour mettre en œuvre ces propriétés sont corrects. Par contre, il permet d'exprimer le principal pour la vérification d'un programme , c'est-à-dire, ce qui doit être réalisé de façon concrète, et ce qui ne doit pas apparaître dans la réalisation.

Pour conclure, précisons que les constructions utilisées pour la formalisation des propriétés, sont dépendantes de la méthode de spécification employée. En effet, spécifier pour une sortie donnée, ses conditions d'apparition, a conduit à faire référence à l'état passé du système et à employer des constructions telles que *au cycle précédent*. Spécifier pour certaines entrées, les sorties qu'elles produiront, aurait plutôt conduit à employer des termes comme *au cycle suivant*. L'énoncé de la cohérence entre les sorties est à l'origine de la construction *jamais... sans*. Il est possible ainsi de justifier par la méthode toutes les constructions employées.

C'est pourquoi, bien que notre étude porte sur l'expression et de la validation de propriétés, et non sur la méthode à employer pour extraire ces propriétés d'un cahier des charges, elle ne sera pas indépendante de cette méthode.

Cette extraction de propriétés, directement à partir du cahier des charges, peut être réalisée indépendamment de la conception du programme en SAGA. Aussi, la démarche employée ne

conduit pas forcément à un découpage des propriétés qui suit l'affinement réalisé pour la conception du programme.

Il paraît alors difficile de s'appuyer sur l'expression des propriétés pour concevoir le programme. Si cela était possible malgré tout, la confiance mise dans le programme serait accrue, et cela pourrait permettre de vérifier les propriétés, de façon modulaire sur les différents niveaux d'affinement du programme.

Remarquons encore que des constructions semblables aux nôtres, ont été utilisées (et formalisées) dans d'autres domaines, notamment la validation de protocoles [RRSV,87a], [GSV,88].

## 3.2 LANGAGE D'EXPRESSION DES PROPRIÉTÉS : LEP

Dans ce paragraphe, nous allons définir un langage (LEP) qui va nous permettre d'exprimer les propriétés de logiciels d'automatisme temps réel de façon non ambigüe. Il reprend les notations définies au paragraphe précédent et leur donne une sémantique formelle. Cette sémantique va être exprimée en termes d'arbres. Nous ferons le lien entre l'interprétation de ces arbres en tant que modèles d'une formule et les arbres des exécutions définis au chapitre 2. Cette approche dans l'expression de la sémantique du langage nous permettra d'envisager dans les chapitres suivants, la vérification des propriétés exprimées en LEP par des méthodes d'évaluation de formules sur des modèles.

### 3.2.1 Définition de la syntaxe

Parmi les notations apparues au paragraphe précédent, certaines ont été choisies comme opérateurs de base du langage LEP. Les autres sont conservées comme des notations et définies en fonction des premières.

Trois opérateurs de base de LEP n'apparaissent pas au paragraphe précédent. Ils expriment des modalités différentes de celles apparaissant dans les études de cas réalisées, et ont été introduits pour étendre le type des propriétés exprimables.

Soit P un ensemble de variables propositionnelles et la métavariable p∈ P. On définit la syntaxe des formules de la façon suivante:

**g ::= vrai | p | non g | $g_1$ ou $g_2$ | initial | au cycle précédent g**
**| toujours cont g entre $g_1$ et $g_2$ | toujours exist g entre $g_1$ et $g_2$**
**| préservable cont g entre $g_1$ et $g_2$ | préservable exist g entre $g_1$ et $g_2$.**

On définit les notations :

**faux, et,** =>, et ≡ de la manière habituelle.

et les notations :

**toujours** g = toujours cont g entre vrai et faux
**jamais** g = toujours (non g)
$g_1$ **chaque fois que** $g_2$ = toujours ($g_2$ =>$g_1$)
**jamais** $g_1$ **sans** $g_2$ = toujours ($g_1$ =>$g_2$)
**possible** g = non (toujours (non g))
**possible cont** g **entre** $g_1$ **et** $g_2$ = non (toujours exist (non g) entre $g_1$ et $g_2$)
**possible exist** g **entre** $g_1$ **et** $g_2$ = non (toujours cont (non g) entre $g_1$ et $g_2$)
**préservable** g = préservable cont g entre vrai et faux
**inévitable cont** g **entre** $g_1$ **et** $g_2$ = non (préservable exist (non g) entre $g_1$ et $g_2$)
**inévitable exist** g **entre** $g_1$ **et** $g_2$ = non (préservable cont (non g) entre $g_1$ et $g_2$)
**inévitable** g = non (préservable (non g))
**initialement** g = initial => g

Les définitions de *toujours*, *possible*, *préservable* et *inévitable* peuvent ne pas être intuitives (par rapport à la définition habituelle de tels opérateurs), mais seront justifiées par la sémantique des opérateurs de LEP.

### 3.2.2 Définition de la sémantique

#### 3.2.2.1 Modèles d'une formule

Soit A = (Q, $q_{racine}$, →) un arbre,
et I une interprétation des variables propositionnelles : I : Q->$2^P$,
(A,I) est un modèle de f, formule de LEP, (et on écrira A,I ⊨ f ) ssi $q_{racine} \models_I f$

### 3.2.2.2 Satisfaction d'une formule par un état

$q \models_I f$, où $q \in Q$ et f est une formule bien formée, signifie que la formule f est vraie dans l'état q sous l'interprétation I.

a)

$q \models_I$ **vrai**, $\forall q \in EX(q_{racine})$ (1)

$q \models_I$ **initial** ssi $q_{racine} \rightarrow q$ (2)

q correspond au premier instant de déroulement du programme

$q \models_I p$ ssi $q \neq q_{racine}$ et $p \in I(q)$ (3)

la proposition p est vraie dans l'état q

$q \models_I g$ **ou** g' ssi $(q \models_I g$ ou $q, \models_I g')$ (4)

l'une des propriétés g et g', est vraie dans l'état q

$q \models_I$ **non** g ssi $(q \not\models_I g)$ (5)

la propriété g est fausse dans l'état q

On définit la fonction père : Q -> Q telle que $\forall q, q' \in Q$, q = père(q') ssi $q \rightarrow q'$

$q \models_I$ **au cycle précédent**(g) ssi $q \neq q_{racine} \wedge$ père(q) $\models_I$ g (6)

le prédécesseur de q vérifie la propriété g.

b)

Nous nous intéressons maintenant aux formules définies en fonction de sous-séquences d'exécution. Il s'agit des formules: *toujours cont* g *entre* $g_1$ *et* $g_2$, *toujours exist* g *entre* $g_1$ *et* $g_2$, *préservable cont* g *entre* $g_1$ *et* $g_2$ et *préservable exist* g *entre* $g_1$ *et* $g_2$. Ces formules traitent de l'apparition d'une propriété g pendant les sous-séquences d'exécution dont le premier état vérifie la propriété $g_1$ et dont le dernier (lorsque la sous-séquence est finie) vérifie la propriété $g_2$.

Elles ne tiennent pas compte des sous-séquences d'exécution débutant par un état qui vérifierait $g_2$ en même temps que $g_1$. En effet, les propriétés $g_1$ et $g_2$ peuvent être vues comme, respectivement, les conditions d'apparition et de suppression de la propriété g.

Ces formules vont être définies différemment pour l'état $q_{racine}$ que pour les autres états. Ceci est dû à ce que nous nous intéressons à des arbres qui seront modèles d'un programme LUSTRE et dont la racine n'aura pas de signification vis-à-vis de l'exécution de ce programme.

La sémantique de la propriété *toujours cont* g *entre* g1 *et* g2 telle qu'elle est apparue dans les études de cas, consiste à dire que pour toute séquence d'exécution, la propriété g doit être continuellement vraie dans toute portion d'exécution débutant par la propriété $g_1$ et finissant par la propriété $g_2$. Sa traduction immédiate est alors :

Pour $q \neq q_{racine}$ :

$q \models_I$ **toujours cont g entre** $g_1$ **et** $g_2$ ssi

$\forall s \in EX(q), \forall k, ($ $(s_k \models_I g_1$ et non $g_2) \Rightarrow$

$(\exists k', k'>k \wedge s_{k'} \models_I g_2 \wedge \forall k'', k'>k''\geq k \Rightarrow s_{k''} \models_I g$ et non $g_2)$

$\vee$ $(\forall k', k'\geq k \Rightarrow s_{k'} \models_I g$ et non $g_2)$ $)$

Pour $q=q_{racine}$, la même définition est appliquée avec $k\geq 1$.

Ainsi, dans toute situation, la propriété $g_1$ provoque l'apparition de la propriété g, qui ne disparait alors qu'à l'apparition de la propriété $g_2$. Par exemple g représente une alarme, $g_1$ représente la situation anormale déclenchant l'alarme représentée par g, et $g_2$ représente le retour à une situation normale.

La condition $g_2$ de suppression de la propriété g peut ne jamais apparaître après l'arrivée de la propriété $g_1$; alors, tout doit se passer comme si la propriété $g_2$ était repoussée à l'infini: la propriété g ne doit jamais cesser d'être vraie (suivant le même exemple que précédemment, si les conditions dans l'environnement restent à jamais anormales, l'alarme ne doit jamais être supprimée).

Le schéma 3.2.2.a. résume, en terme de séquences d'exécution, la sémantique de la propriété *toujours cont* g *entre* g1 *et* g2 : toute séquence d'exécution de type ④ invalide cette propriété, alors que celles de type ① , ② ou ③ contribuent à la vérifier.

Cette sémantique peut être exprimée de manière plus concise (l'équivalence des deux définitions est montrée en l'annexe 2.2):

Pour $q \neq q_{racine}$ :

$q \models_I$ **toujours cont g entre** $g_1$ **et** $g_2$ ssi

$\forall s \in EX(q), \forall k, ($ $(s_k \models_I g_1$ et non $g_2) \Rightarrow$

$(\forall k'', (k\leq k'' \wedge s_{k''} \models_I$ non $g) \Rightarrow \exists k', k\leq k'\leq k'' \wedge s_{k'} \models_I g_2 ))$ (7)

Pour $q=q_{racine}$, la définition ne s'applique qu'aux $k\geq 1$.

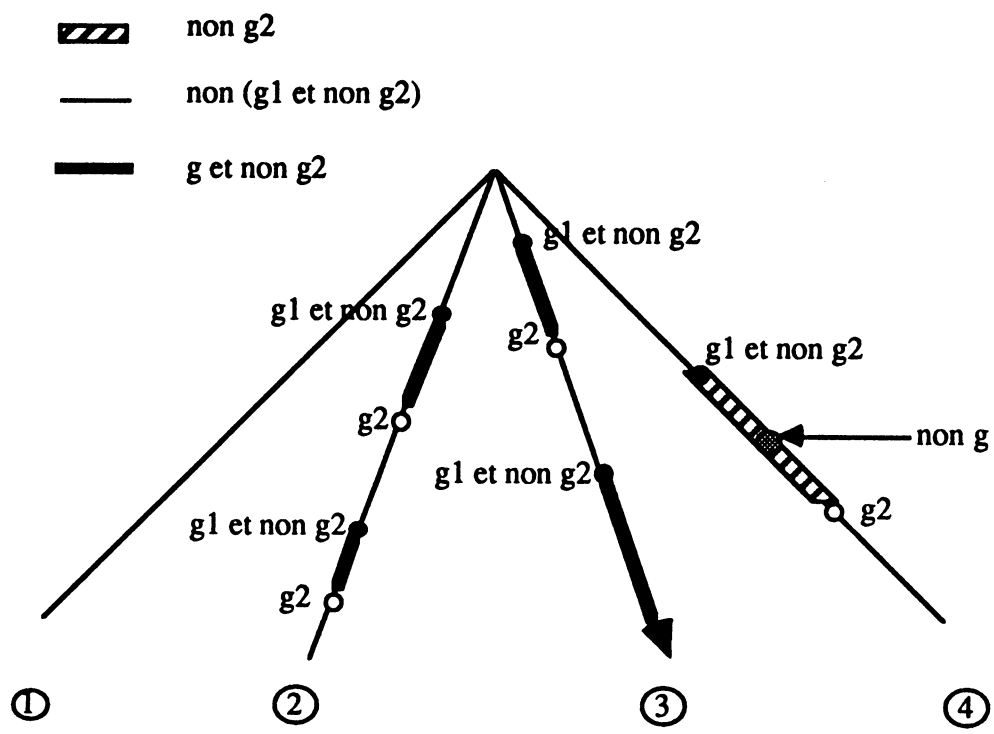



Figure 3.2.2.2.a: toujours cont g entre $g_1$ et $g_2$

L'expression de la sémantique de la propriété *toujours exist* g *entre* $g_1$ *et* $g_2$ a pour première forme:

Pour $q \neq q_{racine}$ :

$q \models_I$ **toujours exist g entre** $g_1$ **et** $g_2$ ssi

$\forall s \in EX(q), \forall k, ($ $(s_k \models_I g_1 \text{ et non } g_2)$ =>

$(\exists k', k'>k \wedge s_{k'} \models_I g_2 \quad \wedge (\exists k'', k'>k''\geq k \Rightarrow s_{k''} \models_I g)$

$\wedge (\forall k''', k'>k'''\geq k \Rightarrow s_{k'''} \models_I \text{non } g_2))$

$\vee ((\forall k', k'\geq k \Rightarrow s_{k'} \models_I \text{non } g_2) \wedge (\exists k'', k''\geq k \Rightarrow s_{k''} \models_I g) \quad )$

Pour $q=q_{racine}$, la définition ne s'applique qu'aux $k\geq 1$.

Cette expression de la sémantique correspond au fait que, intuitivement, pour toute séquence d'exécution, la propriété g doit être vérifiée au moins une fois dans toute portion d'exécution débutant par la propriété $g_1$ et finissant par la propriété $g_2$. C'est-à-dire encore, que, dans toute situation, la propriété $g_1$ provoque au moins une fois l'apparition de la propriété g avant l'apparition de la propriété $g_2$.

La condition $g_2$ de suppression de la propriété g pouvant ne jamais apparaître après l'arrivée de la propriété $g_1$, tout doit aussi se passer comme si la propriété $g_2$ était repoussée à l'infini: la propriété g doit apparaître au moins une fois dans le futur de $g_1$.
Le schéma 3.2.2.b résume, en terme de séquences d'exécution, la sémantique de la propriété *toujours cxist* g *entre* g1 *et* g2 : toute séquence d'exécution de type ④ invalide cette propriété, alors que celles de type ① , ② ou ③ contribuent à la vérifier.

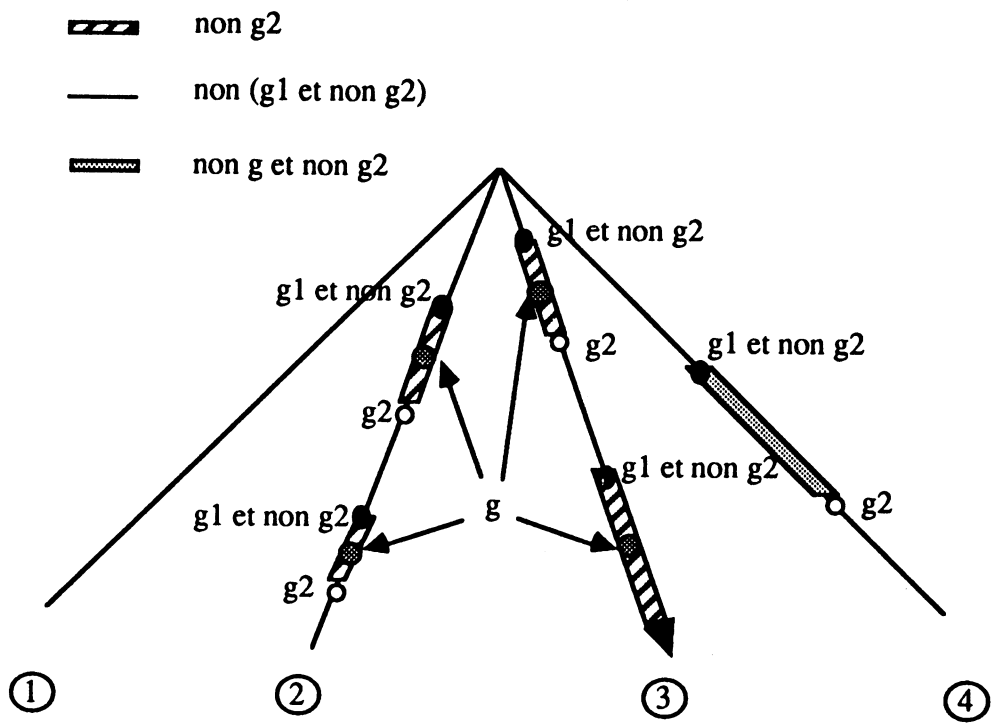



Figure 3.2.2.2.b: toujours exist g entre $g_1$ et $g_2$

La sémantique de la propriété *toujours exist* g *entre* $g_1$ *et* $g_2$ peut aussi être exprimée de manière plus concise (l'équivalence des deux définitions se trouve l'annexe 2.3).
Pour $q \neq q_{racine}$ :

$q \models_I$ **toujours exist g entre** $g_1$ **et** $g_2$ ssi
$\forall s \in EX(q), \forall k, ( \quad (s_k \models_I g_1 \text{ et non } g_2) \Rightarrow$
$( \exists k', k \leq k' \wedge s_{k'} \models_I g \wedge \forall k'', k \leq k'' \leq k' \Rightarrow s_{k''} \models_I \text{non } g_2))$ (8)

Pour $q=q_{racine}$, la même définition est appliquée avec $k \geq 1$.

Les formules duales de *toujours cont* g *entre* $g_1$ *et* $g_2$, *toujours exist* g *entre* $g_1$ *et* $g_2$, c.a.d. *possible exist* g *entre* $g_1$ *et* $g_2$ et *possible cont* g *entre* $g_1$ *et* $g_2$ ont la signification intuitive suivante :

**$q \models_I$ possible cont g entre $g_1$ et $g_2$** ssi

pour une certaine séquence d'exécution, il existe une portion de l'exécution débutant par la propriété $g_1$ et finissant par la propriété $g_2$, sur laquelle la propriété g est continuellement vraie.

**$q \models_I$ possible exist g entre $g_1$ et $g_2$** ssi

pour une certaine séquence d'exécution, il existe une portion de l'exécution débutant par la propriété $g_1$ et finissant par la propriété $g_2$, au sein de laquelle la propriété g est vérifiée au moins une fois.

Nous donnons maintenant la sémantique des deux derniers opérateurs de base : *préservable cont* g *entre* $g_1$ *et* $g_2$ et *préservable exist* g *entre* $g_1$ *et* $g_2$

Pour $q \neq q_{racine}$ :

$q \models_I$ **préservable cont g entre $g_1$ et $g_2$** ssi

$\exists s \in EX(q), \forall k, ( \quad (s_k \models_I g_1 \text{ et non } g_2) \Rightarrow$

$\forall k'', \neg(k'' \geq k \wedge s_{k''} \models_I \text{non } g) \Rightarrow (\exists k', k'' \geq k' \geq k \wedge s_{k'} \models_I g_2) \quad )$

Pour $q=q_{racine}$, la même définition est appliquée avec $k \geq 1$.

La sémantique intuitive de cet opérateur est que, tout au long d'une certaine exécution, la propriété $g_1$ provoque immédiatement l'apparition de la propriété g , qui ne disparait ensuite qu'à l'apparition de la propriété $g_2$.

Pour $q \neq q_{racine}$ :

$q \models_I$ **préservable exist g entre $g_1$ et $g_2$** ssi

$\exists s \in EX(q), \forall k, ( \quad (s_k \models_I g_1 \text{ et non } g_2) \Rightarrow$

$(\exists k', k' \geq k \wedge s_{k'} \models_I g \wedge \forall k'', k' \geq k'' \geq k \Rightarrow s_{k''} \models_I \text{non } g_2) \quad )$

Pour $q=q_{racine}$, la même définition est appliquée avec $k \geq 1$.

La différence entre la sémantique de cet opérateur et celle du précédent est que g n'est tenu d'apparaître qu'une fois entre $g_1$ et $g_2$.

Les propriétés duales ont la sémantique intuitive suivante : un état satisfait la formule *inévitable cont* g *entre* $g_1$ *et* $g_2$ ssi, au moins une fois au cours de toute séquence d'exécution partant de cet état, la propriété g doit être vérifiée continuellement dans une portion d'exécution débutant par la propriété $g_1$ et finissant par la propriété $g_2$,
et, un état satisfait la formule *inévitable exist* g *entre* $g_1$ *et* $g_2$ ssi, au moins une fois au cours de toute exécution au départ de cet état, la propriété $g_1$ provoque l'apparition de la propriété g avant l'apparition de la propriété $g_2$, et ce au moins une fois. Si la propriété $g_2$ n'apparaît jamais dans le futur de $g_1$, alors, la propriété g est tenue d'arriver, mais sans délai imposé.

**Remarque:**

Dans le cas des propriétés de type *toujours* et *préservable*, s'il n'existe pas de sous-séquence déterminée par $g_1$ et $g_2$ pendant les exécutions, la propriété est vraie. En effet, ces propriétés doivent être vues de la façon suivante: elles forcent toute sous-séquence d'exécution déterminée par $g_1$ et $g_2$, existant sur n'importe quelle séquence d'exécution à vérifier une certaine propriété: g.

Par contre, les propriétés duales: de type *possible* et *inévitable*, ne peuvent pas être vraies s'il n'existe pas de sous-séquence déterminée par $g_1$ et $g_2$. En effet elles requièrent l'existence d'une telle sous-séquence et lui imposent de plus de vérifier une certaine propriété : g.

c)

Nous avons défini le langage LEP avec l'objectif de formaliser toutes les notations apparues dans les études de cas. La construction *après... et_avant ...* n'a pas jusqu'à présent été prise en compte. Elle apparaît dans SIREX pour exprimer par exemple la propriété :

*toujours*
(PS-MinBN *équivalent à*
(BN-valide *et*
(*après* (Flux-BN < MinBN *et* BN-actif *et* BN-valide)
*et_avant* (Flux-BN > MinBN+hystérésis))))

Cette propriété peut être réécrite à l'aide des constructions du langage LEP et des notations définies, de la façon suivante:

*jamais* PS-MinBN *sans* BN-valide
et *toujours continuellement* (non PS_minBN)
*entre initial*
*et* ((Flux-BN<MinBN) et BN-actif et BN-valide)

et *toujours continuellement* (non PS_minBN)
*entre* (Flux-BN>MinBN+hystérésis)
*et* ((Flux-BN<MinBN) et BN-actif et BN-valide)
et *toujours continuellement* (non BN-valide ou PS-MinBN)
*entre* ((Flux-BN<MinBN) et BN-actif et BN-valide)
*et* (Flux-BN>MinBN+hystérésis)

Cette traduction est intuitivement compréhensible, lorsqu'on se souvient que les intervalles désignés par l'expression "entre $g_1$ et $g_2$" débutent en un état quelconque satisfaisant $g_1$ et finissent par le **premier** état satisfaisant $g_2$ qui lui succède.

Les trois premiers termes correspondent à:

*toujours*
(PS-MinBN *implique*
(BN-valide *et*
(*après* (Flux-BN < MinBN *et* BN-actif *et* BN-valide)
*et_avant* (Flux-BN > MinBN+hystérésis))))

et le dernier à:

*toujours*
((BN-valide *et*
(*après* (Flux-BN < MinBN *et* BN-actif *et* BN-valide)
*et_avant* (Flux-BN > MinBN+hystérésis))))
*implique* PS-MinBN)

L'expression *après... et_avant ...* n'est jamais apparue dans d'autres contextes d'opérateurs que celui de la propriété ci-dessus. On voit donc que cette expression n'a pas besoin d'être incorporée à LEP, pour que le schéma d'imbrication des constructions :
*toujours* (... *équivalent à* (... *et après ... et_avant ...*)),
soit exprimable.
Nous avons alors choisi de ne pas définir *après... et_avant ...*, seul, comme opérateur de LEP.

### 3.2.2.3 Satisfaction d'une formule par un programme

Un programme vérifie une formule du langage si son arbre des exécutions est un modèle de la formule :
Si A est un arbre tel que (A,J) est l'arbre d'exécution d'un programme LUSTRE Pr,
Si P est tel que : P=B
où B est l'ensemble des expressions booléennes sur les variables de Pr,

Pr vérifie la formule f (et on note Pr $\models$ f ) ssi
$\exists$I telle que A,I $\models$ f, et I est telle que :
$\forall p \in B$, $\forall s \in EX(q_{racine})$, $\forall i \in \mathbb{N}$, $s_i \in I(p)$ ssi $J(s_0)...J(s_i) \vdash p$ : vrai

Un programme LUSTRE vérifie une formule f ssi il existe une interprétation de son arbre, compatible avec l'exécution, qui rende cet arbre modèle de f.

## 3.3 CONCLUSION

En étudiant différentes applications dans le domaine des systèmes réactifs de sûreté, nous avons extrait l'allure générale des opérateurs nécessaires à l'expression des propriétés de ces logiciels. Nous pouvons faire les remarques suivantes concernant ces études de cas :
• Les constructions utilisées pour l'expression des propriétés sont en général temporelles.
• Elles sont en nombre restreint.
• La plupart des propriétés sont des invariants temporels ("safety properties").
• La méthode de spécification influence le mode d'expression.
• La spécification est indépendante de l'affinement de la conception SAGA.

A partir de ces résultats, nous avons défini un langage formel pour l'expression des propriétés : LEP. Nous allons essayer de traduire les opérateurs de LEP suivant différents langages de spécification existants et de faire le bilan des possibilités de vérification qui nous sont alors offertes.

Le langage LEP ayant de fortes ressemblances avec les logiques temporelles, nous avons privilégié l'approche de spécification basée sur les logiques, plutôt que sur des machines abstraites.

Dans les chapitres suivants, nous nous intéressons à la vérification de propriétés exprimées en LEP sur des programmes LUSTRE ( et par cet intermédiaire sur des programmes SAGA). Cette vérification sera réalisée par des méthodes d'évaluation sur des modèles.

Ces méthodes consistent à générer un modèle du programme à vérifier, et à évaluer sur ce modèle les propriétés requises pour ce programme.

Cette approche de vérification a été mise en œuvre pour des propriétés décrites à l'aide de logiques temporelles et de $\mu$-calculs propositionnels. Nous étudions dans le chapitre 4 la possibilité d'adapter les résultats de ces travaux à la vérification de propriétés en LEP sur des programmes LUSTRE. Pour cela, nous traduirons les formules du langage LEP à l'aide d'une logique

temporelle arborescente et d'un μ-calcul propositionnel, et nous décrirons l'utilisation de logiciels réalisant la validation de formules décrites dans ces langages.

# 4 Logique temporelle et μ-calcul pour la vérification de propriétés

Les logiques utilisées pour la vérifications de systèmes réactifs sont en général des logiques temporelles [RU,71]. D'autres travaux concernent des extensions de la logique de Hoare [Hoa,69] aux réseaux de processus [MC,81] [ZRE,85].

[GPSS,80] ayant montré que, dans le cas d'un passé borné, toute logique temporelle peut être réduite à son fragment futur pur sans perte d'expressivité, les logiques temporelles utilisées sont principalement des logiques du futur.

Les logiques temporelles sont divisées en deux grandes catégories : les logiques linéaires et les logiques arborescentes. Les premières considèrent l'exécution d'un programme selon un axe temporel unique passé -> présent -> futur. Le présent représente ce qui est en train d'avoir lieu, le passé ce qui a eu lieu, et le futur ce qui aura **effectivement** lieu.

Les opérateurs communément utilisés dans la logique du temps linéaire incluent :

- $F_p$ (quelques fois p),
- $G_p$ (toujours p),
- $X_p$ (la prochaine fois p)
- [pUq] (p jusqu'à ce que q)

(par exemple L(X,F,U) de [GPSS,80])

Les logiques arborescentes modélisent, par contre, le temps sous forme d'un arbre. Cet arbre permet de prendre en compte, à partir du premier instant du déroulement du programme, toutes les exécutions qui **pourraient** avoir lieu. L'exécution effective du programme correspond au déroulement d'une branche de l'arbre.

Les modalités de base dans les logiques du temps arborescent sont généralement de la forme: A (pour tout futur) ou E (pour un certain futur), suivi d'une combinaison des opérateurs usuels de la logique temporelle linéaire ; F,G,X et U (par exemple CTL* de [EH,86], CL de [QS,83]).

Dans le domaine de la vérification de formules de logique sur des programmes, deux approches sont généralement envisagées.

La première approche consiste à déduire les propriétés du programme à partir d'un ensemble d'axiomes et de règles d'inférence. Cet ensemble est généralement divisé en trois parties. La première partie permet de déduire des théorèmes universellement vrais, ses axiomes et ses règles d'inférence décrivent les propriétés des opérateurs du langage. La deuxième partie permet de prouver des théorèmes relatifs à des domaines particuliers, par exemple des théorèmes sur $\mathbb{N}$. La troisième partie décrit le programme à l'aide d'axiomes, ses règles d'inférence permettent de déduire les propriétés particulières des exécutions de ce programme. Cette approche a été utilisée pour la vérification de propriétés décrites à l'aide de logiques temporelles : linéaires [MP,82] et arborescentes [GS,86], [BMP,83]. Misra et Chandy [MC,81] ont défini le premier système de preuve modulaire pour une logique à la Hoare. Ce système permet de déduire la spécification d'un réseau de processus à partir des spécifications de ses composants. On trouve dans le même domaine les travaux de [ZRE,85].

La deuxième approche ("model checking") réalise la vérification par évaluation de formules sur des modèles. Elle définit un modèle des programmes, en général un graphe d'états muni d'une interprétation. Elle définit ensuite une relation de satisfaction reliant les formules de la logique et les modèles. Une propriété est vérifiée par un programme, si elle est satisfaite par le modèle du programme. Cette approche aussi, a été utilisée dans le cadre des logiques temporelles, surtout arborescentes [QS,82], [EL,85].

Nous avons choisi pour notre travail, de réaliser la validation de programmes LUSTRE, et donc SAGA, par la méthode d'évaluation sur des modèles.

Emerson et Lei [EL,85] ont montré que les algorithmes d'évaluation sont toujours meilleurs en logique temporelle arborescente qu'en logique temporelle linéaire : pour un algorithme d'évaluation donné, il existe un algorithme d'évaluation du même ordre de complexité (en taille à la fois de la structure représentant le modèle et de la taille de la formule) dans la logique temporelle arborescente correspondante (ajout des opérateurs arborescents A ou E, aux opérateurs linéaires) dont le pouvoir d'expression englobe celui de la logique temporelle linéaire.

Nous avons donc choisi de nous intéresser aux logiques temporelles arborescentes (plutôt qu'aux linéaires) pour lesquelles il existe des procédures d'évaluation. Notre but est d'exprimer à l'aide d'une de ces logiques, les propriétés des systèmes temps réel réactifs programmés en LUSTRE. Nous désirons donc que le langage LEP défini au chapitre 2 soit traduisible dans la logique choisie.

Nous nous sommes aussi intéressés aux μ-calculs propositionnels [Koz,83] car ils permettent d'étendre le pouvoir d'expression des logiques (Lμ de [EL,86] pour CTL*, STL [GS,86] pour CL).

Dans ce chapitre, nous tentons de traduire toute formule du langage LEP suivant le langage CL [QS,83] dans un premier temps, suivant le langage μ-CL dans un deuxième temps. Le langage CL est une logique temporelle arborescente; il ne permet pas d'exprimer toutes les formules de LEP. Le langage μ-CL est un μ-calcul propositionnel étendant CL.

Nous évoquons ensuite une expérience [Rat, 88] de validation de propriétés décrites dans le μ-calcul propositionnel STL [GS,86] sur des programmes SAGA ou LUSTRE représentés par leur automate de contrôle, à l'aide du logiciel XESAR [RRSV,87b], [Ro,88]. Le langage μ-CL peut être vu comme un cas particulier de STL.

## 4.1 LA LOGIQUE TEMPORELLE ARBORESCENTE CL

La logique temporelle CL [QS,83] est composée des opérateurs du calcul propositionnel ainsi que des opérateurs temporels *pot* et *inev* (et de leurs duaux *al* et *some* ). Les opérateurs temporels sont définis sur des graphes d'états. Une logique équivalente a été définie dans [Abr,79] et étudiée dans [CE,81] où une procédure de décision est fournie.

### 4.1.1 Syntaxe:

Soit P un ensemble de variables propositionnelles, $p \in P$, f, $f_1$, $f_2$, g des formules bien formées du langage :

f ::= **true** | **init** | p | $f_1 \vee f_2$ | $\neg$ f | **next** f | **pot** [g] (f) | **inev** [g] (f) .

Originellement la syntaxe de l'opérateur *next* est *pre* pour "précondition". Nous avons préféré la modifier afin d'éviter toute confusion avec l'opérateur LUSTRE du même nom.
On emploie les notations suivantes:

**Next** (f) = $\neg$next ($\neg$f)
**al** [g] (f) = $\neg$ pot [g] ($\neg$ f)
**some** [g] (f) = $\neg$ inev [g] ($\neg$ f)

### 4.1.2 Description des modèles

Un **graphe d'états** (ou système de transitions) est un triplet $(E, E_i, \rightarrow)$ où

- E est un ensemble dénombrable d'états,
- $E_i \subseteq E$, est l'ensemble des états initiaux,
- et $\rightarrow$ est une relation binaire sur E ( $\rightarrow \subseteq ExE$) dite relation de transition;

Soient e et e', deux états de E, nous noterons $e \rightarrow e'$ au lieu de $(e,e') \in \rightarrow$; e' est dit **successeur** de e.

Remarque: aucune hypothèse n'est faite sur la relation de transition: notamment, un état peut avoir plusieurs successeurs, et même aucun.

**Une séquence d'exécution** partant d'un état $e_0$ est une séquence d'états $s = e_0, e_1, \ldots, e_i, e_{i+1}, \ldots$ telle que $\forall i$, $e_i \rightarrow e_{i+1}$ et qui est maximale, c'est-à-dire: si cette séquence est finie, son dernier élément n'a pas de successeur.

On appelle EX(e) l'ensemble des séquences d'exécution à partir de e. On note $s_k$ le $(k+1)^{ième}$ élément de la séquence s, s'il existe.

Remarquons que la différence entre un arbre (cf § 2.2.1) et un système de transitions, réside essentiellement dans la relation de transition. Dans un système de transitions, aucune hypothèse n'est faite sur l'unicité du passé de tout état. Remarquons aussi qu'un système de transitions peut avoir plusieurs états sans prédécesseurs, alors qu'un arbre n'en a qu'un: l'état racine.

Un **modèle** pour une formule f de la logique est un système de transitions $G = (E, E_i, \rightarrow)$ muni d'une fonction d'interprétation $I: E \rightarrow 2^P$, qui associe à chaque état un ensemble de variables propositionnelles, tels que : $\forall e \in E,\ e \models_I f$.

On dit alors que la formule f est vraie sur (satisfaite par) le graphe G sous l'interprétation I, et on note $G \models_I f$.

Nous allons définir la relation de satisfaction $\models_I$ entre un état et une formule de la logique suivant l'interprétation I.

### 4.1.3 Sémantique

| | | |
|---|---|---|
| $e \models_I$ true | $\forall e \in E$ | |
| $e \models_I$ init | ssi | $e \in E_i$ |
| $e \models_I p$ | ssi | $p \in I(e)$ |
| $e \models_I f1 \vee f2$ | ssi | $e \models_I f1$ ou $e \models_I f2$ |
| $e \models_I \neg f$ | ssi | $e \not\models_I f$ |
| $e \models_I$ next (f) | ssi | $\exists e', e \rightarrow e' \wedge e' \models_I f$ |
| $e \models_I$ pot [g] (f) | ssi | $\exists s \in EX(e), \exists k, s_k \models_I f$ $\wedge (\forall n, 1 \leq n < k \Rightarrow s_n \models_I g)$ |
| $e,I \models_I$ inev [g] (f) | ssi | $\forall s \in EX(e), \exists k, s_k \models_I f$ $\wedge (\forall n, 1 \leq n < k \Rightarrow s_n \models_I g)$ |

### 4.1.4 Traduction du langage d'expression des propriétés en CL

La traduction est possible du fait qu'un arbre est un système de transitions particulier. En effet, soit A = (Q, $q_{racine}$, $\rightarrow$), alors A est un graphe d'états dont l'ensemble des états est Q, dont l'ensemble des états initiaux est réduit à $q_{racine}$, et dont la relation de transition est $\rightarrow$. Ainsi, les opérateurs de la logique CL sont définis sur les arbres.

$T_{la}$ permet de traduire les formules ci-dessous de LEP, exprimées en un état, dans la logique temporelle arborescente CL par :

$T_{la}$ (vrai) = true

$T_{la}$ (initial) = initial où initial est une variable propositionnelle de P telle que :

$q \in I(initial)$ ssi $q_{racine} \rightarrow q$

$T_{la}$ (P) = P

$T_{la}$ (g ou g') = $T_{la}$ (g) $\vee$ $T_{la}$ (g')

$T_{la}$ (non g) = $\neg T_{la}$ (g)

$T_{la}$ (toujours cont g entre $g_1$ et $g_2$) =

al ( $\neg$init $\wedge T_{la}$ ($g_1$)$\wedge\neg T_{la}$ ($g_2$) => al [$\neg T_{la}$ ($g_2$)] ($T_{la}$ (g)$\vee T_{la}$ ($g_2$)) )

La précision $\neg$init permet à $q_{racine}$ de ne pas intervenir dans la vérification de la propriété.

$T_{la}$ (toujours exist g entre $g_1$ et $g_2$) =

al ( $\neg$init $\wedge T_{la}$ ($g_1$)$\wedge\neg T_{la}$ ($g_2$) => inev [$\neg T_{la}$ ($g_2$)] ($T_{la}$ (g)$\wedge\neg T_{la}$ ($g_2$)) )

Cette logique ne permet pas d'exprimer la construction préservable exist, ni sa duale. En effet, les deux opérateurs permettant de désigner un chemin dans le graphe d'états sont *pot* et *some*. Pour réaliser le *préservable exist* avec l'opérateur *some*, la logique CL étant une logique du futur, il est nécessaire de se placer à l'aide de cet opérateur sur le début des intervalles auxquels on s'intéresse, c'est-à-dire : $[g_1 \wedge \neg g_2, g_2[$. Il est alors impossible d'utiliser un des opérateurs de la logique pour spécifier que les propriétés g doivent apparaître sur la même branche que tous les $g_1 \wedge \neg g_2$ décrits par l'opérateur *some*. Un autre problème se pose au niveau de l'opérateur *pot* : il faudrait combiner avec cet opérateur un mécanisme de récurrence. En effet, plusieurs occurrences de l'opérateur *pot* permettent de se placer de proche en proche sur un état vérifiant $g_1 \wedge \neg g_2$, puis un état vérifiant g, puis un état vérifiant $g_2$. Il est possible d'exprimer le fait qu'il n'y a pas eu, en g, d'occurrence de $g_2$ depuis $g_1 \wedge \neg g_2$, et en $g_2$, qu'il n'y a pas eu d'occurrence de $g_2$ depuis g. Le problème est alors d'exprimer une récurrence potentiellement **infinie**, qui reboucle sur un état vérifiant $g_1 \wedge \neg g_2$ après l'état vérifiant $g_2$. Cette récurrence n'est pas exprimable en CL.

La logique CL permet d'exprimer la restriction suivante de l'opérateur *préservable exist* : préservable exist g entre true et false = préservable g. Cet opérateur est similaire à celui de la logique classique exprimant la modalité de préservabilité d'une propriété, c'est-à-dire l'équivalent de l'opérateur *some*. La traduction proposée est la suivante :

$T_{la}$ (préservable g) = some ( $\neg$init $\Rightarrow T_{la}$ (g) )

La logique CL est une logique du futur et ne permet pas de traduire la construction *au cycle précédent(g)* de façon automatique. Néanmoins, certaines des propriétés faisant intervenir cet opérateur peuvent être traduites à l'aide de l'opérateur *next*. La traduction est réalisée en suivant l'intuition du spécificateur (et prouvée à posteriori). Par exemple la propriété :

toujours ( g $\equiv$ au cycle précédent (g'))

peut se traduire par :

al ( Next($T_{la}$ (g))$\equiv T_{la}$ (g') )

Tout état dont tous les successeurs (s'il en existe) vérifient g, vérifie g', et réciproquement.

Une formule du langage d'expression des propriétés est satisfaite par un arbre d'exécution ssi elle est satisfaite par son état initial. Par ailleurs, une formule de CL est satisfaite par un graphe d'états si et seulement si elle est satisfaite en tout état du graphe. La traduction du langage d'expression des propriétés doit prendre en compte cette différence lors de la traduction d'une propriété à satisfaire par l'arbre des exécutions. Ceci sera réalisé de la façon suivante :

A,I $\models$ f ssi G,I $\models$ init $\Rightarrow T_{la}$ (f).

## 4.2 LE μ-CALCUL PROPOSITIONNEL μ-CL

Nous présentons ici un μ-calcul propositionnel cas particulier de celui de [Koz,83]. La particularisation est due au fait que les transitions des modèles de programmes LUSTRE ne sont pas étiquetées par des noms d'actions. Le tirage d'une transition est alors représenté par un unique opérateur *next*, et non par un opérateur pour chaque action ou ensemble d'actions.

### 4.2.1 Le langage

**Syntaxe :**

Soit P un ensemble de variables propositionnelles et X un ensemble de variables. Les formules de μ-CL (ensemble F) sont définies de la façon suivante :

$f ::= \mathbf{true} \mid p \mid x \mid f_1 \vee f_2 \mid \neg f \mid \mathbf{init} \mid \mathbf{next}\ f \mid \mu x.f(x)$ .

où $p \in P$, $x \in X$, $f_1$ et $f_2$ sont des formules, et $\mu x.f(x)$ est tel que toute occurrence de x dans f(x) est imbriquée sous un nombre pair de négations.

On définit les abréviations suivantes :

**Next** $f = \neg$ next $\neg f$

**Succes** $f =$ next $f \wedge$ Next f

$\nu x.f(x) = \neg\ \mu x.\neg f(x)$

**Sémantique :**

Les modèles de ce μ-calcul propositionnel sont des graphes d'états $G = (E, \{i_G\}, \rightarrow)$ muni d'une fonction d'interprétation I.
La fonction d'interprétation $I : E \rightarrow 2^P$, associe à chaque état un ensemble de variables propositionnelles.

Soit f une formule du langage ayant comme variables libres (c.a.d. qui ne sont pas sous la portée d'un opérateur μ ou ν) : $\vec{X} = \{x_1, x_2, \ldots, x_n\}$. On écrit $f(\vec{X})$ pour indiquer que toutes les variables libres de f se trouvent dans $\vec{X}$. Une valuation $\vec{V} = \{v_1, v_2, \ldots, v_n\}$ est une affectation de sous-ensembles de E aux variables libres $\vec{X}$.

Un graphe G = (E, {$i_G$}, $\rightarrow$) est un modèle pour une formule f de F ssi :

$\forall\, \vec{\nu},\ i_G, \vec{\nu} \models_I f.$

où la relation de satisfaction $\models_I$ sur E x $(2^E)^n$ x F est définie comme suit:

| | | |
|---|---|---|
| $e, \vec{\nu} \models_I$ true | $\forall e \in E$ | |
| $e, \vec{\nu} \models_I p$ | ssi | $p \in I(e)$ |
| $e, \vec{\nu} \models_I x_i$ | ssi | $e \in \nu_i$ |
| $e, \vec{\nu} \models_I f1 \vee f2$ | ssi | $e, \vec{\nu} \models_I f1$ ou $e, \vec{\nu} \models_I f2$ |
| $e, \vec{\nu} \models_I \neg f$ | ssi | $e, \vec{\nu} \not\models_I f$ |
| $e, \vec{\nu} \models_I$ init | ssi | $e = i_G$ |
| $e, \vec{\nu} \models_I$ next f | ssi | $\exists e' \in E,\ e \rightarrow e'$ et $e', \vec{\nu} \models_I f$ |
| $e, \vec{\nu} \models_I \mu x.f(x, \vec{X})$ | ssi | $e \in \cap \{E' \subseteq E / \{e' \in E / e', (E', \vec{\nu}) \models_I f(x, \vec{X})\} \subseteq E'\}$ |

L'opérateur $\mu$ est l'opérateur de plus petit point fixe sur le treillis $(2^E, \subseteq)$. Nous allons fournir une explication de sa sémantique.

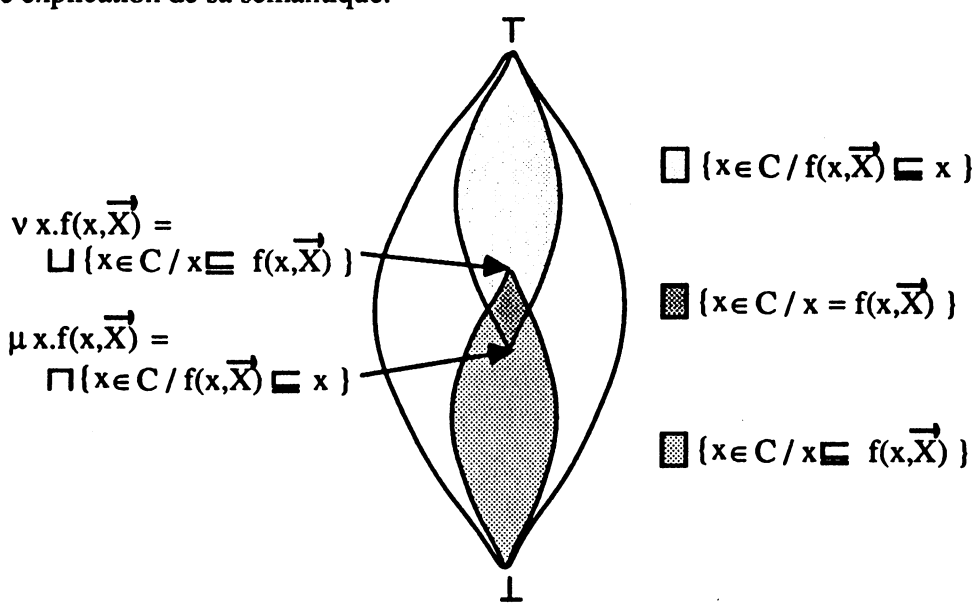



Figure 4.2.1.a : Fonction f totale monotone par rapport à x sur le treillis complet (C, $\sqsubseteq$)

On appelle **treillis complet** un couple (C, $\sqsubseteq$ ) où

- C est un ensemble,
- $\sqsubseteq$ est un ordre partiel

tel que $\forall$ F $\subseteq$ C, F admet une borne supérieure $\sqcup$F et une borne inférieure $\sqcap$F.
En particulier C possède un plus petit élément $\perp$ et un plus grand élément $\top$.

