Test de logiciels synchrones spécifiés en Lustre

Ioannis Parissis

THESE

présentée par

Ioannis PARISSIS

pour obtenir le titre de

DOCTEUR

de

L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(*arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992*)

Spécialité : INFORMATIQUE

Test de logiciels synchrones spécifiés en LUSTRE

Thèse soutenue le 30 Septembre 1996

Composition du Jury :

Président	**J. Voiron**
Rapporteurs	**G. Bernot** **C. Jard**
Examinateurs	**N. Halbwachs** **M. Müllerburg** **F. Ouabdesselam**

Thèse préparée au sein du
Laboratoire Logiciels, Systèmes et Réseaux (LSR) - Institut IMAG

A Ariane et son sourire éternel,
ma source inépuisable d'énergie,
d'affection et d'amour.

A Marie-Odile

A mes parents

Remerciements

Cette thèse a débuté au sein du Laboratoire de Génie Informatique et s'est poursuivie au sein du Laboratoire Logiciels, Systèmes et Réseaux de l'Institut IMAG. Je tiens à exprimer ma reconnaissance à leurs directeurs respectifs, Yves Chiaramella et Paul Jacquet, pour m'avoir accueilli et m'avoir fourni des conditions de travail exceptionnelles.

Je tiens à exprimer mes plus sincères remerciements à :

Gilles Bernot, Professeur à l'Université d'Evry - Val d'Essonne, et Claude Jard, Chargé de Recherches au CNRS, pour avoir accepté de juger ce travail en tant que rapporteurs;

Monika Müllerburg, Chercheur au GMD (Sankt Augustin, Allemagne), Nicolas Halbwachs, Directeur de Recherches au CNRS et Jacques Voiron, Professeur à l'Université Joseph Fourier, pour m'avoir fait l'honneur de participer au jury de cette thèse;

Farid Ouabdesselam, Professeur à l'Université Joseph Fourier, pour m'avoir dirigé, conseillé, guidé (et supporté!) pendant ces trois années, pour m'avoir proposé de me lancer dans cette aventure exceptionnelle et pour avoir fait preuve de tant de patience, de confiance, d'amitié et de bienveillance à mon égard;

Pascal Raymond, Chargé de Recherches au CNRS, pour ses conseils précieux et pour les outils de compilation de LUSTRE qu'il m'a fournis, sans lesquels une grande partie de ce travail n'aurait pas pu être réalisée;

Jean-Luc Richier, Chargé de Recherches au CNRS, pour m'avoir permis par sa lecture attentive de mon manuscrit d'en améliorer la qualité;

Chantal Robach, Chargé de Recherches au CNRS, et Paul Amblard, Maître de Conférences à l'Université Joseph Fourier, pour l'intérêt qu'ils ont porté à mon travail;

Anne Garcia, Florence Georges, Frédéric Gloppe, Cyril Perreau et Fabrice Vallet, pour le sérieux de leur travail de stages qui a contribué à la création de l'outil de test;

François "Bob" Challier, pour avoir été disponible chaque fois que nos machines étaient souffrantes;

Liliane Di-Giacomo, Martine Pernice, Solange Roche et Monique Boussey qui ont su transformer toute tâche administrative en un plaisir et pour avoir apporté quelques notes de gaieté à notre monde trop sérieux.

Cette thèse n'aurait pas pu avoir lieu si pendant mes années d'études supérieures précédant les études doctorales je n'avais été boursier de la fondation "Alexander S. Onassis". Je suis reconnaissant à son fondateur et aux membres de son Conseil d'Administration de leur générosité et leur confiance.

Enfin, un grand merci à mes parents, dont les pensées affectueuses m'ont atteint malgré la distance qui nous sépare et à ma petite famille, Marie-Odile et Ariane, pour avoir toujours soutenu "papa", surtout pendant la période éprouvante de rédaction.

Introduction

SÛRETÉ DE FONCTIONNEMENT, VÉRIFICATION ET VALIDATION

L'utilisation de logiciels dans des domaines à la fois variés et sensibles nécessite une confiance accrue dans les services qu'ils réalisent. La propriété qui permet aux utilisateurs d'un logiciel d'avoir cette confiance est sa *sûreté de fonctionnement* [Lap92]. La sûreté de fonctionnement d'un logiciel est influencée par toutes les activités du cycle de développement mais elle est plus particulièrement concernée par deux d'entre elles : la *vérification* et la *validation* [WF89]. Ces deux activités, qui visent précisément à contrôler et à améliorer la sûreté de fonctionnement du logiciel, ont un rôle majeur dans le cycle de vie de ce dernier, aussi bien par l'importance de leur fonction que par leur coût élevé.

Boehm [Boe81][1] définit l'activité de vérification comme le moyen d'établir la correspondance entre un produit logiciel et sa spécification, et la validation comme le moyen d'assurer que le logiciel accomplit bien la fonction pour laquelle il a été conçu. On dit souvent, de manière moins formelle, que la validation vise à s'assurer que l'on a réalisé "le bon logiciel" tandis que la vérification décide si on l'a "bien réalisé".

En réalité, par vérification et validation on désigne un ensemble de techniques portant sur tout le cycle de développement du logiciel et sur tous ses produits intermédiaires. Ainsi, les revues et les inspections de documents ainsi que le test font partie de ces techniques.

Les activités de validation et de vérification se déroulent bien souvent conjointement. Par exemple, l'examen d'un document a comme but, d'une part, d'établir la conformité de ce dernier aux normes de rédaction en vigueur ainsi que l'adéquation de

[1] Une définition similaire est donnée par l'AFNOR et l'IEEE.

la solution qu'il propose aux problèmes exposés dans les documents en amont du projet. Il s'agit là d'une activité de vérification. D'autre part, l'examen du document doit porter sur la pertinence de son contenu par rapport aux besoins exprimés mais non consignés et, plus généralement, implicites au début du développement. Il est en effet possible que le document ne réponde pas aux attentes initiales malgré sa conformité aux normes et sa cohérence avec les documents précédents (soit parce que les normes sont mal adaptées, soit parce que les documents précédents sont imprécis). L'activité opérée ici relève clairement de la validation puisqu'au delà de la conformité aux règles étudiée lors de la vérification ("bien fait"), on s'intéresse à la conformité aux besoins ("le bon logiciel").

Ces mêmes observations s'appliquent à l'examen d'un programme ou module pour lequel on dispose par ailleurs d'une spécification. Il est, en effet, important de s'assurer à la fois que les résultats de l'exécution du programme sont bien conformes à ceux décrits par la spécification mais aussi que cette exécution répond aux besoins initiaux (souvent informels et implicites) qui peuvent ne pas être reflétés exactement par la spécification, éventuellement incomplète, du programme.

La sûreté de fonctionnement est d'une importance capitale pour une catégorie particulière de logiciels : les logiciels critiques. Un logiciel est qualifié de *critique* si une anomalie dans son fonctionnement peut avoir des conséquences beaucoup plus importantes que le bénéfice procuré par le service qu'il assure en absence d'anomalies [Lap95]. Les logiciels de pilotage d'avion ou de train ainsi que ceux assurant le contrôle de réacteurs nucléaires en sont des exemples typiques, puisque leur défaillance peut causer des pertes de vies humaines ainsi que des dégâts matériels importants. Ces logiciels sont généralement en interaction permanente avec leur environnement et sont, de ce fait, appelés *réactifs*. Ils sont en plus qualifiés de *temps-réel* quand ils doivent satisfaire des contraintes temporelles imposées par cet environnement [BB91] (notons, cependant, que le terme "réactif" est souvent utilisé à la place de "réactif temps-réel" [HCRP91]). Ainsi, la correction d'un logiciel réactif temps-réel se définit aussi bien par rapport aux résultats qu'il fournit (*correction fonctionnelle*) que par rapport aux délais dans lesquels il les fournit (*correction temporelle*).

Dans le cadre du développement de logiciels critiques, les activités de vérification et de validation requièrent une rigueur et une efficacité exceptionnelle, aussi bien sur les aspects fonctionnels que temporels. Pour les logiciels réactifs temps-réel, divers travaux (par exemple [AG93], [JM86], [RLO92]) ont mené à des approches permettant la mise en œuvre de techniques de preuve ou de test sophistiquées. Ces techniques reposent généralement sur une description formelle du comportement du logiciel (autre, bien entendu, que son implantation).

VÉRIFICATION ET VALIDATION DES LOGICIELS SYNCHRONES

Nous nous intéressons dans cette thèse à une catégorie particulière de logiciels réactifs, les logiciels *synchrones*. Un logiciel est qualifié de synchrone si sa réaction à ses

entrées est instantanée ou, d'une manière plus pratique, s'il est assez rapide pour prendre en compte toute variation significative de son environnement. Cette propriété représente l'*hypothèse de synchronisme*. L'approche synchrone [BB91] a connu un grand essor depuis une dizaine d'années, en particulier grâce à l'apparition de langages qui lui sont propres, tels que LUSTRE [HCRP91], ESTEREL [BDS91] et SIGNAL [LGLL91]. Ces langages ont été conçus dans le but de rendre la programmation des logiciels réactifs simple. Ils disposent tous de compilateurs qui, grâce à l'hypothèse de synchronisme, produisent un code très efficace du point de vue du temps d'exécution, souvent sous forme d'automate d'états finis. Par ailleurs, leur sémantique étant définie de manière très rigoureuse, ils peuvent servir de support à la spécification formelle ainsi qu'à la preuve de programmes [HPOG89] [JPVO95].

Le travail présenté dans ce document porte plus particulièrement sur le langage synchrone LUSTRE. Conçu au Laboratoire de Génie Informatique de l'IMAG entre 1986 et 1987 [Ber86] [CHPP87], LUSTRE a été l'objet de plusieurs travaux de recherche portant aussi bien sur sa compilation que sur sa vérification (voir par exemple [Glo89] [Ray91] [Rat92]). Poursuivant la voie ouverte par [PH88] ces travaux ont mis en évidence, entre autres, que LUSTRE peut servir à la fois de langage de programmation et de logique temporelle du passé. Ainsi, LUSTRE est utilisable pour la description de certaines propriétés que doit satisfaire le logiciel, dites *de sûreté*, qui expriment l'absence de comportements particulièrement indésirables. L'outil LESAR [Glo89] [Rat92] permet de prouver formellement qu'un programme LUSTRE satisfait un ensemble de propriétés de sûreté. Il automatise un processus de preuve qui requiert l'expression en LUSTRE de ces propriétés. Du programme est construit un modèle (un automate d'états finis) qui est exhaustivement parcouru afin de montrer que les propriétés sont satisfaites dans chacun de ses états (par la technique dite de "model-checking"). En cas de preuve réussie, on peut conclure que le logiciel satisfait les propriétés spécifiées. Cette technique de preuve relève de la vérification.

Les propriétés de sûreté sont dans la plupart des cas des relations simples entre les signaux d'entrée et de sortie du logiciel. De ce fait, les modèles manipulés par la preuve restent souvent d'une taille raisonnable, si bien que des applications réelles ont pu être vérifiées. Il est cependant fréquent que l'utilisation de cette technique de preuve se heurte à un problème classique des techniques de ce type : l'explosion du nombre d'états des modèles. En effet, ce nombre devient prohibitif au fur et à mesure que la complexité et la taille du programme et des propriétés à vérifier augmentent. De nombreux travaux ont été effectués pour trouver des solutions à ce problème crucial, en ne représentant qu'une partie des états en mémoire (vérification à la volée), en comprimant les représentations (ordres partiels), en décomposant le système à vérifier ou encore en le traitant partiellement (réduction, abstraction). Une autre approche, la manipulation symbolique de modèles, connaît un fort essor mais elle rend souvent les traitements plus longs si bien que dans plusieurs cas la diminution du nombre d'états est compensée par une augmentation très importante du temps de traitement.

La preuve formelle de programmes LUSTRE s'appuie sur l'exécution symbolique du logiciel et vise exclusivement à prouver que les propriétés considérées sont respec-

tées par tous ses comportements. Cependant, cette technique ne nous permet pas d'affirmer que le logiciel répond *réellement* à nos attentes et ceci au moins pour deux raisons :

- La première raison est relative au problème de la *validation des spécifications.* En effet, les propriétés de sûreté spécifiées peuvent être insuffisantes ou fausses. Ainsi, en cas d'échec de la preuve, il peut être difficile de décider si nous sommes en présence d'un défaut du logiciel ou d'un défaut de sa spécification. De plus, une preuve réussie ne montre que la satisfaction de propriétés dont on n'a aucun moyen d'assurer qu'elles excluent bien tous les comportements dangereux. Ce point de vue est conforté par la définition donnée dans [Lap95] de la défaillance : une défaillance est une non conformité à la *fonction* du système, et non à sa spécification qui peut être erronée.
- La deuxième raison est que la preuve s'intéresse essentiellement aux propriétés de sûreté du logiciel et plus généralement les propriétés qui peuvent être exprimées en LUSTRE. Or, certains défauts du logiciel peuvent ne pas avoir d'impact sur la préservation de ces propriétés et passeront, de ce fait, inaperçus.

Par ailleurs, si la preuve ne peut pas aboutir, suite à un manque de ressources physiques, on ne peut rien conclure sur les chances du logiciel de satisfaire sa spécification.

LE TEST DES LOGICIELS

Le *test des logiciels* nous a semblé une approche intéressante pouvant combler certaines insuffisances des techniques de vérification fondées sur la preuve. Une des définitions les plus acceptées du test [Mye79] l'identifie à un processus consistant à exécuter le logiciel dans l'intention de détecter des défauts. Ces défauts peuvent être introduits à différentes étapes du cycle de développement si bien que chaque étape se prête à un type de test différent. Ainsi, le test unitaire concerne les composants élémentaires du logiciel (modules ou objets) et à détecter les défauts introduits lors de la programmation. Le test d'intégration s'intéresse à l'interaction entre ces composants et peut mettre en évidence de problèmes d'interface entre modules ou des défauts dans l'architecture du logiciel. Le test système, enfin, consiste à tester l'ensemble du logiciel et à comparer les résultats obtenus aux spécifications initiales dans le but de détecter des incohérences.

Quelle que soit l'étape du cycle de développement, l'activité de test se déroule généralement en deux temps. Un ensemble de données d'entrée (souvent appelées *jeux de test* ou *jeux d'essai*) est d'abord sélectionné. Le logiciel est ensuite exécuté avec ces données en entrée et son comportement est observé. Cette observation du comportement est souvent effectuée par un humain qui doit prononcer un "oracle" (i.e. il doit décider si les résultats de l'exécution sont corrects).

Dans certains cas, le verdict de conformité entre le comportement observé et celui attendu peut être rendu de manière automatique. Un programme (appelé, par abus de

langage, oracle) est chargé de détecter automatiquement les comportements erratiques du logiciel. Il compare pour cela tous les comportements obtenus à une spécification et signale la présence de ceux qui n'y sont pas conformes.

La nature de la spécification utilisée dans un oracle automatique varie d'une application à l'autre. On retiendra, cependant, que la construction d'un oracle détectant tous les mauvais comportements du logiciel est une tâche aussi difficile que la construction du logiciel lui-même et qu'en pratique, on se contente d'une spécification restreinte (par exemple sous forme de table d'association entre certaines entrées et sorties ou d'un ensemble de propriétés exprimées sur les entrées et les sorties). Notons, enfin, que l'utilité pratique des oracles automatiques a été mise en évidence par des expériences qui ont montré que l'observation par un humain de l'exécution d'un logiciel ne permet pas, à elle seule, la détection de toutes ses défaillances [BS87].

La comparaison du comportement du programme à sa spécification est typiquement une activité de vérification. On vérifie, en effet, que l'implantation correspond aux intentions exprimées par la spécification. Néanmoins, l'obligation d'exécuter le logiciel fait également du test une activité de validation car un observateur humain peut déceler des comportements (éventuellement acceptés par l'oracle automatique) qui ne correspondent pas à ses attentes (cas de spécification incomplète ou erronée). Soulignons ici que l'écriture des spécifications est une tâche complexe, exigeant souvent une bonne maîtrise d'un ou plusieurs langages formels, pendant laquelle des fautes ou des omissions sont loin d'être exclues. La simple comparaison du comportement du logiciel à sa spécification ne révélera pas la présence de ces fautes : seul un observateur humain peut les détecter. Par ailleurs, il se peut que la spécification d'un logiciel ne puisse être entièrement déterminée que par l'observation de ce dernier dans son environnement opérationnel [Lap93].

De nombreux travaux de recherche ont été menés sur le test des logiciels. Bien que certains aient cherché à définir un cadre théorique général pour le test, la plupart d'entre eux ont proposé des techniques spécifiques à des défauts particuliers (défauts de programmation, défauts liés au respect de la spécification etc.). Ainsi, une bonne méthode de recherche de défauts doit adopter plusieurs techniques de test différentes, en fonction des défauts visés. L'efficacité en termes de pouvoir de détection de défauts et de coût de toutes ces techniques est rarement établie de manière formelle. Elle repose plutôt sur des études statistiques [WWH91].

On distingue deux grandes catégories de techniques. Les techniques de la première catégorie, dites en "boîte noire", se contentent d'exécuter le logiciel avec des données de test sélectionnées au hasard ou bien construits à partir d'une spécification (formelle ou informelle) du logiciel (dans ce dernier cas on parle aussi de *test fonctionnel*). Ces techniques ne prennent pas en compte la manière dont le logiciel a été implanté (langage utilisé, code produit etc.). Elles sont adaptées principalement à la détection de défauts dus à une mauvaise compréhension de la spécification du logiciel.

Les techniques de la deuxième catégorie, dites *structurelles* (ou encore en "boîte de verre"), s'appuient, elles, sur une analyse du code réalisant le logiciel. Elles considè-

rent des jeux d'essai dont l'exécution assure l'activation de ce code de différentes manières. Par exemple, la *couverture des instructions* est satisfaite par un jeu de test si toutes les instructions du logiciel sont exécutées au moins une fois. Les défauts visés par les techniques structurelles sont ceux directement liés à l'implantation et au langage de programmation utilisé.

Contrairement à la vérification formelle, aucune technique de test réaliste ne peut servir à la preuve de la correction d'un programme. En effet, prouver la correction du logiciel nécessiterait son exécution dans tous les cas possibles (i.e. avec toutes les données d'entrée possibles) ce qui est généralement impossible en pratique.

Ainsi, la seule chose qu'on peut prouver par le test est la présence (et non l'absence) de défauts dans le logiciel. Notons cependant que cette perception du test est aujourd'hui moins partagée. En effet, si le but est de révéler des défauts, l'absence de défaillance ne mène à aucune conclusion sur la correction du logiciel. Or, il semble raisonnable de vouloir traduire cette absence de défaillance en une mesure, reflétant précisément la *probabilité d'absence* de défauts. Une tentative de définition d'une telle mesure, appelée "correction probable", a été réalisée par Hamlet [Ham95]. Ce même auteur prédit une évolution du test, de moyen de recherche de défauts en un moyen d'estimation de la qualité du logiciel. Une autre illustration industrielle de cet usage du test pour établir la correction vient du domaine de la validation des protocoles [ISO91]. Le test y est clairement utilisé dans le but d'établir la correction d'un logiciel et plus particulièrement de certifier sa conformité à une spécification si bien qu'on parle de *test de conformité*.

TEST DES LOGICIELS RÉACTIFS SYNCHRONES

L'objectif principal du travail que nous présentons ici est de proposer des techniques de test aptes à détecter des défauts dans les logiciels réactifs synchrones. La variété des défauts potentiellement présents dans ces logiciels nous a obligé à considérer plusieurs techniques de test, chacune adaptée à une catégorie particulière de défauts. Plus précisément, comme dans la plupart des travaux connus sur le test, nous avons considéré deux grandes classes de défauts : ceux qui sont dus à une mauvaise compréhension de la spécification et ceux directement issus de l'activité de programmation.

Nous nous sommes par ailleurs efforcés de concevoir des techniques qui ne demandent pas d'effort de spécification supplémentaire par rapport à la technique de preuve formelle de programmes LUSTRE. Notre but est en effet de fournir un complément à cette technique, complément qui s'intègre facilement dans le processus de vérification existant. Pour ces raisons, nous avons considéré les mêmes contraintes de spécification et nous avons restreint les propriétés exprimées à celles qui n'utilisent que des variables booléennes.

Une particularité des logiciels qui sont l'objet de ce travail est la nécessité de produire des *séquences* de valeurs d'entrée pour les tester. En effet, les propriétés qu'ils doivent vérifier expriment des relations entre les valeurs que prennent les entrées et les

sorties du logiciel à plusieurs instants différents. L'approche que nous avons adoptée pour cette production est celle d'une génération *dynamique*, par opposition à la constitution avant l'opération de test de l'ensemble des séquences d'entrées. Pour cela, nous avons défini des processus de dérivation *entièrement automatique* des jeux d'essai à partir de la spécification en LUSTRE des différentes caractéristiques du logiciel. Nous obtenons ainsi des programmes *générateurs* de données d'entrée qui sont exécutés en même temps que le logiciel sous test et qui simulent le comportement de son environnement.

La spécification retenue pour la génération des jeux de test est la même que celle utilisée par le processus de preuve formelle de programmes LUSTRE. Ainsi, la spécification développée pour vérifier formellement le programme peut être réutilisée afin de le tester. Cette caractéristique permet d'atteindre notre objectif d'intégration facile de nos techniques dans le processus de vérification.

Plus précisément, nous proposons trois techniques de test avec des motivations différentes pour chacune d'entre elles. La première permet de mettre en place une simulation aléatoire de l'environnement du logiciel. L'observation du comportement du logiciel dans un environnement simulé ne peut que renforcer notre confiance dans son bon fonctionnement dans les conditions réelles d'opération, même si ce type de test ne favorise la détection d'aucun type particulier de défauts. Cette technique, qui s'appuie sur une spécification LUSTRE non déterministe de l'environnement du logiciel, permet par ailleurs la prise en compte de la spécification explicite de probabilités d'occurrence de certains comportements (on parle aussi de *profil opérationnel*).

La deuxième technique a comme objectif de produire des données de test appropriées à la recherche des défauts directement liés au non respect des propriétés de sûreté (on dira que ces données *testent* les propriétés). Pour cela, les propriétés sont automatiquement analysées dans le but d'identifier les cas d'exécution mettant en évidence leur non respect. Cette identification nous permet, en particulier, d'exclure les entrées qui rendraient les propriétés vraies indépendamment de la valeur des sorties et de rendre, ainsi, le processus de test plus efficace, en matière de coût et de probabilité de détection de défauts.

Ces deux premières techniques [PO96a] sont de type "boîte noire" (i.e. elles n'imposent pas que le logiciel soit implanté dans un langage particulier). Cela rend possible leur utilisation pour la vérification et la validation des applications synchrones programmées dans d'autres langages, à condition, bien entendu, que la spécification de l'environnement et des propriétés de sûreté soit exprimée en LUSTRE.

Contrairement aux techniques précédentes, la dernière technique que nous proposons considère que le logiciel réactif est réalisé en LUSTRE. Il s'agit d'une adaptation à LUSTRE de différentes techniques de test structurel utilisées dans le cadre de langages de programmation séquentiels. Sa vocation est de rechercher les défauts liés à l'utilisation de LUSTRE pour l'implantation du logiciel [OP94b].

D'un point de vue méthodologique, cette dernière technique est plutôt adaptée au test unitaire de nœuds LUSTRE de taille modérée. Au contraire, les techniques de géné-

ration aléatoire sous contraintes peuvent être également utilisées pour d'autres types de test (intégration, système), à condition qu'on puisse exprimer sous forme de propriétés invariantes les comportements souhaités des composants logiciels sous test.

Nous espérons que cet ensemble de techniques constitue un outil de vérification d'emploi toujours possible (même si les résultats obtenus sont modestes) et dont l'automatisation permet d'observer un grand nombre de cas d'exécution du logiciel. Par ailleurs, le processus de test étant formellement défini, nous pourrons envisager l'évaluation (et, si possible, la quantification) des résultats obtenus. En même temps, le test permet l'observation des comportements du logiciel et facilite le diagnostic en cas de non respect des spécifications offrant, ainsi, un moyen de validation.

Le test apparaît donc comme un complément irremplaçable à la preuve. La complémentarité des deux approches, test et preuve, pour la vérification et la validation des programmes LUSTRE est d'ailleurs défendue par des travaux de recherche récents sur l'utilisation conjointe des techniques de vérification formelle et de test systématique [MHM+95]. Cette approche considère une spécification du comportement du logiciel sous forme de machine d'états finis. Les données de test sont ensuite engendrées moyennant une analyse de cette spécification. Notons également que d'autres travaux portant sur le test des programmes LUSTRE ont été menés récemment [Hsi94, Cal95] [Maz94]. Dans [Maz94], l'automate engendré par le compilateur LUSTRE est utilisé comme modèle du programme afin d'effectuer du test statistique guidé par différents critères de couverture (couverture des états ou des transitions de l'automate). L'approche adoptée dans [Hsi94] est fondée sur une modélisation algébrique des programmes LUSTRE et plus particulièrement des flots d'entrée et de sortie. L'utilisation d'un outil d'analyse de spécifications algébriques permet, ensuite, d'obtenir de manière automatique des données de test.

Notre approche est fondamentalement différente de celle de [Maz94] car l'automate représentant le programme n'est utilisé par aucune de nos techniques. Par ailleurs, le type de test que l'on effectue ne vise pas, *a priori*, à établir une quelconque couverture de cet automate. Il faut noter qu'un programme LUSTRE peut aussi être compilé en un code "naïf" [Ray91], beaucoup plus compact que l'automate associé, consistant en une simple boucle infinie. Nous nous sommes efforcés de concevoir des techniques pour lesquelles cette forme de code est suffisante et qui ne sont donc pas soumises aux limitations sévères relatives à la taille de l'automate souvent explosive.

Nos techniques de génération aléatoire sous contraintes présentent une certaine analogie avec les approches proposées dans [Hsi94, Cal95] et [MHM+95] qui sont fondées sur l'analyse de spécifications formelles. Toutefois, aussi bien le type des spécifications utilisées que la démarche adoptée pour la génération des jeux de test sont clairement différentes (cf. chapitre 7).

EVALUATION EXPÉRIMENTALE DES TECHNIQUES DE TEST

Une véritable évaluation de l'efficacité de nos techniques ne peut s'opérer que par leur application sur un logiciel de type industriel. Malheureusement, le cadre dans lequel s'est déroulée cette thèse ne nous a pas permis de confronter ces techniques au problème du test d'un tel logiciel. Toutefois, conscients de la nécessité de leur validation par l'expérience, nous avons développé en LUSTRE une version simplifiée d'un logiciel de contrôle d'ascenseur sur laquelle nous avons appliqué certaines des techniques proposées.

Le choix de cet exemple présente plusieurs avantages :

- L'expérience présente peu de difficultés de compréhension. La spécification informelle du fonctionnement d'un ascenseur est très simple. La seule difficulté pour le lecteur réside, donc, d'une part dans la compréhension de la spécification formelle et, d'autre part, dans l'application des techniques de test. On note que cet exemple a été utilisé dans plusieurs études sur la vérification des logiciels réactifs [RLO92] [MHM+95] [DKM+94].
- La taille de l'application et sa complexité rendent, à notre avis, l'expérience convaincante quant aux possibilités d'application des techniques de test à d'autres logiciels.

Le développement de cet exemple nous a permis, entre autres, de nous convaincre de la pertinence de la méthodologie que nous préconisons, nos outils étant d'une aide précieuse pour l'écriture des spécifications (cf. chapitre 6).

PLAN DU DOCUMENT

Le premier chapitre de ce document consiste en un exposé rapide des principaux résultats des travaux de recherche sur la compilation et la vérification du langage LUSTRE. Il inclut la présentation des caractéristiques les plus importantes du langage ainsi qu'une illustration de son utilisation pour la programmation des logiciels réactifs.

Dans le chapitre 2 nous introduisons le test des logiciels et nous résumons les résultats théoriques et pratiques les plus significatifs de ce domaine. Dans ce même chapitre, nous présentons de manière sommaire les techniques de test que nous avons conçues. Nous suggérons également une méthodologie de vérification et validation aussi bien pour le cas où le test est utilisé comme un complément à la vérification formelle que pour le cas où il constitue le seul moyen de vérification et validation.

Les trois chapitres suivants présentent en détail nos techniques de test. Plus précisément, le chapitre 3 concerne les techniques de génération aléatoire de jeux de test qui sont définies formellement tandis que le chapitre 4 est consacré à l'implantation de ces mêmes techniques. Enfin, le chapitre 5 décrit la dernière technique de test qui est de type structurel.

Le chapitre 6 récapitule les réalisations et expériences pratiques que nous avons mises en œuvre durant cette thèse et en particulier, l'illustration de leur utilisation sur le logiciel de contrôle d'ascenseur dont nous avons développée une spécification complète en LUSTRE (fournie en annexe B).

Enfin, dans le chapitre 7 nous replaçons notre travail par rapport à un ensemble de travaux de recherche portant aussi bien sur le test que sur la vérification formelle des logiciels.

1

Programmation et vérification des logiciels synchrones en LUSTRE

1.1 INTRODUCTION

Ce chapitre est consacré à la présentation de LUSTRE. Nous donnons une description informelle du langage avant d'illustrer son utilisation pour la spécification des logiciels réactifs synchrones sur un exemple simple, un système de climatisation. Auparavant, nous présentons rapidement les domaines concernés par le langage : les logiciels réactifs et l'approche synchrone.

Nous nous intéressons ensuite de manière plus approfondie au langage, en particulier en exposant rapidement les techniques de compilation et de vérification formelle associées. Ces résultats ayant à notre travail, leur présentation est nécessaire.

1.2 LES LOGICIELS RÉACTIFS

Selon une de leurs premières définitions donnée dans [Pnu86], les logiciels (ou systèmes) réactifs sont des programmes dont l'objectif est de maintenir une *interaction continue* avec leur environnement et dont l'exécution, idéalement, ne se termine jamais. Les logiciels de contrôle de procédés industriels, les systèmes d'exploitation, les logiciels intégrés aux automates de billetterie en sont des exemples représentatifs. Les logiciels dits *temps-réel* font également partie des logiciels réactifs. Leur particularité réside dans les contraintes de temps de réponse auxquelles ils sont soumis et qui leur sont imposées par leur environnement [BB91].

La définition des logiciels réactifs nécessite l'utilisation de formalismes appropriés puisque nous ne pouvons pas les décrire de la même manière que les logiciels *transformationnels* qui lisent leurs entrées au début de leur exécution et produisent leurs sorties lors de leur terminaison. En effet, un logiciel transformationnel est vu comme

une fonction d'un état initial vers un état final (ou comme une relation entre états, si le logiciel est non déterministe) tandis qu'au contraire, un système réactif est décrit plutôt par ses *comportements* qui sont des séquences, éventuellement infinies, d'états ou d'événements. La description de ces comportements est le plus souvent effectuée à l'aide de *logiques temporelle*s [Pnu86] qui permettent d'exprimer des propriétés portant sur le déroulement même de l'exécution (par exemple des contraintes d'ordonnancement des actions du logiciel).

1.3 L'APPROCHE SYNCHRONE

L'importance attachée au bon fonctionnement des logiciels réactifs impose un certain nombre de règles auxquelles doivent obéir les outils d'aide à leur développement [Ben89]. En particulier, ces outils doivent permettre une spécification claire et uniforme du logiciel et de son environnement d'exécution matériel, qui prend en compte l'aspect asynchrone des communications avec l'extérieur. Ils doivent par ailleurs permettre la génération automatique de code exécutable. Ainsi, les défauts introduits lors des phases de spécification et d'implantation seront sensiblement réduits. Des possibilités de preuve ou de test automatiques doivent par ailleurs permettre de garantir la satisfaction de certaines propriétés cruciales par les logiciels développés.

Plusieurs outils destinés à répondre à ces besoins ont été proposés, suggérant soit la programmation directe des tâches séquentielles communicant à l'aide d'exécutifs temps réel, soit la programmation d'automates pour la réalisation de protocoles (par exemple XESAR [RRSV87]), soit, enfin, la programmation concurrente de haut niveau (ADA, OCCAM). Tous ces outils considèrent que les communications entre les différents composants du logiciel ainsi qu'avec l'extérieur, sont asynchrones. Cette hypothèse leur permet de décrire pratiquement tout type de système réactif mais, *a contrario*, elle a rendu la validation des programmes que l'on développe très difficile.

L'approche *synchrone* [BB91] consiste à ignorer l'asynchronisme de ces communications, les systèmes considérés étant supposés réagir de manière *instantanée* à leurs entrées. En d'autres termes, le temps nécessaire au logiciel pour calculer ses nouvelles sorties à partir de ses entrées est supposé *nul*. Cette hypothèse simplificatrice, dite de *synchronisme*, permet de spécifier proprement les logiciels réactifs ainsi que d'engendrer automatiquement à partir de cette spécification un code exécutable efficace et, surtout, rend possible la preuve de la satisfaction par un programme de certaines propriétés. En pratique, l'hypothèse de synchronisme peut être plus souple, en fonction de l'environnement du logiciel réactif. Il suffit, en effet, que la vitesse de réaction du logiciel soit supérieure à la vitesse d'évolution de son environnement, ou, en d'autres termes, que le temps de réaction du logiciel soit inférieur au délai minimal nécessaire à un changement d'état significatif de l'environnement. En particulier, dans de nombreuses applications où les événements extérieurs sont asynchrones, l'utilisation des moniteurs d'interface permet de se ramener au cas synchrone et de profiter ainsi de ses avantages [BB91].

Plusieurs langages de programmation synchrones ont vu le jour à partir du milieu des années 80. Esterel [BDS91] est un langage impératif tandis que LUSTRE [CHPP87] et Signal [LGLL91] sont des langages déclaratifs. Ces deux derniers exploitent l'hypothèse de synchronisme et la simultanéité des calculs qu'elle implique pour représenter tout système réactif par un système d'équations dynamiques. Cela permet l'utilisation de techniques mathématiques de preuve de propriétés sur ce système d'équations.

Par ailleurs, les compilateurs associés aux langages synchrones produisent un code très efficace et, surtout, mesurable. Dans le cas de LUSTRE [HRR91] et d'Esterel [BDS91] ce code a la forme d'un automate d'états finis dont les transitions ne comportent ni itération ni appel de fonctions récursives, ce qui permet de borner supérieurement le temps nécessaire à leur exécution. Ainsi, la pertinence de l'hypothèse de synchronisme peut être prouvée par le calcul de cette borne supérieure.

1.4 LE LANGAGE LUSTRE

1.4.1 Un langage flot de données synchrones

Le principe de l'approche *flot de données* est de considérer un programme comme un *réseau d'opérateurs* constitué d'un ensemble de *nœuds* (les opérateurs du réseau) connectés par des canaux de communication (appelés également *arcs*), les données étant traitées en traversant ce réseau. Un réseau d'opérateurs peut posséder une *mémoire* et, de ce fait, le calcul d'une sortie peut dépendre, à un instant donné, aussi bien des valeurs courantes des entrées que des valeurs passées. En d'autres termes, un programme est considéré comme une fonction associant à une *séquence d'entrées* une *séquence de sorties*.

Un des langages de programmation flot de données les plus connus est LUCID [AW85] dont LUSTRE [CHPP87] est la version synchrone. Construire un programme LUSTRE revient à décrire un réseau d'opérateurs qui, à partir de ses entrées, calcule ses sorties de manière instantanée. Plus précisément, on considère en LUSTRE que l'exécution d'un programme est pilotée par une *horloge* dont chaque cycle correspond à une lecture des entrées et un calcul des nouvelles sorties. Cela revient à avoir une vue *discrète* du temps, assimilé ici à une suite d'instants. Ainsi, toute expression dans un programme peut être vue comme une *fonction du temps*, associant à un instant la valeur que prend l'expression à cet instant ou encore comme *la suite* des valeurs successives prises par l'expression depuis l'instant initial.

Chaque programme LUSTRE est la définition d'un nouvel opérateur qui peut, à son tour, faire partie d'un réseau plus important (i.e. un autre programme LUSTRE), ce qui permet une implantation hiérarchique. Par analogie au modèle du réseau d'opérateurs, un programme LUSTRE est appelé *nœud* (*node* en anglais).

Un nœud LUSTRE représente le comportement d'un programme au moyen d'un ensemble non ordonné d'équations. Ceci est possible grâce à un résultat fondamental

```
node Never (A : bool) returns (never_A : bool);
let
   never_A = not A -> (not A and pre (never_A));
tel
```

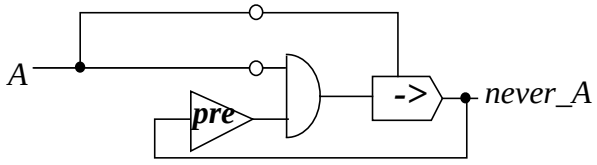

Figure 1-1 : Exemple de programme LUSTRE

obtenu par Kahn [Kah74] qui a montré que le comportement d'un réseau d'opérateurs fonctionnel est l'unique solution minimale d'un ensemble d'équations.

Exemple 1-1

Un exemple de nœud LUSTRE n'utilisant que des opérateurs de base du langage est présenté dans la figure 1-1. Dans cette même figure on trouve la représentation graphique du réseau d'opérateurs associé. Il s'agit d'un programme dont la sortie booléenne prend une valeur vraie si et seulement si son unique entrée n'a jamais été vraie depuis le début de l'exécution. Par exemple, à la séquence de valeurs d'entrée ***(false**, **false**, **false**, **true**, **false)*** le nœud *Never* associe la séquence de valeurs de sortie ***(true**, **true**, **true**, **false**, **false)***.

Il faut souligner que l'approche flot de données est utilisée depuis de nombreuses années pour la spécification d'applications industrielles; ceci a donné naissance à des outils pour accroître l'efficacité du travail de spécification. En particulier, SAGA [PB88] est un environnement de programmation incluant une version graphique simplifiée de LUSTRE.

1.4.2 Présentation de LUSTRE

Nous donnons ici une description sommaire et informelle du langage, mais amplement suffisante pour la lecture de ce document (pour une description plus détaillée on pourra se référer à [HCRP91]).

Un nœud LUSTRE consiste en un ensemble d'équations définissant ses sorties en fonction de ses entrées, éventuellement en utilisant des variables locales. La syntaxe d'un nœud de nom *N* est informellement présentée dans la figure 1-2.

Une expression LUSTRE est constituée de constantes, de variables et d'opérateurs. Les opérateurs booléens et arithmétiques habituels font partie du langage ainsi que deux opérateurs spécifiques à LUSTRE, l'opérateur "précédent", noté ***pre***, et l'opéra-

node *N (<variables d'entrée>)* ***returns*** *(<variables de sortie>);*
var *<variables locales>;*
let
    <équations et assertions>
tel

Figure 1-2 : Syntaxe d'un nœud LUSTRE

teur "suivi de", noté **->**, dont la définition est la suivante (ces deux opérateurs sont utilisés dans l'exemple de la figure 1-1) :

- Si *E* est une expression dénotant la séquence de valeurs $(e_0, e_1, ..., e_n, ...)$ alors ***pre*** *E* dénote la séquence des valeurs $(nil, e_0, e_1, ..., e_{n-1}, ...)$ où *nil* est une valeur indéterminée. En d'autres termes, ***pre*** permet d'accéder à un instant *t* à la valeur prise par une expression à l'instant *t-1*.
- Si *E* et *F* sont des expressions dénotant respectivement les séquences de valeurs $(e_0, e_1, e_2, ..., e_n, ...)$ et $(f_0, f_1, f_2, ..., f_n, ...)$ alors *E* **->** *F* dénote la séquence des valeurs $(e_0, f_1, f_2, ..., f_n, ...)$. De manière informelle, l'opérateur **->** permet de définir la valeur prise par une expression à l'instant initial $t = 0$.

Toutes les expressions LUSTRE possèdent un type. Les types proposés par LUSTRE sont les booléens, les entiers et les réels. Tous les opérateurs habituels de ces types sont disponibles, leur application sur les séquences d'éléments se faisant point par point.

Notons l'existence d'un opérateur peu commun : **#**. Cet opérateur booléen appliqué sur une liste finie d'expressions booléennes, retourne une valeur vraie quand *au plus une* de ces expressions est vraie.

Des types composés peuvent être construits au moyen de *tuples* qui sont des listes finies d'éléments de type quelconque.

L'utilisation de tableaux est également possible mais uniquement en tant que facilité syntaxique, puisque les indices des éléments accédés dans le programme doivent être tous calculables au moment de la compilation (cf. paragraphe 1.4.3).

Ainsi, les équations d'un nœud sont de la forme

<identificateur> = <expression> ;

où *<identificateur>* est soit un nom de variable, soit un tuple.

Toute variable autre qu'une entrée doit être définie au moyen d'une et une seule équation. Le *principe de substitution* permet de remplacer toute expression *E* par une variable *x* si l'équation $x = E$ fait partie du programme (et inversement).

Une variable étant une fonction du temps, sa valeur à un instant ne peut dépendre que des valeurs qu'elle a prises aux instants précédents. Ainsi, l'équation *x = x + 1* n'est pas autorisée en LUSTRE (cela signifierait *x(t) = x(t) +1* et entraînerait une récursion infinie). Ce type de définitions, appelées "boucles combinatoires", sont détectées et refusées par le compilateur LUSTRE.

Enfin, les *assertions* sont un moyen de spécifier des hypothèses sur l'environnement du logiciel et peuvent être introduites grâce à l'opérateur ***assert***. Une assertion a la forme

assert <*expression*> *;*

et spécifie que l'expression booléenne <*expression*> restera continuellement satisfaite tout au long de l'exécution du programme.

Notons qu'une caractéristique de LUSTRE que nous n'avons pas pris en compte dans ce travail est la possibilité de définir des *horloges multiples*. Cela consiste à associer à chaque expression (au moyen de l'opérateur ***when***) une condition qui détermine à quel moment l'expression sera évaluée. L'opérateur ***current*** (dit *de projection*) permet de réaliser des opérations sur des expressions possédant des horloges différentes.

Notre choix d'ignorer cette caractéristique est motivé par le fait que les travaux sur la vérification formelle des programmes LUSTRE [Rat92] n'ont pas traité cet aspect. Ces travaux concernaient, en effet, la preuve des applications écrites en SAGA qui ne permet pas la définition d'horloges multiples. Cela nous autorise à croire que notre choix n'a pas de conséquences graves sur le champ d'application des techniques que nous avons développées. Par ailleurs, l'absence d'horloges multiples rend la sémantique de LUSTRE beaucoup plus simple.

En conséquence, nous considérons que toutes les expressions du programme sont définies sur la même horloge, l'*horloge de base* du programme. Cette dernière correspond à un cycle d'exécution, commençant par une lecture des entrées et se terminant par l'émission des sorties après leur calcul.

1.4.3 Tableaux et nœuds récursifs

Les tableaux et les nœuds récursifs sont destinés à rendre les programmes LUSTRE plus courts, plus simples et plus structurés (cf exemple du chapitre 6). Il s'agit, en réalité, de facilités syntaxiques. En effet, avant la compilation, les tableaux sont remplacés par autant de variables qu'ils ont d'éléments. Au même moment, les nœuds récursifs qui définissent des traitement répétitifs sur les tableaux sont remplacés par un ensemble d'équations. Cela implique que la taille des tableaux, mais aussi les indices des éléments référencés, doivent être connus au moment de la compilation. Cette restriction est nécessaire afin de garantir le respect de l'hypothèse de synchronisme.

```
const p = 5;

node OR(const n: int; Tab : bool^n) returns (OR_de_Tab : bool);
let
   OR_de_Tab = with n = 1 then Tab[0]
               else Tab[0] or OR(n-1, Tab[1..n-1]);
tel

node Exemple1(tab1, tab2 : bool^p) returns ( AND_de_OR : bool)
let
   AND_de_OR = OR(p, tab1) and OR(p, tab2);
tel

node Exemple2(tab1, tab2 : bool^p) returns ( AND_de_OR_partiel : bool)
let
   AND_de_OR_partiel = OR(p-1, tab1[0..p-2]) and OR(p, tab2[1..p-1]);
tel

node Exemple3(tab1, tab2 : bool^p) returns ( OR_de_AND : bool)
let
   OR_de_AND = OR(p, tab1 and tab2);
tel

node Exemple4(tab1 : bool^p; e : bool) returns ( tab2 : bool)
let
    tab2[0..p-1] = tab1[0..p-1] or e^p;
tel
```

Figure 1-3 : Tableaux et nœuds récursifs en LUSTRE

Exemple 1-2

Quelques exemples simples d'utilisation des tableaux et des nœuds récursifs sont donnés dans la figure 1-3. Le nœud *OR* est un nœud récursif défini sur des tableaux de booléens et calcule la disjonction des éléments du tableau. Il utilise pour cela l'opérateur conditionnel statique ***with then else*** dont la fonction est de définir des traitements récursifs. Le premier paramètre de ce nœud est la taille du tableau, le deuxième étant le tableau lui-même (***bool^****n* signifie "tableau de *n* booléens").

Le nœud *Exemple1* définit un prédicat à deux paramètres : deux tableaux de booléens de taille *p*. Sa valeur est vraie uniquement quand au moins un des éléments de chaque tableau vaut vrai.

Le nœud *Exemple2* réalise un calcul similaire mais en se limitant à la "tranche" des *p-1* premiers éléments du premier paramètre et des *p-1* derniers éléments du second.

Le nœud *Exemple3* retourne une valeur vraie si le tableau résultant de la conjonction élément par élément de ses deux paramètres comporte au moins un élément vrai. Nous remarquons que l'opérateur booléen ***and*** peut être appliqué aux tableaux (*polymorphisme*).

Enfin, le nœud *Exemple4* retourne un tableau *tab2* dont chaque élément est la disjonction de l'élément correspondant de son premier paramètre *tab1* avec son deuxième paramètre *e*. Notons que pour toute expression LUSTRE *E, E^p* dénote le tableau de longueur *p* dont tous les éléments ont la valeur de *E*.

1.4.4 Un programme réactif simple

Nous illustrons l'utilisation de LUSTRE sur un exemple simple extrait de [AG93] (un exemple plus conséquent est fourni dans le chapitre 6). Il s'agit d'un système de climatisation comportant des dispositifs de diffusion d'air chaud ou froid. Il est commandé à l'aide d'un interrupteur de mise sous tension et d'un thermostat réglable. En fonction de la température ambiante, dont il est informé au moyen d'une sonde, il doit diffuser de l'air chaud ou froid jusqu'à ce que la température de son thermostat soit atteinte. Nous souhaitons réaliser un logiciel permettant de contrôler ce climatiseur suivant cette spécification informelle.

Il est intéressant de noter que le modèle retenu pour la spécification initiale du système, fournie dans [AG93], n'est pas synchrone (spécification SCR [Hen80]). Au paragraphe 1.4.5 nous donnons les hypothèses justifiant l'application de l'approche synchrone à cet exemple.

Le système est représenté par une machine à quatre états (appelés *modes*) :

- *ARRETE*, correspondant à l'état "hors service" du climatiseur,
- *CHAUD*, signifiant que le climatiseur est en marche et en train de diffuser de l'air chaud,
- *FROID*, symétrique au précédent (en marche et diffusion d'air froid)
- *INACTIF*, signifiant que le climatiseur est en marche mais ne diffuse pas d'air (cette situation se produit quand la température ambiante est identique à celle indiquée par thermostat).