**Théorème de Tarski** (cf figure 4.2.1.a)

Si f est une fonction totale monotone d'un treillis complet dans lui même, alors f admet un ensemble non vide de points fixes et cet ensemble a un plus petit élément : $\mu.f$

Dans le cas des formules de $\mu$-CL, le treillis complet considéré est $(2^E, \subseteq)$ où $\subseteq$ est l'opérateur d'inclusion des ensembles.

Pour une valuation $(E', \vec{V})$, on définit :

$$\|f\| : 2^E \times (2^E)^n \rightarrow 2^E$$
$$(E', \vec{V}) \rightarrow \{ e \in E \ / \ e,(E', \vec{V}) \models f(x, \vec{X}) \}$$

$\|f\|(E', \vec{V})$ est appelé ensemble caractéristique de la formule $f(x, \vec{X})$ pour la valuation $(E', \vec{V})$.

D'après la sémantique des formules f des formules f de $\mu$-CL, $\|f\|$ est une fonction totale, monotone par rapport à son premier paramètre x, sur $(2^E, \subseteq)$. Elle admet donc un plus petit point fixe noté $\|\mu x.f\|$ qui, pour une valuation $\vec{V}$ donnée, est égal à

$$\cap \{x \in 2^E / \|f\|(x, \vec{V}) \subseteq x \}$$
$$= \cap \{E' \subseteq E / \|f\|(E', \vec{V}) \subseteq E' \}$$
$$= \cap \{E' \subseteq E / \{e \in E / e,(E', \vec{V}) \models f(x, \vec{X}) \} \subseteq E' \}$$

La sémantique de la formule $\mu x.f(x, \vec{X})$ est telle que son ensemble caractéristique est $\|\mu x.f\|$.

### 4.2.2 Traduction du langage d'expression des propriétés en $\mu$-calcul

Ce $\mu$-calcul permet d'exprimer toutes les constructions définies au paragraphe 2.2. En effet, d'une part, tous les opérateurs temporels de la logique CL, sont traduisibles à l'aide des opérateurs du $\mu$-calcul (cas particulier de la traduction en STL [Ro,88]) :

- true, init, $\vee$, $\neg$, et next sont identiques,
- on choisit le même ensemble de variables propositionnelles,
- pot [f] (g) = $\mu x. (g \vee (f \wedge \text{next } x)$
- inev [f] (g) = $\mu x. (g \vee (f \wedge \text{Succes } x)$

et par conséquent, les constructions définies au paragraphe 2.2 dont la traduction peut être réalisée en CL, sont exprimables en $\mu$-calcul.

D'autre part, nous avons traduit les constructions restantes à l'aide de l'opérateur de traduction $T_{mu}$ par :

**1) en $q_{racine}$**

$T_{mu}$ (inévitable cont g entre $g_1$ et $g_2$) =

$\mu x\ [\text{Succes}\ \mu x_1[\ \ ((\neg g1 \vee g2 \vee \neg g) \wedge \text{Succes}\ x_1)$

$\vee\ (g1 \wedge \neg g2 \wedge g \wedge \text{Succes}\ \forall x_2(\ \ (g \wedge \neg g2 \wedge \text{Succes}\ x_2)$

$\vee\ g2$

$\vee\ (\neg g \wedge \neg g2 \wedge \text{Succes}\ x_1)\ ))\ ]]$

$T_{mu}$ (inévitable exist g entre $g_1$ et $g_2$) =

$\mu x\ [\text{Succes}\ \mu x_1[\ \ ((\neg g1 \vee g2) \wedge \text{Succes}\ x_1)$

$\vee\ (g1 \wedge \neg g2 \wedge g)$

$\vee\ (g1 \wedge \neg g2 \wedge \neg g \wedge \text{Succes}\ \mu x_3\ (\ \ (g2 \wedge \text{Succes}\ x_1)$

$\vee\ (g \wedge \neg g2)$

$\vee\ (\neg g \wedge \neg g2 \wedge \text{Succes}\ x_3)\ ))\ \ \ ]]$

$T_{mu}$ (préservable cont g entre $g_1$ et $g_2$) =

$\mu x\ [\text{next}\ \mu x_1[\ \ ((\neg g1 \vee g2) \wedge \text{next}\ x_1)$

$\vee\ (g1 \wedge \neg g2 \wedge g \wedge\ \text{next}\ \mu x_2\ (\ \ (g2 \wedge \text{next}\ x_1)$

$\vee\ (g \wedge \neg g2 \wedge \text{next}\ x_3)\ \ \ \ ))\ \ ]]$

$T_{mu}$ (préservable exist g entre $g_1$ et $g_2$) =

$\mu x\ [\text{next}\ \forall x_1[\ \ ((\neg g1 \vee g2 \vee g) \wedge \text{next}\ x_1)$

$\vee\ (g1 \wedge \neg g2 \wedge \neg g \wedge\ \text{next}\ \mu x_2\ (\ \ (g \wedge \neg g2 \wedge \text{next}\ x_1)$

$\vee\ (\neg g \wedge \neg g2 \wedge \text{next}\ x_3)\ \ \ ))\ \ ]]$

**2) en $q \neq q_{racine}$**

la traduction commence au niveau du $\mu x_1$ pour les trois premières formules et du $\forall x_1$ pour la dernière. Cela correspond dans la définition des opérateurs à la suppression de la condition $k \geq 1$.

Nous avons montré en annexe 6 que ces traductions sont correctes. La démarche suivie comporte plusieurs étapes. Premièrement, la propriété linéaire qui se trouve sous la portée de $\forall s \in EX(q_{racine})$, pour les propriétés de style *préservable*, ou de $\exists s \in EX(q_{racine})$, pour les propriétés de style *inévitable*, est représentée sous forme d'un automate de Rabin [Rab,69]. Celui-ci est ensuite décrit à l'aide d'un système d'équations linéaires aux points fixes [Par,81], lui-même

transformé en un système d'équations arborescentes. On extrait la formule de μ-calcul de ce système.
Remarquons que le problème du *au cycle précédent* se pose dans les mêmes termes que pour la logique CL.

## 4.3 VALIDATION À L'AIDE DE XESAR

XESAR [Ro, 88] est un outil de validation de programmes parallèles. Il permet de vérifier des propriétés sur ces programmes. Il est développé au Laboratoire de Génie Informatique de Grenoble. Bien que sa vocation première soit la validation de protocoles de communication et d'algorithmes distribués, l'utilisation d'un μ-calcul propositionnel en tant que langage d'expression des propriétés permet son utilisation pour la validation de systèmes de contrôle-commande temps réel. Ce μ-calcul est le langage STL. Nous évoquons ici ses différences avec le μ-calcul μ-CL :

La syntaxe des formules fait intervenir un ensemble A d'actions. *next* f n'appartient plus au langage de formules, mais est remplacé par deux autres types de formules faisant intervenir les actions :

$f ::= \mathbf{B}.f \mid \mathbf{B_1}.f_1 + \mathbf{B_2}.f_2$

où f, $f_1$ et $f_2$ sont des formules et B, $B_1$ et $B_2 \subseteq A$.

On définit l'abréviation suivante :

$\langle \mathbf{B} \rangle f = B.f + A.true$

Les modèles des formules de ce μ-calcul propositionnel sont des graphes d'états étiquetés $G = (E, \{i_G\}, \rightarrow)$ muni d'une fonction d'interprétation I. Le graphe d'états étiqueté diffère du graphe d'états présenté pour la logique CL et le μ-calcul μ-CL, par le fait que la relation de transition $\rightarrow$ est une union de relations de transitions étiquetées par des noms d'actions :

$\rightarrow = \{\rightarrow_a\}_{a \in A}$

avec chacune des relations $\rightarrow_a \subseteq ExE$.

La sémantique des deux nouveaux opérateurs est la suivante :

$e, \vec{\forall} \models B.f$ ssi $\exists b \in B, \exists e' \in E, e \rightarrow_b e'$,

et $\forall a \in A, \exists e' \in E, e \rightarrow_a e' \Rightarrow (b \in B$ et $e', \vec{\forall} \models f)$

$e, \vec{\nabla} \models B_1.f_1 + B_2.f_2$ ssi $\exists a_1 \in B_1, \exists e_1 \in E, e \rightarrow_{a1} e_1,$

et $\exists a_2 \in B_2, \exists e_2 \in E, e \rightarrow_{a2} e_2,$

et $\forall a \in A, \exists e' \in E, \; e \rightarrow_a e' \Rightarrow$

$(a \in B_1$ et $e', \vec{\nabla} \models f_1)$ ou $(a \in B_2$ et $e', \vec{\nabla} \models f_2)$

Remarquons que l'opérateur next de μ-CL peut être défini en STL :

**next** f = <A> f

L'approche étudiée pour la validation de programmes LUSTRE à l'aide de XESAR [Rat,88], définit les expressions LUSTRE comme les prédicats de base du langage STL, puisque le rôle des prédicats de base dans de tels langages est d'assurer la liaison entre les propriétés et les entités du programme.

Les programmes en entrée de XESAR sont décrits dans le langage ESTELLE\R, une variante du langage ESTELLE [Est,85]. ESTELLE (extension de PASCAL) a été conçu pour la spécification de protocoles et de leurs services sous la forme d'un système de composantes séquentielles, structurées de manière hiérarchique. Les composantes du système communiquent via des liens bidirectionnels établis entre des ports, appelés points d'interaction. ESTELLE\R garde les mêmes principes pour l'organisation du programme de description: celui-ci est décrit comme un système d'automates étendus, c'est-à-dire des automates tels que des actions, décrites par des blocs d'instructions, sont associées aux transitions.

L'approche utilisée dans XESAR est l'évaluation des propriétés sur un graphe d'états fini représentant le programme ESTELLE\R. Ce graphe d'états est obtenu en générant un ensemble d'automates séquentiels communicants à partir du système de processus défini par la description ESTELLE\R, puis en composant ces automates par couplage des paires émission-réception correspondant à un point d'interaction, et enfin en simulant l'automate obtenu de manière exhaustive.

L'utilisation de XESAR pour la vérification des propriétés sur des programmes LUSTRE a été étudiée dans [Rat,88]. Voici les principaux points que l'on peut dégager de cette étude :

1) L'utilisation de XESAR pour vérifier des propriétés sur un programme LUSTRE nécessite la traduction de ce programme en ESTELLE\R car un graphe d'états ne peut être fourni directement en entrée de XESAR.

Les formalismes sous-jacents à LUSTRE et ESTELLE\R sont totalement disjoints:

• LUSTRE s'inscrit dans l'approche déclarative alors qu'ESTELLE\R est un langage impératif,

- LUSTRE est muni d'une interprétation synchrone et ESTELLE\R d'une interprétation asynchrone.

La traduction d'un programme LUSTRE en ESTELLE\R est alors réalisée via l'automate de contrôle du programme LUSTRE. Cet automate est obtenu grâce au compilateur LUSTRE (cf §2.2.3) et représente un programme séquentiel sémantiquement équivalent au programme LUSTRE.

La description ESTELLE\R obtenue comporte un processus unique décrivant l'automate de contrôle.

2) L'évaluation des propriétés sur le graphe d'états généré par XESAR doit être telle que : si f est une formule bien formée satisfaite par l'arbre des exécutions du programme LUSTRE et f' sa traduction en STL, XESAR valide f'.

Or le graphe d'états généré par XESAR est différent de l'arbre des exécutions du programme. Il comporte des états supplémentaires par rapport au repliage de l'arbre sous forme d'un graphe d'états LUSTRE. Ce qui a deux conséquences principales :

- d'une part un accroissement inutile du nombre d'états du graphe. Or, l'un des problèmes de l'approche de vérification de propriétés par évaluation sur des modèles est "l'explosion" du nombre d'états du graphe, qui ralentit l'évaluation proportionnellement à l'accroissement du nombre d'états, et qui empêche parfois la génération complète du graphe par manque de place mémoire.
- d'autre part, la vérification d'une propriété sur le graphe XESAR ne correspond pas à la vérification de la même propriété sur l'arbre des exécutions. Par conséquent, il faut définir un opérateur de traduction $T_X$ sur STL tel que :

  $\forall f \in STL,\ A \models f$ ssi $G \models T_X(f)$

  où A et G sont respectivement l'arbre d'exécution et le graphe d'états XESAR d'un même programme, et la relation de satisfaction $\models$ est définie pareillement sur les états du graphe et sur les états de l'arbre des exécutions.

  De plus, une modification du graphe XESAR via l'automate ESTELLE\R est nécessaire : les états du graphe correspondant à l'arbre des exécutions vont être repérés grâce à la notion de "label" définie dans XESAR. Les labels sont des étiquettes placées dans les programmes auxquelles on peut faire référence dans les propriétés. Ces labels sont modélisés sur le graphe d'états par les actions sur les transitions. Un label particulier va être introduit afin de caractériser l'état du graphe correspondant aux états initiaux de l'arbre des exécutions et permettre l'évaluation du prédicat *init*. Les labels servant à repérer les états du graphe appartenant au graphe quotient de l'arbre d'exécution du programme LUSTRE, vont être introduits sur les transitions lors de la

traduction en ESTELLE\R. Les formules de STL sont alors transformées par $T_X$ afin de porter uniquement sur les états repérés par les labels.

3) Enfin, la notion temporelle du μ-calcul propositionnel fait référence au futur, alors que celle de LUSTRE fait référence au passé. Aussi, si le graphe d'état est une représentation adaptée à la vérification de propriétés en μ-calcul propositionnel, la vérification d'une propriété en *pre* sur un tel graphe n'est pas toujours évidente.
Par exemple, considérons la propriété: toujours (f=>pre(g)), où f et g sont des expressions LUSTRE. Soit un état s vérifiant f et ayant deux prédécesseurs s' et s" (cf figure 4.3.a). Si g est vraie dans s' mais non dans s", il est impossible de donner une valeur à *pre(g)* dans l'état s. Ce problème résulte du fait que le repliage de l'arbre des exécutions ne dépend pas des mémorisations booléennes rencontrées dans les propriétés, si celles-ci n'apparaissent pas dans le programme.

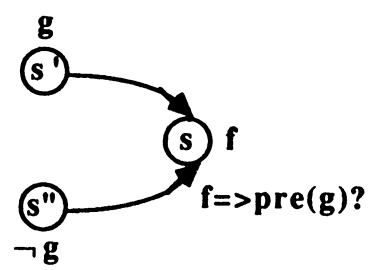



Figure 4.3.a: problème de l'opérateur *pre*

Comme nous l'indique cet exemple, l'évaluation d'une propriété peut nécessiter une expansion de l'automate (et donc du graphe) (cf figure 4.3.b).

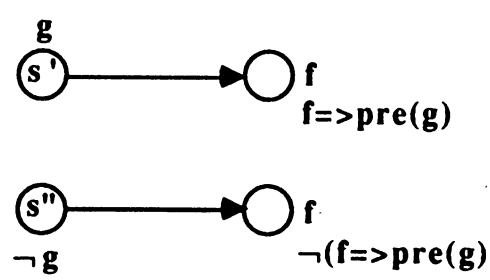



Figure 4.3.b : expansion de l'automate

Cette expansion n'étant pas prise en compte dans XESAR (puisque l'opérateur *pre* n'appartient pas au langage qu'il reconnait), les propriétés en STL à vérifier à l'aide de XESAR ne pouvait contenir l'opérateur *pre* après la traduction par $T_X$. Or, on ne sait pas définir un processus systématique pour traduire une formule en *pre* en une formule de STL de façon à pouvoir

automatiser cette traduction. Le remplacement de l'opérateur *pre* par une sous-formule de STL n'a donc pas pu être automatisée dans la définition de $T_X$.

Néanmoins, pour contourner ce problème, toute expression "en *pre* " apparaissant dans une propriété est incorporée au programme sous forme d'une équation, et la variable associée à l'expression devient une sortie du programme. L'automate généré tient alors compte de ces mémorisations, qui deviennent décidables dans tout état, puisque toute expression "en *pre* " ou "en *current* " apparaissant dans le calcul d'au moins une sortie du programme devient variable d'état de l'automate.

Dans la propriété en STL, les expressions "en *pre* " sont remplacées par les variables qui leur sont associées dans le programme. La traduction vérifiée en XESAR portera donc sur ces variables. Cela se traduit dans l'exemple ci-dessus par :

- l'ajout dans le corps du programme de l'équation v = pre(g),
- et l'ajout de la variable v en sortie.

La propriété en STL à vérifier est alors :

$T_X$( toujours (f=>v) ) = al ($T_X$(f) => v)

4) D'autre part, les prédicats utilisés dans les formules ne peuvent porter sur les variables locales du programme LUSTRE. En effet, le nom de celles-ci n'apparaît pas sur l'automate de contrôle généré par le compilateur LUSTRE. Par conséquent, toute variable apparaissant dans une propriété à vérifier doit être déclarée en sortie du programme.

Cette validation de programmes LUSTRE à l'aide de XESAR peut être appliquée à des programmmes développés à l'aide de SAGA. En effet, comme nous l'avons vu au chapitre 1, un programme SAGA (application complètement affinée) est traduisible en un programme LUSTRE. Toute méthode intéressant des programmes LUSTRE est donc applicable à des programmes SAGA.

Néanmoins, nous avons pu constater que l'utilisation de XESAR impose certaines limites à la vérification de propriétés :

- tout d'abord, l'éloignement du graphe d'états généré par XESAR de l'arbre des exécutions du programme, conduit
  - d'une part, à un accroissement inutile du nombre d'états sur lesquels s'exercent l'évaluation, ce qui est indubitablement un handicap au niveau du temps d'exécution, qui peut de plus devenir rédibitoire au niveau de la place mémoire,
  - d'autre part, à une modification des formules à vérifier.

- ensuite, la limitation du langage STL à une approche du temps tournée vers le futur, nécessite une modification des propriétés exprimées à l'aide de l'opérateur *pre*, et souvent l'ajout de nouvelles équations dans le programme.

## 4.4 AUTRES APPROCHES DE SPÉCIFICATION ET DE VÉRIFICATION DANS LE DOMAINE DES LOGIQUES POUR LE TEMPS RÉEL

Des travaux d'adaptation des logiques temporelles au temps réel ont été réalisés, aussi bien pour des logiques du temps linéaire que pour d'autres du temps arborescent. On peut citer les travaux suivants :

- [KdR,85] étend la logique temporelle linéaire avec une notion de temps quantitative. Les modèles d'une telle logique restent fondés sur les machines à états, avec un unique état initial. A chaque état est associé une composante temps réel et un ensemble de prédicats vrais dans cet état. Le domaine du temps réel est une extension de $\mathbb{N}$ muni de < et +.
  Une séquence d'exécution est une séquence infinie d'états partant de l'état initial, dont la composante temps réel est 0, et telle que les composantes temps réel associées aux états croîssent le long de la séquence.
  La logique comporte trois opérateurs temporels : O, B (before), $\mathcal{U}_{=t}$ (until in real time t). Elle comporte aussi l'égalité. Seul l'opérateur $\mathcal{U}_{=t}$ fait intervenir la notion quantitative du temps. L'expression $P\mathcal{U}_{=t}Q$ spécifie que l'état dans lequel Q est vrai, a une composante temps réel supérieure de t à celle de l'état courant. La quantification sur les variables du temps réel permet de définir d'autres opérateurs comme $\mathcal{U}_{<t}$ ou $\mathcal{U}_{\leq t}$, ainsi que de retrouver les opérateurs usuels de la logique linéaire.
- Dans l'approche de [BH,81], la spécification d'un programme est une assertion sur l'exécution du programme énoncée en logique temporelle linéaire. Dans le modèle de calcul sous-jacent, une exécution du programme est décrite par la séquence infinie des états rencontrés durant une exécution; dans une exécution finie, l'état final est répété infiniment. Chaque action (indivisible) du programme cause une transition.
  Le modèle utilise un domaine du temps global pour tous les processus; ce domaine est $\mathbb{R}$. A chaque état du modèle est associé le temps auquel il est atteint dans l'exécution. Le temps réel est introduit dans le langage de spécification en étendant la notion "d'implication temporelle" avec des bornes de temps, c'est-à-dire :
  la formule : $A_1 \sim><^n A_2$ spécifie que si l'exécution atteint un état satisfaisant $A_1$, alors il atteindra un état satisfaisant $A_2$ en moins de n unités de temps.

la formule : $A_1 \sim>>^n A_2$ spécifie que si l'exécution atteint un état satisfaisant $A_1$, alors $A_2$ ne sera pas vraie dans cet état et l'exécution n'atteindra pas un état satisfaisant $A_2$ durant les prochaines n unités de temps.
Le système de preuve associé est limité à la vérification de propriétés de "safety" particulières. Il utilise une sémantique des programmes à base d'entrelacement.

- [EMSS,89] étend la logique CTL [EH,85] au temps réel. L'extension correspond à celle de [KdR,85] mais pour une logique du temps arborescent. Les nouvelles formules sont :
  A $(pU^{\leq k}q)$ qui signifie que sur chaque séquence d'exécution du modèle, q arrive au cours des k prochains pas d'exécutions, et p est vrai dans tous les états précédant strictement celui où q est vrai.
  E $(pU^{\leq k}q)$ qui signifie qu'il existe au moins une séquence d'exécution du modèle, telle que q arrive au cours des k prochains pas d'exécutions, et p est vrai dans tous les états précédant strictement celui où q est vrai.
  La vérification des formules de RTCTL est réalisée par évaluation, selon un algorithme organisé en étapes : à la $i^e$ étape sont évaluées les formules de longueur i (les formules ci-dessus sont de longueur k). Chaque état est alors étiqueté par les sous-formules de longueur inférieure ou égale à i, vraies dans cet état. L'algorithme s'appuie sur des théorèmes spécifiant l'équivalence entre les formules de longueur i et des formules de longueur strictement inférieure à i.

Nous ne nous sommes pas inspirés de ces différents travaux, car les modalités temporelles qualitatives (des logiques temporelles non temps réel) suffisent à l'expression du langage LEP. Cela provient du fait que le langage de programmation (SAGA ou LUSTRE) est dégagé de toute notion de temps d'exécution à cause de l'hypothèse de synchronisme. La seule notion de temps étant le cycle, la seule modalité "quantitative" nécessaire, doit permettre de compter les cycles (ce que permet l'opérateur *au cycle précédent* de LEP, et d'une autre manière, l'opérateur *next* des logiques du futur).

Notons encore que d'autres travaux ont été réalisés sur l'introduction du temps réel dans les langages de spécification :

- [AK,86] adapte le langage de spécification ASLAN au temps réel. ASLAN est un langage de spécification pour les systèmes séquentiels. Il est construit sur un calcul des prédicats du premier ordre et emploie une méthode de vérification s'appuyant sur les machines à états. Le calcul des prédicats du premier ordre est utilisé pour définir des propriétés qui doivent avoir lieu dans chaque état (invariants) ou à chaque changement d'état par une transition (contraintes). Une spécification ASLAN peut comporter plusieurs niveaux (jusqu'au niveau implantation). ASLAN

déduit les lemmes (conjonctures) nécessaires pour construire une preuve inductive, à partir des différents niveaux de la spécification. Ces conjonctures sont aussi exprimées en logique du premier ordre. RT-ASLAN langage de spécification de systèmes temps réel est fondé sur ASLAN. Les systèmes spécifiables en RT-ASLAN sont des systèmes faiblement couplés communiquant à travers des interfaces formelles. La prise en compte du temps en RT-ASLAN permet d'introduire une notion de séquencement entre les transitions, de séparer les processus d'interface avec l'environnement, des processus internes communicants, d'attacher un temps à chaque transition et de déduire des conjonctures sur ces temps de transition. Des systèmes de preuve et d'analyse de code doivent être adjoints à RT-ASLAN pour prouver la correction des deux types de conjonctures.

- [BW,87] utilise une spécification fonctionnelle basée sur des spécifications algébriques, combinée avec des expressions régulières pour exprimer le comportement temporel. Les systèmes décrits sont composés de processus communiquant à travers des ports auxquels sont attachés des files d'attente. La spécification fonctionnelle décrit le comportement d'une tâche en terme de prédicats sur les données dans les files d'attente de la tâche, avant et après l'exécution de la tâche. La spécification temporelle décrit le comportement d'une tâche en terme des opérations (vues de l'extérieur) qu'elle réalise sur les ports d'entrée et de sortie. A ces opérations peuvent être attachées des délais : durée d'une opération, durée entre deux opérations, ainsi que des gardes spécifiant le nombre de fois que la tâche doit être exécutée, ou des bornes de temps pour le début de son exécution, ou des conditions sur les files d'attente ou sur le temps nécessaire à cette exécution. Les expressions temporelles sont fondées sur la logique RTL de [JM,86].
- [JM,86] est basé sur un modèle de systèmes distribués qui utilise un ensemble prédéfini de noms d'événements et un ensemble d'actions primitives dont le temps d'exécution est fini et connus. A chaque action sont associés deux événements : l'un marquant son début, l'autre sa fin. La logique RTL est une logique du premier ordre dont les prédicats sont des relations algébriques sur les occurrences des événements du système. A chaque occurrence est associée par la "fonction d'occurrence", le temps de cette occurrence. Les prédicats sont de deux sortes : les prédicats sur les états et les contraintes de temps. RTL est utilisé pour spécifier et vérifier des propriétés de "safety". Une spécification RTL consiste en un ensemble d'actions primitives et composées, un ensemble d'événements externes, un ensemble de prédicats d'états et de contraintes de temps. Les actions interagissent simplement, aussi un ordre partiel sur les actions primitives peut-il être dérivé. La spécification d'un modèle actions-événements peut être traduite en formules de RTL. Les contraintes de temps sont des restrictions sur la fonction d'occurrence de RTL. Une propriété de "safety" est vraie, s'il n'existe pas de fonction d'occurrence

consistante avec la négation de la propriété de "safety" et avec les restrictions spécifiées. Les preuves sont réalisées via l'arithmétique de Presburger avec fonctions non interprétées.

## 4.5 CONCLUSION

Dans ce chapitre, nous nous sommes attachés à traduire les formules du langage LEP à l'aide d'une logique temporelle arborescente du futur : ici CL, et d'un μ-calcul propositionnel du futur : ici μ-CL. Ceci afin d'utiliser les langages de spécifications déjà éprouvés et sur lesquels il existe de nombreux résultats, notamment sur les méthodes de vérification de formules de ces langages. Nous avons aussi retracé une utilisation du logiciel XESAR pour la vérification de programmes LUSTRE.

De ce chapitre nous pouvons extraire deux résultats.

D'une part, les logiques temporelles arborescentes et les μ-calculs propositionnels sont des logiques du futur et ne permettent donc pas de traduire l'opérateur *au cycle précédent* de LEP. Elles sont par contre adaptées (suivant leur pouvoir d'expression) à la traduction des autres opérateurs.

D'autre part, le modèle du programme à valider doit être fini pour permettre une vérification automatique. Dans XESAR, l'obligation de passer par une traduction en ESTELLE\R, éloigne le modèle de l'arbre des exécutions. La vérification nécessite alors une adaptation des formules.

De plus, le passage obligé par ESTELLE\R accroît le nombre d'états du modèle, ce qui est un handicap lors de la vérification.

Dans le chapitre 5, nous proposons une approche qui conserve la notion d'état prédécesseur dans le langage de spécification, et pour laquelle le modèle des programmes est le graphe d'états SAGA/LUSTRE lui-même.

# 5 LUSTRE pour la vérification de propriétés

Dans ce chapitre, nous nous attachons à supprimer deux limitations apparues lors de la validation de programmes SAGA à l'aide de XESAR. Nous conservons au langage de spécification une notion d'état prédécesseur et réalisons la validation sur le graphe d'états ou l'automate LUSTRE directement.

Deux approches sont étudiées. Toutes deux sont fondées sur le langage LUSTRE en tant que langage de spécification.

Une première étude [PH,88] a déjà été réalisée concernant les liens existants entre LUSTRE et les logiques temporelles. De cette étude, il ressort que la restriction des expressions de LUSTRE aux seules expressions booléennes, permet de considérer LUSTRE comme un sous-ensemble d'une logique temporelle. Chaque équation LUSTRE est traduisible dans une logique linéaire à l'aide des opérateurs $\Box$, $\Diamond$, F (instant initial), O et U. La traduction fait intervenir l'horloge de l'équation. Elle peut être simplifiée en définissant de nouveaux opérateurs temporels $\Box_f$, $\Diamond_f$, $F_f$, $O_f$ et $U_f$ (où f est une formule de la logique). La sémantique de ces opérateurs s'appuie sur celle des opérateurs traditionnels, mais ne considère que les états de la séquence satisfaisant f. Ces opérateurs vont servir à exprimer la notion d'horloge, en instanciant f par l'horloge de l'équation à traduire.

Les deux approches de spécification basée sur LUSTRE, et présentées dans ce chapitre, prennent en compte un temps arborescent.

- La première permet de traduire toutes les constructions du langage LEP apparues dans les études de cas. Elle conserve les modalités des logiques temporelles arborescentes du futur en les composant avec les opérateurs LUSTRE. Une méthode de validation est esquissée pour des propriétés décrites dans ce langage.
- La deuxième approche consiste à restreindre le langage de spécification à LUSTRE muni d'un opérateur d'invariance sur l'arbre des exécutions tout entier. Un tel langage ne permet de traduire que les constructions invariantes de LEP. Néanmoins, nous avons vu que l'utilisation de ces dernières couvre la majorité des propriétés à exprimer pour des systèmes réactifs de sûreté. De plus, nous avons développé pour ce langage de spécification une méthode originale de vérification de propriétés sur les programmes SAGA ou LUSTRE.

Nous allons débuter ce chapitre par la traduction en LUSTRE d'une propriété issue de l'application SIREX. Cette dernière sera réalisée de manière intuitive en réexprimant la propriété et en la décomposant en sous-propriétés exprimables par des nœuds LUSTRE, eux-mêmes définis par affinements successifs. Le but de ce premier paragraphe est de montrer la cohérence entre la vision LUSTRE des propriétés et la vision intuitive dégagée lors des études de cas.

## 5.1 PASSAGE D'UNE PROPRIÉTÉ À UNE EXPRESSION LUSTRE

Avec les nœuds LUSTRE, nous disposons d'un moyen de décrire des propriétés et de les structurer par composition de propriétés élémentaires. Nous allons le montrer sur un exemple provenant de SIREX. Cette traduction d'une propriété en LUSTRE sera définie plus formellement dans les paragraphes à venir.

Soit la propriété :

*toujours cont* PS_Mx1PE *entre* (Période>Mx1PE) *et* (Période<Mx1PE - hystérésis).

Ré-exprimée en français, son interprétation en terme d'exécution devient :

> si, en un point quelconque de l'exécution, on se trouve dans un intervalle [Période>Mx1PE, Période<Mx1PE - hystérésis], alors on a en ce point la propriété PS_Mx1PE.

On peut définir deux nœuds LUSTRE : ENTRE et IMPLIQUE, chacun à deux paramètres, et écrire :

**toujours** ( IMPLIQUE ( ENTRE(Période>Mx1PE,Période<Mx1PE-hystérésis), PS_Mx1PE) )

Le nœud IMPLIQUE est facile à exprimer (calcul booléen) :

```
node IMPLIQUE (a, b :bool) returns (c : bool);
let
        c = not a or b;
tel.
```

Pour définir le nœud ENTRE, il faut se rappeler qu'une expression LUSTRE ne peut faire référence qu'au passé de l'exécution et jamais au futur.

Un instant de l'exécution se situe entre les deux instants Période>Mx1PE et Période<Mx1PE-hystérésis, si et seulement si, dans son passé, il a existé un instant où

Période>Mx1PE, et depuis cet instant, il n'y a jamais eu Période<Mx1PE-hystérésis, c'est-à-dire qu'il y a eu : continuellement non Période<Mx1PE-hystérésis.

Ce qui revient à dire : si un instant est situé dans un tel intervalle, soit cet instant est le premier de l'intervalle et il doit vérifier Période>Mx1PE, soit il n'est pas ce premier instant et il doit vérifier not(Période<Mx1PE-hystérésis) et son prédécesseur est lui aussi situé dans l'intervalle. Cette vision est décrite par le nœud ENTRE suivant (généralisé au fait que les deux paramètres a et b peuvent avoir lieu au même moment) :

```
node ENTRE (a, b : bool) returns (entre_a_b : bool);
let
        entre_a_b= a and not b -> (a and not b) or (not b and pre(entre_a_b));
tel.
```

## 5.2 LUSTRE AUGMENTÉ D'OPÉRATEURS MODAUX

### 5.2.1 Le langage : MODAL LUSTRE

Le langage de spécification est obtenu en augmentant le langage des expressions LUSTRE à l'aide des opérateurs temporels de la logique L [QS,82]. Cette logique est une restriction de la logique CL [QS,83] (cf chapitre 4). Une logique similaire à L est la logique UB [BMP,83].

La **syntaxe** du langage de spécification est la suivante :
soit f une formule du langage, f est telle que :

f ::= true | p | $op_L(f_1,...,f_n)$ | al(g) | some(g).

où p ∈ P ensemble de variables propositionnelles

et $op_L$ = opérateur LUSTRE booléen d'arité n.

On notera :

pot (f) = not al (not f)

inev (f) = not some (not f)

Un modèle d'une formule f de MODAL LUSTRE est un arbre d'exécution $A=(Q,q_{racine},\rightarrow)$ muni d'une fonction d'interprétation $I : Q \rightarrow 2^P$ tel que : $q_{racine} \models_I f$.
On note $A \models_I f$

Nous allons définir la relation de satisfaction $\models_I$ sur les états :

1) La sémantique des opérateurs de L est donnée sur des arbres, plutôt que sur des graphes d'états comme celle des opérateurs originaux.
Soit un état q quelconque de Q,

$q \models_I true$ $\quad \forall q \in Q$ (ML 1)

$q \models_I p$ $\quad$ ssi $\quad q \in I(p)$ (ML 2)

• Pour $q \neq q_{racine}$ :

$q \models_I al\ (g)$ $\quad$ ssi $\quad \forall s \in EX(q), \forall k, s_k \models_I g$ (ML 3)

$q \models_I some\ (g)$ $\quad$ ssi $\quad \exists s \in EX(q), \forall k, s_k \models_I g$ (ML 4)

• En $q_{racine}$ :

$q_{racine} \models_I al\ (g)$ $\quad$ ssi $\quad \forall s \in EX(q_{racine}), \forall k, k \geq 1 \Rightarrow s_k \models_I g$ (ML 5)

$q_{racine} \models_I some\ (g)$ $\quad$ ssi $\quad \exists s \in EX(q_{racine}), \forall k, k \geq 1 \Rightarrow s_k \models_I g$ (ML 6)

2) La sémantique des opérateurs LUSTRE sur l'arbre des exécutions est obtenue à partir de la sémantique dénotationnelle de LUSTRE [Pla,88]. Ces opérateurs sont maintenant appliqués à des formules et non plus seulement à des expressions LUSTRE.
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, \forall f, f_1$ et $f_2$ formules de MODAL LUSTRE,

$s_k \models_I not\ (f)$ $\quad$ ssi $\quad s_k \not\models_I f$ (ML 7)

$s_k \models_I f_1\ or\ f_2$ $\quad$ ssi $\quad s_k \models_I f_1 \vee s_k \models_I f_2$ (ML 8)

$k \geq 1 \Rightarrow$ ( $s_k \models_I f_1 \rightarrow f_2$ $\quad$ ssi $\quad (k=1 \Rightarrow s_k \models_I f_1)$

$\wedge\ (k>1 \Rightarrow s_k \models_I f_2)$ ) (ML 9)

$k>1 \Rightarrow$ ( $s_k \models_I pre\ (f)$ $\quad$ ssi $\quad s_{k-1} \models_I f$ ) (ML 10)

$k \geq 1 \wedge s_k \models_I f_2 \Rightarrow$ ( $s_k \models_I f_1\ when\ f_2$ $\quad$ ssi $\quad s_k \models_I f_1$ ) (ML11)

$k \geq 1 \wedge s_k \models_I h \Rightarrow$ ( $s_k \models_I current\ (f)$ $\quad$ ssi $\quad (s_k \models_I ck \Rightarrow s_k \models_I f)$

$\wedge\ (s_k \not\models_I ck \Rightarrow s_k \models_I pre\ current(f))$ ) (ML12)

où ck est l'horloge de f et h l'horloge de ck.

### 5.2.2 Traduction du langage d'expression des propriétés en MODAL LUSTRE

Les formules de MODAL LUSTRE et celles du langage d'expression des propriétés ont le même modèle : les arbres des exécutions des programmes. On donne alors la traduction des formules exprimant les propriétés par :

$T_{ml}(vrai) = true$

$T_{ml}(g1\ ou\ g2) = T_{ml}(g1)\ or\ T_{ml}(g2)$

$T_{ml}$(non g) = not $T_{ml}$(g)
$T_{ml}$(p) = p
$T_{ml}$(initial) = true → false
$T_{ml}$(au cycle précédent(g)) = false → pre( $T_{ml}$(g) )

La traduction des constructions temporelles en terme de *toujours* et *préservable*, fait intervenir la définition de deux nœuds LUSTRE : les nœuds Cont et Cont_depuis.
L'expression Cont (e) est vraie à un instant donné si et seulement si, l'expression e a été continuellement vraie depuis le début de l'exécution, et est encore vraie à cet instant :

```
node Cont (a : bool) returns (cont_a : bool);
let
        cont_a = a -> a and pre (cont_a);
tel.
```

L'expression Cont_Depuis (e, f) (abréviation de Continuellement e Depuis f) est vraie à un instant donné si et seulement si,
- soit, l'expression f n'a jamais été vraie depuis le début de l'exécution
- soit, l'expression e a été continuellement vraie depuis la dernière fois, y compris, que f a été vraie, et est encore vraie à l'instant en question.

```
node Cont_Depuis (a, b : bool) returns ( cont_a_depuis_b : bool);
let
   cont_a_depuis_b = not b or a
          -> Cont (not b) or (b and a) or (a and pre (cont_a_depuis_b));
tel.
```

Trois autres nœuds peuvent être utilisés. Ils sont définis en fonction des précédents :

```
node Après (a : bool) returns (après_a : bool);
let
   après_a = not Cont (not a);
tel.
```

La sortie après_a peut donc se réécrire en

```
après_a = a -> a or pre (après_a),
```

ainsi l'expression Après (e) est vraie à un instant donné si et seulement si, l'expression e a été vraie au moins une fois entre le début de l'exécution et cet instant compris.

```
node IlExist_Depuis (a, b : bool) returns ( exist_a_depuis_b : bool);
let
   exist_a_depuis_b = not Cont_Depuis (not a, b);
tel.
```

La sortie exist_a_depuis_b peut donc se réécrire en :

a **and** b -> Après(b) **and** (**not** b **or** a) **and** (a **or pre** (exist_a_depuis_b)),

ce qui est équivalent à :

a **and** b -> (Après(b) **and** a) **or** (**not** b **and pre** (exist_a_depuis_b))

et ainsi, l'expression IlExiste_Depuis (e, f) (abbréviation de Il Existe e Depuis f) est vraie à un instant donné si et seulement si, l'expression e a été vraie au moins une fois entre la dernière fois, comprise, où l'expression f a été vraie et l'instant en question, compris.

```
node ENTRE (a,b : bool) returns (entre_a_b : bool);
let
   entre_a_b = Après(a and not b) and Cont_Depuis(not b, a and not b );
tel.
```

Remarquons que la définition du nœud ENTRE en terme de Après et Cont_Depuis, est équivalente à celle du paragraphe 5.1 ; en effet, l'expression Après(a and not b) annule l'expression Cont (not (a and not b)) contenue dans Cont_Depuis(a and not b, not b) et est redondante avec le reste de cette expression, à savoir : (a and not b) or (not b and pre (Cont_Depuis(not b, a and not b))).

Nous allons donner maintenant la traduction des constructions temporelles du langage d'expression des propriétés à l'aide de ces nœuds (la preuve de la correction de ces traductions se trouve en annexe 3.2)

$T_{ml}$ (toujours cont g entre $g_1$ et $g_2$) =

al( Cont_Depuis ($T_{ml}(g)$ and not $T_{ml}(g_2)$, $T_{ml}(g_1)$ and not $T_{ml}(g_2)$)

or IlExiste_Depuis ( $T_{ml}(g_2)$, $T_{ml}(g_1)$ and not $T_{ml}(g_2)$ ))

$T_{ml}$ (toujours exist g entre $g_1$ et $g_2$) =

al [ Implique ( $T_{ml}(g2)$,
Cont (not ($T_{ml}$ (g1) and not $T_{ml}$ (g2)))
or pre IlExiste_Depuis($T_{ml}$ (g) or $T_{ml}$ (g2),
$T_{ml}$ (g1) and not $T_{ml}$ (g2)) )
and Implique ( $T_{ml}$ (g1) and not $T_{ml}$ (g2),
inev ($T_{ml}$ (g) or $T_{ml}$ (g2)) ) ]

$T_{ml}$ (préservable cont g entre $g_1$ et $g_2$) =

some ( Cont_Depuis($T_{ml}$ (g) and not $T_{ml}$ ($g_2$), $T_{ml}$ ($g_1$) and not $T_{ml}$ ($g_2$))
or IlExiste_Depuis($T_{ml}$ ($g_2$), $T_{ml}$ ($g_1$) and not $T_{ml}$ ($g_2$)) )

La construction *préservable exist* g *entre* $g_1$ *et* $g_2$ n'est pas traduisible telle quelle en MODAL LUSTRE. En effet, les expressions LUSTRE ne permettent d'exprimer que des propriétés faisant intervenir une histoire bornée, puisque le mécanisme de progression dans le temps est une récurrence (à l'aide de l'opérateur *pre* ) depuis un instant fixé de l'exécution jusqu'à l'instant initial. Il faut donc pour traduire une propriété temporelle non bornée dans le futur, introduire dans l'expression un opérateur modal. Dans le cas de *préservable exist g entre* $g_1$ *et* $g_2$ , dont nous rappelons la définition :

• Pour $q \neq q_{racine}$ :

$q \models_I$ préservable exist g entre $g_1$ et $g_2$ ssi

$\exists s \in EX(q), \forall k, ( (s_k \models_I g_1$ et non $g_2)$ =>
$(\exists k', k' \geq k \wedge s_{k'} \models_I g \wedge \forall k'', k' \geq k'' \geq k \Rightarrow s_{k''} \models_I$ non $g_2)$ )

• Pour $q=q_{racine}$, la définition ne s'applique qu'aux $k \geq 1$.

on sélectionne une séquence d'exécution à l'aide de l'opérateur *some*. Sur cette séquence, on ne peut se placer en des points vérifiants $g_2$ puisque, dans certains cas, l'apparition de $g_2$ ne conditionne pas la validité de la formule, on ne peut se placer en des points vérifiant g puisque c'est la propriété qui doit être impliquée par la présence des deux autres, il faut donc se placer en des points vérifiant : $g_1$ et non $g_2$. En ces points, il faut alors décrire l'existence future, non forcément bornée dans le temps et sur la même séquence d'exécution, de la propriété g. Puisque LUSTRE ne convient pas, il faudrait employer un opérateur modal, or aucun ne permet d'exprimer la situation : "sur la même séquence".

Nous avons alors restreint la signification de la construction *préservable exist g entre* $g_1$ *et* $g_2$ . Nous définissons la construction qui suit et qui force l'apparition de g uniquement lorsque $g_2$ existe après $g_1$ et non $g_2$.

• Pour $q \neq q_{racine}$ :

$q_{racine} \models$ *préservable exist g entre* $g_1$ *et* $g_2$ *strict* ssi

$\exists s \in EX(q_{racine}), \forall k \in \mathbb{N},$

$(s_k \models g_1$ et non $g_2 \wedge \exists k', k'>k \wedge s_{k'} \models g_2 \wedge \forall k''', k'>k'' \geq k \Rightarrow s_{k'''} \models$ non $g_2)$

$\Rightarrow \quad (\exists k'', k'>k'' \geq k \wedge s_{k''} \models g)$

• Pour $q=q_{racine}$, la définition ne s'applique qu'aux $k \geq 1$.

L'apparition de g est alors bornée dans le temps par l'arrivée de $g_2$, et nous pouvons donner la traduction en nous plaçant aux points vérifiant $g_2$ et en exprimant en ces points une propriété du passé à l'aide des opérateurs LUSTRE (la preuve de la correction de la traduction se trouve en annexe 3.2.4):

$T_{ml}$(préservable exist g entre $g_1$ et $g_2$ strict) =

some [ not $T_{ml}(g_2)$

or pre ( Cont(not ($T_{ml}(g_1)$ and not $T_{ml}(g_2)$))

or IlExiste_Depuis($T_{ml}(g)$ or $T_{ml}(g_2)$,$T_{ml}(g_1)$ and not $T_{ml}(g_2)$)) ]

Nous définissons aussi la duale de cette construction :

*inévitable cont g entre* $g_1$ *et* $g_2$ strict = non (préservable exist (non g) entre $g_1$ et $g_2$ strict).

### 5.2.3 Vérification de propriétés en MODAL LUSTRE

La vérification va être réalisée sur le **graphe d'états LUSTRE**. Le programme doit donc ne contenir que des variables dont les domaines de définition sont bornés, afin que ce modèle soit fini et que la vérification soit réalisable.

On définit le langage VMODAL LUSTRE des formules vérifiées sur le graphe et une relation $T_{MV}$ : MODAL LUSTRE -> VMODAL LUSTRE, telle que :

soit A l'arbre des exécutions d'un programme P et G le graphe d'états de ce même programme P, soit f une formule de MODAL LUSTRE,

$$A \models f \text{ ssi } G \models T_{MV}(f)$$

VMODAL LUSTRE, $T_{MV}$, et la relation de satisfaction $\models$ sur les états du graphe vont être définis en considérant les formules de MODAL LUSTRE de manière inductive. L'opérateur de traduction $T_{MV}$ cherche à conserver les opérateurs de MODAL LUSTRE. Il le fera lorsque l'évaluation d'un opérateur sur le graphe peut conserver la même définition que sur l'arbre des exécutions, en conservant la même sémantique.

A) Soit une formule f de MODAL LUSTRE à un seul niveau d'imbrication d'opérateurs, alors f est soit *true* soit une variable x.

- *true* est conservée telle quelle par l'opérateur $T_{MV}$, puisque tout état de l'arbre des exécutions satisfaisant *true*, tout état du graphe (classe d'équivalence) satisfait *true*.
- x est conservée telle quelle aussi, car tout état du graphe représente des états de l'arbre ayant même mémoire, et donc associant à x la même valeur.

B) Soit une formule f de MODAL LUSTRE à un ou plusieurs niveaux d'imbrication d'opérateurs, alors l'opérateur de plus haut niveau de f est soit un opérateur LUSTRE, soit *al* ou *some*.

B.1) Si f est une formule d'opérateur LUSTRE au plus haut niveau, cet opérateur est soit instantané booléen soit temporel.

B.1.1 - $T_{MV}$ conserve tels quels les opérateurs instantanés (*and*, *or*, ...). En effet, si leurs opérandes ont la même valeur dans tous les états de l'arbre regroupés en un état du graphe, ils auront aussi la même valeur dans tous ces états.

B.1.2 - Parmi les opérateurs temporels LUSTRE, il en est deux dont l'évaluation sur l'arbre peut être conservée sur le graphe: ce sont -> et *when*. En effet, d'une part, le domaine de définitions des mémoires du graphe d'état comporte les variables nécessaires pour déterminer le premier instant de toute horloge d'un opérateur ->. Ainsi, cet opérateur peut être évalué dans chaque état du graphe en utilisant la définition (ML 9) du paragraphe 5.2.1. D'autre part, l'évaluation de l'opérateur *when* ne nécessite rien de plus que l'évaluation de ces opérandes, $T_{MV}$ le conserve donc pour la même raison que les opérateurs instantanés.

B.1.3 - L'évaluation des deux opérateurs temporels LUSTRE restants : *pre* et *current*, pose plus de problèmes.

B.1.3.1 - Le problème de l'opérateur *pre* de LUSTRE est identique à celui de l'évaluation de l'opérateur *after* de STL en XESAR (voir [Ro,88] §6.3). Nous allons donc le résoudre de façon similaire. Ce problème est que la satisfaction des formules construites à l'aide de ces opérateurs ne dépend pas seulement de l'état du graphe dans lequel est évaluée la formule, mais aussi de l'exécution sur le graphe qui y a mené. En effet, soit g une formule de MODAL LUSTRE et q un état du graphe tel que :

$$\exists q', q' \to q \wedge q' \models g \wedge \exists q'', q'' \to q \wedge q'' \not\models g$$

suivant les exécutions, q représente un état de l'arbre des exécutions qui satisfait ou ne satisfait pas "pre(g)" (voir le paragraphe 4.3 et la figure 4.3.a).

Nous désirons attacher la propriété de satisfaire "pre(g)" à un état indépendamment de toute exécution. La méthode envisagée est la même qu'utilise le logiciel XESAR pour la vérification d'une formule en *after*. Elle consiste à éclater l'état q en $q_1$ et $q_2$ tels que :

$(\forall q', q' \rightarrow q \wedge q' \models g) \Rightarrow q' \rightarrow q_1$

$(\forall q', q' \rightarrow q \wedge q' \not\models g) \Rightarrow q' \rightarrow q_2$

Ainsi, nous dirons que $q_1 \models pre_G(g)$ et $q_2 \not\models pre_G(g)$ et donnerons la définition de $pre_G(g)$ pour un état non initial du graphe :

$q_i \models pre_G(g)$ ssi $(\forall q', q' \rightarrow q_i \Rightarrow q' \models g)$ (PRE)

Le graphe d'état srésultat de l'éclatement des états suivant pre(g) peut être obtenu par repliage de l'arbre des exécutions en rajoutant dans le domaine de définition des mémoires, une variable v égale à pre(g). En effet, tout état de l'arbre anciennement de mémoire $\sigma$ définie sur V, est maintenant de mémoire $\sigma_1$ ou $\sigma_2$ définie sur $V \cup \{v\}$ par :

$\forall x \in V, \sigma_i(x) = \sigma(x)$ pour i = 1, 2

$\sigma_1(v)$ = true

$\sigma_2(v)$ = false

suivant que la propriété g est évaluée à *true* ou *false* dans l'état précédent.

Le repliage de l'arbre est réalisé sur des états de mémoires identiques. Il sépare alors les états de mémoire $\sigma_1$ et ceux de mémoire $\sigma_2$. Ce qui revient à dire que deux états sont dans la même classe d'équivalence (même état du graphe) si les restrictions de leurs mémoires sur V sont identiques et si leurs prédécesseurs évaluent la propriété g à la même valeur. Ainsi, tout état de l'arbre satisfaisant pre(g) est représenté par un état du graphe satisfaisant $pre_G(g)$ et tout état de l'arbre ne satisfaisant pas pre(g) est représenté par un état du graphe ne satisfaisant pas $pre_G(g)$.

Nous intégrons alors les formules en $pre_G$ à VMODAL LUSTRE, et définissons la relation de traduction sur les formules en *pre* par :

soit g une formule de MODAL LUSTRE,

$T_{MV}(pre(g)) = pre_G(T_{MV}(g))$

Ainsi, un état de l'arbre satisfait pre(g) si et seulement si l'état représentant de sa classe dans le graphe satisfait $pre_G(g)$.

Comme la formule $pre_G(g)$ dépend de la sous-formule g, l'algorithme de vérification se fera en deux étapes :

• détermination des états satisfaisant g.

• éclatement des états suivant $pre_G(g)$.

B.1.3.2 - La définition du *current* (règle ML 12 du paragraphe 5.2.1) est exprimée récursivement à l'aide de l'opérateur *pre*. Lorsque le problème du *pre* est résolu, la seule nécessité pour évaluer l'opérateur *current*, en un état q, est de pouvoir réaliser un récursion sur le passé de q.

Sur l'arbre, ce passé correspond à une sous-séquence d'exécution finie partant de la racine et finissant en q. Il faut retrouver sur le graphe à quoi correspond une telle séquence.

D'après la définition du graphe d'états LUSTRE (cf paragraphe 2.2.2), à toute séquence de l'arbre est associée une unique séquence du graphe, et réciproquement. De plus, tout état de la séquence de l'arbre est associé à un unique état de la séquence du graphe, et réciproquement. La récursion nécessaire à l'évaluation de l'opérateur *current* sera donc réalisée sur le graphe de la même manière que sur l'arbre des exécutions et l'opérateur *current* peut être conservé tel quel par $T_{MV}$.

B.2) Les opérateurs *al* et *some* sont évaluables sur le graphe d'état en conservant les définitions (ML 3) à (ML 6) du paragraphe 5.2.1. En effet, ils sont aussi définis sur l'arbre des exécutions en fonction de séquences d'états. Les arguments valables pour l'opérateur *current* le restent donc pour eux. $T_{MV}$ conserve alors ces opérateurs tels quels.
L'idée est alors d'utiliser des algorithmes similaires à ceux de XESAR pour la vérification de ces opérateurs.

En conclusion, VMODAL LUSTRE est l'ensemble des formules de MODAL LUSTRE dans lesquelles toute occurrence de l'opérateur *pre* est remplacée par une occurrence de l'opérateur $pre_G$. $T_{MV}$ est donc la fonction de remplacement de *pre* en $pre_G$. La relation de satisfaction $\models$ sur le graphe est similaire à celle définie sur l'arbre des exécutions :
soit $G = (Q, q_0, \rightarrow)$, soit f une formule de VMODAL LUSTRE,

$G \models f$ ssi $q_0 \models f$

où la relation de satisfaction $\models$ entre un état du graphe et une formule est définie par les règles (ML 1) à (ML 9) et (ML 11) et (ML 12) du paragraphe 5.2.1 et par la règle (PRE) de ce paragraphe-ci.

Cette vérification de propriétés en MODAL LUSTRE n'a pas été réalisée. Notamment, il n'existe pas d'outil de génération d'un graphe d'états LUSTRE tel que celui-ci est défini au chapitre 2. Cette vérification semble néanmoins tout à fait réalisable.

Nous avons préféré nous attacher à une voie moins explorée qui consiste à utiliser LUSTRE seul comme langage de spécification et à réaliser la validation de programmes suivant une méthode particulière aux langages flots de données synchrones.

## 5.3 LUSTRE AUGMENTÉ DE L'OPÉRATEUR "TOUJOURS"

### 5.3.1 Le langage : INVARIANTS LUSTRE

Les formules de ce langage ont pour modèles des arbres des exécutions $A=((Q,q_{racine},\rightarrow),I)$. Elles sont de la forme :

f ::= toujours(EL) | f et f' | f ou f'.

où EL est une expression booléenne LUSTRE bien formée (notamment correcte au niveau des horloges et de l'absence de blocage).

On définit la sémantique de ces formules par :

$A \models_I f \iff q_{racine} \models_I f$

et

- $q_{racine} \models_I$ toujours(EL) ssi $\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow s_k \models_I EL$.
- $q_{racine} \models_I f$ et f' ssi $q_{racine} \models_I f$ et $q_{racine} \models_I f'$
- $q_{racine} \models_I f$ ou f' ssi $q_{racine} \models_I f$ ou $q_{racine} \models_I f'$.

La relation de satisfaction entre un état et une expression LUSTRE est définie à l'aide des règles (§5.1) de la sémantique des opérateurs LUSTRE sur un arbre des exécutions (ces opérateurs sont maintenant appliqués exclusivement à des expressions LUSTRE).

Les formules d'INVARIANTS LUSTRE sont des propriétés de "safety" au sens de [LPZ,85]. En effet, les auteurs qualifient les propriétés de "safety" par les expressions de la logique linéaire $\Box\varphi$, où $\varphi$ est une formule du passé. On peut remarquer que, dans un arbre infini représentant un programme, toute propriété du passé exprimée en un état est forcément linéaire sur le chemin menant de la racine à cet état (elle ne fait pas intervenir la notion de branchement

caractéristique de la vision arborescente puisque tout état a un passé unique). Ainsi, il est équivalent, dans ce cas précis, d'avoir une vision arborescente ou linéaire du temps, et les propriétés de sûreté sont qualifiables à l'aide d'un opérateur arborescent "toujours" par *toujours* $\varphi$. D'après les auteurs, cette définition des propriétés de "safety" correspond à celle de [AS,85].

Dans sa classification des propriétés [Lam,84], les modèles définis par Lamport sont des couples $(S,\Sigma)$ où S est un ensemble d'états et $\Sigma$ un ensemble de séquences vérifiant l'hypothèse :

$s \in \Sigma \Rightarrow s^+ \in \Sigma$

où si $s = s_0,s_1,\ldots,s_n,\ldots$ alors $s^+=s_1,\ldots,s_n,\ldots$

Ce qui signifie que le comportement futur du programme dépend uniquement de l'état courant. Ce qui implique que les propriétés sont préservées lors de répétitions finies d'états individuels ("stuttering").

Un arbre d'exécution de programme LUSTRE représente un ensemble de séquences d'exécution qui toutes vérifient la propriété de Lamport. Aussi d'après [AS,85], les formules d'INVARIANTS LUSTRE sont aussi des propriétés de safety suivant Lamport [Lam,84].

### 5.3.2 Traduction du langage d'expression des propriétés en INVARIANTS LUSTRE

INVARIANTS LUSTRE est un langage dans lequel la modalité temporelle *toujours* ne peut être imbriquée. Il ne peut donc pas exprimer toutes les imbrications d'opérateurs du langage LEP. Seules les formules de LEP traduisibles par des expressions LUSTRE pourront être imbriquées. Soit $F_{imb}$ le langage des formules de LEP pouvant être imbriquées.
Soit g, g' formules de $F_{imb}$,

g::= vrai | initial | p | au cycle précédent (g) | g ou g' | non g.

Leur traduction en un état $q \neq q_{racine}$ est :

$T_{il}$(vrai) = true

$T_{il}$(initial) = true → false

$T_{il}$(p) = p

$T_{il}$(au cycle précédent(g)) = false → pre( $T_l$(g) )

$T_{il}$(g ou g') = $T_l$(g) or $T_l$(g')

$T_{il}$(non g) = not $T_l$(g)

Il reste alors à traduire en $q_{racine}$ les formules suivantes :

toujours cont g entre $g_1$ et $g_2$,
toujours exist g entre $g_1$ et $g_2$,
préservable cont g entre $g_1$ et $g_2$,
préservable cont g entre $g_1$ et $g_2$.

En fait, seule la première formule est traduisible et uniquement dans le cas où g, g1, g2 sont des formules de $F_{imb}$. La traduction obtenue utilise les nœuds LUSTRE définis au §5.1.2.

$T_{il}$( toujours cont g entre $g_1$ et $g_2$ ) = toujours [
Cont_depuis ( $T_{il}(g)$ and not $T_{il}(g)$ , $T_{il}(g_1)$ and not $T_{il}(g_2)$ )
or IlExiste_Depuis( $T_{il}(g_2)$ , $T_{il}(g_1)$ and not $T_{il}(g_2)$ ) ]

La preuve de cette traduction est la même que celle de la traduction en MODAL LUSTRE. En effet, l'opérateur toujours est une restriction de l'opérateur *al* du langage MODAL LUSTRE et $\forall g \in F_{imb}$, $T_{il}(g) = T_{ml}(g)$.

La traduction définie intuitivement au § 5.1 en terme des nœuds IMPLIQUE et ENTRE, reste valide (cf annexe 4).

La formule *toujours exist g entre* $g_1$ *et* $g_2$ telle qu'elle a été définie, n'est pas exprimable en INVARIANTS LUSTRE. En effet, une expression LUSTRE fait toujours référence, soit au présent, soit au passé. Pour exprimer une propriété temporelle du futur, le seul opérateur utilisable est donc *toujours* , qui nous permet de nous situer en un point quelconque de ce futur et d'exprimer en ce point une propriété à l'aide d'une expression LUSTRE. Pour la propriété *toujours exist* g *entre* $g_1$ *et* $g_2$, dans le cas où la propriété $g_2$ a bien toujours lieu dans tout futur de la propriété $g_1$, il suffit de se placer en un état quelconque vérifiant $g_2$, à l'aide de l'opérateur *toujours*, et de dire en cet état : s'il y a eu un $g_1$ dans le passé de $g_2$ alors il y eu un g depuis le dernier $g_1$, ce qui est une propriété traduisible par une expression LUSTRE. Par contre, sur une branche où $g_2$ n'apparait pas après $g_1$, il est impossible de trouver un état quelconque du futur de $g_1$ dans lequel énoncer, en utilisant le passé, l'apparition de g avant l'infini (en effet, pour tout état, si g n'est pas encore apparu avant cet état, cela ne veut pas dire qu'il n'apparaitra pas dans le futur de celui-ci).

Par contre, on peut restreindre la signification de cette construction afin qu'elle devienne exprimable en INVARIANTS LUSTRE.

La restriction consiste à ne plus obliger l'apparition de la propriété g après celle de $g_1$, lorsque la propriété $g_2$ n'a ensuite jamais lieu. Nous définissons alors la construction *toujours exist g entre $g_1$ et $g_2$ strict* :

$q_{racine} \models$ toujours exist g entre $g_1$ et $g_2$ strict ssi

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N},$

$( (s_k \models g_1$ et non $g_2)$

$\wedge (\exists k', k'>k \wedge s_{k'} \models g_2)$

$\wedge (\forall k'', k'>k''\geq k \Rightarrow s_{k''} \models$ non $g_2)$ $) \Rightarrow (\exists k'', k'>k''\geq k \wedge s_{k''} \models g)$

Que l'on traduit en INVARIANTS LUSTRE lorsque g, g1, g2$\in F_{imb}$ par :

$T_{il}$ (toujours exist g entre $g_1$ et $g_2$ strict) =

toujours( not $T_{il}$ ($g_2$)

or (true -> pre( Cont (not ($T_{il}$ ($g_1$) and not $T_{il}$ ($g_2$))) or

IlExiste_Depuis($T_{il}$ (g) or $T_{il}$ ($g_2$), $T_{il}$ ($g_1$) and not $T_{il}$ ($g_2$)) )))

Remarquons qu'il est possible de traduire, en INVARIANTS LUSTRE, les opérateurs *toujours cont* et *toujours exist strict* en un état q différent de $q_{racine}$ à l'aide de l'opérateur *when*. Mais pour cela, il faut connaître l'expression LUSTRE caractérisant q. Cette traduction est alors inutilisable pour imbriquer les opérateurs *toujours cont* et *toujours exist strict* , car, par imbrication, on ne sélectionne pas un seul état q mais un ensemble d'états.

Les formules de type "*possible*", "*préservable*" et "*inévitable*" ne sont pas traduisibles avec l'opérateur *toujours*.

### 5.3.3 Possibilités supplémentaires d'expression

On peut imaginer définir d'autres notations que celles apparues dans les études de cas, à l'aide des nœuds LUSTRE définis au paragraphe 5.1.2. Par exemple :

toujours (ENTRE ($g_1$, $g_2$) ≡ g)

toujours (g => ENTRE ($g_1$, $g_2$))

toujours (ENTRE ($g_1$, $g_2$) ≡ ENTRE ($g_3$, $g_4$))

toujours ((ENTRE ($g_1$, $g_2$) and $g_3$) => g)

toujours (ENTRE ($g_1$, $g_2$) and ENTRE ($g_3$, $g_4$) => g)

avec g, $g_1$, $g_2$, $g_3$, $g_4 \in F_{imb}$.

Une de ces constructions permet d'exprimer de façon plus concise la propriété :

**jamais** PS-MinBN **sans** BN-valide
et **toujours cont** (non PS_minBN)
**entre** initial
**et** ((Flux-BN<MinBN) et BN-actif et BN-valide)
et **toujours cont** (non PS_minBN)
**entre** (Flux-BN>MinBN+hystérésis)
**et** ((Flux-BN<MinBN) et BN-actif et BN-valide)
et **toujours cont** (non BN-valide ou PS-MinBN)
**entre** ((Flux-BN<MinBN) et BN-actif et BN-valide)
**et** (Flux-BN>MinBN+hystérésis)

énoncée précédemment, par :

toujours ( (ENTRE ( Flux-BN>MinBN+hystérésis,
Flux-BN<MinBN et BN-actif et BN-valide)
and BN-valide)
≡ PS-MinBN)

Le langage INVARIANTS LUSTRE ne permet pas de traduire toutes les formules du langage d'expression des propriétés. Néanmoins, il permet d'exprimer toutes les propriétés invariantes de l'arbre des exécutions rencontrées lors des études de cas.

### 5.3.4 Principe de la vérification en INVARIANTS LUSTRE

Les différentes étapes de cette approche sont synthétisées en figure 5.3.4.a

Si le programme à vérifier est écrit en SAGA, il est d'abord traduit en LUSTRE. La vérification sera, en effet, réalisée à partir de programmes LUSTRE. De plus, cette vérification ne porte que sur des propriétés exprimables en INVARIANTS LUSTRE. Les propriétés peuvent cependant être énoncées à l'aide des opérateurs définis au chapitre 3 (§ 3.2), auquel cas une traduction est réalisée.

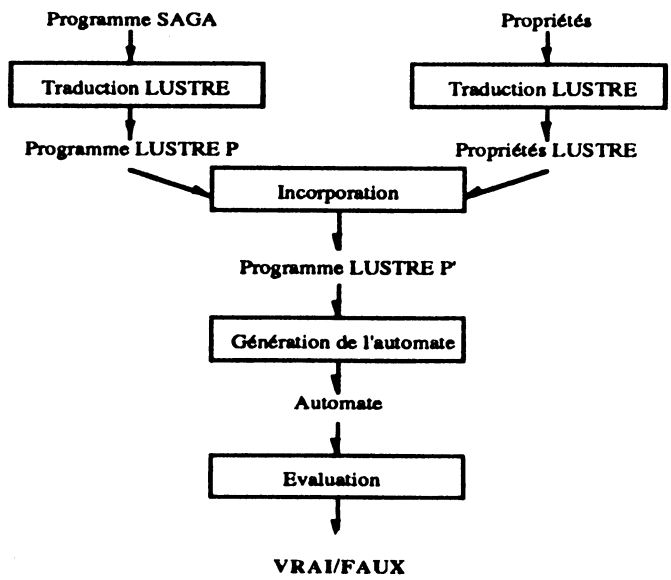



Figure 5.3.4.a : Principe de la vérification

### 5.3.4.1 Automate et propriétés de sûreté

Le passage de l'arbre des exécutions en arbre d'interprétation abstraite peut augmenter le nombre d'états et de chemins de l'arbre sous-jacent.

Par exemple, soit le nœud EXEMPLE défini par :

```
node EXEMPLE (z : bool) returns (y : bool);
var
   x : int;
   v : bool;
let
   x = COUNT (0, 1, pre(x)=1);
   v = (x>4);
   y = if v then z else false;
tel.
```

où COUNT est le nœud défini au paragraphe 1.3.2.2.

L'arbre des exécutions du programme EXEMPLE est le suivant :

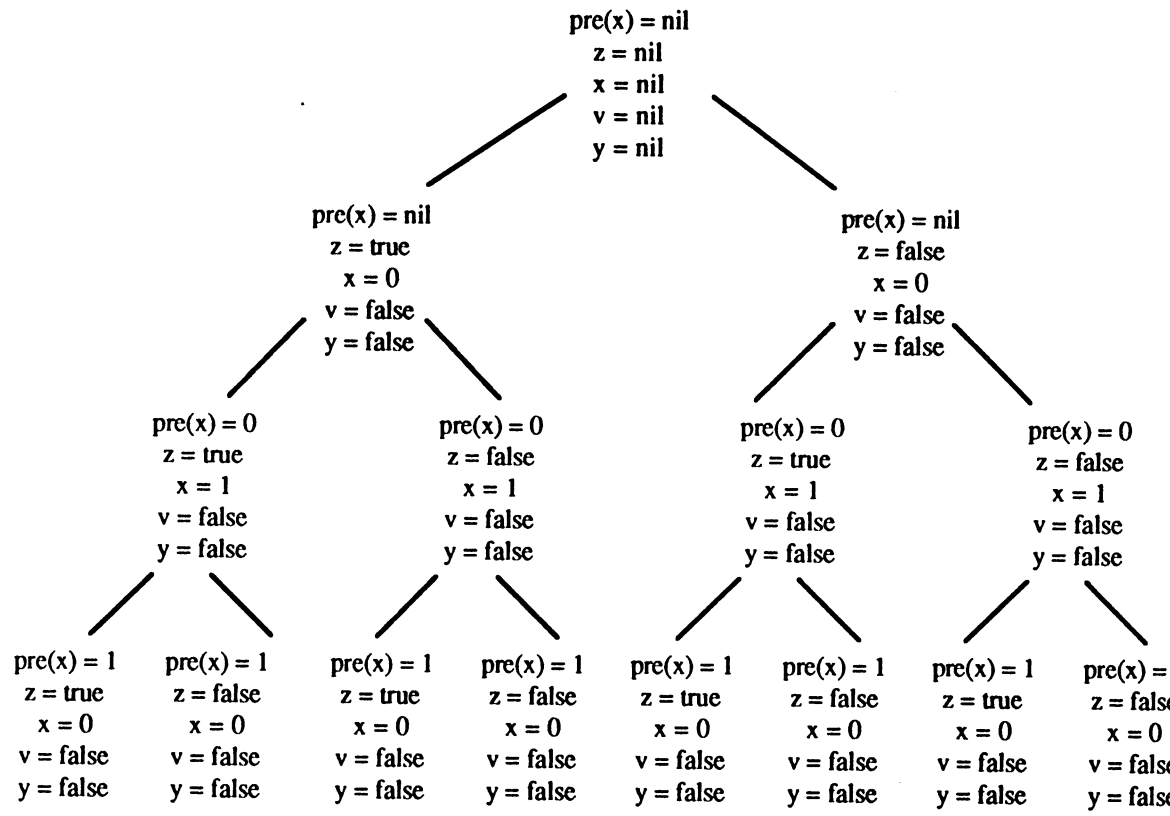



Figure 5.3.4.1.a : Arbre des exécutions du programme EXEMPLE

On remarque que la variable v est toujours fausse, car le test x>4 est toujours faux puisque x est incrémenté modulo 2 à partir de 0. Par conséquent, la valeur de y est toujours : *false*.

Dans l'arbre étendu d'interprétation abstraite, la variable x n'apparaît plus. L'expression x>4 est évaluée comme un booléen dont on ne connait pas la valeur.

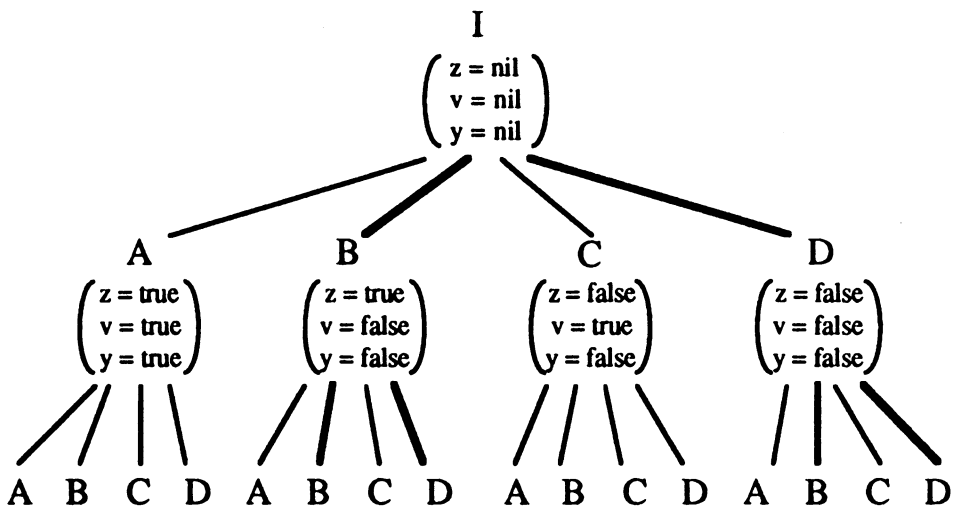



Figure 5.3.4.1.b : Arbre étendu d'interprétation abstraite du programme EXEMPLE

Nous voyons que seuls les chemins de l'arbre étendu d'interprétation en gras sur la figure 5.3.4.1.b, correspondent à un chemin sur l'arbre des exécutions (figure 5.3.4.1.a).

En faisant abstraction des entiers, on perd de l'information : des combinaisons de valeurs sont envisagées alors qu'elles ne sont pas réellement possibles. Ainsi, une propriété évaluée à faux dans tout état de l'arbre étendu d'interprétation abstraite (par exemple *toujours v* ) n'est pas forcément fausse sur l'arbre des exécutions. Mais toute propriété vérifiée sur l'arbre étendu d'interprétation abstraite est vraie sur l'arbre des exécutions. Il en est de même pour la vérification sur le repliage de l'arbre étendu d'interprétation abstraite : l'automate de contrôle.

On voit donc que, notamment, les propriétés arithmétiques ne pourront être vérifiées sur l'automate de contrôle.

On voit aussi que l'arbre étendu d'interprétation abstraite n'est pas un modèle adéquat à la vérification de propriété de "liveness". Car, par exemple, la propriété *possible v* est vérifiée sur l'arbre étendu d'interprétaion abstraite (figure 5.3.4.1.b) alors qu'elle est fausse sur l'arbre des exécutions du programme (figure 5.3.4.1.a). C'est pourquoi, nous avons restreint le langage de spécification à des propriétés de sûreté.

### 5.3.4.2 Incorporation de la propriété au programme

La propriété à vérifier est de la forme : toujours(E), où E est une expression LUSTRE booléenne bien formée. L'expression E est alors incorporée au programme LUSTRE P obtenu à

partir du programme SAGA, de façon à obtenir un programme équivalent à l'application en SAGA, qui de plus calcule E. Ceci est réalisé de la façon suivante :

Une variable booléenne $V_p$ est déclarée en sortie du nœud principal, et l'équation $V_p = E$, est incorporée dans le corps de ce nœud.

Toute variable apparaissant dans le calcul de E et n'appartenant pas à P, est déclarée en variable locale et définie par une équation dans le corps du même nœud.

Tout nœud apparaissant dans le calcul de l'expression E, est incorporé au programme P, de façon à être visible depuis le nœud principal; le plus simple étant de l'introduire au niveau d'imbrication le plus haut.

La définition d'une équation supplémentaire n'affecte pas les calculs précédents du programme. En effet, la variable $V_p$ n'apparaît dans aucune expression en partie droite d'équation. Sa présence ne modifie donc le calcul d'aucune variable (toute variable étant définie par une équation). De même, les nouveaux nœuds et équations incorporés n'apparaissent que dans l'expression E, et ne modifient donc aucune variable en dehors de $V_p$.

$V_p$ étant maintenant une sortie **booléenne** du programme P' ainsi obtenu, les sous-expressions booléennes en *pre* et *current* et les initialisations d'horloges, nécessaires au calcul de E, sont désignées comme variables d'état lors de la génération de l'automate de contrôle de P'.

La valeur de vérité de $V_p$ peut alors être évaluée dans chaque état . Elle ne dépend que de conditions sur les valeurs d'entrée puisque l'histoire nécessaire à son calcul est mémorisée dans chaque état sous la forme des variables d'état.

L'intérêt d'une telle approche est de résoudre à priori le problème de l'opérateur *pre* . En effet, si l'expression E comporte des sous-expressions booléennes en *pre* , celles-ci seront prises comme variables d'état lors de la génération de l'automate. Ainsi, dans tout état, la valeur de ces sous-expressions sera calculable, sans éclatement supplémentaire, indépendamment de l'exécution ayant conduit à cet état.

Notons que si l'automate généré à partir du programme P' est en général différent de celui généré à partir de P, il reste un modèle abstrait du programme P.

#### 5.3.4.3 Automate incorporant la propriété = modèle abstrait du programme

Les arbres des exécutions des deux programmes P et P' ont une structure identique puisqu'ils possèdent les mêmes entrées. Leur seule différence réside dans le domaine de définition

des mémoires associées aux états : le domaine de P' est celui de P dans lequel ont été rajoutées la variable $V_p$ et celles nécessaires au calcul de celle-ci.

- Soit $V_{VE}$ l'ensemble des variables correspondant à des expressions booléennes en *pre* ou en *current* ou à des initialisations d'horloges, nécessaires au calcul des sorties de P.
- Soit $V'_{VE}$ l'ensemble des variables correspondant à des expressions booléennes en *pre* ou en *current* ou à des initialisations d'horloges, nécessaires au calcul des sorties de P'.

alors $V_{VE} \subseteq V'_{VE}$

Ainsi, si deux mémoires sont équivalentes pour P' (mêmes valeurs associées aux variables de $V'_{VE}$), elles le restent pour P (mêmes valeurs associées aux variables de $V_{VE}$). Et par conséquent, si deux nœuds de l'arbre des exécutions de P' se bisimulent, les deux nœuds correspondants dans l'arbre de P se bisimulent. Donc, l'automate généré à partir de P' est un modèle abstrait de P.

La propriété toujours(E) est vraie si et seulement si tout état de l'arbre des exécutions satisfait E. Or, tout état de l'automate regroupe un ensemble d'états de l'arbre des exécutions de telle façon que l'ensemble des états de l'automate regroupe tous les états de l'arbre des exécutions. Ainsi, E est vraie dans tout état de l'arbre des exécutions si E (et donc $V_p$) est évaluée à vrai dans tout état de l'automate.

#### 5.3.4.4 Différences avec XESAR dans l'approche d'évaluation

Une des différences entre notre approche et le "model checking" mis en œuvre à l'aide de XESAR, est que notre modèle, l'automate LUSTRE, est une abstraction du graphe d'états. Il ne prend en compte que les expressions booléennes. C'est pourquoi, il n'est pas possible sur un tel modèle de vérifier des propriétés de vivacité ("liveness") (cf paragraphe 2.2.3.2).

C'est d'ailleurs pour cette même raison que toutes les variables apparaissant dans les programmes en entrée de XESAR ont des domaines de définition bornés. Ainsi, le graphe d'états peut envisager toutes les combinaisons possibles de valeurs durant l'exécution, et les propriétés de vivacité peuvent alors être vérifiées.

Une deuxième différence est que ce modèle est le programme qui va s'exécuter sur la machine. Ce que nous prouvons est donc ce que nous exécutons, et non simplement un modèle théorique dont le programme exécutable n'est pas dérivé automatiquement.

Enfin, une troisième différence est que nous incorporons dans le modèle la propriété à vérifier au lieu de la conserver à part dans un formalisme différent.

### 5.3.5 Une autre approche pour la vérification de langage flots de données

[Bar,84] définit un langage flots de données pour la description de circuits matériels (notons que LUSTRE est aussi employé à cette fin [HLP,86]).

La description d'un circuit consiste en une collection de modules organisés hiérarchiquement. Chaque module est modélisé comme un automate d'états finis et comprend :

- un ensemble de ports d'entrée et de sortie
- un ensemble de variables d'état
- un ensemble d'équations spécifiant le comportement des signaux de sorties en fonction des signaux d'entrée et de l'état courant
- un ensemble d'équations spécifiant le comportement des états successeurs en fonction des signaux d'entrée et de l'état courant.

Ce langage, tout comme LUSTRE, utilise les notions d'état prédécesseur et d'équations aux points fixes. Il s'appuie sur la présence implicite d'une horloge globale, mais n'a pas comme LUSTRE la possibilité de définir des sous-horloges.

Un module composé est une interconnection de modules via leurs ports. Les variables d'états du module sont constituées par des variables d'états des composants de ce module.

Comme dans notre approche, la spécification d'un module est fournie dans le même langage que sa description. La vérification est réalisée de façon modulaire en suivant la décomposition hiérarchique du module.

- Un module sans structure interne est supposé correct.
- La vérification d'un module composé est réalisée en trois étapes. La première étape consiste à prouver ses composants par rapport à leur spécification. La deuxième étape consiste à dériver la description du comportement d'un module, à partir des comportements spécifiés de ses composants, par manipulation algébrique des équations. Cette dérivation, pour rester simple, nécessite que toutes les boucles dans les équations soient coupées par des variables d'états (ce qui se rapporte à l'absence de blocage en LUSTRE). La dernière étape est ensuite la comparaison pour chaque sortie des expressions exp1 et exp2, où sortie:=exp1 est l'équation du comportement dérivé et sortie:=exp2 est l'équation du comportement spécifié. Cette comparaison est réalisée, par énumération si le nombre d'entrées et de variables d'état est faible, par manipulation algébrique sinon, avec possibilité d'aide interactive.

Nous pouvons donc noter, que, de la même façon que nos propriétés son décrites en LUSTRE, les spécifications sont décrites, ici, dans le même langage que la description du comportement. Par contre, la vérification proprement dite, est réalisée par manipulation algébrique

des équations, alors que la vérification, en LUSTRE, consiste à évaluer des expressions sur un modèle. Notons aussi, qu'il est possible de réaliser un programme LUSTRE comparant une spécification et une description toutes deux décrites par un nœud, en ajoutant comme propriété à vérifier que chaque sortie de la description est égale à la sortie correspondante de la spécification.

### 5.3.6 Abstraction par rapport à la propriété

#### 5.3.6.1 Définition

L'abstraction par rapport à la propriété est utile pour réduire le temps de réponse et diminuer la taille mémoire nécessaires à la vérification. Elle consiste à réaliser seulement le calcul des variables nécessaires à la propriété.
Par exemple dans le cas de SIREX, pourquoi calculer les alarmes sur le haut niveau de capteurs, si on désire vérifier une propriété sur l'alarme AL_minBN et si celle-ci n'interfère pas avec les premières?

Cette séparation entre l'utile et le superflu, peut être réalisée en n'admettant pour sortie du programme que la propriété à vérifier. En effet, lors de la génération de code, seules les variables qui interviennent directement dans le calcul des sorties, sont calculées à chaque cycle. Par suite, supprimer des sorties diminue les calculs.
Les autres sorties du programme à vérifier ne sont pas supprimées (la propriété peut dépendre de ces sorties) mais seulement déclarées en variables locales.

L'automate généré alors est toujours plus petit que l'automate calculant toutes les sorties en plus de la propriété, ou au pire, aussi important. En effet, lors de la génération du code sous forme d'automate, les variables d'état sont associées non pas à toutes les expressions en *pre* ou en *current* du programme, mais seulement aux expressions de ce type qui interviennent dans le calcul d'une des sorties. Il y a donc moins (ou autant) de variables d'état pour le calcul d'une seule sortie que pour le calcul de toutes les sorties. Par suite, le nombre d'états de l'automate est inférieur. Il y a donc gain de place mémoire, ainsi que de temps d'exécution puisque le parcourt des états de l'automate se fait plus rapidement.

Il convient maintenant de montrer que l'abstraction conserve les propriétés de sûreté.
Notons par ailleurs que c'est la même idée d'abstraction qui existe dans le rapport entre le graphe d'états et l'automate, mis à part que dans ce cas là, l'abstraction est réalisée par rapport aux variables non booléennes.

### 5.3.6.2 Conservation des propriétés de sûreté

Soit P le programme à vérifier, et S l'ensemble de ses sorties. Soit D l'ensemble de définition des mémoires associées à son arbre des exécutions.

Soit P' la version du programme P auquel a été incorporé le calcul de la propriété : P' = P + calcul de $V_p$. Les sorties de P' sont $S \cup \{V_p\}$. Le domaine de définition des mémoires associées à l'arbre des exécutions de P' est : $D \cup \{V_p\} \cup D'$ où D' est l'ensemble des variables nécessaires au calcul de $V_p$ et qui n'appartiennent pas au programme P (et donc à D).

Nous avons vu au paragraphe précédent que A' l'automate de P' est un modèle abstrait de P.

Partitionnons l'ensemble D en deux sous-ensembles disjoints $D_p$ et $D_{np}$ tels que :

- $D_p$ est l'ensemble des variables de D nécessaires au calcul de $V_p$,
- $D_{np}$ est l'ensemble des variables de D non nécessaires au calcul de $V_p$.

Définissons les ensembles $E_p$, $E_{np} \subseteq D$ et $E'_p \subseteq D'$ comme , respectivement, l'ensemble des variables d'états du programme P intervenant dans le calcul de $V_p$, l'ensemble des variables d'états du programme P n'intervenant pas dans le calcul de $V_p$, et l'ensemble des variables d'états du programme P' qui ne sont pas variables d'états du programme P et qui par conséquent intervenent dans le calcul de $V_p$.

Soit maintenant le programme P" qui est identique au programme P', à la seule différence qu'il n'a qu'une sortie : $V_p$ (les sorties S de P' sont pour P" des variables locales). Le domaine de définition des mémoires de l'arbre des exécutions de P" est le même que celui de P' : $D \cup \{V_p\} \cup D'$. Ces mémoires associent à tout élément de $D_{np}$ (variables non nécessaires au calcul de $V_p$) la valeur indéfinie : nil. En effet, ces variables n'intervenant pas dans le calcul de la sortie, ne sont pas calculées.

Ainsi, l'automate A" est le repliage de l'arbre étendu d'interprétation abstraite de P" suivant $E_p \cup E'_p$.Or la seule différence entre les arbres étendus d'interprétation abstraite des programmes P' et P" est la valeur associée par les mémoires aux variables booléennes de $D_{np}$. Il aurait donc été équivalent, pour obtenir l'automate A", de replier l'arbre étendu d'interprétation abstraite de P' suivant $E_p \cup E'_p$ (ce qui est égal au repliage de l'automate A' suivant $E_p \cup E'_p$) puis de faire abstraction des variables de $D_{np}$ en leur associant la valeur nil.

Par conséquent, toute propriété vérifiable sur l'automate A' sans nécessiter l'évaluation de variables booléennes de $D_{np}$ est vérifiable sur l'automate A", et réciproquement. La propriété toujours(E) est donc satisfaite par A' si et seulement si elle est satisfaite par A".

### 5.3.7 Utilisation des assertions

Les assertions LUSTRE permettent d'énoncer des propriétés invariantes de l'environnement du système. Grâce à elles, il va être possible de restreindre la vérification aux conditions réelles d'utilisation du programme et de supprimer des branches de l'arbre étendu d'interprétation abstraite ne correspondant à aucune exécution réelle du programme :
soit P le programme à vérifier, et toujours(E) la propriété qu'il doit vérifier, soit A les propriétés connues de l'environnement et décrites sous forme d'assertions, la vérification à réaliser est donc : sous les assertions A, le programme P satisfait la propriété toujours(E).

#### 5.3.7.1 Prise en compte des assertions

Nous allons maintenant décrire le mécanisme de prise en compte d'un ensemble d'assertions A dans la génération de l'automate d'un programme Pr.

Si on ne considère pas le rôle particulier que ces assertions ont à jouer, leur intégration peut être vue comme celle de la propriété décrite au paragraphe 5.3.4, c'est-à-dire qu'elle nécessite l'augmentation du domaine de définition des mémoires associées à l'arbre des exécutions de Pr, par l'ajout des nouvelles variables définies pour l'évaluation de ces assertions. On notera une telle vision du programme avec les assertions : Pr+A.

Le rôle des assertions est donc de décrire des faits qui sont toujours vérifiés. Il est donc inutile de réaliser les calculs du programme dans les cas où les assertions sont fausses.

Nous avons vu (cf paragraphe 2.2.1) que l'arbre des exécutions du programme $Pr\,|_A$ est restreint par rapport à celui de Pr+A : il ne conserve que les états dans lesquels les assertions sont vraies.

En théorie, l'automate du programme Pr sous les assertions A est le résultat du repliage de l'arbre étendu d'interprétation abstraite de $Pr\,|_A$ suivant les expressions booléennes en *pre* ou en *current* intervenant dans le calcul des sorties (ensemble $E_{Pr}$) ou dans la calcul des assertions (ensemble $E_A$).

Ceci équivaut en pratique à la réalisation suivante : le compilateur choisit les variables d'état comme s'il allait réaliser l'automate de Pr+A, c'est-à-dire toutes les variables de ensemble $E_{Pr} \cup E_A$. Dans chaque état (combinaison des variables d'état) accessible, il évalue les assertions. Il génère

ensuite le code de l'état (et notamment l'accès aux états suivants) en supprimant tout ce qui dépend de conditions qui invalident les assertions.

Si de plus, lors de la génération d'un état accessible q, les assertions sont fausses dans cet état sous n'importe quelle condition, q est supprimé. De même, tout état q' tel que $\forall q, q' \rightarrow^t q \Rightarrow q \not\models A$ (où $\rightarrow^t$ est la fermeture transitive de la relation ->) est supprimé. Cela est réalisé par retour arrière ("backtracking") sur les prédécesseurs de ces états q.

Des assertions pour lesquelles il existe dans l'automate de tels états q sont dites "non causales", dans le sens où l'histoire de leur flux ne suffit pas pour savoir dans un état si un état successeur est accessible ou pas. Nous avons donc rapproché ceci au principe de causalité (défini au chapitre 1) que respecte tout programme LUSTRE.

Nous allons voir maintenant comment cet automate intervient dans la vérification de propriétés.

### 5.3.7.2 Vérification de propriétés sous des assertions

Dans ce paragraphe nous considérons que le programme Pr évoqué ci-dessus est en fait un programme P auquel a été incorporée la propriété toujours(E) sous la forme décrite au paragraphe 5.3.4. Les sorties de Pr sont indifféremment la variable correspondant à la propriété, seule ou avec les sorties de P.

Nous avons montré aux paragraphes 5.3.4.2 et 5.3.5.2 que :
si l'expression E est évaluée à vrai dans tout état de l'automate de Pr, alors $P \models$ toujours(E) .

Vérifier que, sous les assertions A, E est évaluée à vrai dans tout état de l'automate de Pr, vérifie donc que, sous les assertions A, $P \models$ toujours(E).

Or, de vérifier que sous les assertions A, E est évaluée à vrai dans tout état de l'automate de Pr, revient à vérifier que E est évaluée à vrai dans tout état de l'automate calculant Pr+A qui vérifie A (rappelons que l'automate de Pr+A est modèle de Pr). Et d'après ce qui précède (paragraphe 5.3.6.1), cela revient à vérifier que E est évaluée à vrai dans tout état de l'automate de $Pr\,|_A$, puisqu'il est équivalent de supprimer de l'automate de Pr+A ce qui équivaut aux états de $Q|_A$, avant ou après l'évaluation de E.

Nous vérifierons donc E dans tout état de $Pr\,|_A$ pour vérifier que sous les assertions A, $P \models$ toujours(E).

#### 5.3.7.3 Assertions : aide à la manipulation arithmétique

Nous avons annoncé que les assertions permettent de prendre en compte des propriétés de l'environnement. Elles permettent aussi de spécifier des propriétés de domaines connus tels que $\mathbb{N}$ ou $\mathbb{R}$. De ce fait, elles permettent de restaurer des renseignements perdus lors de l'abstraction.

Par exemple dans le cas de SIREX, les seuils de gestion des capteurs ou de déclenchement des alarmes sont des constantes de type réel, et sont donc liés par une relation d'ordre. Cette relation a été définie sous forme d'assertions. Voilà par exemple celle qui définit la relation entre les seuils MinBN et Hy_MinBN :

**assert** IMPLIQUE ( (flux < MinBN), **not** (flux > Hy_MinBN) )

Définir de telles assertions nécessite un minimum de connaissances du programme ou le respect d'une convention de programmation. En effet, le compilateur ne gère les expressions (flux < MinBN) et (flux > Hy_MinBN) qu'en tant que booléens (abstraction). Il aurait été équivalent de son point de vue, de les remplacer par des variables booléennes de nom quelconque. Le compilateur gère ainsi un certain nombre de *conditions* booléennes comme les comparaisons et les appels à des fonctions externes booléennes. Il connait ces conditions sous une syntaxe particulière et toute syntaxe différente est pour lui une condition différente. Ainsi, la condition (flux < MinBN) n'est pas pour lui la condition (MinBN > flux).