Les changements de mode (i.e. les transitions du système) sont opérés sous certaines *conditions* :

- *Marche* est vraie quand l'interrupteur de mise sous tension est en position "marche", fausse sinon,
- *TempInf*, *TempOK* et *TempSup* sont vraies quand la température ambiante est respectivement inférieure, égale ou supérieure à la température indiquée par le

Mode courant	*Marche*	*TempInf*	*TempOK*	*TempSup*	*Nouveau Mode*
ARRETE	*@T*	***f***	*t*	***f***	*INACTIF*
	@T	*t*	***f***	***f***	*CHAUD*
	@T	*f*	***f***	*t*	*FROID*
INACTIF	*@F*	-	-	-	*ARRETE*
	t	*@T*	***@F***	***f***	*CHAUD*
	t	***f***	***@F***	*@T*	*FROID*
CHAUD	*@F*	-	-	-	*ARRETE*
	t	***@F***	*@T*	***f***	*INACTIF*
FROID	*@F*	-	-	-	*ARRETE*
	t	***f***	*@T*	***@F***	*INACTIF*

Figure 1-4 : Spécification originale du climatiseur

thermostat du climatiseur. Ces trois conditions constituent une vue simplifiée des informations en provenance de la sonde et du thermostat.

Le changement de valeur logique d'une condition correspond à un *événement*. Ainsi, deux types d'événements sont considérés :

- *@T* correspond au passage de la condition de faux à vrai.
- *@F* correspond au passage de la condition de vrai à faux.

Selon ces définitions, la spécification du système peut se résumer par la table présentée dans figure 1-4. Chaque ligne de cette table décrit une transition du modèle. Par exemple, la deuxième ligne spécifie que le climatiseur passe du mode *ARRETE* au mode *CHAUD* au moment où la condition *Marche* devient vraie, à condition que la température ambiante soit, à ce moment, inférieure à celle indiquée par le thermostat (i.e. que la condition *TempInf* soit vraie).

La première hypothèse sur laquelle est fondée cette spécification est qu'une et une seule des conditions *TempInf, TempOK* et *TempSup* peut être vraie à la fois. De ce fait, les événements et les valeurs des conditions de la table de la figure 1-4 ne sont pas indépendants. Ainsi, on note en caractères gras les événements et conditions qui sont déduits des autres informations constituant la transition. Par exemple, dans la première ligne de la table, les valeurs des conditions *TempInf* et *TempSup* sont fixées à faux du moment que la condition *TempOK* est vraie.

Par ailleurs, nous pouvons remarquer qu'il n'existe aucune transition permettant de passer directement du mode *CHAUD* au mode *FROID* et inversement. Nous consta-

tons, donc, qu'une autre hypothèse est nécessaire pour que cette spécification soit pertinente : entre le passage à vrai de la condition *TempInf* et celui de la condition *TempSup*, la condition *TempOK* sera vraie au moins une fois (et inversement). On note, par exemple, que l'occurrence de l'événement *@T* sur la condition *TempInf* implique l'occurrence de *@F* sur la condition *TempOK*. Cela revient à supposer une *continuité* dans la variation de la température sondée, c'est à dire à considérer que la température ambiante ne peut pas évoluer d'une valeur inférieure à celle indiquée par le thermostat à une valeur supérieure qu'en étant, entre temps, égale à la température indiquée par le thermostat.

Ces deux hypothèses portent sur l'*environnement* du climatiseur, qui est assimilé aux séquences de valeurs prises par les conditions *Marche*, *TempInf*, *TempOK* et *TempSup* (cf. paragraphe 1.4.7.1).

1.4.5 Application de l'hypothèse de synchronisme

Afin de pouvoir spécifier le logiciel de l'exemple en LUSTRE, nous sommes obligés de justifier l'adéquation de l'approche synchrone. Il suffit pour cela de montrer sous quelles hypothèses le modèle synchrone est réaliste.

Une première hypothèse que nous exprimons porte sur la vitesse d'évolution de la température. Nous supposons que le temps nécessaire pour percevoir une variation *significative* de la température (i.e. correspondant à la modification de la valeur des conditions *TempInf*, *TempOK*, *TempSup*) est supérieur au temps de réaction du programme. En d'autres mots, le temps écoulé entre le passage à vrai d'une des trois conditions relatives à la température et son passage à faux (et inversement) doit être supérieur au temps de réaction du programme.

Le même genre d'hypothèse doit s'exprimer sur l'interrupteur de mise sous tension qui doit toujours rester dans une position significative ("marche" ou "arrêt") pour un temps au moins aussi long que le temps de réaction du programme.

La première hypothèse garantit que le logiciel pourra prendre en compte tout changement significatif de température. Dans le cas où la deuxième hypothèse n'est pas satisfaite, une action de mise sous ou hors tension pourrait être ignorée si elle est suivi de très près d'une autre modification de l'état de l'interrupteur.

1.4.6 LUSTRE vu comme une logique temporelle

Les travaux portant sur la définition de la sémantique de LUSTRE ont montré que cette dernière peut être entièrement exprimée au moyen d'une logique temporelle [PH88]. Ainsi, LUSTRE peut être considéré comme un sous-ensemble de cette logique.

Par ailleurs, il a été montré que les propriétés les plus importantes pour les logiciels réactifs (propriétés de sûreté) peuvent être exprimées à l'aide de ce sous-ensemble et, donc, en LUSTRE [Glo89] [HPOG89]. Ainsi, une approche mono-langage a été adop-

tée pour la vérification des programmes LUSTRE, consistant à exprimer dans le même formalisme le programme et ses propriétés de sûreté.

Plus précisément, les propriétés de sûreté sont des invariants temporels dont l'expression est possible grâce à l'opérateur ***pre*** du langage. Or, un petit ensemble d'opérateurs temporels suffit à l'expression de ces propriétés [Glo89]. Nous présentons ici la définition intuitive, puis en LUSTRE, de deux de ces opérateurs que nous utiliserons dans la suite.

A, *B* et *C* étant des expressions booléennes LUSTRE quelconques :

- L'opérateur *Always A from B to C* est vrai uniquement si entre les deux derniers instants où respectivement *B* et *C* ont été vraies, *A* a *toujours* été vraie.
- De manière analogue, l'opérateur *Once A from B to C* est vrai uniquement si entre les deux derniers instants où respectivement *B* et *C* ont été vraies, *A* a été vraie *au moins une fois*.

L'expression en LUSTRE de ces opérateurs est réalisée à l'aide de nœuds dont la description complète [Rat92] [HLR92] est donnée dans la figure 1-5.

1.4.7 Spécification d'un logiciel synchrone en LUSTRE

Nous considérons que la description d'un logiciel synchrone nécessite trois types de spécifications :

- Une *spécification fonctionnelle* consistant à décrire au moyen d'un nœud LUSTRE le comportement du logiciel. Il s'agit de définir le calcul exact des sorties du logiciel à partir de ces entrées.
- Une *spécification de l'environnement* du logiciel formée d'un ensemble de propriétés invariantes que satisfait l'environnement.
- Une *spécification des propriétés de sûreté* du logiciel consistant de nouveau en une liste de propriétés invariantes portant, cette fois, sur le comportement du logiciel et exprimant l'absence de comportements dangereux.

Ce principe de spécification est suggéré par les travaux sur la vérification formelle de programmes LUSTRE [Rat92].

Nous considérons pour la suite de ce paragraphe que ces trois spécifications sont développées de manière *indépendante* (cela explique, en particulier, certaines redondances dans l'exemple illustrant le principe de spécification). De plus, nous présentons en premier la spécification de l'environnement et des propriétés de sûreté du logiciel en laissant pour la fin la spécification fonctionnelle. Nous pensons ainsi faciliter la compréhension du problème puisque cette démarche nous semble plus appropriée pour ce type de logiciels : la spécification fonctionnelle n'est abordée qu'une fois l'environnement étudié et spécifié et les situations dangereuses identifiées.

```
node Always_from_to(A, B, C : bool) returns (always_A_from_B_to_C : bool);
let
   always_A_from_B_to_C = Once_since(C,B) or Always_since(A,B);
tel

node Once_from_to(A, B, C : bool) returns (once_A_from_B_to_C : bool);
let
   once_A_from_B_to_C = Implies(C, Once_since_(A,B));
tel

node Once_since(A, B : bool) returns (once_A_since_B : bool);
let
   once_A_since_B =  if B then A else (true -> (A or pre (once_A_since_B)));
tel

node Always_since(A, B : bool) returns (always_A_since_B : bool);
let
   always_A_since_B =   if Never(B) then true
                        else   if B then A
                               else (true -> A and pre (always_A_since_B));
tel

node Implies(A, B: bool) returns (A_implies_B : bool);
let
   A_implies_B = not A or B;
tel
```

Figure 1-5 : Opérateurs temporels définis en LUSTRE

1.4.7.1 Spécification de l'environnement

Nous assimilons l'environnement du logiciel à l'*ensemble des variables d'entrée* de ce dernier. Par conséquent, spécifier les comportements de l'environnement revient à spécifier les séquences de valeurs que prennent ces variables. Dans l'exemple, il s'agit des séquences de valeurs de *(Marche, TempInf, TempOK, TempSup).* En effet, ces conditions résument tous les événements externes significatifs pour le climatiseur, à savoir un changement de température et une mise sous ou hors tension.

Nous nous contentons ici d'exprimer en LUSTRE les deux hypothèses sur la variation de la température identifiées dans le paragraphe 1.4.4.

1. Une seule des conditions *TempInf*, *TempOK* et *TempSup* peut être vraie à la fois :
   (TempSup ***or*** *TempOK* ***or*** *TempInf)* ***and*** *#(TempSup, TempOK, TempInf)*
2. Entre le passage à vrai de la condition *TempInf* et celui de la condition *TempSup*, la condition *TempOK* sera vraie au moins une fois (et inversement) :
   Once_from_to(TempOK, TempInf, TempSup) ***and***
   Once_from_to(TempOK, TempSup, TempInf)

Ces contraintes doivent être vérifiées par toute séquence d'entrées du programme. Une discussion plus détaillée sur le type de contraintes qui peuvent être utilisées pour la spécification de l'environnement est présentée au chapitre 3.

1.4.7.2 Spécification des contraintes de sûreté

Comme nous l'avons mentionné auparavant, les propriétés de sûreté sont des formules de logique temporelle qui doivent être satisfaites par toute séquence d'entrées et sorties du programme. A l'opposé des contraintes d'environnement, les propriétés de sûreté sont sensées exprimer des contraintes sur le calcul des sorties du logiciel.

Nous reportons, dans un premier temps, les propriétés de sûreté données dans [AG93][1] :

1. Le système est arrêté uniquement quand l'interrupteur de mise en marche est en position 'arrêt' :
   ARRETE $\Rightarrow$ *~Marche*
2. Le système est inactif uniquement quand il est sous tension et la température ambiante est identique à celle indiquée par le thermostat :
   INACTIF $\Rightarrow$ *(Marche & TempOK)*
3. Le système diffuse de l'air chaud uniquement quand il est sous tension et la température ambiante est inférieure à celle indiquée par le thermostat :
   CHAUD $\Rightarrow$ *(Marche & TempInf)*
4. Le système diffuse de l'air froid uniquement quand il est sous tension et la température ambiante est supérieure à celle indiquée par le thermostat :
   FROID $\Rightarrow$ *(Marche & TempSup)*
5. Le système étant sous tension, dès que la température ambiante devient inférieure à celle indiquée par le thermostat, il se met à diffuser de l'air chaud :
   (Marche & TempInf) $\Rightarrow$ *(CHAUD|O(CHAUD))*
   où *O* est l'opérateur 'état suivant' : *O(A)* signifie que *A* sera vrai dans l'état suivant du système.

[1] Les symboles $\Rightarrow$, ~, &, | désignent respectivement les opérateurs logiques d'implication, de négation, de conjonction et de disjonction.

6. La même propriété est définie pour le cas de la diffusion d'air froid :
   (Marche & TempSup) ⇒ (FROID|O(FROID))

L'expression en LUSTRE des quatre premières propriétés avec les seuls opérateurs de base est immédiate. En revanche, les deux dernières formules ne peuvent pas être directement traduites en LUSTRE suite à l'utilisation de l'opérateur *O* qui fait référence au futur. En effet, LUSTRE ne possède que l'opérateur temporel ***pre***. Cependant, l'opérateur *O* n'implique que des valeurs de l'état suivant du logiciel ce qui permet la traduction des formules l'utilisant, après une légère transformation qui consiste simplement à décaler l'instant de référence :

1. ***not*** *ARRETE* ***or not*** *Marche*
2. ***not*** *INACTIF* ***or*** *Marche* ***and*** *TempOK*
3. ***not*** *CHAUD* ***or*** *Marche* ***and*** *TempInf*
4. ***not*** *FROID* ***or*** *Marche* ***and*** *TempSup*
5. ***not pre*** *(Marche* ***and*** *TempInf)* ***or*** *(****pre*** *CHAUD* ***or*** *CHAUD)*
6. ***not pre*** *(Marche* ***and*** *TempSup)* ***or*** *(****pre*** *FROID* ***or*** *FROID)*

Enfin, nous pouvons ajouter une nouvelle propriété pour garantir que le système se trouve à tout moment dans un mode et un seul (d'ailleurs, cette propriété peut être déduite des hypothèses sur l'environnement et des six autres propriétés de sûreté) :

7. *#(ARRETE, INACTIF, CHAUD, FROID)* ***and*** *(ARRETE* ***or*** *INACTIF* ***or*** *CHAUD* ***or*** *FROID).*

1.4.7.3 Réalisation en LUSTRE

La figure 1-6 présente une possibilité d'implantation du logiciel de contrôle du climatiseur en LUSTRE au moyen du nœud *Clim*. Les entrées de ce nœud sont les conditions sur la température et la marche du système, ses sorties étant les modes du logiciel de contrôle. Les équations du nœud définissent les conditions qui doivent être vérifiées pour que le système se trouve dans un mode donné.

Notons que ce nœud est une implantation simplifiée de la spécification initiale de la figure 1-4. En effet, les équations définissant les modes *CHAUD* et *FROID* permettent le passage du mode *FROID* au mode *CHAUD* sans passage intermédiaire par le mode *INACTIF*. Une définition plus exacte pour ces deux modes serait :

```
CHAUD = Marche and TempInf ->
            pre(ARRETE or INACTIF or CHAUD) and Marche and TempInf ;
FROID = Marche and TempSup ->
            pre(ARRETE or INACTIF or FROID) and Marche and TempSup ;
```

```
node Clim(
   Marche,              -- position de l'interrupteur
    TempInf,            -- temp. ambiante < thermostat
   TempOK,              -- temp. ambiante = thermostat
   TempSup : bool);     -- temp. ambiante > thermostat
returns(
   ARRETE,              -- hors tension
   INACTIF,             -- sous tension, pas de diffusion
   CHAUD,               -- diffusion d'air chaud
   FROID: bool)         -- diffusion d'air froid
let
   assert ((TempSup or TempOK or TempInf) and
      #(TempSup, TempOK, TempInf));
   assert(once_from_to(TempOK, TempInf, TempSup) and
      once_from_to(TempOK, TempSup, TempInf));
   ARRETE = not Marche;
   INACTIF = Marche and TempOK ;
   CHAUD = Marche and TempInf;
   FROID = Marche and TempSup;
tel;
```

Figure 1-6 : Implantation du logiciel de contrôle du climatiseur

Nous constatons très facilement que les quatre premières propriétés spécifiées au paragraphe 1.4.7.2 sont vérifiées par cette implantation. Cette facilité n'est (malheureusement) due qu'à la simplicité de l'exemple. Dans le cas général, (voir l'exemple plus conséquent fourni au chapitre 6) la lecture du programme ne permet pas d'émettre un avis sur la satisfaction des propriétés de sûreté.

1.5 COMPILATION ET VÉRIFICATION DE PROGRAMMES LUSTRE

Nous nous intéressons maintenant à la définition formelle du langage et aux techniques de compilation et de vérification de programmes LUSTRE. Dans un premier temps, nous reportons la sémantique des programmes LUSTRE formellement exprimée sur leurs traces d'exécution [Rat92]. Intuitivement, une trace d'un programme *P* correspond à la suite des valeurs prises par les variables de *P* pendant une exécution.

En s'appuyant sur cette sémantique, nous rappelons ensuite que tout programme LUSTRE peut être représenté par un modèle abstrait, une machine d'états finis, dont nous expliquons le principe de définition des états et des transitions. Ce modèle est en fait construit par le compilateur LUSTRE pour la production d'un code exécutable efficace, ainsi que par l'outil de vérification formelle LESAR.

1.5.1 Sémantique opérationnelle de LUSTRE

I étant l'ensemble des identificateurs d'un programme LUSTRE et *V* l'ensemble de leurs valeurs, on appelle *mémoire* toute fonction σ de *I* dans *V*. Toute mémoire caractérise formellement un état d'un programme à un instant donné en associant à toute variable *x* du programme sa valeur $\sigma(x)$ dans cet état. Toute séquence non vide (finie ou infinie) de mémoires est une *trace d'exécution* du programme.

Un programme LUSTRE associe à une séquence d'entrées infinie une séquence de sorties également infinie. Sa sémantique est donc définie sur les traces d'exécution infinies. Toutefois, il suffit de donner cette sémantique sur les traces finies, le passage aux traces infinies étant ensuite immédiat.

Nous pouvons définir la valeur de toute expression d'un programme LUSTRE pour une trace finie $\Sigma = (\sigma_0, ..., \sigma_n)$ (on note $\Sigma \vdash E \mid v$ le fait que l'expression *E* prend la valeur *v* après une exécution dont la trace est Σ [Rat92]) :

- k étant une constante et *k* sa valeur,
  $\Sigma \vdash k \mid k$
- x étant une variable,
  $(\sigma_0, ..., \sigma_n) \vdash x \mid \sigma_n(x)$
- $*$ dénotant tout opérateur *n*-aire ($n \geq 1$) arithmétique, booléen ou conditionnel, E_i étant des expressions quelconques,
  $$\frac{\Sigma \vdash E_1 \mid v_1, ..., \Sigma \vdash E_n \mid v_n}{\Sigma \vdash * (E_1, ..., E_n) \mid * (v_1, ..., v_n)}$$
- E_1 et E_2 dénotant deux expressions du langage et $\Sigma.\sigma$ désignant la trace d'exécution obtenue par concaténation de la trace Σ et de la mémoire σ),
  $$\frac{\sigma_0 \vdash E_1 \mid v_1}{\sigma_0 \vdash (E_1 \text{ -> } E_2) \mid v_1} \qquad \frac{\Sigma.\sigma \vdash E_2 \mid v_2}{\Sigma.\sigma \vdash (E_1 \text{ -> } E_2) \mid v_2}$$
- *E* étant une expression quelconque du langage et *nil* étant une valeur indéterminée,
  $$\sigma_0 \vdash \textbf{\textit{pre}}\ E \mid nil \qquad \frac{\Sigma \vdash E \mid v}{\Sigma.\sigma \vdash (\textbf{\textit{pre}}\ E) \mid v}$$

Toute trace Σ représentant le comportement d'un certain programme, il est important de définir les rapports entre les traces et les programmes. Une trace finie Σ est *compatible* avec un programme *P* (on note $\Sigma \vdash P$) si elle est compatible avec les équations et les assertions de ce dernier. Cette compatibilité est définie par les règles suivantes :

- Compatibilité avec les équations et avec leur composition :
  $$\frac{(\sigma_0, ..., \sigma_n) \vdash E \mid v,\ \sigma_n(x) = v}{(\sigma_0, ..., \sigma_n) \vdash x = E}$$
  $$\frac{\Sigma \vdash P_1 \mid v_1,\ \Sigma \vdash P_2 \mid v_2}{\Sigma \vdash P_1; P_2}$$
- Compatibilité avec les assertions :
  $$\frac{\Sigma \vdash A \mid \textbf{\textit{true}}}{\Sigma \vdash \textbf{\textit{assert}}\ A}$$

Une trace infinie $\Sigma = (\sigma_0, ..., \sigma_n, ...)$ est compatible avec un programme LUSTRE si tous ses préfixes finis le sont. Elle est dite *exécutable* si elle est compatible avec les équations et *valide* si elle est, en plus, compatible avec les assertions [Rat92].

1.5.2 Modèle d'exécution des programmes LUSTRE

A partir de la sémantique sur les traces d'exécution, il est possible de définir une sémantique plus simple dans laquelle la valeur d'une expression ne dépend que des deux dernières mémoires d'une trace. Cela permet, en particulier, d'associer à tout programme LUSTRE un modèle fini sous forme d'automate, comme on le montre dans la suite de ce paragraphe.

Pour construire cette nouvelle sémantique, il suffit de supposer que dans tout programme LUSTRE l'opérateur ***pre*** s'applique uniquement à des variables (et non à des expressions plus complexes). Cette hypothèse ne restreint pas le pouvoir d'expression du langage puisque, par application successive du principe de substitution, il est toujours possible de transformer un programme en un autre équivalent la satisfaisant. Sous cette hypothèse, toute expression du programme peut être évaluée sur les deux dernières mémoires de la trace σ, σ'. Plus précisément, toute expression ne comportant pas d'occurrence de l'opérateur ***pre*** est évaluée uniquement à l'aide de σ' tandis que toute expression de la forme ***pre*** *x* est évaluée sur σ.

Nous reportons brièvement cette nouvelle sémantique dans laquelle on note $\sigma, \sigma' \vdash E \mid v$ le fait que l'expression *E* prend la valeur *v* sur toute trace dont les deux dernières mémoires sont, dans l'ordre, σ et σ' [Rat92] :

- k étant une constante et *k* sa valeur,
  $\sigma, \sigma' \vdash k \mid k$
- x étant une variable,
  $\sigma, \sigma' \vdash x \mid \sigma'(x)$

- $*$ dénotant tout opérateur n-aire ($n \geq 1$) arithmétique, booléen ou conditionnel, et E_i étant des expressions quelconques,
$$\frac{\sigma, \sigma' \vdash E_1 \mid v_1, ..., \sigma, \sigma' \vdash E_n \mid v_n}{\sigma, \sigma' \vdash *(E_1, ..., E_n) \mid *(v_1, ..., v_n)}$$

- E_1 et E_2 dénotant deux expressions du langage et $\perp$ dénotant une mémoire indéfinie,
$$\frac{\perp, \sigma' \vdash E_1 \mid v_1}{\perp, \sigma' \vdash (E_1 \text{ -> } E_2) \mid v_1} \qquad \frac{\sigma, \sigma' \vdash E_2 \mid v_2}{\sigma, \sigma' \vdash (E_1 \text{ -> } E_2) \mid v_2}$$

- x étant une variable et *nil* étant une valeur indéterminée,
$\perp, \sigma' \vdash \textbf{\textit{pre}}(x) \mid nil$ $\sigma, \sigma' \vdash \textbf{\textit{pre}}(x) \mid \sigma(x)$

La même simplification s'applique aux règles de compatibilité d'un programme avec les équations et leur composition ainsi qu'avec les assertions :

- Compatibilité avec les équations :
$$\frac{\sigma, \sigma' \vdash E \mid v, \sigma'(x) = v}{\sigma, \sigma' \vdash x = E}$$
$$\frac{\sigma, \sigma' \vdash P_1 \mid v_1, \sigma, \sigma' \vdash P_2 \mid v_2}{\sigma, \sigma' \vdash P_1; P_2}$$

- Compatibilité avec les assertions :
$$\frac{\sigma, \sigma' \vdash A \mid \textbf{\textit{true}}}{\sigma, \sigma' \vdash \textbf{\textit{assert}}\ A}$$

Ainsi, à tout programme P nous pouvons associer un modèle consistant en un système de transitions dont les états sont les mémoires σ et dont la relation de transition $\rightarrow$ est définie comme suit :

$$\forall\sigma, \forall\sigma', \sigma \rightarrow \sigma' \Leftrightarrow \sigma, \sigma' \vdash P$$

Le nombre d'états de ce système est théoriquement infini. En effet, les variables du programme étant éventuellement numériques (entières ou réelles) le nombre de leurs valeurs est infini. Il en est donc de même pour le nombre de mémoires (i.e le nombre d'états).

Un moyen d'en extraire un modèle fini est de considérer une *abstraction booléenne* des états en ne retenant que la *mémoire booléenne* du programme. Plus précisément, on considère qu'une mémoire σ est un couple $(\sigma_B, \sigma_{\overline{B}})$ où σ_B, $\sigma_{\overline{B}}$ sont respectivement une mémoire booléenne (i.e. fonction définie uniquement sur les variables booléennes) et une mémoire non booléenne du programme. Les états du nouveau modèle abs-

trait sont les mémoires booléennes σ_B de *P.* Une variable booléenne n'ayant que deux valeurs possibles, le nombre d'états de ce modèle est fini. La nouvelle relation de transition $\rightarrow_B$ est définie comme suit :

$$\forall\sigma_B\ \forall\sigma_B'\ (\sigma_B \rightarrow_B \sigma_B' \Leftrightarrow \exists\sigma_{\overline{B}}\ \exists\sigma_{\overline{B}}'\ (\sigma_B, \sigma_{\overline{B}}) \rightarrow (\sigma_B', \sigma_{\overline{B}}'))$$

Le système de transitions ainsi obtenu à partir d'un programme *P* est une machine d'états finis $\mathcal{M} = (Q, E, S, q_{init}, a, s, t)$. V_E et V_S dénotant respectivement les ensembles $\{0, 1\}^{|E|}$, $\{0, 1\}^{|S|}$, cette machine est telle que :

- E est l'ensemble des variables d'entrée de *P*,
- S est l'ensemble des variables de sortie de *P*,
- Q est l'ensemble des états du modèle consistant en l'ensemble des mémoires booléennes de *P*,
- $q_{init} \in Q$ est l'état initial,
- $a : Q \times V_E \rightarrow \{0, 1\}$ est la fonction d'assertion dont le but est d'identifier les états et les entrées conformes aux propriétés d'environnement spécifiées au moyen de l'opérateur ***assert***,
- $s : Q \times V_E \rightarrow V_S$ est la fonction de sortie qui permet de calculer la valeur des variables de sortie du programme,
- $t : Q \times V_E \rightarrow Q$ est la fonction de transition, telle que
  $\forall(q, q') \in Q^2\ \forall e \in V_E\ (t(q, e) = q' \Leftrightarrow (q'(E) = e \wedge q, q' \vdash P)$

1.5.3 Compilation

Le compilateur LUSTRE [Ray91] produit, pour tout programme *P*, un code exécutable implantant un automate représentant le modèle d'exécution abstrait de *P* ou, en d'autres termes, la *structure de contrôle* de *P.* Une minimisation de cet automate peut ensuite être opérée consistant à regrouper des états *équivalents* en un seul état. Deux états q_1 et q_2 sont considérés comme équivalents s'ils calculent leurs états successeurs et les sorties du programme de manière identique.

Pour certains programmes, les modèles associés ont un nombre d'états trop important, ce qui rend leur génération impossible en pratique. Pour pallier cet inconvénient, un algorithme de génération de *modèles minimaux* [BFH90] a été adapté à LUSTRE. Cet algorithme permet la construction d'un automate directement minimal (génération *dirigée par la demande*) [Ray91]. Il considère qu'au départ, tous les états de l'automate constituent une seule classe d'équivalence (selon le critère énoncé plus haut). Cette classe est ensuite divisée si elle contient des états qui diffèrent sur le calcul de la sortie ou sur le calcul de leurs successeurs. Deux nouvelles sous-classes sont ainsi

créées, la première contenant les états dont le calcul des sorties et des successeurs est identique, la seconde contenant tous les autres états. Le même procédé est successivement appliqué à la deuxième classe et ainsi de suite jusqu'à stabilité des classes obtenues. L'ensemble d'états du modèle final est l'ensemble des classes d'équivalence résultant de ce processus de génération.

Les ensembles d'états sont représentés par leur fonction caractéristique ce qui permet de mettre en œuvre l'algorithme ci-dessus de manière efficace, en utilisant des techniques symboliques. Il arrive cependant que, malgré l'utilisation de cet algorithme, le modèle ne puisse pas être entièrement engendré en particulier quand la taille du modèle minimal est explosive ou quand les manipulations de fonctions nécessaires au calcul symbolique des classes d'équivalence d'états sont trop gourmandes en espace mémoire ou en temps d'exécution. Dans ce cas, le compilateur LUSTRE propose la génération d'un modèle comportant un seul état. Cela revient à produire simplement une boucle infinie traduisant de manière triviale le traitement effectué par le programme. Ce mode de génération produit un code qui est loin d'être optimal du point de vue du temps d'exécution. En effet, l'absence d'états nécessite la vérification de nombreuses conditions supplémentaires par la fonction de transition qui devient, de ce fait, beaucoup plus complexe. Il a néanmoins le mérite de produire un code de taille très réduite et de permettre ainsi la compilation de pratiquement tout programme LUSTRE indépendamment de la taille de son modèle associé.

La fonction d'assertion est utilisée pendant la compilation pour minimiser la taille du modèle produit. En effet, les transitions rendant fausse cette fonction sont supprimées de cet automate. Notons, cependant, que parfois la prise en compte de la fonction d'assertion entraîne une augmentation de la taille du modèle. Dans ce cas, le compilateur ignore cette fonction [Ray91].

1.5.4 Vérification formelle de propriétés de sûreté

Nous avons déjà évoqué la possibilité d'utiliser LUSTRE comme une logique temporelle dans le but d'exprimer des propriétés de sûreté que doit respecter un programme réactif *P* (cf. paragraphe 1.4.7.2). Cette particularité permet de prouver simplement la validité de ces propriétés dans *P.* Il suffit pour cela de construire un nouveau programme *P'* incluant à la fois *P,* les propriétés de sûreté qu'il doit vérifier et les assertions sur son environnement. L'unique sortie booléenne du programme *P'* ainsi construit est la conjonction des propriétés de sûreté.

Exemple 1-3

Dans le cas du logiciel de contrôle de climatiseur présenté au paragraphe 1.4, le programme de vérification associé est donné dans la figure 1–7 (nœud *VerifClim*).

Le processus de vérification formelle, réalisé par l'outil LESAR [Glo89] [Rat92] peut se faire par deux méthodes différentes communément appelées "en avant" et "en arrière". Quelle que soit l'approche utilisée, LESAR est capable, en cas de non satisfac-

```
node VerifClim(
    Marche,                 -- position de l'interrupteur
    TempInf,                -- temp. ambiante < thermostat
    TempOK,                 -- temp. ambiante. = thermostat
    TempSup : bool)         -- temp. ambiante. > thermostat
returns(
    ok : bool);             -- vrai si les propriétés de sûreté
                            -- sont respectées
var   ARRETE,               -- hors tension
      INACTIF,              -- sous tension, pas de diffusion
      CHAUD,                -- diffusion d'air chaud
      FROID: bool           -- diffusion d'air froid
let
    -- Hypothèses sur l'environnement
    assert ((TempSup or TempOK or TempInf) and
            #(TempSup, TempOK, TempInf));
    assert(once_from_to(TempOK, TempInf, TempSup) and
            once_from_to(TempOK, TempSup, TempInf));

    -- Calcul de la validité des propriétés de sûreté
    ok = (not ARRETE or not Marche) and
         (not INACTIF or Marche and TempOK) and
         (not CHAUD or Marche and TempInf) and
         (not FROID or Marche and TempSup) and
         (not pre (Marche and TempInf) or (pre CHAUD or CHAUD) ) and
         (not pre (Marche and TempSup) or (pre FROID or FROID);

    -- Appel du nœud Clim
    (ARRETE , INACTIF, CHAUD, FROID) =
       Clim(Marche, TempInf, TempOK, TempSup);
tel;
```

Figure 1–7 : Nœud de vérification du logiciel de contrôle du climatiseur

tion des propriétés de sûreté, de fournir une séquence d'entrées menant le programme dans un état de violation de ces propriétés.

1.5.4.1 La méthode en avant

La vérification en avant d'un modèle peut elle-même suivre deux approches, appelées respectivement "classique" et "à la volée".

- L'approche classique consiste à explorer de manière exhaustive les transitions du modèle depuis l'état initial et de mémoriser l'ensemble des états accessibles

ainsi que les transitions permettant d'y accéder. Pour chaque état $q' \in Q$ accédé depuis l'état $q \in Q$ par la transition $t(q, e)$, le triplet $(q, e, q') \in Q \times V_E \times Q$ est mémorisé si bien qu'à la fin de l'exploration on se retrouve avec un ensemble de triplets décrivant la totalité des traces exécutables du modèle. Il suffit alors d'évaluer la valeur de la fonction de sortie $s(q, e)$ pour chaque triplet (q, e, q') de cet ensemble qui doit être toujours vraie pour que le programme vérifie les propriétés.
L'inconvénient de cette approche est que si le nombre d'états du modèle est élevé, l'ensemble de triplets ainsi construit risque d'être prohibitif.

- L'approche "à la volée" cherche à minimiser le nombre d'informations mémorisées à l'issue de l'exploration du modèle. Elle consiste à évaluer la valeur de la fonction de sortie $s(q, e)$ par les transitions au moment de la génération du modèle. Cela répond à l'objectif de minimisation car il suffit de ne mémoriser que les états déjà explorés. Bien que cette deuxième approche soit plus performante que l'approche classique, elle n'offre pas une solution au problème de l'explosion du modèle qu'il faut également explorer de manière exhaustive.

Notons qu'il existe deux techniques d'implantation possibles de ces deux approches. La première est énumérative et consiste à représenter explicitement les ensembles manipulés. Sa mise en œuvre est très simple mais ne peut être appliquée que pour des ensembles de taille réduite. La deuxième technique représente les ensembles de manière symbolique c'est à dire à l'aide de leur fonction caractéristique. La taille des ensembles représentés peut dans ce cas être beaucoup plus importante.

1.5.4.2 La méthode en arrière

La méthode de vérification en arrière repose sur l'utilisation de l'algorithme de génération de modèle minimal dont le principe d'application à la compilation a été exposé au paragraphe 1.5.3. Dans le cadre de la vérification, il suffit d'appliquer ce même algorithme au nœud de vérification *P'*. Cependant, cette fois il ne s'agit pas de construire toutes les classes d'équivalence mais d'éliminer celles dont les états violent les propriétés de sûreté ou bien mènent dans un tel état.

Plus précisément, la seule sortie de *P'* étant la conjonction des propriétés de sûreté, nous obtenons, après division de la classe d'équivalence initiale, deux sous-classes : une contenant les états où la sortie du programme prend la valeur vraie et une autre contenant les états où la sortie est fausse. La classe retenue est la première qui est de nouveau divisée en deux parties comportant respectivement les états dont les successeurs sont dans ou en dehors de la classe. Les états de la deuxième partie étant éliminés, on obtient une classe contenant tous les états satisfaisant les propriétés de sûreté et dont les successeurs les satisfont également. Ce principe de division est appliqué de nouveau et ainsi de suite jusqu'à obtention d'une classe fermée. Il suffit alors de vérifier que l'état initial du modèle figure dans cette classe, auquel cas le programme vérifie les propriétés de sûreté.

1.5.4.3 Utilisation de la fonction d'assertion

Quelle que soit la méthode de vérification utilisée, la fonction d'assertion est prise en compte dans le but d'éliminer toute transition qui ne la vérifie pas. Ainsi, les violations des propriétés de sûreté dues à des valeurs d'entrées incohérentes avec la spécification de l'environnement sont toujours identifiées et ignorées (par opposition au processus de compilation, présenté au paragraphe 1.5.3, qui ne tient compte des assertions que quand en résulte une minimisation du code produit).

1.5.5 Représentation symbolique du modèle d'exécution

1.5.5.1 Encodage du modèle à l'aide de fonctions booléennes

Comme nous l'avons vu au paragraphe 1.5.2, le modèle associé à un programme LUSTRE est une machine d'états finis dont les états sont les mémoires booléennes du programme. Un moyen de représenter de manière compacte ce modèle est de considérer que les états sont définis à l'aide de *variables d'état*, qui sont des variables booléennes définies de la manière suivante [Ray91] [Rat92] :

- Une variable d'état est associée à chaque expression ***pre*** *x* distincte du programme, où *x* est une variable booléenne.
- Une variable d'état supplémentaire caractérise l'état initial du modèle. Sa valeur est vraie à l'état initial et fausse à tout autre état.

Un état est associé à chaque valeur distincte du vecteur de variables d'état $(v_i)_{i = 1, n}$ ainsi obtenu. La fonction de transition est définie comme un vecteur $(t_i)_{i = 1, n}$ de fonctions de transition partielles telles que chaque fonction t_i est associée à la variable d'état v_i. Etant donnés l'état courant (i.e. la valeur de variables d'état $(v_i)_{i = 1, n}$) et les valeurs des variables d'entrée, t_i associe à la variable d'état v_i la valeur que celle-ci prendra à l'état suivant.

Cette représentation du modèle, bien que compacte, n'implique aucune perte d'information par rapport au modèle original. Elle consiste simplement à ne conserver que les informations nécessaires au calcul des transitions et des sorties. En effet, la totalité des variables du programme sont définies du moment que les entrées et les variables d'état le sont. Il en est de même pour la fonction de transition.

En résumé, le modèle d'un programme est décrit par un ensemble de fonctions booléennes (fonctions de transition, de sortie et d'assertion) portant sur les variables d'état et les entrées. La manipulation de ces fonctions, aussi bien dans le cadre de la vérification formelle que de la compilation, est largement facilitée quand leur représentation est faite sous forme de *graphes de décision binaires* (souvent désignés par leur acronyme en anglais *BDD : "Binary Decision Diagrams"*) [Ake78] [Bry86]. Cette structure, que nous avons également utilisée dans le cadre de notre travail, per-

met une représentation canonique des fonctions et rend l'implantation des opérateurs booléens très efficace. Nous en donnons ici un bref aperçu.

1.5.5.2 Représentation des fonctions booléennes en BDD

Expansion de Shannon des fonctions booléennes

Le résultat fondamental sur lequel sont fondés les graphes de décision binaires est le *théorème d'expansion* de Shannon [Sha38].

Soit $f : (\{0, 1\})^n \rightarrow \{0, 1\}$ une fonction booléenne à n variables. L'expansion de Shannon de f par rapport à la i-ème variable x_i ($1 \leq i \leq n$) consiste en un couple de fonctions (f_{x_i}, $f_{\overline{x_i}}$) :

$$f_{x_i} : (\{0, 1\})^{n-1} \rightarrow \{0, 1\} / f_{x_i} = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$$
$$f_{\overline{x_i}} : (\{0, 1\})^{n-1} \rightarrow \{0, 1\} / f_{\overline{x_i}} = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$$

La fonction f_{x_i} (resp. $f_{\overline{x_i}}$) est le *cofacteur de f par rapport à* x_i (resp. $\overline{x_i}$) tandis que x_i est la *variable d'expansion*. La fonction f peut être exprimée en fonction de ses deux cofacteurs :

$$f = x_i \wedge f_{x_i} \vee \overline{x_i} \wedge f_{\overline{x_i}}$$

Le théorème de Shannon garantit que pour une variable d'expansion donnée x_i, les cofacteurs (f_{x_i}, $f_{\overline{x_i}}$) de f sont *uniques*. Il en résulte que si deux fonctions $f : (\{0, 1\})^n \rightarrow \{0, 1\}$ et $g : (\{0, 1\})^n \rightarrow \{0, 1\}$ ont les mêmes cofacteurs par rapport à une même variable x_i, elles sont identiques.

Le principe d'expansion peut s'appliquer aux cofacteurs de f (par rapport, bien entendu, à une autre variable d'expansion) et ainsi de suite jusqu'à obtention de cofacteurs constants (de domaine de définition vide). On obtient ainsi une *expansion complète* de f.

Malheureusement, l'expansion complète d'une fonction n'est pas unique. Il est cependant possible de définir une expansion complète *canonique*. Pour cela, on définit un *ordre d'expansion* sur les variables de f.

Soit une permutation p sur $(1, ..., n)$. L'ordre d'expansion $<_p$ est défini comme suit :

$$x_i <_p x_j \Leftrightarrow p(i) < p(j)$$

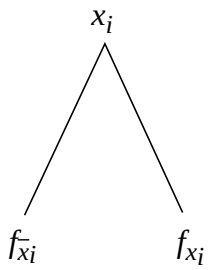

Figure 1-8 : Arbre de Shannon

L'expansion de Shannon canonique d'une fonction $f : (\{0, 1\})^n \rightarrow \{0, 1\}$ pour un ordre $<_p$ est obtenue en choisissant la même variable $x_{p(i)}$ pour l'expansion de tous les cofacteurs définis sur $\{0, 1\}^i$.

L'expansion de toute fonction booléenne f peut être représentée par un arbre (*arbre de Shannon*) dont la racine est associée à la variable d'expansion et dont les fils sont associés aux cofacteurs. Par convention, le fils droit sera associé au cofacteur par rapport à x_i et le fils gauche sera associé au cofacteur par rapport à $\bar{x}_i$, comme le montre la figure 1-8. L'expansion complète d'une fonction f peut ainsi être représenté par un arbre de Shannon dont toutes les feuilles sont *0* ou *1*.

Exemple 1-4

Considérons la fonction booléenne $f : (\{0, 1\})^3 \rightarrow \{0, 1\}$ définie comme suit :

$$f(x_1, x_2, x_3) = x_1 \wedge x_2 \vee x_2 \wedge x_3 \vee x_1 \wedge x_3$$

Soit l'ordre d'expansion $<_p$ tel que $p(i) = i$ pour $1 \leq i \leq 3$. L'expansion de f par rapport à x_1 est donnée par :

$$f = x_1 \wedge f_{x_1} \vee \bar{x}_1 \wedge f_{\bar{x}_1} \text{ où } f_{x_1} = x_2 \vee x_3 \text{ et } f_{\bar{x}_1} = x_2 \wedge x_3$$

L'expansion des cofacteurs f_{x_1} et $f_{\bar{x}_1}$ par rapport à x_2 est :

$$f_{x_1} = x_2 \wedge (f_{x_1})_{x2} \vee \bar{x}_2 \wedge (f_{x_1})_{\bar{x}2} \text{ où } (f_{x_1})_{x2} = 1 \text{ et } (f_{x_1})_{\bar{x}2} = x_3$$
$$f_{\bar{x}_1} = x_2 \wedge (f_{\bar{x}_1})_{x2} \vee \bar{x}_2 \wedge (f_{\bar{x}_1})_{\bar{x}2} \text{ où } (f_{\bar{x}_1})_{x2} = x_3 \text{ et } (f_{\bar{x}_1})_{\bar{x}2} = 0.$$

Enfin,

$$((f_{x_1})_{\bar{x}2})_{x3} = 1, ((f_{x_1})_{\bar{x}2})_{\bar{x}3} = 0, ((f_{\bar{x}_1})_{x2})_{x3} = 1, ((f_{\bar{x}_1})_{x2})_{\bar{x}3} = 0$$

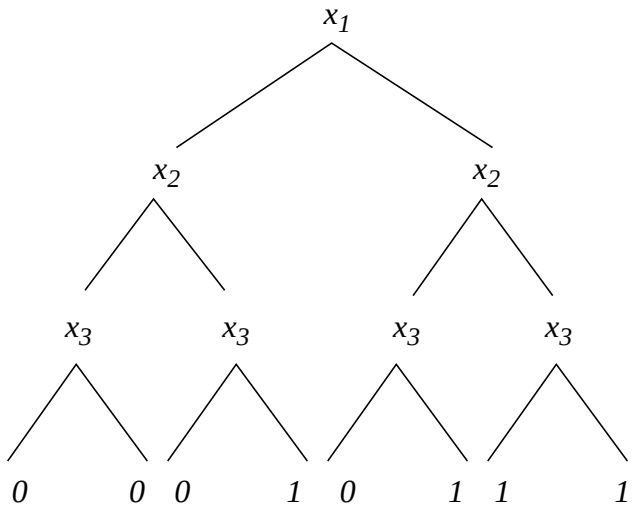

Figure 1-9 : Arbre de Shannon de $x_1 \wedge x_2 \vee x_2 \wedge x_3 \vee x_1 \wedge x_3$

L' arbre de Shannon associé à l'expansion complète de *f* suivant l'ordre d'expansion $<_p$ est donné par la figure 1-9.

Les graphes de décision binaires (BDD)

Le nombre de nœuds d'un arbre de Shannon est exponentiel par rapport au nombre de variables de la fonction. Les graphes de décision binaires, que nous noterons souvent dans la suite BDD par souci de brièveté, est une structure semblable à l'arbre de Shannon mais de taille plus réduite. Un BDD est obtenu à partir d'un arbre de Shannon au moyen de deux opérations, l'*élimination des nœuds redondants* et le *partage des sous arbres isomorphes* :

- L'élimination des nœuds redondants consiste à remplacer tout nœud dont les fils sont identiques par un de ses fils. Dans le cas de l'exemple 1-4 on obtient, après élimination des nœuds redondants, le graphe de la figure 1-10.
- Le partage des sous arbres isomorphes consiste à ne stocker en mémoire qu'une seule fois les nœuds représentant la même expansion. Ainsi, pour la fonction de l'exemple 1-4 nous obtenons le BDD de la figure 1-11.

Bien que ces deux opérations permettent, en général, de réduire considérablement le nombre des nœuds du BDD, il faut rappeler que sa taille dépend de l'ordre d'expansion considéré (la taille d'un BDD représentant une fonction booléenne *f* à *n* variables est potentiellement exponentielle : $O(2^n/n)$ [Cou91]). Néanmoins, les BDD s'avèrent être, en pratique, une représentation plutôt compacte. Dans [Rat92] on trouve des heuristiques permettant d'optimiser la taille des graphes obtenus en intervenant sur l'ordre des variables d'expansion.

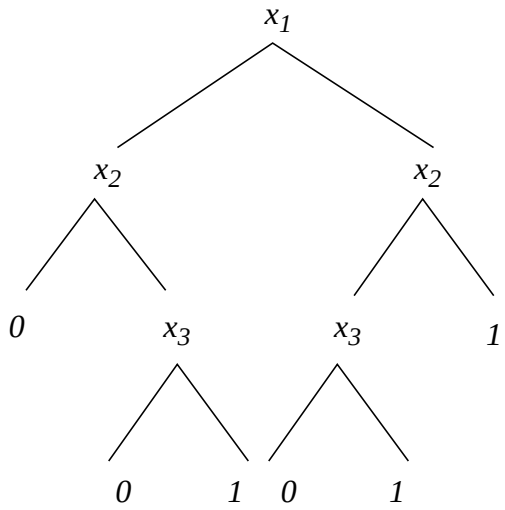

Figure 1-10 : Elimination des nœuds redondants

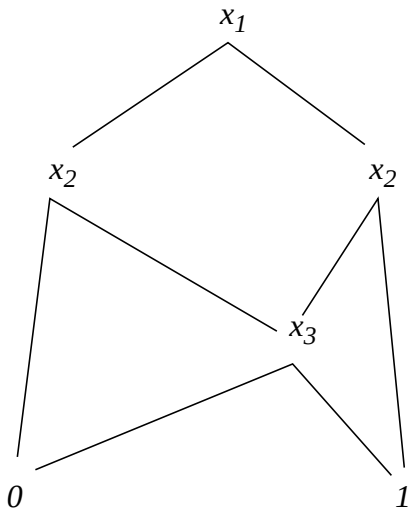

Figure 1-11 : Partage des sous-arbres isomorphes

Tous les opérateurs booléens habituels peuvent être définis sur les graphes de décision binaires. L'efficacité de leur implantation exige une bonne gestion de la mémoire utilisée (association de caches aux opérateurs, gestion des caches, ...) [Cou91] [Rat92]. Les détails de cette implantation débordant du cadre de notre travail, nous nous contentons de fournir certains opérateurs utiles avec leur complexité algorithmique respective. Dans le tableau de la figure 1-12 [Rat92] *f*, *g* et *h* sont des fonctions booléennes. La taille des graphes de décision binaires (i.e. le nombre de leurs nœuds distincts) qui leur sont associés sont respectivement $|f|$, $|g|$ et $|h|$. Enfin, dans la définition des opérateurs de quantification (i.e. deux dernières lignes du tableau de la figure 1-12) *x* désigne une variable de *f* tandis que *X* désigne l'ensemble des variables de *f*.