Il faut donc pour que les assertions puissent servir à supprimer des chemins dans l'automate, que les conditions qui apparaissent dans les assertions soient les mêmes **syntaxiquement** que celles qui gèrent l'accès aux états successeurs dans le code d'un état, et qui apparaissent, elles, dans le programme.

Ce léger inconvénient dans l'utilisation des assertions peut être pallié par la définition d'un standard de programmation : on peut par exemple décider de n'utiliser dans le cas des comparaisons booléennes que les signes < et =, et de décrire les relations de supériorité et de différence en terme d'infériorité et d'égalité.

Par ailleurs, les assertions ne doivent être utilisées qu'à bon escient et nous allons voir pourquoi.

#### 5.3.7.4 Influence des assertions sur la taille des automates

Nous avons parlé du fait que les variables d'état choisies lors de la génération de l'automate de $Pr\,|_A$ contiennent les expressions booléennes en *pre* ou en *current* qui interviennent dans le

calcul des assertions. Cela correspond au programme Pr+A et est naturel puisque nécessaire à l'évaluation des assertions.

Ainsi, le nombre d'états de l'automate de Pr+A est au moins aussi important que celui de l'automate de Pr. C'est pourquoi, dans certains cas, le gain en nombre d'états et en volume de code produit pour un état, apporté par l'utilisation des assertions, est compensé, voire inversé, par la présence d'un nombre important d'opérateurs *pre* ou *current* dans la formulation de ces assertions.

Dans de tels cas, l'intérêt des assertions ne réside plus dans l'amélioration de l'automate, mais bien dans la compréhension par l'outil de vérification des rapports existant entre les variables, comme par exemple les rapports arithmétiques évoqués dans le paragraphe 5.3.6.3.

### 5.3.8 Vérifications modulaires

Il est possible de vérifier des propriétés sur des nœuds internes d'un programme, et d'utiliser ensuite les connaissances acquises sur ces nœuds pour prouver d'autres propriétés sur le programme entier.

Soit P un programme d'entrées $\vec{e}_P$ et de sortie $\vec{s}_P$. Soit N un nœud interne de P qui a pour paramètres formels : en entrée le tuple $\vec{x}$, en sortie le tuple $\vec{v}$.

Il est possible de prouver des propriétés sur N en faisant abstraction de son environnement dans P. Le nœud N est alors vu vomme un programme d'entrées $\vec{x}$ et de sorties $\vec{v}$.

Soit $F(\vec{x}, \vec{v})$ une propriété prouvée sur N sous l'assertion $A(\vec{x})$, nous allons nous servir de F pour prouver des propriétés sur P : tout appel de N va être remplacé par l'assertion F correspondante.

En effet, pour tout appel $N(\vec{e}_N)$, N va être considéré comme une partie de l'environnement de P, ces entrées $\vec{e}_N$ vont être des sorties de P, et $N(\vec{e}_N)$ va être une entrée de P. Pour ce faire, l'appel $N(\vec{e}_N)$ va être remplacé par un tuple $\vec{s}_N$ rajouté en entrée de P. Notons P\N le programme P ainsi modifié. Le corps du nœud N est simulé dans P\N par l'assertion sur les entrées/sorties de P\N : assert $F(\vec{x}, \vec{v})$.

Nous allons alors montrer que sous l'assertion $B(\vec{e}_P)$ et sous l'assertion $F(\vec{e}_N, \vec{s}_N)$, le programme P\N satisfait, d'une part, la propriété de l'environnement N : $A(\vec{e}_N)$ (afin que la preuve modulaire soit valide), et d'autre part, la propriété que l'on désire vérifier sur P : $G(\vec{e}_P, \vec{s}_P)$ :

$P|_a \models A(\vec{e}_N) \wedge G(\vec{e}_P, \vec{s}_P)$ où $a = B(\vec{e}_P) \wedge F(\vec{e}_N, \vec{s}_N)$

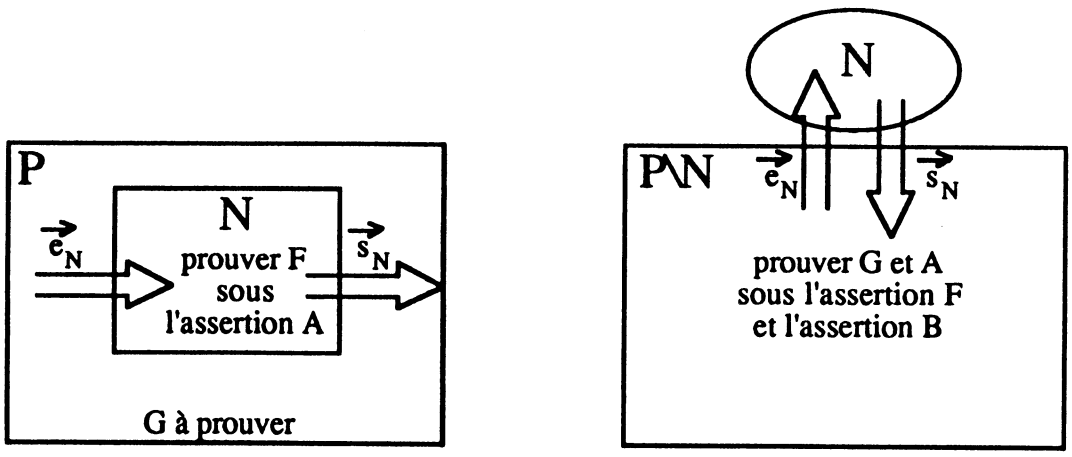



Figure 5.3.8.a : preuve modulaire

L'intérêt d'une telle décomposition dans la vérification est d'éviter l'expansion du programme P tout entier. Le processus décrit ci-dessus pour P et N peut être itéré si P est lui-même appelé par un nœud englobant.

## 5.4 CONCLUSION

Nous avons donc présenté dans ce chapitre une adaptation de la méthode de vérification de l'outil XESAR à la validation de programme LUSTRE, adaptation qui permettrait la vérification des propriétés directement sur le graphe d'états quotient de l'arbre d'exécution du programme, et dont le langage d'expression des propriétés couvrirait les modalités du langages CL présenté au chapitre 4, et en plus l'opérateur de passé immédiat qui est le *pre* de LUSTRE.

Nous avons ensuite restreint notre approche de validation à un langage d'expression des propriétés plus spécifiques qui ne permet que l'expression d'invariants en LUSTRE (propriété de safety au sens de [AS, 85]), mais pour lequel nous avons développé une méthode de vérification originale.

L'utilisation de LUSTRE comme langage de spécification a plusieurs avantages :

**il est bien adapté à la traduction des constructions apparues dans les études de cas :**

LUSTRE est logique du passé, son opérateur *pre* permet d'appréhender la construction "*au cycle précédent*".

Toutes les propriétés de sûreté [AS,85] sont traduisibles en LUSTRE augmenté d'un opérateur

d'invariance : toujours. La notion d'invariance est d'ailleurs intrinsèque à la notion d'équation LUSTRE. En effet, l'équation X=E signifie : à tout moment, dans tout contexte, la variable X a la même valeur et la même horloge que l'expression E. C'est cette notion qui permet d'incorporer la propriété au programme .

**LUSTRE permet de ne pas figer le langage de spécification à ces seules constructions :**

Il est possible à la personne chargée des spécifications, de définir, grâce au mécanisme de "nœud" en LUSTRE, ses propres opérateurs prédicatifs ou temporels, et donc de ne pas se restreindre à des constructions prédéfinies.

Par exemple, exprimons qu'une alarme doit être déclanchée après l'arrivée d'un événement E dans un temps limité à 4 cycles :

toujours exist alarme entre E et CHIEN_DE_GARDE(E,5)

où CHIEN_DE_GARDE est un nœud LUSTRE défini comme suit :

```
node CHIEN_DE_GARDE( E:bool; nb_cycles : int ) .
   returns (fin : bool) ;
var cpt : int ; reset : bool ;
let
   fin = (nb_cycles = cpt ) ;
   reset = E->if (pre(cpt)<nb_cycles) then false else E ;
   cpt = if reset then 1 -> COMPTEUR(reset,1,1) else (nb_cycles+1)
tel.
node COMPTEUR( reset:bool; val_init,incr : int )
   returns(cpt : int) ;
let
   cpt = ->if reset then val_init else pre(cpt)+incr ;
tel.
```

Remarquons que la définition du nœud CHIEN_DE_GARDE a été réalisée de sorte que toute occurrence de l'événement E arrivant dans un délai de 4 cycles après le déclenchement du chien de garde, n'est pas prise en compte.

**il est adapté à la vérification :**

Les assertions LUSTRE permettent de simuler le comportement de l'environnement du système (assertions sur les entrées du programme), et donc de restreindre la vérification aux conditions

réelles d'utilisation du programme. Les assertions permettent aussi d'envisager des méthodes de preuves modulaires. La formalisation des propriétés en LUSTRE facilite la vérification par évaluation sur des modèles : ces propriétés peuvent être incoporées aux programmes afin que leur valeur apparaisse directement dans tout état de l'automate.

**il est adapté à l'abstraction :**
La génération de l'automate peut être réalisée en s'abstrayant automatiquement de toutes les équations du programme qui n'influencent pas le calcul de la propriété. La vérification va ainsi "droit au but" sur un automate restreint au nécessaire.

**il est basé sur les mêmes principes que le langage de programmation** :
LUSTRE, comme nous l'avons vu au chapitre 1, est un langage flots de données synchrone, similaire en cela à SAGA. Ainsi la personne chargée de la conception SAGA ou LUSTRE du programme, peut s'appuyer sur les spécifications pour réaliser cette conception, sans faire d'efforts d'adaptation entre deux modèles de pensée différents. Cette facilité pour le programmeur ne peut que conduire à une amélioration de la qualité du logiciel.

# 6 Expérimentation

Nous allons présenter maintenant les expériences pratiques que nous avons menées pour la vérification de l'exemple tiré de l'application SIREX et les conclusions que nous en avons tirées.

La première partie de ce chapitre présente l'utilisation du compilateur LUSTRE pour la génération de l'automate. Certains problèmes, que nous allons détailler, ont conduit à la réalisation d'un prototype de vérificateur LESAR qui évalue la propriété au cours de la génération de l'automate. Ces expériences, nous ont permis de classer les erreurs rencontrées en quatre catégories; cela fera l'objet de la troisième partie de ce chapitre. La dernière partie retrace ces expériences sous forme de tableaux permettant de mettre en évidence la rapidité de la vérification par LESAR.

## 6.1 UTILISATION DU COMPILATEUR LUSTRE POUR LA GÉNÉRATION DE L'AUTOMATE

Cette première expérience peut être vue comme une étude de faisabilité. Le programme SAGA réalisant l'extrait de SIREX, a été traduit en LUSTRE à la main, en respectant la sémantique.

Les propriétés ont été introduites dans ce programme, séparément, une par une, suivant la méthode décrite dans le chapitre 5. Cela a été réalisé manuellement. Le compilateur LUSTRE a ensuite été utilisé pour produire l'automate du programme contenant la propriété, définie comme une sortie.

La vérification consiste à s'assurer, dans chaque état, que la valeur associée à la propriété est vraie.

### 6.1.1 Problème : automate non minimal

Lors de cette vérification, un premier problème est apparu : certains états de l'automate étaient redondants. En effet, l'automate est le repliage de l'arbre d'interprétation abstraite suivant la bisimulation $\mathfrak{R}_A$ définie au paragraphe 2.2.3.2. Cette bisimulation n'est pas en général la plus grande bisimulation contenue dans $\approx$, où $q \approx q'$ ssi le code généré pour q et q' est identique.

En effet, prenons par exemple le nœud suivant:

```
node exemple (x, y : bool) returns (z : bool);
let
   z = true -> if pre(x) then true else pre(y);
tel.
```

La sortie z dépend de deux mémoires pre(x) et pre(y) qui deviendront variables d'états à la génération de l'automate. Or, lorsque pre(x) est vrai, z ne dépend pas de pre(y). Néanmoins, les deux états caractérisés par les combinaisons des trois variables d'état init, pre(x) et pre(y) : (false, true, true) et (false, true, false) seront générés -car ils sont accessibles-, et leur code sera alors identique : il consistera à mettre la variable z à la valeur *true*.

Pour pallier cet inconvénient, le compilateur LUSTRE a été interfacé avec le logiciel ALDEBARAN [Fer,88] développé au sein de l'équipe SAS du LGI. Ce logiciel réalise la minimisation d'automates selon différents critères d'équivalence.

### 6.1.2 Minimisation de l'automate

Nous présentons ici la représentation concrète de l'automate de contrôle et la transformation qui doit être réalisée sur cet automate pour pouvoir réaliser la minimisation à l'aide d'ALDEBARAN.

#### 6.1.2.1 Représentation de l'automate de contrôle

L'automate généré lors de la compilation d'un programme LUSTRE est décrit selon un format appelé OC [CSP,87]. La structure de l'automate est exprimée en terme d'états et de transitions exécutables entre ceux-ci :

- Chaque état est dénoté par un entier naturel.
- L'état initial est unique: c'est l'état 0.
- A chaque état est associée une liste d'actions codées par des entiers naturels. Celle-ci définit une relation de transition entre cet état et les autres états de l'automate. La syntaxe des transitions est donnée par la grammaire ci-dessous.

**Remarque**: les symboles terminaux de la grammaire sont en gras.

```
/* transition */
   trans ::= act trans | test_act (trans) (trans) | goto_act.
/* action quelconque */
   act ::= test_act () (l_act) | test_act (l_act) () |
           test_act (l_act) (l_act) | test_act (l_act) (l_act) l_act | aff_act.
/* liste d'actions quelconques */
   l_act ::= act l_act | act.
/* action "goto" */
   goto_act ::= 0 state_index.
/* actions d'affectation et autres, actions de test, nom d'états */
   aff_act, test_act, state_index ∈ N.
```

Dans cette grammaire, toute dérivation d'un non terminal "trans" aboutit à une action de branchement sur un état de l'automate: "goto *état*". Pour un état et une combinaison des entrées donnés, la séquence d'actions exécutées dans cet état aboutit au branchement vers un état successeur unique, car les actions de test peuvent être évaluées selon la valeur des entrées et sont donc déterministes. Ainsi la liste d'actions associées à un état permet d'atteindre plusieurs états différents, en fonction de la valeur retournée par les actions de test. La liste d'actions associée à chaque état définit donc une sorte de transition "arborescente".

**Exemple:**

Soit la liste d'actions: 2 5 6 (1 3 0 1) ( 4 7 (0 2) (2 0 3) ) partant de l'état 4. Elle correspond à l'exécution d'une transition "arborescente" qu'on peut représenter comme ci-dessous:

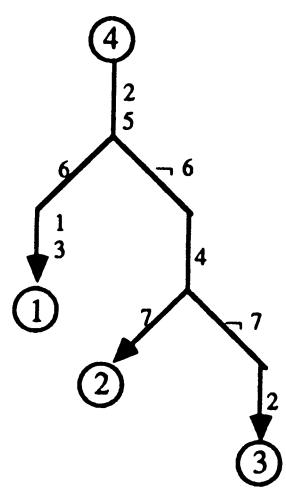



Figure 6.1.2.1.a : exemple de transition "arborescente"

Chaque transition "arborescente" est en fait un ensemble de transitions "linéaires". La "mise à plat" d'une telle transition est réalisée via l'opérateur MP et sera nécessaire lors de l'utilisation d'ALDEBARAN :

MP (trans) :

MP (act trans) = MP (act) MP (trans)

MP (**test_act** ($trans_1$) ($trans_2$)) = { **test_act** MP ($trans_1$), **test_act n** MP ($trans_2$)}

MP (goto_act) = goto_act

MP (act) :

MP (**test_act** () (l_act)) = { **test_act, test_act n** MP (l_act)}

MP (**test_act** (l_act) ()) = { **test_act** MP (l_act), **test_act n**}

MP (**test_act** ($l\_act_1$) ($l\_act_2$)) = { **test_act** MP ($l\_act_1$), **test_act n** MP ($l\_act_2$)}

MP (**test_act** ($l\_act_1$) ($l\_act_2$) $l\_act_3$) = { **test_act** MP ($l\_act_1$) MP ($l\_act_3$),
**test_act n** MP ($l\_act_2$) MP ($l\_act_3$)}

MP (aff_act) = aff_act

MP (l_act) :

MP (act l_act) = MP (act) MP (l_act)

MP (act)

Les noms terminaux **test_act** représentent maintenant des tests sans "sinon". Si **test_act** est suivi de **n**, c'est l'inverse de la condition apparaissant dans l'expression liée à **test_act** qu'il faut maintenant tester.

Les nouvelles transitions (linéaires) ne sont tirées que lorsque les réponses apportées à tous les tests s'avèrent positives. Les tests représentant les conditions booléennes que nous avons citées au paragraphe 2.2.3.2, lors de l'exécution, une seule transition linéaire sera tirable dans l'état courant.

Ainsi, dans l'exemple du paragraphe précédent, la transition arborescente de la figure 6.1.2.1.a regroupe les 3 transitions linéaires suivantes :

2 5 6 1 3 0 1

2 5 6n 4 7 0 2

2 5 6n 4 7n 0 3

où les actions 6, 6n, 7 et 7n représentent maintenant des tests sans "sinon".

L'automate de contrôle peut donc être vu comme un système de transitions étiquetées : S = (Q, A, T, 0), où :

- Q : ensemble des états de l'automate (tous sont atteignables à partir de l'état 0)
- A : ensemble d'étiquettes. Les étiquettes sont composées de la suite des actions des transitions "mises à plat", séparées par le caractère "_".
- T : relation de transition étiquetée par A : $T \subseteq QxAxQ$
- 0 : état initial du système

### 6.1.2.2 Utilisation d'ALDEBARAN

La première utilisation du logiciel ALDEBARAN qui a été réalisée pour la minimisation de l'automate prend en compte la relation de congruence forte sur un système de transitions. Deux états sont liés par cette relation lorsqu'on peut dérouler les mêmes séquences d'actions à partir de ces états (pas de notion d'observation). La forme normale d'un système de transitions étiquetées, modulo la congruence forte, est obtenue en calculant la plus grande bisimulation sur Q définie comme suit (la bisimulation définie au chapitre 2 est étendue aux systèmes de transitions et prend en compte la notion d'action liée à une transition) :

Soit deux systèmes de transitions étiquetés $S_i = (Q_i, A_i, T_i, q_i)$ i = 1, 2,
et soit $\rho \subseteq Q_1xQ_2$,
$\rho$ est une bisimulation ssi :

1) $(q_1, q_2) \in \rho$
2) $\forall (p_1, p_2) \in \rho, \forall a \in$ Act (ensemble d'étiquettes tel que $A_i \subseteq$ Act),
$\forall p'_1 \in Q_1, (p_1, a, p'_1) \in T_1 \Rightarrow \exists p'_2 \in Q_2, (p_2, a, p'_2) \in T_2 \wedge (p'_1, p'_2) \in \rho$
$\forall p'_2 \in Q_2, (p_2, a, p'_2) \in T_2 \Rightarrow \exists p'_1 \in Q_1, (p_1, a, p'_1) \in T_1 \wedge (p'_1, p'_2) \in \rho$

L'automate de contrôle minimisé est obtenu en plusieurs passes. Chaque passe consiste en trois phases :

- phase 1 : "mise à plat" des transitions (cf paragraphe 6.1.2.1)
- phase 2 : appel d'ALDEBARAN avec la liste des transitions résultant de la phase 1
- phase 3 : à l'aide des classes d'équivalence obtenues en phase 2, suppression des états redondants et simplification des transitions arborescentes.

La simplification des transitions arborescentes consiste à supprimer les tests (**test_act**) devenus inutiles parce que les branches "alors" et "sinon" associées conduisent à des états de même classe d'équivalence, par la même succession d'actions.

En sortie de chaque passe, on obtient un nouvel automate de contrôle (écrit en code portable) sémantiquement équivalent au précédent. Cet automate est l'entrée de la passe suivante. Le processus est arrêté lorsque le nombre d'états arrive à stabilisation.

Les phases 1 et 2 ont été programmées en C++. Un fichier de commandes "shell" sous UNIX réalise la boucle d'appel.

Cette première approche de la minimisation des automates de contrôle, bien que très satisfaisante, ne produit pas toujours l'automate minimal préservant la sémantique du programme. En effet, soit l'automate à deux états (Figure 6.1.2.2.a) :

0 : 4 (2 0 1) (2 0 0)

1 : 2 0 1

la relation de congruence forte ne permet pas d'associer les états 0 et 1 dans la même classe d'équivalence, puisque les étiquettes des transitions partant de l'état 0 : 4_2_ et 4n_2_ diffèrent de celles des transitions partant de l'état 1 : 2_

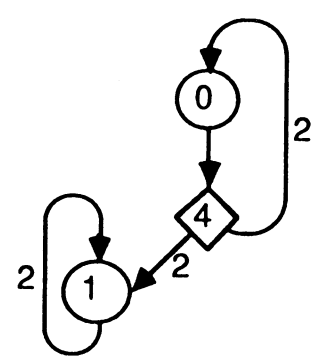



Figure 6.1.2.2.a : exemple d'automate non minimal

C'est pourquoi, une deuxième approche a été étudiée. Elle consiste à définir une nouvelle relation d'équivalence prenant en compte les caractéristiques des automates OC (signification des actions), et à incorporer cette relation comme nouvelle option d'ALDEBARAN.

Dans l'exemple représenté en figure 6.1.2.2.a, on ne peut minimiser l'automate qu'en considérant à priori que les états 0 et 1 sont équivalents. C'est le principe qui a été adopté pour la minimisation. L'automate à minimiser est toujours gardé en mémoire comme référence. Dans un premier temps, tous les états sont considérés équivalents. Cette hypothèse permet de supprimer des états redondants et de simplifier les transitions arborescentes. ALDEBARAN fournit alors, à partir de ces nouvelles transitions, de nouvelles classes d'équivalence.

Revenant à l'automate de référence, ces nouvelles hypothèses permettent de supprimer des états redondants et de simplifier les transitions arborescentes. Le processus est alors itéré jusqu'à stabilisation du nombre d'états. Ainsi, à chaque passe, le processus est recommencé sur l'automate de référence (automate à minimiser), avec, à chaque fois, des hypothèses différentes (de plus en plus précises) concernant l'équivalence des états.

Cette méthode de minimisation a été implantée par J-C. Fernandez et N. Halbwachs, et forme l'outil OCMIN. OCMIN est utilisable pour tout automate en format OC, et donc pour les automates générés par le compilateur ESTEREL [BMR,83].

Grâce à cette minimisation, la recherche des états et des conditions dans ces états, pour lesquels la propriété était fausse, se trouve nettement raccourcie. De même, le chemin parcouru pour arriver à de tels cas, devient plus court et plus simple (suppression de tests inutiles), ce qui rend plus aisé un diagnostic d'erreur.

De plus, en général, l'automate généré pour une propriété vraie (lorsque la propriété est la seule sortie) se réduit à un seul état.

### 6.1.3 Problème de mémoire

Malgré tout, la minimisation de l'automate ne résoud pas entièrement le problème de redondance des états. En effet, dans certains programmes, le nombre de variables d'état est très important. C'est en général le cas pour les gros logiciels qui manipulent un grand nombre de données et dont le code séquentiel est composé d'un grand nombre d'actions. Pour de tels programmes, l'automate de contrôle est si volumineux que la mémoire allouée à l'utilisateur lors de sa génération, est insuffisante pour le contenir tout entier. C'est le cas par exemple de la version LUSTRE du logiciel inspiré de l'application SIREX, dont les propriétés sont énoncées au chapitre 3 et que nous avons vérifiées. Une minimisation de l'automate à postériori ne peut alors résoudre le problème.

C'est pourquoi, des études sont en cours dans l'équipe LUSTRE concernant :

- d'une part, une minimisation de l'automate en cours de génération
- d'autre part, la production d'un automate directement minimal par une nouvelle méthode de génération :
  - **"remontée" automatique des opérateurs *pre*** qui permet de diminuer le nombre de variables d'état en factorisant les opérateurs *pre* dans les expressions. Cette méthode est basée sur les propriétés de distributivité de l'opérateur *pre* . Par exemple :

    pre(a) and pre(b) = pre(a and b).

    Dans ce cas, alors que l'expression de gauche fournit deux variables d'état (si a et b sont

booléens), l'expression de droite n'en fournit plus qu'une. L'étude de cette méthode fait partie du travail de thèse de P. Raymond.

- **compilation dirigée par la demande** [HPP,89] qui consiste à ne calculer dans un état donné que les variables booléennes effectivement nécessaires au calcul des sorties ou des états suivants.

Pour ma part, j'ai résolu de me restreindre à des vérifications sur des automates ne représentant que des fragments du programme. Nous qualifierons cette approche de vérification modulaire.

### 6.1.4 Vérification modulaire

Nous avons évoqué l'idée d'une preuve modulaire dans le chapitre précédent. Voyons maintenant comment nous avons pu l'appliquer à SIREX.

Cette application a la propriété intéressante d'être affiné au niveau le plus haut (figure 6.1.4.a) en trois sous-fonctions n'interférant que très peu les unes avec les autres.

La fonction Calcul_des_mesures ne produit que des valeurs réelles (flux_valide, période,...). Celles-ci serviront d'entrées aux deux autres fonctions. Cette première fonction ne contient pas de propriétés intéressant la sûreté.

Les deux autres fonctions peuvent être vérifiées indépendamment l'une de l'autre car elles réalisent des calculs entièrement disjoints (d'une part le calcul du mode de fonctionnement des capteurs, de l'autre, la gestion des alarmes).
Aucune assertion sur le comportement de l'une des deux boîtes n'a été nécessaire lors de la validation de l'autre. Cette vérification n'est donc qu'une version simplifiée de la preuve modulaire évoquée au chapitre 5.

Nous avons combiné cette approche de modularité avec la vérification par abstraction définie au paragraphe 5.3.5, en ne conservant donc en sortie de ces deux fonctions que la propriété à vérifier.

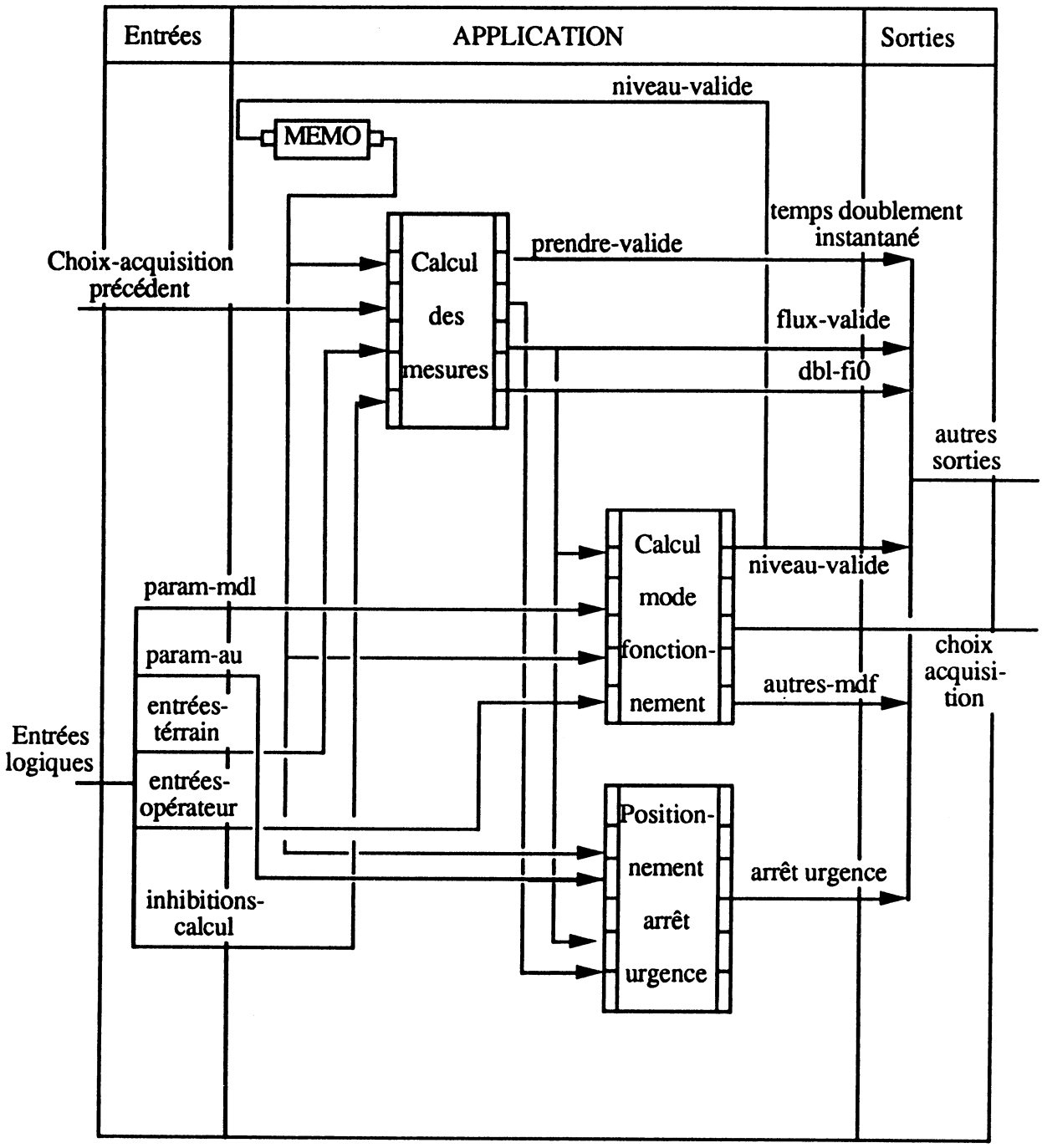



Figure 6.1.4.a : première vue SAGA de SIREX

### 6.1.5 Conclusion

Malgré les solutions apportées pour minimiser la taille de l'automate : abstraction et modularité lors de la génération de l'automate, utilisation du minimiseur ALDEBARAN sur l'automate généré, celui-ci est parfois encore trop important pour être contenu dans la mémoire. C'est pourquoi, nous avons pensé réaliser la vérification au fur et à mesure de la génération des états, et ne conserver en mémoire qu'un seul état à la fois. Ceci est possible :

- car nous vérifions des propriétés de type toujours(E) où E est une expression LUSTRE booléenne.
- grâce à la bisimulation choisie pour replier l'arbre d'interprétation abstraite en automate.

En effet, toujours(E) signifie : E est vraie dans tout état de l'automate. Or, l'automate est généré avec l'expression E comme sortie. Celle-ci est donc évaluable dans chaque état de l'automate **indépendamment des autres états** grâce au mécanisme des variables d'état qui mémorisent le passé nécessaire au calcul de toutes les sorties, et donc de E, à chaque cycle.

L'approche suivie, à savoir : le flot de données, le synchronisme, le choix d'un langage assimilable à une logique [PH,88], a donc permis cette vérification "à la volée".

## 6.2 LÉSAR : PROTOTYPE POUR LA VÉRIFICATION DE PROPRIÉTÉS EN COURS DE GÉNÉRATION

Au lieu de réaliser un vérificateur prenant en entrée l'automate de contrôle, nous avons réalisé un prototype de vérificateur, LESAR, qui génère lui-même chaque état de l'automate à partir du programme. Lors de la génération de chaque état, LESAR évalue la valeur de la propriété et émet un message d'erreur associé à un diagnostic si cette valeur est fausse.

Dès que la propriété est invalidée dans un état, le prototype s'arrête. Il fournit donc une cause d'erreur mais pas forcément toutes les causes possibles. Lorsque tous les états ont ainsi été passés en revue, si l'expression n'a jamais été fausse, un message indique que la propriété est vraie. Cet outil prend en entrée un programme LUSTRE mis à plat. La traduction automatique d'un programme SAGA en un programme LUSTRE expansé est en cours de réalisation à Merlin Gerin, LESAR pourra donc être utilisé dans un futur proche pour la vérification de propriétés sur des programmes SAGA.

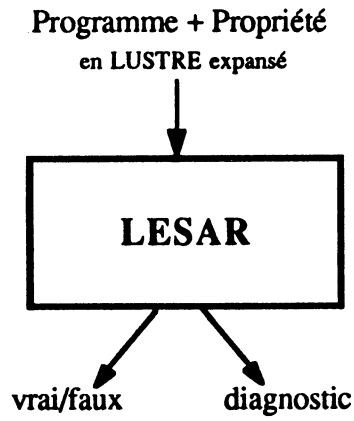



Figure 6.2.a : LESAR

La même idée de vérification en cours de génération est trouvée dans [JJ,89]. Le langage d'expression des propriétés utilisé dans ce cas est une logique temporelle linéaire interprétée en terme de séquences finies d'états. La propriété à prouver (formule de la logique) est transformée en un automate reconnaisseur de séquences. Le graphe d'états représentant le programme est alors généré et chaque nouvel état est validé (ou pas) par l'automate reconnaisseur.

La seule innovation de LESAR est l'idée de vérification en cours de génération. Le principe de génération de l'automate est repris du compilateur LUSTRE (cf paragraphe 1.3.3.3.2) :

- choix des variables d'état et de condition,
- génération des états avec évaluation dans chaque état des valeurs de sortie et des valeurs représentant les états successeurs.

La plupart des modules utilisés par le prototype proviennent du compilateur V3 en cours de développement par P. Raymond dans le cadre de son travail de thèse. Ces modules sont écrits en C++. Ils créent la représentation interne du programme LUSTRE sous forme d'un réseau d'opérateurs, à partir duquel va être élaboré l'automate. Ils choisissent sur cette représentation les variables d'état et les variables conditions. Ils évaluent les états accessibles à partir d'un état donné.

En collaboration avec P. Raymond, j'ai développé un nouvel évaluateur d'expressions qui permet de prendre en compte les assertions en cours d'évaluation plutôt qu'à postériori. Cet évaluateur est utilisé à la fois dans la phase d'évaluation des sorties et celle d'évaluation des états successeurs. J'ai ensuite amélioré le prototype pour conserver certains chemins de l'automate afin de permettre un diagnostic. Ces deux aspects sont présentés succintement ci-après.

### 6.2.1 Evaluation sous des conditions

L'évaluation d'une expression est réalisée en sachant que certaines conditions sont vraies. Cela revient à dire que ces conditions sont remplacées par "true" dans le sous-réseau lié à l'expression.

Cette forme d'évaluation est intéressante pour prendre en compte les assertions. En effet, l'automate doit être restreint afin de vérifier partout ces assertions. Par conséquent, dans un état, l'expression correspondant à la propriété ne sera évaluée que sous les conditions qui valident les assertions.

D'autre part, les variables d'état (dont l'évaluation fournit les états accessibles à partir de l'état courant) seront aussi évaluées suivant le même principe, ceci afin que l'accession aux états successeurs respecte les assertions de l'état courant.

### 6.2.2 Diagnostic d'erreur

Lorsque, dans l'état courant, l'évaluation des variables d'état produit un nouvel état (état non rencontré au cours d'évaluations antérieures), celui-ci est rangé dans l'ensemble des états accessibles (ensemble des états faisant partie de l'automate et qui n'ont pas encore été explorés). On mémorise avec lui l'état à partir duquel on peut l'atteindre (état courant en l'occurrence) et les conditions nécessaires à la transition. Même si cet état est ensuite successeur d'autres états, aucune nouvelle information ne sera enregistrée à son sujet. Ainsi, pour chaque état accessible à partir de l'état initial, nous ne conservons qu'un des chemins y parvenant. Ce chemin est étiqueté par les conditions nécessaires aux transitions.

Lorsqu'un état invalide la propriété à vérifier, le chemin d'accès à cet état est fourni comme diagnostic d'erreur.

Nous verrons des exemples du diagnostic émis par LESAR dans le paragraphe 6.3 pour illustrer les types d'erreurs que nous avons rencontrés au cours de la validation de notre exemple extrait du logiciel SIREX.

### 6.2.3 Limitations

#### 6.2.3.1 Une seule propriété à la fois

Notre outil ne permet de vérifier qu'une propriété à la fois. Elle devra être insérée comme première sortie du programme à vérifier. Seule cette propriété est évaluée. Les autres sorties ne le sont pas mais participent à l'élaboration des variables d'états et des conditions; il est ainsi préférable de n'avoir pour toute sortie que cette propriété.

Malgré tout, cette limitation n'est pas trop importante. En effet, pour vérifier à la fois deux propriétés toujours(X) et toujours(Y), il suffit d'insérer dans le programme la variable $V_p$ correspondant à "X and Y", sachant que:

toujours(X) et toujours(Y) = toujours(X and Y).

D'autre part, nous considérons qu'il est plus intéressant de vérifier séparément chaque propriété car, dans l'hypothèse où l'une d'entre elles est fausse, le diagnostic est plus aisé. Dans ce cas, en effet, l'automate est généré à partir d'un nombre de variables d'état, en général, moins important; les chemins qu'on en extrait sont donc plus concis.

#### 6.2.3.2 Assertions "non causales"

Par notre méthode d'évaluation "à la volée", nous ne pouvons traiter correctement les assertions "non causales" (cf paragraphe 5.3.6.1). En effet, lorsqu'on arrive dans un état, on ne connait qu'un seul de ses prédécesseurs (celui que nous avons conservé pour le diagnostic). Il n'est donc pas possible de faire un retour sur les prédécesseurs d'un état où les assertions sont fausses à cause des variables d'état (ce qui est réalisé pour la génération de l'automate à l'aide du compilateur - paragraphe 5.3.6.1 - ), ni de retrouver, pour le prédécesseur connu, tous ses états successeurs. De ce fait, on ne peut déterminer les états qui doivent aussi être supprimés. Ainsi, lorsque nous obtenons un état invalidant totalement les assertions, nous arrêtons le processus en spécifiant que les assertions sont "non causales".

Mais, cette limitation va plus loin : lorsque LESAR s'arrête en annonçant que la propriété est fausse, si les assertions sont non causales, nous ne sommes pas sûrs que la propriété soit effectivement fausse. Nous pourrions être dans un état ne menant qu'à des états invalidant les assertions, état qui serait supprimé si un retour était possible. Par conséquent la propriété pourrait en fait être vraie sous les assertions fournies. On sait par contre qu'elle serait fausse sans assertion.

Par conséquent, une propriété déclarée vraie par LESAR l'est effectivement. Une propriété déclarée fausse par LESAR peut être vraie si les assertions sont non causales ou si la propriété ne peut être évaluée après abstraction des expressions non booléennes.

## 6.3 TYPES D'ERREUR

Lorsque LESAR s'arrête avec pour diagnostic "propriété fausse", cela peut provenir de cinq sources différentes :

- le programme est effectivement incorrect.
- la propriété est incorrecte.

  Les spécifications d'un logiciel sont en général informelles, les cahiers des charges rédigés en français. Lors de l'expression formelle des propriétés, des erreurs peuvent survenir parce que l'énoncé de ces propriétés dans les cahiers des charges est ambigu. Une mauvaise connaissance du langage d'expression peut être aussi une autre cause d'erreur dans la spécification.
- l'erreur provient de la mise en correspondance des variables apparaissant dans la propriété avec les variables et expressions du programme.

  En général, la personne qui exprime les propriétés, le fait sans connaissance du programme. Les notations qu'elle emploie pour désigner les différentes entités mises en jeu par l'application, ne correspondent pas forcément à celles du programmeur. Au moment de la vérification, il faut donc faire un lien entre les entités du programme et celles de la propriété. Ceci peut être source d'erreur, surtout si, le vérificateur et le spécificateur ne sont pas la même personne. Dans mon cas, le problème est provenu d'un grand laps de temps (plusieurs mois) entre d'une part, l'expression du programme et des propriétés, et d'autre part, leur vérification.
- le programme ne vérifie pas la propriété dans le cas général, car il a, par exemple, été réalisé en émettant des hypothèses sur l'environnement. Par contre, il vérifierait la propriété s'il était vérifié sous des assertions décrivant l'évolution connue de l'environnement.
- le programme vérifie la propriété mais l'abstraction engendre des états parasites : on peut alors utiliser les assertions pour interdire ces états.

  Les deux derniers points montrent l'intérêt des assertions pour la vérification. Précisons, qu'avant d'utiliser une assertion il faut être certain de sa validité pour ne pas fausser la validation du programme. Notons aussi que les erreurs faisant l'objet des points 2 et 3, peuvent apparaître lors de l'expression des assertions.

Nous allons maintenant donner un exemple d'une erreur de chaque type, dans l'ordre de leur présentation. Ces exemples ont été tirés de l'application SIREX. Pour chacun d'eux, nous montrerons comment nous avons trouvé l'erreur à partir des informations fournies par LESAR :

### 6.3.1 Erreur de programmation

Nous désirons vérifier la propriété (35) :
*jamais* HN_valide *sans* (*au cycle précédent* (HN_actif))

HN_valide et HN_actif ont été définis en fonction de variables du programme par les équations :

HN_valide = false -> (pre (niveau_valide) = HN) (1)

HN_actif = false -> pre (niveau_hn_autorisé) (2)

**HN_valide** représente le fait que la chaîne de capteurs valide au moment des mesures réalisées pour ce cycle était HN.
**HN_actif** représente le fait que la chaîne de capteurs HN était active au moment des mesures réalisées pour ce cycle.
**niveau_valide** représente la chaîne qui sera valide au prochain cycle. C'est une sortie du programme.
**niveau_hn_autorisé** représente le fait que la chaîne HN sera active au prochain cycle. C'est une sortie du programme.

Les informations obtenues en sortie de LESAR sont exposées ci-dessous. Nous ne retraçons pas leur énoncé exact afin d'obtenir une présentation plus synthétique.

*passage de l'état 0 à l'état 1 par* :
non (flux > p1) et non (flux < p2) et (flux > debrc) et (flux < finrc)
*passage de l'état 1 à l'état 15 par* :
(sens_flux = croissant) et (flux > p1) et sélection_hn et non (flux < p2) et (flux > debrc) et (flux < finrc)
*dans l'état 15, sous la condition :*
(flux > p1) et non (flux < p2) et (flux > debrc)
*on a :*
pre (niveau_valide) = HN
pre (HN_actif) = false
*et donc, la propriété est fausse.*

Une première analyse des causes d'échec montre qu'HN-valide est vrai puisque pre (niveau_valide) = HN (équation 1). Il faut donc comprendre pourquoi, dans l'état 15, pre (HN_actif) = false. Or, on note que, dans l'état 1, HN_actif = false ne peut provenir que du fait que, dans l'état 0, niveau_hn_autorisé est faux (équation 2).

Nous avons alors constaté, dans le programme, que la variable niveau_hn_autorisé était initialisée à faux à l'aide de l'opérateur ->. L'explication de cette erreur est une confusion entre les deux entités HN_actif et niveau_hn_autorisé, l'une ayant la valeur précédente de l'autre. HN-actif représente l'activité courante de la chaîne haut-niveau : cette variable est vraie ssi, au cycle courant, il est possible de lire une mesure du flux sur HN. Par hypothèse tirée du cahier des charges, la chaîne haut-niveau est inactive au démarrage du réacteur; la valeur de HN-actif au premier cycle est donc faux. La variable niveau_hn_autorisé, par contre, représente la valeur, calculée par le programme, de ce que sera l'activité de la chaîne haut-niveau au cycle suivant. Cette variable est une sortie du programme alors que HN-actif pourrait être assimilée à une entrée. Sa valeur doit être le résultat d'un calcul, au premier cycle comme aux autres.

Remarquons que cette erreur n'aurait pas été découverte si une assertion supplémentaire avait été ajoutée, précisant que le flux doit être nul au premier cycle (d'après l'hypothèse énoncée au paragraphe 3.1.2.1 et donc inférieur à debrc, puisque dans ce cas-là, niveau_hn_autorisé n'aurait pu que prendre la valeur faux au premier cycle.

### 6.3.2 Erreur de spécification

En guise d'exemple du deuxième type d'erreur, nous nous intéressons à la propriété (39) que nous désirons vérifier :

*toujours cont* HN_valide *entre* (*au cycle précédent* (Commut_BH) *et* Commande_Op) *et* (*au cycle précédent* (HN_actif et Flux < p2))

Nous noterons dans la suite de ce paragraphe :

A = *au cycle précédent* (Commut_BH) *et* Commande_Op

B = *au cycle précédent* (HN_actif et Flux < p2)

HN_valide, HN_actif, Commut_BH et Commande_Op ont été définis en fonction de variables du programme :

- HN_valide et HN_actif de la même façon qu'au paragraphe 6.3.1
- Commut_BH existe tel quel
- Commande_Op correspond à la variable sélection_hn.

**Commut_BH** représente le fait que l'opérateur est autorisé à décider le changement de chaîne valide du BN vers le HN. C'est une sortie du programme.
**Commande_Op** est l'instruction de l'opérateur commandant le changement de niveau valide. C'est une entrée du programme.

Les informations obtenues en sortie de LESAR sont les suivantes :

*passage de l'état 0 à l'état 1 par* :
non (flux > p1) et (flux < p2) et non (flux > debrc) et (flux < finrc)
*passage de l'état 1 à l'état 2 par* :
non (flux > p1) et (flux < p2) et (flux > debrc) et (flux < finrc)
*passage de l'état 2 à l'état 9 par* :
(sens_flux = croissant) et (flux > p1) et non (flux < p2) et (flux > debrc) et (flux < finrc)
*dans l'état 9, sous la condition :*
(flux > p1) et sélection_hn et non (flux < p2) et (flux > debrc)
*on a :*
pre (Commut_BH)
sélection_hn
non pre (niveau_valide = HN)
non pre ( B)
non pre [ (B and not Depuis (A and non B))
or (not A and not B and pre (Depuis (A and not B))) ]
*et donc, la propriété est fausse.*

Nota : le nœud Depuis a été reconstruit ici à partir des informations fournies, pour les besoins de la présentation.