Opération	*Coût*
$f = g, f \neq 0, f = 1$	$O(1)$
$f \Rightarrow g, f \wedge g, f \vee g$	$O(\|f\| \times \|g\|)$
Evaluation de $f(x_1, ..., x_n)$	$O(n)$
$\neg f$	$O(\|f\|)$
si f ***alors*** g ***sinon*** h	$O(\|f\| \times \|g\| \times \|h\|)$
$\exists x\, f, \forall x\, f$	$O(\|f\|^2)$
$\exists X\, f, \forall X\, f$	$O(2^{\sqrt{\|f\|}})$

Figure 1-12 : Coût des opérateurs booléens sur les BDD

Dans la suite du document nous éviterons parfois de prendre en compte la totalité des partages de sous-arbres isomorphes dans la représentation graphique des BDD. Cela sera fait uniquement dans un souci de lisibilité et ne constitue en aucune manière une suggestion de modification de la structure réelle des graphes.

2

Test des logiciels synchrones

2.1 INTRODUCTION

Nous abordons maintenant la partie principale de notre travail qui consiste en la définition de techniques de test pour les logiciels synchrones. Avant de présenter les techniques que nous avons développées, nous situons notre problématique par rapport au thème plus général de la sûreté de fonctionnement de ces logiciels. En particulier, nous identifions certains besoins auxquels la technique de vérification formelle actuellement disponible ne peut apporter une réponse satisfaisante et que le test peut satisfaire sous certaines conditions.

Après avoir justifié l'intérêt du test, nous consacrons une partie du chapitre à l'exposé synthétique des principaux travaux théoriques et empiriques sur le test, dont nous nous sommes inévitablement inspirés lors de la conception de nos propres techniques. Ces dernières sont présentées ensuite de manière sommaire, leur développement détaillé faisant l'objet des trois chapitres suivants.

2.2 TEST ET SÛRETÉ DE FONCTIONNEMENT DES LOGICIELS

2.2.1 Sûreté de fonctionnement

La propriété qui permet d'avoir confiance dans le service délivré par un système informatique est sa *sûreté de fonctionnement* [Lap95] (les termes *crédibilité* [PvSPK90] et *correction probable* [Ham95] désignent cette même propriété). La sûreté de fonctionnement dépend de plusieurs facteurs dont l'importance varie avec le domaine d'utilisation du logiciel. Ces facteurs sont les *attributs* de la sûreté de fonctionnement :

- La *disponibilité* exprime le fait que le logiciel est prêt à l'utilisation.

- La *fiabilité* mesure la continuité du service assuré.
- La *sécurité-innocuité* requiert l'absence d'occurrences de comportements particulièrement dangereux, dans le but d'éviter les risques de catastrophe.
- La *confidentialité* et *l'intégrité* expriment l'absence respectivement de divulgations non autorisées et d'altérations non appropriées d'informations (ces deux attributs sont parfois regroupés sous le nom de *sécurité-confidentialité*).
- La *maintenabilité* correspond à l'aptitude du logiciel aux réparations et aux évolutions.

Dans le domaine des logiciels réactifs, un de ces attributs est dominant, la *sécurité-innocuité* [Lap93] ou *sûreté* [Lev90]. Les contraintes de sûreté d'un système réactif consistent souvent en un ensemble de spécifications imposant l'absence de comportements dangereux et peuvent de ce fait s'opposer à certaines des spécifications fonctionnelles du système [Lev91]. Leveson [Lev86] suggère que la sûreté soit différenciée de la fiabilité et de la sécurité-confidentialité. En effet, les contraintes de fiabilité ont comme objectif ultime l'obtention d'un logiciel sans risque de défaillance tandis que les exigences de sûreté sont satisfaites par un logiciel qui ne présente pas de risque d'incident susceptible de provoquer une catastrophe. Quant au concept de sécurité-confidentialité, bien que beaucoup plus proche de celui de la sûreté, il a principalement pour objet des actions malveillantes (par exemple des intrusions) tandis que la sûreté est également concernée par des actions accidentelles. Il est par ailleurs évident que la disponibilité n'a, en général, qu'un faible rapport avec la sûreté, puisque dans plusieurs cas l'absence totale de disponibilité peut être une garantie de sûreté ("l'avion le plus sûr est celui qui ne quitte jamais le sol"). Citons, enfin, la perception différente de Parnas [PvSPK90] qui ne considère pas la sûreté comme un attribut à part, mais plutôt comme un but à atteindre par l'amélioration de la fiabilité du logiciel.

Selon la terminologie de Laprie [Lap92], une *défaillance* correspond à une déviation du service assuré par le logiciel de la fonction supposée de ce dernier. Les défaillances d'un logiciel peuvent être de gravité variée, chaque niveau de gravité étant appelé *mode de défaillance*. La *criticité* d'un logiciel correspond à son mode de défaillance le plus élevé. On qualifie de *critiques* les logiciels de criticité très élevée, c'est à dire ceux dont la défaillance peut avoir un coût supérieur au bénéfice qu'apporte leur utilisation.

Suivant cette même terminologie, une *erreur* est la partie de l'état du logiciel susceptible d'entraîner une défaillance. Typiquement, une erreur est la prise d'une mauvaise valeur par une variable ou expression du programme ou par le compteur d'instructions.

Enfin, une *faute* ou un *défaut* est la cause supposée d'une erreur dans le programme (par exemple une instruction manquante ou erronée).

La sûreté de fonctionnement d'un logiciel peut être obtenue à l'aide de *moyens* qui, en fonction de la manière dont ils traitent les fautes, sont classés en quatre catégories :

- La *prévention des fautes* a comme but d'empêcher l'introduction des fautes dans le logiciel.
- L'*élimination des fautes* regroupe les moyens ayant comme objectif de réduire le nombre et la gravité des fautes présentes dans le logiciel.
- La *tolérance aux fautes* concerne les mécanismes permettant au logiciel de continuer à assurer un service malgré la présence de fautes.
- La *prévision des fautes* a pour objet l'estimation de la présence, de la création et des conséquences des fautes.

Des techniques diverses sont actuellement proposées dans le cadre des trois dernières catégories. Aucune technique particulière n'est, au contraire, consacrée à la prévention des fautes. En effet, cette dernière concerne l'ensemble des méthodes et techniques utilisées dans le cadre du développement des logiciels (méthodes et langages de spécification, logistique, ...).

2.2.2 Sûreté de fonctionnement des logiciels synchrones

Dans le cadre de ce travail nous nous sommes intéressés à la définition de nouveaux moyens d'élimination de fautes dans les logiciels synchrones. La preuve formelle des propriétés de sûreté sur les programmes LUSTRE, telle qu'elle a été présentée dans le chapitre 1, offre un moyen d'élimination de défauts incontestablement utile mais ne suffit pas à elle seule à assurer l'élimination de tous les défauts d'un logiciel synchrone pour les raisons suivantes :

- La taille du modèle du programme nécessaire pour la preuve peut être prohibitive. Dans ce cas, la preuve échoue et aucune information sur la validité des propriétés n'est fournie.
- La preuve a comme seul but de montrer la conformité du programme à une spécification formelle de propriétés. Or, une défaillance est définie par rapport à la fonction supposée du logiciel et non pas par rapport à sa spécification [Lap95]. Ainsi, même si la preuve réussit, nous ne pouvons pas affirmer que le logiciel répond bien à nos besoins réels; il ne satisfait que ceux exprimés par les propriétés, qui peuvent être elles-mêmes erronées.
- La preuve ne s'intéresse qu'aux fautes relatives aux propriétés de sûreté. Il est donc nécessaire de disposer de moyens de détection d'autres types de fautes.
- La preuve nécessite que le logiciel soit entièrement implanté en LUSTRE.

Ces remarques peuvent s'appliquer à la plupart des techniques de preuve automatique, en dehors du seul cadre de LUSTRE; elles mettent bien en évidence leur incapacité de répondre de manière satisfaisante au défi consistant à développer des logiciels sûrs de fonctionnement. Cela semble être confirmé par le fait que le moyen de vérification et

de validation le plus utilisé aujourd'hui dans le monde industriel reste le *test des logiciels* [Bei90]. Ainsi, nous nous sommes tournés vers le test pour rechercher des techniques de vérification et de validation capables d'apporter un complément à la preuve formelle.

2.3 TEST DES LOGICIELS : UN BREF ÉTAT DE L'ART

2.3.1 Définition

Contrairement à la preuve, le test ne peut pas démontrer la correction d'un logiciel mais vise à *découvrir des défauts* dans son implantation [Mye79]. Cependant, il est incontestable que ce faisant, le test permet d'évaluer et d'améliorer la sûreté de fonctionnement du logiciel [Ham94] [Ham95].

D'une manière générale, tester un logiciel consiste à l'exécuter en ayant la totale maîtrise des données qui lui sont fournies en entrée tout en vérifiant que son comportement est celui attendu. Cette définition met en évidence deux tâches distinctes nécessaires à l'activité de test :

- La constitution d'un ensemble de données qui seront fournies en entrée au logiciel à tester (i.e des *jeux de test*) suivant un certain *critère.*
- L'observation de l'exécution du logiciel, qui doit permettre l'identification des défaillances.

Il faut noter que la définition du test que nous avons donnée n'inclut pas deux autres activités nécessaires à l'élimination des défauts, à savoir leur localisation et leur correction (cf. paragraphe 2.4.5).

2.3.2 Construction des jeux de test

2.3.2.1 Critères de sélection, d'adéquation et d'arrêt

Etant donné que la prise en compte de la totalité du domaine d'entrée du logiciel (*test exhaustif*) est presque toujours impossible, un ensemble de jeux de test doit être un sous-domaine de taille réaliste. Idéalement, un critère doit définir un tel sous-domaine dont le pouvoir de révélation de défauts est identique à celui du test exhaustif.

La construction d'un ensemble de jeux de test conforme à un critère peut être effectuée suivant deux approches. La première consiste en la sélection à l'avance des éléments de cet ensemble suivant le critère qui est dans ce cas appelé *critère de sélection.* La deuxième approche est de considérer un jeu de test quelconque et d'évaluer *a posteriori* sa conformité au critère qui est dans ce cas appelé *critère d'adéquation.* Concrètement, cette dernière approche revient à tester le logiciel avec des données quelconques en entrée jusqu'à ce que le critère soit satisfait (pour cette raison on parle parfois également de *critère d'arrêt*) [WWH91]. Dans la suite, nous utiliserons le

terme "critère de sélection" dans tout contexte où la manière de construire l'ensemble de jeux de test est sans importance.

Le *test aléatoire* est souvent assimilé au critère le plus élémentaire puisque les jeux de test sont sélectionnés au hasard. Dans les autres cas, un critère de sélection définit généralement une décomposition du domaine d'entrée du logiciel en un nombre fini de sous-domaines. Bien que ces sous-domaines ne soient pas toujours disjoints, on parle souvent de "partition" du domaine des entrées. Le pouvoir de détection de défauts des éléments d'un même sous-domaine est considéré équivalent, ce qui signifie qu'un seul élément du sous-domaine met en évidence tous les défauts que peut détecter le sous-domaine entier. Plusieurs critères de cette nature ont été proposés dans la littérature pendant ces vingt dernières années dont nous donnons un aperçu synthétique plus loin dans ce paragraphe. Il faut cependant noter que l'efficacité des critères fondés sur une partition du domaine d'entrée est contestée car, dans de nombreux cas, le nombre de défauts détectés ne dépasse pas celui obtenu en testant le logiciel de manière purement aléatoire [HT90]. Cela est dû, entre autres, à la difficulté de choisir de manière judicieuse un élément représentant chaque sous-domaine [Mül94].

2.3.2.2 Evaluation théorique des critères de sélection

Lors d'une des premières tentatives d'établir une théorie pour le test des logiciels, Goodenough et Gerhart [GG75] se sont intéressés aux propriétés que doivent posséder les critères de sélection. Ainsi, ils qualifient un critère de *fiable* si tous les jeux de test conformes à ce critère détectent les mêmes défauts. Il sera, de plus, dit *valide* si tout défaut du logiciel peut être détecté par au moins un jeu de test conforme au critère. En pratique, malheureusement, nous n'avons aucun moyen de montrer qu'un critère de sélection est fiable ou valide ou de concevoir un critère possédant ces deux qualités (cela nécessiterait, en effet, la connaissance de tous les défauts du logiciel). Néanmoins, ces deux attributs expriment bien les propriétés qu'on souhaite voir vérifiées par tout critère de sélection. Un moyen pratique d'évaluation de la fiabilité et de la validité d'un critère est présenté au paragraphe 2.3.2.3.

Plusieurs travaux ont suivi depuis dont certains ont tenté de consolider et d'affiner ces premières considérations [Gou83] [How76] tandis que d'autres les ont plutôt critiquées [WO80]. Dans [Gou83] et [Whi81] on trouvera un ensemble de définitions formelles et de résultats théoriques résumant la plupart des travaux de recherche sur ce sujet. Ces travaux ont, en particulier, établi les résultats théoriques fondamentaux suivants :

- Le problème de l'équivalence de deux programmes est non décidable. Une conséquence de ce résultat est que la sélection d'un jeu de test montrant la correction d'un programme est également un problème non décidable.
- La sélection d'un jeu de test activant une partie donnée du programme est un problème non décidable. Ce résultat est particulièrement important pour les techniques de test structurel (cf. paragraphe 2.3.5).

Cependant, dans plusieurs cas particuliers ces problèmes sont décidables. Ainsi, bien que le développement de techniques universelles soit impossible, plusieurs techniques adaptées à ces cas particuliers ont pu voir le jour.

Une autre tentative de construction d'un cadre théorique pour la définition de critères de sélection est celle de Weyuker [Wey86] [Wey88] qui a proposé un ensemble d'*axiomes* que doivent vérifier ces critères. Il s'agit d'une approche pragmatique qui peut être vue comme une spécification minimale que doit satisfaire tout critère. Cependant, bien que certains auteurs aient tenté de l'améliorer [ZG89] [PZ93], cette approche reste imprécise quant à son application et surtout quant à sa pertinence.

Un dernier point intéressant est la possibilité de *comparer* deux critères de sélection entre eux. Le moyen le plus courant de faire une telle comparaison est une *relation d'inclusion* : un critère *X* est inclus dans le critère *Y* si tout jeu de test satisfaisant *Y* satisfait également *X*. Dans le cadre des techniques de test structurelles (cf. paragraphe 2.3.5) une telle classification est donnée dans [Nta88]. Tous les critères n'étant pas comparables, cette relation définit un ordre partiel entre critères, représenté habituellement par un treillis. Notons que l'inclusion d'un critère dans un autre n'implique pas une relation analogue entre leurs pouvoirs de détection de défauts respectifs [WWH91]. Une hypothèse (très forte et souvent non vérifiée) est faite en pratique, consistant à admettre que les critères sont fiables (i.e. tous les jeux de test conformes à un critère ont le même pouvoir de détection de défauts). Sous cette hypothèse, la relation d'inclusion reflète également le pouvoir de détection de défauts des critères.

L'intérêt d'une telle relation d'ordre est qu'elle fournit un moyen de planifier les activités de test. Il est, en effet, inutile de tester en utilisant un critère *X* si un autre critère *Y* l'incluant a déjà été appliqué. Nous pouvons cependant noter un inconvénient : la relation d'inclusion ne tient pas compte du *coût* de la génération des données nécessaires à la satisfaction des critères. Ce coût est fonction de la taille minimale de l'ensemble de jeux de test satisfaisant le critère ou de la difficulté de sa construction.

En pratique, le choix d'un critère de sélection nécessite un compromis entre le pouvoir de détection de défauts présumé du critère et le coût de la construction d'un ensemble de jeux de test le satisfaisant.

2.3.2.3 Evaluation pratique : la technique des mutants

L'absence d'un cadre théorique satisfaisant et uniforme a favorisé le développement d'approches empiriques pour l'évaluation des critères de sélection de jeux de test. Une des méthodes les plus utilisées à ce jour est celle des *mutants* [DLS78]. Suivant cette méthode, des défauts simples sont introduits au logiciel par une opération appelée "mutation" qui consiste en le remplacement d'une constante ou d'une variable par une autre de même type ou bien la modification d'un opérateur arithmétique ou relationnel (par exemple + par - ou < par <=). Chaque défaut ainsi introduit donne naissance à un nouveau programme, appelé "mutant".

Après avoir ainsi constitué un ensemble conséquent de mutants, ces derniers sont exécutés avec le jeu de test dont on veut évaluer l'efficacité et leurs résultats sont comparés à ceux du programme original. Idéalement, le jeu de test devrait provoquer la production de résultats différents par les mutants (on dit alors que les mutants sont "tués"). Dans ce cas, nous admettons que le critère de sélection ayant servi à la construction du jeu d'essai est valide. Toutefois, cette conclusion ne sera acceptable que si plusieurs jeux d'essai issus du même critère de sélection ont fait l'objet de cette procédure d'évaluation.

Quand un mutant n'a pas pu être tué par l'exécution des jeux de test, deux cas se présentent :

- Le mutant est équivalent au programme initial. Cela signifie que la modification apportée était sans conséquence sur le fonctionnement du programme. Cette information peut être intéressante pour le programmeur mais n'a aucune incidence sur la qualité du jeu de test ou du critère utilisé.
- Le mutant et le programme original ne sont pas équivalents (i.e. on peut mettre en évidence une donnée d'entrée produisant des résultats différents pour les deux programmes). Dans ce cas, on peut conclure soit que le critère n'est pas valide (si aucun des jeux de test utilisés n'a pas pu tuer le mutant), soit qu'il n'est pas fiable (si certains des jeux de test seulement l'ont tué).

L'avantage de la technique des mutants est qu'elle est très facilement automatisable, aussi bien en ce qui concerne la production des mutants que leur exécution, bien que les cas des mutants non tués nécessitent une analyse manuelle. En revanche, la pertinence des évaluations des critères de sélection que l'on peut obtenir est conditionnée par la validité de deux principes :

- L'hypothèse du "programmeur compétent" stipule que le programme que l'on teste est "très proche" du programme correct. En d'autres termes, le premier peut être obtenu à partir du second au moyen d'une suite de mutations.
- L'*effet de couplage* est un principe empirique : tout défaut multiple du logiciel est correlé à un défaut simple. Cela permet de construire des mutants contenant une seule mutation par rapport au programme initial, l'effet de couplage garantissant que si un jeu de test détecte un défaut simple, alors il détectera tout défaut multiple le contenant.

Notons que l'hypothèse du programmeur compétent est une hypothèse très forte dont la validité dans le cas général a été contestée [Gou83].

2.3.3 Observation du comportement : le problème de l'oracle

La deuxième tâche nécessaire à l'activité de test est l'observation de l'exécution du logiciel. Elle doit permettre l'identification des comportements erratiques (défaillan-

ces), faute de quoi le test serait inutile. L'observation du comportement du logiciel est un problème délicat, connu sous le nom du "problème de l'oracle".

Dans certains cas, un observateur humain peut accomplir de manière satisfaisante cette fonction d'observation, en particulier quand les défaillances se manifestent par un arrêt du logiciel. Cependant, ce type d'observation n'offre pas une solution satisfaisante au problème de l'oracle, comme l'ont par ailleurs montré des expériences [BS87] pendant lesquelles un nombre de défaillances considérable (de l'ordre de 30%) n'a pas pu être détecté par des observateurs expérimentés. Cette constatation met en évidence la nécessité de moyens automatiques d'observation. Ainsi, on essaie de construire des programmes, appelés "oracles", capables de vérifier que le logiciel sous test se comporte correctement pour les cas d'exécution considérés.

En d'autres mots, un oracle consiste en une spécification partielle du logiciel, portant sur un nombre restreint de cas d'exécution. Plus le nombre de ces cas est important, plus l'oracle sera difficile à construire. Ainsi, le développement d'un oracle reconnaissant tous les comportements corrects du programme serait aussi difficile que celui du programme lui-même.

Notons, enfin, qu'au même titre que les spécifications peuvent être erronées, un oracle peut lui-même contenir des fautes. C'est pour cette raison que nous pensons qu'un oracle n'est qu'un assistant de l'observateur humain et non pas, dans le cas général, son remplaçant. Ce dernier reste, en effet, le dernier rempart contre les spécifications erronées. Dans le paragraphe 2.3.8, nous exposons les techniques les plus courantes de construction d'oracles.

2.3.4 Classification des techniques de test

Plusieurs techniques de test on été proposées ces vingt dernières années. Elles consistent en la définition de critères de sélection, éventuellement accompagnés de moyens de génération des jeux de test associés.

Selon la phase du cycle de développement du logiciel pendant laquelle il est effectué, le test est qualifié d'*unitaire*, d'*intégration* ou de *système*. Le test unitaire concerne chacun des modules qui constituent le logiciel en le traitant isolément : son but est la recherche de défauts dans la réalisation de ce module en ne considérant que la spécification de ce module. L'objectif du test d'intégration, au contraire, est la mise en évidence d'anomalies dans les appels entre modules ou dans la répartition des traitements entre modules pour réaliser les principales fonctions du logiciel. Le test système, enfin, porte sur l'ensemble du logiciel et vise à identifier les cas où sa spécification initiale n'est pas respectée.

Presque tous les travaux de recherche connus sur le test se situent au niveau unitaire. Les techniques qui sont proposées dans ces travaux peuvent être classées en deux catégories, en fonction du modèle du programme sur lequel sont définis leurs critères de sélection :

- Les techniques *structurelles* ou "boîte de verre" considèrent pour la sélection des jeux de test le code source du programme représenté par son *graphe de contrôle*. Les critères de sélection de ce type de techniques se définissent en termes de *couverture* d'éléments définis sur ce graphe.
- Les techniques *fonctionnelles* ou "boîte noire" sélectionnent des jeux de test en s'appuyant uniquement sur une spécification externe du logiciel, informelle ou formelle. Quand cette spécification est formelle, on peut automatiser totalement ou partiellement ou, à défaut, rendre systématique la mise en œuvre de la technique.

Les deux paragraphes suivants (2.3.5 et 2.3.6) sont consacrés à la présentation de ces catégories de techniques.

2.3.5 Techniques de test structurel

Le test structurel [Hua75] est certainement l'approche la plus largement utilisée en milieu industriel. Il comprend des techniques définis sur la structure de contrôle du programme qui est le plus souvent représentée par un *graphe de contrôle*. Ce modèle est le plus approprié pour représenter des logiciels programmés dans des langages séquentiels (Pascal, C, etc).

Après une définition rapide du graphe de contrôle et de ses composants, nous survolons les principaux critères définis à ce jour ainsi que les moyens de procéder à la génération de jeux de test.

2.3.5.1 Graphe de contrôle

Un graphe de contrôle est un graphe orienté $C = (N, A, e, s)$ tel que :

- N est un ensemble de *nœuds*;
- A est une relation binaire sur N, appelé ensemble d'*arcs*;
- $e \in N$ et $s \in N$ sont les *nœuds d'entrée* et *de sortie* du graphe.

Un nœud de N correspond à une instruction du programme autre qu'une instruction conditionnelle ou bien à l'évaluation de la condition d'une instruction conditionnelle ou d'une boucle.

Un arc $(n_i, n_j) \in A$ représente un transfert du contrôle de n_i à n_j. Un arc sera appelé *branche* si n_i est une condition. A chaque branche du graphe de contrôle on peut associer un prédicat, appelé *prédicat de branche*, qui décrit les conditions requises pour que la branche soit traversée.

Un *chemin* du graphe de contrôle est une séquence $(n_1, ..., n_k)$ de nœuds de N tels que $n_1 = e$, $n_k = s$ et pour tout i, $1 \leq i < k$, $(n_i, n_{i+1}) \in A$. Une *variable d'entrée* est une variable du programme apparaissant dans une instruction de lecture. Un chemin est

exécutable s'il existe une valeur des variables d'entrée du programme pour lesquelles le chemin est traversé pendant l'exécution associée. S'il n'existe aucune telle valeur, le chemin est dit *impossible*.

2.3.5.2 Critères de sélection

Une multitude de critères de sélection définis sur le graphe de contrôle des programmes ont été proposés dont une classification au moyen de la relation d'inclusion est donnée dans [Nta88]. Le critère le plus élémentaire est la *couverture des instructions* (ou couverture des nœuds) qui consiste à exécuter le logiciel de sorte que tous les nœuds de son graphe de contrôle soient exécutés au moins une fois.

La *couverture des branches* est le critère le plus largement utilisé ces dernières années en milieu industriel [Bei90]. Elle consiste à exécuter toutes les branches du graphe de contrôle ce qui revient à exécuter le logiciel de telle sorte que toutes les conditions figurant dans les instructions de branchement conditionnel soient évaluées au moins une fois à vrai et au moins une fois à faux.

Enfin, la *couverture des chemins* est satisfaite si chaque chemin du graphe de contrôle est exécuté au moins une fois.

La couverture des instructions est incluse dans la couverture des branches, elle même incluse dans la couverture des chemins.

En présence de boucles, le nombre de chemins d'un graphe de contrôle est potentiellement infini. En conséquence, la couverture des chemins est théoriquement impossible à atteindre. Pour cette raison, plusieurs autres critères ont été définis dans le but de combler le vide entre la couverture des branches et celle des chemins. Par exemple, la couverture des chemins intérieur-extérieur[1] [How75] consiste à sélectionner pour chaque boucle un chemin tel que la condition d'entrée soit fausse (extérieur) et au moins un chemin tel que la condition soit vraie (intérieur). La couverture des chemins de longueur *k* consiste, elle, à sélectionner tous les chemins provoquant au plus *k* itérations des boucles. Enfin, la couverture des *LCSAJ* ("Linear Code Sequence and Jump") [WHH80] est fondée sur la définition de portions de code comportant une séquence d'instructions consécutives dont la première est soit l'instruction d'entrée du programme soit la destination d'une instruction de branchement et la dernière est soit une instruction de branchement soit l'instruction de sortie du programme.

Un autre type de critères procède à une sélection de chemins fondée sur l'*analyse du flux de données* dans le programme [RW85] [CPRZ89]. Les chemins considérés sont ceux reliant une affectation de variable à une de ces utilisations ultérieures dans un calcul. On espère ainsi mettre en évidence des défauts dans l'utilisation ou la définition des variables. Les critères de sélection de ce type s'expriment en termes de couverture de ces chemins. Par exemple [RW85], la *couverture des définitions* consiste en l'exécution, pour chaque affectation de variable, d'au moins un chemin la reliant à une

1. Traduction libre de *"boundary-interior path testing"*

de ses utilisations ultérieures tandis que la *couverture des utilisations* exécute en plus un tel chemin pour chacune de ces utilisations (des critères intermédiaires de couverture partielle des utilisations sont également définis tels que la couverture des utilisations dans une condition d'instruction conditionnelle ou de boucle ou la couverture des utilisations de la variable dans un autre calcul).

Un point commun de tous les critères de type structurel est qu'ils décomposent le domaine d'entrée du logiciel en des sous-domaines (non forcément disjoints) dont chacun est représenté dans l'ensemble de jeux de test sélectionné par au moins un élément. Par exemple, la couverture des instructions divise le domaine d'entrée du logiciel en un ensemble de sous-domaines tel que les entrées de chaque sous-domaine exécutent une instruction. Il est ensuite supposé qu'il suffit de sélectionner un seul élément par sous-domaine pour révéler tous les défauts de l'instruction correspondante.

Notons que lors d'une évaluation expérimentale de la fiabilité (au sens donné au paragraphe 2.3.2.2) de l'approche structurelle [How76], il a été constaté qu'un jeu de tests satisfaisant la couverture des chemins garantissait la détection d'au moins 65% des défauts des programmes considérés.

Les techniques de test de type structurel sont utilisées pour le test unitaire des logiciels. Elles sont de ce fait appliquées à des programmes de taille modérée (modules ou fonctions). Cela est dû au coût élevé de représentation et d'analyse du graphe de contrôle de composants logiciels plus importants.

2.3.5.3 Génération des jeux de test

La génération des jeux de test satisfaisant un critère peut se faire, comme nous l'avons expliqué au paragraphe 2.3.1, de deux manières. La première consiste à construire à l'avance les jeux de test en s'assurant qu'ils satisfont le critère de sélection retenu. Ce type de génération, parfois qualifié de *déterministe* [Lap95], comporte deux étapes :

- Lors de la première étape il faut sélectionner un ensemble (si possible minimal) de chemins satisfaisant le critère. La difficulté de cette tâche vient de l'existence potentielle de chemins impossibles parmi ceux sélectionnés. Puisque ces chemins sont sélectionnés de manière statique, ce problème ne possède pas de solution, le calcul d'une entrée exécutant un chemin étant non décidable dans le cas général [Whi81]. Ainsi, on se contente d'utiliser des heuristiques qui tentent de minimiser le nombre de chemins impossibles (voir par exemple [BM94]).
- La deuxième étape consiste à calculer pour chacun des chemins sélectionnés un jeu de test l'exécutant. Il est en général non décidable de déterminer de manière statique les données d'entrée provoquant l'exécution d'un chemin donné. Même dans les cas particuliers où ce calcul devient décidable, il est nécessaire de procéder à une évaluation symbolique du programme dont le coût devient vite prohibitif au fur et à mesure que la complexité du graphe de contrôle et des structures de données du programme augmentent (par exemple en présence de tableaux).

Les problèmes de génération que nous venons de présenter nous amènent souvent à adopter la deuxième approche à la génération de jeux de test qui consiste à exécuter le logiciel avec des jeux de test aléatoires et à s'arrêter dès que le critère d'arrêt est satisfait. De cette manière nous ne sommes plus confrontés aux problèmes de complexité de l'évaluation symbolique nécessaires au calcul des jeux de test. Un cas intéressant de génération de ce type est le *test statistique* [Thé89] [Wae93] : la distribution des entrées est déterminée par une analyse de la structure du programme.

Notons que des approches se situant entre les deux mentionnées plus haut sont envisagées. Une approche courante est l'utilisation des jeux de test aléatoires pour les phases initiales du test. Cela permet dans la plupart des cas d'obtenir rapidement un taux de satisfaction suffisamment élevé du critère considéré (par exemple 70% de branches exécutées). L'expérience montre qu'au delà d'un certain taux, le test aléatoire est peu efficace, certaines parties des programmes ayant une très faible probabilité d'être activées. Des techniques de génération déterministe sont donc utilisées pour améliorer ce taux.

Une autre approche est celle proposée par Korel [Kor90]. Il s'agit d'une technique qui remplace une partie de l'exécution symbolique nécessaire à la génération déterministe par une analyse dynamique du programme. Cette analyse se déroule pendant l'exécution de ce dernier et permet, à l'aide d'heuristiques, de sélectionner plus facilement des données d'entrée satisfaisant le critère visé.

Il faut remarquer que, dans tous les cas, il est nécessaire de disposer d'outils d'aide à la génération des jeux de test et à l'analyse du programme. Sans de tels outils, le coût de l'application des techniques de test structurel peut devenir prohibitif, le calcul manuel des chemins, de leurs prédicats et des entrées les satisfaisant étant souvent très complexe. Par ailleurs, la confiance qu'on pourrait placer dans le processus de test serait nettement affectée, les jeux de test manuellement développés pouvant plus facilement être erronés (i.e. ne pas garantir la satisfaction du critère).

Ainsi, on utilise d'une part des outils d'évaluation symbolique de programmes, qui permettent le calcul automatique d'un prédicat de chemin et, éventuellement, des entrées le satisfaisant et, d'autre part, de compilateurs spécialisés produisant pour le logiciel sous test un code enrichi d'instructions destinées à mesurer le taux de satisfaction d'un critère donné.

2.3.6 Techniques de test fonctionnel

Les techniques de test fonctionnel, ou "boîte noire", sont généralement fondées sur l'analyse des spécifications, formelles ou informelles, du logiciel.

Dans [How87], la spécification du programme consiste en une *abstraction fonctionnelle*, c'est à dire en la description des fonctions que doit réaliser le programme. Cette abstraction n'est pas nécessairement donnée dans un langage formel. De ce fait, son analyse pour la génération de données testant chaque fonction identifiée ne peut pas être automatique.

L'existence de spécifications formelles permet de rendre - totalement ou partiellement - automatique cette génération. Par exemple, une méthode de dérivation de jeux de test à partir de spécifications algébriques a été proposée dans [BGM91]. Son intérêt découle, en particulier, de la définition d'un cadre théorique qui prend en compte à la fois la sélection des jeux de test, les hypothèses sous lesquelles les jeux de test sont valides et, enfin, la construction d'un oracle. Plus précisément, un *contexte de test* est défini comme un triplet *(H, T, O)* tel que *T* est un jeu de test fini et *O* est un oracle construit pour identifier les défaillances provoquées par les éléments de *T*. *H* est un ensemble d'hypothèses sous lesquelles :

- le contexte de test reconnaît tous les programmes erronées (i.e. *T* est valide et *O* détecte toutes les défaillances),
- le contexte de test ne rejette pas de programmes corrects (i.e *O* ne détecte pas de défaillance inexistante).

La spécification formelle du logiciel est donnée sous forme d'axiomes. Une *instance* d'un axiome est obtenue en affectant à chaque variable d'entrée apparaissant dans ce dernier une valeur. Ainsi, sélectionner un ensemble de jeux de test consiste à construire un nombre *fini* d'instances pour chaque axiome. Cette construction peut reposer sur les hypothèses suivantes :

- L'*hypothèse d'*Ω*-régularité* stipule que chaque axiome peut être testé au moyen d'un nombre fini d'instances. Plus précisément, on considère une *fonction d'intérêt* associant une valeur entière à toute instance de l'axiome (typiquement sa longueur). Ensuite, une borne supérieure est fixée et toute instance dont la fonction d'intérêt y est supérieure est ignorée.
- L'*hypothèse d'*Ω*-uniformité*, stipule qu'une seule des instances de l'axiome suffit pour le tester.

Il est intéressant de noter qu'une hypothèse d'uniformité, bien que cela ne soit pas toujours clairement mentionné, est utilisée par la plupart des critères de sélection, structurels ou fonctionnels, fondés sur la division du domaine d'entrée. Par exemple, la couverture des instructions suppose qu'il suffit d'exécuter chaque instruction du programme une seule fois pour détecter tous les défauts éventuels qu'elle comporte.

L'expérience montre que cette hypothèse n'est pas toujours justifiée. En particulier, il est fréquent que des défauts soient localisés à des parties du programme effectuant des traitements singuliers, c'est à dire des traitements correspondant à des éléments très spécifiques du domaine d'entrée. C'est le cas, par exemple, des valeurs proches des frontières des sous-domaines identifiés par les critères de sélection : l'exécution du logiciel avec ces entrées est qualifiée de "test aux limites"[1] [Mye79]. Cependant, l'identification de ces éléments est très difficilement automatisable.

1. *"Boundary-value testing"*

Une autre approche [ROT89] suggère d'appliquer des critères de sélection analogues à ceux utilisés pour le test structurel à des spécifications écrites dans des langages formels. Les spécifications considérées sont toujours ramenées à un ensemble de couples (précondition, postcondition). Un processus de test automatique peut être alors défini permettant d'engendrer les jeux de tests satisfaisant les préconditions tout en vérifiant que la postcondition est vérifiée.

Notons également une approche de test utilisant à la fois le code du programme et sa spécification pour construire une partition du domaine d'entrée [RC85]. Des techniques de vérification formelle par évaluation symbolique et de test structurel sont alors appliquées conjointement à partir de cette partition pour vérifier que les fonctions réalisées par le logiciel sont identiques à celles définies dans la spécification.

Enfin, une approche intéressante est proposée dans [Cho78]. La spécification formelle sur laquelle elle s'appuie est un automate d'états finis. Des critères de couverture semblables à ceux considérés dans le cadre des techniques structurelles sont définis sur cet automate, tels que la couverture des états, la couverture des transitions et la couverture des séquences de transitions d'une longueur donnée.

2.3.7 Test structurel et test fonctionnel : deux approches complémentaires

Les deux approches de test qui viennent d'être présentées visent la détection de défauts de nature différente. En effet, les techniques de test fonctionnel ont pour objectif de mettre en évidence l'existence de défauts dont l'origine est la mauvaise compréhension de la spécification du logiciel. La difficulté essentielle dans leur conception est la définition de critères permettant la sélection judicieuse de données d'entrée à partir de la spécification.

Les techniques de test structurel, au contraire, visent des défauts dont l'origine peut être l'inattention du programmeur ou la maîtrise insuffisante d'un langage de programmation utilisé. Pour cette raison, elles cherchent à exécuter différents composants du graphe de contrôle du programme de sorte que des défauts de ce type puissent, le cas échéant, se manifester.

Il est aujourd'hui reconnu que les deux approches sont complémentaires. Par exemple, les techniques de test structurel ne sont pas capables d'identifier des traitements manquants dans le programme, que seule une technique fonctionnelle basée sur sa spécification pourra révéler. Les techniques fonctionnelles, en revanche, sont dans l'incapacité d'activer de manière satisfaisante les composants du programme et peuvent, de ce fait, laisser échapper certains défauts de programmation.

2.3.8 Techniques de construction d'oracles

Bien que l'oracle soit une composante essentielle du processus de test, très peu de travaux de recherche ont à ce jour concerné ce thème. Afin de résumer les pratiques cou-

rantes et les travaux de recherche relatifs aux oracles, nous identifions deux cas de figure :

- Une spécification formelle (même partielle) du logiciel est disponible.
- Seules des spécifications informelles sont fournies.

Dans le deuxième cas, la seule possibilité est l'observation du comportement du logiciel par un humain. Cela peut être une solution acceptable pour certains types de logiciels dont les défaillances se manifestent de manière nette (par exemple arrêt d'un système d'exploitation). Cependant, ce mode d'observation ne peut être une solution générale au problème de l'oracle car il ne permet pas la détection de défaillances plus subtiles (cf. paragraphe 2.3.3).

L'existence d'une spécification formelle du logiciel facilite considérablement la construction d'un oracle. Par exemple, si tout jeu de test est accompagné de la valeur des sorties attendue, il est très simple de construire un oracle automatique comparant, après exécution, les résultats obtenus à ceux attendus. Malheureusement, il n'est pas toujours facile de disposer de cette information. En effet, calculer "à la main" le résultat attendu peut être une opération très fastidieuse, voire impossible en pratique.

Cette remarque s'applique particulièrement au cadre des logiciels réactifs dont les entrées et les sorties sont des *séquences* de valeurs et dont on cherche à observer le comportement pour une période de temps relativement longue (ce qui implique une longueur élevée des séquences à étudier). Dans [RLO92] on trouvera une méthodologie de dérivation systématique d'oracles pour des logiciels réactifs à partir de spécifications formelles RTIL (Real-Time Interval Logic) et Z. Ces spécifications sont des propriétés invariantes qui caractérisent les comportements acceptables du système. Cependant, cette méthodologie ne permet pas la construction automatique de l'oracle.

L'utilisation de langages de spécification plus restreints mais décidables simplifie sensiblement cette tâche. Dans [DY94] on trouvera une méthode de spécification d'oracles au moyen de la logique temporelle graphique GIL (Graphic Interval Logic). Dans ce cas l'oracle peut être engendré de manière automatique.

2.4 TEST DES LOGICIELS RÉACTIFS SYNCHRONES

2.4.1 Motivations, objectifs et hypothèses

Ce travail est principalement motivé par le besoin de nouveaux moyens de vérification et de validation de logiciels synchrones. Ces moyens devraient combler certaines faiblesses des approches fondées sur la preuve formelle signalées au paragraphe 2.2.2.

Notre objectif est de proposer des techniques de test pour des logiciels synchrones fondées sur l'utilisation du langage LUSTRE. Il s'agit, en particulier, de définir des *critères de sélection* et des moyens de *génération automatique* de jeux de test adaptés au contexte particulier de ces logiciels [OP94b].

Nous envisageons le test de deux points de vue. Le premier consiste à se placer dans un cadre identique à celui de la preuve automatique de programmes LUSTRE en supposant en particulier que la spécification de l'environnement du logiciel, de ses propriétés de sûreté et de son comportement sont toutes exprimées en LUSTRE. Dans un tel contexte, nous considérons que le test est un moyen de vérification et de validation *complémentaire* à la preuve formelle de LUSTRE. Cette complémentarité a plusieurs aspects :

- Le test peut être une *solution de secours* au cas où la preuve formelle échoue, suite à un manque de ressources physiques. Il serait alors intéressant de pouvoir utiliser les mêmes spécifications pour tester le logiciel même si les résultats obtenus sont plus modestes que ceux de la preuve.
- Le test peut être un moyen de *validation des spécifications* utilisées pour la preuve. En effet, le développement des spécifications sur lesquelles repose la preuve formelle est souvent complexe si bien que ces dernières peuvent contenir des défauts. L'animation de ces spécifications avant la preuve peut permettre d'acquérir une certaine confiance dans leur validité et, par ce biais, dans les résultats de la preuve.
- Le test peut permettre la *vérification et la validation de programmes* pour lesquels la preuve est impossible parce que leur spécification utilise des variables numériques.

Mais il est possible de percevoir le test comme une technique de vérification et de validation qui peut être appliquée *indépendamment de la preuve formelle*. Ceci est en particulier vrai dans les deux cas suivants :

- On s'intéresse à la recherche de défauts qui n'ont pas d'impact sur les propriétés de sûreté mais dont la détection et l'élimination sont néanmoins importantes. La preuve formelle est dans l'incapacité de mettre en évidence ces défauts.
- Le logiciel à tester n'est pas entièrement spécifié en LUSTRE. Dans ce cas, l'application de la technique de preuve formelle n'est simplement pas envisageable.

Pour le premier cas, nous pouvons envisager des moyens de simulation qui permettent, par observation, la détection de certains défauts tandis que pour le second il est nécessaire de concevoir des techniques qui sont fondées sur l'existence de spécifications partielles du logiciel.

Les techniques que nous avons développées et qui sont rapidement présentées dans la suite de ce paragraphe, expriment ces deux manières d'envisager le test. Elles consistent, d'une part, en des critères de sélection, de type aussi bien fonctionnel que structurel. Nous estimons qu'en incluant des critères de ces deux approches complémentaires, le pouvoir de détection de défauts de l'ensemble des techniques proposées ne saurait qu'être renforcé (cf. paragraphe 2.3.7).

Nous nous sommes efforcés, d'autre part, de définir des moyens de génération entièrement automatique des jeux de test conformes à ces critères.

Quel que soit le cadre dans lequel nous nous plaçons, notre choix est de *ne pas imposer un effort de spécification supplémentaire* au programmeur par rapport à celui déjà demandé pour la vérification formelle : nos techniques de test nécessiteront, au plus, le même ensemble de spécifications, à savoir le programme LUSTRE, la spécification des contraintes d'environnement et la spécification des propriétés de sûreté.

Ce choix, accompagné de la possibilité de génération automatique de jeux de test, permet à nos techniques d'être facilement intégrées dans le processus de vérification actuel par la preuve, lui aussi entièrement automatique.

Notons, enfin, que les techniques de test que nous proposons peuvent, dans certaines conditions, être utilisées aussi bien comme techniques de test unitaire que d'intégration ou système. Leur utilisation, cependant, pour ces deux derniers types de test dépend de la complexité des spécifications composant le logiciel. Ce point relève de la méthodologie de test utilisée pour la validation des logiciels synchrones et est discuté de manière plus détaillée au paragraphe 2.4.4.

2.4.2 Terminologie

Comme nous l'avons vu au chapitre 1, un logiciel synchrone peut être assimilé à une fonction associant une séquence de sorties à une séquence d'entrées. Construire un jeu de test pour un logiciel synchrone revient donc à construire un ensemble de séquences d'entrées. Nous adopterons pour la suite du document une terminologie adaptée à cette particularité.

Soient E et S respectivement les ensembles des variables d'entrée et de sortie d'un logiciel synchrone et soient V_E et V_S leurs ensembles de valeurs respectifs.

- Un élément de V_E est une *entrée*.
- Un élément de V_S est une *sortie*.
- Une *donnée de test* est une séquence d'entrées, c'est à dire un élément de V_E^n où n est un entier positif non nul quelconque (on parlera aussi de *donnée de test de longueur n*).
- Enfin, un *jeu de test* est un ensemble fini ou infini de données de test de longueur quelconque.

2.4.3 Aperçu des techniques de test proposées

Parmi les techniques de test que nous proposons, nous qualifions de fonctionnelles celles qui utilisent pour la génération de jeux de test la spécification de l'environnement du logiciel et, éventuellement, celle de ses propriétés de sûreté. En revanche, les

techniques qui utilisent la spécification fonctionnelle du logiciel (i.e. son implantation LUSTRE) sont qualifiées de structurelles.

La conception des techniques structurelles a consisté en la définition d'un cadre analogue à celui utilisé pour les programmes séquentiels. Plus précisément, nous avons choisi un modèle représentant la structure du programme LUSTRE et nous avons défini des critères de sélection ainsi que des moyens de génération automatique de données de test les satisfaisant.

Les techniques fonctionnelles présentent des similitudes avec certaines techniques de test fondées sur des spécifications formelles et plus particulièrement avec les méthodes de sélection de jeux de test à partir de spécifications algébriques (cf. paragraphes 2.3.6). En effet, les propriétés de sûreté de logiciel peuvent être vues comme des "axiomes" dont le test doit examiner la validité. Toutefois, les spécifications étant dans notre cas des formules de logique temporelle exprimées en LUSTRE, le principe de l'analyse nécessaire à la génération est différent.

2.4.3.1 Techniques de test fonctionnel

Nous avons d'abord étudié la mise en œuvre du critère le plus élémentaire, le *test aléatoire*. Bien que ce type de test soit, en général, d'une grande simplicité, il s'avère qu'il n'en est pas ainsi dans le cas des logiciels réactifs. En effet, ces derniers sont généralement construits sous certaines hypothèses sur leur environnement d'exécution; elles doivent donc être satisfaites par les données de test engendrées.

Ainsi, la première technique que nous avons développée est fondée sur l'utilisation de la spécification de l'ensemble des comportements valides de l'environnement. Cette spécification est la même que celle introduite sous forme d'assertions dans le nœud de vérification formelle. Les données de test sont choisies de manière aléatoire parmi toutes les séquences satisfaisant cette spécification. Il s'agit de ce fait d'une *simulation aléatoire de l'environnement* [PO96b].

Une telle génération de données de test aléatoires peut être utile pour plusieurs raisons :

- Elle offre un moyen simple de simuler le comportement de l'environnement afin de valider sa spécification qui peut éventuellement être ensuite utilisée à des fins de vérification formelle.
- Elle permet de tester le logiciel avec des données valides vis à vis de la spécification de l'environnement. Ainsi, les défaillances dues à un comportement incohérent de ce dernier et non à un défaut du logiciel ne perturbent pas le processus de test.
- Elle crée des conditions propices à la mesure de la *fiabilité* du logiciel. En effet, la fiabilité d'un logiciel ne peut être mesurée de manière significative que si le comportement de son environnement est cohérent.

La simulation de l'environnement définit implicitement un premier critère de sélection, puisque les données de test sont choisies en fonction de leur conformité à une spécification. Nous estimons que ce critère est le *plus faible* (au sens de l'inclusion des critères) que l'on puisse définir pour les logiciels réactifs synchrones. En effet, tout autre critère incluant strictement cette simulation de l'environnement permettrait la sélection de données de test qui ne satisferaient pas la spécification de l'environnement et qui, de ce fait, seraient sans intérêt pour la détection des défauts. En d'autres termes, la simulation de l'environnement est synonyme du *test exhaustif* pour les logiciels synchrones.

Nous avons ensuite étudié la prise en compte de la spécification des propriétés de sûreté par le processus de test. Nous envisageons deux utilisations de cette spécification. La première consiste en une extension intéressante du principe de la simulation de l'environnement, la *simulation d'un logiciel sûr.* Elle consiste à reproduire, de manière aléatoire, les comportements du logiciel sous test conformes à l'ensemble de ses propriétés de sûreté. L'utilisation conjointe des deux types de simulation (de l'environnement et du logiciel) peut fournir un moyen intéressant de validation des spécifications en vue de la preuve (cf. paragraphe 2.4.4.1).