Dans l'état 9, HN_valide est faux, puisque (pre (niveau_valide) = HN) = false. Or, les conditions requises par la propriété pour que cette variable soit vraie, sont vérifiées dans cet état :
- la propriété "pre (Commut_BH) et Commande_Op" a été vraie dans le passé ou est vraie à l'instant courant,
  puisqu'elle est vraie à l'instant courant
- la propriété "pre ((flux < p2) et HN_actif)" n'a jamais été vraie depuis l'instant, compris, où la propriété "pre (Commut_BH) et Commande_Op" a été vraie,
  puisqu'elle est fausse à l'instant courant

Nous avons donc cherché à comprendre pourquoi la valeur de niveau_valide dans l'état 2 est égale à BN (ce qui correspond au fait que dans l'état 9, HN_valide soit faux). Dans le programme, le niveau valide passe de BN à HN sur la condition :
false -> pre (Commut_BH) and Commande_Op
et donc HN_valide qui est pre (niveau_valide) = HN, ne peut être vrai qu'un cycle après ce passage, donc lorsque l'expression suivante est vraie :

**false -> pre** ( pre (Commut_BH) and Commande_Op)

La condition du programme réalisant la commutation du niveau valide de BN vers HN étant correcte, c'est l'énoncé de la propriété qui est faux.

### 6.3.3 Erreur de mise en correspondance entre la spécification et le programme

La propriété que nous désirons vérifier est la proriété (20) :
*jamais* Commut_BH sans (BN_valide et (flux > p1) et (sens_flux = croissant))

Nous avons vu dans le paragraphe 6.3.2 ce que représente Commut_BH et comment cette variable est traduite dans le programme. BN_valide représente la même notion que HN_valide mais pour le niveau bas des capteurs. Il est traduit par :

BN_valide = true -> (pre (niveau_valide) = BN)

Les informations obtenues en sortie de LESAR sont les suivantes :

*passage de l'état 0 à l'état 1 par* :
non (flux > p1) et (flux < p2) et non (flux > debrc) et (flux < finrc)
*passage de l'état 1 à l'état 2 par* :
non (flux > p1) et (flux < p2) et (flux > debrc) et (flux < finrc)
*passage de l'état 2 à l'état 5 par* :
(sens_flux = croissant) et (flux > p1) et (flux < p2) et (flux > debrc) et (flux < finrc)
*dans l'état 5, sous la condition :*
(sens_flux = croissant) et (flux > p1) et sélection_hn et (flux < p2) et (flux > debrc)
et (flux < finrc)
*on a notamment :*
sens_flux = croissant
flux > p1
non (pre niveau_valide = HN)
*et donc, la propriété est fausse.*

Commut_BH est vrai, mais (BN_valide et (flux > p1) et (sens_flux = croissant)) ne l'est pas. Or, les expressions (sens_flux = croissant) et (flux > p1) sont vraies. Par conséquent BN_valide ne l'est pas.

Dans le programme, la commutation est permise lorsque l'expression :
true -> pre (niveau_valide = BN) and (sens_flux = croissant) and (flux > p1)
est vraie. La propriété ne peut donc être vraie puisqu'on teste : niveau_valide = BN, et non pas pre (niveau_valide = BN).

En fait, c'est l'interprétation qui a été faite de la variable BN_valide en terme des variables du programme, qui est fausse :
BN_valide est vraie ssi la chaîne valide sur laquelle on vient de capter le flux, est BN, et non si la chaîne qui captera le flux au prochain cycle, et dont on vient de déterminer la valeur, est BN.
Il faut donc définir BN_valide par :

true -> pre (niveau_valide = BN)

### 6.3.4 Erreur due à un manque d'assertion

Reprenons la propriété 35 (cf paragraphe 6.3.1) sur un programme corrigé mais en supprimant toute assertion. Les assertions utilisées dans la vérification présentée au paragraphe 6.3.1 positionnaient les seuils entiers les uns par rapport aux autres. Elles comprenaient par exemple les assertions suivantes :

**assert** IMPLIQUE ((flux > p1), (flux > debrc))

**assert true** -> IMPLIQUE ((flux > p1) **and not pre** (flux > p1), **pre** (flux > debrc))

qui signifient, d'une part, que p1 est supérieur à debrc, et d'autre part, que la différence entre les deux est plus grande que la valeur maximale dont le flux peut augmenter entre deux cycles.

Les informations obtenues en sortie de LESAR sont les suivantes :

*passage de l'état 0 à l'état 4 par* :
(sens_flux = croissant) et (flux > p1) et non (flux > debrc)
*passage de l'état 4 à l'état 13 par* :
non (sens_flux = croissant) et sélection_hn et (flux > debrc)
*dans l'état 13, sous la condition :*
true
*on a :*
pre niveau_valide = HN
non pre HN_actif

*et donc, la propriété est fausse.*

Nous voyons que dans la réalité, au premier cycle pas plus qu'aux autres, le flux ne peut être à la fois plus grand que p1 et plus petit que debrc, puisque, comme nous l'avons dit ci-dessus, p1 est supérieur à debrc. La propriété n'est donc pas fausse dans ce cas, au vu de l'environnement. Pour que la vérification puisse avoir lieu dans des conditions correctes, il faut donc spécifier les propriétés de l'environnement par des assertions.

## 6.4 EXPÉRIENCE DE VALIDATION DE SIREX

Nous allons présenter dans ce paragraphe le résultat de la vérification de l'étude de cas extraite de SIREX ( cf chapitre 2) . La spécification complète se trouve en annexe 1. La vérification va être présentée sous forme de tableaux de synthèse. Le programme a été réalisé en SAGA puis traduit manuellement en LUSTRE. Le programme LUSTRE obtenu, et sur lequel a été réalisé la vérification, possède 22 entrées.

La vérification est d'abord réalisée en générant l'automate LUSTRE tout entier à l'aide du compilateur LUSTRE. Manuellement, il est vérifié sur l'automate minimisé que l'expression correspondant à la propriété est vraie dans tout état. Cette vérification a été réalisée sur une première version du programme et a permis de mettre à jour une erreur de ce programme. Le programme a alors été modifié et toutes les propriétés ont été vérifiées à nouveau. Dans les tableaux représentant cette première expérience (figures 6.4.a et 6.4.b), le numéro de certaines propriétés est suivi d'une lettre. Cela représente le fait que la propriété a été vérifiée plusieurs fois, soit que les assertions étaient insuffisantes les fois précédentes, soit que l'énoncé de la propriété était faux et a été modifié. Dans les tableaux, le nombre d'assertions est celui des assertions élémentaires, c'est à dire ne contenant pas d'opérateurs *and* au plus haut niveau.

Le programme ne comporte ici que la partie régulation des capteurs. Les entrées restent au nombre de 22. Rappelons que LUSTRE est un langage déclaratif. C'est pourquoi, en général, le nombre de lignes du programme est faible par rapport à un programme impératif.

| numéro de propriété | nombre de lignes | nombre d'assertions | nombre de sorties | nombre d'états avant minimisation | nombre d'états après minimisation | remarque |
|---|---|---|---|---|---|---|
| 31 | 422 | 2 | 1 | 15 | 8 | (A) (V) |
| 32 | 421 | 2 | 1 | 15 | 8 | (A) (V) |
| 35 | 418 | 2 | 1 | 29 | 10 | (A) (F) |
| 35b | 427 | 3 | 1 | 57 | 13 | (A) (F) |
| 35c | 441 | 7 | 1 | 817 | 2 | (A) (V) |
| 36 | 440 | 7 | 1 | 825 | 3 | (A) (V) |
| 38 | 491 | 7 | 1 | 1305 | 38 | (A) (F) |
| 39 | 490 | 7 | 1 | 2633 | 46 | (A) (F) |

Figure 6.4.a : vérification de la version 1 en simulant les assertions

(A) : les assertions sont seulement simulées, c'est-à-dire que la formule vérifiée est :

simul_assertions => exp_propriété

où simul_assertions est une expression booléenne LUSTRE vraie dans un état ssi les assertions ont toujours été vraie dans le passé de cet état, et exp_propriété est l'expression correspondant à la propriété.

(V) : la propriété est vraie sous les assertions spécifiées

(F) : la propriété n'est pas vraie sous les assertions spécifiées

| numéro de propriété | nombre de lignes | nombre d'assertions | nombre de sorties | nombre d'états avant minimisation | nombre d'états après minimisation | remarque |
|---|---|---|---|---|---|---|
| 31 | 418 | 2 | 1 | 4 | 2 | (B) (V) |
| 32 | 416 | 2 | 1 | 4 | 2 | (B) (V) |
| 35 | 414 | 2 | 1 | 7 | 4 | (B)(F) |
| 35b | 415 | 3 | 1 | 7 | 5 | (B) (F) |
| 35c | 431 | 7 | 1 | | | (B') |
| 36 | 437 | 7 | 1 | | | (B') |
| 38 | 481 | 7 | 1 | | | (B') |
| 39 | 486 | 7 | 1 | | | (B') |

Figure 6.4.b : vérification de la version 1 avec utilisation des assertions

Remarquons, d'après les tableaux précédents, que

- l'automate généré n'est en général pas minimal,
- pour les premières propriétés (32 à 35), la présence d'assertions diminue le code produit,
- pour les propriétés à partir de 35c, le code semble croître, ou en tout cas, le calcul des assertions occupe davantage de place mémoire. Ceci est normal dans la mesure où les 4 assertions ajoutées lors de la vérification de 35c font intervenir 4 variables d'états supplémentaires (expressions en *pre* n'apparaissant pas dans le corps du programme).

Les propriétés 38 et 39 étant fausses suite à une erreur de programmation, une nouvelle version a été réalisée, puis une troisième, que nous présentons ici (figure 6.4.c). Les versions 2 et 3 du programme ne comportant que la régulation des capteurs, ont été vérifiées à l'aide de LESAR. Le nombre d'états lorsque la propriété n'est pas vraie sous les assertions données, correspond au nombre des états accessibles à partir des états déjà évalués (compris dans ce nombre). Le temps d'exécution utilisateur est celui relevé lors de l'utilisation de LESAR, c'est-à-dire celui de la vérification réalisée sur le programme LUSTRE étendu.

---

(B) : utilisation des assertions

(V) : la propriété est vraie sous les assertions spécifiées

(F) : la propriété n'est pas vraie sous les assertions spécifiées

(B') : automate non générable par la version 2 du compilateur

| numéro de propriété | nombre de lignes | nombre d'assertions | nombre de sorties | nombre d'états | temps utilisateur | remarque |
|---|---|---|---|---|---|---|
| 18 | 482 | 7 | 1 | 25 | 1,1 | (V) |
| 30 | 480 | 7 | 3 | 28 | 1,4 | (V) |
| 31 | 485 | 7 | 1 | 27 | 1,4 | (V) |
| 32 | 485 | 7 | 1 | 26 | 1,3 | (V) |
| 35 | 485 | 7 | 1 | 33 | 1,4 | (V) |
| 36 | 483 | 7 | 1 | 32 | 1,4 | (V) |
| 36 | 483 | 7 | 3 | 32 | 1,4 | (V) (C) |
| 36 | 483 | 7 | 5 | 32 | 1,4 | (V) (D) |
| 38 | 533 | 7 | 1 | 37 | 1,8 | (V) |
| 38b | 528 | 7 | 1 | 54 | 2,6 | (V) |
| 39 | 528 | 7 | 1 | 34 | 1,9 | (F) état 14 |
| 39b | 529 | 7 | 1 | 99 | 4,6 | (V) |
| 40 | 528 | 7 | 1 | 91 | 4,2 | (V) |
| 49 | 486 | 7 | 1 | 25 | 1,3 | (V) |
| 58 | 489 | 7 | 1 | 28 | 1,6 | (V) |
| 59 | 489 | 7 | 1 | 27 | 1,2 | (V) |

Figure 6.4.c : version 3 de la régulation des capteurs

Le tableau suivant (figure 6.4.d) montre la vérification de la deuxième partie du programme, qui concerne le déclenchement des alarmes et arrêts d'urgence. Il correspond à la deuxième version de ce programme, une première erreur ayant été corrigée. Toutes les propriétés sont vérifiées.

(V) : la propriété est vraie sous les assertions spécifiées

(C) : le résultat est identique avec une ou trois sorties car, dans ce cas, les deux sorties supplémentaires sont nécessaires au calcul de la propriété et n'introduisent donc pas de variables d'état supplémentaires

(D) : le nombre d'états de l'automate a augmenté car, les deux nouvelles sorties ne sont pas nécessaires au calcul de la propriété et introduisent des variables d'états supplémentaires

(F) : la propriété n'est pas vraie sous les assertions spécifiées

| numéro de propriété | nombre de lignes | nombre d'assertions | nombre de sorties | nombre d'états | temps utilisateur | remarque |
|---|---|---|---|---|---|---|
| 02 | 425 | 24 | 1 | 17 | 9,6 | (V) |
| 03 | 425 | 24 | 1 | 17 | 9,4 | (V) |
| 04 | 425 | 24 | 1 | 17 | 9,6 | (V) |
| 05 | 425 | 24 | 1 | 29 | 16,3 | (V) |
| 06 | 425 | 24 | 1 | 29 | 15,1 | (V) |
| 07 | 425 | 24 | 1 | 29 | 15,2 | (V) |
| 08 | 423 | 24 | 1 | 15 | 7,2 | (V) |
| 09 | 423 | 24 | 1 | 15 | 6,9 | (V) |
| 10 | 425 | 24 | 1 | 17 | 9,4 | (V) |
| 11 | 425 | 24 | 1 | 17 | 9,7 | (V) |
| 12 | 425 | 24 | 1 | 17 | 9,8 | (V) |
| 13 | 425 | 24 | 1 | 29 | 16,5 | (V) |
| 14 | 425 | 24 | 1 | 29 | 15,4 | (V) |
| 15 | 425 | 24 | 1 | 29 | 15,1 | (V) |
| 16 | 423 | 24 | 1 | 15 | 7,5 | (V) |
| 17 | 423 | 24 | 1 | 15 | 7,1 | (V) |
| 22 | 425 | 24 | 1 | 21 | 12,1 | (V) |
| 23 | 425 | 24 | 1 | 17 | 9,3 | (V) |
| 24 | 425 | 24 | 1 | 17 | 9,5 | (V) |
| 40 | 399 | 4 | 1 | 21 | 0,7 | (V) |
| 43 | 370 | 4 | 1 | 21 | 0,8 | (V) |
| 44 | 368 | 4 | 1 | 21 | 0,7 | (V) |
| 45 | 368 | 4 | 1 | 21 | 0,7 | (V) |
| 46 | 368 | 4 | 1 | 21 | 0,7 | (V) |
| 47 | 397 | 4 | 1 | 7 | 0,5 | (V) |
| 48 | 388 | 4 | 1 | 7 | 0,5 | (V) |
| 51 | 430 | 24 | 1 | 17 | 9,5 | (V) |
| 55 | 430 | 24 | 1 | 17 | 9,5 | (V) |

Figure 6.4.d : vérification des alarmes / version 2 du programme

Certaines propriétés ont ensuite été reprises et leur vérification réalisée sur le programme tout entier. Nous montrons ainsi le gain obtenu par l'approche de modularité sur la taille de l'automate et le temps d'exécution (figure 6.4.e).

| numéro de propriété | nombre de lignes | nombre d'assertions | nombre de sorties | nombre d'états | temps utilisateur | remarque |
|---|---|---|---|---|---|---|
| 03 | 771 | 52 | 1 | 1441 | 2050,2 | (V) |
| 17 | 802 | 52 | 1 | 487 | 641,6 | (V) |
| 41 | 798 | 52 | 1 | 1561 | 2305,9 | (V) |
| 47 | 802 | 52 | 1 | 649 | 886,2 | (V) |

Figure 6.4.e : vérification des alarmes / version 2 du programme

## 6.5 CONCLUSION

Ce chapitre a retracé les différents problèmes pratiques que pose la validation de programmes LUSTRE, et les solutions apportées à ces problèmes.

- Le premier problème concerne la taille de l'automate généré. Deux solutions ont été apportées : la minimisation de l'automate à l'aide de l'outil OCMIN (qui est utilisable pour les automates ESTEREL), et la vérification sur l'automate en cours de génération. D'autres solutions sont envisagées dans le cadre de la compilation.
- Au deuxieme problème, relatif aux assertions non causales aucune solution n'a pu être fournie : il est en fait intrinsèque à la vérification en cours de génération.

Ce chapitre a également présenté l'outil de vérification LESAR, ainsi qu'une expérimentation en grandeur réelle menée avec cet outil. Il nous a permis de montrer l'intérêt d'une vérification "à la volée" (au cours de la génération de l'automate) par rapport à une vérification sur l'automate entièrement généré : **gain en place mémoire** et **gain en temps** (puisque la structure de l'automate n'est pas mémorisée).

Cet avantage est mis en balance par le fait qu'un résultat négatif de la vérification n'est pas significatif en présence d'assertions "non causales".

---

(V) : la propriété est vraie sous les assertions spécifiées

En effet, l'arrêt de LESAR dès que la propriété n'est pas vraie dans un état, et la non-mémorisation de la structure de l'automate rend impossible le retour sur les états prédécesseurs; or, ce retour est nécessaire dans le cas d'assertions "non causales". C'est donc à l'utilisateur de LESAR de s'assurer que ses assertions sont bien causales.

La réalisation d'une expérimentation de LESAR et du principe de vérification adopté, en grandeur réelle (partie d'une application développée à Merlin Gerin / SES) a permis de mettre en avant l'utilité de la vérification, même restreinte, aux seules propriétés de sûreté. Car, rappelons le, ces propriétés sont majoritaires dans le domaine des systèmes réactifs temps réel.

D'autre part, nous avons pu tirer de cette expérience des remarques intéressantes concernant le type des erreurs apparaissant lors d'une vérification.

# Conclusion

Le travail de cette thèse a concerné la vérification de propriétés sur des programmes conçus chez Merlin Gerin / SES à l'aide de l'atelier SAGA. L'étude des besoins s'est déroulée en deux phases.

La formalisation de la sémantique du langage de conception de l'atelier SAGA formait la première phase, alors que la deuxième concernait l'étude de logiciels développés à SES, afin de qualifier le type de propriétés à vérifier.

Dans un premier temps, notre but était d'utiliser des logiciels existants, notamment le logiciel XESAR. Dans un deuxième temps, nous nous sommes intéressés à la vérification de propriétés décrites en LUSTRE à l'aide de l'automate de contrôle généré par le compilateur LUSTRE.

Le langage SAGA est un langage flots de données synchrone, similaire en cela à LUSTRE. Tout programme SAGA en fin de phase de conception peut être traduit en un programme LUSTRE (la réciproque n'est pas vraie). Nous nous sommes alors intéressés à la vérification des programmes LUSTRE qui englobe celle des programmes SAGA.

Par ailleurs, deux points sont ressortis des études de cas :

- la plupart des propriétés sont des propriétés de sûreté ("safety"),
- la méthode de spécification employée a conduit à l'utilisation d'une construction du passé : *au cycle précédent*, pour l'expression des propriétés.

Notre but étant l'utilisation de XESAR pour la vérification des propriétés, nous avons voulu traduire les constructions utilisées pour exprimer les propriétés de nos études de cas, d'abord en logique temporelle : CL, ensuite en μ-calcul propositionnel : μ-CL (adaptation de STL, langage reconnu par XESAR). Les constructions utilisées dans les études de cas ont pour cela été formalisées, et trois nouvelles constructions ont été ajoutées afin de ne pas trop restreindre nos prétentions au niveau de l'expressivité. Nous avons ainsi défini un langage LEP. Les modalités temporelles des langages CL et μ-CL concernent le futur.

Un travail de DEA [Rat,88] ayant été réalisé en parallèle sur la validation de programme LUSTRE à l'aide de XESAR, nous avons pu en tirer des enseignements. Notamment, nous en avons déduit comment résoudre le problème que pose l'opérateur *au cycle précédent* pour la vérification sur un graphe d'état.

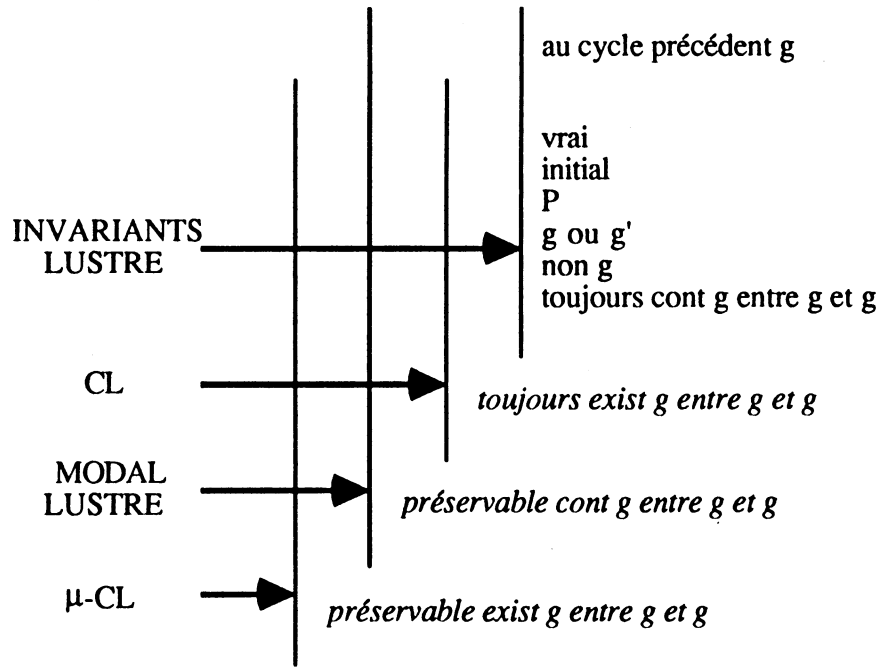
Nous avons alors défini un nouveau langage : MODAL LUSTRE; alliant les modalités des logiques temporelles du futur, aux modalités du passé de LUSTRE. Nous avons esquissé le processus de validation en nous inspirant de XESAR pour la vérification des opérateurs temporels du futur. Cette approche prend comme modèle du programme un graphe d'états défini directement

à partir du programme LUSTRE et qui tient compte de la notion de synchronisme inhérente à LUSTRE.

Nous avons ensuite restreint le langage de spécification à des formules LUSTRE sur lesquelles est appliqué l'opérateur du futur arborescent *toujours*. Ce nouveau langage est appelé INVARIANTS LUSTRE. Une méthode de vérification de propriétés a été définie qui utilise la compilation de programmes LUSTRE sous forme d'automates de contrôle. La formule LUSTRE contenue dans la propriété est incorporée au programme sous la forme d'une équation afin que l'automate généré à la compilation tienne compte des opérateurs du passé apparaissant dans la propriété. L'évaluation de la propriété *toujours(e)* devient alors triviale puisque la valeur de la formule LUSTRE *e* apparait dans chaque état de l'automate.

Nous avons réalisé un prototype mettant en œuvre cette méthode : LESAR. Afin d'éviter le problème de "l'explosion" du nombre d'états, commun à toutes les approches de vérification sur des graphes d'états, nous avons réalisé l'évaluation au cours de la génération de l'automate et non après. Cela nous a été possible car nous ne vérifions que des propriétés de sûreté.

Nous résumons sur un schéma la capacité de chacun des langages, utilisés dans cette thèse pour la spécification, à exprimer les constructions du langage LEP. Les constructions en italique sont celles n'apparaissant pas dans les études de cas.




Le travail de cette thèse peut être poursuivi sous différentes formes :

Nous pensons qu'il serait intéressant d'approfondir le diagnostic fourni par LESAR, par exemple, en caractérisant l'état invalidant la propriété, par une formule retraçant le chemin depuis la racine (notamment en terme des entrées successives). Alors, si F est la formule caractéristique de l'état d'échec, ajouter une assertion spécifiant $\neg F$, permettrait à LESAR de dépasser cet état, et de trouver un autre état d'échec, s'il en existe un. Ainsi, par une vérification "incrémentale", on serait capable de déterminer toutes les causes d'erreur d'une propriété.

Une autre voie, cependant liée à la précédente, concerne l'étude des changements apportés par la modification des assertions sur l'automate engendré. Le but étant de ne regénérer que les parties de l'automate touchées par la modification. On peut envisager de même des modifications de la propriété à vérifier.

Par ailleurs, nous envisageons d'étudier les liens entre la spécification des propriétés et la décomposition SAGA, afin de favoriser une vérification modulaire. De plus, nous désirons étudier l'aide que peuvent apporter les assertions dans la découverte des erreurs au cours de la conception du programme SAGA. Par exemple, la définition d'un nouveau type de boîtes non affinables, représentées par des assertions, pourrait permettre d'obtenir une conception SAGA incomplète mais cependant transformable en un automate de contrôle, acceptant ainsi des premières vérifications.

D'autre part, il resterait à implanter la méthode de vérification définie pour MODAL LUSTRE.

# ANNEXES

# 1 Etude de cas extraite de SIREX : énoncé des propriétés

## 1.1 COHÉRENCE DES SORTIES

Dans ce paragraphe sont énoncés des invariants liant les sorties entre elles. Certains de ces invariants sont commentés de façon à faciliter la compréhension de l'exposé. C'est pourquoi, parmi ces commentaires, certains font référence à des propriétés développées ultérieurement dans la spécification.

### 1.1.1 Cohérence entre les sorties "niveau valide"

L'une des chaînes de capteurs: bas niveau ou haut niveau, doit toujours être valide, mais jamais les deux à la fois:

*toujours* (HN-valide *ou* BN-valide) *et jamais* (HN-valide *et* BN-valide) (1)

En effet, la surveillance du flux doit être continue. Donc, à tout instant, une des chaînes de capteurs doit fournir la valeur représentative du flux. Cette valeur doit être unique, ce qui induit que les deux chaînes ne peuvent être valides à la fois.

### 1.1.2 Cohérence entre les sorties "passage de seuil" et les autres

On doit observer le passage du seuil MinBN, chaque fois que la valeur de Flux-BN est inférieure à la valeur MinBN, le bas niveau étant valide:

PS-MinBN *chaque fois que* ((Flux-BN < MinBN) *et* BN-valide) (2)

Il faut préciser que lorsque les capteurs du bas niveau ne sont pas en état de marche, la sortie Flux-BN vaut zéro (propriété 32). Dans ce cas là, malgré la valeur de Flux-BN, le seuil MinBN ne doit pas être considéré comme franchi. La propriété 4 ne contredit pas cela, car lorsque les capteurs du bas niveau ne sont pas actifs, cette chaîne ne peut être valide (propriété 30).

On doit observer le passage du seuil Mx1BN, chaque fois que la valeur de Flux-BN est supérieure à la valeur Mx1BN, le bas niveau étant valide:

PS-Mx1BN *chaque fois que* ((Flux-BN > Mx1BN) *et* BN-valide) (3)

On doit observer le passage du seuil Mx2BN, chaque fois que la valeur de Flux-BN est supérieure à la valeur Mx2BN, le bas niveau étant valide:

PS-Mx2BN *chaque fois que* ((Flux-BN > Mx2BN) *et* BN-valide) (4)

On doit observer le passage du seuil MinHN, chaque fois que la valeur de Flux-HN est inférieure à la valeur MinHN, le haut niveau étant valide:

PS-MinHN *chaque fois que* ((Flux-HN < MinHN) *et* HN-valide) (5)

On peut faire ici aussi une remarque concernant la valeur par défaut donnée à la sortie Flux-HN lorsque les capteurs du haut niveau sont inactifs: celle-ci vaut zéro (propriété 33), mais le haut niveau n'est valide qu'au delà de la valeur FINRC (propriété 3).

On doit observer le passage du seuil Mx1HN, chaque fois que la valeur de Flux-HN est supérieure à la valeur Mx1HN, le haut niveau étant valide:

PS-Mx1HN *chaque fois que* ((Flux-HN > Mx1HN) *et* HN-valide) (6)

On doit observer le passage du seuil Mx2HN, chaque fois que la valeur de Flux-HN est supérieure à la valeur Mx2HN, le haut niveau étant valide:

PS-Mx2HN *chaque fois que* ((Flux-HN > Mx2HN) *et* HN-valide) (7)

On doit observer le passage du seuil Mx1PE, chaque fois que la Période est supérieure à la valeur Mx1PE:

PS-Mx1PE *chaque fois que* (Période > Mx1PE) (8)

La valeur par défaut donnée à la période quand celle-ci ne peut être calculée, est aussi zéro. Par conséquent, elle n'intervient ni dans cette propriété ni dans la suivante.

On doit observer le passage du seuil Mx2PE, chaque fois que la Période est supérieure à la valeur Mx2PE:

PS-Mx2PE *chaque fois que* (Période > Mx2PE) (9)

On ne peut jamais observer le passage du seuil MinBN, sans que le bas niveau soit valide et la valeur de Flux-BN inférieure à la valeur MinBN + hystérésis:

*jamais* PS-MinBN *sans* (BN-valide *et* Flux-BN < MinBN+hystérésis) (10)

Rappellons que, lorsque les capteurs du bas niveau ne sont pas en état de marche, bien que la sortie Flux-BN vaille zéro (propriété 32), le seuil MinBN ne doit pas être considéré comme franchi. La propriété 12 ne contredit pas cela, car lorsque les capteurs du bas niveau ne sont pas actifs, cette chaîne ne peut être valide (propriété 30).

On ne peut jamais observer le passage du seuil Mx1BN, sans que le bas niveau soit valide et la valeur de Flux-BN supérieure à la valeur Mx1BN - hystérésis:

*jamais* PS-Mx1BN *sans* (BN-valide *et* Flux-BN > Mx1BN-hystérésis) (11)

On ne peut jamais observer le passage du seuil Mx2BN, sans que le bas niveau soit valide et la valeur de Flux-BN supérieure à la valeur Mx2BN - hystérésis:

*jamais* PS-Mx2BN *sans* (BN-valide *et* Flux-BN > Mx2BN-hystérésis) (12)

On ne peut jamais observer le passage du seuil MinHN, sans que le haut niveau soit valide et la valeur de Flux-HN inférieure à la valeur MinHN + hystérésis

*jamais* PS-MinHN *sans* (HN-valide *et* Flux-HN < MinHN+hystérésis) (13)

On peut faire ici aussi une remarque concernant la valeur par défaut donnée à la sortie Flux-HN lorsque les capteurs du haut niveau sont inactifs: celle-ci vaut zéro (propriété 33), mais le haut niveau n'est valide qu'au delà de la valeur FINRC (propriété 3).

On ne peut jamais observer le passage du seuil Mx1HN, sans que le haut niveau soit valide et la valeur de Flux-HN supérieure à la valeur Mx1HN - hystérésis:

*jamais* PS-Mx1HN *sans* (HN-valide *et* Flux-HN > Mx1HN-hystérésis) (14)

On ne peut jamais observer le passage du seuil Mx2HN, sans que le haut niveau soit valide et la valeur de Flux-HN supérieure à la valeur Mx2HN - hystérésis:

*jamais* PS-Mx2HN *sans* (HN-valide *et* Flux-HN > Mx2HN-hystérésis) (15)

On ne peut jamais observer le passage du seuil Mx1PE, sans que la valeur de Période soit supérieure à la valeur Mx1PE - hystérésis:

*jamais* PS-Mx1PE *sans* (Période > Mx1PE-hystérésis) (16)

La valeur par défaut donnée à la période quand celle-ci ne peut être calculée, est aussi zéro. Par conséquent, elle n'intervient ni dans cette propriété ni dans la suivante.

On ne peut jamais observer le passage du seuil Mx2PE, sans que la valeur de Période soit supérieure à la valeur Mx2PE - hystérésis:

*jamais* PS-Mx2PE *sans* (Période > Mx2PE-hystérésis) (17)

### 1.1.3 Cohérence de la sortie "commutation possible" avec les autres

A chaque fois que la sortie CommutBH offre à l'opérateur la possibilité de changer le niveau valide de bas niveau en haut niveau, la chaîne de capteurs bas niveau doit effectivement être valide et la valeur de Flux-BN doit être supérieure à P1:

*jamais* CommutBH *sans* (BN-valide *et* Flux-BN > P1) (18)

### 1.1.4 Cohérence entre les sorties "passage de seuil"

*jamais* PS_Mx2BN *sans* PS_Mx1BN (19)
*jamais* PS_Mx2HN *sans* PS_Mx1HN (20)
*jamais* PS_Mx2PE *sans* PS_Mx2PE (21)

Ayant considéré que les sorties exerçant un contrôle sur un matériel donné sont observables au même titre que les autres, nous pouvons définir des propriétés de cohérence entre ces sorties et les autres:

### 1.1.5 Cohérence entre les sorties "alarme" et "arrêt d'urgence" et les sorties "passage de seuil"

A aucun moment, une alarme ou un arrêt d'urgence ne peut être déclenché si le seuil correspondant n'est pas franchi:

*jamais* Al-MinBN *sans* PS-MinBN (22)
*jamais* Al-Mx1BN *sans* PS-Mx1BN (23)
*jamais* Al-Mx2BN *sans* PS-Mx2BN (24)
*jamais* Al-MinHN *sans* PS-MinHN (25)
*jamais* Al-Mx1HN *sans* PS-Mx1HN (26)
*jamais* Al-Mx2HN *sans* PS-Mx2HN (27)
*jamais* Al-Mx1PE *sans* PS-Mx1PE (28)

*jamais* Al-Mx2PE *sans* PS-Mx2PE (29)

### 1.1.6 Cohérence entre la sortie "haute tension" et les autres

A aucun moment, la haute tension n'est arrêtée, sans que le haut niveau de capteurs soit valide et la valeur de Flux-HN supérieure au seuil FINRC:

*jamais* (*non* MarcheHT) *sans* (HN-valide *et* Flux-HN > FINRC) (30)

Remarquons que le passage du seuil FINRC ne correspond à aucun affichage.

## 1.2 CONDITIONS À L'OBSERVATION

Nous allons répondre maintenant à la question "sous quelles conditions une sortie prend-elle une valeur donnée ?". Pour cela, nous serons amenés à parler des entrées, voire du comportement interne du système.

Remarquons que les trois premiers paragraphes énoncent des propriétés d'un niveau d'abstraction plus élevé que les suivants. Ces derniers approchent davantage la description d'un fonctionnement.

### 1.2.1 Première observation des sorties "niveau valide"

Un niveau de capteurs ne peut être valide s'il ne fournit pas de mesure. La connaissance de l'état de chaque chaîne de capteurs (active ou arrêtée) est gérée par le système, elle correspond donc à une variable interne: BN-actif, respectivement HN-actif, signifie que la chaîne de capteurs bas niveau, respectivement haut niveau, est active dans l'état courant.

*jamais* BN-valide *sans* BN-actif (31)

*jamais* HN-valide *sans* HN-actif (32)

### 1.2.2 Observation des sorties "flux"

Lorsqu'une chaîne est active, elle fournit une valeur du flux qui doit être affichée. Dans le cas où cette chaîne est inactive, une valeur doit pourtant être affichée. C'est pourquoi une valeur par défaut a été définie: elle est égale à zéro pour chaque chaîne de capteurs.

*jamais* (*non* BN-actif) *sans* (Flux-BN=0) (33)

*jamais* (*non* HN-actif) *sans* (Flux-HN=0)

Dans le cas de la chaîne haut niveau, cette valeur doit être affichée uniquement lorsque le niveau n'est pas actif car elle ne correspond à aucune mesure réelle réalisée sur cette chaîne.

*jamais* (Flux-HN=0) *sans* (*non* HN-actif)

Les deux dernières propriétés peuvent être résumées par la propriété suivante:

*jamais* (HN-actif *équivalent à* Flux-HN=0) (34)

Lorsque, dans la suite, nous voudrons parler d'une valeur de Flux-BN ou de Flux-HN qui a été mesurée, et non affichée par défaut, nous parlerons de valeur "significative". Ce qui correspondra dans l'énoncé des propriétés où il y a ambiguïté, à l'ajout de la précision: BN, ou HN, est actif.

### 1.2.3 Observation de la sortie "Période"

La période, à un instant donné, est fonction des valeurs courante et précédente du flux mesuré sur le niveau qui est valide à cet instant. Pour que la valeur de la période soit significative, il faut que les deux valeurs du flux le soient aussi, et par conséquent que la chaîne valide à l'instant du calcul soit active et qu'elle l'ait été à l'instant précédent. La première partie de cette propriété ayant été exprimée précédemment, nous n'énonçons ici que la deuxième partie.

*jamais* ( *non initial* et BN-valide *sans* (*au cycle précédent* BN-actif) (35)

*jamais* HN-valide *sans* (*au cycle précédent* HN-actif) (36)

De plus, la période prendra obligatoirement sa valeur par défaut au premier cycle (cycle où la valeur précédente du flux n'est pas définie).

*initialement* (Période = 0) (37)

### 1.2.4 Deuxième observation des sorties "niveau valide"

Ces nouvelles conditions sur l'observation des sorties niveau valide étant d'un niveau d'abstraction inférieur aux précédentes, nous avons préféré en faire deux paragraphes séparés.

Le bas niveau est valide à l'instant initial. Il doit le rester jusqu'au moment où l'opérateur accepte le changement qui lui était proposé par la sortie CommutBH. Puis, à chaque fois que la valeur du flux captée sur le haut niveau devient inférieure à P2, le bas niveau doit redevenir valide, et ce, jusqu'à ce que, de nouveau, l'opérateur décide de valider le haut niveau (après que la sortie CommutBH lui ait indiqué que cela était possible).

*toujours continuellement* BN-valide *entre initial*
*et* (*au cycle précédent* (CommutBH) *et* CommandeOp)) (38)
*toujours continuellement* BN-valide
*entre* (*au cycle précédent* (Flux-HN < P2 *et* HN-actif))
*et* (*au cycle précédent* (CommutBH) *et* CommandeOp)) (39)

Nous devrions préciser à l'aide d'une propriété, qu'après la validation du changement par l'opérateur, le bas niveau n'est plus valide. Mais cela apparaît dans les conditions d'observation du haut niveau lorsqu'on sait qu'on ne peut avoir les deux niveaux valides à la fois (propriétés 1 et 39).

A chaque fois que l'opérateur commande le changement de niveau valide du bas au haut niveau (après que cela lui ait été proposé par la sortie CommutBH), le haut niveau devient valide. Il doit le rester jusqu'à ce que le flux capté sur ce niveau devienne inférieur à la valeur P2, puis au cycle d'après, devenir invalide.

*toujours continuellement* HN-valide
*entre* (*au cycle précédent* (CommutBH) *et* CommandeOp)
*et* (*au cycle précédent* (HN-actif *et* Flux-HN < P2))) (40)

Nous devrions préciser à l'aide d'une propriété, qu'après la validation du changement par l'opérateur, le bas niveau n'est plus valide. Mais cela apparaît dans les conditions d'observation du haut niveau lorsqu'on sait qu'on ne peut avoir les deux niveaux valides à la fois (propriétés 1 et 38).

### 1.2.5 Observation des sorties "passage de seuil"

Lorsque la valeur de Flux-BN est inférieure au seuil MinBN alors que le bas niveau est valide, ce seuil doit être considéré comme franchi. Il doit le rester tant que la valeur de Flux-BN est inférieure ou égale à MinBN+hystérésis. Pendant tout ce temps là, le bas niveau reste valide.

*toujours*
(PS-MinBN *équivalent à*
(BN-valide *et*
(*après* (Flux-BN < MinBN *et* BN-actif *et* BN-valide)
*et_avant* (Flux-BN > MinBN+hystérésis)))) (41)

Remarques: la propriété indiquant que PS-MinBN devient faux à l'instant décrit par le "et_avant", apparaît dans les propriétés 12 et 32.

La précision "BN-actif" dans le corps du "après", afin de spécifier que la valeur de Flux-BN n'est pas une valeur par défaut, est inutile, car la propriété 30 précise que si BN est valide alors BN est actif.

Pour les seuils Mx1BN et Mx2BN, le franchissement est réalisé par valeur supérieure:

*toujours*
(PS-Mx1BN *équivalent à*
(BN-valide *et*
(*après* (Flux-BN > Mx1BN *et* BN-valide)
*et_avant* (Flux-BN < Mx1BN-hystérésis *et* BN-actif)))) (42)

Remarque: la propriété indiquant que PS-Mx1BN devient faux à l'instant décrit par le "et_avant", apparaît dans les propriétés 13 et 32.

*toujours*
(PS-Mx2BN *équivalent à*
(BN-valide *et*
(*après* (Flux-BN > Mx2BN *et* BN-valide)
*et_avant* (Flux-BN < Mx2BN-hystérésis *et* BN-actif)))) (43)

Remarque: la propriété indiquant que PS-Mx2BN devient faux à l'instant décrit par le "et_avant", apparaît dans les propriétés 14 et 32.

Les mêmes propriétés sont désirées pour le haut niveau:

*toujours*
(PS-MinHN *équivalent à*
(HN-valide *et*
(*après* (Flux-HN < MinHN *et* HN-actif *et* HN-valide)
*et_avant* (Flux-HN > MinHN+hystérésis)))) (44)

Remarques: la propriété indiquant que PS-MinHN devient faux à l'instant décrit par le "et_avant", apparaît dans les propriétés 15 et 33.

La précision "HN-actif" dans le corps du "après", afin de spécifier que la valeur de Flux-HN n'est pas une valeur par défaut, est inutile, car la propriété 31 précise que si HN est valide alors HN est actif.

*toujours*
(PS-Mx1HN *équivalent à*
(HN-valide *et*
(*après* (Flux-HN > Mx1HN *et* HN-valide)
*et_avant* (Flux-HN < Mx1HN-hystérésis *et* HN-actif)))) (45)

Remarque: la propriété indiquant que PS-Mx1HN devient faux à l'instant décrit par le "et_avant", apparaît dans les propriétés 16 et 33.

*toujours*
(PS-Mx2HN *équivalent à*
(HN-valide *et*
(*après* (Flux-HN > Mx2HN *et* HN-valide)
*et_avant* (Flux-HN < Mx2HN-hystérésis *et* HN-actif)))) (46)

Remarque: la propriété indiquant que PS-Mx2HN devient faux à l'instant décrit par le "et_avant", apparaît dans les propriétés 17 et 33.

Il en est encore de même avec les seuils sur la période (mis à part le conditionnement par le niveau valide):

*toujours continuellement* PS-Mx1PE
*entre* (Période > Mx1PE)
*et* (Période < Mx1PE-hystérésis) (47)

Remarque: la propriété indiquant que PS-Mx1PE devient faux à l'instant où Période<Mx1PE_hyst, apparaît dans la propriété 18.

*toujours continuellement* PS-Mx2PE
*entre* (Période> Mx2PE)
*et* (Période < Mx2PE-hystérésis) (48)

Remarque: la propriété indiquant que PS-Mx2PE devient faux à l'instant où Période<Mx2PE_hyst, apparaît dans la propriété 19.

### 1.2.6 Observation de la sortie "commutation possible"

Le changement de niveau valide de bas niveau en haut niveau, n'est possible que lorsque, d'une part, le niveau valide est effectivement le niveau bas, et d'autre part, la valeur de Flux-BN est supérieure à P1:

*toujours* (CommutBH *équivalent à* (BN-valide *et* Flux-BN > P1)) (49)

### 1.2.7 Observation des sorties "alarme"

Le déclenchement d'une alarme correspond au passage du seuil correspondant. L'alarme sur le passage du seuil MinBN ne peut cependant pas être déclenchée au démarrage du réacteur. En effet, il est alors normal que le flux capté par le bas niveau reste inférieur à MinBN durant un

certain temps. Le système gère alors une variable: init, qui est vraie à l'instant initial et tant que le flux n'a pas dépassé le seuil d'alarme MinBN+ hystérésis.

*toujours* (Al-MinBN *équivalent à* (PS-MinBN *et non* init)) (50)

*toujours* (Al-Mx1BN *équivalent à* PS-Mx1BN) (51)

*toujours* (Al-MinHN *équivalent à* PS-MinHN) (52)

*toujours* (Al-Mx1HN *équivalent à* PS-Mx1HN) (53)

*toujours* (Al-Mx1PE *équivalent à* PS-Mx1PE) (54)

### 1.2.8 Observation des sorties "arrêt d'urgence"

Le déclenchement des arrêts d'urgence correspond lui aussi, au passage des seuils correspondants:

*toujours* (AU-Mx2BN *équivalent à* PS-Mx2BN) (55)

*toujours* (AU-Mx2HN *équivalent à* PS-Mx2HN) (56)

*toujours* (AU-Mx2PE *équivalent à* PS-Mx2PE) (57)

### 1.2.9 Observation de la sortie "haute tension"

La haute tension n'est en marche que lorsque le bas niveau est valide ou que la valeur de Flux-HN est inférieure au seuil de coupure (FINRC):

*toujours* (MarcheHT *équivalent à*

(BN-valide *ou* (Flux-HN < FINRC *et* HN-actif))) (58)

Le bas niveau doit être actif à l'instant initial et chaque fois qu'au cycle précédent, la commande MarcheHT aura été émise. En dehors de ces cas là, le bas niveau doit être arrêté.

*toujours* ((*initial ou au cycle précédent* MarcheHT)*équivalent à* BN-actif) (59)

## 1.3 PROPRIÉTÉS GÉNÉRALES

Toute la gestion des capteurs du bas et du haut niveau a été développée afin d'avoir une meilleure connaissance du flux à l'intérieur du réacteur. La représentation de ce flux a été définie comme suit:

Flux = Si BN-valide alors Flux-BN sinon Flux-HN

On désire que toutes les valeurs prises par le flux (en supposant que les afficheurs de Flux-BN et de Flux-HN le permettent) puissent être captées et affichées,

c'est-à-dire:

*toujours* ($\forall$ x $\in$ [0, X], *possible* (Flux = x))

On en déduit donc qu'il faut les propriétés suivantes:

*toujours* ($\forall$ x $\in$ [0, FINRC], *possible* (Flux-BN = x)) (60)

*toujours* ($\forall$ x $\in$ [DEBRC, X], *possible* (Flux-HN = x)) (61)

(*possible* (HN-actif)) *chaque fois que* BN-actif (62)

(*possible* (BN-actif)) *chaque fois que* HN-actif (63)

(*possible* (HN-valide)) *chaque fois que* BN-valide (64)

(*possible* (BN-valide)) *chaque fois que* HN-valide (65)

*initialement* (BN-valide *ou* HN-valide)

*initialement* (BN-actif *ou* HN-actif)

En fait, comme l'état initial n'apparaît qu'au démarrage du réacteur, le flux est faible à cet instant et par conséquent ne peut être capté que par le bas niveau. On désire donc que ce niveau soit le seul valide au départ:

*initialement* BN-valide (66)

et qu'il soit en état de marche:

*initialement* BN-actif (67)

La régulation de l'activité des chaînes doit être la suivante : le bas niveau doit être actif chaque fois que la chaîne haut niveau n'est pas valide, ou que le flux_capté sur le haut niveau, lorsque celui-ci est valide, n'est pas supérieur à FINRC, et uniquement dans ce cas là.