La deuxième technique fondée sur la spécification des propriétés de sûreté du logiciel, appelée *test des propriétés de sûreté* [PO96a], permet d'engendrer automatiquement des comportements de l'environnement qui sont à la fois conformes à sa spécification et aptes à détecter des violations des propriétés de sûreté. Elle utilise pour cela à la fois la spécification de l'environnement et celle des propriétés de sûreté du logiciel, développées de la même manière que dans le cadre de la vérification formelle. Le critère de sélection ainsi défini est inclus dans la simulation de l'environnement et cerne mieux le problème de la vérification de la validité des propriétés de sûreté.

Le test des propriétés de sûreté est une technique dont les motivations sont analogues à celles de la preuve formelle. En effet, les données de test sont sélectionnées en fonction de leur aptitude de mettre en évidence un non respect des propriétés de sûreté. Le domaine d'entrée du logiciel est ainsi réduit aux données possédant cette aptitude. Cette réduction du domaine des entrées est similaire à la minimisation du modèle engendré lors de la preuve formelle (cf. paragraphe 7.2).

Les techniques ci-dessus sont regroupées sous le nom de *génération aléatoire sous contraintes*, les contraintes étant la spécification de l'environnement ou des propriétés de sûreté ou l'aptitude à tester les propriétés de sûreté. Elles sont formellement définies au moyen de machines d'entrées - sorties synchrones qui modélisent les spécifications considérées. Leur définition est donnée au chapitre 3 tandis qu'au chapitre 4 nous exposons la méthode d'implantation utilisée qui est basée sur les graphes de décision binaire. Dans ce même chapitre nous étudions la possibilité d'étendre la génération aléatoire à des spécifications utilisant des contraintes numériques.

Grâce à la génération aléatoire sous contraintes, nous estimons avoir répondu aux deux premiers besoins de complémentarité à la preuve formelle définis au paragraphe

2.4.1. Le troisième besoin, au contraire, concernant la prise en compte de variables numériques, n'est que partiellement abordé ici puisque seules les contraintes numériques élémentaires peuvent être traitées de manière réaliste.

Une caractéristique intéressante de ces techniques est qu'on peut les utiliser pour tester des logiciels synchrones implantés dans d'autres langages de programmation que LUSTRE. Dans ce cas, LUSTRE est simplement utilisé pour la description de l'environnement et des propriétés de sûreté du logiciel.

2.4.3.2 Test structurel

La dernière technique que nous proposons est de type structurel. Elle consiste en la définition de critères de sélection de données de test exprimés en termes de couverture du réseau d'opérateurs associé à un programme LUSTRE. Il s'agit d'une technique de test unitaire, dont la définition s'inspire directement des techniques de test structurelles adaptées aux programmes séquentiels, présentées au paragraphe 2.3.5. La génération de données de test conformes à ces critères est automatique, en restant dans le cadre restreint de l'utilisation exclusive de variables booléennes.

Notre objectif en introduisant cette technique, présentée au chapitre 5, est de pouvoir détecter des défauts qui sont généralement issus d'erreurs de programmation et qui n'affectent pas forcément la validité des propriétés de sûreté. Nous définissons, en particulier, des chemins dans un réseau d'opérateurs et nous montrons qu'une condition d'activation peut être calculée pour chaque chemin. Nous discutons également le problème de la sélection d'un ensemble de chemins satisfaisant un critère de couverture donné et le problème de la génération automatique de données activant un chemin donné.

2.4.3.3 Oracles

L'observation du comportement du logiciel sous test est le moyen essentiel dont dispose un testeur pour détecter des défauts dans l'exécution du logiciel, en particulier ceux qui proviennent des défauts qui ne sont pas pris en compte dans l'expression des spécifications.

La spécification des propriétés de sûreté en LUSTRE peut être transformée de manière immédiate en un oracle automatique dont le rôle est d'observer si ces propriétés sont préservées à chaque instant de l'exécution (voir par exemple l'oracle construit pour l'étude de cas fournie au chapitre 6). De plus, toute la capacité de LUSTRE peut être utilisée pour l'expression des oracles : ces derniers ne sont donc pas limités au seul emploi de variables booléennes.

Comme l'écriture d'oracles ne diffère en rien de l'écriture de programmes LUSTRE, elle n'est pas abordée de manière approfondie dans ce document.

2.4.4 Aspects de méthodologie

Nous suggérons d'appliquer nos techniques de test dans deux contextes différents. Le premier cas d'utilisation envisagé est celui du développement de programmes LUSTRE destinés à être prouvés formellement. Le test fournit alors un moyen de validation des spécifications utilisées pour la preuve. Nous considérons ensuite le cas où le test est le seul moyen retenu de vérification et de validation [Par96].

2.4.4.1 Le test en tant que complément à la preuve formelle

Nous supposons que l'implantation du logiciel en LUSTRE ainsi que la spécification des invariants d'environnement et des propriétés de sûreté sont disponibles. Les techniques de test peuvent être utilisées de la manière suivante :

- Afin de valider la spécification de l'environnement, on peut procéder à l'exécution du logiciel en lui fournissant des entrées produites par simulation de l'environnement. L'observation des comportements de l'environnement produits peut nous révéler des éventuels défauts.
  Il est également envisageable de ne pas utiliser le logiciel dans cette opération en se contentant de fournir en entrée du simulateur d'environnement des données purement aléatoires.
- La validation des propriétés de sûreté peut, ensuite, avoir lieu de deux manières :
  - En les transformant en un oracle introduit dans le processus de test du logiciel, on peut confirmer, par observation, que leur satisfaction correspond aux comportements souhaités du logiciel.
  - En effectuant une simulation de l'environnement et une simulation de comportement sûr du logiciel simultanées on peut également par observation déceler des incohérences éventuelles dans la spécification des propriétés de sûreté.

Nous montrons au chapitre 7 que l'utilisation de la simulation de l'environnement et d'un oracle comportant les propriétés de sûreté du logiciel permet d'obtenir une *approximation de la preuve*. En revanche, l'exécution concurrente de la simulation de l'environnement et de la simulation des comportements sûrs du logiciel est une *animation de spécifications* visant leur validation par observation.

L'utilisation de spécifications ainsi validées ne saurait que renforcer notre confiance dans les résultats obtenus par le processus de preuve.

2.4.4.2 Utilisation exclusive du test

Considérons maintenant que le test est le seul moyen pratiqué de vérification et de validation. Nous supposons que les contraintes d'environnement et les propriétés de sûreté sont toutes disponibles et exprimées en LUSTRE. Nous distinguons deux cas :

- Le logiciel est lui aussi implanté en LUSTRE (ce cadre est identique à celui de la vérification formelle, du point de vue des spécifications disponibles).
- Le logiciel est développé dans un langage quelconque; seul son code exécutable est disponible.

Dans les deux cas nous définissons trois objectifs pour le test :

- Valider la spécification de l'environnement.
- Valider la spécification des propriétés de sûreté.
- Vérifier et valider le logiciel.

Pour la réalisation des deux premiers objectifs le langage d'implantation du logiciel est sans importance. Ces objectifs sont atteints de la même manière qu'au paragraphe 2.4.4.1. Pour la vérification et la validation du logiciel le choix du langage d'implantation a une incidence. Nous suggérons l'application des techniques de test dans l'ordre suivant :

- L'application de la simulation de l'environnement en phase initiale de test devrait permettre de détecter rapidement un certain nombre de défauts dans le logiciel par la simple observation de son comportement.
- Ensuite, l'application de la technique de test des propriétés de sûreté peut permettre la mise en évidence des défauts liés aux propriétés de sûreté, dont la détection par la seule simulation de l'environnement est difficile (par exemple des défauts provoquant une erreur uniquement sous certaines conditions spécifiques).
- Dans le cas où le logiciel est implanté en LUSTRE, l'utilisation de la technique de test structurel est recommandée. Elle peut avoir lieu avant ou après le test des propriétés de sûreté. D'ailleurs, on peut engendrer les données de test en utilisant un simulateur d'environnement, si on opte pour un mode de génération aléatoire (cf. chapitre 5).

2.4.4.3 Test unitaire, test d'intégration, test système

Pour compléter la présentation de nos techniques, nous devons préciser les types de test pour lesquels elles sont appropriées (unitaire, d'intégration ou système).

Nous distinguons de nouveau la situation où nous disposons du code source LUSTRE du logiciel à tester de celle où seul le code exécutable du logiciel est présent.

Dans le deuxième cas, les techniques applicables (génération aléatoire sous contraintes) ne peuvent être considérées que comme des techniques de test système, leur but étant la vérification de la conformité de l'ensemble du logiciel à ses spécifications.

Dans le premier cas, au contraire, on peut envisager de procéder successivement aux trois types de test du logiciel synchrone.

Le test unitaire d'un logiciel écrit en LUSTRE peut être défini comme le test des nœuds le composant. Toutes nos techniques s'appliquent à ce type de test. Plus précisément, les techniques de génération aléatoire demandent, pour chaque nœud testé, le développement d'une spécification sous forme d'invariants décrivant ses comportements et son environnement (i.e. les séquences de ses entrées). De plus, la technique structurelle est spécifiquement conçue pour ce type de test.

Bien que l'application de nos techniques au test d'intégration ne soit pas étudiée dans ce document, nous pouvons noter que la génération aléatoire sous contraintes peut servir à ce type de test. En effet, il suffit pour cela de spécifier sous forme d'invariants les propriétés que l'on souhaite vérifier sur l'interaction des différents nœuds. Par ailleurs, la technique de test structurel permet de tester de manière progressive chaque nœud, sans expansion préalable des nœuds qu'il utilise.

2.4.4.4 Estimation du coût

Il n'est pas possible de donner, *a priori*, une estimation du temps et de l'effort qui doit être consacré à chaque type de test car ces grandeurs dépendent des ressources disponibles et de la qualité visée du logiciel. A titre indicatif, la simulation de l'environnement doit être pratiquée jusqu'à ce qu'on ait gagné une certaine confiance dans les spécifications de l'environnement et qu'un nombre suffisant de données de test aient été exécutées sans aucune défaillance. Il en est de même pour le test des propriétés de sûreté. Par ailleurs, l'application de plusieurs critères de test structurel doit avoir lieu à condition qu'on adopte l'ordre d'application induit par la relation d'inclusion.

2.4.5 Correction des défauts

La plupart des travaux sur le test des logiciels ne traitent pas le problème de la localisation et de la correction des défauts, se contentant de la mise en évidence des défaillances correspondantes. Cependant, l'identification des défauts à l'origine d'une défaillance est une tâche très complexe. Des outils permettant une automatisation partielle de ce procédé ont été proposés, mais la localisation automatique des défauts est loin d'être atteinte [Duc93].

Dans ce travail, nous avons abordé ce problème d'un point de vue purement pratique en prévoyant des moyens ergonomiques de visualisation des résultats du test [OP95b] (cf. chapitre 6).

Une autre démarche qui nous semble adaptée au cas de LUSTRE est celle adoptée dans des travaux de recherche récents portant sur la testabilité des programmes flot de données [LR95]. Dans ces travaux, l'effort de localisation des défauts est évalué en fonction de la stratégie de test utilisée. Une stratégie définit un ordre dans lequel seront exécutées les différents composants du programme sous test. Les composants considérés sont des chemins (appelés écoulements) au sein d'un graphe de transfert d'information. Ce graphe est une représentation dans une même structure du graphe de contrôle et du graphe de dépendances de données. Un écoulement correspond à la sélection, pour une variable de sortie donnée, des variables d'entrée et d'un traitement (sous la forme d'une séquence d'instructions) qui affectent sa valeur.

Bien que dans ce travail nous n'ayons pas étudié le développement de techniques minimisant l'effort de localisation de défauts, nous pensons qu'une telle étude constituerait une perspective intéressante. Par ailleurs, il serait intéressant d'étudier la possibilité de procéder à une analyse des résultats de l'opération de test (les entrées et les sorties du programme sous test) afin d'identifier les parties fautives du code (un exemple d'analyse de ce type est fourni dans [AHLW95]).

2.4.6 Autres travaux sur le test des logiciels synchrones

Des travaux de recherche relatifs au test de programmes LUSTRE ont été menés récemment :

- Dans [Maz94], on trouvera une approche de test structurel statistique fondée sur l'exploration de l'automate engendré par le compilateur LUSTRE. Plus précisément, l'automate est transformé en un modèle stochastique du comportement du logiciel par association d'une probabilité à chaque transition. La longueur des séquences d'entrées est ensuite calculée de sorte que la probabilité de couverture des états, des transitions ou des séquences de deux transitions soit supérieure à un seuil fixé au préalable. Une extension au test d'intégration est également proposée.

- L'approche proposée dans [MHM+95] préconise l'utilisation conjointe de techniques de test et de vérification formelle (au moyen de LESAR) pour la validation de programmes LUSTRE. La technique de test est fondée sur l'hypothèse de l'existence d'une spécification comportementale du programme LUSTRE sous test, donnée sous forme d'un automate d'états finis (autre que celui produit par le compilateur LUSTRE). La génération de données de test s'effectue après une analyse automatique de cet automate.

- Enfin, une dernière approche [Hsi94] consiste à appliquer dans le cas de LUSTRE la technique de génération automatique de jeux de test à partir d'une spécification algébrique [BGM91] citée au paragraphe 2.3.6. A cet effet, les types de données et les opérateurs de LUSTRE sont exprimés dans une algèbre particulière. Tout programme LUSTRE peut ainsi être transformé en une spécification algébrique qui sert à la génération des jeux de test.

Ces approches sont différentes de la nôtre sur plusieurs points et de ce fait ne sont pas discutées en détail dans ce chapitre. En revanche, nous leur avons consacré, ainsi qu'à d'autres travaux récents, le chapitre 7 où nous les comparons avec nos techniques préalablement exposées.

2.5 PRÉVISION DES FAUTES ET TOLÉRANCE AUX FAUTES

Le test et la preuve de logiciels sont des moyens d'élimination de fautes. Cependant, certaines des techniques de test que nous avons développées peuvent évoluer vers deux autres moyens de la sûreté de fonctionnement, la prévision des fautes et la tolérance aux fautes.

Plus précisément, la simulation aléatoire de l'environnement peut fournir un moyen de mesure de la fiabilité du logiciel (temps moyen de fonctionnement correct et continu du logiciel). Pour cela, il est nécessaire de pouvoir spécifier de manière plus fine le comportement de l'environnement simulé, en particulier en affectant des *probabilités d'occurrence* aux différents éléments du domaine d'entrée. Cette possibilité peut être aisément prise en compte par l'implantation des techniques de génération aléatoire sous contraintes (cf. chapitre 4) [OP95a].

La tolérance aux fautes vise à limiter les effets d'une perturbation, c'est à dire accroître la probabilité qu'une erreur soit tolérée par le système. Nous nous intéressons ici à la tolérance aux fautes de conception qui concernent le développement de logiciels et pour laquelle il existe deux approches [Lap95] :

- La première a pour objectif d'éviter qu'une défaillance n'entraîne la défaillance de tout le système et cherchera donc à détecter au plus tôt la tâche erronée et de l'arrêter.
- La deuxième approche, qui nous intéresse plus particulièrement ici, cherche à assurer la continuité de service. Cet objectif est atteint par la *diversification fonctionnelle* qui consiste à mettre en œuvre un deuxième composant à même d'assurer la tâche. Une technique de diversification fonctionnelle largement utilisée est la *programmation N-autotestable*. Elle consiste en l'exécution en parallèle d'au moins deux composants logiciels, chaque composant étant constitué soit de l'association d'une variante et d'un test d'acceptation, soit de deux variantes et d'un algorithme de comparaison. Dans ce dernier cas, une des deux variantes peut se contenter de fournir un traitement de précision limitée. Le principe de fonctionnement d'un composant autotestable est décrit par la figure 2-1.

Nous montrons au chapitre 3 que la simulation d'un logiciel sûr peut très facilement évoluer vers une variante secondaire destinée à fonctionner au sein d'un composant autotestable. Cette variante fournira continuellement des sorties aléatoires satisfaisant les propriétés de sûreté du logiciel. Les sorties produites ne seront prises en compte qu'en cas de non respect de ces propriétés par la variante principale.

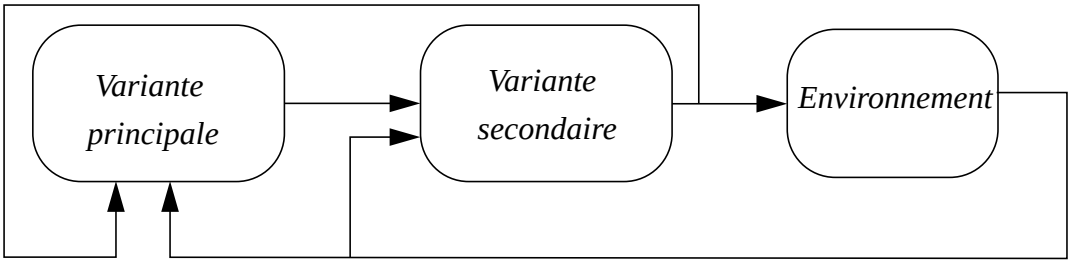

Figure 2-1 : Principe de fonctionnement d'un composant autotestable

3

Techniques de génération aléatoire sous contraintes

3.1 INTRODUCTION

Nous commençons l'exposé de nos techniques de test en présentant dans ce chapitre les techniques de génération aléatoire sous contraintes. Plus précisément nous définissons la syntaxe utilisée pour la spécification des propriétés à partir desquelles la génération est effectuée et la sémantique formelle qui lui est associée. Les aspects d'implantation de ces techniques sont exposés au chapitre 4.

Les techniques de génération aléatoire sous contraintes consistent, comme nous l'avons annoncé au chapitre 2, à engendrer des données conformes à un ensemble de spécifications LUSTRE. Suivant le type de spécifications dont on dispose, la génération aléatoire est une *simulation de l'environnement*, un *test des propriétés de sûreté* ou une *simulation d'un programme sûr.*

Conformément à notre intention de concevoir un outil complémentaire à la vérification formelle et compatible avec elle, nous considérons que les spécifications dont nous disposons sont de la même nature que celles développées dans le cadre de cette dernière. En particulier, nous restreignons notre étude aux logiciels manipulant exclusivement des variables booléennes. En fin de chapitre, nous discutons l'extension des techniques de génération aléatoire sous contraintes au cas des logiciels utilisant des variables numériques.

La simulation de l'environnement est fondée sur l'analyse d'un ensemble de propriétés invariantes décrivant les comportements valides de l'environnement du logiciel. L'environnement étant représenté par l'ensemble des variables d'entrée du logiciel, les propriétés invariantes précitées définissent les valeurs prises successivement par ces variables dans le temps, c'est à dire des séquences de valeurs. Le proces-

sus de génération aléatoire doit tenir compte de ces propriétés en engendrant des données qui les satisfont continuellement.

Quant au test des propriétés de sûreté, il fait appel aussi bien à la spécification des comportements valides de l'environnement qu'à la spécification des propriétés de sûreté du logiciel, également fournies sous forme d'invariants. Pour ce type de test, les données d'entrée produites par le processus de génération aléatoire doivent à la fois correspondre à des comportements valides de l'environnement et tester de manière significative la validité des propriétés de sûreté.

Quel que soit le type de génération (simulation d'environnement ou test des propriétés de sûreté), l'objectif de la génération aléatoire est de produire des séquences d'entrées conformes à un ensemble de contraintes. Nous montrerons dans la suite du chapitre que ces contraintes définissent, pour tout instant de l'exécution, un *ensemble d'entrées* du logiciel correspondant aux réactions de l'environnement qui sont conformes aux contraintes. La génération aléatoire sous contraintes consiste, d'une part, à représenter cet ensemble et, d'autre part, à choisir un de ses éléments qui sera l'entrée à fournir au logiciel sous test.

Les spécifications utilisées portant sur des séquences d'entrées, la conformité des réactions de l'environnement aux contraintes à un instant donné dépend généralement des sorties que le logiciel a émises en direction de son environnement aux instants précédents. De ce fait, la production de manière *statique* de séquences conformes aux contraintes d'environnement est impossible si on ne dispose pas d'une spécification du logiciel décrivant de manière exacte ses réactions. Or, la seule connaissance que nous avons du logiciel dans le cadre d'une technique de test boîte noire se résume à ses ensembles de variables d'entrée et de sortie. En particulier, nous ne disposons pas du code source implantant le logiciel et qui pourrait nous permettre de prédire ses réactions. De plus, les propriétés de sûreté du logiciel, quand elles sont disponibles, ne peuvent pas être facilement utilisées pour une telle prédiction car elles ne constituent pas une description déterministe des réactions du logiciel.

Nous sommes ainsi amenés à définir non pas une méthode de construction statique des données d'entrée mais plutôt une procédure de construction de programmes capables de produire ces données de manière *dynamique*, c'est à dire *pendant l'exécution du logiciel sous test et en interaction continue avec lui*, comme le montre la figure 3-1. Nous appelons ces programmes *générateurs aléatoires contraints*.

Un générateur aléatoire contraint calcule de manière aléatoire à tout instant, en fonction des valeurs passées des entrées qu'il a produites et des sorties correspondantes du logiciel, une nouvelle entrée satisfaisant les contraintes imposées à la génération par la spécification.

Etudions d'abord un principe simple de mise en œuvre de la génération aléatoire sous contraintes, présenté par la figure 3-2. Il s'agit d'un générateur aléatoire de valeurs booléennes produisant des entrées sans se préoccuper de la validité des contraintes d'environnement. Ce générateur est associé à un *filtre* chargé d'éliminer les données violant ces contraintes. Les contraintes d'environnement étant des expres-

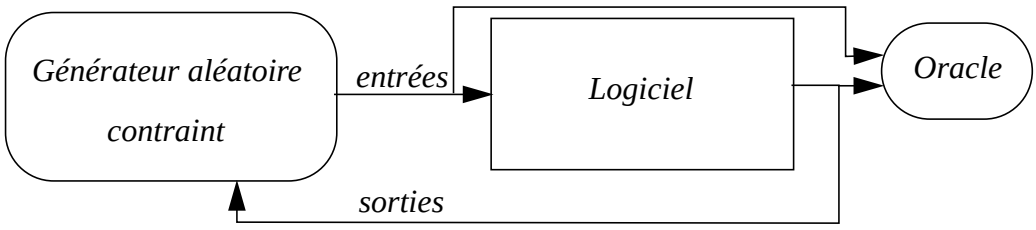

Figure 3-1 : Principe de test aléatoire sous contraintes

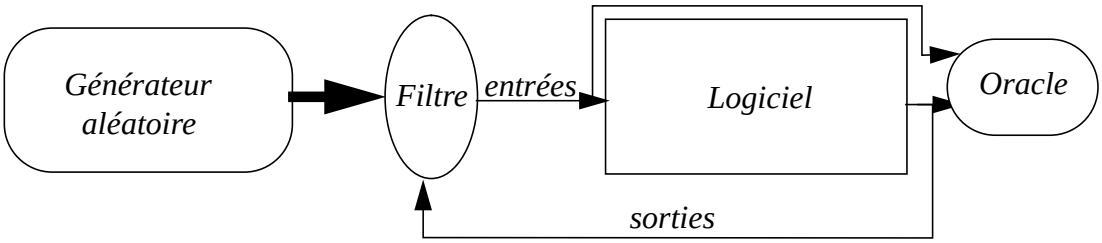

Figure 3-2 : Utilisation d'un filtre

sions booléennes, le filtre se contente d'évaluer la satisfaction de ces expressions. Si leur valeur est vraie, cela signifie que l'entrée produite par le générateur aléatoire est conforme aux contraintes et sera fournie au logiciel. Dans le cas opposé, l'entrée est rejetée et le générateur aléatoire devra engendrer une autre valeur.

Malheureusement, cette première approche rudimentaire à la génération aléatoire sous contraintes n'est pas réaliste pour un grand nombre de cas. En effet, la probabilité pour qu'une entrée engendrée de manière aléatoire à un instant donné soit conforme aux contraintes d'environnement est souvent très faible, en particulier quand le nombre des variables d'entrée du logiciel et des contraintes d'environnement est élevé. Cela nous obligerait à produire un nombre très important de valeurs d'entrée avant d'en obtenir une capable de traverser le filtre; le coût du test en serait considérablement augmenté [OP94a].

Pour cette raison, nous avons préféré définir un générateur aléatoire contraint comme un programme capable de choisir, à tout moment, une valeur conforme aux contraintes d'environnement sans que cela nécessite une étape de filtrage. Ce programme est construit de manière entièrement automatique par compilation des contraintes d'environnement. Le coût de cette construction est supérieur à celui de la solution avec filtre évoquée plus haut mais, en contrepartie, le temps de génération d'une valeur pour les variables d'entrée est sensiblement inférieur.

Le but de ce chapitre est de présenter de manière formelle les principes de la construction des générateurs aléatoires sous contraintes. Dans les paragraphes 3.3 et 3.4

nous définissons le principe de spécification des générateurs respectivement comme moyens de simulation de l'environnement et comme moyens de test des propriétés de sûreté. En particulier, nous définissons une syntaxe pour la description des générateurs sous la forme de nœuds LUSTRE. Le paragraphe 3.5 est consacré à la définition formelle des générateurs au moyen des machines d'entrées-sorties synchrones et d'algorithmes de génération appropriés.

Nous ne traitons pas dans ce chapitre un aspect important de la génération aléatoire qui est la *distribution* des données d'entrée engendrées. Ce point est discuté au chapitre 4 où nous exposons, en particulier, les moyens de mettre en œuvre une loi uniforme mais aussi une technique qui permet la définition d'autres types de distribution. Cela devra permettre, entre autres, d'intégrer des techniques de test statistique (cf. paragraphe 7.3.1).

3.2 SPÉCIFICATION À L'AIDE D'INVARIANTS

Dans le cadre de la vérification formelle de programmes LUSTRE (cf. chapitre 1) les contraintes sur l'environnement du logiciel s'expriment sous forme d'expressions booléennes invariantes LUSTRE (assertions), introduites dans le nœud de vérification au moyen de l'opérateur ***assert***. Les propriétés de sûreté du logiciel s'expriment également au moyen d'invariants.

Les techniques de génération aléatoire sous contraintes reposent aussi sur ces deux spécifications. Les invariants qui les constituent peuvent être informellement caractérisés de deux manières orthogonales.

En fonction du *type de variables* dont ils dépendent, les invariants peuvent appartenir à une des trois catégories suivantes :

1. Les invariants qui ne dépendent que des valeurs présentes ou passées des variables d'entrée du logiciel. Ces invariants portent clairement sur le comportement de l'environnement.
2. Les invariants qui ne dépendent que des valeurs présentes ou passées des variables de sortie du logiciel. Il s'agit d'invariants qui portent uniquement sur le comportement du logiciel et qui n'impliquent, de ce fait, aucune restriction sur le comportement de l'environnement.
3. Les invariants qui dépendent des valeurs présentes ou passées à la fois des variables d'entrée et de sortie du logiciel. Ces invariants peuvent exprimer une restriction du comportement du logiciel mais aussi de celui de son environnement.

En fonction des *dépendances temporelles* entre leurs variables, on identifie deux types d'invariants :

4. Les invariants exprimant des relations instantanées entre les entrées et/ou sorties du logiciel.

5. Les invariants exprimant, au moyen de l'opérateur ***pre***, une relation entre les entrées et/ou sorties du logiciel à des instants différents.

Cette caractérisation informelle fait abstraction de la présence éventuelle de variables locales dans les invariants pour des raisons de simplicité. Elle peut être facilement étendue au cas des invariants comportant ce type de variables puisque ces dernières sont également définies en fonction de variables d'entrée et de sortie.

L'utilisation d'invariants comme moyen de spécification est relativement délicate. Il est possible, en effet, que l'ensemble d'invariants fournisse une description insuffisamment précise du comportement souhaité. De même, des éventuelles "surspécifications" restreignent trop le comportement décrit.

Ces défauts de spécification peuvent être nuisibles à un processus de vérification et en particulier au test. Plus précisément, dans le cadre de la simulation de l'environnement on peut se trouver face à deux anomalies :

- En cas de spécification insuffisante, les défaillances révélées par le test ne seront pas forcément dues à des défauts du logiciel mais à des comportements incohérents de l'environnement auxquels le logiciel n'est pas sensé réagir.
- En cas de spécification trop restrictive, des comportements valides de l'environnement ne seront pas pris en compte pendant le test et, de ce fait, les anomalies éventuelles dans la réaction à ces comportements passeront inaperçus.

Nous estimons, cependant, que ce problème déborde du cadre de cette thèse. Nous nous contentons de supposer que la constitution d'une spécification correcte de l'environnement et des propriétés de sûreté du logiciel est toujours possible. Il n'en demeure pas moins que les techniques de test que nous proposons peuvent permettre d'identifier, par observation, des incohérences dans la spécification. De plus, les définitions que nous donnons dans la suite du chapitre (cf. paragraphe 3.5) permettent de caractériser certaines situations correspondant à des spécifications potentiellement erronées.

3.3 SIMULATION DE L'ENVIRONNEMENT

3.3.1 Syntaxe

Un générateur aléatoire contraint est spécifié au moyen d'un nouveau type de nœud LUSTRE que nous introduisons à cet effet, appelé *nœud testeur* ('***testnode***'). Il s'agit d'un nœud particulier dont la syntaxe est informellement suggérée par la figure 3–3. Ce nœud décrit l'environnement d'un logiciel réactif dont les variables d'entrée sont $(e_i)_{i=1,p}$ et les variables de sortie sont $(s_i)_{i=1,n}$.

L'opérateur ***environment*** ne fait pas partie du langage LUSTRE standard et ne peut figurer que dans un nœud testeur. Ses arguments sont des expressions booléennes quelconques (I_1, ..., I_r) exprimant les propriétés invariantes de l'environnement du

```
testnode N (s1, ... , sn : bool) returns (e1, ... , ep : bool);
var l1, ... , lq : bool;
let
    environment(I1, ..., Ir);        -- contraintes d'environnement

    l1 = ...; ... lq = ... ;          -- définition des variables locales
tel;
```

Figure 3–3 : Nœud testeur

```
testnode EnvClim(
    ARRETE,                   -- hors tension
    INACTIF,                  -- sous tension, pas de diffusion
    CHAUD,                    -- diffusion d'air chaud
    FROID: bool)              -- diffusion d'air froid
    returns(
    Marche,                   -- position de l'interrupteur
    TempInf,                  -- temp. ambiante < thermostat
    TempOK,                   -- temp. ambiante = thermostat
    TempSup : bool);          -- temp. ambiante > thermostat
let
    environment ((TempSup or TempOK or TempInf),
                 #(TempSup, TempOK, TempInf),
                 once_from_to(TempOK, TempInf, TempSup),
                 once_from_to(TempOK, TempSup, TempInf));
tel;
```

Figure 3–4 : Nœud testeur pour le logiciel de contrôle du climatiseur

logiciel. Ces expressions booléennes peuvent comporter des variables d'entrée et de sortie ainsi que sur les variables locales du nœud testeur.

Les variables locales doivent être toutes définies dans le nœud testeur au moyen d'équations. En revanche, à l'opposé des nœuds LUSTRE habituels, aucune équation ne définit les variables de sortie d'un nœud testeur. Le calcul de ces variables est effectué suivant la sémantique informellement expliquée au paragraphe suivant.

Exemple 3-1

Le nœud testeur associé au logiciel de climatisation présenté au chapitre 1, prend la forme donnée dans la figure 3–4.

3.3.2 Sémantique informelle

Un nœud testeur fournit une spécification *non déterministe* du comportement de l'environnement d'un logiciel réactif. Il définit, pour chaque instant de l'exécution, l'ensemble de valeurs des variables d'entrée e_i du logiciel préservant les contraintes d'environnement. Ces valeurs sont telles que les expressions booléennes associées aux contraintes d'environnement sont vraies. Ce nœud est transformé en un programme (un générateur aléatoire contraint) qui est exécuté en même temps que le logiciel sous test et qui à tout moment de l'exécution sera capable de réagir aux sorties produites par ce dernier, en émettant une entrée. La définition formelle d'un générateur ainsi que le principe de sa construction à partir d'un nœud testeur sont respectivement présentés au paragraphe 3.5 et au chapitre 4.

Une propriété invariante est une expression booléenne LUSTRE définie sur les ensembles *E* de variables d'entrée et *S* de variables de sortie du programme. Reprenons les notations de la sémantique des traces d'exécution d'un programme LUSTRE définie au chapitre 1. Considérons une trace d'exécution Σ du nœud testeur et une mémoire σ associant une valeur à chacune de ses variables. La propriété invariante *I* est vérifiée par la trace $\Sigma.\sigma$ si et seulement si $\Sigma.\sigma \vdash I \mid \textbf{\textit{true}}$.

Le processus de génération aléatoire sous contraintes se définit alors de la manière suivante :

Etant donnée une trace finie Σ*, choisir une entrée* $e \in V_E$ *telle que*
$\sigma(E) = e$ *et* $\Sigma.\sigma \vdash I \mid \textbf{\textit{true}}$.

Cette définition nous amène à imposer certaines restrictions aux invariants pouvant servir à la spécification de l'environnement :

1. Nous interdisons les invariants de la deuxième catégorie, à savoir ceux qui ne portent que sur des variables de sortie. En effet, ces invariants portent exclusivement sur le comportement du logiciel et ne peuvent pas constituer une spécification de son environnement. De ce fait, ils ne peuvent servir qu'à la spécification des propriétés de sûreté du logiciel qui sont prises en compte par le test des propriétés de sûreté présenté au paragraphe 3.4.

2. Nous excluons également les invariants qui appartiennent à la fois à la troisième et la quatrième catégorie. Ces invariants expriment des relations instantanées entre les entrées et les sorties du logiciel. De plus, parmi les invariants de la cinquième catégorie, nous excluons ceux dont le calcul dépend de la valeur courante des variables de sortie.
   En effet, le calcul des invariants ainsi exclus dépend non seulement de $\sigma(E)$, mais aussi de $\sigma(S)$. Or, la génération aléatoire sous contraintes est une approche de test boîte noire, ce qui implique qu'au moment de la détermination de $\sigma(E)$ nous n'avons aucune information sur la valeur que prennent les variables de *S*.

Ces restrictions sont traduites de manière formelle sur le modèle d'exécution défini pour les générateurs aléatoires contraints au paragraphe 3.5. Cela permet la reconnaissance automatique des invariants non conformes à ces règles. Dans ce même paragraphe nous montrons que d'autres contraintes doivent être imposées sur les invariants d'environnement pour assurer le bon fonctionnement des générateurs aléatoires contraints associés.

Nous pensons que les contraintes imposées aux invariants (et en particulier la seconde) ne restreignent pas le pouvoir d'expression du langage en ce qui concerne la spécification de l'environnement, la signification intuitive des spécifications éliminées étant pour le moins ambiguë. En effet, une contrainte sur l'environnement peut être intuitivement interprétée à tout instant *t* de l'exécution comme une propriété que doivent satisfaire les entrées que va recevoir le logiciel à cet instant. Ces entrées serviront au calcul des sorties du logiciel à ce même instant *t*. Il nous semble clair que la propriété exprimée par les contraintes d'environnement ne peut dépendre que des valeurs passées des variables de sortie (i.e. aux instants *t-1*, *t-2*, etc) et jamais de leurs valeurs à l'instant *t* qui ne sont pas encore connues.

3.3.3 Sémantique formelle

Comme pour la définition de la sémantique opérationnelle de LUSTRE présentée au paragraphe 1.5.1, nous supposons que toute occurrence de l'opérateur ***pre*** dans un nœud testeur *N* s'applique uniquement à des variables et jamais à des expressions plus complexes. Nous rappelons que, grâce au principe de substitution, cette restriction est sans aucune conséquence sur le pouvoir d'expression du langage.

Nous imposons également pour la suite de ce chapitre une autre restriction sur l'utilisation de l'opérateur ***environment***. Au lieu d'une liste d'expressions booléennes quelconques, nous admettons que cet opérateur accepte comme argument une *liste de variables booléennes locales* qui doivent être toutes définies dans le nœud testeur au moyen d'équations. Le pouvoir d'expression de l'opérateur n'est pas affecté puisque, par applications successives du principe de substitution, on peut toujours se ramener à un nœud satisfaisant cette hypothèse.

Afin de caractériser formellement les comportements d'un nœud testeur, nous pouvons définir sa sémantique sur ses traces d'exécution de manière analogue à la sémantique d'un programme LUSTRE exposée au chapitre 1. Suite à l'hypothèse formulée plus haut concernant l'utilisation de l'opérateur ***pre***, nous exprimons directement cette sémantique en fonction uniquement des deux dernières mémoires d'une trace d'exécution :

- k étant une constante et *k* sa valeur,
  $\sigma, \sigma' \vdash k \mid k$
- x étant une variable d'entrée ou locale,
  $\sigma, \sigma' \vdash x \mid \sigma'(x)$

- x étant une *variable de sortie* et *out* étant une valeur indéterminée,
  $\sigma, \sigma' \vdash x \mid out$
  Cette règle est introduite pour prendre en compte la deuxième et troisième restriction du paragraphe 3.3.2 qui interdisent à la spécification de l'environnement d'être calculée en fonction de la valeur courante des sorties. Elle ne signifie pas que σ' est indéfini pour la variable x mais uniquement que sa valeur ne peut pas être utilisée par un quelconque calcul du nœud testeur.

- E étant une expression booléenne,

$$\frac{\sigma, \sigma' \vdash E \mid \textbf{\textit{true}}}{\sigma, \sigma' \vdash \textbf{\textit{not}}\ E \mid \textbf{\textit{false}}} \qquad \frac{\sigma, \sigma' \vdash E \mid \textbf{\textit{false}}}{\sigma, \sigma' \vdash \textbf{\textit{not}}\ E \mid \textbf{\textit{true}}}$$

- E_1, E_2 et E_3 étant des expressions booléennes et α étant égal à *nil* ou *out*,

$$\frac{\sigma, \sigma' \vdash E_1 \mid \textbf{\textit{true}}}{\sigma, \sigma' \vdash \textbf{\textit{if}}\ E_1\ \textbf{\textit{then}}\ E_2\ \textbf{\textit{else}}\ E_3 \mid E_2} \qquad \frac{\sigma, \sigma' \vdash E_1 \mid \textbf{\textit{false}}}{\sigma, \sigma' \vdash \textbf{\textit{if}}\ E_1\ \textbf{\textit{then}}\ E_2\ \textbf{\textit{else}}\ E_3 \mid E_3}$$

$$\frac{\sigma, \sigma' \vdash E_1 \mid \alpha}{\sigma, \sigma' \vdash \textbf{\textit{if}}\ E_1\ \textbf{\textit{then}}\ E_2\ \textbf{\textit{else}}\ E_3 \mid \alpha}$$

- E_1, E_2 étant des expressions booléennes,

$\sigma, \sigma' \vdash E_1\ \textbf{\textit{or}}\ E_2 \mid \textbf{\textit{if}}\ E_1\ \textbf{\textit{then true else}}\ E_2$
$\sigma, \sigma' \vdash E_1\ \textbf{\textit{and}}\ E_2 \mid \textbf{\textit{if}}\ E_1\ \textbf{\textit{then}}\ E_2\ \textbf{\textit{else false}}$

- E_1 et E_2 dénotant deux expressions du langage et $\perp$ dénotant une mémoire indéfinie,

$$\frac{\perp, \sigma' \vdash E_1 \mid v_1}{\perp, \sigma' \vdash (E_1 \text{ -> } E_2) \mid v_1} \qquad \frac{\sigma, \sigma' \vdash E_2 \mid v_2}{\sigma, \sigma' \vdash (E_1 \text{ -> } E_2) \mid v_2}$$

- x étant une variable d'entrée, de sortie ou locale,
  $\perp, \sigma' \vdash \textbf{\textit{pre}}(x) \mid nil$ $\qquad$ $\sigma, \sigma' \vdash \textbf{\textit{pre}}(x) \mid \sigma(x)$
- Compatibilité avec les équations (définissant les variables locales) :

$$\frac{\sigma, \sigma' \vdash E \mid v,\ \sigma'(x) = v}{\sigma, \sigma' \vdash x = E}$$

$$\frac{\sigma, \sigma' \vdash P_1 \mid v_1,\ \sigma, \sigma' \vdash P_2 \mid v_2}{\sigma, \sigma' \vdash P_1; P_2}$$

- Compatibilité avec l'opérateur ***environment*** :

$$\frac{\sigma, \sigma' \vdash I_1 \mid \textbf{\textit{true}} \wedge \ldots \wedge I_r \mid \textbf{\textit{true}}}{\sigma, \sigma' \vdash \textbf{\textit{environment}}\ (I_1, \ldots, I_r)}$$

Notons que la règle de compatibilité avec l'opérateur ***environment*** est identique à celle définissant la compatibilité avec l'opérateur ***assert*** de la sémantique de LUSTRE standard.

3.4 TEST DES PROPRIÉTÉS DE SÛRETÉ

3.4.1 Syntaxe

Pour la mise en œuvre du test des propriétés de sûreté, nous considérons de nouveau un nœud testeur. De plus, nous introduisons un nouvel opérateur, ***safety***, qui permet de spécifier la liste des propriétés de sûreté du logiciel réactif. L'utilisation de ce nouvel opérateur est illustrée par le nœud testeur de la figure 3–5. Dans ce nœud, P_1, ..., P_u

testnode *N* (s_1, ... , s_n : ***bool***) ***returns*** (e_1, ... , e_p : ***bool***);
var l_1, ... , l_q : ***bool***;
let
    environment(I_1, ..., I_r); -- *contraintes d'environnement*
    safety(P_1, ..., P_u); -- *propriétés de sûreté*

    l_1 = ...; ... l_q = ... ; -- *définition des variables locales*
tel;

Figure 3–5 : Opérateur ***safety***

sont les expressions booléennes exprimant les propriétés de sûreté que doit satisfaire le logiciel réactif associé.

Chaque propriété de sûreté exprime une contrainte sur un aspect particulier du comportement du logiciel. Pour cette raison, nous avons préféré représenter au sein d'un nœud testeur les propriétés de sûreté non pas comme une unique expression booléenne (comme c'est le cas pour la vérification formelle proposée par LESAR) mais comme une liste d'expressions qui constitue l'argument de l'opérateur ***safety***. Cette représentation nous permettra de définir une notion d'équité du test des propriétés de sûreté (cf. paragraphe 3.5.8.2).

3.4.2 Sémantique informelle

Lors du test des propriétés de sûreté nous nous intéressons à une sélection plus fine des données de test que celle effectuée dans le cadre de la simulation de l'environnement. Plus précisément, nous souhaitons *orienter le test* vers la détection des défauts du logiciel liés au non respect des propriétés de sûreté.

Les données de test engendrées doivent, bien entendu, correspondre toujours à des comportements conformes aux contraintes d'environnement. Il est donc nécessaire de disposer à la fois de la spécification des contraintes d'environnement et de celle des

propriétés de sûreté du logiciel réactif (d'où la présence dans le nœud testeur aussi bien de l'opérateur ***safety*** que de l'opérateur ***environment***).

Contrairement à la spécification de l'environnement, les propriétés de sûreté expriment des restrictions sur le comportement du logiciel. Plus précisément, elles expriment à l'aide d'invariants l'ensemble des *comportements sûrs* de ce dernier.

Au même titre que pour la spécification de l'environnement, nous pouvons imposer des restrictions sur le type des invariants utilisés pour l'expression des propriétés de sûreté (en particulier, en excluant les invariants utilisant exclusivement des variables d'entrée). Cependant, la définition du processus de test des propriétés de sûreté ne nécessitant pas l'introduction de telles restrictions, nous nous abstenons d'en imposer par souci de simplicité.

La technique que nous proposons ici repose sur une caractérisation des données de test en fonction de leur *pouvoir de détection de défauts liés à la sûreté*. Informellement, cette caractérisation s'exprime par le principe simple suivant :

"A un instant de l'exécution t, une entrée ne peut révéler un défaut que s'il existe une sortie pour laquelle la valeur des propriétés de sûreté est fausse à ce même instant t".

Ce principe stipule qu'il est inutile pour ce type de test d'exécuter le logiciel dans des situations où les sorties qu'il émet n'ont aucun impact sur la satisfaction des propriétés de sûreté.

Exemple 3-2

Illustrons ces propos sur l'exemple simple de la propriété $p = A$ ***or*** B où A est une variable d'entrée et B une variable de sortie du logiciel. Intuitivement, p signifie que si l'entrée A est fausse, la sortie B doit impérativement être vraie afin que la propriété soit préservée. Au contraire, si l'entrée A est vraie, nous n'avons aucune attente particulière de la valeur prise par B car, dans ce cas, p est vraie pour toute valeur de cette sortie. Il est clair que les seules données capables de révéler un défaut du logiciel vis à vis de la satisfaction de la propriété de sûreté p sont celles pour lesquelles A est vraie.

Plus formellement, soit Σ une trace d'exécution du nœud testeur et soit P la conjonction des propriétés de sûreté figurant dans la liste d'arguments de l'opérateur ***safety***. Soient E et S les ensembles de variables d'entrée et de sortie du logiciel et soit σ une mémoire telle que $\sigma(E) = e \in V_E$. La mémoire σ peut révéler un défaut lié aux propriétés de sûreté si et seulement si :

$$\exists s \in V_S\ (\sigma(S) = s \Rightarrow \Sigma.\sigma \vdash P \mid \textbf{\textit{false}}).$$

Ce principe nous permet de définir de manière simple un critère de sélection des données de test mais présente un inconvénient : il exclue les entrées qui rendent toujours vraies les propriétés de sûreté à l'instant t mais qui représentent une condition néces-

saire à la non satisfaction des propriétés *à un instant ultérieur*. Nous ne traitons pas en détail ce problème dans ce paragraphe, une solution pouvant être difficilement apportée sans la définition formelle du processus de génération. Une discussion plus détaillée accompagnée d'une proposition de solution sera faite au paragraphe 3.5.8.2.

3.4.3 Sémantique formelle

A la définition de la sémantique formelle du nœud testeur donnée au paragraphe 3.3.3, nous ajoutons la règle de compatibilité avec l'opérateur ***safety*** :

- Compatibilité avec l'opérateur ***safety*** :
  $$\frac{\sigma, \sigma' \vdash P_1 \mid out \vee ... \vee P_u \mid out}{\sigma, \sigma' \vdash \textbf{\textit{safety}}(P_1, ..., P_u)}$$

Pour interpréter cette règle, il suffit de remarquer que si la satisfaction d'une propriété de sûreté P_i dépend de la valeur des variables de sortie de σ', alors elle ne peut que prendre la valeur indéfinie *out*. En effet, suivant la règle de sémantique définissant la valeur des variables de sortie, ces dernières s'évaluent toujours à *out*. Par ailleurs, selon la sémantique des opérateurs du langage, le résultat de tout opération impliquant *out* est également *out*. Ainsi, les mémoires σ, σ' sont conformes au principe du test des propriétés de sûreté énoncé au paragraphe 3.4.2 si la satisfaction d'au moins une de ces propriétés dépend des sorties (et donc s'évalue à *out*).

3.5 DÉFINITION D'UN MODÈLE D'EXÉCUTION

3.5.1 Système de transitions associé à un nœud testeur

A l'aide des règles de sémantique des paragraphes 3.3.3 et 3.4.3 nous définissons les notations suivantes :

- $\sigma, \sigma' \vdash N$ signifie que les mémoires σ, σ' sont compatibles avec les équations et avec l'opérateur ***environment*** du nœud *N*.
- $\sigma, \sigma' \vdash_{=} N$ dénote la compatibilité avec uniquement les équations du nœud *N*.
- $\sigma, \sigma' \vdash_{\mathbf{s}} N$ signifie que σ, σ' sont compatibles avec l'opérateur ***safety*** tout en vérifiant $\sigma, \sigma' \vdash N$.

Sous ces hypothèses, considérons un système de transitions T_N associé à un nœud testeur *N* dont les états sont les mémoires σ et dont la relation de transition $\rightarrow$ est définie comme suit :

$$\forall\sigma, \forall\sigma', \sigma \rightarrow \sigma' \Leftrightarrow \sigma, \sigma' \vdash N$$

Remarquons que dans un nœud LUSTRE habituel, toutes les variables (autres que ses entrées) sont définies par des équations. Dans le système de transitions associé, pour une mémoire donnée σ et pour une valeur donnée *e* des variables d'entrée *E*, il existe *au plus une* mémoire σ' telle que $\sigma'(E) = e$ et $\sigma \rightarrow \sigma'$. En effet, les équations définissent une manière unique de calculer les valeurs des variables locales et des variables de sortie du nœud dans σ' à partir des entrées *e* et de la mémoire σ. Ainsi, la donnée des entrées permet à tout moment de calculer la nouvelle mémoire σ'. De plus, la fonction de transition est définie sur les états et les entrées du modèle.

Cela n'est pas vrai pour un générateur aléatoire contraint. En effet, la valeur $\sigma'(E)$ qu'il calcule pour les variables d'entrée du logiciel n'est pas unique : toute valeur conforme à la sémantique ci-dessus (préservant en particulier la compatibilité avec l'opérateur ***environment***) peut leur être affectée.

De plus, la définition d'un générateur ne contient aucune information sur le calcul de la valeur des variables de sortie $\sigma'(S)$ par le logiciel. Or, cette valeur est nécessaire à la détermination complète de σ'.

Cette dernière différence a pour conséquence que les transitions du modèle associé à un générateur aléatoire contraint sont définies différemment de celles du modèle des programmes LUSTRE : les sorties du logiciel interviennent au même titre que son état et ses entrées dans la définition de la fonction de transition.

Nous sommes ainsi amenés à définir un nouveau modèle d'exécution sous forme d'une machine d'entrées-sorties synchrone [HLR93] spécialisée, qui est un cas particulier d'automate d'entrées-sorties [HU79]. Sa définition nous permet de différencier les entrées et les sorties du logiciel et répond de ce fait mieux aux besoins résultant de la sémantique particulière des nœuds testeurs.

Nous définissons le modèle d'exécution d'un générateur aléatoire contraint de manière progressive en utilisant, en particulier, des notions introduites dans [HLR93] et [RW87]. Dans un premier temps nous considérons une machine d'états finis modélisant un automate acceptant toute séquence d'entrées et sorties satisfaisant les contraintes d'environnement I_1, ..., I_r. Ensuite nous montrons comment ce modèle est utilisé pour la génération aléatoire sous contraintes au moyen d'algorithmes appropriés.

Par ailleurs, nous caractérisons formellement les types non autorisés de propriétés pour la spécification des générateurs aléatoires contraints, qui ont été informellement présentés au paragraphe 3.3.2.

3.5.2 Machine E/S associée à un nœud testeur

Définition 3-1 : Une *machine d'entrées sorties (machine E/S)* est une machine d'états finis $\mathcal{R} = (Q, E, S, q_{init}, r)$ telle que :

- *E* est un ensemble de variables d'entrée.

- S est un ensemble de variables de sortie.
- Q est un ensemble d'états.
- $q_{init} \in Q$ est l'état initial.
- $r \subseteq Q \times V_E \times V_S \times Q$ est la relation de transition.

Définition 3-2 : Une machine E/S $\mathcal{R} = (Q, E, S, q_{init}, r)$ est *réactive* si et seulement si

$$\forall q \in Q \; \forall e \in V_E \; \forall s \in V_S \; \exists q' \in Q \; (q, e, s, q') \in t$$

La machine $\mathcal{R}$ sera en mesure d'exécuter depuis tout état une transition pour toute valeur des variables d'entrée et de sortie.

Définition 3-3 : Une machine E/S $\mathcal{R} = (Q, S, E, q_{init}, r)$ est *associée à un nœud testeur* N si et seulement si :

- E est l'ensemble des variables d'entrée du logiciel réactif dont N est le nœud testeur (E est donc l'ensemble des variables de sortie de N).
- S est l'ensemble des variables d'entrée de N (i.e. l'ensemble des variables de sortie du logiciel réactif associé).
- Q est l'ensemble des mémoires booléennes du nœud. Ces mémoires associent une valeur aux variables de E et de S ainsi qu'aux variables locales du nœud testeur réunies dans l'ensemble L.
- q_{init} est l'état initial.
- La relation de transition $r \subseteq Q \times V_E \times V_S \times Q$ est telle que

$$\forall (q, q') \in Q^2 \; \forall (e, s) \in V_E \times V_S$$
$$((q, e, s, q') \in r \Leftrightarrow (q'(E) = e) \wedge (q'(S) = s) \wedge (q, q' \vdash_{=} N))$$

où $q'(E)$ et $q'(S)$ sont les valeurs des variables de E et de S respectives dans l'état q'.
En d'autres termes, une transition n'est définie qu'entre les états compatibles avec les équations du nœud N.
Nous remarquons que, la définition de r respecte bien la sémantique opérationnelle des nœuds testeurs donnée au paragraphe 3.5.1 puisque q' est déterminé de manière unique à partir de q, e et s. En effet, q' est une mémoire telle que $q'(E) = e$ et $q'(S) = s$. Les variables locales étant toutes définies par des équations, $q'(L)$ (et de ce fait, q') est déterminé de manière unique à partir de q, $q'(E) = e$ et $q'(S)$. Ainsi, nous représentons les transitions de la machine par une fonction de transition $t : Q \times (V_E \times V_S) \rightarrow Q$ telle que $t(q, e, s) = q'$ si et seulement si $(q, e,$

$s, q') \in r$.

Nous étendons la fonction de transition à une séquence de couples sur $V_E \times V_S$ en considérant pour tout entier $n > 0$ la fonction $t^n : Q \times (V_E \times V_S)^n \rightarrow Q$, définie comme suit :

$$t^1 = t ,$$

$$\forall n > 1 \; \forall (e_i, s_i)_{i = 1,n} \in (V_E \times V_S)^n \; \forall q \in Q$$

$$t^n(q, (e_i, s_i)_{i = 1,n}) = q' \Leftrightarrow (\exists q'' \; t^{n-1}(q, (e_i, s_i)_{i = 1, n-1}) = q'' \wedge t(q'', e_n, s_n) = q').$$

Par abus de langage, nous désignerons pour tout $n > 0$ la fonction de transition t^n par t.

Nous pouvons par ailleurs constater qu'une machine associée à un nœud testeur est réactive, sa fonction de transition étant définie pour tout état, toute entrée et toute sortie.

3.5.3 Machine contrainte par une fonction

Etant donnée une machine $\mathcal{R} = (Q, S, E, q_{init}, t)$ associée à un nœud testeur N, nous pouvons assimiler l'opérateur ***environment*** à une fonction booléenne totale $f_{env} : Q \times V_E \rightarrow \{0, 1\}$. Cette fonction est définie comme suit :

$$\forall q \in Q \; \forall e \in V_E \; (f_{env}(q, e) = 1 \Leftrightarrow \exists q' \in Q \; q'(E) = e \wedge q, q' \vdash N)$$

Nous remarquons que f_{env} est définie uniquement sur l'état et la sortie courante de la machine (i.e. l'entrée du logiciel), ce qui exprime bien le fait que les contraintes d'environnement ne peuvent pas dépendre de l'entrée courante de la machine (i.e. la sortie du logiciel) (cf. paragraphe 3.3.2).

Cette fonction permet d'opérer une restriction de la machine associée à un nœud testeur N. Les transitions de cette machine restreinte ne sont pas définies pour ses états et ses sorties qui ne vérifient pas f_{env}.

Définition 3-4 : Soit $\mathcal{R} = (Q, S, E, q_{init}, t)$ une machine E/S associée à un nœud testeur N et soit $f : Q \times V_E \rightarrow \{0, 1\}$ une fonction booléenne totale. La machine $\mathcal{R}_f = (Q, S, E, q_{init}, t_f)$ est *contrainte* par la fonction f si et seulement si

$$\forall (q, q') \in Q^2 \; \forall (e, s) \in V_E \times V_S$$

$$(f(q, e) = 1 \wedge t(q, e, s) = q') \Leftrightarrow t_f(q, e, s) = q'$$

En d'autres termes, pour qu'une transition de $\mathcal{R}_f$ soit définie pour un état *q*, une entrée *e* et une sortie *s* il faut et il suffit qu'elle le soit dans $\mathcal{R}$ et que *f* soit satisfaite par *q* et *e*.

La restriction ainsi obtenue de la machine associée à un nœud testeur, est semblable à celle opérée par la fonction d'assertion sur le modèle de programmes LUSTRE défini au chapitre 1. Informellement, la machine contrainte par une fonction *f* est obtenue à partir de la machine associée au nœud testeur en supprimant de cette dernière toutes les transitions ne satisfaisant pas *f*.

Il est facile de constater que, par définition, la machine $\mathcal{R}$ associée à un nœud testeur *N* est unique. Il en est de même pour la machine $\mathcal{R}'$ contrainte par une fonction *f*.

3.5.4 Machines accessibles

Le modèle introduit par la définition 3-4 étant obtenu par suppression de certaines transitions, il peut comporter des états qui ne peuvent être atteints depuis l'état initial. Cela peut en particulier être le cas d'une machine contrainte par une fonction *f* dans les deux cas suivants :

- La fonction *f* peut être satisfaite par des transitions issues d'états inaccessibles de la machine $\mathcal{R}$.
- La restriction de la fonction de transition de la nouvelle machine $\mathcal{R}_f$ contrainte par *f* peut rendre inaccessibles certains états pour lesquels *f* est vérifiée.

Dans ce paragraphe nous donnons les définitions nécessaires à l'identification et la suppression de ces états.

Définition 3-5 : Soit $\mathcal{R} = (Q, E, S, q_{init}, t)$ une machine E/S. Un état $q \in Q$ est *accessible* si et seulement si

$$\exists n>0\ \exists (e_i, s_i)_{i=1,n} \in (V_E \times V_S)^n\ t(q_{init}, (e_i, s_i)_{i=1,n}) = q$$

Un état est donc accessible s'il existe une séquence d'entrées et de sorties menant à cet état depuis l'état initial.

Définition 3-6 : Soit $\mathcal{R} = (Q, E, S, q_{init}, t)$ une machine E/S. $\mathcal{R}_{acc} = (Q_{acc}, E, S, q_{init}, t_{acc})$ est la *composante accessible* de $\mathcal{R}$ si et seulement si elle est telle que :

- Q_{acc} est l'ensemble de tous les états accessibles de *Q*
- t_{acc} est la restriction de *t* sur Q_{acc} :
  $t_{acc}(q, e, s) = q' \Leftrightarrow t(q, e, s) = q' \wedge (q, q') \in Q_{acc}^2$

La machine $\mathcal{R}$ est accessible si $\mathcal{R} = \mathcal{R}_{acc}$.

3.5.5 Générateurs

Un générateur aléatoire contraint simule le comportement de l'environnement d'un logiciel réactif. On exige de ce simulateur qu'il ne soit jamais en situation de blocage, c'est à dire être dans l'impossibilité de fournir des entrées satisfaisant les contraintes. Pour cette raison, avant de définir un générateur comme une association d'une machine E/S $\mathcal{R}$ et d'une fonction f contraignant $\mathcal{R}$, nous exprimons une contrainte sur f qui garantit l'absence de blocage.

Définition 3-7 : Soit $\mathcal{R} = (Q, S, E, q_{init}, t)$ une machine E/S associée à un nœud testeur N et soit $\mathcal{R}_f = (Q, S, E, q_{init}, t_f)$ la machine contrainte par la fonction booléenne totale $f : Q \times V_E \rightarrow \{0, 1\}$. f est *génératrice (par rapport à $\mathcal{R}$)* si et seulement si la composante accessible de $\mathcal{R}_f$ est réactive.

Concrètement, une machine contrainte par une fonction génératrice sera toujours en mesure de fournir une entrée satisfaisant cette fonction.

Définition 3-8 : Soit $\mathcal{R} = (Q, S, E, q_{init}, t)$ une machine d'états finis associée à un nœud testeur N et soit $f : Q \times V_E \rightarrow \{0, 1\}$ une fonction totale.

- Si f est génératrice par rapport à $\mathcal{R}$, $\mathcal{G} = (\mathcal{R}, f)$ est un *générateur*. On dit également que $\mathcal{G}$ *engendre* la fonction f.
- Si f n'est pas génératrice par rapport à $\mathcal{R}$, $\mathcal{G} = (\mathcal{R}, f)$ est un *générateur faible*.

Nous disposons, ainsi, de deux moyens de représentation des générateurs aléatoires contraints. Le premier ne convient que pour le cas où on a la certitude que la fonction f est génératrice. Or, il se peut qu'on soit dans l'impossibilité de prouver cette propriété de f. En effet, montrer qu'une fonction est génératrice nécessite, d'après la définition 3-7, le calcul des états accessibles de la machine contrainte associée. Nous verrons plus loin (paragraphe 3.5.6.2) que ce calcul peut être coûteux, voire impossible en pratique. C'est pour cette raison, entre autres, que nous avons introduit la notion de générateur faible. Pour chaque type de générateur, nous définissons au paragraphe 3.5.7 un algorithme de génération qui complète le modèle d'exécution des générateurs aléatoires contraints.

Il est intéressant de noter que la fonction f associée aux contraintes d'environnement du nœud testeur peut ne pas être génératrice, sans que cela corresponde forcément à une spécification erronée de l'environnement. En effet, il se peut que le comportement du logiciel sous test soit tel que les états de la machine contrainte par f qui correspondent à des situations de blocage ne soient jamais accédés. Notre modèle ne faisant aucune hypothèse sur le comportement du logiciel à tester, il nous est

impossible de détecter ces situations. Dans ce cas, cependant, l'utilisation d'un générateur faible est une solution strictement équivalente à celle d'un générateur.

3.5.6 Fonctions génératrices

La définition des générateurs repose sur l'existence d'une fonction génératrice décrivant les contraintes d'environnement. Nous étudions dans ce paragraphe de manière plus approfondie ce type de fonctions. En particulier, nous les comparons avec les assertions *causales*, bien connues dans le domaine des logiciels synchrones. Nous rappelons ensuite les principales difficultés que présente le calcul des états accessibles d'une machine E/S, étape nécessaire à l'identification d'une fonction génératrice.

3.5.6.1 Fonctions génératrices et assertions causales

Considérons de nouveau le modèle $\mathcal{M} = (Q, E, S, q_{init}, a, s, t)$ associé à un programme LUSTRE (cf. paragraphe 1.5.2). La fonction d'assertion $a : Q \times V_E \rightarrow \{0, 1\}$ est dite *causale* [Rat92] si :

$$\forall q \in Q \; \forall e \in V_E \;\; (a(q, e) = 1 \Rightarrow \exists e' \in V_E \; a(t(q, e), e') = 1)$$

Informellement, cela signifie que si la fonction d'assertion est vérifiée pour un état q et une entrée e, l'*unique transition* associée mène à un état q' où la satisfaction de la fonction d'assertion n'est pas impossible.

Cette définition n'est pas directement applicable à la fonction $f_{env} : Q \times V_E \rightarrow \{0, 1\}$ contraignant la machine E/S $\mathcal{R} = (Q, S, E, q_{init}, t)$ que nous avons retenue comme modèle du générateur. En effet, pour un état et une entrée satisfaisant f_{env} il existe éventuellement plusieurs transitions dans le modèle, la fonction de transition t étant définie sur les états, les entrées *et* les sorties.

En supposant que $\mathcal{R} = (Q, S, E, q_{init}, t)$ est accessible, nous pouvons définir trois types de causalité pour f_{env} :

- Si f_{env} est vérifiée pour un état et une entrée, *il existe au moins une transition* menant à un état où la satisfaction de f_{env} n'est pas impossible :

$$\forall q \in Q \; \forall e \in V_E$$
$$f_{env}(q, e) = 1 \Rightarrow \exists s \in V_S \; \exists q' \in Q \; \exists e' \in V_E \; ((t(q, e, s) = q') \wedge (f_{env}(q', e') = 1)) \quad \textbf{(3-a)}$$

- En plus de 3-a, $\mathcal{R}$ vérifie la propriété suivante : si f_{env} est vérifiée pour un état et une entrée, *toute transition définie* mène à un état où la satisfaction de f_{env} n'est pas impossible :

$$\forall q \in Q\ \forall e \in V_E\ \ \forall s \in V_S\ \forall q' \in Q$$
$$((f_{env}(q, e) = 1) \wedge t(q, e, s) = q') \Rightarrow \exists e' \in V_E\ f_{env}(q', e') = 1 \qquad \textbf{(3-b)}$$

- En plus de 3-b, $\mathcal{R}$ vérifie la propriété suivante : si f_{env} est vérifiée pour un état et une entrée, *une transition est définie pour toute sortie* et mène à un état où la satisfaction de f_{env} n'est pas impossible :

$$\forall q \in Q\ \forall e \in V_E$$
$$f_{env}(q, e) = 1 \Rightarrow \forall s \in V_S\ \exists q' \in Q\ \exists e' \in V_E\ ((t(q, e, s) = q') \wedge (f_{env}(q', e') = 1)) \qquad \textbf{(3-c)}$$

D'après la définition 3-7, f_{env} est génératrice si la composante accessible de la machine $\mathcal{R} = (Q, S, E, q_{init}, t)$ contrainte par f_{env} est réactive, c'est à dire :

$$\forall q \in Q\ \exists e \in V_E\ \forall s \in V_S\ \exists q' \in Q\ (f_{env}(q, e) = 1 \wedge t(q, e, s) = q') \qquad \textbf{(3-d)}$$

Soient $q \in Q$ et $e \in V_E$ tels que $f_{env}(q, e) = 1$. D'après 3-d on a :

$$f_{env}(q, e) = 1 \Rightarrow \forall s \in V_S\ \exists q' \in Q\ t(q, e, s) = q'$$

En appliquant de nouveau 3-d à l'état q' on obtient :

$$f_{env}(q, e) = 1 \Rightarrow \forall s \in V_S\ \exists q' \in Q\ \exists e' \in V_E\ ((t(q, e, s) = q' \wedge f_{env}(q', e') = 1))$$

Nous constatons donc que 3-d $\Rightarrow$ 3-c (f génératrice $\Rightarrow$ f causale). La réciproque (f causale $\Rightarrow$ f génératrice) n'est manifestement pas vraie. En effet, la formule 3-c ne garantit pas pour tout état l'existence d'une entrée satisfaisant f_{env}.

3.5.6.2 Calcul des états accessibles

D'après la définition 3-7, afin de déterminer si une fonction $f : Q \times V_E \rightarrow \{0, 1\}$ est génératrice par rapport à une machine E/S $\mathcal{R} = (Q, S, E, q_{init}, t)$, il est nécessaire, d'une part, de calculer la composante accessible de la machine associée contrainte par f et, d'autre part, de vérifier que cette composante est réactive.

Le calcul de l'ensemble *Acc* des états accessibles d'une machine d'E/S est un calcul de plus petit point fixe [HLR93]. Plus précisément, soit $post: Q \rightarrow 2^Q$ la fonction permettant de calculer les états successeurs d'un ensemble d'états :

$$post(X) = \{q' \mid \exists q \in X, \exists (e,s) \in V_E \times V_S, t(q, e, s) = q'\}$$

Dans le cas d'une machine contrainte par une fonction *f*, les seuls états successeurs retenus sont ceux atteints par des transitions satisfaisant *f*. Cela revient à redéfinir une fonction $post_f$ remplaçant *post* de la manière suivante :

$$post_f(X) = \{q' \mid \exists q \in X, \exists (e,s) \in V_E \times V_S, t(q, e, s) = q' \wedge f(q, e) = 1\}$$

Si $\mu x.g$ dénote le plus petit point fixe d'une fonction *g*, l'ensemble *Acc* des états accessibles de la machine $\mathcal{R}_f$ contrainte par *f* est défini comme suit :

$$Acc = \mu X.\ q_{init} \cup post_f(X)$$

Le calcul des états accessibles nécessite donc de parcourir de manière exhaustive les états de la machine $\mathcal{R}_f$ depuis son état initial.

Pour déterminer si la composante accessible de $\mathcal{R}$ est réactive nous calculons d'abord la restriction f_{Acc} de la fonction *f* sur l'ensemble *Acc* des états accessibles ainsi défini. Ensuite, nous vérifions que pour tout état il existe une entrée satisfaisant f_{Acc}, ce qui revient à démontrer que la fonction booléenne $\forall q \in Acc\ \forall s \in V_S\ \exists e \in V_E\ f_{Acc}(q, e)$ n'est pas une contradiction.

La représentation du modèle au moyen de fonctions booléennes représentées par des graphes de décision binaires (cf. chapitre 4) limite sensiblement les besoins en mémoire de ces calculs. Néanmoins, leur complexité reste très élevée, si bien qu'ils s'avèrent souvent impossibles en pratique.

3.5.7 Etude du processus de simulation de l'environnement

Nous pouvons maintenant définir des algorithmes de génération de jeux de test dans le cadre de la simulation de l'environnement sur le modèle de générateur que nous venons de construire. Après la présentation de ces algorithmes, nous discutons les aspects d'équité du processus de génération.

3.5.7.1 Génération de données

Nous proposons deux algorithmes, respectivement pour les cas de générateur et de générateur faible. Dans ce dernier cas, il est nécessaire de détecter les éventuelles situations de blocage.

Cas d'un générateur

Soit un générateur $\mathcal{G} = (\mathcal{R}, f)$ où $\mathcal{R} = (Q, S, E, q_{init}, t)$ est une machine E/S associée à un nœud testeur et $f : Q \times V_E \rightarrow \{0, 1\}$ est une fonction génératrice. L'algorithme de génération aléatoire standard est donné par la figure 3–6. Il inclut une fonction *tirage*

```
variables
    q: Q; e: V_E; s: V_S;
début
    q <- q_init;
    faire toujours
        e <- tirage({x ∈ V_E | f(q, x)});
        écrire(e);
        lire(s);
        q <- t(q, e, s);
    finfaire;
fin.
```

Figure 3–6 : Algorithme de génération standard

```
variables
    q: Q; e: V_E; s: V_S;
début
    q <- q_init;
    faire tantque ∃e'∈ V_E f(q, e')
        e <- tirage({x ∈ V_E | f(q, x)});
        écrire(e);
        lire(s);
        q <- t(q, e, s);
    finfaire;
fin.
```

Figure 3–7 : Algorithme de génération pour générateur faible

qui, appliquée à un ensemble, retourne un élément de cet ensemble choisi de manière aléatoire. Notons que, *f* étant génératrice, *tirage* n'est jamais appliquée à l'ensemble vide.

Cas d'un générateur faible

Soit un nœud testeur et soit $\mathcal{R} = (Q, S, E, q_{init}, t)$ sa machine E/S associée. Soit $f : Q \times V_E \rightarrow \{0, 1\}$ une fonction booléenne non nécessairement génératrice. L'algorithme de génération standard n'est pas directement applicable au générateur faible $(\mathcal{R}, f)$ (le cas où la fonction *tirage* s'applique à un ensemble vide n'étant plus exclu). L'algorithme de la figure 3–7 en est une adaptation qui permet d'arrêter la génération dès qu'un blocage survient.

3.5.7.2 Equité de la simulation de l'environnement

Nous nous intéressons dans ce paragraphe à l'étude du comportement du processus de simulation de l'environnement et plus précisément à l'étude de l'équité de l'ensemble des transitions du modèle associé. Rappelons que la propriété de l'équité ("fairness") est définie sur les ensembles de transitions de la manière suivante [Pnu86] :

Soit un ensemble de transitions F. Si au cours d'une exécution les transitions de F peuvent être exécutées un nombre infini de fois, alors elles sont exécutées un nombre infini de fois.

Dans le cadre de la génération aléatoire sous contraintes et plus particulièrement de la simulation de l'environnement nous pouvons reformuler cette définition en considérant que l'ensemble *F* contient la totalité des transitions du modèle :

Si un état est accédé un nombre infini de fois, alors toute entrée satisfaisant, pour cet état, la fonction f_{env} du générateur sera émise un nombre infini de fois.

Nous constatons que l'équité de la simulation de l'environnement dépend de la définition de la fonction *tirage* et plus précisément de la *probabilité* qu'a une entrée $x \in X$ d'être sélectionnée par *tirage(X)*. Cette probabilité peut dépendre de plusieurs facteurs et caractérise l'*environnement opérationnel* du logiciel.

Dans le chapitre 4, nous montrons que les générateurs aléatoires contraints peuvent être implantés de sorte que la fonction *tirage* sélectionne une valeur d'entrée parmi celles satisfaisant les contraintes d'environnement avec *la même probabilité*, ce qui garantit un tirage aléatoire équitable. Dans ce même chapitre, nous expliquons comment des probabilités peuvent être associées de manière individuelle à chaque entrée.

3.5.8 Etude du processus de test des propriétés de sûreté

Nous définissons dans ce paragraphe un algorithme de génération de données d'entrée pour le cas du test des propriétés de sûreté et nous étudions, de la même manière qu'au paragraphe précédent, son équité. Nous montrons, en particulier, que certaines anomalies peuvent survenir lors de la génération dues à l'aspect temporel des spécifications utilisées.

3.5.8.1 Génération de données

La définition formelle du principe du test des propriétés de sûreté énoncé au paragraphe 3.4 est la suivante :

variables
q∈ Q; e∈ V_E; s∈ V_S;
début
q <- q_{init};
faire toujours
*e <- **si** ∃x∈ V_E (f_S(q, x) = 1 ∧ f_E(q, x) = 1) **alors***
tirage({x∈ V_E | (f_S(q, x) = 1 ∧ f_E(q, x) = 1)});
sinon
tirage({x∈ V_E | f_E(q, x) = 1});
écrire(e);
lire(s);
q <- t(q, e, s);
***finfaire**;*
***fin**.*

Figure 3–8 : Algorithme de génération de données testant la propriété f_S

Définition 3-9 : Soit $\mathcal{R}$= *(Q, E, S, q_{init}, t)* une machine E/S associée à un nœud testeur et soit *f* : *Q x V_E x V_S → 2* une fonction booléenne. Une valeur d'entrée *e ∈ V_E teste f dans l'état q∈ Q* si et seulement si ∃s∈ V_S *f(q, e, s) = 0.*

Soit $\mathcal{R}$ = *(Q, E, S, q_{init}, t)* une machine E/S associée à un nœud testeur *N* et soit f_P : *Q x V_E x V_S → 2* la fonction booléenne associée à la conjonction des propriétés de sûreté S_i. Soit la fonction booléenne f_S *: Q x V_E → 2* définie comme suit :

$$\forall q \in Q\ \forall e \in V_E\ (f_S(q,e) = 1 \Leftrightarrow \exists s \in V_S\ f(q, e, s) = 0) \qquad \textbf{(3-e)}$$

Idéalement, nous aimerions pouvoir construire un générateur aléatoire contraint à la fois par la fonction f_S et par la fonction f_E associée aux contraintes d'environnement (i.e. par la fonction $f_S \wedge f_E$). Or, nous n'avons aucune raison de supposer que la fonction f_S est génératrice par rapport à $\mathcal{R}$. En effet, il est possible qu'à un instant donné de l'exécution, il n'existe aucune entrée testant les propriétés (en d'autres termes il n'existe pas toujours un état successeur *q'* de l'état courant *q* tel que *q, q'* $\vdash_s$ *N*). Par conséquent, $f_S \wedge f_E$ n'est pas nécessairement génératrice. Au contraire, f_E est supposée génératrice (il existe toujours un état successeur *q'* de l'état courant *q* tel que *q, q'* $\vdash$ *N*) et donc ($\mathcal{R}$, f_E) est un générateur. L'algorithme de la figure 3–8 permet, à partir de ce générateur, d'engendrer des entrées qui satisferont $f_S \wedge f_E$ chaque fois que cela est possible. Dans le cas opposé, les entrées satisferont simplement f_E.

3.5.8.2 Equité du test des propriétés de sûreté

Nous pouvons définir l'équité du test des propriétés de sûreté de la même manière que pour la simulation de l'environnement. Dans ce cas, l'équité dépendra également de l'implantation de la fonction *tirage*.

Toutefois, l'équité de ce type de test peut être aussi définie par rapport aux *propriétés testées* :

Si une propriété peut être testée un nombre infini de fois pendant une exécution, alors des entrées la testant seront émises un nombre infini de fois.

Cette nouvelle définition de l'équité revient à considérer plusieurs ensembles de transitions F_i, chacun contenant les transitions testant une propriété de sûreté P_i. Par ailleurs, les exécutions du logiciel sous test ne sont pas infinies en pratique. Il serait intéressant pour le processus de test de garantir que toutes les propriétés de sûreté seront testées après un nombre raisonnable d'exécutions de données de test de longueur réaliste.

La définition du test des propriétés de sûreté que nous avons donnée dans ce chapitre porte sur la conjonction des propriétés de l'opérateur ***safety***. Cette démarche présente l'avantage de tester l'ensemble des propriétés avec les mêmes données de test mais, *a contrario*, elle n'offre aucune certitude que toutes les propriétés seront testées. En particulier, il se peut que certaines propriétés ne soient jamais testées. Cette situation peut se produire dans les cas suivants :

- Les entrées testant une propriété sont spécifiques et en petit nombre : leur probabilité d'occurrence peut donc être très faible.
- Le choix d'entrées testant une propriété peut s'opposer au test d'autres propriétés.

Ce dernier cas est illustré par l'exemple suivant :

Exemple 3-3

Considérons de nouveau le logiciel de contrôle de climatiseur en supposant, cette fois, que les propriétés de sûreté qu'il doit vérifier sont les suivantes :

- ***not*** *TempSup* ***or*** *FROID*
- ***not*** *(TempOK* ***and pre*** *TempInf)* ***or*** *(INACTIF* ***or*** *CHAUD)*

Suivant l'algorithme de génération de la figure 3–8, la valeur émise pour *(TempInf, TempOK, TempSup)* au premier instant de la génération est *(0, 0, 1)*. En effet, cette valeur est la seule testant la première des deux propriétés (la deuxième ne peut pas être testée à l'instant initial, ***pre*** *TempInf* étant indéfini). Cette même valeur sera émise au

deuxième instant ainsi qu'à tous les autres instant suivants, car, ***pre*** *TempInf* étant faux, le seul moyen de tester une des propriétés est de rendre *TempSup* vrai. Ainsi, la deuxième propriété ne sera jamais testée.

Ce problème est lié à la définition des entrées testant les propriétés de sûreté donnée au paragraphe 3.4.2. Rappelons que suivant ce principe une entrée qui rend *à un instant t* vraies les propriétés de sûreté indépendamment de la valeur des variables de sortie sera ignorée par le processus de génération à cet instant. Or, comme le montre l'exemple ci-dessus, il se peut que la sélection de telles entrées soit nécessaire afin que d'autres entrées puissent ultérieurement tester les propriétés.

Une solution immédiate à ce problème serait d'appliquer de nouveau le processus de test en considérant une propriété de sûreté à la fois en supposant, bien entendu, que chaque propriété, prise individuellement, ne pose pas ce problème.

Un autre moyen d'éviter ces situations (qu'on pourrait qualifier de "blocages" puisqu'elles empêchent l'évolution du système vers des états testant d'autres propriétés) consiste en l'introduction d'un non déterminisme dans le choix de la transition par l'algorithme de génération. Plus précisément, l'algorithme choisira avec une fréquence spécifiée à l'avance (déterminée de manière empirique) des entrées ne satisfaisant pas $f_S \wedge f_E$ mais uniquement f_E même quand la satisfaction de $f_S \wedge f_E$ est possible. Toutefois, nous estimons que ces problèmes méritent une étude plus approfondie que nous n'avons pas menée.

Les problèmes que nous venons de soulever nous ont conduit à envisager une autre manière d'utiliser la spécification des propriétés de sûreté à des fins de test et plus précisément pour *évaluer la qualité* d'une séquence d'entrées produite par le simulateur d'environnement. Cette qualité s'exprime en termes de pouvoir de détection des violations des propriétés de sûreté (défini par la formule 3-e). L'évaluation peut avoir lieu au moyen de l'algorithme de la figure 3-9. A chaque instant de l'exécution, l'algorithme évalue si l'entrée produite par le simulateur d'environnement teste les propriétés de sûreté. De plus il affiche le taux des entrées qui ont testé les propriétés depuis le début de l'exécution .

La figure 3-10 montre le principe de fonctionnement de cet évaluateur de qualité pendant l'opération de test.

3.6 EXTENSION À LA SIMULATION D'UN PROGRAMME SÛR

3.6.1 Objectif

Les techniques de génération aléatoire que nous venons de décrire ont comme objectif de simuler le comportement de l'environnement du logiciel sous test. Une extension intéressante du principe de simulation aléatoire sous contraintes est la *simulation d'un programme sûr*. Cette technique ne consiste plus à simuler le comportement de l'environnement mais celui du logiciel lui-même. On s'appuie pour cela sur la spécification des propriétés de sûreté (cf. paragraphe 3.4). Plus précisément nous définissons un

```
variables
    q∈ Q; e∈ V_E; s∈ V_S;
    total = 0; ts = 0;
début
    q <- q_init;
    faire toujours
        lire(e, s);
        si f_S(q, e) = 1 alors
            ts <- ts + 1;
        total <- total + 1;
        écrire(ts / total);
        q <- t(q, e, s);
    finfaire;
fin.
```

Figure 3-9 : Evaluateur de la qualité d'un jeu de test

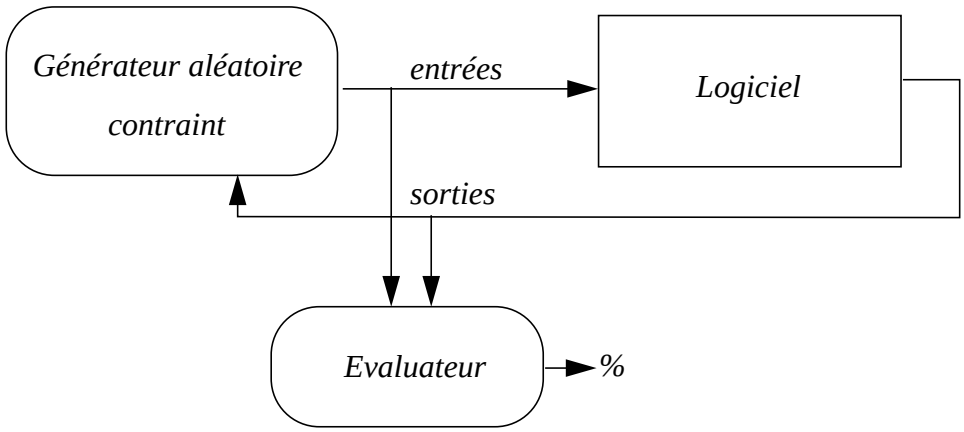

Figure 3-10 : Principe de fonctionnement de l'évaluateur

générateur aléatoire contraint capable de produire les comportements du logiciel satisfaisant ces propriétés.

Afin d'effectuer cette extension de la génération aléatoire sous contraintes, nous reprenons certaines des définitions du paragraphe 3.5 et les adaptons au nouveau type de spécifications considérées.

Pour la spécification des contraintes d'environnement nous avons énoncé certaines restrictions. Ici, aucune restriction n'est imposée sur les expressions booléennes qui peuvent représenter des propriétés de sûreté. Toute propriété est assimilée, de ce fait, à

safenode *N (*e_1*, ... ,* e_p *:* ***bool****)* ***returns*** *(*s_1*, ... ,* s_n *:* ***bool****);*
var l_1*, ... ,* l_q *:* ***bool****;*
let
  safety*(*P_1*, ..., *P_u*);* *-- propriétés de sûreté*

  l_1 *= ...; ...* l_q *= ... ;* *-- définition des variables locales*
tel*;*

Figure 3–11 : Nœud sûr

une fonction booléenne $f : Q \times V_E \times V_S \rightarrow \{0, 1\}$ (contrairement aux contraintes d'environnement qui sont assimilées à une fonction définie uniquement sur l'état et l'entrée courante).

Ce type de simulation peut avoir au moins deux applications intéressantes :

- La simulation du comportement sûr d'un logiciel permet de valider la spécification des propriétés de sûreté. Cela peut être utile en particulier dans le cadre de l'utilisation de cette spécification à des fins de vérification formelle.
- Un logiciel produisant des comportements sûrs peut être intégré dans un composant *tolérant aux fautes*. Il s'agit, plus précisément, de construire une variante qui accompagnera un logiciel principal au sein d'un composant N-autotestable (cf. paragraphe 2.5). Cette variante produit des résultats avec moins de précision que le logiciel d'origine, puisqu'elle se contente de produire une sortie aléatoire satisfaisant les propriétés de sûreté. Le résultat de la variante n'est utilisé qu'en cas de défaillance du logiciel (i.e. violation des propriétés de sûreté).

3.6.2 Syntaxe

La syntaxe de la spécification dans le cadre de la simulation d'un programme sûr est analogue à celle utilisée pour la simulation de l'environnement. Toutefois, nous appellerons le nœud associé *nœud sûr (*'***safenode***'*)*. Par ailleurs, seul l'opérateur ***safety*** est autorisé à apparaître dans ce type de nœud. La syntaxe d'un nœud sûr est donnée par la figure 3–11, où $e_1, \ldots, e_p$ et $s_1, \ldots, s_n$ désignent respectivement les entrées et les sorties du logiciel dont ce nœud simule le comportement.

3.6.3 Sémantique informelle

Un nœud sûr est transformé en un générateur aléatoire contraint qui à tout instant de son exécution produit une sortie aléatoire telle que les propriétés de sûreté de l'opérateur ***safety*** seront satisfaites.

3.6.4 Sémantique formelle

Contrairement à la sémantique de la simulation de l'environnement, la sémantique d'un nœud sûr n'interdit pas l'utilisation pour le calcul des expressions du nœud sûr des valeurs courantes de ses variables d'entrée. Ainsi, la sémantique d'un générateur aléatoire contraint issu de la compilation d'un nœud sûr est formellement définie par des règles identiques à celles définissant la sémantique d'un nœud LUSTRE standard (cf. paragraphe 1.5.1) à l'exception de la règle de compatibilité avec l'opérateur safety suivante :

- Compatibilité avec l'opérateur ***safety*** :

$$\frac{\sigma, \sigma' \vdash P_1 \mid \textbf{\textit{true}} \wedge ... \wedge P_u \mid \textbf{\textit{true}}}{\sigma, \sigma' \vdash \textbf{\textit{safety}}(P_1, ..., P_u)}$$

3.6.5 Extension du modèle

Nous considérons de nouveau une machine E/S $\mathcal{R} = (Q, E, S, q_{init}, t)$ associée au nœud sûr. Nous adaptons la définition d'une fonction génératrice au nouveau profil de la fonction associée aux propriétés de sûreté.

Définition 3-10 : Soit $\mathcal{R} = (Q, E, S, q_{init}, t)$ une machine E/S associée à un nœud sûr N et soit $f : Q \times V_E \times V_S \rightarrow \{0, 1\}$ une fonction booléenne totale. La machine $\mathcal{R}_f = (Q, E, S, q_{init}, t_f)$ est contrainte par la fonction f si et seulement si

$$\forall (q, q') \in Q^2 \; \forall (e, s) \in V_E \times V_S$$
$$(f(q, e, s) = 1 \wedge t(q, e, s) = q') \Leftrightarrow t_f(q, e, s) = q'$$

Les autres définitions du paragraphe 3.5 s'adaptent facilement au nouveau profil de f.

3.6.6 Génération de données

Soit un générateur $\mathcal{G} = (\mathcal{R}, f_P)$ où $\mathcal{R} = (Q, E, S, q_{init}, t)$ est une machine E/S associée à un nœud sûr N et $f_P : Q \times V_E \times V_S \rightarrow \{0, 1\}$ la fonction booléenne associée à la conjonction des propriétés de sûreté S_i. Nous supposons que f_P est génératrice par rapport à $\mathcal{R}$.

Le premier algorithme que nous proposons se contente d'engendrer de manière aléatoire des sorties conformes aux propriétés de sûreté. Il est présenté dans la figure 3-12, et correspond au premier usage imaginé des nœuds sûrs (validation de spécifications).

En revanche, l'algorithme de génération utilisé au sein d'un composant tolérant aux fautes incorpore un test d'acceptation des valeurs de sortie produites par le logi-

```
variables
   q∈ Q; e∈ V_E; s ∈ V_S;
début
   q <- q_init;
   faire toujours
      lire(e);
      s <- tirage({x ∈ V_S | f_P(q, e, x) = 1});
      écrire(s);
      q <- t(q, e, s_1);
   finfaire;
fin.
```

Figure 3-12 : Algorithme de simulation d'un logiciel sûr

```
variables
   q∈ Q; e∈ V_E; s, s_1∈ V_S;
début
   q <- q_init;
   faire toujours
      lire(e,s);
      s_1 <- si f_P(q, e,s) = 1 alors s
         sinon tirage({x ∈ V_S | f_P(q, e, x) = 1});
      écrire(s_1);
         q <- t(q, e, s);
   finfaire;
fin.
```

Figure 3-13 : Variante conservant les propriétés de sûreté

ciel principal. A chaque instant, les entrées et sorties courantes du composant logiciel sont lues et comparées à la spécification des propriétés de sûreté. Si les propriétés sont respectées, la sortie produite par le composant est envoyée vers l'environnement externe. Au contraire, en cas de non respect des propriétés, une valeur de sortie aléatoire est choisie satisfaisant les propriétés de sûreté et est substitué à la sortie défaillante.

Ce type de fonctionnement suppose que la variante principale du composant logiciel est capable de prendre en compte les corrections faites par la variante garantissant le respect des propriétés de sûreté. Cette prise en compte ne nous préoccupe pas ici.

La figure 3-13 donne l'algorithme réalisant une variante destinée à seconder la version complète au sein d'un composant autotestable.

3.7 EXTENSION AUX CONTRAINTES NUMÉRIQUES

L'extension des techniques de génération proposées dans ce chapitre au cas des variables non booléennes ne présente aucune difficulté théorique. Il suffit de considérer un modèle $\mathcal{R} = (Q, E, S, q_{init}, t)$ dans lequel E et S sont des variables de type quelconque. Néanmoins, la représentation en pratique d'un tel modèle est impossible, le nombre des états et la complexité des transitions croissant trop rapidement. Dans le chapitre 4, en même temps que l'implantation des générateurs dans le cas exclusivement booléen, nous présentons un moyen d'étendre cette dernière afin de prendre en compte des contraintes portant sur des variables numériques et nous discutons les conséquences de cette extension sur le processus de génération.

4

Implantation de la génération aléatoire sous contraintes

4.1 INTRODUCTION

Un générateur aléatoire contraint est défini comme une association d'une machine E/S et d'une fonction booléenne définie sur les états, les entrées et éventuellement les sorties de cette dernière. Nous exposons dans ce chapitre les choix d'implantation que nous avons effectués en commençant par la représentation au moyen de graphes de décision binaires de la machine et de la fonction booléenne constituant le générateur. Nous détaillons ensuite la réalisation des algorithmes de génération sur ces structures de données et nous évaluons leur complexité.

Nous explorons ensuite la possibilité d'intervenir explicitement dans l'attribution des probabilités de sélection des valeurs des variables d'entrée et de construire, ainsi, un *profil opérationnel* du logiciel.

En fin de chapitre, nous abordons l'extension de la génération aléatoire aux contraintes utilisant des variables de type entier.

4.2 REPRÉSENTATION DE LA MACHINE E/S

Rappelons que la machine $\mathcal{R} = (Q, E, S, q_{init}, t)$ associée à un nœud testeur est telle que :

- E est l'ensemble de variables de sortie du nœud testeur,
- S est l'ensemble de ses variables d'entrée,
- Q est l'ensemble d'états, correspondant aux mémoires booléennes du nœud testeur,

- $q_{init} \in Q$ est l'état initial,
- $t : Q \times (V_E \times V_S) \rightarrow Q$ est la fonction de transition.

Ce modèle est encodé suivant le principe adopté pour le modèle classique de programmes LUSTRE exposé au paragraphe 1.5.2. D'après ce principe, les états sont représentés au moyen de variables d'état. Chaque variable d'état correspond à une expression distincte de la forme ***pre*** *x* apparaissant dans le nœud testeur, une variable supplémentaire caractérisant l'état initial. Nous obtenons ainsi un ensemble $K = \{v_1, ..., v_n\}$ de n variables d'état.

La fonction de transition est un vecteur $(t_i)_{i = 1, n}$ de n fonctions booléennes telles qu'une fonction $t_i : V_K \times V_E \times V_S \rightarrow \{0, 1\}$ est associée à chaque variable d'état v_i. Etant donnés l'état courant (i.e. la valeur des variables d'état) et les valeurs des variables d'entrée et de sortie, t_i calcule la valeur que prendra à l'état suivant la variable d'état v_i.

Enfin, l'état initial est représenté par une fonction booléenne $f_{init} : V_K \rightarrow \{0, 1\}$.

Ainsi, l'ensemble du générateur est encodé au moyen de variables et de fonctions booléennes qui constituent un automate booléen. La syntaxe que nous avons adoptée pour cette description est celle de l'outil BAC [Hal94] que nous présentons en annexe A.

Exemple 4-1

Pour illustrer cette transformation, nous donnons dans la figure 4-1 la machine correspondant au nœud testeur de l'exemple 3-1 (associé au logiciel de contrôle de climatiseur introduit au chapitre 1).

Le passage du nœud testeur écrit en LUSTRE à l'automate booléen associé est effectuée de manière entièrement automatique au moyen d'une partie du compilateur LUSTRE [Ray91] adaptée à nos besoins.

4.3 IMPLANTATION DE LA SIMULATION D'ENVIRONNEMENT

4.3.1 Représentation des contraintes d'environnement

Rappelons que les contraintes d'environnement sont assimilées à une fonction booléenne f_{env} définie sur les variables d'état et les entrées du modèle. Nous représentons cette fonction à l'aide d'un BDD (cf. paragraphe 1.5.5). Sur un tel BDD il est facile de constater si les restrictions concernant la dépendance des propriétés invariantes de la valeur courante des sorties (cf. paragraphe 3.3.2) sont respectées. Il suffit pour cela qu'aucun des nœuds du graphe ne soit associé à une variable de sortie.

```
state                    -- variables d'état
   v0,v1,v2,v3,v4;

inputs                   -- entrées du générateur
   ARRETE, INACTIF, CHAUD, FROID ;

outputs                  -- sorties du générateur
   Marche, TempSup, TempOK, TempInf;

initial                  -- état initial
   v0 and ( not v1) and ( not v2) and ( not v3) and ( not v4);

transition               -- fonctions de transition partielles
   v0' = (v0 and ( not  1 ));

   v1' = (TempInf or (not v0 and v1));

   v2' = if TempInf then TempOK
         else not (not v0 and v1) or (TempOK or v2)
         fi ;

   v3' = (TempSup or (not v0 and v3));

   v4' = if TempSup then TempOK
         else not (not v0 and v3) or (TempOK or v4)
         fi ;
```

Figure 4-1 : Machine associée au nœud testeur *EnvClim*

Nous imposons pour la construction de ce BDD un ordre d'expansion < tel que pour toute variable d'état v_i et pour toute variable d'entrée x on ait $v_i < x$. Cela signifie simplement que les nœuds associés aux variables d'état sont ceux de la partie supérieure du BDD tandis que les variables d'entrée correspondent aux nœuds de la partie inférieure. Cette règle ne définit pas de manière unique l'ordre d'expansion puisqu'elle n'impose aucune contrainte sur la relation entre les variables d'état ou entre les variables d'entrée. Bien que nous n'ayons envisagé dans ce travail aucun critère particulier pour ce choix, ce critère doit inclure le souci de minimisation de la taille des BDD produits.