*toujours* ( BN-actif *équivalent à*

(BN-valide *ou* (HN-valide *et* flux-HN < FINRC)) ) (68)

Le haut niveau doit être actif chaque fois que la chaîne bas niveau n'est pas valide, ou que le flux capté sur le bas niveau, lorsque celui-ci est valide, n'est pas inférieur à DEBRC, et uniquement dans ce cas là.

*toujours* (HN-actif *équivalent à*

(HN-valide *ou* (BN-valide *et* flux-BN > DEBRC)) ) (69)

Nous définissons comme suit la variable init qui caractérise l'état de démarrage du réacteur pendant lequel l'alarme Al-MinBN ne peut être déclenchée. Cette variable est vraie entre l'instant initial et le premier instant où le flux est supérieur à MinBN+hystérésis. Elle est ensuite fausse à jamais.

*toujours continuellement* init
*depuis initial jusqu'à* (Flux-BN > MinBN+hystérésis) (70)

Cette variable doit vérifier les propriétés suivantes:
(*non* init)*chaque fois que* (Flux-BN>MinBN+hystérésis) (71)
(*non* (*possible* (init)) *chaque fois que* (*non* init) (72)

**Remarques :**

• L'énoncé des propriétés ci-dessus est tel qu'il apparaissait avant la vérification. Au cours de cette vérification, la formulation de certaines propriétés s'est avérée erronée. Nous donnons ici la correction de ces propriétés :

*toujours continuellement* BN-valide *entre initial*
*et* (***au cycle précédent***
(*au cycle précedent* (Commut-BH) *et* (Commande-Op)) ) (38)

*toujours continuellement* BN-valide
*entre* (*au cycle précédent* (Flux-HN < P2 *et* HN-actif ***et*** **sens-flux**) )
*et* (***au cycle précédent***
(*au cycle précedent* (Commut-BH) *et* (Commande-Op)) ) (39)

*toujours continuellement* HN-valide
entre (***au cycle précédent***
(*au cycle précedent* (Commut-BH) *et* (Commande-Op)) )
*et* (*au cycle précédent* (Flux-HN < P2 *et* HN-actif ***et*** **sens-flux**) ) (40)

• Les propriétés (33) (34) (37) (60) (61) étant des propriétés arithmétiques ne peuvent être vérifiées par LESAR.

• Les propriétés (62) à (65), et (72), faisant intervenir la modalité temporelle *possible*, ne peuvent être vérifiées à l'aide de LESAR.

# 2 Equivalence des deux définitions des opérateurs *toujours cont(exist) g entre g1 et g2*

## 2.1 DÉMONSTRATION PRÉLIMINAIRE

Nous allons faire ici la démonstration d'une propriété générale des états qui sera utile pour nombre des démonstrations qui vont suivre. Cette propriété est la suivante (elle est intuitivement évidente):

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, \forall g$ **formule bien formée,**

$(k \geq 1 \wedge \exists k', k<k' \wedge s_{k'} \models g) \Rightarrow$

$(\exists k'_1, k<k'_1 \leq k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \not\models g)$ (A)

Ceci va être réalisé par induction sur la distance entre $s_k$ et $s_{k'}$

(avec $d(s_k, s_{k'}) = |k-k'|$):

*k' = k+1:*

$(\forall k", k<k"<k' \Rightarrow s_{k"} \not\models g) \equiv$ true puisqu'il n'existe pas k" vérifiant k<k"<k'

donc $(\exists k'_1, k'_1=k'=k+1 \wedge k'_1>k \wedge s_{k'_1} \models g \wedge \forall k", k'_1 >k">k \Rightarrow s_{k"} \not\models g)$

donc $(\exists k'_1, k' \geq k'_1>k \wedge s_{k'_1} \models g \wedge \forall k", k'_1 >k">k \Rightarrow s_{k"} \not\models g)$

*Supposons que* $\forall p, 1 \leq p \leq n, (\exists k'=k+p, k'>k \wedge s_{k'} \models g)$

$\Rightarrow (\exists k'_1, k'<k'_1 \leq k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \not\models g)$

*alors montrons que* $\exists k'=k+n+1, k<k' \wedge s_{k'} \models g)$

$\Rightarrow (\exists k'_1, k<k'_1 \leq k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \not\models g)$

Deux cas (complémentaires) sont à étudier:

k' est tel que $\forall k", k<k"<k' \Rightarrow s_{k"} \not\models g$

k' est tel que $\exists k", k<k"<k' \wedge s_{k"} \models g$

Soit k' tel que $\forall k", k<k"<k' \Rightarrow s_{k"} \not\models g$, la propriété est démontrée en prenant $k'_1 = k'$

Soit k' tel que $\exists k", k<k"<k' \Rightarrow s_{k"} \models g$,

$k<k"<k' \wedge k'=k+n+1 \Rightarrow k"=k+p \wedge 1 \leq p \leq n$

On peut alors appliquer l'hypothèse d'induction:

$(\exists k'_1, k<k'_1 \leq k" \wedge s_{k'_1} \models g \wedge \forall k"', k<k"'<k'_1 \Rightarrow s_{k"'} \models$ non g)

sachant que k"<k', la propriété est démontrée pour n+1 :
$(\exists k'_1, k<k'_1\leq k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \models$ non g)
*La propriété est alors vraie pour tout n.*

Nous ferons référence par la suite à la propriété (A) en parlant de $s_{k'_1}$ comme "le plus proche état à droite" de $s_k$ vérifiant la propriété g. Remarquons que la démonstration aurait été similaire pour la propriété:
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, \forall g$ **formule bien formée,**
$(k\geq 1 \wedge \exists k', 1\leq k'<k \wedge s_{k'} \models g) \Rightarrow$
$(\exists k'_1, k'\leq k'_1<k \wedge s_{k'_1} \models g \wedge \forall k", k'_1<k"<k \Rightarrow s_{k"} \not\models g)$
Dans ce dernier cas, nous parlerons de $s_{k'_1}$ comme "le plus proche état à gauche" de $s_k$ vérifiant la propriété g.

De même, nous pouvons démontrer la propriété suivante:
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, \forall g$ **formule bien formée,**
$(k\geq 1 \wedge \exists k', k\leq k' \wedge s_{k'} \models g) \Rightarrow$
$(\exists k'_1, k\leq k'_1\leq k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \not\models g)$ (B)

Pour cela, il suffit de changer dans la démonstration de (A), l'étude du premier pas d'induction en k'=k, au lieu de k'=k+1, ce qui nous donne:

Alors, k est tel que $s_k \models g$, auquel cas,
$\exists k'_1, k'_1=k=k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \not\models g$
donc $\exists k'_1, k\leq k'_1\leq k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \not\models g$
donc: $\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, \forall g$ formule bien formée,
$(\exists k', k'=k \wedge s_{k'} \models g) \Rightarrow (\exists k'_1, k\leq k'_1\leq k' \wedge s_{k'_1} \models g \wedge \forall k", k<k"<k'_1 \Rightarrow s_{k"} \not\models g)$

Le deuxième pas d'induction reste identique à celui de la démonstration de (A).

De façon similaire, nous aurions pu montrer la propriété:
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, \forall g$ **formule bien formée,**
$(k\geq 1 \wedge \exists k', 1\leq k'\leq k \wedge s_{k'} \models g) \Rightarrow$
$(\exists k'_1, k'\leq k'_1\leq k \wedge s_{k'_1} \models g \wedge \forall k", k'_1<k"<k \Rightarrow s_{k"} \not\models g)$

## 2.2 TOUJOURS CONT G ENTRE G1 ET G2

1) En $q_{racine}$

définition intuitive:

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \wedge s_k \models g1$ et non g2 =>

$(\exists k', k'>k \wedge s_{k'} \models g2 \wedge \forall k'', k'>k''\geq k \Rightarrow s_{k''} \models g$ et non g2) (B")

$\vee$ $(\forall k', k' \geq k \Rightarrow s_{k'} \models g$ et non g2) (B')

définition concise:

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \wedge s_k \models g1$ et non g2 =>

$\forall k'', (k'' \geq k \wedge s_{k''} \models$ non g => $\exists k', k'' \geq k' \geq k \wedge s_{k'} \models g2)$ (A)

La preuve d'équivalence des deux définitions va être établie en montrant que pour une séquence d'exécution s quelconque à partir de q, et un état quelconque $s_k$ de cette séquence tel que $k \geq 1 \wedge s_k \models g1$ et non g2 :

$s_k \models g1$ et non g2 => [(A) $\equiv$ (B")$\vee$ (B')]

On s'intéresse dans un premier temps à la formule $s_k \models g1$ et non g2 => [(A)=>(B")$\vee$(B')] c.a.d

**$[s_k \models$ g1 et non g2 $\wedge$ (A)] => [(B") $\vee$ (B')]**

Sachant que $s_k \models g1$ et non g2, $s_{k'}$ qui satisfait g2, ne peut pas être égal à $s_k$.

De ce fait, (A) c.a.d $\forall k'', (k'' \geq k \wedge s_{k''} \models$ non g => $\exists k', k'' \geq k' \geq k \wedge s_{k'} \models g2)$

implique $\forall k'', (k'' \geq k \wedge s_{k''} \models$ non g => $\exists k', k'' \geq k' > k \wedge s_{k'} \models g2)$. (A1)

(A1) se réécrit en :

$\forall k'', (k'' \geq k \wedge s_{k''} \models$ non g => $\exists k', k'' \geq k' > k \wedge s_{k'} \models g2)$

$\wedge$ $[(\forall k'_1, k'_1 > k \Rightarrow s_{k'1} \models$ non g2) $\vee (\exists k'_1, k'_1 > k \wedge s_{k'1} \models g2)]$

soit, en distribuant le "et" sur le deuxième terme :

$[\forall k'', (k'' \geq k \wedge s_{k''} \models$ non g => $\exists k', k'' \geq k' > k \wedge s_{k'} \models g2)$ (t1)

$\wedge (\forall k'_1, k'_1 > k \Rightarrow s_{k'1} \models$ non g2)] (t2)

$\vee$ $[\forall k'', (k'' \geq k \wedge s_{k''} \models$ non g => $\exists k', k'' \geq k' > k \wedge s_{k'} \models g2)$

$\wedge (\exists k'_1, k'_1 > k \wedge s_{k'1} \models g2)]$ (A")

puis, en remarquant que (t2) contient la négation de la conclusion de (t1), on peut réécrire le premier terme de la disjonction, par :

$(\forall k'', k'' \geq k \Rightarrow s_{k''} \models g) \wedge (\forall k'_1, k'_1 > k \Rightarrow s_{k'1} \models$ non g2) (A')

Pour montrer [$s_k \models$ g1 et non g2 $\wedge$ (A)] => [(B") $\vee$ (B')], nous allons montrer que :
[$s_k \models$ g1 et non g2 $\wedge$ ((A')$\vee$ (A"))] => [(B") $\vee$ (B')]
sachant que [$s_k \models$ g1 et non g2 $\wedge$ (A)] => [(A') $\vee$ (A")]

En examinant les formules (A') et (B'), on constate la relation suivante entre elles:
$s_k \models$ g1 et non g2 => [(A') $\equiv$ (B')]
Par conséquent $s_k \models$ g1 et non g2 => [(A') => (B") $\vee$ (B')].

Il reste donc à montrer : [$s_k \models$ g1 et non g2 $\wedge$ (A")] => [(B") $\vee$ (B')]

Pour cela, nous allons montrer une propriété plus forte, à savoir :
[$s_k \models$ g1 et non g2 $\wedge$ (A")] => (B").

La démonstration se fera en deux étapes:
(a) $\exists k'$, $k'>k \wedge s_{k'} \models g2 \wedge \forall k''$, $k' >k'' \geq k \Rightarrow s_{k''} \models$ non g2
(b) pour le k' déterminé dans la première étape: $\forall k''$, $k'>k'' \geq k \Rightarrow s_{k''} \models g$

Pour montrer (a), il convient de montrer que
$(\exists k'_1, k'_1>k \wedge s_{k'1} \models g2) \Rightarrow (\exists k', k'>k \wedge s_{k'} \models g2 \wedge \forall k'', k'>k'' \geq k \Rightarrow s_{k''} \models$ non g2)
l'existence de $k'_1$ étant énoncée dans (A")

Or nous avons montré *(démonstration préliminaire (annexe §3.1))* l'existence de k' tel que
$k'>k \wedge s_{k'} \models g2 \wedge \forall k''$, $k'>k''>k \Rightarrow s_{k''} \not\models g2$
Il suffit alors de rappeler que $s_k \models$ non g2, pour avoir:
$(\exists k'_1, k'_1>k \wedge s_{k'} \models g2) \Rightarrow (\exists k', k'>k \wedge s_{k'} \models g2 \wedge \forall k'', k'>k'' \geq k \Rightarrow s_{k''} \models$ non g2).
La prémisse de cette dernière formule étant une hypothèse, la conclusion est donc déductible.

Montrons alors (b) par l'absurde:
Soit k" tel que $k'>k'' \geq k \wedge s_{k''} \not\models g$
d'après (A"): $\exists k'_2$, $k'' \geq k'_2 > k \wedge s_{k'2} \models g2$
or $k'' \geq k'_2 > k \Rightarrow k' > k'_2 > k$
et donc d'après (a): $s_{k'2} \not\models g2$
on obtient une contradiction invalidant l'hypothèse:
$\neg(\exists k''$, $k'>k'' \geq k \wedge s_{k''} \not\models g)$
$\Rightarrow \forall k''$, $k'>k'' \geq k \Rightarrow s_{k''} \models g$

Nous allons maintenant montrer la deuxième partie de l'équivalence des définitions, c.a.d
**[$s_k \models$ g1 et non g2 $\wedge$ ((B") $\vee$(B'))] => (A)**
Nous montrerons [$s_k \models$ g1 et non g2 $\wedge$ ((B") $\vee$ (B'))] => [(A')$\vee$(A")]
car (A')$\vee$(A") est équivalent à (A1), et (A1) => (A)

Comme $s_k \models g1$ et non g2 => [(A') ≡ (B')],

$s_k \models g1$ et non g2 ∧ (B') => (A') => (A')∨(A").

Pour conclure la démonstration, il reste à prouver [$s_k \models g1$ et non g2 ∧ (B")] => [(A')∨(A")], ce que nous montrerons par [$s_k \models g1$ et non g2 ∧ (B")] => (A") .

Prenons $k'_1$dans (A") = k' dans (B");

le deuxième terme de (A") est établi.

La démonstration du premier terme de (A") est réalisée sous la forme d'une analyse par cas des positions relatives de $k"_1$ de (A") par rapport à celles de k' de (B"):

cas $k"_1 > k'$

$\forall k"_1, k"_1 \geq k'$, d'après (B") on déduit :

$\exists k'_2, k'_2 = k' \wedge k"_1 \geq k'_2 > k \wedge s_{k'2} \models g2$.

On peut donc construire la formule suivante valide :

$(k"_1 \geq k \wedge s_{k"1} \models$ non g => $\exists k'_2, k"_1 \geq k'_2 > k \wedge s_{k'2} \models g2)$

le premier terme de (A") est montré .

cas $k' > k"_1 \geq k$

$\forall k"_1, k' > k"_1 \geq k$

=> $s_{k"1} \models$ g et non g2 d'après (B").

=> $(k"_1 \geq k \wedge s_{k"1} \models$ non g) ≡ false

La formule suivante est donc valide :

$(k"_1 \geq k \wedge s_{k"1} \models$ non g => $\exists k'_2, k"_1 \geq k'_2 > k \wedge s_{k'2} \models g2)$

le deuxième terme de (A") est montré.

Ceci achève notre démonstration.

2) pour $q \neq q_{racine}$

Remarquons que la démonstration ci dessus (1) est indépendante du fait que la séquence d'exécution débute en qracine et que k soit supérieur à 1. La même démonstration s'applique donc au cas $q \neq q_{racine}$.

## 2.3 TOUJOURS EXIST G ENTRE G1 ET G2

1) En $q_{racine}$

définition intuitive:

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \wedge s_k \models g1 \text{ et non } g2 \Rightarrow$

$(\exists k', k' > k \wedge s_{k'} \models g2 \wedge \exists k'', k' > k'' \geq k \wedge s_{k''} \models g \text{ et non } g2$

$\wedge \forall k''', k' > k''' \geq k \Rightarrow s_{k'''} \models \text{non } g2)$ (B')

$\vee (\forall k', k' \geq k \Rightarrow s_{k'} \models \text{non } g2 \wedge \exists k'', k'' \geq k \wedge s_{k''} \models g)$ (B")

définition concise:

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \wedge s_k \models g1 \text{ et non } g2 \Rightarrow$

$\exists k', k' \geq k \wedge ((\forall k'', k' \geq k'' \geq k \Rightarrow s_{k''} \models \text{non } g2) \wedge s_{k'} \models g)$ (A)

Soit s une séquence d'exécution quelconque à partir de q, $s_k$ un état quelconque de cette séquence tel $k \geq 1 \wedge s_k \models g1$ et non g2, montrons que $s_k \models g1$ et non g2 => [(A) ≡ (B')∨ (B")].

(A) peut être réécrit en :

$(\exists k', k' \geq k \wedge (\forall k'', k' \geq k'' \geq k \Rightarrow s_{k''} \models \text{non } g2) \wedge s_{k'} \models g)$

$\wedge \quad [(\forall k'_1, k'_1 > k \Rightarrow s_{k'1} \models \text{non } g2) \vee (\exists k'_1, k'_1 > k \wedge s_{k'1} \models g2)]$

qui donne par distribution du "et" sur le second terme, et par regroupement des états satisfaisant non g2 :

$[\exists k', k' \geq k \wedge s_{k'} \models g \wedge (\forall k'_1, k'_1 \geq k \Rightarrow s_{k'1} \models \text{non } g2)]$ (A")

$\vee \quad [(\exists k', k' \geq k \wedge (\forall k'', k' \geq k'' \geq k \Rightarrow s_{k''} \models \text{non } g2) \wedge s_{k'} \models g)$

$\wedge (\exists k'_1, k'_1 > k \wedge s_{k'1} \models g2)]$ (A')

La nouvelle formule à prouver est donc : $s_k \models g1$ et non g2 => [(A")∨ (A') ≡ (B')∨ (B")].

Après avoir développé cette formule en une conjonction d'implications, et constaté que (A") est exactement équivalent à (B"), il reste à prouver :

$s_k \models g1$ et non g2 => [(A') => (B')∨ (B")] ∧ [(B') => (A")∨ (A')].

Nous allons montrer tout d'abord que $s_k \models g1$ et non g2 => [(A') => (B')] :

sachant que d'après (A') :

$\exists k'_1, k'_1 > k \wedge s_{k'1} \models g2$

nous pouvons en déduire*(cf démonstration préliminaire (annexe §3.1))* :

$\exists k', k' > k \wedge s_{k'} \models g2 \wedge \forall k''', k'_1 > k''' > k \Rightarrow s_{k'''} \models \text{non } g2$

et comme $s_k \models$ non g2 :

$\exists k', k' > k \wedge s_{k'} \models g2 \wedge \forall k''', k'_1 > k''' \geq k \Rightarrow s_{k'''} \models \text{non } g2$

il reste donc à montrer que $s_k \models g1$ et non g2 ∧ (A') => $\exists k'', k' > k'' \geq k \wedge s_{k''} \models g$ et non g2

(A') => $\exists k''$, $k'' \geq k \wedge s_{k''} \models g$ et non g2

Montrons par l'absurde que k'>k"
Supposons k"≥k' alors ($\exists k_1 = k'$, $k'' \geq k_1 \geq k \wedge s_{k1} \models g2$)
or (A') => ($\forall k_1$, $k'' \geq k_1 \geq k \Rightarrow s_{k1} \models$ non g2) d'où contradiction.
donc k'>k", et donc $\exists k''$, $k' > k'' \geq k \wedge s_{k''} \models g$ et non g2. CQFD

Pour conclure la preuve, nous allons montrer que $s_k \models g1$ et non g2 => [(B') => (A')] :
Le premier terme de (A') est déduit de (B') en prenant k' dans (A') égal au k" de (B').
Le deuxième terme de (A') est montré en prenant $k'_1 = k'$ de (B').

2) pour $q \neq q_{racine}$

Remarquons que la démonstration ci dessus (1) est indépendante du fait que la séquence d'exécution débute en qracine et que k soit supérieur à 1. La même démonstration s'applique donc au cas $q \neq q_{racine}$

# 3 Preuve de la correction des traductions en MODAL LUSTRE et en INVARIANTS LUSTRE.

## 3.1 SÉMANTIQUE DES NŒUDS LUSTRE CONT, APRES, CONT_DEPUIS ET ILEXISTE_DEPUIS SUR UN ARBRE D'EXÉCUTION.

### 3.1.1 Cont(a)

Montrons:
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$s_k \models Cont(a)$ ssi $\forall k', 1 \leq k' \leq k \Rightarrow s_{k'} \models a$

Par définition, Cont(a) = a -> a and pre Cont(a)
La sémantique intuitive des opérateurs LUSTRE nous donne:
$s_k \models Cont(a)$ ssi $(k=1 \Rightarrow s_k \models a) \wedge (k \neq 1 \Rightarrow s_k \models a \wedge s_{k-1} \models Cont(a))$

Soit s quelconque tel que $s \in EX(q_{racine})$, soit k naturel quelconque tel que $k \geq 1$:
1er cas: k=1:
$s_k$ étant un état initial, $s_k \models Cont(a)$ ssi $s_k \models a$ ssi $\forall k', 1 \leq k' \leq k \Rightarrow s_{k'} \models a$
2e cas: k≠1
Supposons la propriété vraie pour k=1,..., n-1, montrons la par récurrence pour k=n.
$s_n \models Cont(a)$ ssi $s_n \models a \wedge s_{n-1} \models Cont(a)$
ssi $s_n \models a \wedge \forall k', 1 \leq k' \leq n-1 \Rightarrow s_{k'} \models a$ ssi $\forall k', 1 \leq k' \leq n \Rightarrow s_{k'} \models a$

### 3.1.2 Après(a)

Puisque Après(a) = not Cont (not a), on obtient :
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$s_k \models$ Après(a) ssi $\exists k', 1 \leq k' \leq k \wedge s_{k'} \models a$

### 3.1.3 Cont_Depuis (b, a)

Il faut montrer que:

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$

$s_k \models$ Cont_Depuis(b, a) ssi

$\forall k', 1 \leq k' \leq k \Rightarrow s_{k'} \not\models a$

$\vee$ ( $\exists k', 1 \leq k' \leq k \wedge s_{k'} \models a$

$\wedge$ $\forall k'', k' \leq k'' \leq k \Rightarrow s_{k''} \models b$

$\wedge$ $\forall k''', k' < k''' \leq k \Rightarrow s_{k'''} \not\models a$ ) (CD)

Par définition :

Cont_Depuis(b, a) = not a or b ->

Cont(not a) or (a and b) or (b and pre Cont_Depuis(b, a))

La sémantique intuitive des opérateurs LUSTRE permet de réécrire ceci en :

$s_k \models$Cont_Depuis(b, a) ssi

$[k=1 \Rightarrow s_k \not\models a \vee s_k \models b]$

$\wedge$ $[k \neq 1 \Rightarrow$ $s_k \models$Cont(not a)

$\vee$ $(s_k \models a \wedge s_k \models b)$

$\vee$ $(s_k \models b \wedge s_{k-1} \models$Cont_Depuis(b, a))]

Soit s quelconque tel que $s \in EX(q_{racine})$, soit k naturel quelconque tel que $k \geq 1$:

<u>1er cas: k=1</u>:

$k=1 \wedge s_k \models$Cont_Depuis(b, a)

ssi $k=1 \wedge (s_k \not\models a \vee s_k \models b)$

ssi $k=1 \wedge (s_k \not\models a \vee (s_k \models a \wedge s_k \models b))$

ssi $k=1 \wedge [ (\forall k', 1 \leq k' \leq k \Rightarrow s_{k'} \not\models a)$

$\vee$ $( \exists k', 1 \leq k' \leq k \wedge s_{k'} \models a$

$\wedge \forall k'', k' \leq k'' \leq k \Rightarrow s_{k''} \models b$

$\wedge \forall k''', k' < k''' \leq k \Rightarrow s_{k'''} \models a)]$

<u>2e cas: k≠1</u>:

Supposons la propriété vraie pour k=1,..., n-1, montrons la par récurrence pour k=n.

$s_n \models$Cont_Depuis(b, a)

ssi $s_n \models$Cont(not a)

$\vee$ $(s_n \models a \wedge s_n \models b)$

$\vee$ $(s_n \models b \wedge s_{n-1} \models$Cont_Depuis(b, a))]

Ce qui en rendant tous les cas disjoints, devient:

$s_n \models Cont(not\ a)$

$\vee$ $(s_n \models a \wedge s_n \models b)$

$\vee$ $(s_n \not\models a \wedge s_n \models b \wedge s_{n-1} \models Cont\_Depuis(b, a))]$

On expanse alors les définitions des différents nœuds:

$(\forall k', 1 \leq k' \leq n \Rightarrow s_{k'} \not\models a)$ (A)

$\vee$ $(s_n \models a \wedge s_n \models b)$ (B)

$\vee$ $(s_n \not\models a \wedge s_n \models b \wedge [ \quad (\forall k', 1 \leq k' \leq n-1 \Rightarrow s_{k'} \not\models a)$

$\vee$ $( \quad \exists k', 1 \leq k' \leq n-1 \wedge s_{k'} \models a$

$\wedge \quad \forall k'', k' \leq k'' \leq n-1 \Rightarrow s_{k''} \models b$

$\wedge \quad \forall k''', k' < k''' \leq n-1 \Rightarrow s_{k'''} \not\models a )])$ (C)

(B) peut être réécrit en (B') soit :

$( \exists k', k' = n \wedge s_{k'} \models a$

$\wedge \forall k'', k' \leq k'' \leq n \Rightarrow s_{k''} \models b$

$\wedge \forall k''', k' < k''' \leq n \Rightarrow s_{k'''} \not\models a)$ (B')

et dans le troisième terme de la disjonction c.a.d (C), on peut distribuer $s_n \not\models a \wedge s_n \models b$, pour obtenir:

$(s_n \models b \wedge (\forall k', 1 \leq k' \leq n-1 \vee k' = n \Rightarrow s_{k'} \not\models a))$

$\vee$ $( \exists k', 1 \leq k' \leq n-1 \wedge s_{k'} \models a$

$\wedge \forall k'', k' \leq k'' \leq n-1 \vee k'' = n \Rightarrow s_{k''} \models b$

$\wedge \forall k''', k' < k''' \leq n-1 \vee k''' = n \Rightarrow s_{k'''} \not\models a )$

le premier terme de cette nouvelle disjonction est inclus dans (A), il peut donc être supprimé; on appelle le reste (C').

$s_n \models Cont\_Depuis(b, a)$ est alors équivalent à (A) $\vee$ (B') $\vee$ (C').

Par regroupement de (B') et (C'), on obtient la formule à prouver:

$(\forall k', 1 \leq k' \leq n \Rightarrow s_{k'} \not\models a)$

$\vee$ $( \exists k', 1 \leq k' \leq n \wedge s_{k'} \models a$

$\wedge \forall k'', k' \leq k'' \leq n \Rightarrow s_{k''} \models b$

$\wedge \forall k''', k' < k''' \leq n \Rightarrow s_{k'''} \not\models a )$

### 3.1.4 IlExiste_Depuis(b, a)

Puisque IlExiste_Depuis(b, a) = not Cont_Depuis(not b, a), nous obtenons par négation de la formule (CD):

$s_k \models$ IlExiste_Depuis(b, a)

ssi $\exists k', 1 \leq k' \leq k \wedge s_{k'} \models a$

$\wedge$ ( $\forall k', 1 \leq k' \leq k \Rightarrow s_{k'} \not\models a$

$\vee$ $\exists k'', k' \leq k'' \leq k \wedge s_{k''} \models b$

$\vee$ $\exists k''', k' < k''' \leq k \wedge s_{k'''} \models a$)

On peut supprimer $\forall k', 1 \leq k' \leq k \Rightarrow s_{k'} \not\models a$ car $X \wedge (\neg X \vee Y) = X \wedge Y$.

De plus, on sait que *(cf démonstration préliminaire (annexe §3.1))* , s'il existe un état $s_{k'}$ antérieur à un état $s_k$ (c'est-à-dire tel que $k' \leq k$), vérifiant la propriété a, alors il existe un état $s_{k'1}$ *le plus proche à gauche* de $s_k$ vérifiant la même propriété a (c'est-à-dire que $\forall k'', k'_1 \leq k'' \leq k \Rightarrow s_{k''} \not\models a$).
Remarque $s_{k'}$ et $s_{k'1}$ peuvent être confondus.
Ceci rend inutile la présence de $\exists k''', k' < k''' \leq k \wedge s_{k'''} \models a$. Par conséquent :

**$s_k \models$ IlExiste_Depuis(b, a)**

**ssi** $\exists k'_1, 1 \leq k'_1 \leq k \wedge s_{k'1} \models a$

$\wedge$ $(\forall k'', k'_1 \leq k'' \leq k \Rightarrow s_{k''} \not\models a)$

$\wedge$ $(\exists k'', k'_1 \leq k'' \leq k \wedge s_{k''} \models b)$ )

## 3.2 PREUVE DE LA TRADUCTION DES OPÉRATEURS TOUJOURS EXIST G ENTRE $G_1$ ET $G_2$, PRÉSERVABLE CONT (EXIST) G ENTRE G1 ET G2 (STRICT) EN MODAL LUSTRE

### 3.2.1 toujours cont g entre $g_1$ et $g_2$

1) en $q = q_{racine}$

$q_{racine} \models$ **toujours cont g entre $g_1$ et $g_2$** ssi
$\forall s \in EX(q_{racine}), \forall k_2 \in \mathbb{N}, ( \quad k_2 \geq 1 \wedge (s_{k2} \models g_1$ et non $g_2) \quad =>$
$(\exists k'_2, k'_2 > k_2 \wedge s_{k'2} \models g_2$
$\wedge \forall k''_2, k'_2 > k''_2 \geq k_2 => s_{k''2} \models g$ et non $g_2)$
$\vee (\forall k'_2, k'_2 \geq k_2 => s_{k'2} \models g$ et non $g_2) \quad )$
ce qui est équivalent d'après la sémantique des opérateurs "et" et "non", à:
$\forall s \in EX(q_{racine}), \forall k_2 \in \mathbb{N}, ( \quad k_2 \geq 1 \wedge (s_{k2} \models g_1 \wedge s_{k2} \not\models g_2) \quad =>$
$(\exists k'_2, k'_2 > k_2 \wedge s_{k'2} \models g_2$
$\wedge \forall k''_2, k'_2 > k''_2 \geq k_2 => s_{k''2} \models g \wedge s_{k''2} \not\models g_2)$
$\vee (\forall k'_2, k'_2 \geq k_2 => s_{k'2} \models g \wedge s_{k'2} \not\models g_2) \quad ) \qquad (2)$

$q_{racine} \models$ **al (Cont_Depuis (g and not $g_2$, $g_1$ and not $g_2$) or IlExiste_Depuis($g_2$, $g_1$ and not $g_2$))** ssi
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 =>$
$s_k \models$Cont_Depuis (g and not $g_2$, $g_1$ and not $g_2$) or IlExiste_Depuis( $g_2$, $g_1$ and not $g_2$)

Remarque: ci-dessus, les notations devraient être T(g), T($g_1$) et T($g_2$) au lieu de g, $g_1$, et $g_2$, puisqu'il s'agit de la traduction de la formule, et donc de ses sous-formules. Nous avons préféré ne pas alourdir la notation, sachant que la démonstration se fait par induction (sur la longueur de la chaîne en nombre d'opérateurs), et que par hypothèse d'induction T(g), T($g_1$) et T($g_2$) sont des traductions correctes et peuvent donc, par un abus de langage, être considérées comme g, $g_1$ et $g_2$.

Par expansion des nœuds Cont_depuis, et IlExiste_Depuis, et selon la sémantique des opérateurs not et and, on obtient :

$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 =>$

$\forall k', 1 \leq k' \leq k => s_{k'} \not\models g_1 \vee s_{k'} \models g_2)$

$\vee$ ( $\exists k', 1 \leq k' \leq k \wedge s_{k'} \models g_1 \wedge s_{k'} \not\models g_2$

$\wedge$ $\forall k'', k' \leq k'' \leq k => s_{k''} \models g \wedge s_{k''} \not\models g_2$

$\wedge$ $\forall k''', k' < k''' \leq k => s_{k'''} \not\models g_1 \vee s_{k'''} \models g_2$ )

$\vee$ ( $\exists k', 1 \leq k' \leq k \wedge s_{k'} \models g_1 \wedge s_{k'} \not\models g_2$

$\wedge$ $(\forall k''', k' < k''' \leq k => s_{k'''} \not\models g_1 \vee s_{k'''} \models g_2 => \exists k'', k' \leq k'' \leq k => s_{k''} \models g_2))$

On peut alors simplifier la cinquième ligne par $s_{k'''} \models g_2$ au vu de la ligne précédente. Nous appellerons (1) la formule ainsi obtenue.

Pour montrer la correction de notre traduction, il suffit de montrer que (1) $\equiv$ (2).

Cela se fera en deux temps, tout d'abord, montrons (1) => (2):

Soit $s \in EX(q_{racine})$, soit $k_2 \geq 1$, tels que $s_{k2} \models g_1 \wedge s_{k2} \not\models g_2$

montrons,

$(\exists k'_2, k'_2 > k_2 \wedge s_{k'2} \models g_2$

$\wedge \forall k''_2, k'_2 > k''_2 \geq k_2 => s_{k''2} \models g \wedge s_{k''2} \not\models g_2)$

$\vee (\forall k'_2, k'_2 \geq k_2 => s_{k'2} \models g \wedge s_{k'2} \not\models g_2)$ )

Séparons notre étude en deux cas complémentaires:

a) 1er cas: $\exists k'_3, k_2 < k'_3 \wedge s_{k'3} \models g_2$

Nous avons montré que cela implique l'existence d'un état $s_{k'2}$ "le plus proche à droite" de $s_{k2}$ vérifiant $g_2$, c'est-à-dire:

$\exists k'_2, k_2 < k'_2 \leq k'_3 \wedge s_{k'2} \models g_2 \wedge \forall k''_3, k_2 \leq k''_3 < k'_2 => s_{k''3} \not\models g_2$

d'après (1), puisque non $(\forall k', 1 \leq k' \leq k => s_{k'} \not\models g_1 \vee s_{k'} \models g_2)$ et

en prenant $k = k''_3$ et $k' = k_2$ puisque $k_2 \leq k''_3 < k'_2$, on a:

( $\exists k_2, 1 \leq k_2 \leq k''_3 \wedge s_{k2} \models g_1 \wedge s_{k2} \not\models g_2$

$\wedge \forall k'', k_2 \leq k'' \leq k''_3 => s_{k''} \models g \wedge s_{k''} \not\models g_2$

$\wedge \forall k''', k_2 < k''' \leq k''_3 => s_{k'''} \not\models g_1$ )

$\vee$ ( $\exists k_2, 1 \leq k_2 \leq k''_3 \wedge s_{k2} \models g_1 \wedge s_{k2} \not\models g_2$

$\wedge (\forall k''', k_2 < k''' \leq k''_3 => s_{k'''} \not\models g_1 \vee s_{k'''} \models g_2 => \exists k'', k_2 \leq k'' \leq k''_3 => s_{k''} \models g_2)$ )

donc $k''_3$ est tel que:

( $\forall k'', k_2 \leq k'' \leq k''_3 => s_{k''} \models g \wedge s_{k''} \not\models g_2$

$\wedge \forall k''', k_2 < k''' \leq k''_3 => s_{k'''} \not\models g_1$ )

$\vee$ ( $\forall k''', k_2 < k''' \leq k''_3 => s_{k'''} \not\models g_1 \vee s_{k'''} \models g_2 => \exists k'', k_2 \leq k'' \leq k''_3 => s_{k''} \models g_2$ )

or $\exists k", k_2 \leq k" \leq k"_3 \Rightarrow s_{k"} \models g_2 \equiv$ faux, par conséquent, $k"_3$ est tel que:

$( \forall k", k_2 \leq k" \leq k"_3 \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$

$\wedge \forall k"', k_2 < k"' \leq k"_3 \Rightarrow s_{k"'} \not\models g_1 \quad )$

$\vee ( \exists k"', k_2 < k"' \leq k"_3 \wedge s_{k"'} \models g_1 \vee s_{k"'} \not\models g_2 )$ (A)

et ce, pour tout $k"_3$ tel que $k_2 \leq k"_3 < k'_2$ .

On peut de nouveau, séparer l'étude en deux sous-cas:

a.a) 1er cas:$\neg (\exists l, k_2 \leq l < k'_2 \wedge s_l \models g_1 \vee s_l \not\models g_2)$

d'après (A) en prenant $k"_3 = k'_2 - 1$, $\forall k", k_2 \leq k" < k'_2 \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$

CQFD a.a

a.b) 2e cas: $\exists l, k_2 \leq l < k'_2 \wedge s_l \models g_1 \vee s_l \not\models g_2$

Nous avons montré que cela implique l'existence d'un état $s_{l'}$ "le plus proche à droite" de $s_{k2}$ vérifiant g et ne vérifiant pas $g_2$, c'est-à-dire:

$\exists l', k_2 \leq l' \leq l \wedge s_{l'} \models g_1 \vee s_{l'} \not\models g_2 \wedge \forall l", k_2 \leq l" < l' \Rightarrow \neg (s_{l"} \models g_1 \vee s_{l"} \not\models g_2)$

d'après (A) en prenant $k"_3 = l'$, $\forall k", k_2 \leq k" \leq l' \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$

supposons $\forall k", l' \leq k" < k'_2 \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$

alors $\forall k", k_2 \leq k" < k'_2 \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$

Pour montrer $\forall k", l' \leq k" < k'_2 \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$, on itère a.b en prenant $k_2 = l'$, jusqu'à aboutir au cas a.a (où $k_2 = l'$).

CQFD a.b

CQFD a

b) 2e cas: $\forall k'_2, k_2 < k'_2 \Rightarrow s_{k'2} \not\models g_2$

ce qui est équivalent à $\forall k'_2, k_2 \leq k'_2 \Rightarrow s_{k'2} \not\models g_2$, sachant que $s_{k2} \not\models g_2$

Montrons $s_{k'2} \models g$.

Soit $k'_2$ quelconque tel que $k_2 \leq k'_2$,

on démontre alors (même démonstration que lors du cas a, en bornant non plus par $k'_2$ tel que $s_{k'2} \models g_2 \wedge \forall k"_3, k_2 \leq k"_3 < k'_2 \Rightarrow s_{k"3} \not\models g_2$, mais par $k'_3$ quelconque tel que $k_2 < k'_3$ donc tel que $\forall k"_3, k_2 \leq k"_3 \leq k'_3 \Rightarrow s_{k"3} \not\models g_2$ par l'hypothèse b) que

$\forall k'_3, k_2 < k'_3 \Rightarrow \forall k", k_2 \leq k" \leq k'_3 \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$.

Or, $\exists k'_3, k_2 < k'_3$ (propriété de $\mathbb{N}$) et donc, $\forall k", k_2 \leq k" \leq k'_3 \Rightarrow s_{k"} \models g \wedge s_{k"} \not\models g_2$.

Or $k_2 \leq k'_2 < k'_3$ et donc $s_{k'2} \models g \wedge s_{k'2} \not\models g_2$, par conséquent on en déduit $s_{k'2} \models g$.

CQFD b

Nous avons donc montré (1) => (2); reste à montrer (2) => (1).

Soit $s \in EX(q_{racine})$, soit $k \geq 1$, nous pouvons séparer l'étude en deux cas:

a) $\forall k',\ 1 \leq k' \leq k \Rightarrow \neg\ (s_{k'} \models g_1 \wedge s_{k'} \not\models g_2)$

b) $\exists k',\ 1 \leq k' \leq k \wedge s_{k'} \models g_1 \wedge s_{k'} \not\models g_2$

Il suffit donc pour montrer (1), de montrer que dans le cas b, on a de plus:

$$( \forall k'',\ k' \leq k'' \leq k \Rightarrow s_{k''} \models g \wedge s_{k''} \not\models g_2$$
$$\wedge\ \forall k''',\ k' < k''' \leq k \Rightarrow s_{k'''} \not\models g_1 ) \qquad \text{(i)}$$
$$\vee\ ( \forall k''',\ k' < k''' \leq k \Rightarrow s_{k'''} \not\models g_1 \vee s_{k'''} \models g_2 \Rightarrow \exists k'',\ k' \leq k'' \leq k \Rightarrow s_{k''} \models g_2 ) \qquad \text{(ii)}$$

Pour montrer cela nous allons séparer le cas b en deux sous-cas:

b.a) $\exists k''',\ k' < k''' \leq k \wedge s_{k'''} \models g_1 \wedge s_{k'''} \not\models g_2$

auquel cas $\forall k''',\ k' < k''' \leq k \Rightarrow s_{k'''} \not\models g_1 \vee s_{k'''} \models g$ est équivalent à faux,

et donc (ii) est vraie.

CQFD b.a

b.b) $\forall k''',\ k' < k''' \leq k \Rightarrow \neg\ (s_{k'''} \models g_1 \wedge s_{k'''} \not\models g_2)$

La démonstration se fera suivant 3 sous-cas:

b.b.a) $\exists k'_2,\ k' < k'_2 \leq k \wedge s_{k'2} \models g_2$

$\Rightarrow \exists k'_2,\ k' \leq k'_2 \leq k \wedge s_{k'2} \models g_2$

par hypothèse b.b et en prenant $k'' = k'_2$ dans (ii) => CQFD b.b.a

b.b.b) $\exists k'_2,\ k' < k < k'_2 \wedge s_{k'2} \models g_2$

d'après (2) en prenant $k_2 = k'$:

$$(\exists k'_2,\ k'_2 > k' \wedge s_{k'2} \models g_2$$
$$\wedge\ \forall k''_2,\ k'_2 > k''_2 \geq k' \Rightarrow s_{k''2} \models g \wedge s_{k''2} \not\models g_2)$$
$$\vee\ (\forall k'_2,\ k'_2 \geq k' \Rightarrow s_{k'2} \models g \wedge s_{k'2} \not\models g_2) \quad ) \qquad \text{(B)}$$

d'où $\forall k''_2,\ k' \leq k''_2 < k'_2 \Rightarrow s_{k''2} \models g \wedge s_{k''2} \not\models g_2$

en particulier, $\forall k''_2,\ k' \leq k''_2 \leq k \Rightarrow s_{k''2} \models g \wedge s_{k''2} \not\models g_2$

par hypothèse b.b et en prenant $k'' = k''_2$ dans (i) CQFD b.b.b

b.b.c) $\forall k'_2,\ k' < k'_2 \Rightarrow s_{k'2} \not\models g_2$

ce qui est équivalent à $\forall k'_2,\ k' \leq k'_2 \Rightarrow s_{k'2} \not\models g_2$ car $s_{k'} \models g_1 \wedge s_{k'} \not\models g_2$

d'après (B), cela implique:

$\forall k'_2,\ k' \leq k'_2 \Rightarrow s_{k'2} \not\models g_2$ car $s_{k'} \models g \wedge s_{k'} \not\models g_2$

en particulier, $\forall k'_2,\ k' \leq k'_2 \leq k \Rightarrow s_{k'2} \not\models g_2$ car $s_{k'} \models g \wedge s_{k'} \not\models g_2$

par hypothèse b.b et en prenant $k'' = k'_2$ dans (i) => CQFD b.b.c

CQFD b.b

CQFD b

Nous avons alors conclu la démonstration de (2) => (1) et donc de (1) ≡ (2).

2) en $q \neq q_{racine}$
La démonstration ci-dessus n'utilisant pas le fait que $s \in EX(q_{racine})$, $k \geq 1$ et $k_2 \geq 1$, elle est valable en $q \neq q_{racine}$.

La traduction de *toujours cont g entre $g_1$ et $g_2$* en MODAL LUSTRE , est donc correcte.

### 3.2.2 toujours exist g entre $g_1$ et $g_2$

Remarque :
La preuve sera réalisée sur la propriété duale : POSSIBLE CONT g ENTRE g1 ET g2.

1) en $q = q_{racine}$

$q_{racine} \models$ possible cont g entre $g_1$ et $g_2$ ssi (définition intuitive)
$\exists s \in EX(q_{racine}), \exists k_1 \in \mathbb{N}, \quad k_1 \geq 1 \wedge (s_{k1} \models g_1$ et non $g_2) \quad \wedge$
$(\exists k', k' > k_1 \wedge s_{k'} \models g_2 \wedge \forall k'', k' > k'' \geq k_1 \Rightarrow s_{k''} \models g$ et non $g_2)$
$\vee$ $(\forall k', k' \geq k_1 \Rightarrow s_{k'} \models g$ et non $g_2)$
ssi
$\exists s \in EX(q_{racine}), \exists k_1 \in \mathbb{N}, \quad k_1 \geq 1 \wedge (s_{k1} \models g_1 \wedge s_{k1} \not\models g_2) \quad \wedge$
$(\exists k', k' > k_1 \wedge s_{k'} \models g_2 \wedge \forall k'', k' > k'' \geq k_1 \Rightarrow s_{k''} \models g \wedge s_{k''} \not\models g_2)$ (2.1)
$\vee$ $(\forall k', k' \geq k_1 \Rightarrow s_{k'} \models g \wedge s_{k'} \not\models g_2)$ (2.2) (2)

$q_{racine} \models$ pot (
[$g_2$ and Après ($g_1$ and not $g_2$) and pre Cont_Depuis (g and not $g_2$, $g_1$ and not $g_2$)]
or [$g_1$ and not $g_2$ and some (g and not $g_2$)] )

Remarque: ci-dessus, les notations devraient être T(g), T($g_1$) et T($g_2$) au lieu de g, $g_1$, et $g_2$, puisqu'il s'agit de la traduction de la formule, et donc de ses sous-formules. Nous avons préféré ne pas alourdir la notation, sachant que la démonstration se fait par induction (sur la longueur de la chaîne en nombre d'opérateurs), et que par hypothèse d'induction T(g), T($g_1$) et T($g_2$) sont des traductions correctes et peuvent donc, par un abus de langage, être considérées comme g, $g_1$ et $g_2$.