Exemple 4-2

Considérons le graphe associé à l'opérateur ***environment*** du nœud testeur *EnvClim* (cf. figure 3–4), donné par la figure 4-2. Pour une meilleure lisibilité, pour chaque nœud associé à l'expansion d'une fonction f par rapport à une variable x nous avons

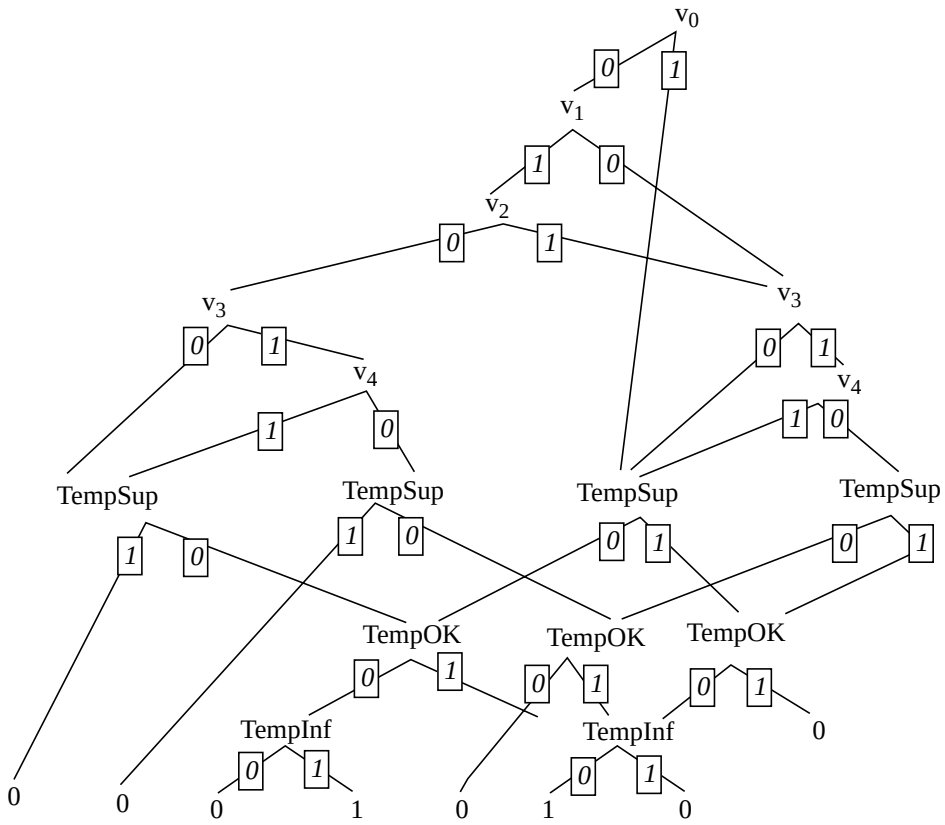

Figure 4-2 : BDD de l'environnement du logiciel de climatisation

étiqueté par *0* ou *1* les arcs menant respectivement aux cofacteurs $f_{\overline{x}}$ et f_x. Pour cette même raison, nous n'avons pas pris en compte le partage des feuilles du BDD.

4.3.2 Identification d'une fonction génératrice

Suite à la définition d'une fonction génératrice donnée au chapitre 3, nous présentons brièvement l'implantation des deux étapes qui sont nécessaires pour déterminer si une fonction *f* possède cette propriété. La première étape consiste en le calcul de l'ensemble *Acc* des états accessibles de la machine contrainte par *f* la deuxième étant l'exploration de la restriction de *f* à cet ensemble.

4.3.2.1 Calcul des états accessibles

L'ensemble *Acc* des états accessibles de la machine $\mathcal{R}_f = (Q, E, S, q_{init}, t_f)$ contrainte par *f* est un plus petit point fixe :

$$Acc = \mu X.\ q_{init} \cup post_f(X)$$

En d'autres termes, $Acc = lim\ Acc_i$ où $Acc_0 = q_{init}$ et $Acc_{i+1} = Acc_i \cup post_f(Acc_i)$. Le calcul de chaque Acc_i peut s'effectuer en construisant explicitement le BDD associé à la fonction $post_f$.

Une autre méthode plus efficace est proposée dans [Hal94]. Elle repose sur l'utilisation de l'opérateur booléen "constrain" qui permet de réduire considérablement le nombre de variables des BDD manipulés et, par ce biais, la taille de ces derniers.

4.3.2.2 Calcul de la restriction de la fonction contraignant $\mathcal{R}$

Pour restreindre la fonction f à l'ensemble Acc des états accessibles de $\mathcal{R}_f$ il suffit d'utiliser l'opérateur booléen de restriction ("restrict"). Si f_A est la fonction caractéristique de l'ensemble Acc alors $f \downarrow f_A$ est la restriction de f recherchée (i.e. f_{Acc}).

Une fois la fonction f_{Acc} calculée, déterminer si f est génératrice revient à démontrer que pour tout état q de Acc il existe une entrée e telle que $f_{Acc}(q, e) = 1$. Cela est effectué en parcourant le BDD associé à f_{Acc} et en vérifiant qu'aucun des nœuds associés à des variables d'état n'a de fils égal à la feuille *0*. Nous supposons, bien entendu, que l'ordre d'expansion de f_{Acc} est le même que celui considéré au paragraphe 4.3.1.

4.3.3 Algorithme de génération standard

Les algorithmes de génération aléatoire proposés dans le chapitre 3 s'implantent de manière efficace sur une telle représentation du générateur. Considérons, en premier, l'algorithme de génération standard (cf. paragraphe 3.5.7.1) :

variables
    q: Q; e: V_E*; s:* V_S*;*
début
    q <- q_{init}*;*
    faire toujours
        e <- tirage({x ∈ V_E *| f(q, x)});*
        écrire(e);
        lire(s);
        q <- t(q, e, s);
    finfaire*;*
fin*.*

Etudions la signification de chaque instruction de cet algorithme sur la représentation du générateur :

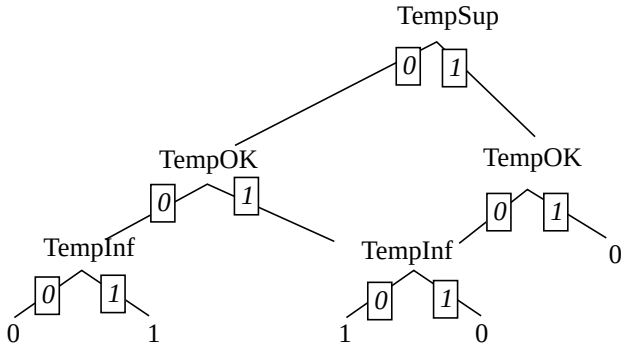

Figure 4-3 : Sous-graphe associé à $v_0 = 1$ (état initial).

1. q <- q_{init}
   Cette instruction correspond au calcul de la valeur des variables d'état vérifiant le prédicat caractérisant l'état initial (partie ***initial*** de la description de l'automate booléen). Pour l'automate de la figure 4-1, on obtient $v_0 = 1$, $v_1 = v_2 = v_3 = 0$.
2. e <- *tirage*($\{e_0 \in V_E \mid f(q, e)\}$)
   Cette instruction se décompose en deux étapes :
   - Dans un premier temps, on calcule à partir du graphe de la fonction f_{env} le sous-graphe correspondant à la valeur courante des variables d'état. Ce sous-graphe ne comporte que des variables d'entrée.
     Par exemple, pour la valeur de la variable d'état $v_0 = 1$ du graphe de la figure 4-2 on obtient le sous-graphe de la figure 4-3.
   - Ensuite, on sélectionne de manière aléatoire une valeur pour les entrées telle que le chemin associé dans ce sous-graphe mène à une feuille portant la valeur booléenne *1* (i.e. telles que la fonction booléenne représentée par le sous-graphe soit satisfaite).
3. *écrire(e)*
   Il s'agit simplement de l'émission de la valeur sélectionnée des variables d'entrée.
4. *lire(s);*
   Après l'émission des entrées, le logiciel synchrone sous test effectue le calcul de la nouvelle valeur de ses variables de sortie. C'est cette valeur qui est lue par cette instruction.
5. q <- *t*(q, e, s)
   A partir de l'état courant, des valeurs courantes des variables d'entrée et des valeurs des variables de sortie calculées par le logiciel, on calcule l'état suivant au moyen de la fonction de transition.

Nous remarquons que la réalisation de l'instruction

$$e \text{ <- } tirage(\{e_0 \in V_E \mid f(q, e)\})$$

comporte deux opérations dont la première (sélection du sous-graphe) s'effectue de manière évidente par une évaluation partielle du graphe de la fonction f_{env} (le coût de cette évaluation est de $O(n_v)$ où n_v est le nombre de variables d'état).

La deuxième opération consiste en la recherche d'un chemin dans ce sous-graphe. Le coût de cette recherche est de ($O(n_e)$, où n_e est le nombre de variables de la fonction f_e associée au sous-graphe. En effet, il suffit de choisir pour chaque nœud du sous-graphe une valeur associée à un cofacteur différent de *0* (un tel cofacteur existe toujours si la fonction décrivant l'environnement est génératrice).

La réalisation des autres instructions est triviale. Le calcul de l'état suivant consiste en une simple évaluation des fonctions partielles de transition. Si ces fonctions sont représentées par des BDD, le coût de ce calcul est de $O(n)$, n étant le nombre de variables d'état, d'entrée et de sortie.

4.3.4 Algorithme de génération pour générateur faible

L'implantation de cet algorithme s'effectue de manière similaire à celle de l'algorithme de génération standard. La seule différence entre les deux implantations réside à l'existence de la condition d'entrée à la boucle. En pratique, il suffit de vérifier l'existence d'une valeur satisfaisant f pendant le parcours du BDD (deuxième opération de la 5ième étape du paragraphe 4.3.3).

4.4 IMPLANTATION DU TEST DES PROPRIÉTÉS DE SÛRETÉ

4.4.1 Représentation des contraintes

L'implantation de cet algorithme nécessite la représentation en BDD des fonctions f_E et $f_S \wedge f_E$. Le tirage aléatoire de valeurs se réalise donc de la même manière que dans le cas de l'algorithme de génération standard en utilisant les BDD associés respectivement à $f_S \wedge f_E$ et à f_E.

4.4.2 Algorithme de génération

Cet algorithme ne diffère de l'algorithme de génération standard que par la manière dont est calculée la valeur des variables d'entrée :

e <- ***si*** $\exists x \in V_E\ (f_S(q, x) = 1 \wedge f_E(q, x) = 1)$ ***alors***
    $tirage(\{e' \in V_E \mid (f_S(q, e') = 1 \wedge f_E(q, e') = 1)\});$
sinon
    $tirage(\{e' \in V_E \mid f_E(q, e') = 1\})$

La sélection d'une valeur aléatoire par la fonction tirage se fait de la même manière qu'au paragraphe 4.3.3 à l'exception du fait que dans le cas où la condition de l'expression conditionnelle est vérifiée, le BDD utilisé est celui représentant la fonction $f_S \wedge f_E$ (au lieu de f_E).

4.5 IMPLANTATION DE LA SIMULATION D'UN PROGRAMME SÛR

De manière analogue à la simulation d'environnement, nous représentons les propriétés de sûreté par un graphe de décision binaire. L'ordre d'expansion < de ce graphe est tel que pour toute variable d'état *v*, pour toute variable d'entrée *x* et pour toute variable de sortie *y* on ait $v < x < y$. En procédant ainsi, l'implantation de l'algorithme de simulation est similaire à celle de l'algorithme de simulation de l'environnement.

4.6 EXTENSION À LA CONSTRUCTION DES PROFILS OPÉRATIONNELS

4.6.1 Profils opérationnels

L'implantation des algorithmes de génération proposée aux paragraphes 4.3, 4.4, 4.5 reste imprécise quant à la loi probabiliste suivant laquelle sont sélectionnées les valeurs aléatoires. Nous remarquons, en effet, que la fonction *tirage* est amenée à sélectionner de manière aléatoire une valeur booléenne pour chaque variable de sortie (quand une des deux valeurs possibles n'est pas interdite). La manière dont sera faite cette sélection est déterminante pour la génération aléatoire puisqu'elle influe sur la nature des séquences d'entrée produites.

Un moyen de réaliser ce choix aléatoire est de considérer que chaque valeur d'entrée a une *probabilité d'être sélectionnée* spécifiée à l'avance. Une telle spécification de probabilités de sélection (ou *probabilités d'occurrence*) constitue un complément de la spécification de l'environnement. Ce nouvel ensemble de spécifications (propriétés invariantes + probabilités d'occurrence) constitue un *profil opérationnel* [Mus93], c'est à dire une description de *l'environnement réel* d'exécution du logiciel. Les profils opérationnels sont d'une grande utilité dans le cadre de la validation des logiciels et en particulier pour l'évaluation de leur fiabilité qui dépend directement de leurs conditions d'utilisation.

La construction d'un profil opérationnel est le produit d'une analyse détaillée des conditions d'utilisation du logiciel et se déroule en cinq étapes [Mus93] :

1. La première étape consiste en l'identification du *profil des clients*. Plus précisément, il s'agit de déterminer les personnes ou organisations qui feront l'acquisition du logiciel et d'associer une probabilité d'occurrence à chacune d'entre elles.
2. Le but de la deuxième étape est de définir le *profil des utilisateurs* c'est à dire d'identifier des groupes d'utilisateurs homogènes du logiciel au sein des organi-

sations identifiées lors de l'étape précédente. Une probabilité d'occurrence doit être associée à chacun de ses groupes.

3. La troisième étape consiste en la définition du *profil des modes* du logiciel. Il s'agit de déterminer des ensembles de fonctions ou opérations (les "modes") que le logiciel doit fournir simultanément. A chacun de ces modes on associe une probabilité d'occurrence qui dépend du profil des utilisateurs déterminé au cours de l'étape précédente.
4. Lors de la quatrième étape, chaque mode identifié est décomposé en fonctions élémentaires du logiciel dont on détermine la probabilité d'occurrence. On constitue ainsi le *profil fonctionnel* du logiciel.
5. Enfin, la dernière étape consiste en l'élaboration du *profil opérationnel* du logiciel, ce qui revient à définir les opérations élémentaires qu'effectuent les utilisateurs à l'aide du logiciel ainsi que leur fréquence (i.e. la manière dont les utilisateurs se servent des fonctions identifiées dans le profil fonctionnel). Le domaine d'entrée du logiciel est divisé en des classes de données disjointes, dont chacune correspond à une opération, et une probabilité d'occurrence est affectée à chacune d'entre elles.

Nous nous intéressons ici uniquement à la mise en œuvre consécutive à la dernière étape de cette construction qui aboutit à l'association de probabilités d'occurrence aux différentes entrées du logiciel. Nous supposons donc que l'analyse préalable nécessaire à la détermination de ces probabilités est accomplie.

Un grand nombre parmi les approches proposées à la construction de profils opérationnels consistent en la simple association de probabilités aux valeurs des variables d'entrée du logiciel de manière *inconditionnelle* (voir par exemple [DN84]). Whittaker [Whi92] a proposé une autre approche qui suggère de considérer que la probabilité d'occurrence d'une entrée à l'instant *t* dépend d'une condition portant sur la valeur des entrées à l'instant *t-1*. Cela revient à décrire le profil opérationnel sous la forme d'une chaîne de Markov.

Dans [Woi93] on trouve une approche plus générale permettant d'associer des probabilités qui peuvent dépendre de conditions portant sur un passé plus lointain des entrées. Suivant cette approche, le domaine d'entrée du logiciel est divisé en classes d'équivalence. De même, les séquences de valeurs passées des variables d'entrée sont divisées en classes d'équivalences appelées "histoires". Une probabilité d'occurrence est ensuite affectée à chaque couple *(c, h)* formé d'une classe d'entrées *c* et d'une histoire *h*. Par exemple, si la probabilité d'occurrence p_1 est associé au couple (c_1, h_1), cela signifie que si, à un moment de l'exécution donné, les valeurs passées des variables d'entrée font partie de l'histoire h_1 alors la valeur des variables d'entrée à cet instant appartiendra à la classe c_1 avec une probabilité p_1. La définition des classes d'entrées et des histoires ainsi que l'association des probabilités se font simplement au moyen d'une table.

Nous étudions dans la suite comment des probabilités d'occurrence peuvent être d'abord spécifiées et ensuite intégrées dans le processus de génération aléatoire [OP95a]. Cela devrait permettre la mise en place de simulations réalistes de l'environnement d'exécution du logiciel et la définition, par ce biais, d'un moyen utile d'évaluation de sa fiabilité.

Inspirés des travaux cités plus haut, nous considérons deux types d'association de probabilités d'occurrence, inconditionnelle et conditionnelle (on parlera respectivement de *probabilités conditionnelles* et *probabilités inconditionnelles*). Dans les deux cas, l'attribution de probabilités aux entrées se fait simplement au moyen de tables d'association. Les conditions nécessaires à l'association de probabilités du deuxième type sont des expressions booléennes LUSTRE. En particulier, l'utilisation des opérateurs temporels de LUSTRE permet d'exprimer des conditions portant sur des valeurs passées des entrées ou sorties du logiciel sans se limiter aux valeurs prises à l'instant précédant immédiatement l'instant courant.

Avant de traiter, aux paragraphes 4.6.3 et 4.6.4 ces deux cas d'association de probabilités, nous exposons au paragraphe 4.6.2 une technique qui permet de doter un générateur aléatoire contraint d'une propriété intéressante, à savoir la capacité d'effectuer le choix d'une valeur d'entrée à tout instant de l'exécution de manière *équiprobable*. Cette technique n'affecte pas le coût de la construction du générateur ou celui des algorithmes de génération et permet, par ailleurs, de disposer d'une implantation standard équitable (cf. paragraphe 3.5.7.2).

4.6.2 Génération équiprobable

4.6.2.1 Objectif

L'objectif de la génération équiprobable est de garantir qu'à chaque instant de l'exécution, la fonction *tirage* choisira une valeur pour les variables d'entrée de manière équiprobable. A cet effet, nous définissons un principe d'étiquetage du BDD représentant les fonctions booléennes f_E (associée aux contraintes d'environnement) et $f_S \wedge f_E$ (utile pour le test des propriétés de sûreté) et nous proposons une implantation de la fonction *tirage* appropriée.

4.6.2.2 Etiquetage du graphe

Tous les nœuds du graphe représentant une fonction booléenne *f* qui sont associés à des variables d'entrée sont étiquetés par un couple (v_0, v_1) de valeurs *entières positives ou nulles*. Ces valeurs sont définies de la manière suivante :

Soient e_i $(1 \leq i \leq p)$ les variables d'entrée du graphe. On suppose de nouveau que l'ordre d'expansion $<_{exp}$ utilisé est tel que $i < j \Leftrightarrow e_i <_{exp} e_j$. Soit e_t une entrée associée à un nœud *n*. Alors, v_1 (resp. v_0) est le nombre de valeurs distinctes du vecteur $(e_t = 1, e_{t+1}, ..., e_p)$ (resp. $(e_t = 0, e_{t+1}, ..., e_p)$) qui correspondent à des chemins du graphe

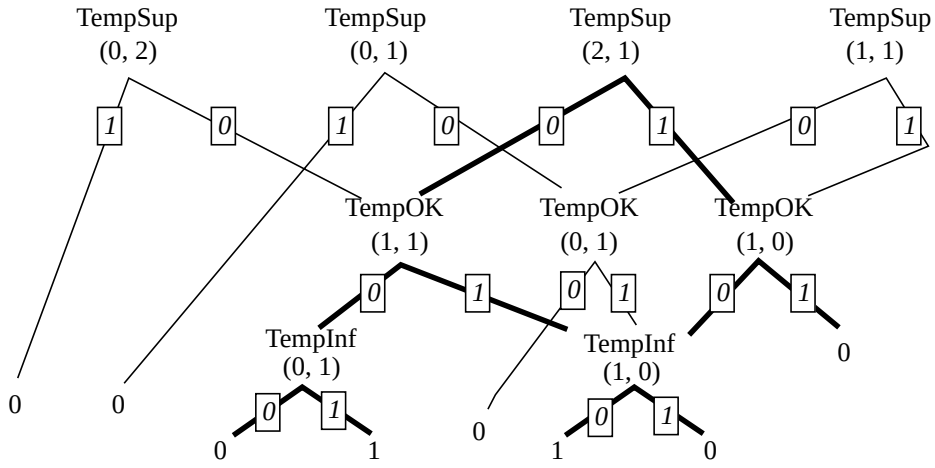

Figure 4-4 : Graphe étiqueté

de racine *n* menant à une feuille *1*. Plus formellement, si *f* est la fonction associée au graphe de racine *n* :

$$v_1 = card(\{(e_{t+1}, ..., e_p) \mid f_{e_t}(e_{t+1}, ..., e_p) = 1\})$$
$$v_0 = card(\{(e_{t+1}, ..., e_p) \mid f_{\overline{e_t}}(e_{t+1}, ..., e_p) = 1\})$$

Ce nouvel étiquetage du graphe se fait au moyen d'un parcours récursif de coût *O(|f|)*.

Exemple 4-3

La version étiquetée du diagramme de la figure 4-3 suivant ce principe est donnée dans la figure 4-4. Dans cette figure, par souci de simplicité, nous n'avons conservé que les nœuds correspondant aux variables d'entrée. Par ailleurs, nous donnons pour chaque nœud soit la valeur de (c_0, c_1), soit celle de (c_1, c_0) en fonction de la position des arcs sortants.

Considérons le sous-graphe en traits épais dont la racine est associée à *TempSup* et porte l'étiquette *(v0, v1) = (2, 1)*. Nous remarquons en effet que dans le sous-graphe associé à *TempSup = 0* il existe deux valeurs pour *(TempOK, TempInf)* menant à une feuille *1*, à savoir *(0, 1)* et *(1, 0)*. Dans le cas où *TempSup = 1*, il existe une valeur de *(TempOK, TempInf)* menant à une feuille *1*, à savoir *(0, 0)*.

4.6.2.3 Algorithme de génération

Afin de garantir, à un instant donné, la même probabilité de sélection pour toutes les valeurs d'entrée satisfaisant la fonction *f* (i.e. permettant d'atteindre une feuille *1*), la fonction *tirage* sélectionnera, pour toute variable e_t étiquetée par (v_0, v_1), une valeur *0* ou *1* avec une probabilité $p_0(e_t)$ ou $p_1(e_t)$ telle que :

```
procédure tirage(BDD : f) ;
début
   si f = 0 alors
      erreur                          -- rien à engendrer
   sinon si f   1alors
      si f.v0 = 0 alors               -- valeur obligatoire = 1
         f.valeur <- 1
      sinon si f.v1 = 0 alors         -- valeur obligatoire = 0
         f.valeur <- 0
      sinon                           -- choix aléatoire d'une valeur booléenne
         f.valeur <- 1 avec une probabilité de f.v1 / (f.v1 + f.v0)
      si f.valeur = 0 alors
         tirage(f0)                   -- appel récursif sur le cofacteur
      sinon
         tirage(f1)
fin
```

Figure 4-5 : Implantation de la fonction tirage

$$p_1(e_t) = v_1 / (v_1 + v_0)$$
$$p_0(e_t) = v_0 / (v_1 + v_0)$$

L'implantation de la fonction *tirage* est donnée par la figure 4-5. Notons que le cas $f.v_1 + f.v_0 = 0$ ne peut pas se produire au moment de l'exécution de l'instruction ci-dessus. En effet, ce cas correspond à $f = 0$ et est détecté par la fonction *tirage* avant tout calcul de probabilités.

Exemple 4-4

Pour illustrer cet algorithme, considérons de nouveau le sous-graphe en trait épais de la figure 4-4. Le nœud associé à *TempSup* étant étiqueté par *(2, 1)*, l'algorithme de génération donnera à *TempSup* la valeur *1* avec une probabilité de *1/(1+2) = 1/3*. Dans ce cas, *TempOK* et *TempInf* prennent obligatoirement la valeur *0* (leurs étiquettes associées sont égales à *(1, 0)*). Ainsi, la probabilité d'occurrence de la valeur d'entrée *(TempSup, TempOK, TempInf) = (1, 0, 0)* est de *(1/3)x(1/1)x(1/1) = 1/3*. Dans le cas où *TempSup* prend la valeur *0* (la probabilité pour que cela arrive est de *2/(1+2) = 2/3), TempOK* sera mis à *1* ou à *0* avec une probabilité identique de *1/(1+1) = 1/2.* Ainsi, chacune des valeurs *(0, 0, 1)* et *(0, 1, 0)* sera affectée à *(TempSup, TempOK, TempInf)* avec une probabilité de *(2/3)x(1/2)x(1/1) = 1/3*.

TempSup	*TempOK*	*TempInf*
0,4	*0,2*	*0,4*

Figure 4-6 : Association de probabilités inconditionnelles

4.6.3 Spécification de probabilités inconditionnelles

Nous supposons maintenant que les probabilités d'occurrence des valeurs des variables d'entrée sont explicitement spécifiées par l'utilisateur au moyen d'une table comme celle de la figure 4-6. Toutes les variables étant booléennes, cette table ne comporte que la probabilité qu'a une entrée de prendre la valeur *1*. Pour toute variable d'entrée *e*, nous désignerons par *prob(e)* cette probabilité.

L'introduction de ce type de probabilités ne demande pas un étiquetage particulier du graphe de *f* mais une modification simple de la fonction *tirage* qui doit prendre en compte les probabilités spécifiées. Plus précisément, cette fonction est modifiée en remplaçant l'instruction :

f.valeur <- 1 avec une probabilité de $f.v_1 / (f.v_1 + f.v_0)$

par l'instruction :

f.valeur <- 1 avec une probabilité de prob(f.var)

Cependant, ce nouvel algorithme ne prend pas en compte le *taux effectif* d'occurrence des entrées. En effet, les valeurs p_0 et p_1 sont prises en compte pour l'attribution d'une valeur à une variable *x* uniquement quand cela est possible, c'est à dire quand les valeurs de v_0 et v_1 sont toutes les deux non nulles. Dans le cas opposé, la probabilité spécifiée sera ignorée. Il se peut donc que le taux effectif d'occurrence d'une entrée présente un écart important par rapport à la probabilité spécifiée.

Une amélioration de cet algorithme permettant la prise en compte de ce taux effectif consiste en l'association à chaque variable d'entrée d'une information supplémentaire consistant en un couple (t_0, t_1) de valeurs entières positives ou nulles telles que t_0 (resp. t_1) est le nombre de fois que la variable a pris la valeur *0* (resp.*1*) depuis le début de la génération. Pour toute variable d'entrée *e*, nous désignerons ces nombres respectivement par $t_0(e)$ et $t_1(e)$.

L'implantation de la fonction *tirage* issue de cette modification est donnée par la figure 4-7. Cet algorithme va tenter de rendre le taux d'occurrence effectif des valeurs des variables d'entrée le plus proche possible de la probabilité d'occurrence spécifiée. Il n'est cependant pas garanti qu'à la fin de la génération ces deux taux seront identiques.

```
procédure tirage(BDD : f) ;
début
   si f = 0 alors
      erreur                          -- rien à engendrer
   sinon si f   1alors
      si f.v0 = 0 alors               -- valeur obligatoire = 1
         f.valeur <- 1
      sinon si f.v1 = 0 alors         -- valeur obligatoire = 0
         f.valeur <- 0
      sinon
         f.valeur <- si p1 < t1(f.var) / (t1(f.var) + t0(f.var)) alors 0
                  sinon si p1 > t1(f.var) / (t1(f.var) + t0(f.var)) alors 1
                  sinon 0 (resp. 1) avec une probabilité de f.v0 / (f.v1 + f.v0)
                     (resp. f.v1 / (f.v1 + f.v0))
      si f.valeur = 0 alors
      t0(f.var) <- t0(f.var) + 1
      tirage(f0)                      -- appel récursif sur le cofacteur
      sinon
         t1(f.var) <- t1(f.var) + 1
         tirage(f1)
fin
```

Figure 4-7 : Nouvelle implantation de la fonction *tirage*

TempSup	*TempOK*	*TempInf*
0,6	*0,6*	*0,6*

Figure 4-8 : Exemple d'association incohérente de probabilités

Notons, par ailleurs, qu'un problème intéressant que nous n'avons pas abordé dans le cadre de ce travail est la *vérification de la cohérence* des probabilités spécifiées. Il est en effet possible que des incohérences soient présentes dans cette spécification, en particulier quand les probabilités d'occurrence des variables ne sont pas indépendantes. Cette situation est illustrée par l'exemple suivant :

Exemple 4-5

Considérons la table d'association de probabilités d'occurrence de la figure 4-8. Il est évident qu'étant donné que la spécification de l'environnement stipule qu'une et une

seule des entrées *TempSup*, *TempOK* et *TempInf* est vraie à un instant donné, la somme de leurs probabilités d'occurrence devrait être égale à *1*, ce qui n'est pas le cas.

4.6.4 Spécification de probabilités conditionnelles

L'association de probabilités conditionnelles est une généralisation de la technique précédente. Elle consiste à considérer que la probabilité d'occurrence d'une entrée dépend d'une condition. L'entrée prendra une valeur *1* avec la probabilité spécifiée uniquement si la condition associée est vraie. Dans le cas où aucune condition concernant une variable n'est vraie, cette variable prendra une valeur suivant le principe de sélection équiprobable du paragraphe 4.6.2. Les conditions dont dépendent les probabilités sont des expressions booléennes LUSTRE quelconques.

Exemple 4-6

Dans l'exemple du logiciel de contrôle du climatiseur, une telle association pourrait être celle de la figure 4-9. Selon cette table, la probabilité pour que l'entrée *TempSup*

Condition	*TempSup*	*TempOK*	*TempInf*
***pre** TempSup*	*0,8*	*0,2*	*0,0*
***pre** TempOK*	*0,45*	*0,1*	*0,45*
***pre** TempInf*	*0,0*	*0,2*	*0,8*

Figure 4-9 : Association de probabilités conditionnelles

soit vraie est plus grande (*0,8*) quand sa valeur précédente est également vraie. Notons également que, conformément à la spécification de l'environnement, la probabilité pour que *TempInf* devienne vrai quand la valeur précédente de *TempSup* est vrai est nulle. Dans le cas opposé, il s'agirait d'une incohérence dans la spécification des probabilités.

Les incohérences comme celle évoquée dans l'exemple 4-6 peuvent être facilement détectées. En effet, supposons que quand la condition *C* est vérifiée la probabilité d'occurrence de la valeur x_0 d'une variable *x* est non nulle. Supposons que la spécification des contraintes d'environnement est donnée par l'expression booléenne *I*. La probabilité d'occurrence spécifiée est cohérente si et seulement si la conjonction $I \wedge C \wedge (x = x_0)$ n'est pas une contradiction (i.e. différente de faux).

En revanche, le problème de spécifications incohérentes relatif aux variables dépendantes évoqué au paragraphe 4.6.3 est également présent dans le cas des probabilités conditionnelles.

4.7 EXTENSION AUX CONTRAINTES NUMÉRIQUES

La technique de génération aléatoire exposée dans les chapitres précédents se limite à la manipulation de variables booléennes. Bien que cette limitation existe également dans le cadre de la vérification formelle de programmes LUSTRE, il est évident qu'une extension au cas des contraintes utilisant des variables numériques serait d'un intérêt incontestable, puisqu'elle permettrait d'élargir sensiblement le champ d'application de nos techniques de test. Ce paragraphe présente les résultats d'une réflexion préliminaire que nous avons menée sur ce problème, dans le but d'identifier les principales difficultés.

Nous supposons dans la suite du paragraphe que le logiciel réactif que l'on veut tester possède des entrées et des sorties de type booléen *ou entier*. Les contraintes d'environnement peuvent donc être soit des formules booléennes, soit des relations sur des variables entières. Les relations que nous avons considérées sont des *inéquations linéaires*.

Nous avons décomposé cette étude[1] en deux parties :

- Dans un premier temps, nous nous sommes intéressés à l'étude d'inéquations ne comportant qu'une variable (i.e. comparaisons à une constante). Dans ce cas, la méthode de génération aléatoire peut s'adapter de manière relativement simple, moyennant une modification des étiquettes des nœuds du BDD représentant les contraintes d'environnement. En revanche, l'équiprobabilité du choix des valeurs numériques est plus difficile à mettre en œuvre.
- Nous nous sommes ensuite intéressés aux inéquations linéaires quelconques, qui est un problème d'une complexité nettement supérieure.

4.7.1 Inéquations linéaires à variable unique

4.7.1.1 Définition d'un modèle

Afin d'associer au nœud testeur un modèle identique à celui utilisé dans le cas purement booléen, nous associons à chaque inéquation linéaire C_i distincte du nœud une variable booléenne c_i vraie uniquement quand l'inéquation est vérifiée. Ces variables peuvent être associées à deux types d'inéquations :

- *x R k* où *x* est une variable entière d'entrée, de sortie ou locale, *k* est une constante et *R* un opérateur de comparaison (<=, <, >=, = ou >). Nous appellerons ce type d'inéquations *conditions simples*.

[1] Cette étude a été réalisée en grande partie dans le cadre du stage de Frédéric Gloppe, étudiant en 1ère année de Magistère d'Informatique de Grenoble, en été 1995.

```
testnode N(...) returns (x, y : int);
let
    environment(
        if (pre x >= 80 or pre x <= 40) then
            if x >= 20 or x <= 10 then
                y<=40 and y >= 20
            else
                y>=40 or y <= 20
        else y >= 0)
tel.
```

Figure 4-10 : Exemple de nœud testeur avec variables numériques

- ***pre****(x) R k* où *x*, *R* et *k* sont définis comme précédemment. Nous appellerons ce type d'inéquations *conditions d'état*.

Ainsi, le modèle associé à un nœud testeur contenant des contraintes numériques est obtenu à partir de celui utilisé en absence de ces dernières en considérant que l'ensemble des variables d'état contient, en plus, les variables booléennes associées aux conditions d'état. En conséquence, nous pouvons considérer de nouveau que l'ensemble des contraintes d'environnement est de nouveau une fonction booléenne qui peut être représentée par un BDD. Les nœuds de ce BDD sont soit des variables booléennes du programme (variables d'entrée, de sortie, locales ou variables d'état) soit les variables booléennes associées aux conditions ci-dessus.

Nous proposons une approche à la génération aléatoire à partir d'une telle structure illustrée sur un exemple simple de nœud LUSTRE, donné par la figure 4-10. Plus précisément, le paragraphe 4.7.1.2 définit l'ordre d'expansion utilisé, tandis que le paragraphe 4.7.1.3 introduit un principe d'étiquetage du BDD obtenu analogue à celui utilisé en absence de variables numériques. Enfin, l'algorithme de génération associé au BDD étiqueté est présenté au paragraphe 4.7.1.4.

Afin de simplifier la présentation, nous considérons pour la suite du paragraphe que les contraintes d'environnement sont exclusivement constituées de conditions simples ou d'état. L'extension de la technique proposée au cas des contraintes utilisant également d'autres variables booléennes est immédiate.

4.7.1.2 Construction du BDD

L'ordre d'expansion du BDD associé aux contraintes d'environnement doit être tel que les nœuds associés aux inéquations linéaires portant sur la même variable occupent des nœuds contigus dans le graphe.

Plus précisément, soit un ordre total $<_v$ sur les variables numériques du nœud tel que pour toutes variables numériques d'entrée *e* et de sortie *s* on ait :

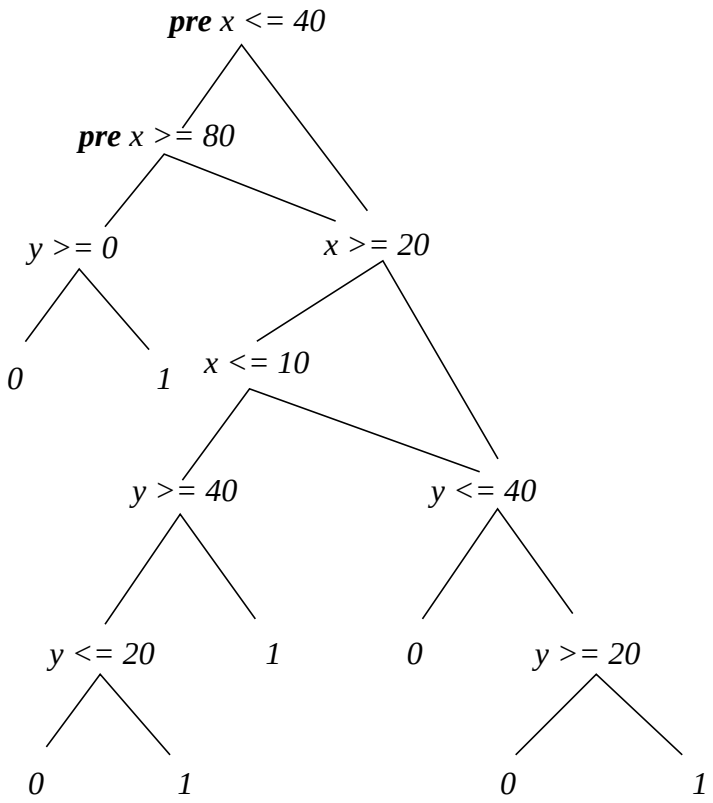

Figure 4-11 : BDD avec contraintes numériques

$$e <_v s$$

Dans ces conditions, pour tout couple d'inéquations *w, w'* portant respectivement sur deux variables v_1 et v_2 distinctes, l'ordre d'expansion $<_{exp}$ du BDD doit être tel que :

- *w* est une condition d'état et *w'* est une condition simple $\Rightarrow w <_{exp} w'$
- *w* et *w'* sont des conditions simples $\Rightarrow (w <_{exp} w' \Leftrightarrow v_1 <_v v_2)$
- *w* et *w'* sont des conditions d'état $\Rightarrow (w <_{exp} w' \Leftrightarrow v_1 <_v v_2)$

Un exemple de BDD associé à la contrainte d'environnement du nœud testeur de la figure 4-10 est donné par la figure 4-11. L'ordre d'expansion de ce BDD est tel que $x <_v y$.

4.7.1.3 Etiquetage du BDD

Un étiquetage des nœuds du BDD est nécessaire afin d'éviter les parcours inutiles et de garantir un temps de génération linéaire par rapport au nombre des variables. Cet étiquetage consiste à associer à chaque nœud l'ensemble des valeurs que peut prendre la

```
procédure Etiqueter(f : BDD)
var ens0, ens1 : ensemble
début
    si f = 0 alors f.ensemble <- ∅
    sinon si f = 1 alors f.ensemble <- ]-∞, +∞[
    sinon
        début
            -- Etiquetage des cofacteurs
            Etiqueter(f0)
            Etiqueter(f1)
            -- Construction des ensembles de valeurs ens0 et ens1 en
            -- synthétisant les informations remontées des cofacteurs
            pour i = 0, 1 faire
            si fi.variable = f.variable alors
                ensi <- fi.ensemble
            sinon si fi.ensemble = ∅ alors
                ensi <- ∅
            sinon
                ensi <- ]-∞, +∞[
            -- Construction de l'ensemble de valeurs de f à partir de
            -- ens0 et ens1 et de la contrainte numérique du nœud courant
            si f.inéquation = "x <= k" alors
                f.ensemble = (ens0 ∩ ]k, +∞[) ∪ (ens1 ∩ ]-∞, k])
            sinon si f.inéquation = "x < k" alors
                f.ensemble = (ens0 ∩ [k, +∞[) ∪ (ens1 ∩ ]-∞, k[)
            sinon si f.inéquation = "x >= k" alors
                f.ensemble = (ens0 ∩ ]-∞, k[) ∪ (ens1 ∩ [k, +∞[)
            sinon si f.inéquation = "x > k" alors
                f.ensemble = (ens0 ∩ ]-∞, k]) ∪ (ens1 ∩ ]k, +∞[)
        fin
fin.
```

Figure 4-12 : Algorithme d'étiquetage

variable qui lui est associée pour que la propriété puisse être préservée. Cet ensemble est exprimé à l'aide d'unions d'intervalles.

L'algorithme d'étiquetage est donné par la figure 4-12 et effectue un parcours complet des nœuds du BDD pour construire les ensembles de valeurs associés. Le résultat de son application sur le BDD de la figure 4-11 est donné par la figure 4-13.

Cet algorithme étiquette chaque nœud suivant le principe suivant :

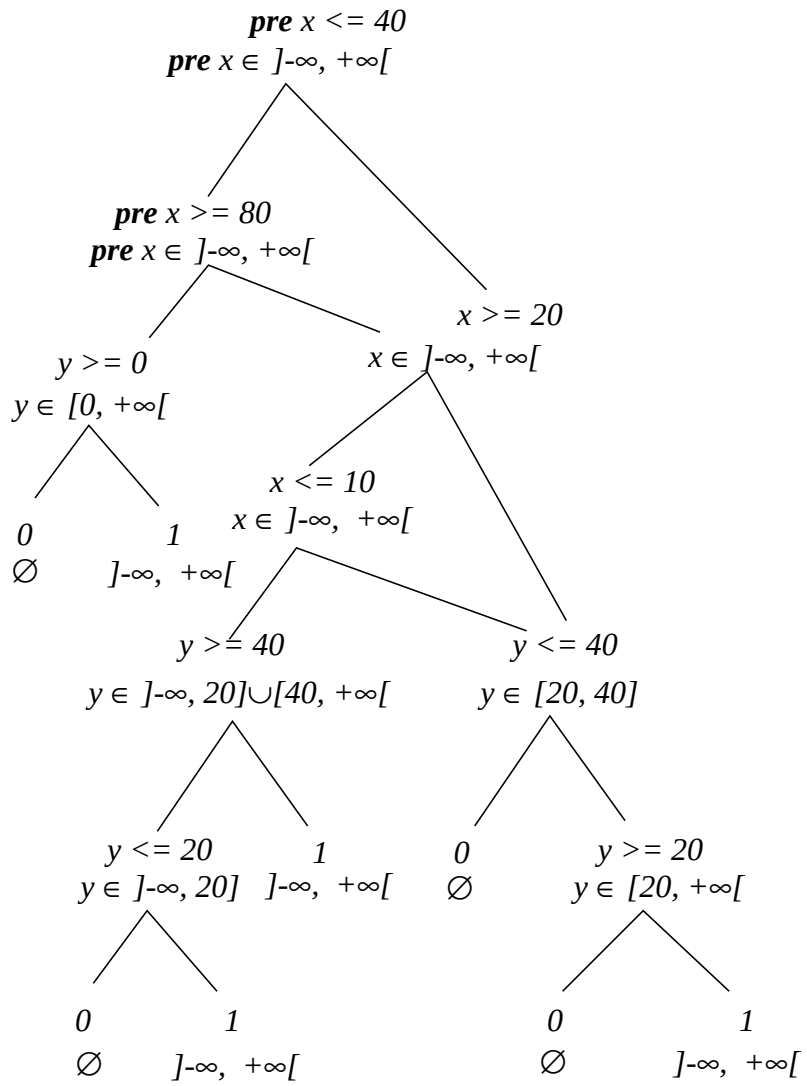

Figure 4-13 : BDD étiqueté

- Les deux cofacteurs sont d'abord étiquetés. Pour calculer l'ensemble de valeurs du nœud courant (*f.ensemble*) nous considérons les cas suivants :
  - La variable du cofacteur gauche (resp. droit) est la même que celle du nœud courant. Dans ce cas, l'ensemble de valeurs du cofacteur est copié dans la variable ens_0 (resp. ens_1). Cet ensemble correspond aux valeurs que la variable du nœud courant peut prendre pour que les contraintes d'environnement soient satisfaites, étant données les inéquations situées aux nœuds inférieurs.
  - La variable du cofacteur gauche (resp. droit) est différente de celle du nœud courant. Dans ce cas, l'ensemble ens_0 (resp. ens_1) est vide, si l'ensemble du

```
procédure tirage(BDD : f) ;
début
   si f.ensemble = ∅ alors
      erreur                    -- rien à engendrer
   sinon
      f.valeur <- x ∈ f.ensemble
      -- Si l'inéquation et vérifiée par la valeur sélectionnée
      si f.inéquation(f.valeur) = 1 alors
         tirage(f0)             -- appel récursif sur le cofacteur gauche
      sinon
         tirage(f1)             -- sinon appel sur le cofacteur droit
fin
```

Figure 4-14 : Tirage de valeurs d'entrée entières

cofacteur associé est vide (cela signifie qu'aucune valeur de la variable du cofacteur ne satisfait les contraintes d'environnement; il est donc inutile de chercher une telle valeur pour les autres variables). Dans le cas opposé (ensemble du cofacteur non vide) l'ensemble ens_0 (resp. ens_1) est considéré comme maximal ($]-\infty, +\infty[$).

- Les ensembles ens_0 et ens_1 ainsi construits sont ensuite restreints afin de prendre en compte l'inéquation du nœud courant.

4.7.1.4 Algorithme de génération

L'algorithme de génération de valeurs est identique à celui utilisé dans le cas booléen (cf. figure 3–6). Seule la fonction effectuant le tirage aléatoire d'une valeur d'entrée est différente.

Considérons la version simple de cette fonction donnée par la figure 4-14. Il est facile de constater qu'aucune disposition n'est prise pour que la valeur engendrée pour le vecteur des variables d'entrée du logiciel sous test soit sélectionnée de manière équiprobable parmi celles satisfaisant les contraintes d'environnement dans un état donné. En effet, la fonction se contente de tirer de manière aléatoire une valeur dans l'ensemble décorant le nœud courant. Ce mode de sélection garantit, néanmoins, qu'aucun retour ne sera effectué pendant le parcours du BDD (temps de parcours linéaire par rapport au nombre de variables).

4.7.1.5 Génération équiprobable

Une sélection équiprobable de la valeur des variables d'entrée du logiciel nécessite un étiquetage plus riche des nœuds du BDD. Plus précisément, il est nécessaire de connaî-

tre pour chaque nœud parcouru et pour chacune des valeurs de l'ensemble qui lui est associé, le nombre *n* de valeurs des variables des nœuds le succédant dans le BDD pour lesquelles les contraintes d'environnement sont satisfaites. Il est facile de constater qu'un tel étiquetage peut être très coûteux, étant donné le nombre de valeurs associées à chaque nœud.

Un moyen de rendre ce coût moins élevé est de représenter cet ensemble de valeurs sous forme d'intervalles tels que le nombre *n* soit identique pour chacun de leurs éléments. Ainsi, l'étiquette de chaque nœud associé à une variable *v* consiste en une liste de couples *(I, n)* où *I* est un intervalle d'entiers et *n* est le nombre de valeurs des variables *v'* telles que $v <_< v'$ pour lesquelles les contraintes d'environnement sont satisfaites quand la valeur de *v* appartient à *I*.

Ce nouveau principe d'étiquetage est illustré sur l'exemple du BDD de la figure 4-11 dont la version étiquetée est proposée dans la figure 4-15. Dans cette figure, I_n dénote le couple *(I, n)* constitué de l'intervalle *I* et de l'entier *n*[1].

L'algorithme effectuant ce nouveau type d'étiquetage est donné par la figure 4-16 où l'on considère que pour tous intervalles *I, I'* et tout entier *n* on a $I_n \cap I' = (I \cap I')_n$.

Soit un nœud du BDD et soit *v* sa variable associée. Soit $(I_j, n_j)_{j=1,p}$ la liste d'intervalles étiquetant le nœud. Soit $p(I_j)_{j=1,p}$ la probabilité pour que la valeur de *v* appartienne à l'intervalle I_j. Pour que la sélection de la valeur des variables d'entrée soit équiprobable il faut que pour tout *j :*

$$p(I_j) = card(I_j) * n_j / \sum_{k=1}^{p} (card(I_k) * n_k)$$

Pour obtenir une génération équiprobable, il suffit, donc, de remplacer dans l'algorithme de la figure 4-14 la ligne :

$$f.valeur <\text{-}\ x \in f.ensemble$$

par la ligne :

$$f.valeur <\text{-}\ x \in I_j \text{ avec une probabilité de } p(I_j) = card(I_j) * n_j / \sum_{k=1}^{p} (card(I_k) * n_k).$$

Nous pouvons constater que la complexité de l'étiquetage ainsi que de l'algorithme de génération dépend de la nature des inéquations qui peuvent amener les ensembles de valeurs associés aux nœuds à être décomposés en un nombre d'intervalles considéra-

[1] Bien entendu, les symboles -∞ et +∞ utilisés dans les figures 4-13 et 4-15 ne dénotent pas l'infini au sens mathématique mais les valeurs respectivement minimale et maximale que peut prendre une variable entière.