Par expansion des nœuds Après et Cont_Depuis, et d'après la sémantique des opérateurs Lustre, on obtient :

$\exists s \in EX(q_{racine}), \exists k \in \mathbb{N}, k \geq 1 \wedge$

$[s_k \models g_2 \wedge (\exists k_1, 1 \leq k_1 \leq k \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2)$

$\wedge \{ \quad (\forall k_2, 1 \leq k_2 \leq k-1 \Rightarrow \neg(s_{k2} \models g_1 \wedge s_{k2} \not\models g_2))$

$\vee \quad (\exists k_3, 1 \leq k_3 \leq k-1 \wedge s_{k3} \models g_1 \wedge s_{k3} \not\models g_2$

$\wedge \forall k_4, k_3 \leq k_4 \leq k-1 \Rightarrow s_{k4} \models g \wedge s_{k4} \not\models g_2$

$\wedge \forall k_5, k_3 < k_5 \leq k-1 \Rightarrow \neg(s_{k5} \models g_1 \wedge s_{k5} \not\models g_2))\}]$

$\vee [ s_k \models g_1 \wedge s_k \not\models g_2 \wedge \forall k', k \leq k' \Rightarrow s_{k'} \models g \wedge s_{k'} \not\models g_2 ]$

Or, les formules $(\exists k_1, 1 \leq k_1 \leq k \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2)$
et $(\forall k_2, 1 \leq k_2 \leq k-1 \Rightarrow \neg(s_{k2} \models g_1 \wedge s_{k2} \not\models g_2))$ sont incompatibles sauf en $s_k$, et la formule $(\forall k_2, 1 \leq k_2 \leq k-1 \Rightarrow \neg(s_{k2} \models g_1 \wedge s_{k2} \not\models g_2))$ est de plus incompatible avec $s_k \models g_2$ en $s_k$.

Par conséquent, la formule ci-dessus devient:

ssi

$\exists s \in EX(q_{racine}), \exists k \in \mathbb{N}, k \geq 1 \wedge$

$[s_k \models g_2 \wedge \exists k_1, 1 \leq k_1 \leq k \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$

$\wedge \exists k_3, 1 \leq k_3 \leq k-1 \wedge s_{k3} \models g_1 \wedge s_{k3} \not\models g_2$

$\wedge \forall k_4, k_3 \leq k_4 \leq k-1 \Rightarrow s_{k4} \models g \wedge s_{k4} \not\models g_2$

$\wedge \forall k_5, k_3 < k_5 \leq k-1 \Rightarrow \neg(s_{k5} \models g_1 \wedge s_{k5} \not\models g_2)]$ (1.1)

$\vee [ s_k \models g_1 \wedge s_k \not\models g_2 \wedge \forall k', k \leq k' \Rightarrow s_{k'} \models g \wedge s_{k'} \not\models g_2 ]$ (1.2)

Pour montrer la correction de notre traduction, il suffit de montrer que (1) ≡ (2).
Cela se fera en deux temps, **tout d'abord, montrons (1) => (2)**:

Soit $s \in EX(q_{racine})$, la séquence exhibée pour prouver (1), montrons que l'exhibition de la même séquence suffit à prouver (2).

a) *Supposons (1.2)* et montrons $\exists k_1 \in \mathbb{N}, k_1 \geq 1 \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2 \quad \wedge$
$\exists k', k' > k_1 \wedge s_{k'} \models g_2 \wedge \forall k'', k' > k'' \geq k_1 \Rightarrow s_{k''} \models g \wedge s_{k''} \not\models g_2,$
et par conséquent montrons (2).
Il suffit de prendre $k_1 = k$, et la propriété est démontrée.

b)*Supposons (1.1)* et montrons (2.1) c'est-à-dire
$\exists k_1 \in \mathbb{N}, k_1 \geq 1 \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2 \wedge \forall k', k' \geq k_1 \Rightarrow s_{k'} \models g \wedge s_{k'} \not\models g_2,$

et par conséquent, montrons (2).

(1.1) => $\exists k, \exists k_3, 1 \leq k_3 \leq k\text{-}1 \wedge s_{k3} \models g_1 \wedge s_{k3} \not\models g_2$

Prenons $k_1 = k_3$, et montrons (2.1).

On peut réécrire (1.1) en permutant les quantificateurs existantiels :

$$\exists k_3, k_3 \geq 1 \wedge s_{k3} \models g_1 \wedge s_{k3} \not\models g_2$$
$$\wedge \exists k, k_3 \leq k\text{-}1 \wedge s_k \models g_2$$
$$\wedge \exists k_1, 1 \leq k_1 \leq k \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$$
$$\wedge \forall k_4, k_3 \leq k_4 \leq k\text{-}1 \Rightarrow s_{k4} \models g \wedge s_{k4} \not\models g_2$$
$$\wedge \forall k_5, k_3 < k_5 \leq k\text{-}1 \Rightarrow \neg(s_{k5} \models g_1 \wedge s_{k5} \not\models g_2)$$

On sait donc (cf démonstration préliminaire (annexe §3.1)), qu'il existe k' (le "plus proche" à droite de $k_3$) tel que:

$k_3 < k' \leq k \wedge s_{k'} \models g_2 \wedge \forall k'', k_3 \leq k'' < k' \Rightarrow s_{k''} \not\models g_2$

Il reste alors à montrer que, de plus, $\forall k'', k_3 \leq k'' < k' \Rightarrow s_{k''} \models g$, ce qui est réalisé en constatant que:

$k_3 \leq k'' < k' \Rightarrow k_3 \leq k'' < k \Rightarrow s_{k''} \models g$ en prenant $k_4 = k''$ dans (1.1).

Nous avons donc montré (1) => (2); **reste à montrer (2) => (1).**

Soit $s \in EX(q_{racine})$, la séquence exhibée pour prouver (2), montrons que l'exhibition de la même séquence suffit à prouver (1).

a) *Supposons* $\exists k_1 \in \mathbb{N}, s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$ *et tel que (2.2)*

en prenant $k = k_1$, on prouve (1.2) et donc (1).

b) *Supposons* $\exists k_1 \in \mathbb{N}, s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$ *et tel que (2.1)*

montrons (1.1) et donc (1).

$\exists k_1, k_1 \geq 1 \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2 \wedge \exists k', k' > k_1 \wedge s_{k'} \models g_2$ est équivalent à:

$\exists k', s_{k'} \models g_2 \wedge \exists k_1, k' > k_1 \geq 1 \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$

On peut prouver (cf démonstration préliminaire (annexe §3.1)) qu'il existe $k_3$ tel que :

$\exists k_3, k_1 \leq k_3 < k' \wedge s_{k3} \models g_1 \wedge s_{k3} \not\models g_2 \wedge \forall k_5, k_3 < k_5 < k' \Rightarrow \neg(s_{k5} \models g_1 \wedge s_{k5} \not\models g_2)$

Il reste donc à montrer que:

$\forall k_4, k_3 \leq k_4 \leq k'\text{-}1 \Rightarrow (s_{k4} \models g \wedge s_{k4} \not\models g_2)$

Or, $\forall k'', k_1 \leq k'' < k' \Rightarrow (s_{k''} \models g \wedge s_{k''} \not\models g_2)$ et $k_1 \leq k_3$

donc $\forall k'', k_3 \leq k'' \leq k'\text{-}1 \Rightarrow (s_{k''} \models g \wedge s_{k''} \not\models g_2)$

En prenant $k = k'$, $k_4 = k''$, on a tous les éléments prouvant (1.1).

Nous avons alors conclu la démonstration de (2) => (1) et donc de (1) ≡ (2).

2) en $q \neq q_{racine}$
La démonstration ci-dessus n'utilisant pas le fait que $s \in EX(q_{racine})$, $k \geq 1$ et $k_1 \geq 1$, elle est valable en $q \neq q_{racine}$.

La traduction de *possible cont g entre $g_1$ et $g_2$* en MODAL LUSTRE, est donc correcte. Celle de *toujours exist g entre $g_1$ et $g_2$*, qui est sa duale, l'est donc aussi.

### 3.2.3 préservable cont g entre $g_1$ et $g_2$

$q \models$ préservable cont g entre $g_1$ et $g_2$
est équivalent à
$q \models$ some (Cont_Depuis (T(g) and not $T(g_2)$, $T(g_1)$ and not $T(g_2)$)
or IlExiste_Depuis($T(g_2)$, $T(g_1)$ and not $T(g_2)$)))

En effet, la démonstration est la même que pour la traduction de toujours cont g entre $g_1$ et $g_2$ en MODAL LUSTRE.
La séquence s exhibée dans chacun des cas, n'est plus quelconque, mais est celle qui rend valide la propriété hypothèse.

### 3.2.4 préservable exist g entre $g_1$ et $g_2$ strict

1) en $q = q_{racine}$

sémantique (définition intuitive):
$q_{racine} \models$ préservable exist g entre $g_1$ et $g_2$ strict ssi
$\exists s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$(s_k \models g_1 \wedge s_k \not\models g_2) \wedge (\exists k', k'>k \wedge s_{k'} \models g_2 \wedge \forall k'', k'>k''\geq k \Rightarrow s_{k''} \not\models g_2)$
$\Rightarrow \exists k'', k'>k''\geq k \wedge s_{k''} \models g$ (1)

traduction:
$q_{racine} \models$ some ( $g_2$ => pre[ Cont (not($g_1$ and not $g_2$)) or IlExiste_Depuis (g or $g_2$, $g_1$ and not $g_2$) ] )

Remarque: ci-dessus, les notations devraient être T(g), $T(g_1)$ et $T(g_2)$ au lieu de g, $g_1$, et $g_2$, puisqu'il s'agit de la traduction de la formule, et donc de ses sous-formules. Nous avons préféré

ne pas alourdir la notation, sachant que la démonstration se fait par induction (sur la longueur de la chaîne en nombre d'opérateurs), et que par hypothèse d'induction T(g), $T(g_1)$ et $T(g_2)$ sont des traductions correctes et peuvent donc, par un abus de langage, être considérées comme g, $g_1$ et $g_2$.

Par expansion des nœuds Cont et IlExiste_Depuis, et d'après la sémantique des opérateurs Lustre, on obtient la formule équivalente :

$\exists t \in EX(q_{racine}),\ \forall k_1 \in \mathbb{N},\ k_1 \geq 1 \Rightarrow$

$t_{k1} \models g_2 \Rightarrow$

$\forall k'_1,\ 1 \leq k'_1 \leq k_1\text{-}1 \Rightarrow \neg\ (t_{k'1} \models g_1 \wedge t_{k'1} \not\models g_2)$

$\vee\ [\ \exists k'_1,\ 1 \leq k'_1 \leq k_1\text{-}1 \wedge t_{k'1} \models g_1 \wedge t_{k'1} \not\models g_2$

$\wedge\ \forall k'''_1,\ k'_1 \leq k'''_1 \leq k_1\text{-}1 \Rightarrow t_{k'''1} \not\models g_1 \wedge t_{k'''1} \not\models g_2$

$\wedge\ \exists k''_1,\ k'_1 \leq k''_1 \leq k_1\text{-}1 \wedge (t_{k''1} \models g \vee t_{k''1} \models g_2)]$ (2)

**Montrons (1) => (2)**

Soit $s_{k1}$ un état quelconque de la séquence s tel que $k_1 \geq 1$ et $s_{k1} \models g_2$, montrons

$\forall k'_1,\ 1 \leq k'_1 \leq k_1\text{-}1 \Rightarrow \neg\ (s_{k'1} \models g_1 \wedge s_{k'1} \not\models g_2)$

$\vee\ [\ \exists k'_1,\ 1 \leq k'_1 \leq k_1\text{-}1 \wedge s_{k'1} \models g_1 \wedge s_{k'1} \not\models g_2$

$\wedge\ \forall k'''_1,\ k'_1 \leq k'''_1 \leq k_1\text{-}1 \Rightarrow s_{k'''1} \not\models g_1 \wedge s_{k'''1} \not\models g_2$

$\wedge\ \exists k''_1,\ k'_1 \leq k''_1 \leq k_1\text{-}1 \wedge (s_{k''1} \models g \vee s_{k''1} \models g_2)\ ]$

2 cas sont à envisager:

a) Soit $\forall k',\ 1 \leq k' \leq k_1\text{-}1 \Rightarrow \neg\ (s_{k'} \models g_1 \wedge s_{k'} \not\models g_2)$. CQFD

b) Soit $\exists k',\ 1 \leq k' \leq k_1\text{-}1 \wedge s_{k'} \models g_1 \wedge s_{k'} \not\models g_2$

auquel cas, d'après la démonstration préliminaire (annexe §3.1):

$\exists k'_1,\ 1 \leq k'_1 \leq k_1\text{-}1 \wedge s_{k'1} \models g_1 \wedge s_{k'1} \not\models g_2$

$\wedge\ \forall k'''_1,\ k'_1 \leq k'''_1 \leq k_1\text{-}1 \Rightarrow s_{k'''1} \models g_1 \wedge s_{k'''1} \not\models g_2$

Il reste donc à montrer que: $\exists k''_1,\ k'_1 \leq k''_1 \leq k_1\text{-}1 \wedge (s_{k''1} \models g \vee s_{k''1} \models g_2)$.

Pour cela, séparons l'étude en 2 cas:

b.a) $\exists k''_1,\ k'_1 \leq k''_1 \leq k_1\text{-}1 \wedge s_{k''1} \models g_2$

Et donc $\exists k''_1,\ k'_1 \leq k''_1 \leq k_1\text{-}1 \wedge (s_{k''1} \models g \vee s_{k''1} \models g_2)$. CQFD

b.b) $\forall k"_1, k'_1 \leq k"_1 \leq k_1\text{-}1 \Rightarrow s_{k"1} \not\models g_2$
alors d'après (1): $\exists k", k'>k" \geq k \wedge s_{k"} \models g$
ce qui est équivalent à $\exists k"_1, k'_1 \leq k"_1 \leq k_1\text{-}1 \wedge s_{k"1} \models g$
Et donc $\exists k"_1, k'_1 \leq k"_1 \leq k_1\text{-}1 \wedge (s_{k"1} \models g \vee s_{k"1} \models g_2)$. CQFD

**Montrons maintenant (2) => (1)**
Soit $t_k$ un état quelconque de la séquence d'exécution t tel que $k \geq 1$ et
$(t_k \models g_1 \wedge t_k \not\models g_2) \wedge (\exists k', k'>k \wedge t_{k'} \models g_2 \wedge \forall k", k'>k" \geq k \Rightarrow t_{k"} \not\models g_2)$
montrons $\exists k", k'>k" \geq k \wedge t_{k"} \models g$

$t_k \models g_1 \wedge t_k \not\models g_2 \Rightarrow \neg(\forall k'_1, 1 \leq k'_1 \leq k_1\text{-}1 \Rightarrow \neg (t_{k'1} \models g_1 \wedge t_{k'1} \not\models g_2))$
donc d'après (2):
$\exists k'_1, 1 \leq k'_1 \leq k'\text{-}1 \wedge t_{k'1} \models g_1$ and not $g_2$
$\wedge \forall k"'_1, k'_1 < k"'_1 \leq k'\text{-}1 \Rightarrow t_{k"'1} \not\models g_1 \wedge t_{k"'1} \not\models g_2$
$\wedge \exists k"_1, k'_1 \leq k"_1 \leq k'\text{-}1 \wedge (t_{k"1} \models g \vee t_{k"1} \models g_2)$
or, $k<k'$ et $t_k \models g_1$ et non $g_2$ impliquent que $k'_1$ est supérieur à k et donc $k"_1$ aussi.
et donc, puisque $\exists k"_1, k'_1 \leq k"_1 \leq k'\text{-}1 \wedge (t_{k"1} \models g \vee t_{k"1} \models g_2)$
on a $\exists k"_1, k \leq k"_1 \leq k'\text{-}1 \wedge (t_{k"1} \models g \vee t_{k"1} \models g_2)$
or par hypothèse: $\forall k", k \leq k"<k' \Rightarrow t_{k"} \not\models g_2$
donc pour $k"_1$ tel que $k \leq k"_1 \leq k'\text{-}1$, on a: $t_{k"1} \not\models g_2$
et donc $\exists k"_1, k \leq k"_1 \leq k'\text{-}1 \wedge t_{k"1} \models g$.

Nous avons donc montré que, pour la séquence t:
$\forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$(t_k \models g_1 \wedge t_k \not\models g_2) \wedge (\exists k', k'>k \wedge t_{k'} \models g_2 \wedge \forall k", k'>k" \geq k \Rightarrow t_{k"} \not\models g_2)$
$\Rightarrow \exists k", k'>k" \geq k \wedge t_{k"} \models g$
donc:
$\exists s \in EX(q_{racine}), s = t \wedge \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$[(s_k \models g_1 \wedge s_k \not\models g_2) \wedge (\exists k', k'>k \wedge s_{k'} \models g_2 \wedge \forall k", k'>k" \geq k \Rightarrow s_{k"} \not\models g_2)$
$\Rightarrow \exists k", k'>k" \geq k \wedge s_{k"} \models g]$ CQFD.

2) en $q \neq q_{racine}$
La démonstration ci-dessus n'utilisant pas le fait que $s \in EX(q_{racine})$, $k \geq 1$ et $k_1 \geq 1$, elle est valable en $q \neq q_{racine}$.

La traduction de *préservable exist g entre $g_1$ et $g_2$* en MODAL LUSTRE, est donc correcte.

## 3.3 PREUVE DE LA TRADUCTION DE L'OPÉRATEUR TOUJOURS EXIST G ENTRE G1 ET G2 STRICT EN INVARIANTS LUSTRE

1) en $q = q_{racine}$

$q_{racine} \models$ **toujours exist g entre $g_1$ et $g_2$ strict** ssi
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$( (s_k \models g_1 \wedge s_k \not\models g_2)$
$\wedge (\exists k', k'>k \wedge s_{k'} \models g_2)$
$\wedge (\forall k''', k'>k'''\geq k \Rightarrow s_{k'''} \not\models g_2) ) \quad \Rightarrow \quad (\exists k'', k'>k''\geq k \wedge s_{k''} \models g) \qquad (1)$

$q_{racine} \models$ **toujours(not $g_2$ or (true -> pre( Cont(not($g_1$ and not $g_2$)) or IlExiste_Depuis(g or $g_2$, $g_1$ and not $g_2$) )))** (2)
ssi $\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$s_k \models$ not $g_2$ or (true -> pre( Cont(not($g_1$ and not $g_2$)) or IlExiste_Depuis(g or $g_2$, $g_1$ and not $g_2$) ))
ce qui est équivalent, après expansion des nœuds Cont et IlExiste_Depuis et selon la sémantique des opérateurs not, or, ->, et and, à :
$\forall s \in EX(q_{racine}), \forall k \in \mathbb{N}, k \geq 1 \Rightarrow$
$s_k \not\models g_2$
$\vee \quad k \neq 1 \Rightarrow [\forall k', 1 \leq k' \leq k-1 \Rightarrow \neg (s_{k'} \models g_1 \wedge s_{k'} \not\models g_2)]$
$\vee [\exists k_1, \ 1 \leq k_1 \leq k-1 \ \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$
$\wedge \forall k'', k_1 < k'' \leq k-1 \Rightarrow \neg (s_{k''} \models g_1 \wedge s_{k''} \not\models g_2)$
$\wedge \exists k''', k_1 \leq k''' \leq k-1 \wedge s_{k'''} \models g \vee s_{k'''} \models g_2 \ ]$

**Montrons tout d'abord (2) => (1):**
Soit s quelconque $\in EX(q_{racine})$, et k quelconque $\in \mathbb{N}$ tels que $k \geq 1$ et
$s_k \models g_1 \wedge s_k \not\models g_2 \wedge (\exists k', k'>k \wedge s_{k'} \models g_2 \wedge (\forall k''', k'>k'''\geq k \Rightarrow s_{k'''} \not\models g_2))$ montrons, sous l'hypothèse (2), que: $\exists k'', k'>k''\geq k \wedge s_{k''} \models g$.

$k'>k \wedge s_{k'} \models g_2 \wedge k \geq 1 \Rightarrow s_{k'} \models g_2 \wedge k' \neq 1$

D'après (2), nous avons:

$\forall k'_1,\ 1 \le k'_1 \le k'\text{-}1 \Rightarrow \neg\ (s_{k'1} \models g_1 \wedge s_{k'1} \not\models g_2)$

$\vee$

$\exists k_1, 1 \le k_1 \le k'\text{-}1 \quad \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$

$\wedge\ \forall k''_1,\ k_1 < k''_1 \le k'\text{-}1 \Rightarrow \neg\ (s_{k''1} \models g_1 \wedge s_{k''1} \not\models g_2)$

$\wedge\ \exists k'''_1,\ k_1 \le k'''_1 \le k'\text{-}1 \wedge s_{k'''1} \models g \vee s_{k'''1} \models g_2$

or $s_k \models g_1$ et non $g_2$ , et $k \le k'\text{-}1$

donc $\neg(\forall k'_1,\ 1 \le k'_1 \le k'\text{-}1 \Rightarrow \neg\ (s_{k'1} \models g_1 \wedge s_{k'1} \not\models g_2))$

donc $\exists k_1, 1 \le k_1 \le k'\text{-}1 \quad \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$

$\wedge\ \forall k''_1,\ k_1 < k''_1 \le k'\text{-}1 \Rightarrow \neg\ (s_{k''1} \models g_1 \wedge s_{k''1} \not\models g_2)$

$\wedge\ \exists k'''_1,\ k_1 \le k'''_1 \le k'\text{-}1 \wedge s_{k'''1} \models g \vee s_{k'''1} \models g_2$

$s_{k1}$ étant l'état "le plus proche" à gauche de $s_{k'}$ vérifiant $g_1$ et ne vérifiant pas $g_2$, on a: $k \le k_1 < k'$

or $\forall k'''$, $k' > k''' \ge k \Rightarrow s_{k'''} \models$ non $g_2$

donc $\forall k'''$, $k' > k''' \ge k_1 \Rightarrow s_{k'''} \not\models g_2$

donc, puisque $k_1 \le k'''_1 \le k'\text{-}1$, on a $(s_{k'''1} \models g \vee s_{k'''1} \models g_2) \wedge s_{k'''1} \not\models g_2$,

ce qui est équivalent à $s_{k'''1} \models g$ .

Par conséquent,

$\exists k_1,\ k \le k_1 \le k'\text{-}1 \quad \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$

$\wedge\ \forall k''_1,\ k_1 < k''_1 \le k'\text{-}1 \Rightarrow \neg\ (s_{k''1} \models g_1 \wedge s_{k''1} \not\models g_2)$

$\wedge\ \exists k'''_1,\ k_1 \le k'''_1 \le k'\text{-}1 \wedge s_{k'''1} \models g$

donc $\exists k'' = k'''_1,\ k \le k'' < k' \wedge s_{k''} \models g$

Ceci conclut la première partie de la démonstration.

**Montrons maintenant (1) => (2):**

Soit s quelconque $\in EX(q_{racine})$, et k quelconque $\in \mathbb{N}$ tel que $k \ge 1$,

deux cas peuvent alors se présenter:

a) soit $s_k \not\models g_2$, auquel cas, il ne reste rien à démontrer.

b) soit $s_k \models g_2$, auquel cas, montrons alors:

$k \ne 1 \Rightarrow \quad [\forall k',\ 1 \le k' \le k\text{-}1 \Rightarrow \neg\ (s_{k'} \models g_1 \wedge s_{k'} \not\models g_2)]$

$\vee \quad [\exists k_1,\ 1 \le k_1 \le k\text{-}1 \quad \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$

$\wedge\ \forall k'',\ k_1 < k'' \le k\text{-}1 \Rightarrow \neg\ (s_{k''} \models g_1 \wedge s_{k''} \not\models g_2)$

$\wedge\ \exists k''',\ k_1 \le k''' \le k\text{-}1 \wedge s_{k'''} \models g \vee s_{k'''} \models g_2 \quad ]$

Prenons $k \neq 1$, quelconque tel que $s_k \models g_2$,
deux cas peuvent alors se présenter:

b.a) soit $\forall k', 1 \leq k' \leq k-1 \Rightarrow \neg (s_{k'} \models g_1 \wedge s_{k'} \not\models g_2)$, auquel cas, il ne reste rien à démontrer.

b.b) soit $\exists k', \quad 1 \leq k' \leq k-1 \wedge s_{k'} \models g_1 \wedge s_{k'} \not\models g_2$
auquel cas, nous avons montré (cf démonstration préliminaire (annexe §3.1)) qu'il existe $k_1$ tel que $s_{k1}$ est l'état "le plus à gauche" de $s_k$ vérifiant $g_1$ et ne vérifiant pas $g_2$, c'est-à-dire que
$\exists k_1, \quad 1 \leq k_1 \leq k-1 \wedge s_{k1} \models g_1 \wedge s_{k1} \not\models g_2 \wedge \forall k", k_1 < k" \leq k-1 \Rightarrow \neg (s_{k"} \models g_1 \wedge s_{k"} \not\models g_2)$
montrons alors: $\exists k"', k_1 \leq k"' \leq k-1 \wedge s_{k"'} \models g \vee s_{k"'} \models g_2$
Comme nous avons $k_1 < k \wedge s_k \models g_2$,
nous savons (cf démonstration préliminaire (annexe §3.1)) qu'il existe $k_2$ tel que $s_{k2}$ est l'état "le plus à droite" de $s_{k1}$ satisfaisant $g_2$, c'est-à-dire que:
$\exists k_2, k_1 < k_2 \leq k \wedge s_{k2} \models g_2 \wedge \forall k"_2, k_1 \leq k"_2 < k_2 \Rightarrow s_{k"} \not\models g_2$
D'après l'hypothèse (1), $s_{k1} \models g_1 \wedge s_{k1} \not\models g_2$
$\wedge \exists k_2, k_1 < k_2 \wedge s_{k2} \models g_2$
$\wedge \forall k"_2, k_1 \leq k"_2 < k_2 \Rightarrow s_{k"} \not\models g_2$
$\Rightarrow (\exists k"', k_1 \leq k"' < k_2 \wedge s_{k"'} \models g)$
et comme $k_2 \leq k$,
$\Rightarrow (\exists k"', k_1 \leq k"' < k \wedge s_{k"'} \models g)$
donc $(\exists k"', k_1 \leq k"' < k \wedge s_{k"'} \models g \vee s_{k"'} \models g_2)$

Ce qui conclut la deuxième partie de la démonstration.

2) en $q \neq q_{racine}$
La démonstration ci-dessus n'utilisant pas le fait que $s \in EX(q_{racine})$ et $k \geq 1$, elle est valable en $q \neq q_{racine}$.

La traduction de *toujours exist g entre $g_1$ et $g_2$* en INVARIANTS LUSTRE , est donc correcte.

# 4 Preuve de l'équivalence des propriétés *toujours(entre (g1, g2) => g)* et *toujours cont g entre g1 et g2.*

Par définition :

**toujours (entre (g1, g2) => g)**

= toujours ( not [Après (g1 and not g2) and Cont_Depuis (not g2, g1 and not g2)] or g)

Appelons cette formule toujours (A).

(A) se réécrit en :

[Cont (not (g1 and not g2)) or IlExiste_Depuis (g2, g1 and not g2)] or g

Par définition :

**toujours cont g entre g1 et g2**

= toujours (Cont_Depuis (g and not g2, g1 and not g2)
or IlExiste_Depuis (g2, g1 and not g2))

Appelons cette formule toujours (B).

Dans un désir de réécrire (B), montrons d'abord que :

Cont_Depuis (a and b, c) ≡ Cont_Depuis (a, c) and Cont_Depuis (b, c).

Ceci sera réalisé par induction sur la longueur de la séquence d'exécution à partir de $q_{racine}$.

*longueur 1 :*

Cont_Depuis (a, c) and Cont_Depuis (b, c) ≡ [not c or a] and [not c or b]

≡ [not c or (a and b)] ≡ Cont_Depuis (a and b, c)

*longueur n >1:*

Supposons que la propriété soit vraie pour une séquence de longueur inférieure ou égale à n-1, montrons la pour une séquence de longueur n :

Cont_Depuis (a, c) and Cont_Depuis (b, c)

≡ [Cont (not c) or (a and c) or (a and pre Cont_Depuis (a, c))]
and [Cont (not c) or (b and c) or (b and pre Cont_Depuis (b, c))]

≡ Cont (not c) or (a and b and c)
or (a and b and pre (Cont_Depuis (a, c) and Cont_Depuis (b, c)))

car X or (X and Y) ≡ X

≡ Cont (not c) or (a and b and c) or (a and b and pre Cont_Depuis (a and b, c))

par hypothèse d'induction (l'état satisfaisant pre (Cont_Depuis (a, c) and Cont_Depuis (b, c)) étant situé à une distance de longueur n-1 de l'état $q_{racine}$).

≡ Cont_Depuis (a and b, c)

A la suite de cette démonstration, (B) peut se réécrire en :
[Cont_Depuis (g, g1 and not g2) and Cont_Depuis (not g2, g1 and not g2)] or IlExiste_Depuis (g2, g1 and not g2)
c'est-à-dire par distribution du "or" sur le "and" :
[Cont_Depuis (g, g1 and not g2) or IlExiste_Depuis (g2, g1 and not g2)] and [Cont_Depuis (not g2, g1 and not g2) or IlExiste_Depuis (g2, g1 and not g2)]

Cont_Depuis et IlExite_Depuis étant duaux, Cont_Depuis (not g2, g1 and not g2) or IlExiste_Depuis (g2, g1 and not g2) est équivalent à true, et (B) se réécrit en :
Cont_Depuis (g, g1 and not g2) or IlExiste_Depuis (g2, g1 and not g2)

Ce qui, par définition de Cont_Depuis, est équivalent à :
[not (g1 or not g2) or g -> Cont (not (g1 and not g2)) or (g1 and not g2 and g)
or (g and pre Cont_Depuis (g, g1 and not g2)) ]
or IlExiste_Depuis (g2, g1 and not g2)

Sachant qu'au premier cycle, IlExiste_Depuis (g2, g1 and not g2) ≡ g2 and (g1 and not g2) ≡ false, on obtient pour (B) :
not (g1 or not g2) or g -> Cont (not (g1 and not g2)) or (g1 and not g2 and g)
or (g and pre Cont_Depuis (g, g1 and not g2))
or IlExiste_Depuis (g2, g1 and not g2)

Par ailleurs, d'après la définition de Cont et de IlExiste_Depuis au premier cycle, (A) est équivalent à :
not (g1 and not g2) or g -> Cont (not (g1 and not g2))
or IlExiste_Depuis (g2, g1 and not g2) or g

Donc (A) n'est pas équivalent à (B) ssi,
dans le cas :
not [Cont (not (g1 and not g2)) or IlExiste_Depuis (g2, g1 and not g2)]
l'expression : (g1 and not g2 and g) or (g and pre Cont_Depuis (g, g1 and not g2))
n'est pas équivalente à l'expression g.

c'est-à-dire ssi dans le cas :
Après (g1 and not g2) and Cont_Depuis (not g2, g1 and not g2),
on a : not [g ≡ g and ((g1 and not g2) or pre Cont_Depuis (g, g1 and not g2))]

(A) n'est donc pas équivalent à (B) ssi :
Après (g1 and not g2) and Cont_Depuis (not g2, g1 and not g2) and g and not ((g1 and not g2) or pre Cont_Depuis (g, g1 and not g2))
c'est-à-dire ssi :
Après (g1 and not g2) and Cont_Depuis (not g2, g1 and not g2) and g and not (g1 and not g2) and pre IlExiste_Depuis (not g, g1 and not g2).
Appelons (C) cette condition.

Montrons que (C) ne peut se produire dans un état appartenant à un arbre d'exécution modèle de toujours (A). Ainsi, (A) est équivalent à (B) dans tout état d'un modèle quelconque de toujours (A). Tout état d'un tel modèle vérifie (A), il vérifie alors aussi (B). Par conséquent, tout modèle de toujours (A) est modèle de toujours (B).
La démonstration va être réalisée par l'absurde :
Soit T un arbre quelconque modèle de toujours (A). Supposons qu'il existe une exécution s à partir de l'état racine de T, et un entier k quelconque tel que $k \geq 1$ et $s_k \models$ (C).
Par définition de IlExiste_Depuis, Après et Cont_Depuis, cela implique :
$\exists k',\ k > k' \geq 1 \wedge$
$s_{k'} \models$ Après (g1 and not g2) and Cont_Depuis (not g2, g1 and not g2) and not g (D)
Or $q_{racine} \models$ toujours (A), donc :
$\forall k'',\ k'' \geq 1 \Rightarrow s_{k''} \models$ (A)
c'est-à-dire :
$s_{k''} \models$ not [Après (g1 and not g2) and Cont_Depuis (not g2, g1 and not g2)] or g
Par conséquent, cette propriété est vraie pour k', mais ceci est en contradiction avec (D).
Par conséquent, l'hypothèse d'existence d'un état vérifiant (C) dans un arbre d'exécution modèle de toujours (A), est fausse.
Et ainsi, tout modèle de toujours (A) est modèle de toujours (B).

Montrons suivant le même principe que (C) ne peut se produire dans un état appartenant à un arbre d'exécution modèle de toujours (B). Ainsi, (A) étant équivalent à (B) dans tout état d'un modèle quelconque de toujours (B), et (B) étant vrai dans un tel état, (A) est vrai dans tout état de tout modèle de toujours (B). Par conséquent, tout modèle de toujours (B) est modèle de toujours (A), et ainsi toujours (A) et toujours (B) sont deux formules équivalentes.

La démonstration va être réalisée par l'absurde :
Soit T un arbre quelconque modèle de toujours (B). Supposons qu'il existe une exécution s à partir de l'état racine de T, et un entier k quelconque tel que $k \geq 1$ et $s_k \models$ (C).
Par définition de IlExiste_Depuis, Après et Cont_Depuis, cela implique :
$\exists k'$, $k > k' \geq 1 \wedge s_{k'} \models$ (D)

Or $q_{racine} \models$ toujours (B), donc :
$\forall k''$, $k'' \geq 1 \Rightarrow s_{k''} \models$ (B)
c'est-à-dire :
$s_{k''} \models$ Cont_Depuis (g and not g2, g1 and not g2) or IlExiste_Depuis (g2, g1 and not g2)
et par conséquent, $s_{k'}$ vérifie aussi cette propriété :
$s_{k'} \models$ Cont_Depuis (g and not g2, g1 and not g2) or IlExiste_Depuis (g2, g1 and not g2)
soit par distribution du and dans le Cont_Depuis, et par dualité du IlExiste_Depuis avec le Cont_Depuis:
$s_{k'} \models$ [Cont_Depuis (g, g1 and not g2) and Cont_Depuis (not g2, g1 and not g2)]
or not Cont_Depuis (not g2, g1 and not g2)
ssi : $s_{k'} \models$ Cont_Depuis (g, g1 and not g2) or not Cont_Depuis (not g2, g1 and not g2)
Ce qui implique par définition de Cont_Depuis :
$s_{k'} \models$ Cont (not (g1 and not g2)) or g or not Cont_Depuis (not g2, g1 and not g2)
Or $s_{k'} \models$ Après (g1 and not g2) and Cont_Depuis (not g2, g1 and not g2) and not g (D)

Mais :

| | | |
|---|---|---|
| Cont (not (g1 and not g2)) | est contradictoire avec | Après (g1 and not g2) |
| g | est contradictoire avec | not g |
| not Cont_Depuis (not g2, g1 and not g2) | est contradictoire avec | Cont_Depuis (not g2, g1 and not g2) |

Par conséquent, $s_{k'} \models$ Cont (not (g1 and not g2)) or g or not Cont_Depuis (not g2, g1 and not g2) est en contradiction avec (D).
L'hypothèse d'existence d'un état vérifiant (C) dans un arbre d'exécution modèle de toujours (B) est donc fausse.
Ainsi, tout modèle de toujours (B) est modèle de toujours (A), et toujours (A) est équivalente à toujours (B).

# 5 Preuve de la correction des traductions en CL

## 5.1 TOUJOURS CONT G ENTRE G1 ET G2

1) en $q \neq q_{racine}$

sémantique (définition concise):

$q \models$ toujours cont g entre g1 et g2 ssi

$\forall s \in EX(q), \forall k \in \mathbb{N}, \quad s_k \models g1$ et non g2 =>

$\forall k"_1, \ k"_1 \geq k \wedge s_{k"1} \models$ non g => $\exists k'_1, \ k"_1 \geq k'_1 \geq k \wedge s_{k'1} \models g2$ (A)

traduction:

$q \models$ al $(\neg init \wedge g1 \wedge \neg g2 \Rightarrow$ al $[\neg g2]$ $(g \vee g2))$ ssi

$\forall s \in EX(q), \forall k \in \mathbb{N}, s_k \neq q_{racine} \wedge s_k \models g1 \wedge s_k \not\models g2 \Rightarrow$

$\forall k', k' \geq k \Rightarrow ((\forall k", k' > k" \geq k \Rightarrow s_{k"} \not\models g2) \Rightarrow s_{k'} \models g \vee s_{k'} \models g2)$ (B)

$q \neq q_{racine} \Rightarrow \forall k, \ s_k \neq q_{racine}$ et

(A) $\equiv \forall k"_1, \ k"_1 \geq k \wedge \neg(\exists k'_1, k"_1 \geq k'_1 \geq k \wedge s_{k'1} \models g2) \Rightarrow \neg(s_{k"1} \models$ non g)

$\equiv \forall k"_1, \ k"_1 \geq k \wedge \forall k'_1, k"_1 \geq k'_1 \geq k \Rightarrow s_{k'1} \not\models g2 \Rightarrow s_{k"1} \models g$

$\equiv \forall k"_1, \ k"_1 \geq k \Rightarrow [(\forall k'_1, k"_1 \geq k'_1 \geq k \Rightarrow s_{k'1} \not\models g2) \Rightarrow s_{k"1} \models g]$

$\equiv \forall k"_1, \ k"_1 \geq k \Rightarrow [(\forall k'_1, k"_1 > k'_1 \geq k \Rightarrow s_{k'1} \not\models g2) \wedge s_{k"1} \not\models g2 \Rightarrow s_{k"1} \models g]$

$\equiv \forall k"_1, \ k"_1 \geq k \Rightarrow [(\forall k'_1, k"_1 > k'_1 \geq k \Rightarrow s_{k'1} \not\models g2) \Rightarrow \neg(s_{k"1} \not\models g2) \vee s_{k"1} \models g]$

$\equiv$ (B)

2) en $q_{racine}$

Dans la sémantique concise, on rajoute le fait : $k \geq 1$. Or dans la traduction en CL, $sk \neq$ qracine ssi $k \geq 1$. La démonstration ci-dessus s'applique donc encore.

## 5.2 TOUJOURS EXIST G ENTRE G1 ET G2

1) en $q \neq q_{racine}$

sémantique (définition concise):

$q \models$ toujours exist g entre g1 et g2 ssi

$\forall s \in EX(q), \forall k \in \mathbb{N}, \quad s_k \models g1$ et non g2 =>

$\exists k'_1, k'_1 \geq k \wedge (\forall k''_1, k'_1 \geq k''_1 \geq k \Rightarrow s_{k''1} \models$ non g2) $\wedge\ s_{k'1} \models g$ (A)

traduction:

$q \models$ al ($\neg$init $\wedge$ g1 $\wedge \neg$g2 => inev [$\neg$g2] (g $\wedge \neg$ g2)) ssi

$\forall s \in EX(q), \forall k \in \mathbb{N}, s_k \neq q_{racine} \wedge s_k \models g \wedge s_k \not\models g2$ =>

$\exists k'_2, k'_2 \geq k \wedge (\forall k''_2, k'_2 > k''_2 \geq k \Rightarrow s_{k''2} \not\models g2) \wedge s_{k'2} \models g \wedge s_{k'2} \models g2$ (B)

$q \neq q_{racine} \Rightarrow \forall k,\ s_k \neq q_{racine}$ et

(A) $\equiv \exists k'_1, k'_1 \geq k \wedge [(\forall k''_1, k'_1 > k''_1 \geq k \Rightarrow s_{k''1} \not\models g2) \wedge s_{k'1} \not\models g2 \wedge s_{k'1} \models g \equiv$ (B)

2) en $q_{racine}$

Dans la sémantique concise, on rajoute le fait : $k \geq 1$. Or dans la traduction en CL, $sk \neq qracine$ ssi $k \geq 1$. La démonstration ci-dessus s'applique donc encore.

# 6 Preuve de la correction des traductions en μ-CL

Afin de valider les traductions en μ-calcul arborescent de *inévitable cont g entre g1 et g2*, *inévitable exist g entre g1 et g2*, *préservable cont g entre g1 et g2* et *préservable exist g entre g1 et g2*, nous allons présenter ici la démarche qui nous a conduits à ces traductions: tout d'abord sa formulation générale, puis sa particularisation pour chacune des propriétés.

## 6.1 ETAPES DU PASSAGE D'UNE PROPRIÉTÉ CONTENANT UNE SOUS-PROPRIÉTÉ LINÉAIRE À UNE FORMULE DE M-CALCUL ARBORESCENT:

### 6.1.1 Définitions

On appelle **graphe d'automate** un quadruplet $(\Sigma, Q, Q_0, \rho)$ où
- $\Sigma$ est un alphabet non vide,
- Q est un ensemble fini d'états,
- $Q_0 \subseteq Q$ est l'ensemble des états initiaux
- $\rho : \Sigma x Q \rightarrow 2^Q$ est une fonction de transition

Un état $q \in Q$ est déterministe ssi $\forall \sigma \in \Sigma$, $\rho(\sigma, q)$ est un singleton. G est **déterministe** ssi $\forall q \in Q$, q est déterministe.

Soit $w = \sigma_0, \sigma_1, \sigma_2 \ldots$ où $\forall i \geq 0$, $\sigma_i \in \Sigma$. Un **calcul de G** sur w est une séquence infinie $s = s_0, s_1, s_2 \ldots$ où $\forall i \geq 0$, $s_i \in Q$ et $s_{i+1} = \rho(\sigma_i, s_i)$. On note inf(s) l'ensemble des états qui figurent infiniment souvent dans s.

Un **automate séquentiel de Rabin** [Rab,69] est une paire A=(G, E) où $G=(\Sigma, Q, Q_0, \rho)$ est un graphe d'automate, et E est un ensemble de **paires d'acceptation** (I, F) appartenant à $(2^Q)^2$. L'automate A accepte une séquence w ssi $\exists$s, un calcul de G sur w, et $\exists (I,F) \in E$ tel que, $inf(s) \cap I \neq \emptyset$ et $inf(s) \cap F = \emptyset$, c'est-à-dire : un état de I est visité infiniment souvent sur s tandis que tous ceux de F ne le sont qu'un nombre fini de fois.

### 6.1.2 Etapes

On associe aux automates de Rabin des systèmes d'équations à points fixes dont la résolution permet d'obtenir le langage reconnu par l'automate (de façon analogue à l'obtention d'une expression régulière à partir d'un automate d'états finis simple).

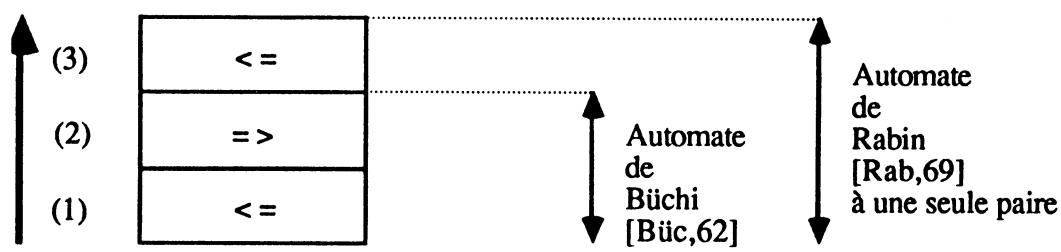
A chaque état X de l'automate, est associée une équation :

$X \leftrightarrow \bigvee ( \sigma_i \wedge OX_i)$ où

- $(\sigma_i, X_i)$ sont toutes les paires de l'automate telles que $\rho(\sigma_i, X) = X_i$
- $\leftrightarrow$ est le signe : <= si $\forall (I,F) \in E$, $X \notin I$
  => si $\exists (I,F) \in E$, $X \in I$

Cette équation signifie que, dans toute séquence modèle, la transition tirée à partir d'un état qui correspond à X est étiquetée par un des $\sigma_i$ apparaissant dans l'équation, et mène à un état qui correspond au $X_i$ associé à ce $\sigma_i$.

Dans le cas des automates de Rabin où E est réduit à une seule paire (I, F), le système d'équations associé à l'automate est de la forme :




où (1) est l'ensemble des équations associées aux états indifférents : {X/ X∉ I et X∉ F}
(2) est l'ensemble des équations associées aux états de répétition infinie : {X/ X∈ I}
(3) est l'ensemble des équations associées aux états de répétition finie : {X/ X∈ F}

La résolution de ces équations dans le sens indiqué par la flèche sur le schéma (1 puis 2 puis 3) fournit le langage des séquences acceptées par l'automate de Rabin [Par,81]. Ce langage sera exprimé dans notre cas par une formule de μ-calcul. l'opérateur <= correspond à un plus petit point fixe alors que l'opérateur => correspond à un plus grand point fixe.