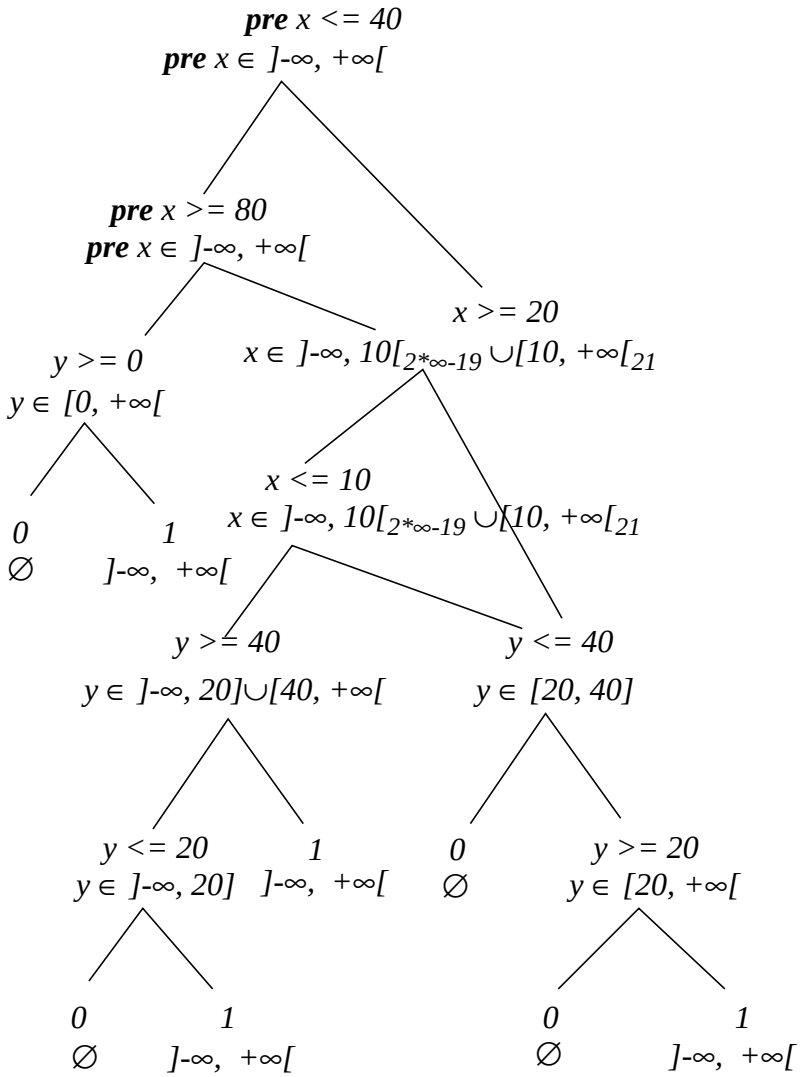

Figure 4-15 : BDD étiqueté pour génération équiprobable

ble. Cette complexité peut être explosive (par exemple, dans le cas ou chaque intervalle possède un seul élément). Nous pensons de ce fait qu'une évaluation de la complexité pratique de cette technique de génération est nécessaire. Cette évaluation ne peut s'effectuer qu'à travers plusieurs études de cas réels que nous n'avons pas pu mener.

4.7.2 Inéquations linéaires à plusieurs variables

Le deuxième type de contraintes que nous avons considérées sont les inéquations linéaires de la forme $\Sigma(\lambda_i x_i + \mu_i$ ***pre***$(x_i)\ R\ 0)$ où λ_i et μ_i sont des valeurs entières, x_i est une variable entière et *R* est un opérateur de comparaison (<=, <, >=, = ou >). Nous

```
procédure Etiqueter(f : BDD)
var   ens0, ens1 : ensemble
      c : entier
début
   si f = 0 alors f.ensemble <- ∅
   sinon si f = 1 alors f.ensemble <- ]-∞, +∞[
   sinon
      début
         -- Etiquetage des cofacteurs
         Etiqueter(f0)
         Etiqueter(f1)
         -- Construction des ensembles de valeurs ens0 et ens1 en
         -- synthétisant les informations remontées des cofacteurs
         pour i = 0, 1 faire
            si fi.variable = f.variable alors
               ensi <- fiensemble
            sinon si fi.ensemble = ∅ alors
               ensi <- ∅
            sinon
               c =  Σ  card(I) * n
                  (I, n) ∈ f0.ensemble
               ensi <- ]-∞, +∞[c
         -- Construction de la liste d'intervalles de f à partir de
         -- ens0 et ens1 et de la contrainte numérique du nœud courant
         si f.inéquation = "x <= k" alors
            f.ensemble = (ens0 ∩ ]k, +∞[) ∪ (ens1 ∩ ]-∞, k])
         sinon si f.inéquation = "x < k" alors
            f.ensemble = (ens0 ∩ [k, +∞[) ∪ (ens1 ∩ ]-∞, k[)
         sinon si f.inéquation = "x >= k" alors
            f.ensemble = (ens0 ∩ ]-∞, k[) ∪ (ens1 ∩ [k, +∞[)
         sinon si f.inéquation = "x > k" alors
            f.ensemble = (ens0 ∩ ]-∞, k]) ∪ (ens1 ∩ ]k, +∞[)
      fin
fin.
```

Figure 4-16 : Algorithme d'étiquetage pour génération équiprobable

```
testnode N(...) returns (x, y : int);
let
   environment(
      if (x >= 80 or x <= 40) then
         if x +2*pre(y) >= 20 or x - 2*y <= 50 then
            y - x <=40 and y + x >= 20
         else
            y + 2*x = 40 or y <= 20
      else y >= 0)
tel.
```

Figure 4-17 : Nœud testeur avec inéquations à plusieurs variables

montrons dans la suite qu'une mise en œuvre de la génération aléatoire sous contraintes analogue à celle adoptée au paragraphe 4.7.1 est beaucoup plus complexe et nous essayons d'identifier les principaux problèmes qu'elle pose. Nous illustrons de nouveau cet exposé sur un exemple simple de nœud de test présenté par la figure 4-17.

Pour représenter la fonction associée aux contraintes d'environnement d'un tel nœud de test nous pouvons adopter, au même titre que pour les inéquations à une seule variable, un graphe de décision binaire dont les nœuds sont décorés soit par les variables booléennes du nœud testeur, soit par les variables d'état, soit enfin par des variables associées aux inéquations. Contrairement au paragraphe 4.7.1 nous ne classons pas les inéquations en conditions simples et d'état. En effet, une même inéquation peut contenir des références à la valeur à la fois courante et passée des variables (au moyen de l'opérateur ***pre***). Nous n'imposons donc aucune contrainte sur l'ordre d'expansion du BDD associé en ce qui concerne les variables entières du nœud testeur. Le BDD associé au nœud testeur de la figure 4-17 est donné par la figure 4-18.

Il est facile de constater que la construction d'ensembles de valeurs décorant les nœuds n'est plus possible, puisque les variables ne sont plus indépendantes. Néanmoins, nous pouvons caractériser l'ensemble des valeurs satisfaisant les contraintes d'environnement par un ensemble de *polyèdres* (un ensemble de systèmes d'inéquations linéaires). Pour l'exemple de la figure 4-18 nous obtenons les polyèdres suivants :

1. *x > 40, x < 80, y >= 0*
2. *x > 40, x>=80, x +2*pre(y) >= 20, y - x <= 40, y + x >= 20*
3. *x <= 40, x +2*pre(y) >= 20, y - x <= 40, y + x >= 20*
4. *x <= 40, x +2*pre(y) < 20, x - 2*y <= 50, y - x <= 40, y + x >= 20*
5. *x > 40, x>=80, x +2*pre(y) < 20, x - 2*y <= 50, y - x <= 40, y + x >= 20*
6. *x <= 40, x +2*pre(y) < 20, x - 2*y > 50, y + 2*x = 40*
7. *x > 40, x>=80, x +2*pre(y) < 20, x - 2*y > 50, y + 2*x = 40*

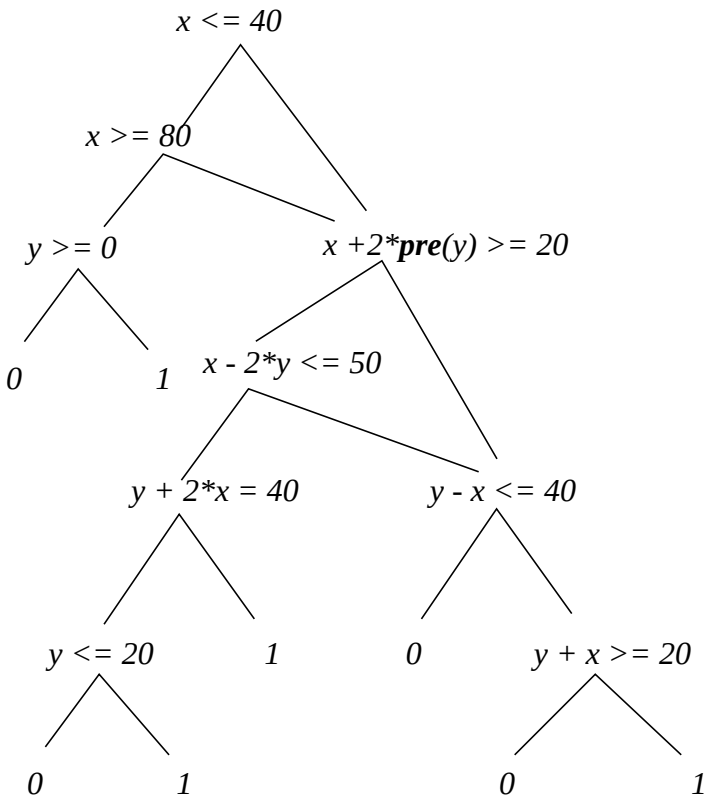

Figure 4-18 : BDD avec inéquations à plusieurs variables

8. *x <= 40, x +2*pre(y) < 20, x - 2*y > 50, y + 2*x 40, y <= 20*
9. *x > 40, x>=80, x +2*pre(y) < 20, x - 2*y > 50, y + 2*x 40, y <= 20*

La construction de cet ensemble de polyèdres peut être effectuée par un parcours en profondeur du BDD. La génération aléatoire sous contraintes revient, donc, à sélectionner une valeur appartenant à au moins un de ces polyèdres.

Les difficultés principales d'une telle mise en œuvre sont les suivantes :

- La construction de l'ensemble des polyèdres est une opération coûteuse, en particulier si le nombre de polyèdres est important.
- La sélection d'une valeur appartenant à un polyèdre nécessite la recherche des solutions du système d'inéquations associée qui dépend du nombre d'équations et des variables. Le temps de calcul peut donc être important.
- Enfin, la mise en œuvre d'une stratégie de sélection équiprobable paraît très difficile, puisqu'il faut pour cela connaître le nombre d'éléments de chaque polyèdre.

5

Test du réseau d'opérateurs

5.1 INTRODUCTION

Nous supposons dans ce chapitre que le programme réactif à tester est écrit en LUSTRE et nous étudions les possibilités de mise en œuvre d'une technique de test structurel. Plus précisément, nous définissons des critères de sélection fondés sur la structure des programmes et nous décrivons les moyens de procéder à la génération automatique de données d'entrée satisfaisant les critères.

5.2 CHOIX D'UN MODÈLE

Les principales techniques de test structurel que nous avons survolées au chapitre 2 sont conçues pour des programmes écrits dans des langages séquentiels. Les critères de couverture proposés pour ces programmes sont définis sur leur graphe de contrôle, modélisation naturelle de leur structure qui représente le flux de contrôle du programme.

Cette représentation n'est pas adaptée aux programmes LUSTRE qui, contrairement aux langages séquentiels, définissent un traitement du flot de données qui les traverse. Cette constatation nous a amené à rechercher un autre modèle que le graphe de contrôle pour représenter la structure d'un programme LUSTRE. Un tel modèle pourrait être l'automate d'états finis produit par le compilateur qui correspond au modèle obtenu par abstraction booléenne du programme. Rappelons que si le nombre d'états est trop important pour que le modèle soit entièrement engendré, l'automate produit a un seul état et une fonction de transition complexe (ce qui correspond à une boucle infinie réalisant une traduction triviale du programme LUSTRE). Des critères de couverture classiques sur les automates tels que la couverture des états ou des transitions,

```
node Front (X: bool) returns (Front_X: bool);
let
    Front_X = X -> (X and not pre(X));
tel
```

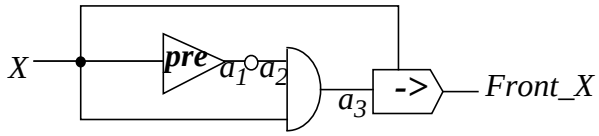

Figure 5-1 : Le nœud *Front* et son réseau d'opérateurs

voire celle des séquences de transitions, peuvent être envisagés sur un tel modèle. Cette approche est adoptée par [Maz94] qui procède au test statistique de l'automate.

Une autre approche consisterait à utiliser le graphe de contrôle du programme C issu de la compilation du programme LUSTRE et qui décrit l'automate associé. Ce modèle présente un certain intérêt parce qu'il permet utiliser les nombreux outils de test disponibles pour le langage C.

Cependant, ces deux modèles présentent un inconvénient commun. En effet, lors de sa compilation, le programme LUSTRE est transformé en un nœud unique "plat" (sans appels d'autres nœuds). Il en résulte que le programme C et l'automate engendrés *ne reflètent plus la structure du programme initial*. Or, les techniques structurelles sont principalement utilisées pour le test unitaire des logiciels, c'est à dire pour le test des fonctions élémentaires ou des modules. Il est difficile de définir des unités analogues sur ces deux modèles.

Notons, toutefois, que [Maz94] a introduit la notion d'*automate d'intégration*, obtenu par simplification de l'automate initial, qui permet la définition du niveau unitaire. Cette définition nécessite l'étude du programme original LUSTRE.

Une dernière critique que nous pouvons formuler concernant le test structurel de l'automate est que les critères définis ne tiennent pas compte des calculs internes des états ou de ceux effectués lors de l'exécution des transitions. Dans [Maz94], cette prise en compte est faite en fin de processus de test et demande une analyse "manuelle" de l'automate.

Dans ce travail nous avons choisi de baser notre étude sur le réseau d'opérateurs qui est associé à un programme LUSTRE (voir par exemple le réseau associé au nœud *Front* de la figure 5-1). Les raisons qui ont motivé ce choix sont les suivantes :

- LUSTRE étant le langage dans lequel sont écrits les programmes, il sera plus facile d'identifier des types de défauts liés à sa mauvaise utilisation et de définir des critères de sélection pertinents, le réseau d'opérateurs reflétant de manière directe la structure du programme à tester.

- Les unités concernées par le test structurel sont très facilement repérables : il s'agit des nœuds composant le programme. Il sera donc simple d'utiliser les techniques définies sur un tel modèle à des fins de test structurel.

Un réseau d'opérateurs étant un type particulier de graphe, nous pouvons définir, au même titre que pour un graphe de contrôle, des critères de sélection exprimés en termes de couverture de ses différents composants. Dans la suite nous donnons l'expression formelle de ces définitions.

5.3 DÉFINITION DES COMPOSANTS DU RÉSEAU D'OPÉRATEURS

Définition 5-1 : Pour tout nœud LUSTRE *N* nous définissons les ensembles suivants :

- *E, S* et *L* sont les ensembles de variables d'entrée, de sortie et locales du nœud.
- *C* est l'ensemble des constantes figurant dans le nœud.
- *A* est un ensemble de variables tel que $A \cap (E \cup S \cup L \cup C) = \emptyset$. *A* est l'ensemble des identificateurs des arcs du réseau qui ne correspondent pas à une variable ou une constante (i.e. ce sont des résultats de calculs intermédiaires).
- $V = A \cup E \cup S \cup L \cup C$ est l'ensemble des *arcs* du réseau.
- *O* est l'ensemble des noms des opérateurs utilisés dans le nœud (opérateurs de base du langage ou autres nœuds).
- Soit *n* et *m* entiers et soit $x \in V^n \times O \times (V \setminus (E \cup C))^m$. On notera *x.entrées*, *x.nom* et *x.sorties* les projections de *x* respectivement sur V^n, *O* et $(V \setminus (E \cup C))^m$.
  x est une *occurrence d'opérateur* de *N* si et seulement s'il existe un opérateur de *N* dont le nom est *x.nom* tel que :
  - l'opérateur a *n* entrées et *m* sorties,
  - *x.entrées* est l'ensemble des arcs d'entrée de l'opérateur dans *N*,
  - *x.sorties* est l'ensemble des arcs de sortie de l'opérateur dans *N*.
- Le *réseau d'opérateurs* R_N associé au nœud *N* est l'ensemble (fini) des occurrences d'opérateurs de *N*.

Exemple 5-1

Dans le cas du nœud *Front* de la figure 5-1 on obtient les ensembles suivants :

- $E = \{X\}$
- $S = \{Front\_X\}$
- $C = \emptyset$

- $A = \{a_1, a_2, a_3\}$
- $O = \{$***pre***, ***not***, ***and***, ***->***$\}$.
- $R_N = \{ (X,$ ***pre***$, a_1), (a_1,$ ***not***$, a_2), (a_2, X,$ ***and***$, a_3), (X, a_3,$ ***->***$, Front\_X) \}$

Définition 5-2 : Soit un réseau d'opérateurs R_N. Un *chemin* de R_N est une séquence finie d'arcs $p = (x_1, ..., x_n) \in V$ telle que :

- $x_1 \subset (E \cup C)$ (le premier arc d'un chemin est une variable d'entrée ou une constante).
- $x_n \subset S$ (le dernier arc d'un chemin est une variable de sortie).
- $\exists (o_1, ..., o_{n-1}) \in R_N^{n-1}$ *tel que* $x_1 \in o_1.entrées \wedge x_{n-1} \in o_{n-1}.sorties \wedge \forall i\ 1 < i < n\text{-}1\ x_i \in o_{i-1}.sorties \cap o_i.entrées$ (toute paire d'arcs consécutifs d'un chemin correspond respectivement à l'entrée et la sortie d'un opérateur du réseau).

Exemple 5-2

Dans le réseau d'opérateurs du nœud *Front* (figure 5-1), nous identifions trois chemins :

- $p_1 = (X, Front\_X)$
- $p_2 = (X, a_1, a_2, a_3, Front\_X)$
- $p_3 = (X, a3, Front\_X)$

Définition 5-3 : Soit un réseau R_N et soit $o \in R_N$ et soient $e \in o.entrées$ et $s \in o.sorties$. Le *prédicat d'arcs* de *e* et *s*, noté *PA*(*e*, *s*), est une fonction $V \times V \setminus (E \cup C) \rightarrow \{0, 1\}$. Elle exprime la condition pour que le calcul de la sortie *s* de l'opérateur dépende de la valeur de l'entrée *e*. Un prédicat d'arcs est une expression booléenne LUSTRE, formellement définie par les règles suivantes :

1. Si *o.nom* = ***not***, alors *PA*(*o.entrées*, *o.sorties*) = ***true***.
2. Si *o.nom* = ***pre***, alors *PA*(*o.entrées*, *o.sorties*) = ***true***.

Le calcul de la valeur de sortie des opérateurs sur lesquels portent les règles précédentes, dépend toujours de la valeur de toutes ses entrées. En revanche, la sortie de l'opérateur ***if then else*** est égale à une de ses entrées en fonction de la valeur de la condition.

3. Si *o.nom* = ***if then else*** et $o.entrées = (e_1, e_2, e_3)$ (i.e. *o* correspond à ***if*** e_1 ***then*** e_2 ***else*** e_3) alors :
   $PA(e_1, o.sorties)$ = ***true***, $PA(e_2, o.sorties) = e_1$ et $PA(e_3, o.sorties)$ = ***not*** e_1.

Les opérateurs ***and*** et ***or*** sont transformés en des expressions conditionnelles équivalentes (e_1 ***and*** e_2 = ***if*** e_1 ***then*** e_2 ***else false*** et e_1 ***or*** e_2 = ***if*** e_1 ***then true else*** e_2) :

4. Si *o.nom* = ***and*** et *o.entrées* = (e_1, e_2) alors *:*
   PA(e_1, *o.sorties*) = ***true*** et *PA*(e_2, *o.sorties*) = e_1.
5. Si *o.nom* = ***or*** et *o.entrées* = (e_1, e_2) alors *:*
   PA(e_1, *o.sorties*) = ***true*** et *PA*(e_2 , *o.sorties*) = ***not*** e_1.
6. Enfin, la sortie de l'opérateur **->** dépend toujours de sa deuxième entrée sauf à l'instant initial (i.e *A* -> *B* est équivalent à ***if*** *init* ***then*** *A* ***else*** *B*). En considérant que *o.entrées* = (e_1, e_2) *:*
   Si *o.nom* = -> alors *PA*(e_1, *o.sorties*) = *init* et *PA*(e_2, *o.sorties*) = ***not*** *init*
   où *init* = ***true -> false*** (i.e *init* est toujours faux à l'exception de l'instant initial).
7. Si *o.nom* ne correspond pas à un des opérateurs du langage, il s'agit d'un nœud. Dans ce cas, nous admettons que $\forall x \in o.entrées\ \forall y \in o.sorties\ PA(x, y)$ = ***true***, c'est à dire que toute sortie d'un nœud dépend de toutes ses entrées.

Cette dernière définition est nécessaire si on veut utiliser cette technique pour le test unitaire d'un nœud sans être obligé de procéder à l'expansion préalable de tous les nœuds qu'il utilise.

Définition 5-4 : Soit *Exp* l'ensemble de toutes les expressions LUSTRE. Pour tout entier $n > 0$ et pour toute expression $E \in Exp$ on définit la fonction $pre^n : Exp \rightarrow Exp$ de la manière suivante :

- $pre^0(E) = E$
- $pre^n(E)$ = ***pre***$(pre^{n-1}(E))$ pour tout $n > 0$.

En d'autres termes, $pre^n(E)$ dénote l'expression LUSTRE obtenue après *n* applications successives de l'opérateur ***pre*** à l'expression *E*.

Définition 5-5 : Soit un réseau R_N et soit $p = (x_1, ..., x_n)$ un chemin du réseau. Le *prédicat de chemin* du chemin *p*, noté *PC*(*p*) est défini comme suit :

$$PC(p) = \mathbf{and}_{i=1...n-1}\ pre^{t_i}(PA(x_i, x_{i+1})) \qquad \textbf{(5-a)}$$

Les valeurs de t_i de la formule ci-dessus sont définis comme suit :

- $t_{n-1} = 0$
- Pour tout $i < n - 1$, $o \in R_N$ étant une occurrence d'opérateur telle que $x_i \in o.entrées$ et $x_{i+1} \in o.sorties$:

$$o.nom = \textbf{pre} \Rightarrow t_i = t_{i+1} + 1$$
$$o.nom \neq \textbf{pre} \Rightarrow t_i = t_{i+1} \quad \textbf{(5-b)}$$

Les prédicats de chemin sont des conditions portant sur la valeur que prennent les arcs à plusieurs instants successifs de l'exécution. De ce fait, un prédicat de chemin *p* définit un ensemble de séquences d'entrées dont l'exécution provoque l'activation de *p*. En d'autres termes, le prédicat du chemin *p* est la condition pour que *p* soit traversé par le flux des données.

Définition 5-6 : Soit *p* un chemin et soit *PC(p)* son prédicat. $Lat(p) = max(\{t_i \mid PC(p) = \textbf{and}_{i=1...n-1}\ pre^{t_i}(PA(x_i, x_{i+1}))\})$ est la *latence* de *p*.

La latence d'un chemin *p* est le nombre d'instants (ou cycles d'exécution) nécessaires à la satisfaction du prédicat de chemin de *p*. En d'autres termes, une séquence d'entrées qui satisfait le prédicat de *p* et dont le premier élément est lu par le programme à l'instant *t* ne pourra avoir un impact sur la sortie du chemin qu'à l'instant *t + Lat(p)*. Il est facile de comprendre que *Lat(p)* est la *longueur minimale* d'une séquence d'entrées satisfaisant *PC(p)*.

Exemple 5-3

Nous illustrons ces définitions sur l'exemple du nœud Front dont les chemins sont $p_1 = (X, Front\_X)$, $p_2 = (X, a_1, a_2, a_3, Front\_X)$ et $p_3 = (X, a3, Front\_X)$.

Conformément aux définitions 5-3 et 5-5, les prédicats de p_1, p_2 et p_3 sont calculés de la manière suivante :

1. D'après la formule 5-a et la règle 6 :
   $PC(p_1) = PC((X, Front\_X)) = PA((X, Front\_X)) = init$
2. D'après la formule 5-a :
   $PC(p_2) = PC((X, a_1, a_2, a_3, Front\_X)) =$
   $\textbf{pre}\ PA(X, a_1)\ \textbf{and}\ PA(a_1, a_2)\ \textbf{and}\ PA(a_2, a_3)\ \textbf{and}\ PA(a_3, Front\_X).$
   D'après la règle 2, $PA(X, a_1) = \textbf{true};$
   d'après la règle 1, $PA(a_1, a_2) = \textbf{true};$
   d'après la règle 4, $PA(a_2, a_3) = X;$
   d'après la règle 6, $PA(a_3, Front\_X) = \textbf{not}\ init.$
   Donc, $PC(p_2) = X\ \textbf{and not}\ init.$
3. D'après la formule 5-a :
   $PC(p_3) = PC((X, a3, Front\_X)) = PA(X, a_3)\ \textbf{and}\ PA(a_3, Front\_X).$
   D'après la règle 4, $PA(X, a_3) = a_2 = \textbf{not}\ a_1 = \textbf{not pre}(X);$
   d'après la règle 6, $PA(a_3, Front\_X) = \textbf{not}\ init.$
   Donc, $PC(p_3) = \textbf{not pre}(X)\ \textbf{and not}\ init.$

En récapitulant :

- $PC(p_1) = PP((X, Front\_X)) = init$
- $PC(p_2) = PP((X, a_1, a_2, a_3, Front\_X)) = X$ ***and not*** *init*
- $PC(p_3) = PP((X, a3, Front\_X)) =$ ***not pre****(X)* ***and not*** *init.*

$PP(p_1)$ stipule que le chemin p_1 peut être traversé seulement à l'instant initial d'une exécution. En revanche, les chemins p_2 et p_3 ne peuvent être traversés qu'à des instants autres que le premier, la condition X (respectivement ***not pre****(X)*) devant également être vérifiées. De manière évidente : $Lat(p_1) = 1$ et $Lat(p_2) = Lat(p_3) = 2$.

5.4 DÉFINITION D'UNE TECHNIQUE DE TEST STRUCTUREL

5.4.1 Objectifs

Une technique de test consiste en la définition à la fois d'un critère de sélection et d'une méthode de génération de données de test satisfaisant ce critère. Dans le paragraphe 5.4.3 nous étudions les moyens d'engendrer de manière automatique des données de test pour les critères définis au paragraphe 5.4.2.

5.4.2 Définition de critères de sélection

5.4.2.1 Signification des critères

Comme nous l'avons vu au chapitre 1, un programme LUSTRE réalise une fonction associant à une séquence d'entrées une séquence de valeurs pour ses variables de sortie. Il s'agit en réalité d'un vecteur de fonctions dont chacune calcule la valeur d'une seule variable de sortie et exprime ainsi l'ensemble de *dépendances* entre cette variable de sortie et les variables d'entrée. Ce sont ces dépendances que nous essayons de mettre en évidence en définissant des chemins dans le réseau d'opérateurs. Elles peuvent être assimilées à des *fonctions partielles* à une seule variable intervenant dans le calcul de la sortie.

La définition de critères de sélection de données de test en termes de couverture de certains chemins du réseau d'opérateurs s'identifie à celle des fonctions significatives parmi les fonctions réalisées par les chemins. La sélection de telles fonctions est difficilement automatisable. Ainsi, nous nous contentons, au même titre que dans le cadre du test structurel des programmes séquentiels (cf. chapitre 2), de définir dans ce paragraphe des critères visant la couverture des différents composants élémentaires du réseau, en commençant par ses opérateurs et ses arcs. En exigeant la couverture de ces composants, les critères fournissent un moyen rudimentaire de forcer le test de toutes les fonctions partielles réalisées par les chemins (ou au moins de tenter de le faire).

Ce type de test est similaire à des techniques appliquées au test des circuits digitaux. Cependant, les fautes recherchées dans ce dernier cas sont clairement identifiées ("collages" de certains fils) [ABF90].

5.4.2.2 Types de test visés

La possibilité d'appliquer ces critères à un nœud dépend bien entendu de sa taille et de sa complexité (nombre d'opérateurs logiques, imbrication d'expressions conditionnelles) qui sont directement liées au nombre et à la complexité de ses chemins. Pour cette raison nous préconisons l'utilisation de ces critères pour le test unitaire de programmes LUSTRE, c'est à dire le test des nœuds constituant le programme, pris individuellement. Il s'agit là d'une définition imprécise du niveau unitaire puisque la taille de ces nœuds peut varier de manière considérable. La seule solution à ce problème est la définition de normes de programmation limitant la taille des nœuds et facilitant ainsi leur test.

Rappelons que le calcul des prédicats de chemin d'un nœud (cf. définitions 5-3 et 5-5) tient compte de ce fait, puisque le traitement des appels à d'autres nœuds dans le réseau d'opérateurs sous test ne nécessite pas la connaissance de leur contenu.

Notons cependant que l'utilisation d'une technique fondée sur l'analyse du réseau d'opérateurs pour le test d'intégration des nœuds est envisageable. Il faudrait pour cela définir des critères appropriés qui permettent la sélection de données de test dont le but est de vérifier la bonne utilisation par le nœud sous test des nœuds qu'il appelle.

5.4.2.3 Définitions des critères

Dans les définitions suivantes nous considérons que R_N est un réseau d'opérateurs et P un ensemble de chemins de R_N.

Définition 5-7 : P satisfait la *couverture des opérateurs* de R_N si

$$\forall o \in R_N \, \exists p \in P \; o.sorties \in p$$

En d'autres termes, il faut qu'au moins une sortie de chaque opérateur de R_N appartienne à l'ensemble des arcs des chemins de *P.* Cela garantit que tout opérateur du réseau sera activé et son résultat sera utilisé, éventuellement partiellement, au moins une fois.

Définition 5-8 : P satisfait la *couverture des arcs* de R_N si

$$\forall o \in R_N \, \forall x \in o.entrées \cup o.sorties \, \exists p \in P \; x \in p$$

Informellement, tout arc du réseau doit appartenir à au moins un des chemins de *P*. Cela garantit que tous les arcs seront utilisés au moins une fois pour le calcul d'une sortie.

Il est clair que la couverture des arcs inclut la couverture des opérateurs (en considérant la relation d'inclusion définie au paragraphe 2.3.2.2). Par ailleurs, ces deux critères sont satisfaits si *P* satisfait la *couverture des chemins* c'est à dire si *P* est l'ensemble de tous les chemins de R_N. La couverture des chemins est un critère généralement impossible à satisfaire puisque le nombre de chemins est potentiellement infini. Toutefois, si la longueur des chemins considérée est bornée (i.e. le nombre d'instants successifs intervenant dans l'expression des prédicats de chemin est borné) le nombre de chemins sera fini.

5.4.3 Génération des données d'entrée

Rappelons que la génération des données de test peut être déterministe ou aléatoire. Dans le premier cas on essaie de calculer à l'avance les données de test satisfaisant le critère au moyen de calculs symboliques. Cela nécessite en particulier la sélection de l'ensemble de chemins qui seront traversés. Dans le deuxième on se contente d'engendrer des données de test aléatoires et d'évaluer, *a posteriori*, la satisfaction du critère. Nous présentons l'application de ces deux techniques de génération dans notre cas aux paragraphes 5.4.3.2 et 5.4.3.3.

5.4.3.1 Sélection des chemins

Dans le cadre des techniques structurelles de test définies sur un graphe de contrôle, la difficulté du problème de la sélection d'un ensemble de chemins satisfaisant un critère vient principalement de l'existence éventuelle de chemins dont l'exécution est impossible (cf. paragraphe 2.3.5.3). Des chemins impossibles peuvent également subsister dans un réseau d'opérateurs, comme le montre l'exemple suivant.

Exemple 5-4

Considérons le nœud de la figure 5-2 et plus particulièrement le chemin *c = (**true**, out, a_3, a_2, out)* de son réseau d'opérateurs. Ce chemin est manifestement impossible. Son prédicat de chemin est *PC(c) = cond **and pre** init*. Cela signifie que la variable *cond* doit être vraie au deuxième instant de l'exécution du nœud. Or, suivant sa définition, la variable *cond* prend toujours une valeur fausse à cet instant, ce qui rend le chemin *c* impossible.

Ce problème qui ne possède pas de solution simple [BM94] n'a pas été abordé dans ce travail. Nous supposerons dans la suite du paragraphe qu'un ensemble fini de chemins $(p_i)_{i=1...n}$ satisfaisant le critère retenu a été calculé.

```
node exemple(in: bool) returns (out: bool);
var cond : bool;
let
    cond = true -> not pre cond ;
    out = true ->  if cond then pre(out)
                   else in;
tel
```

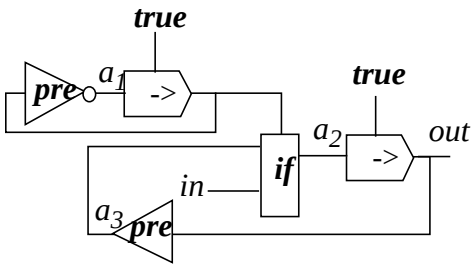

Figure 5-2 : Exemple de nœud contenant un chemin impossible

5.4.3.2 Génération aléatoire

Une manière de procéder à une génération aléatoire de données de test est d'utiliser un simulateur d'environnement (à condition que l'on dispose, bien entendu, de la spécification sous forme d'invariants de l'environnement du programme à tester).

Un dernier point qui doit être traité pour la mise en œuvre d'une telle méthode de test est la mesure du taux de satisfaction du critère retenu. Nous proposons ici un moyen simple d'effectuer cette mesure consistant en l'utilisation d'un nœud LUSTRE encapsulant les prédicats de l'ensemble de chemins satisfaisant le critère. Ce nœud possède un nombre de sorties booléennes égal à celui des chemins à exécuter. Chacune de ses sorties est initialement fausse et devient vraie dès que le chemin associé a été exécuté (i.e. dès que le prédicat de ce chemin a été satisfait).

Exemple 5-5

Le nœud servant à mesurer le nombre de chemins exécutés du nœud *Front* est donné par la figure 5-3.

5.4.3.3 Génération déterministe

Lors de la génération déterministe on cherche à calculer à l'avance les données de test exécutant les chemins satisfaisant le critère retenu. La méthode présentée ici utilise l'outil de vérification LESAR et a été proposée pour la première fois dans [Hil89]. Elle consiste à essayer de prouver, pour chaque chemin p_i, que ***not*** *(PP(p_i)*) est une propriété invariante du programme. Si cette preuve réussit, alors le chemin p_i est impossi-

```
node CheminsFront(X: bool) returns (p1, p2, p3 : bool;)
var init : bool ;
let
    init = true -> false ;
    p1 = init or pre(p1) ;
    p2 = X and not init or pre(p2) ;
    p3 = not pre(X) and not init or pre(p3) ;
tel;
```

Figure 5-3 : Nœud mesurant la couverture de chemins du nœud Front

Dans le cas opposé, LESAR fournit un contre-exemple sous forme d'une séquence d'entrées amenant le programme dans un état violant ***not*** *(PP(p_i))* (et, de ce fait, satisfaisant *PP(p_i)*). La longueur de cette séquence est minimale (*Lat(p_i)*).

Exemple 5-6

La couverture des chemins du nœud *Front* nécessite l'exécution des chemins p_1, p_2 et p_3. Les prédicats de chemin associés sont ceux calculés au paragraphe 5.4.2. Le prédicat de chemin de p_1 est satisfait par toute séquence d'entrées. En conséquence, nous tentons de prouver avec LESAR la propriété ***not*** *(PC(p_2))* = ***not*** *(X* ***and not*** *init)*. La séquence d'entrée résultante est ***(false***, ***true***) et couvre le chemin p_2. De même si nous tentons de prouver ***not*** *(PC(p_3))* = ***not*** *(****not pre****(X)* ***and not*** *init)* nous obtenons la séquence d'entrées ***(false***, ***false)***, qui couvre p_3 (notons que ***(false***, ***true)*** couvre p_2 et p_3).

Exemple 5-7

Considérons le nœud *on_off* de la figure 5-4. La valeur de la sortie booléenne de ce nœud change à chaque fois que son entrée devient vraie. Le nombre de chemins du réseau associé à ce nœud est infini :
p_{11} *= (in, out)* et p_{12} *= (in, a_3, out)* qui peuvent être couverts par une séquence de longueur *1*, p_{21} *= (in, out, a_1, a_3, out)*, p_{22} *= (in, out, a_1, a_2, a_3, out)*, p_{23} *= (in, a_3, out, a_1, a_3, out)*, p_{24} *= (in, a_3, out, a_1, a_2, a_3, out)* qui nécessitent une séquence de longueur minimale égale à *2* et ainsi de suite. Par exemple, la couverture des arcs est satisfaite si le chemin p_{22} *= (in, out, a_1, a_2, a_3, out)* est exécuté. Son prédicat de chemin est calculé de la manière suivante :
D'après • :
PC(p_{22}) = PC(in, out, a_1, a_2, a_3, out) =
pre *PA(in, out)* ***and pre*** *PA(out, a_1)* ***and*** *PA(a_1, a_2)* ***and*** *PA(a_2, a_3)* ***and*** *PA(a_3, out).*
D'après la règle 6, ***pre*** *PA(in, out) =* ***pre*** *init;*
d'après la règle 2, ***pre*** *PA(out, a_1) =* ***pre true*** *=* ***true***;

```
node on_off(in: bool) returns (out: bool);
  let
    out = in -> if in then not pre(out)
                else pre(out);
  tel
```

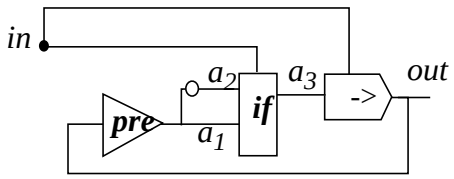

Figure 5-4 : Le nœud *on_off* et son réseau d'opérateurs

d'après la règle 1, $PA(a_1, a_2) =$ ***true***;
d'après la règle 3, $PA(a_2, a_3) =$ *in*;
d'après la règle 6, $PA(a_3, out) =$ ***not*** *init*.
Donc, $PC(p_{22}, t) =$ ***pre****(init)* ***and*** *in* ***and not*** *init.*
LESAR produit pour ce prédicat la séquence ***(false***, ***true)*** qui couvre le chemin p_{22}.

5.5 IMPLANTATION

Au même titre que pour la définition des critères de couverture nous avons décomposé l'implantation en deux étapes[1]. La première consiste à représenter le réseau d'opérateurs sous la forme d'un graphe facilement exploitable tandis que la deuxième concerne la construction des prédicats de chemins qui doivent être couverts.

5.5.1 Représentation du réseau d'opérateurs

Le programme LUSTRE à tester est transformé en un graphe orienté dont la structure découle directement des définitions données au paragraphe 5.3. Plus précisément, les nœuds de ce graphe sont de deux types :

- Les nœuds associés aux opérateurs du programme (opérateurs de base LUSTRE ou nœuds). Dans ce cas, ils contiennent trois informations :
  - Le nom de l'opérateur,
  - la liste des nœuds associés aux arcs d'entrée de l'opérateur,

[1] Le travail d'implantation présenté ici a été mené dans le cadre du stage de Cyril Perreau, étudiant en Maîtrise d'Informatique, en été 1996.

- la liste des nœuds associés aux arcs de sortie de l'opérateur.

- Les nœuds associés aux arcs du réseau d'opérateurs (variable ou résultat de calcul intermédiaire). Ces nœuds contiennent quatre informations :
  - Le type de l'arc (variable ou résultat intermédiaire),
  - le nom de l'arc (s'il s'agit d'une variable),
  - l'opérateur éventuel dont l'arc est une sortie,
  - l'opérateur éventuel dont l'arc est une entrée.

Deux nœuds n_1 et n_2 de ce graphe sont reliés par un arc dans les cas suivants :

- n_1 est un nœud associé à un opérateur et n_2 fait partie de la liste des nœuds associés aux arcs de sortie de l'opérateur.
- n_2 est un nœud associé à un arc et n_1 fait partie de la liste des nœuds associés aux arcs d'entrée de l'opérateur.

5.5.2 Calcul des chemins et de leurs prédicats

Nous nous sommes contentés d'implanter la couverture des chemins de longueur finie n fixée. Le calcul des chemins satisfaisant ce critère se fait simplement par un parcours complet du graphe représentant le réseau d'opérateurs. Un compteur permet de limiter le nombre de parcours successifs de circuits du graphe à n qui est la longueur maximale des chemins.

Le résultat de ce parcours est la liste des chemins du graphe, un chemin étant une suite de nœuds successifs du graphe. Il est ensuite facile de calculer le prédicat de chaque chemin par application directe de la définition 5-5.

6

Réalisations et expérimentations

6.1 INTRODUCTION

Parallèlement à l'étude théorique qui a abouti à la définition formelle des techniques de test présentées aux chapitres précédents, nous avons réalisé un ensemble de développements et expériences pratiques que nous résumons dans ce chapitre.

Dans un premier temps (paragraphe 6.2) nous faisons un bilan chiffré des développement conduits en vue de la réalisation des techniques de génération aléatoire sous contraintes et de test structurel. Nous présentons également un autre travail de développement que nous avons effectué dans le but de valoriser l'ensemble de ses techniques en les intégrant au sein d'un outil interactif.

Nous décrivons ensuite (paragraphe 6.3) une expérience que nous avons menée et qui s'inscrit dans le cadre de la validation pratique de nos techniques. Cette expérience a comporté deux activités :

- A partir d'une spécification informelle simplifiée d'un logiciel réactif de contrôle d'ascenseur, nous avons développé en LUSTRE sa spécification fonctionnelle, la spécification de son environnement ainsi que la spécification de ses propriétés de sûreté. La présentation de ce travail de spécification fait l'objet du paragraphe 6.3.
- Pour le développement des spécifications nous avons utilisé une première version de l'outil de simulation d'environnement préalablement créé. Cela nous a permis de constater l'utilité pratique de la technique et d'arriver à des conclusions intéressantes sur les difficultés dans le développement des spécifications. Le test structurel et le test des propriétés de sûreté ont été également appliqués sur une petite partie du logiciel. Enfin, les spécifications de l'environnement et des propriétés de sûreté ont été validées par l'exécution conjointe de deux simu-

lateurs (d'environnement et de logiciel sûr). Le paragraphe 6.4 résume les principales étapes de cette expérience ainsi que les réflexions qu'elle a suscitées.

6.2 RÉALISATION DES TECHNIQUES DE TEST

La réalisation des techniques de génération aléatoire sous contraintes a été largement facilitée par l'utilisation de deux outils du laboratoire Vérimag :

- Un traducteur permettant de passer d'un programme LUSTRE à l'automate booléen associé exprimé au format BAC.
- Une bibliothèque de fonctions de manipulation de BDD.

Parmi les techniques de génération aléatoire sous contraintes nous avons réalisé la simulation d'environnement et le test des propriétés de sûreté pour le cas des variables booléennes. Cette réalisation a nécessité l'écriture d'environ 4000 lignes de code dont 1400 de yacc et 2600 de C++.

Enfin, la réalisation de la technique de test structurel (couverture de chemins) a nécessité l'écriture de 1500 lignes de C++.

Il faut noter que, parallèlement au développement de ces techniques, nous avons étudié la mise en œuvre d'un outil interactif[1] facilitant leur mise en oeuvre et intégrant un certain nombre d'options de visualisation, utiles pour la recherche des défauts. Cet outil, baptisé LUTESS[2], offre les fonctionnalités suivantes :

- Saisie pour chaque type de test (simulation d'environnement, test des propriétés de sûreté et test structurel) des paramètres nécessaires à la génération des données de test et au déroulement de l'opération de test (fichiers contenant les spécifications, oracle, longueur et nombre souhaités des données de test, type et taux de couverture du réseau d'opérateurs, etc.).
- Lancement automatique de l'opération de test.
- Consultation des résultats du test (valeur des entrées et sorties à chaque instant de l'exécution) sous forme de tableaux avec différentes possibilités de filtrage des variables visualisées.
- Pour le cas des programmes LUSTRE, possibilité d'exécution du programme pas à pas avec mise en surbrillance des parties activées du code pour faciliter la localisation des défauts.

---

1. Ce projet, actuellement en cours, a successivement fait l'objet de deux mémoires d'ingénieur CNAM (Anne Garcia, de Décembre 1994 à Novembre 1995 et Florence Georges, de Janvier 1996 à Octobre 1996) ainsi que d'un stage de fin de MST Experts en Systèmes Informatiques (Fabrice Vallet, été 1996).
2. "LUstre-based Testing Environment for Synchronous Software"

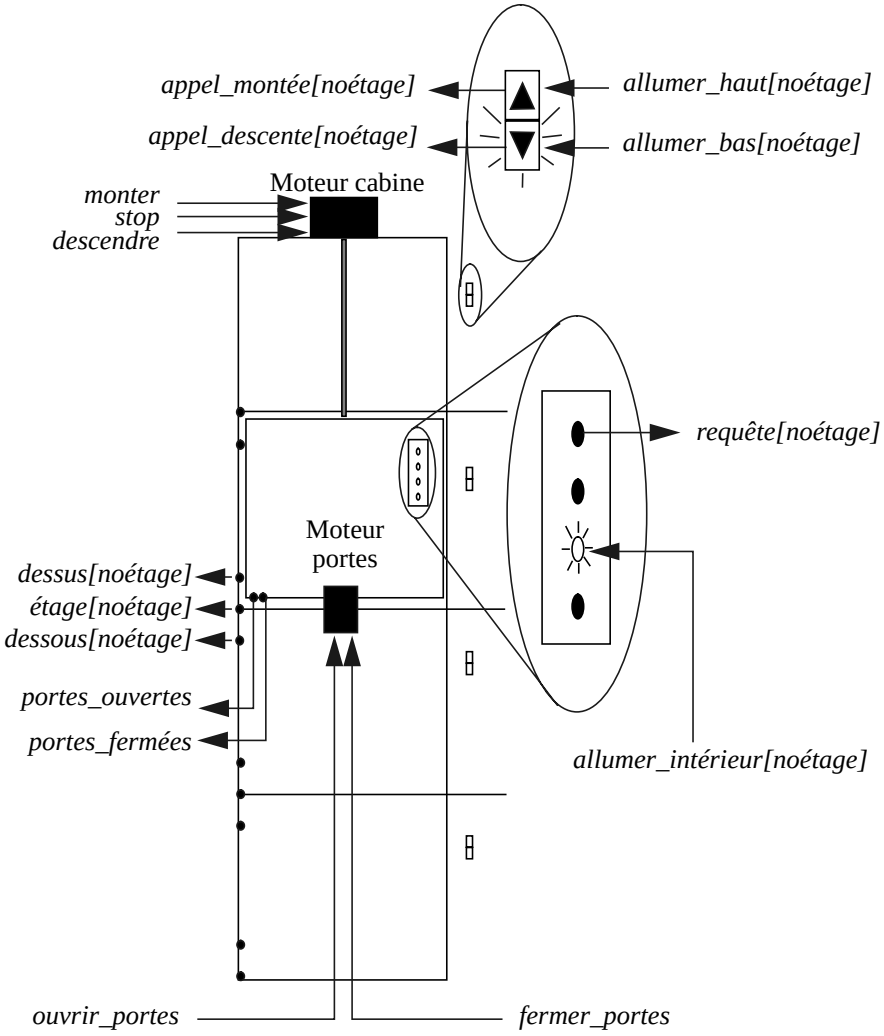

Figure 6-1 : L'ascenseur

L'implantation de la version actuelle de cet outil, qui fait appel à la bibliothèque graphique SUIT[1], a nécessité l'écriture de 15000 lignes de code C++.