Pour la vérification de nos propriétés, nous définissons des automates de Rabin A=(G, E) où G=(Σ, Q, $Q_0$, ρ) et E est **réduit à une paire** : (I,F), qui sont tels que :

- $\Sigma = 2^{\Pi}$ où $\Pi$ est un ensemble de variables propositionnelles (ensemble P des variables propositionnelles de la logique $\mu$-CL augmenté d'une variable pour chaque formule de $\mu$-CL).
- Il existe une relation d'interprétation associant à chaque état et à chaque variable de $\Pi$, une valeur de vérité.

Une transition étiquetée par $\sigma$, est tirée, dans un état X, lorsque la valeur de vérité des propositions apparaissant (respectivement, n'apparaissant pas) dans $\sigma$, est égale à "vrai" dans X (respectivement, à "faux").

Pour la vérification de nos propriétés, nous appellerons par abus de notation: g, g1, g2, les variables propositionnelles de $\Pi$, dont la valeur de vérité est identique à celle des formules du même nom.

Nous désirons reconnaître un **langage d'arbres** plutôt qu'un langage de séquences (puisque les modèles de nos formules sont des arbres). Aussi, nous définissons de nouvelles équations associées à un état :

L'équation :

$$X \leftrightarrow \bigvee ( \sigma_i \wedge \text{next } X_i)$$

où $(\sigma_i, X_i)$ sont toutes les paires de l'automate telles que $\rho(\sigma_i, X) = X_i$

signifie que, sur un arbre modèle, pour un état q de cet arbre correspondant à X, il existe $\sigma_j$ parmi les $\sigma_i$ apparaissant dans l'équation, tel que : il existe une transition partant de q, étiquetée par $\sigma_j$ et qui mène à un état correspondant à $X_j$.

L'équation :

$$X \leftrightarrow \bigvee ( \sigma_i \wedge \text{Succes } X_i)$$

où $(\sigma_i, X_i)$ sont toutes les paires de l'automate telles que $\rho(\sigma_i, X) = X_i$

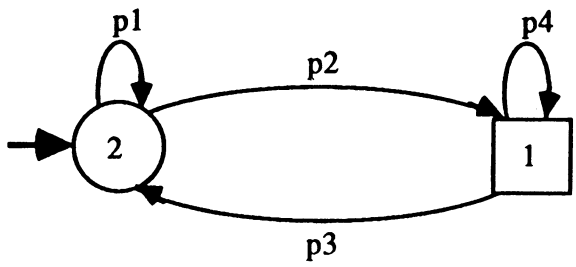
signifie que, sur un arbre modèle, pour un état q de cet arbre correspondant à X, il existe $\sigma_j$ parmi les $\sigma_i$ apparaissant dans l'équation, tel que : sur un arbre modèle, toute transition partant de X est étiquetée $\sigma_j$ et mène à $X_j$.

Soit un système linéaire S de Park, reconnaissant le langage de séquences F.
Le remplacement des symboles O par des symboles **next** dans S, permet d'obtenir un système reconnaissant le langage des arbres possédant au moins une séquence dans F.

Lorsque le système d'équations S est déterministe (ce qui est toujours notre cas), le remplacement des symboles O par des symboles **Succes** dans S, permet d'obtenir un système reconnaissant le langage des arbres possédant dont toutes les séquences sont dans F.

### 6.1.3 Exemple de passage d'un automate de Rabin à une formule de μ-calcul arborescent:

Automate de Rabin:




avec { ({1},{}) }

Système linéaire:

$Z = X_2$ (état initial)

$X_1 => (p_3 \wedge OX_1) \vee (p_4 \wedge OX_2)$

$X_2 <= (p_1 \wedge OX_2) \vee (p_2 \wedge OX_1)$

Système arborescent :

$Z = X_2$ (état initial)

$X_1 => (p_3 \wedge \text{Succes } X_1) \vee (p_4 \wedge \text{Succes } X_2)$

$X_2 <= (p_1 \wedge \text{Succes } X_2) \vee (p_2 \wedge \text{Succes } X_1)$

Résolution:

$X_2 = \mu x_2 ((p_1 \wedge \text{Succes } x_2) \vee (p_2 \wedge \text{Succes } x_1))$

$X_1 = \nu x_1 ( \quad (p_3 \wedge \text{Succes } x_1)$

$\quad \vee \quad (p_4 \wedge \text{Succes } \mu x_2 ((p_1 \wedge \text{Succes } x_2) \vee (p_2 \wedge \text{Succes } x_1)) ) \quad )$

$Z = \mu x_2 ( \quad (p_1 \wedge \text{Succes } x_2)$

$\quad \vee \quad (p_2 \wedge \text{Succes } \nu x_1 ( \quad (p_3 \wedge \text{Succes } x_1)$

$\quad \vee \quad (p_4 \wedge \text{Succes } \mu x_2 ( \quad (p_1 \wedge \text{Succes } x_2)$

$\quad \vee \quad (p_2 \wedge \text{Succes } x_1) \quad )) )) )$

## 6.2 INÉVITABLE CONT G ENTRE G1 ET G2

défini en un état q quelconque

Nous utilisons ici la sémantique intuitive de inévitable cont g entre g1 et g2, dont l'équivalence avec sa définition est prouvée dans l'annexe: "équivalence des définitions".

$q \models$ inévitable cont g entre g1 et g2 ssi

$\forall s \in EX(q), \exists k \in \mathbb{N}, s_k \models g1$ et non g2 $\wedge$

$[\forall k'', \quad (k \leq k'' \wedge s_{k''} \models_I \text{non } g) \quad => \quad (\exists k', k \leq k' \leq k'' \wedge s_{k'} \models_I g2)]$

Remarque:

Si $q = q_{racine}$, il faut adapter la définition et la preuve pour que $k \geq 1$.

Pour une séquence donnée, la **propriété linéaire**

$\exists k \in \mathbb{N}, s_k \models g1$ et non g2 $\wedge$

$[\forall k'', \quad (k'' \geq k \wedge s_{k''} \models \text{non } g) \quad => \quad (\exists k', k'' \geq k' > k \wedge s_{k'} \models g2)]$

est équivalente à la propriété :

$\exists k \in \mathbb{N}, s_k \models g1$ et non g2 $\wedge$

$[\forall k'', \quad (k'' \geq k \wedge \forall k', \ k'' \geq k' > k => s_{k'} \not\models g2) \quad => \quad (s_{k''} \models g)]$

ce qui est équivalent à :

$\exists k \in \mathbb{N}, s_k \models g1$ et non g2 $\wedge$

$[ \quad (\exists k', k' > k \wedge s_{k'} \models g2 \wedge \forall k'', k' > k'' \geq k => s_{k''} \models g \text{ et non } g2)$

$\vee \quad (\forall k', k' \geq k => s_{k'} \models g \text{ et non } g2) \quad ]$

ce qui est encore équivalent à :

$( \exists k \in \mathbb{N}, s_k \models g1$ et non g2 et g

$\wedge \exists k', k' > k \wedge s_{k'} \models g2 \wedge \forall k'', k' > k'' > k => s_{k''} \models g$ et non g2)

$\vee \quad ( \exists k \in \mathbb{N}, s_k \models g1$ et non g2 et g

$\wedge \forall k', k' > k => s_{k'} \models g$ et non g2)

Un **automate** reconnaissant cette propriété dans une séquence, reconnait à la fois, dans une séquence s, la sous-séquence finie $s_k \ldots s_{k'}$ telle que:

$s_k \models g1$ et non g2 et g

$\wedge \quad k' > k \wedge s_{k'} \models g2 \wedge \forall k'', k' > k'' > k => s_{k''} \models g$ et non g2

et la sous-séquence infinie débutant en $s_k$ telle que:

$s_k \models g1$ et non g2 et g

$\wedge \quad \forall k', k' > k => s_{k'} \models g$ et non g2

Nous pouvons alors définir l'automate de Rabin A = (S={1, 2, 3}, $S_0$={1}, ρ, E) avec la relation de transition ρ définie comme suit:

ρ(σ, 1)={1} ssi g1∉σ ou g2∈σ ou g∉σ

ρ(σ, 1)={2} ssi g1∈σ et g2∉σ et g∈σ

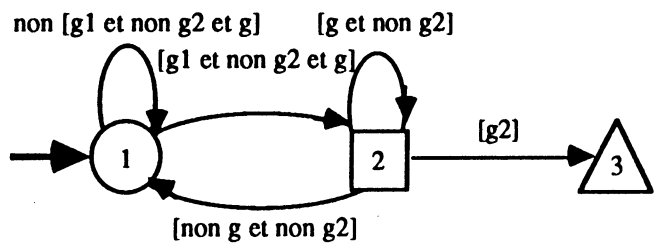
ρ(σ, 2)={1} ssi g∉σ et g2∉σ

ρ(σ, 2)={2} ssi g∈σ et g2∉σ

ρ(σ, 2)={3} ssi g2∈σ

et E = { ({2}, {1}) }

Nous schématiserons les automates, en regroupant sous une seule transition, toutes les transitions de l'automate réel qui partent d'un même état pour aboutir à un même état, et en étiquetant celles-ci par une formule synthétisant les étiquettes de ces transitions:




Remarques : on peut boucler à l'infini sur les états □
les états puits sont représentés par △

avec la précision E = { ({2}, {1}) }

Nous en déduisons le **système linéaire** suivant:

$X_1$ <= ((¬g1 ∨ g2 ∨ ¬g) ∧ O$X_1$) ∨ (g1 ∧ ¬g2 ∧ g ∧ O$X_2$)

$X_2$ => (g ∧ ¬g2 ∧ O$X_2$) ∨ g2 ∨ (¬g ∧ ¬g2 ∧ O$X_1$)

La propriété devant être vérifiée pour toute séquence d'exécution s∈EX(q), le **système arborescent** est le suivant :

$X_1$ <= ((¬g1 ∨ g2 ∨ ¬g) ∧ Succes$X_1$) ∨ (g1 ∧ ¬g2 ∧ g ∧ Succes $X_2$)

$X_2$ => (g ∧ ¬g2 ∧ Succes $X_2$) ∨ g2 ∨ (¬g ∧ ¬g2 ∧ Succes $X_1$)

Et nous pouvons en déduire la **formule de μ-calcul arborescent** :

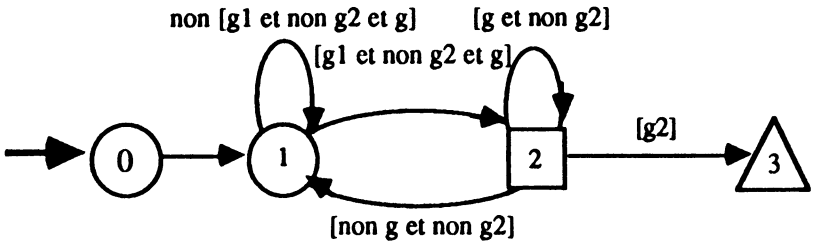
μ$x_1$[ ((¬g1∨g2∨¬g) ∧ Succes $x_1$)

∨ (g1 ∧ ¬g2 ∧ g ∧ Succes ν$x_2$( (g∧¬g2∧ Succes $x_2$)

∨ g2

∨ (¬g∧¬g2 ∧ Succes $x_1$) )) ]

*Remarque*: si q = $q_{racine}$, l'automate à la base de la formule de μ-calcul est le suivant (saut de l'état où k=0):




Il faut donc rajouter l'équation X <= $OX_1$ au système linéaire.

La formule de μ-calcul arborescent est alors:

μx [Succes $\mu x_1$[ ((¬g1∨g2∨¬g) ∧ Succes $x_1$)
∨ (g1 ∧ ¬g2 ∧ g ∧ Succes $\nu x_2$ ( (g∧¬g2∧ Succes $x_2$)
∨ g2
∨ (¬g∧¬g2 ∧ Succes $x_1$) )) ]]

## 6.3 INÉVITABLE EXIST G ENTRE G1 ET G2

en un état q quelconque, q≠$q_{racine}$.

Rappel:

q ⊨ inévitable exist g entre g1 et g2 ssi

∀s∈ EX(q), ∃k∈ **N**, $s_k$ ⊨ g1 et non g2 ∧

[∃k", k"≥k ∧ $s_{k"}$ ⊨ g ∧ (∀k', k"≥k'≥k =>$s_{k'}$ ⊨ non g2)]

Pour une séquence donnée, la **propriété linéaire**

∃k∈ **N**, $s_k$ ⊨ g1 et non g2 ∧ [∃k", k"≥k ∧ $s_{k"}$ ⊨ g ∧ (∀k', k"≥k'≥k =>$s_{k'}$ ⊨ non g2)]

est équivalente à la propriété :

∃k∈ **N**, $s_k$ ⊨ g1 et non g2 et g

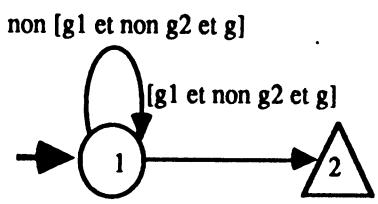
∨ ( ∃k∈ **N**, $s_k$ ⊨ g1 et non g2 et non g

∧ ∃k", k">k ∧ $s_{k"}$ ⊨ g et non $g_2$

∧ (∀k', k">k'>k =>$s_{k'}$ ⊨ non g et non g2) )

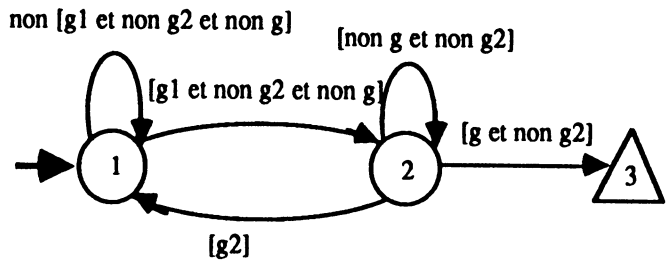
par séparation des cas k"=k et k">k, et utilisation du lemme.

Un **automate** reconnaissant cette propriété dans une séquence, est donc l'union de deux automates d'états finis : C et D, que nous schématiserons de la façon suivante:




avec E={ ({}, {1}) }

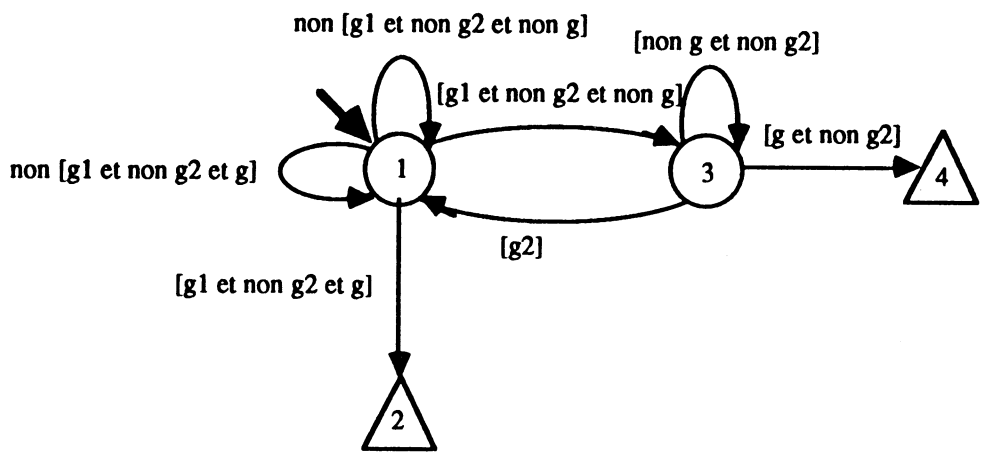
**Automate C**




avec E={ ({}, {1, 2}) }

**Automate D**

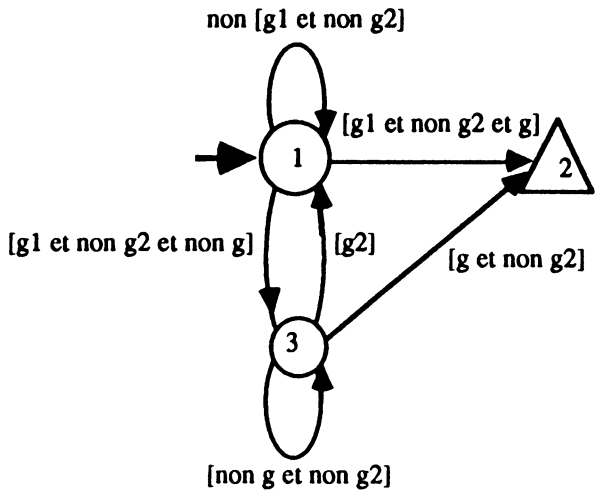
L'automate de l'**UNION: (C+D)**, est le suivant:




nous le rendons déterministe:

| Formules / Classes | g1,g2,g | g1,g2,¬g | g1,¬g2,g | g1,¬g2,¬g | ¬g1,g2,g | ¬g1,g2,¬g | ¬g1,¬g2,g | ¬g1,¬g2,¬g |
|---|---|---|---|---|---|---|---|---|
| {1} | {1} | {1} | {1, 2} Δ | {1, 3} | {1} | {1} | {1} | {1} |
| {1, 3} | {1} | {1} | {1, 2, 4} Δ | {1, 3} | {1} | {1} | {1, 4} Δ | {1, 3} |

puis nous le minimisons, afin d'obtenir l'automate final dont est déduite notre formule de μ-calcul arborescent :




Nous en déduisons le **système linéaire** suivant:

$X_1 <= ((\neg g1 \vee g2) \wedge OX_1) \vee (g1 \wedge \neg g2 \wedge g) \vee (g1 \wedge \neg g2 \wedge \neg g \wedge OX_3)$

$X_3 <= (g2 \wedge OX_1) \vee (g \wedge \neg g2) \vee (\neg g \wedge \neg g2 \wedge OX_3)$

La propriété devant être vérifiée pour toute séquence d'exécution $s \in EX(q)$, le **système arborescent** est le suivant :

$X_1 <= ((\neg g1 \vee g2) \wedge \text{Succes } X_1) \vee (g1 \wedge \neg g2 \wedge g) \vee (g1 \wedge \neg g2 \wedge \neg g \wedge \text{Succes } X_3)$

$X_3 <= (g2 \wedge \text{Succes } X_1) \vee (g \wedge \neg g2) \vee (\neg g \wedge \neg g2 \wedge \text{Succes } X_3)$

Et nous pouvons en déduire la **formule de μ-calcul arborescent** :

$\mu x_1[ \quad ((\neg g1 \vee g2) \wedge \text{Succes } x_1)$

$\vee \quad (g1 \wedge \neg g2 \wedge g)$

$\vee \quad (g1 \wedge \neg g2 \wedge \neg g \wedge \text{Succes } \mu x_3 ( \quad (g2 \wedge \text{Succes } x_1)$

$\vee \quad (g \wedge \neg g2)$

$\vee \quad (\neg g \wedge \neg g2 \wedge \text{Succes } x_3) \quad )) \quad ]$

La même adaptation que pour inévitable cont g entre $g_1$ et $g_2$ à lieu dans le cas $q = q_{racine}$, ce qui donne la formule de μ-calcul arborescent suivante:

$\mu x [ \text{ Succes } \mu x_1[ \quad ((\neg g1 \vee g2) \wedge \text{Succes } x_1)$

$\vee \quad (g1 \wedge \neg g2 \wedge g)$

$\vee \quad (g1 \wedge \neg g2 \wedge \neg g \wedge \text{Succes } \mu x_3 ( \quad (g2 \wedge \text{Succes } x_1)$

$\vee \quad (g \wedge \neg g2)$

$\vee \quad (\neg g \wedge \neg g2 \wedge \text{Succes } x_3) \quad )) \quad ]]$

## 6.4 PRÉSERVABLE CONT G ENTRE G1 ET G2

Rappel ($q \neq q_{racine}$):

$q \models$ préservable cont g entre g1 et g2 ssi

$\exists s \in EX(q), \forall k \in \mathbb{N}, s_k \models g1$ et non g2 =>

$[\forall k'', \quad (k \leq k'' \wedge s_{k''} \models_I \text{non } g) \quad => \quad (\exists k', k \leq k' \leq k'' \wedge s_{k'} \models_I g2)]$

Pour une séquence donnée, la **propriété linéaire**

$\forall k \in \mathbb{N}, s_k \models g1$ et non g2 =>

$[\forall k'', \quad (k \leq k'' \wedge s_{k''} \models_I \text{non } g) \quad => \quad (\exists k', k \leq k' \leq k'' \wedge s_{k'} \models_I g2)]$

est équivalente à la propriété (cf toujours cont g entre g1 et g2):

$[(\exists k', k' > k \wedge s_{k'} \models g2 \wedge \forall k'', k' > k'' \geq k => s_{k''} \models g \text{ et non } g2)$

$\vee \quad (\forall k', k' \geq k => s_{k'} \models g \text{ et non } g2)]$

qui est, elle-même, équivalente à la propriété :

$\forall k \in \mathbb{N}, s_k \models g1$ et non g2 **et g** =>

$[(\exists k', k' > k \wedge s_{k'} \models g2 \wedge \forall k'', k' > k'' > k => s_{k''} \models g \text{ et non } g2)$

$\vee \quad (\forall k', k' > k => s_{k'} \models g \text{ et non } g2)]$

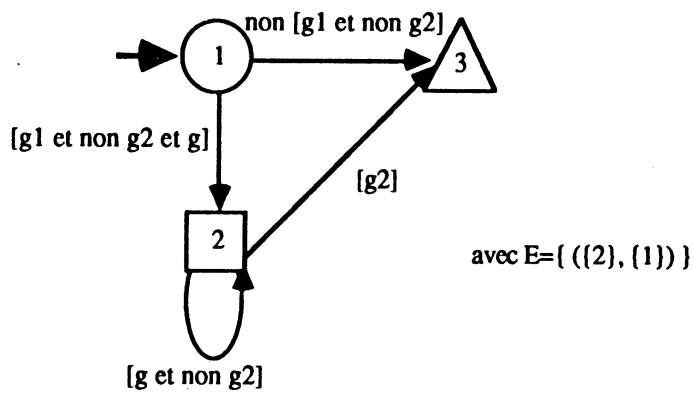
Une séquence s vérifiant cette propriété est telle que: tout suffixe de s débutant par un état $s_k$ vérifiant g1 et non g2 et g, doit vérifier pour être acceptable:

$(\exists k', k' > k \wedge s_{k'} \models g2 \wedge \forall k'', k' > k'' > k => s_{k''} \models g \text{ et non } g2)$

$\vee \quad (\forall k', k' > k => s_{k'} \models g \text{ et non } g2)$

et tout suffixe de s débutant par un état ne vérifiant pas g1 et non g2 et g, est acceptable.

L'**automate** reconnaissant un suffixe acceptable d'une séquence s est schématisé sous la forme suivante:




**Automate I**

Remarquons le point suivant:
Soit un suffixe acceptable d'une chaine s, débutant en $s_k$ tel que $s_k$ satisfait g1 et non g2 et g.
Deux cas peuvent alors se présenter:

soit $\exists k', k'>k \wedge s_{k'} \models g2 \wedge \forall k'', k'>k''\geq k => s_{k''} \models g$ *et non g2* , auquel cas : $\forall k_1$, $k'>k_1>k \wedge s_{k1} \models$ g1 et non g2 => $\forall k''$, $k'>k''\geq k_1 => s_{k''} \models$ g et non g2.

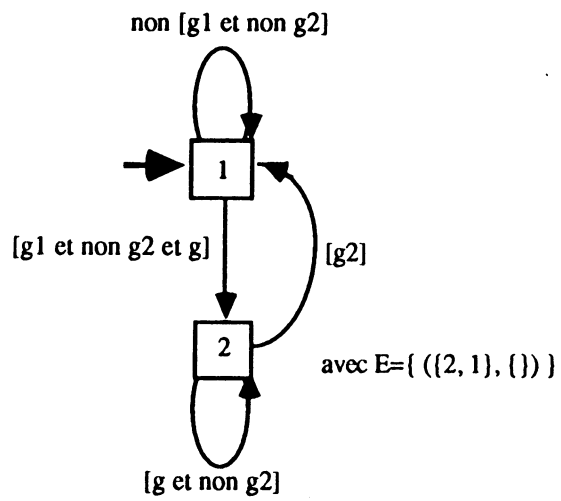
Et par conséquent, pour tout $k_1$ tel que $k'>k_1>k$, le suffixe de s débutant en $k_1$ est acceptable.

soit $\forall k', k'\geq k => s_{k'} \models g$ *et non g2* , auquel cas :

$\forall k_1$, $k'>k_1>k \wedge s_{k1} \models$ g1 et non g2 => $\forall k''$, $k'>k''\geq k_1 => s_{k''} \models$ g et non g2. Et par conséquent, pour tout $k_1$ tel que $k_1>k$, le suffixe de s débutant en $k_1$ est acceptable.

Nous voyons donc dans les deux cas, qu'il est inutile, lorsqu'on a reconnu qu'un suffixe est acceptable, de revenir en arrière sur les entrées pour reconnaitre que les suffixes suivants sont acceptables: il suffit de continuer à progresser sur les entrées.

Nous pouvons alors définir un **automate** de Rabin capable de reconnaître une séquence vérifiant la propriété, en repliant l'automate I sur lui-même d'après la remarque précédente; ce qui nous donne :




Nous en déduisons le **système linéaire** suivant:

$X_1 => ((\neg g1 \vee g2) \wedge OX_1) \vee (g1 \wedge \neg g2 \wedge g \wedge OX_2)$
$X_2 => (g2 \wedge OX_1) \vee (g \wedge \neg g2 \wedge OX_2)$

La propriété devant être vérifiée pour une séquence d'exécution $s \in EX(q)$, le **système arborescent** est le suivant :

$X_1 => ((\neg g1 \vee g2) \wedge$ next $X_1) \vee (g1 \wedge \neg g2 \wedge g \wedge$ next $X_2)$
$X_2 => (g2 \wedge$ next $X_1) \vee (g \wedge \neg g2 \wedge$ next $X_2)$

Et nous pouvons en déduire la **formule de μ-calcul arborescent** :

$\mu x_1[$ $((\neg g1 \vee g2) \wedge \text{next } x_1)$

$\vee$ $(g1 \wedge \neg g2 \wedge g \wedge \text{next } \mu x_2 ($ $(g2 \wedge \text{next } x_1)$

$\vee$ $(g \wedge \neg g2 \wedge \text{next } x_3)$ $))$ $]$

La même adaptation que pour la formule inévitable cont g entre $g_1$ et $g_2$ a lieu dans le cas où $q = q_{racine}$, ce qui donne la formule de μ-calcul arborescent suivante:

$\mu x$ [ next $\mu x_1[$ $((\neg g1 \vee g2) \wedge \text{next } x_1)$

$\vee$ $(g1 \wedge \neg g2 \wedge g \wedge \text{next } \mu x_2 ($ $(g2 \wedge \text{next } x_1)$

$\vee$ $(g \wedge \neg g2 \wedge \text{next } x_3)$ $))$ $]$

## 6.5 PRÉSERVABLE EXIST G ENTRE G1 ET G2

défini en un état q quelconque, $q \neq q_{racine}$.

$q \models$ préservable exist g entre g1 et g2 ssi

$\exists s \in EX(q), \forall k \in \mathbb{N}, s_k \models g1$ et non g2=>

$(\exists k', k' \geq k \wedge s_{k'} \models g \wedge (\forall k'', k'' \geq k' \geq k \Rightarrow s_{k''} \models \text{non } g2))$

Pour une séquence donnée, la **propriété linéaire**

$\forall k \in \mathbb{N}, s_k \models g1$ et non g2 =>

$(\exists k', k' \geq k \wedge s_{k'} \models g \wedge (\forall k'', k'' \geq k' \geq k \Rightarrow s_{k''} \models \text{non } g2))$

est équivalente à la propriété

$\forall k \in \mathbb{N}$,

$s_k \models$ non g1 ou g2 ou g

$\vee$

$s_k \models$ non g1 et non g2 et non g $\wedge$

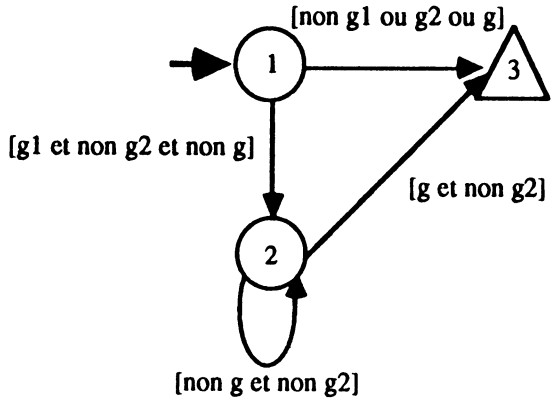
$(\exists k', k' > k \wedge s_{k'} \models g \wedge (\forall k'', k'' \geq k'' > k \Rightarrow s_{k''} \models \text{non } g2))$

Une séquence s vérifiant cette propriété est telle que: tout suffixe de s débutant par un état $s_k$ vérifiant g1 et non g2 et non g, doit vérifier pour être acceptable:

$(\exists k', k' > k \wedge s_{k'} \models g \wedge (\forall k'', k'' \geq k'' > k \Rightarrow s_{k''} \models \text{non } g2))$

et tout suffixe de s débutant par un état ne vérifiant pas g1 et non g2 et non g, est acceptable.

L'**automate** d'états finis reconnaissant un suffixe acceptable d'une séquence s est schématisé sous la forme suivante:



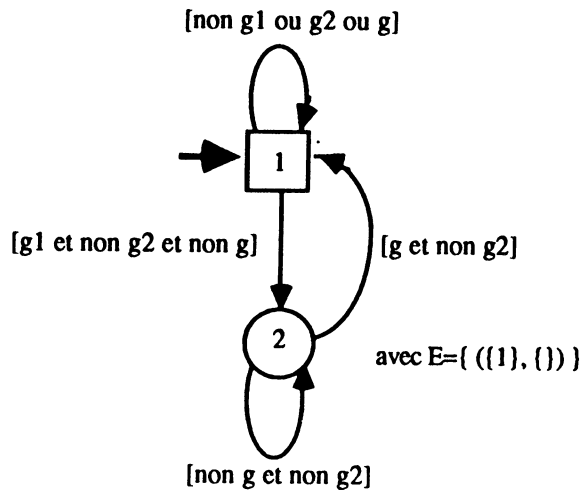

**Automate J**

Remarquons le point suivant:
Soit un suffixe acceptable d'une chaine s, débutant en $s_k$ tel que:
$s_k$ satisfait g1 et non g2 et non g.
Alors $\exists k', k'>k \wedge s_{k'} \models g \wedge (\forall k'', k'\geq k''>k \Rightarrow s_{k''} \models$ non g2)
Donc, pour tout $k_1$ tel que $k'\geq k_1>k \wedge s_{k1} \models$ g1 et non g2, on a $\exists k', k'\geq k_1 \wedge s_{k'} \models g \wedge (\forall k'', k'\geq k''\geq k_1 \Rightarrow s_{k''} \models$ non g2)
Et par conséquent, pour tout $k_1$ tel que $k'>k_1>k$, le suffixe de s débutant en $k_1$ est acceptable.

Nous voyons donc qu'il est inutile, lorsqu'on a reconnu qu'un suffixe est acceptable, de revenir en arrière sur les entrées pour reconnaitre que les suffixes suivants sont acceptables: il suffit de continuer à progresser sur les entrées.

Nous pouvons alors définir un **automate** de Rabin capable de reconnaître une séquence vérifiant la propriété, en repliant l'automate J sur lui-même d'après la remarque précédente;

ce qui nous donne:




Nous en déduisons le **système linéaire** suivant:

$X_1 => ((\neg g1 \vee g2 \vee g) \wedge OX_1) \vee (g1 \wedge \neg g2 \wedge \neg g \wedge OX_2)$
$X_2 <= ( g \wedge \neg g2 \wedge OX_1) \vee (\neg g \wedge \neg g2 \wedge OX_2)$

La propriété devant être vérifiée pour une séquence d'exécution $s \in EX(q)$, le **système arborescent** est le suivant :

$X_1 => ((\neg g1 \vee g2 \vee g) \wedge next\ X_1) \vee (g1 \wedge \neg g2 \wedge \neg g \wedge next\ X_2)$
$X_2 <= ( g \wedge \neg g2 \wedge next\ X_1) \vee (\neg g \wedge \neg g2 \wedge next\ X_2)$

Et nous pouvons en déduire la **formule de μ-calcul arborescent**:

$\nu x_1[ \quad ((\neg g1 \vee g2 \vee g) \wedge next\ x_1)$
$\quad \vee (g1 \wedge \neg g2 \wedge \neg g \wedge next\ \mu x_2 ( \quad (g \wedge \neg g2 \wedge next\ x_1)$
$\quad \vee (\neg g \wedge \neg g2 \wedge next\ x_3) \quad )) \quad ]$

La même adaptation que pour inévitable cont g entre $g_1$ et $g_2$ à lieu dans le cas $q = q_{racine}$, ce qui donne la formule de μ-calcul arborescent suivante:

$\mu x [ next\ \nu x_1[ \quad ((\neg g1 \vee g2 \vee g) \wedge next\ x_1)$
$\quad \vee (g1 \wedge \neg g2 \wedge \neg g \wedge next\ \mu x_2 ( \quad (g \wedge \neg g2 \wedge next\ x_1)$
$\quad \vee (\neg g \wedge \neg g2 \wedge next\ x_3) \quad )) \quad ]]$

# Références Bibliographiques


[Abr,79] K. Abrahamson. Modal Logic of Concurrent Non Deterministic Programs, *Semantics of Concurrent Computation*, Lecture Notes on Computer Science 70, p21-33, Berlin, Heidelberg, New York : Springer Verlag, 1979.

[AK,86] B. Auernheimer, R.A. Kemmerer. RT-ASLAN : A Specification Language for Real-Time Systems, *IEEE Transactions on Software Engineering, vol. SE-12*, n° 9, september 1986.

[AS,85] B. Alpern, F.B. Schneider. Defining Liveness, *Information Processing Letters 21*, p181-185, North-Holland, octobre 85.

[Bar,84] H.G. Barrow. Proving the Correctness of Digital Hardware Designs, VLSI Design, vol 5, n° 7, p64-77, juillet 1984.

[BFM,84] S.A. Babiker, R.A. Fleming, R.E. Milne. *A tutorial for LCS*, Rapport de recherche no 225.84.1, Standard Communication Laboratories, 1984.

[BH,81] A. Bernstein, P.K. Harter. Proving Real-Time Properties of Programs with Temporal Logic, *Proceedings of the eighth Symposium on Operating Systems Principles*. ACM, Californie, décembre 1981

[BK,85] J.A. Bergstra, J.W. Klop. Algebra of Communicating Processes with Abstraction, *TCS 37*, 1, 1985.

[BMP,83] M. Ben-Ari, Z. Manna, A. Pnueli. The Temporal Logic of Branching Time, *Acta Informatica 20*, p207-226, 1983.

[BMR,83] G. Berry, S. Moisan, J.P. Rigault. ESTEREL : Towards a synchronous and semantically sound high level language for real-time applications, *Proc. IEEE Real-Time Symposium*, 1983.

[BP,88] J-L. Bergerand, E.Pilaud. SAGA : a Software Development Environment for Dependability Automatic Control, *SAFECOMP 88, IFAC*, Fulda, Federal Republic of German, 1988.

[Büc,62] J.R. Büchi. On a Decision Method on a Restricted Second Order Arithmetic, *Proc. International Congress on Logic, Method and Philosophy Science 1960*, Standford University Press, p1-12, 1962.

[BW,87] M.R. Barbacci, J.M. Wing. Specifying functional and timing behaviour for real-time applications, PARLE 87, II, p124-140, vol. 259, *Lecture Notes on Computer Science*, Springer Verlag, Berlin 1987.

[CC,77] P.&R. Cousot. Abstract interpretation, a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *4th ACM Symposium on Principles of Programming Languages*, Los Angeles, janvier 77.

[CE,81] E.M. Clarke, E.A. Emerson. Design & Synthesis of synchronisation skeletons using branching time temporal logic, *Logics of Programs*, LNCS 131, p 52-71, 1981.

[CPHP,87] P. Caspi, D. Pilaud, N. Halbwachs, J.A. Plaice. LUSTRE : A declarative language for real-time programming, *14th Annual ACM Symposium on Principles of Programming Languages*, Munich, West Germany, janvier 1987.

[CSP,87] P. Couronné, J.B. Saint, J.A. Plaice. *The LUSTRE-ESTEREL portable format*, Document Technique, Non publié, Juin 1987.

[EH,85] E.A. Emerson, J.Y. Halpern. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time, *Journal of Computer and System Sciences*, vol 30, p1-24, 1985

[EH,86] E.A. Emerson, J.Y. Halpern. "Sometimes" and "Not Never" Revisited : On Branching versus Linear Time Temporal Logic, *ACM Journal*, vol. 33 (1), p151-178, 1986.

[EL,85] E.A. Emerson, C-L. Lei. Modalities for Model Checking : Branching Time Logic strikes back, *Proc.12th ACM Symposioum on POPL*, p84-86, New Orleans, Louisiana, Janvier 1985.

[EL,86] E.A. Emerson, C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus, *First Symposium on Logic in Computer Science*, p267-278, 1986.

[EMSS,89] E.A. Emerson, A.K. Mok, A.P. Sistla, J. Srinivasan. Quantitative Temporal Reasoning, *Worshop on Automatic Verification of Finite State Systems, Grenoble, juin 1989*, à paraître dans Lecture Notes on Computer Science, Springer Verlag.

[Est,85] Estelle : A Formal Description Technique Based on an Extended Transition Model, ISO, 1985. ISO/TC97/SC21.

[Fer,88] J-C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*, Thèse, Université Joseph Fourier, Grenoble, France, mai 1988.

[GC,89] A-C. Glory, P. Caspi. Préétude de la traduction de SAGA en LUSTRE, rapport intermédiaire du contrat APC 1988, Avril 1989.

[GPSS,80] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi. On the Temporal Analysis of Fairness. *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, janvier 1980.

[GS,86] S. Graf, J. Sifakis. A logic for the description of non deterministic programs and their properties, *Information and Control (68)*, 1986.

[GSV,88] S. Graf, J. Sifakis, J. Voiron. *Protocol Validation Methodoly*, Rapport du projet ESPRIT Delta4, Janvier 1988.

[Hoa,69] C.A.R. Hoare. An axiomatic basis for computer programming, *Communication of the ACM*, 12(10), 1969.

[HLP,86] N. Halbwachs, A. Lonchampt, D. Pilaud. Describing and designing circuits by means of a synchronous declarative language, *IFIP Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, 1986.

[HPP,89] N. Halbwachs, D. Pilaud, J.A. Plaice. *Generating Efficient Code From Data-Flow Programs*, Rapport de recherche, Projet Spectre L-8, LGI-IMAG, 1989.

[JJ,89] C. Jard, T. Jeron. On-Line Model-Checking for Finite Linear Temporal Logic Specifications, *Worshop on Automatic Verification of Finite State Systems, Grenoble, juin 1989*, à paraître dans Lecture Notes on Computer Science, Springer Verlag.

[JM,86] F. Jahanian, A.K. Mok. Safety Analysis of timing properties in real time systems, *IEEE Transactions on Software Engineering* SE-12,9, p280-904, Septembre 1986.

[Ka,74] G. Kahn. The Semantics of a Simple Language for Parallel Programming, *Information Processing 74*, North-Holland Publishing Compagny, 1974.

[KdR,85] R. Koymans, W.P. de Roever. Exemples of a real-time temporal logic specification, *The Analysis of Concurrent Systems - Cambridge, September 1983, Proceedings*, Lecture Notes on Computer Science, vol 207, p231-251, Springer Verlag, Berlin, 1985.

[Koz,83] D. Kozen. Results on the Propositional μ-calculus, *TCS* 27, 1983.

[Lam,84] L. Lamport. Basic concepts, *Advanced Course on Distributed Systems-Methods and Tools for Specification*, Lecture Notes on Computer Science, vol. 190 Springer Verlag, Berlin-Heidelberg, 1984.

[LBBG,85] P. LeGuernic, A. Benveniste, P. Bournai, T. Gautier. *SIGNAL : A data flow oriented language for signal processing*, Rapport de recherche n° 246, IRISA, Rennes, France, Janvier 1985.

[Lev,86] N.G. Leveson. Software Safety : Why, What, and How, *Computing Surveys*, Vol.18, n°2, juin 1986.

[LH,86] N.G. Leveson, P.R. Harvey. Analysing Software Safety, *IEEE Transactions on Software Engineering*, vol SE-9, n°5, septembre 1983.

[LPZ,85] O. Lichtenstein, A. Pnueli, L. Zuck. The Glory of The Past, *Proc. Workshop on Logics of Programs*, p196-218, vol.193, LNCS, Springer Verlag, Berlin, juin 1985.

[LS,86] N.G. Leveson, J.L. Stolsy. Safety Analysis using Petri nets, *IEEE Transactions on Software Engineering*, SE-12,9, p860-904, 1986.

[MC,81] J. Misra, K.M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, vol SE-7, n°4, juillet 1981.

[Mil,80] R. Milner. *A Calculus for Communicating Systems*, LNCS 92, 1980.

[Mil,83] R. Milner. Calculi for synchrony and asynchrony, *TCS 25(3)*, juillet 1983.

[Mil,84] R. Milner. *The Standard ML Core Language,* Rapport de recherche CSR_168_84, Université d'Edimbourg, Octobre 1984.

[MP,82] Z. Manna, A. Pnueli. Verification of concurrent programs : a temporal proof system, *Proc.4th School on Advanced Programming*, Amsterdam, Hollande, juin 1982.

[Par,81] D.M.R. Park. Concurrency and Automata on Infinite Sequences, *Proc. 5th Conference on Theoretical Computer Science*, Lecture Notes on Computer Science, vol. 104, p167-183, Springer Verlag, 1981.

[PH,88] D.Pilaud, N.Halbwachs. From a synchronuous declarative language to a temporal logic dealing with multiform time, *Proc. Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Warwick, septembre 1988.

[Pla,88] J.A. Plaice. *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone*, Thèse, Institut National Polytechnique de Grenoble, Grenoble, France, mai 1988.

[Pnu,86] A. Pnueli. *Application of temporal logic to the specification and verification of reactive systems: a survey of the current trends*, LNCS vol.224 p510-584.

[QS,82] J.P. Queille, J. Sifakis. Specification & Verification of Concurrent Systems in CESAR. *International Symposium on Programming*, Lecture Notes on Computer Science 137, p 337-351, Berlin, Heidelberg, New York : Spinger Verlag 1982.

[QS,83] J.P. Queille, J. Sifakis. Fairness & Related Properties in Transition Systems : A Temporal Logic to Deal with Fairness, *Acta Informatica 19*, p 195-200, Springer Verlag 1983.

[Rat,88] C. Ratel. *Etude de la conformité d'un programme LUSTRE et de ses spécifications en logique temporelle arborescente*, D.E.A. d'informatique, Institut National Polytechnique de Grenoble, Grenoble, France, juin 1988.

[Rab,69] M.O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees, *Transactions of American Mathematical Society*, vol. 141, p1-35, 1969.

[Ro,88] C. Rodriguez. *Spécification et validation de systèmes en XESAR*, Thèse, Institut National Polytechnique de Grenoble, Grenoble, France, mai 1988.

[RRSV,87a] J-L. Richier, C. Rodriguez, J. Sifakis, J. Voiron. Verification in XESAR of the sliding window protocol, *Proc. of the 7th international symposium on Protocol Specification, Testing and Verification*, Zurich, 5-8 Mai, 1987.

[RRSV,87b] J-L. Richier, C. Rodriguez, J. Sifakis, J. Voiron. *XESAR : A Tool for Protocol Validation. User's Guide*, LGI-IMAG, 1987.

[Rs,77] D.T. Ross. Structured Analysis (SA) : A Language for Communicating Ideas, *IEEE Transactions on Software Engineering*, vol. SE-3, n°1, Janvier 1977.

[RU,71] N. Rescher, A.Urquhart. *Temporal Logic*, Springer Verlag, 1971.

[WA, 85] W.W. Wadge, E.A. Ashcroft. *LUCID : The data-flow programming language*, Academic Press U.K., 1985.

[ZRE,85] J.Zwiers, W.P. de Roever, P. van Emde Boas. Compositionality and Concurrent Networks : Soudness and Completeness of a Proof System, *Proc. of International Colloqium on Automata Languages and Programming'85*, p509-519, Lecture Notes on Computer Science, Springer Verlag, Berlin, 1985.

# Table des matières

# Annexes

# VERIFICATION DE PROPRIETES DE PROGRAMMES FLOTS DE DONNEES SYNCHRONES

**Résumé :**

Dans le cadre de cette thèse, nous nous intéressons à la vérification de systèmes réactifs critiques et temps réel développés à l'aide de langages flots de données synchrones. Plus particulièrement nous avons considéré les propriétés de sûreté pour les applications réalisées dans un des deux langages, SAGA produit de Merlin Gerin/SES, ou LUSTRE créé au LGI. La méthode de vérification, pour laquelle un prototype a été réalisé, est l'évaluation de propriétés sur un modèle des programmes.

Un langage de spécification adapté au contexte des systèmes réactifs temps réel, avec sa sémantique formelle, est défini; ce langage comprend plusieurs opérateurs temporels. Le désir d'automatiser la vérification a nécessité la définition de la sémantique formelle de SAGA. Plusieurs modèles pour les programmes ont alors été étudiés: les arbres des exécutions comme base d'expression commune des sémantiques, les graphes d'états et automates de contrôle pour la mise en œuvre de la vérification.

L'utilisation de moyens existants de vérification, fondée sur l'évaluation de propriétés sur un modèle des programmes, a été étudiée et évaluée. Ces moyens sont relatifs à des logiques temporelles arborescentes et des mu-calculs propositionnels.

Une nouvelle approche pour la spécification et la vérification de propriétés de sûreté, mettant en œuvre les caractéristiques du langage LUSTRE, est développée. Elle s'appuie sur l'utilisation de LUSTRE lui-même comme langage de spécification et présente les avantages suivants: formalisme commun pour la programmation et la spécification, utilisation du compilateur pour la vérification, possibilité de preuves modulaires.

**Mots clefs :**

systèmes temps réel, spécification, vérification, flots de données, synchronisme, logique temporelle