6.3 ETUDE DE CAS : UN LOGICIEL DE CONTRÔLE D'ASCENSEUR

6.3.1 Description informelle

Nous considérons un logiciel réactif synchrone dont la fonction est de contrôler un ascenseur installé dans un immeuble de trois étages. L'ascenseur, présenté dans la figure 6-1, comporte une cabine qui se déplace dans sa cage de la manière habituelle.

1. "Simple User Interface Toolkit", distribué gratuitement par l'Université de Virginie (Etats-Unis).

L'ouverture et la fermeture des portes de la cabine est réalisée automatiquement par un moteur, commandé par le logiciel au moyen des signaux *ouvrir_portes* et *fermer_portes* qui ne doivent, évidemment, être jamais simultanément actifs. Les portes de la cabine sont munies de capteurs permettant au logiciel de connaître à tout moment leur état au moyen de deux signaux booléens, *portes_ouvertes* et *portes_fermées*. Dans le cas où aucun de ces deux signaux n'est actif, l'état des portes est indéterminé (cela peut correspondre à une position intermédiaire, si les portes sont en train de s'ouvrir ou de se fermer).

Trois capteurs - *dessus*, *étage* et *dessous* - sont installés au niveau de chaque étage et informent le logiciel de la position courante de la cabine via des signaux booléens. Plus précisément, le signal *dessus[noétage]* (resp. *dessous[noétage]*) est actif quand la cabine est sur le point d'atteindre l'étage *noétage* pendant sa descente (resp. sa montée). Ces deux signaux permettent au logiciel d'envoyer, le cas échéant, une commande d'arrêt au moteur de la cabine, afin que cette dernière puisse s'arrêter à temps à l'étage *noétage*. Le signal *étage[noétage]* n'est actif que quand la cabine est arrêtée à l'étage *noétage*. Bien entendu, il n'y a pas de capteur *dessus* au dernier étage ni de capteur *dessous* au rez-de-chaussée.

Un passager souhaitant appeler l'ascenseur peut le faire au moyen de deux boutons d'appel disposés verticalement. Le bouton supérieur sert à appeler l'ascenseur dans le but de monter, le bouton inférieur servant au cas opposé. Les signaux *appel_montée[noétage]* et *appel_descente[noétage]* sont respectivement envoyés au logiciel chaque fois que les boutons d'appel de l'étage *noétage* sont actionnés. Le bouton de montée du dernier étage et le bouton de descente du rez-de-chaussée sont déconnectés.

A l'intérieur de la cabine on trouve quatre boutons, un pour chaque étage. Quand un passager appuie sur le bouton correspondant à l'étage *noétage*, le signal *requête[noétage]* est envoyé au logiciel.

Les boutons de l'intérieur de la cabine ainsi que les boutons d'appel de chaque étage sont munis de voyants lumineux qui sont allumés si et seulement si les signaux *allumer_intérieur[noétage]*, *allumer_haut[noétage]* et *allumer_bas[noétage]*, émis par le logiciel, sont actifs.

L'envoi du signal *stop* par le logiciel vers le moteur de l'ascenseur provoque l'arrêt de la cabine tandis qu'après l'envoi des signaux *monter* et *descendre*, la cabine entame, respectivement, un mouvement d'ascension et de descente.

Le rôle du logiciel est d'émettre les commandes appropriées afin de satisfaire les requêtes des passagers ainsi que d'assurer l'allumage cohérent des voyants lumineux. Les différentes requêtes doivent être satisfaites de sorte à minimiser les déplacements de la cabine. Pour cela, nous considérons qu'une fois en ascension, elle continuera à monter, tout en répondant aux requêtes de montée des passagers, jusqu'à l'étage demandé *le plus haut* (et de manière symétrique pour la descente).

Cet exemple est considérablement simplifié par rapport aux exigences réelles que nous aurions pu avoir d'un tel logiciel. En particulier, nous ne traitons pas les incidents éventuels empêchant l'ouverture ou la fermeture des portes de la cabine, ni le problème de l'ouverture et fermeture des portes de chaque étage que nous considérons pris en charge par des éléments matériels de l'ascenseur. Ce choix est également fait dans un souci de clarté, puisque la prise en compte de ces points serait sans intérêt dans le contexte de ce document.

6.3.2 Spécification en LUSTRE

Nous supposons que la représentation en LUSTRE des signaux *dessus*, *étage*, *dessous*, *appel_montée*, *appel_descente*, *allumer_haut*, *allumer_bas*, *allumer_intérieur* et *requête*, se fait au moyen de tableaux booléens de longueur quatre (d'indice variant de *0*, pour le rez-de-chaussée, à *3*, pour le troisième étage). Les signaux *portes_ouvertes*, *portes_fermées*, *ouvrir_portes*, *fermer_portes*, *monter*, *descendre*, *stop* sont représentés par de simples variables booléennes.

6.3.2.1 Hypothèse de synchronisme

Nous supposons que l'environnement du logiciel (i.e. les indications des capteurs) évolue à une vitesse permettant la prise en compte par l'implantation de tous les événements externes. En particulier, nous admettons que :

- tous les signaux en provenance des capteurs ou du logiciel sont transmis de manière instantanée,
- la durée pendant laquelle les signaux *dessous*, *étage* et *dessus* sont actifs lors du passage de la cabine est au moins égale au temps de réaction du logiciel,
- pour qu'un appel ou une requête d'un passager puissent être pris en compte avec certitude par le logiciel, il faut que le bouton correspondant soit maintenu appuyé pendant un temps au moins égal au temps de réaction du logiciel.

6.3.2.2 Spécification de l'environnement

Nous donnons dans la suite une série de propriétés invariantes extraites de la spécification de l'environnement que nous avons développée (cf. annexe B) :

- A l'état initial du système, l'ascenseur est immobilisé à un étage quelconque :

  $$OR(4, étage) \rightarrow \textbf{\textit{true}}$$

  où *OR* est un opérateur booléen défini sur les tableaux booléens et qui correspond à la disjonction des éléments du tableau. De manière analogue nous pouvons définir un opérateur *AND* utilisé plus loin. Ces deux opérateurs, qui ne font pas partie du langage de base, sont définis sous forme de nœuds récursifs (cf.

chapitre 1).

- Les portes ne peuvent pas être simultanément ouvertes et fermées :

not *(portes_ouvertes* ***and*** *portes_fermées)*

- Un et un seul des signaux envoyés par les capteurs de position de la cabine dans la cage est actif au même instant :

AuPlusUn(4,dessus) ***and*** *AuPlusUn(4,étage)* ***and*** *AuPlusUn(4,dessous)* ***and***
#(OR(4,dessus),OR(4,étage),OR(4,dessous)) ***and***
(OR(4,dessus) ***or*** *OR(4,étage)* ***or*** *OR(4,dessous))*

où *AuPlusUn* est un opérateur booléen défini (sous forme de nœud récursif) sur les tableaux booléens, qui est vrai quand au plus un des éléments du tableau a une valeur vraie.

Ces trois propriétés expriment des relations instantanées sur les entrées du logiciel contrairement aux deux propriétés qui suivent et qui expriment des relations entre les valeurs courantes et passées des entrées et des sorties :

- L'état des portes ne change pas spontanément. En particulier, seule une commande appropriée du logiciel peut provoquer un tel changement d'état :

always_from_to(portes_fermées, portes_fermées, ***pre*** *ouvrir_portes)*

En d'autres termes, du moment où les portes se ferment, elles resteront fermées au moins jusqu'à l'instant suivant l'arrivée du signal *ouvrir_portes* en provenance du logiciel. De manière similaire :

always_from_to(portes_ouvertes, portes_ouvertes, ***pre*** *fermer_portes)*

- La cabine ne quitte un étage qu'après réception du signal *monter* ou *descendre* :

AND(4, always_from_to(étage[0..3], étage[0..3], ***pre*** *(monter* ***or*** *descendre)^(4))*

La liste complète des contraintes d'environnement identifiées est donnée en annexe B.

6.3.2.3 Spécification des propriétés de sûreté

Nous présentons d'abord les propriétés qui garantissent la sécurité des passagers. Ensuite, nous ajoutons certaines propriétés invariantes que nous estimons utiles dans le cadre de la validation de ce logiciel.

- Les portes de la cabine doivent être toujours fermées lors de ses déplacements. L'expression en LUSTRE de cette propriété peut se faire de deux manières différentes. La première est de décrire les comportements *sûrs de la cabine et de ses portes* :

  *Implies(AND(4, **not** étage), portes_fermées)* **(6-a)**

  La deuxième manière consiste à décrire les comportements *sûrs du logiciel* pour que cette propriété soit respectée. Nous devons, dans ce cas, réexprimer la propriété de la manière suivante :

  *Implies(monter **or** descendre, portes_fermées) **and***
  Implies(ouvrir_portes, OR(4, étage)) **(6-b)**

  Cela revient à exiger du logiciel de ne jamais tenter de déplacer la cabine quand ses portes sont ouvertes et, inversement, de ne jamais tenter d'ouvrir les portes si la cabine n'est pas arrêtée à un étage.

- Chaque arrêt de la cabine à un étage doit être suivi d'une ouverture de ses portes avant le nouveau départ de la cabine vers un autre étage. Cette propriété garantit, entre autres, qu'un passager ne restera jamais bloqué dans la cabine. Nous pouvons de nouveau exprimer cette propriété de deux points de vue :

  *once_from_to(portes_ouvertes, front(OR(4, étage)), **not** OR(4, étage))* **(6-c)**

  Cette propriété décrit les comportements sûrs de la cabine tandis que la propriété suivante exprime les comportements sûrs du logiciel :

  *once_from_to(ouvrir_portes, front(OR(4, étage)), **not** OR(4, étage)) **and***
  *once_from_to(portes_ouvertes, front(OR(4, étage)), monter **or** descendre)* **(6-d)**

  En d'autres mots, nous exigeons du logiciel d'envoyer au moins une fois un signal d'ouverture de portes entre l'arrivée de la cabine à un étage et son nouveau départ. De plus, le logiciel ne doit pas ordonner le départ d'un étage de la cabine si ses portes n'ont pas été ouvertes au moins une fois depuis son arrivée à cet étage.

Les propriétés suivantes n'ont pas de conséquences directes sur la sécurité des passagers mais excluent des comportements manifestement anormaux du logiciel qu'on aimerait éviter.

- Un seul des signaux *monter*, *stop* et *descendre* peut être vrai à un instant donné :

  #(monter, stop, descendre)

```
node OracleAscenseur(étage : bool^4;
   portes_ouvertes, portes_fermées, monter, stop, descendre,
   ouvrir_portes, fermer_portes : bool)
returns (ok : bool);
let
   ok =
      Implies(monter or descendre, portes_fermées) and
      Implies(ouvrir_portes, OR(4, étage)) and
      once_from_to(ouvrir_portes, front(OR(4, étage)),
                        front(not OR(4, étage))) and
      once_from_to(portes_ouvertes, front(OR(4, étage)),
                        monter or descendre) and
      #(monter, stop, descendre) and
      #(ouvrir_portes, fermer_portes) ;
tel
```

Figure 6-2 : Oracle pour le logiciel de contrôle d'ascenseur

- Un seul des signaux *ouvrir_portes* et *fermer_portes* peut être vrai à un instant donné :

$$\#(ouvrir\_portes, fermer\_portes)$$

Ces propriétés ont été réunies dans un nœud LUSTRE qui a été utilisé comme oracle et qui est présenté dans la figure 6-2.

6.4 RAPPORT D'EXPÉRIENCE

6.4.1 Validation des outils de test

Le premier objectif du développement du logiciel de contrôle d'ascenseur a été la validation de l'ensemble de programmes implantant nos techniques de test.

Plus précisément, l'implantation de la simulation de l'environnement a été validée au moyen d'un "oracle" vérifiant que les données aléatoires produites sont conformes aux contraintes d'environnement. Cet oracle est écrit en LUSTRE et consiste de manière similaire à l'oracle de la figure 6-2 en la conjonction de ces contraintes. Il est compilé à l'aide du compilateur LUSTRE que l'on suppose correct. Le même type de validation automatique a pu être mis en œuvre pour la simulation de logiciel sûr à partir des propriétés de sûreté du logiciel de contrôle d'ascenseur.

En revanche, ce genre de validation n'est pas envisageable pour le test des propriétés de sûreté ou pour le test structurel, l'écriture d'un oracle n'étant pas possible. Nous nous sommes ainsi contentés d'un contrôle visuel attentif des résultats obtenus. Ce contrôle a été largement facilité par le développement d'une interface graphique de

l'ascenseur représentant la position de la cabine, l'état des boutons d'appel (appuyés, allumés) et des portes ainsi que la valeur des commandes envoyées par le logiciel.

6.4.2 Application des techniques de génération aléatoire sous contraintes

6.4.2.1 La simulation d'environnement comme un outil d'aide au développement des spécifications

Les principales difficultés que nous avons rencontrées pendant l'application de la simulation de l'environnement ont été liées au développement d'une "bonne" spécification sous forme d'invariants du comportement de la cabine, des portes et du moteur (éléments constituant l'environnement). L'obtention de la spécification nécessite une bonne maîtrise de l'utilisation de la logique temporelle.

De plus, nous avons observé que la taille du BDD associé aux contraintes d'environnement varie considérablement avec la manière dont sont exprimées ces contraintes.

Nous avons commencé le développement du logiciel de contrôle par la rédaction de la spécification de son environnement. Nous avons ensuite procédé à l'écriture des nœuds LUSTRE réalisant le logiciel et des propriétés de sûreté.

La simulation de l'environnement a permis d'identifier plusieurs défaillances dans le fonctionnement de l'ensemble logiciel-environnement. Ainsi, les deux spécifications ont évolué en parallèle, les défaillances constatées étant dues à des défauts résidant à la fois dans les contraintes d'environnement et l'implantation du logiciel. De cette manière, nous avons pu nous convaincre de la pertinence de cette démarche de validation en phase initiale du développement qui a permis la détection d'un grand nombre de défauts d'implantation (en particulier dans le nœud *Moteur*) mais aussi d'obtenir rapidement une version acceptable de la spécification de l'environnement.

Toutefois, une simulation simple d'environnement ne réalise qu'un test aléatoire du logiciel suivant une loi uniforme (équiprobabilité) et ne suffit pas à elle seule à la détection de tous les défauts, comme nous l'avons signalé au chapitre 2. Ceci a été confirmé lors de notre expérience durant laquelle un défaut important dans la spécification de l'environnement n'a pas pu être identifié par ce type de test. Une des propriétés de cette dernière considérait en effet que, contrairement aux indications de la spécification informelle, le signal *étage[i]* était émis à chaque passage de la cabine par l'étage *i* (et non uniquement quand la cabine est arrêtée à cet étage). Ainsi, la propriété

*AND(3, always_from_to(dessus[0..2], dessus[0..2], etage[0..2])) **and***
AND(3, always_from_to(dessous[1..3], dessous[1..3], etage[1..3]))

exprime la continuité du mouvement de la cabine en s'appuyant sur cette considération erronée.

Par ailleurs, une deuxième propriété (conforme à la spécification informelle) stipulait que le signal *étage[i]* ne peut être émis que suite à la réception par le logiciel du signal *stop* :

*AND(4, always_from_to(**not** étage, **not** étage, **pre** stop^(4)))*

La conjonction de ces deux propriétés provoque une défaillance dans le cas où la cabine se trouve sur le point d'atteindre un étage *i* (i.e. le signal *dessus[i]* ou *dessous[i]* a été émis) et le logiciel n'émet pas de signal *stop*, ce qui peut être tout à fait normal si aucune requête n'est enregistrée pour l'étage *i*. La cabine se trouve alors bloquée dans sa position courante.

La mise en évidence de cette défaillance suppose qu'au moment du passage de la cabine par un étage, aucun appel et aucune requête n'ont été émis depuis ou vers cet étage. Cette situation s'avère hautement improbable quand le tirage des entrées du logiciel se fait de manière équiprobable (on notera qu'aucune contrainte de la spécification d'environnement ne porte sur l'émission des appels ou des requêtes).

Il est par ailleurs facile de constater que le test des propriétés de sûreté n'aurait pas permis une détection de ce défaut puisqu'aucune de ces propriétés ne s'intéresse à la situation que nous venons de décrire.

Ce défaut pourrait être détecté dans les deux cas suivants :

- Si la génération aléatoire suit un profil opérationnel attribuant une faible probabilité d'occurrence aux signaux de requête et d'appel, la situation décrite plus haut révélant le défaut peut se manifester.
- Nous avons souligné au chapitre 2 que de nombreux défauts concernent souvent des traitements singuliers correspondant à des éléments particuliers du domaine d'entrée du logiciel qui doivent être identifiés et inclus dans les jeux de test. L'absence de signal d'appel et de requête pour un étage nécessite manifestement un tel traitement. Il suffit d'introduire dans la spécification de l'environnement une propriété caractérisant cette singularité pour tester le comportement du système dans le cas résultant. C'est en procédant ainsi que nous avons pu détecter ce défaut de spécification et plus précisément en introduisant successivement la propriété

  ***not** (requête[i] **or** appel_montée[i] **or** appel_descente[i])*

  dans la spécification de l'environnement, pour des valeurs de *i* extrêmes (*0* et *3*) ainsi que pour une valeur intermédiaire (*2*).

Plus généralement, cette observation met en évidence l'impossibilité pour la simulation d'environnement de détecter tout défaut du logiciel qui consiste à l'envoi du signal *stop* à l'approche d'un étage pour lequel aucun appel ou requête n'ont été émis.

6.4.2.2 Test et validation des spécifications et du logiciel

Conformément au cadre méthodologique suggéré au paragraphe 2.4.4, nous avons procédé à la validation de la spécification de l'environnement, des propriétés de sûreté et du logiciel de plusieurs manières :

- En faisant interagir le logiciel développé, le simulateur d'environnement associé et l'oracle comportant les propriétés de sûreté identifiées et en observant le comportement de l'ensemble, de la manière décrite au paragraphe 6.4.2.1.
- En exécutant conjointement le simulateur associé aux contraintes d'environnement et un simulateur d'environnement sûr construit à partir des propriétés de sûreté et en observant les comportements obtenus. En plus de l'observation, nous avons introduit une deuxième version de l'oracle dans laquelle les propriétés 6-a et 6-c ont respectivement été remplacées par les propriétés 6-b et 6-d. Nous avons ainsi voulu nous assurer de l'équivalence de ces propriétés (6-a avec 6-b et 6-c avec 6-d) en comparant les comportements engendrés à ceux acceptés par l'oracle.
  Lors de cette expérience nous avons détecté 2 violations de l'oracle ainsi construit, qui correspondent à des ouvertures des portes de la cabine tandis que cette dernière se trouvait entre deux étages. Cela nous a amené à remplacer les propriétés

  #(monter, stop, descendre), #(ouvrir_portes, fermer_portes)

  par la propriété

  #(monter, stop, descendre, ouvrir_portes, fermer_portes).

  Cette dernière signifie, en particulier, que le logiciel ne peut pas envoyer à la fois une commande de déplacement de la cabine et une commande de modification de l'état des portes.
- En procédant au test des propriétés de sûreté. Afin d'évaluer ce type de test, nous avons introduit certains défauts dans le logiciel. Pour chaque séquence exécutée, nous avons observé le nombre d'instants où une violation de l'oracle s'est manifestée. Nous nous sommes ainsi aperçus que :
  - Certaines situations de "blocage" que nous n'avons pas identifiées au paragraphe 3.5.8.2 se sont manifestées. Plus précisément, considérons la propriété :

    *Implies(monter **or** descendre, portes_fermées)*

    Le processus de test de propriétés de sûreté empêche la fermeture des portes afin de vérifier que le signal *monter* ou *descendre* n'est pas envoyé par le

logiciel. En conséquence, si la cabine se trouve à l'étage *i* les portes ouvertes et si le logiciel vérifie la propriété ci-dessus pour cet étage, elle ne le quitte plus jusqu'à la fin de l'exécution et n'atteint, donc, aucun autre étage (pour lequel le logiciel se comporte éventuellement de manière incorrecte).
L'introduction d'un non déterminisme dans le choix des entrées par le processus de test de propriétés de sûreté (préconisée au paragraphe 3.5.8.2) a permis d'éviter ces situations.

- Quand les propriétés sont considérées individuellement ou quand les défauts introduits concernent l'ensemble des propriétés testées, on obtient un taux de violations de l'oracle nettement plus élevé (20 à 40 %) avec le test des propriétés de sûreté qu'avec la simulation d'environnement simple.
  Au contraire, si plusieurs propriétés sont testées simultanément et si le défaut introduit ne concerne que peu d'entre elles, ce taux peut être moins important que dans le cadre de la simulation d'environnement. Cela s'explique par le fait que le processus de test des propriétés de sûreté engendrera plus de données qui testent les propriétés respectées par le logiciel et qui ne testent pas forcément les propriétés non respectées.

Il semblerait, donc, que le test des propriétés de sûreté considérées individuellement est plus performant que le test global de ces mêmes propriétés.

- En procédant, enfin, au test structurel d'une petite partie du logiciel. Plus précisément, nous avons créé un ensemble de mutants (cf. paragraphe 2.3.2.3) pour le nœud BoutonsCabine (cf. annexe B). Ces mutants (36 au total) ont été obtenus par les remplacements suivants :
  - Suppression de l'opérateur ***not***; remplacement des opérateurs ***and*** par ***or*** et inversement.
  - Remplacement des variables par des constantes.
  - Remplacement d'une variable par une autre.

  Nous avons ensuite testé le programme ainsi obtenu de deux manières :
  - En calculant les données de test satisfaisant la couverture des chemins de longueur 1.
  - En testant le logiciel avec un simulateur d'environnement et en évaluant *a posteriori*, la couverture des chemins de longueur 1 et 2 obtenue.

  De cette expérience très restreinte, nous retirons certaines observations qui demandent à être confirmées par d'autres expériences pour être validées :
  - La couverture de chemins de longueur 1 ne semble donner aucune garantie de détection des défauts. En effet, l'exécution du nœud BoutonsCabine avec des données construites de manière déterministe satisfaisant cette couverture n'a permis de tuer aucun des mutants.

- La couverture des chemins de longueur 2 semble être un critère plus efficace. Les séquences aléatoires couvrant environ 70% de ces chemins ont permis d'éliminer 31 des 36 mutants considérés.
- La couverture des chemins de longueur 3 ne semble pas apporter un pouvoir de détection de défauts supplémentaire considérable.

La première observation n'est pas surprenante, d'une part suite à la présence de l'opérateur ***pre*** qui nécessite l'exécution d'une séquence de longueur au moins égale à 2 pour être évalué et, d'autre part, suite au très petit nombre de séquences qui satisfont le critère.
La deuxième et troisième observation nous laissent espérer que la couverture de chemins de longueur 2 pourrait être fortement correlée au pouvoir de détection d'une séquence de données. Cela ferait de ce critère un moyen plus économique de sélectionner les jeux de test que la couverture des séquences de transition de longueur 2 de l'automate associé au programme, qui a été proposée par [Maz94](cf. paragraphe 7.3.1).

6.5 CONCLUSIONS

La première conclusion de cette expérience concerne la mise en œuvre et l'utilisation d'un simulateur d'environnement comme un outil d'aide au développement du logiciel. En développant le programme de contrôle d'ascenseur, nous avons pu nous persuader de la pertinence et de l'utilité d'un tel outil. L'introduction de profils opérationnels devrait renforcer son pouvoir de détection de défauts.

Le test des propriétés de sûreté semble renforcer ce pouvoir mais sous une condition : qu'une seule des propriétés soit testée à la fois. Cela demande, en particulier, de pouvoir définir la granularité des propriétés afin de pouvoir déterminer celles qui seront considérées individuellement.

La simulation de l'environnement et du logiciel sûr, exécutées en parallèle, nous ont permis, d'une part, de détecter des défauts dans la spécification des propriétés de sûreté et, d'autre part d'avoir confiance dans l'ensemble de propriétés retenues.

Enfin, la définition de nouveaux critères semble une nécessité dans le cadre de la technique de test structurel, les résultats obtenus n'étant pas concluants.

7

Comparaisons

7.1 INTRODUCTION

Ce chapitre est consacré à la comparaison de notre approche à des travaux de recherche récents sur la vérification et la validation des logiciels réactifs. Cette comparaison permet de situer notre travail par rapport à un ensemble de travaux connexes et, par ce biais, de mieux comprendre son apport.

Nous avons plusieurs fois dans les chapitres précédents insisté sur la complémentarité du test et de la vérification formelle de programmes LUSTRE. Il semble donc naturel de comparer, dans un premier temps, nos techniques de test à la technique de preuve formelle que nous avons présentée au chapitre 1. Le paragraphe 7.2 est consacré à cette comparaison.

L'objet du paragraphe suivant (7.3) est l'étude de trois approches du test des logiciels synchrones qui ont été proposées récemment et dont une brève présentation a été faite au chapitre 2. La première de ces approches [Maz94] préconise l'utilisation des techniques de test statistique dans le but d'assurer une bonne couverture de l'automate que le compilateur construit à partir d'un programme LUSTRE. La deuxième approche [Hsi94] est l'adaptation au cas de LUSTRE d'une technique d'aide à la génération automatique de jeux de test à partir d'une spécification algébrique. Enfin, la dernière approche [MHM+95] procède à la génération automatique de jeux de test à partir d'une spécification formelle du programme donnée sous la forme d'un automate d'états finis.

Le dernier paragraphe du chapitre (7.4) propose une brève comparaison de notre approche à celles adoptées dans un domaine voisin, celui du test des protocoles. Cette comparaison est *a priori* intéressante pour plusieurs raisons :

- Les protocoles entrent dans la catégorie des logiciels critiques, et le test reçoit donc un rôle très fort de maîtrise de la qualité.
- Les problèmes étudiés dans les travaux sur le test de protocoles sont proches de nos préoccupations sur les rapports entre la validation et la vérification, et entre le test et la preuve. Ainsi, le test de protocoles est probablement le seul domaine d'application du test à avoir suscité autant de travaux théoriques où l'on cherche à comprendre les mécanismes profonds mis en oeuvre dans le test, en s'intéressant à deux questions : "qu'est-ce que le test peut prouver ?" et "comment sélectionner un nombre raisonnable de données de test ?".
- Enfin, les formalismes les plus couramment utilisés dans ce domaine sont à base d'automates, donc proches de ceux que nous manipulons.

7.2 TEST ET VÉRIFICATION FORMELLE DE PROGRAMMES LUSTRE

La vérification formelle de programmes LUSTRE a été le point de départ de notre travail. Nous avons voulu que nos techniques de test soient à la fois complémentaires et compatibles avec elle. Dans ce paragraphe nous tentons de comparer les deux approches - test et preuve - de sorte à mettre en évidence leurs similitudes et leurs différences. Nous espérons ainsi mieux montrer leur complémentarité.

Nous adoptons pour cette comparaison trois points de vue. Dans un premier temps nous mettons en évidence les différences qui existent dans le type des spécifications nécessaires en vue de la preuve ou du test. Nous montrons également que les techniques de génération aléatoire sous contraintes peuvent être utiles dans le processus de développement des spécifications en vue de la preuve.

Nous procédons ensuite à une comparaison plus formelle, en profitant de la définition des techniques de génération aléatoire sous contraintes dans un formalisme identique à celui utilisé pour la définition du processus de preuve formelle. Cette comparaison fait apparaître un certain nombre de motivations communes qui sont à l'origine des deux approches, bien que leurs mises en œuvre soient différentes.

Nous terminons cette comparaison en traitant les aspects de coût. Nous montrons, en particulier, que dans le cas général, les besoins en place mémoire des techniques de test sont plus faibles que dans le cadre de la preuve formelle.

7.2.1 Comparaison des processus de spécification

La vérification formelle des programmes LUSTRE nécessite trois types de spécifications :

- la spécification fonctionnelle du programme (i.e. son implantation),
- la spécification de son environnement,
- la spécification de ses propriétés de sûreté.

Il est évident que si une de ces spécifications est absente ou exprimée dans un langage autre que LUSTRE la preuve ne peut pas avoir lieu.

Lors de la conception de nos techniques nous avons adopté une optique différente puisque nous n'avons pas voulu nous borner au cas des programmes rédigés entièrement en LUSTRE. Ainsi, un logiciel implanté dans un langage de programmation quelconque (et dont on ne dispose pas, par exemple, du code source) peut être testé avec nos techniques de génération aléatoire sous contraintes à condition de développer en LUSTRE la spécification de l'environnement et des propriétés de sûreté.

Cette caractéristique permet d'utiliser LUSTRE comme un langage d'aide à la validation des logiciels réactifs, même si un autre langage lui a été préféré pour leur implantation. Mais elle permet également d'envisager d'utiliser nos techniques afin d'accompagner le développement des spécifications d'un logiciel en vue de sa preuve et plus particulièrement d'*animer les spécifications* dans le but d'acquérir une certaine confiance dans leur validité et dans le lait qu'elles sont complètes. Cette animation de spécifications peut être opérée de deux manières :

- *Approximation de la preuve*. Cela consiste simplement à exécuter le logiciel sous test en utilisant un générateur aléatoire contraint par la spécification de l'environnement et un oracle encapsulant les propriétés de sûreté. Contrairement à la preuve, dans le cas du test, la satisfaction des propriétés de sûreté (vérifiée par l'oracle) ne sera examinée que pour un nombre limité de comportements reproduits par le générateur aléatoire sous contraintes (d'où le terme "approximation"). Il peut être utile de procéder à une telle simulation pour acquérir une certaine confiance dans les spécifications et le logiciel avant de lancer la (coûteuse) procédure de preuve.
- *Animation des spécifications*. Nous pouvons imaginer que l'équipe développant la spécification de l'environnement et les propriétés de sûreté du logiciel est indépendante de celle développant le logiciel lui-même (ce qui est naturel dans le cadre du développement de logiciels critiques). Dans ce cas, la première équipe dispose d'un moyen de valider les spécifications qu'elle développe sans être en possession du logiciel. Il suffit pour cela de procéder à l'animation conjointe des deux spécifications en faisant intervenir un simulateur d'environnement et un simulateur de logiciel sûr. Cela revient, en clair, à animer les propriétés de sûreté et la spécification de l'environnement en parallèle et à vérifier, par observation, que l'ensemble fonctionne correctement.

Nous constatons ainsi que les techniques de génération aléatoire sous contraintes peuvent constituer un outil puissant d'aide au développement des spécifications, qui peut être utilisé en parfaite complémentarité avec le processus de preuve formelle.

Nous n'avons pas mentionné dans ce qui précède les techniques de test structurel. En effet, ces dernières ne peuvent être utiles que si le logiciel à tester est implanté en LUSTRE, ce qui les place en situation de concurrence avec la preuve. Toutefois, il ne faut pas oublier que dans le cas où la preuve est impossible à cause d'un manque de

ressources, ces techniques (mais aussi les techniques de génération aléatoire sous contraintes) peuvent fournir une solution de secours appréciable.

7.2.2 Comparaison des moyens de mise en œuvre

7.2.2.1 Génération aléatoire sous contraintes

Le processus de preuve formelle transforme les trois spécifications (fonctionnelle, d'environnement et des propriétés de sûreté) en un modèle unique qui représente les comportements du système réactif composé du logiciel et de son environnement. Cela revient à spécifier deux *observateurs* [HLR93] associés aux contraintes d'environnement et aux propriétés de sûreté. Il s'agit de machines d'entrées-sorties émettant un signal d'alarme chaque fois que les propriétés qui leur sont associées ne sont pas respectées. La preuve de la validité des propriétés de sûreté consiste en la *composition parallèle* de ces deux observateurs avec la machine synchrone associée au logiciel. Les propriétés sont vérifiées si la machine résultante ne comporte aucun état provoquant l'émission d'un signal d'alarme (i.e. tous les comportements de la machine satisfont les propriétés).

Le rôle de la spécification d'environnement dans ce modèle est d'éliminer les comportements non valides de l'environnement, ce qui a comme résultat la suppression de certaines transitions. Cette suppression se traduit généralement par une réduction de la taille du modèle.

Les propriétés de sûreté sont également utilisées dans un souci de minimisation de taille, en particulier lors de l'application de la méthode de vérification "en arrière" (cf. paragraphe 1.5.4.2). En effet, les états sont regroupés en classes lors de l'exploration du modèle. Ces classes sont définies à partir de la manière dont est fait le calcul de la valeur logique de ces propriétés.

Ces deux spécifications sont utilisées dans le cadre de la génération aléatoire sous contraintes avec une motivation similaire. Notons d'abord que, contrairement au processus de preuve, les spécifications ne sont pas traduites en un seul modèle (en particulier, l'implantation LUSTRE du logiciel n'est pas utilisée). Le modèle considéré est une abstraction de l'environnement du logiciel.

Ce modèle représente tous les comportements valides de l'environnement. De manière analogue à la preuve où la réaction du logiciel n'est pas vérifié pour des comportement non valides de son environnement, un logiciel testé au moyen d'un simulateur d'environnement ne sera jamais amené à réagir à de tels comportements. Ainsi, tandis que dans le cadre de la preuve on utilise la spécification de l'environnement pour réduire la taille du modèle, dans le cadre de la génération aléatoire sous contraintes nous nous contentons de limiter le domaine d'entrées du logiciel (i.e. l'ensemble de comportements de son environnement).

Le test des propriétés de sûreté amène une réduction supplémentaire de ce domaine. De tous les comportements valides de l'environnement cette technique éli-

mine une partie de ceux qui n'ont pas d'impact sur la satisfaction des propriétés de sûreté.

7.2.2.2 Test structurel

Lors du test structurel on s'intéresse à l'exécution d'un petit nombre de cas qui sont jugés significatifs par rapport à la structure du programme et non par rapport à un modèle comportemental quelconque. Pour cette raison, une technique de test structurel ne peut pas être simplement comparée à la vérification formelle. De plus, chaque critère de type structurel définit une politique de sélection de données différente, ce qui rend cette comparaison encore plus difficile.

Toutefois, l'utilisation de LESAR pour la génération des jeux de test permet de faire un lien entre les deux approches. L'outil de vérification est vu, dans le cas du test, comme un moyen simple de rechercher un contre-exemple. Nous pouvons noter que la recherche d'un tel contre-exemple est en général beaucoup plus simple que la preuve d'une propriété. Le besoin de ressources résultant de l'utilisation de LESAR sera donc moindre dans ce cas.

7.2.3 Eléments de comparaison en termes de coût

Une comparaison des coûts du test et de la vérification formelle n'est pas envisageable dans l'absolu, les deux techniques opérant sur des modèles différents. Nous rappelons, cependant, dans ce paragraphe le coût de chaque technique en soulignant les étapes de calcul les plus coûteuses. Nous avons écarté de cette étude l'approche structurelle de test, estimant que la comparaison dans ce cas n'est pas significative.

7.2.3.1 Facteurs de coût de la vérification formelle

Quelle que soit la méthode de vérification utilisée (en avant ou en arrière), la vérification formelle de programmes LUSTRE comporte deux étapes :

- Construction des fonctions booléennes de transition, d'assertion et de sortie du modèle associé au nœud de vérification.
- Parcours des états du modèle représenté par ces trois fonctions.
  - Dans le cadre de la méthode en avant, ce parcours consiste en un calcul itératif de l'ensemble des états accessibles, puis l'évaluation de la fonction de sortie (correspondant aux propriétés de sûreté) dans chaque état de cet ensemble.
  - Dans le cadre de la vérification en arrière, l'ensemble des états accessibles n'est pas engendré, l'algorithme consistant en la manipulation symbolique de classes d'états.

Dans les deux cas, la difficulté vient de la deuxième étape (parcours d'états) soit parce que le nombre d'états est important (vérification en avant), soit parce que la manipulation des fonctions booléennes décrivant les classes d'états est trop coûteuse en temps (vérification en arrière).

7.2.3.2 Rapports entre les facteurs de coût des techniques de test et de vérification

La mise en œuvre de la génération aléatoire sous contraintes nécessite les trois étapes suivantes :

- Construction de la fonction de transition, de la fonction associée aux contraintes d'environnement et, éventuellement, de la fonction associée aux propriétés de sûreté du modèle associé au nœud testeur.
- Etiquetage du BDD associé aux contraintes d'environnement.
- Application d'un des algorithmes de génération.

La transformation en BDD des fonctions associées au contraintes d'environnement, aux propriétés de sûreté et aux transitions du modèle d'un générateur aléatoire contraint est similaire à celle des fonctions d'assertion, de sortie et de transition du modèle dans le cadre de la vérification formelle.

Cependant, dans le cas de la vérification formelle, les variables de sortie étant toutes définies dans le nœud de vérification, les fonctions de transition et de sortie sont uniquement définies sur les entrées et les états du modèle. En revanche, dans le cas de la génération aléatoire sous contraintes les variables de sortie ne sont pas définies dans le nœud testeur (i.e. nous ignorons la manière exacte dont réagit le logiciel sous test). La fonction de transition ainsi que la fonction décrivant les propriétés de sûreté sont donc également définies sur les variables de sortie. Une conséquence de cette différence est que les BDD représentant ces deux fonctions ont un nombre de variables plus grand mais, en contrepartie, le coût de leur construction est plus faible, puisque le calcul des sorties (qui implique la manipulation de fonctions booléennes supplémentaires) n'est pas pris en compte.

En ce qui concerne l'étiquetage du BDD associé aux contraintes d'environnement, l'expérience nous a montré que son coût reste nettement inférieur à celui de la construction de ce même BDD. Nous considérons, ainsi, que l'étiquetage des BDD nécessaire à la génération équiprobable de valeurs n'affecte pas le coût total de la génération aléatoire.

Une manière d'évaluer le coût des algorithmes de génération est d'estimer celui de chaque itération. Ce dernier est linéaire par rapport au nombre total des variables de la fonction d'environnement. Concrètement, cela signifie que le temps de réaction à une sortie du logiciel sous test d'un générateur aléatoire contraint est très court et mesurable. En conséquence, la simulation du fonctionnement du logiciel devrait être réaliste.

Une autre évaluation de ce coût peut être obtenue en considérant le modèle abstrait associé au nœud de vérification du logiciel sous test (dans le cas où il est, bien entendu, disponible) et d'estimer le nombre d'itérations nécessaires à la génération d'un nombre de séquences assurant une couverture de tous les états de ce modèle. Cette couverture assurerait la satisfaction des propriétés de sûreté par le logiciel au même titre que la preuve. Bien que nous n'ayons pas étudié les moyens de procéder à une telle évaluation, nous pouvons noter que le coût ainsi défini dépend du modèle du logiciel sous test. Quand ce modèle est explicitement engendré, une analyse peut nous permettre d'attribuer des probabilités d'exécution à chaque état et de calculer ainsi le nombre d'itérations de l'algorithme de génération nécessaire pour que chaque état soit accédé au moins une fois durant le test. Une méthode analogue consistant à calculer la probabilité d'exécution des transitions est utilisée dans [Maz94] (cf. paragraphe 7.3.1).

7.2.4 Discussion

En récapitulant les comparaisons ci-dessus nous pouvons retenir les point suivants qui confirment la complémentarité du test et de la preuve formelle :

- Les techniques de génération aléatoire sous contraintes donnent un moyen de valider les spécifications développées en vue de la preuve formelle. Cette validation peut avoir la forme d'une animation de spécifications ou d'une approximation de la preuve.
- La simulation de l'environnement effectue une restriction du domaine d'entrée du logiciel analogue à celle qu'opère la fonction d'assertion sur le modèle de la vérification formelle. De même, le test des propriétés de sûreté est un moyen de restreindre encore plus ce domaine suivant un critère basé sur le calcul des propriétés de sûreté, de manière semblable à la minimisation du modèle effectuée par la méthode de vérification "en arrière".
- Le coût de la construction d'un générateur aléatoire contraint est généralement inférieur à celui de la construction du modèle associé à un nœud de vérification. Cela devrait permettre l'utilisation des techniques de test comme un outil de vérification de secours quand la preuve s'avère impossible.

7.3 AUTRES APPROCHES DU TEST DES LOGICIELS SYNCHRONES

La récente parution d'un ensemble de travaux ayant pour objet le test des logiciels synchrones et plus particulièrement des logiciels spécifiés en LUSTRE, montre que ce thème suscite un certain intérêt aussi bien pour la recherche que pour l'industrie. Les trois approches que nous présentons ici tentent d'adapter au cas de LUSTRE un ensemble de techniques conçues dans d'autres contextes et dont les auteurs cherchent à montrer l'adéquation au problème particulier du test des programmes synchrones.

Le test statistique de programmes est une approche qui a été appliquée avec succès aux programmes séquentiels [Wae93]. Il consiste à analyser la structure du graphe de contrôle et à déterminer une loi de distribution pour les données d'entrée assurant une probabilité élevée de satisfaction des critères de couverture retenus. Ainsi, plutôt que de calculer les données exécutant un ensemble de chemins satisfaisant un critère, le programme est exécuté avec un grand nombre de données aléatoires dont la distribution permet de supposer que le critère a été satisfait. L'adaptation de cette technique au cas des programmes LUSTRE a fait l'objet d'une thèse [Maz94] dont nous présentons un bref résumé synthétique suivi d'une discussion au paragraphe 7.3.1.

La deuxième approche que nous analysons tente d'appliquer au cas de LUSTRE une méthode de génération de jeux de test à partir de l'analyse d'une spécification algébrique [BGM91], déjà évoquée au paragraphe 2.3.6. Cette méthode considère que la spécification du programme à tester est donnée sous la forme d'un module comportant des variables, des opérations ainsi qu'un ensemble d'axiomes spécifiant ces opérations. Un jeu de test est sélectionné pour chacun de ces axiomes. Une première thèse [Hsi94] a exploré la possibilité d'adapter cette méthode à LUSTRE dont les principaux résultats sont résumés au paragraphe 7.3.2.

Enfin, la dernière approche [MHM+95] repose sur l'utilisation conjointe de techniques de *test systématique* et de vérification formelle (LESAR). Elle est illustrée sur l'exemple de l'ascenseur que nous avons également utilisé (bien que les deux implantations soient différentes). Cette approche est présentée au paragraphe 7.3.3.

Nous finissons ce paragraphe par une comparaison globale des ces trois méthodes à notre approche en mettant en évidence ce qui, à notre avis, constitue son originalité.

7.3.1 Test statistique des programmes synchrones

La conception d'une technique statistique structurelle nécessite la définition d'un modèle de la structure du programme. Le modèle choisi dans le cadre de cette technique est l'automate construit par le compilateur LUSTRE qui représente le contrôle du programme. La taille de l'automate étant généralement très importante, l'approche adoptée consiste à définir deux techniques de test :

- Une technique de *test unitaire* adaptée au test des automates de taille réduite (typiquement les automates issus des nœuds qui ne font appel à aucun autre nœud). Cette technique est présentée au paragraphe 7.3.1.1.
- Une technique de *test d'intégration* qui procède à une simplification de l'automate en éliminant certaines informations associées aux nœuds appelés par le nœud sous test.

7.3.1.1 Test unitaire

Différents critères de sélection de jeux de test connus sur les automates sont examinés : la couverture des états, la couverture des transitions et la couverture des séquences.

La loi de distribution des données d'entrée, appelée *profil du test*, est calculée à partir de l'analyse du *graphe stochastique* du programme. Ce graphe n'est autre que l'automate produit par le compilateur dans lequel les conditions étiquetant les transitions sont remplacées par leur probabilité d'occurrence. Le profil du test est tel qu'il maximise la probabilité d'occurrence la plus faible étiquetant une transition du graphe stochastique.

Une fois le profil du test ainsi déterminé, le graphe stochastique est transformé en un graphe fortement connexe par suppression de l'état initial. Cette transformation permet d'effectuer le test avec une seule séquence d'entrées. La longueur de cette séquence est la *longueur du test* et est calculée en fonction de la *qualité du test* visée. La qualité du test par rapport à un critère *C* est la probabilité pour que le critère soit satisfait après exécution de la séquence d'entrées. Les notions de qualité et de longueur de test sont directement issues des techniques de test de circuits digitaux [ABF90].

La suppression de l'état initial du graphe stochastique n'est pas sans conséquences, puisque les éventuelles affectations de valeurs initiales aux variables (au moyen de l'opérateur LUSTRE ->) ne sont pas prises en compte. L'auteur affirme qu'une analyse manuelle postérieure permet de pallier ce problème.

Le test unitaire est complété par une recherche de *valeurs spéciales* d'entrée susceptibles d'entraîner des défaillances. La probabilité d'occurrence de ces valeurs reste très faible dans le cadre du test statistique. Ces valeurs concernent, en particulier, l'état initial de l'automate (exclu du test statistique) mais aussi certaines comparaisons figurant sur les transitions. L'analyse permettant la détermination de ces valeurs est manuelle ainsi que la construction des nouvelles séquences d'entrée correspondantes.

Une expérience visant à évaluer l'approche a été menée. Elle a porté sur une application industrielle et a montré que la couverture des états assure une détection de défauts insuffisante. Ceci est naturel étant donné qu'une partie importante des informations de l'automate (calcul des sorties et des états successeurs) sont portées par ses transitions. La couverture des transitions semble être la solution la plus appropriée en termes de coût et d'efficacité. La couverture des séquences, même d'une longueur au plus égale à 2, ne peut être envisagée que pour des automates de très petite taille, la longueur du test devenant très rapidement prohibitive.

La technique utilisée pour l'évaluation de cette approche a été l'analyse de mutation, dont nous avons esquissé le principe au chapitre 2. Les mutations considérées ont été les suivantes :

- Modification d'un symbole du programme LUSTRE (constante, constantes importées, variables, tableaux, appels de fonctions importées).
- Modification des opérateurs arithmétiques, logiques et relationnels.
- Insertion et suppression d'opérateurs unaires.

Les résultats de cette expérience ont montré qu'après un test statistique satisfaisant le critère, de couverture des états, seulement 65% des 311 mutants construits ont été tués. Au contraire, la couverture des transitions a tué pratiquement la totalité des mutants (99%). Après recherche des valeurs spéciales ces taux sont respectivement passés à 92% et 100%. Notons que dans cet exemple, la couverture des séquences de longueur 2 n'a pas eu de meilleurs résultats que la couverture des transitions.

Un dernier problème abordé dans le cadre du test unitaire est le choix des nœuds sur lesquels s'appliquera la méthode ci-dessus. En effet, contrairement aux langages procéduraux habituels, les nœuds ne sont pas compilés séparément, une "mise à plat" du programme étant effectuée avant la génération du code. Il n'est donc pas toujours envisageable d'appliquer une telle technique de test unitaire à un nœud qui fait appel à un ou plusieurs autres puisque dans ce cas la taille de l'automate produit peut rendre toute tentative de couverture des transitions impossible. De plus, l'analyse du graphe stochastique est une opération complexe qui devient d'un coût prohibitif au fur et à mesure que la taille de l'automate augmente.

En pratique, on ne teste de cette manière que les nœuds "terminaux", c'est à dire ceux dont la réalisation ne nécessite l'appel d'aucun autre nœud. Toutefois, la formulation de ce critère de choix est imprécise car la complexité de l'automate d'un nœud, même terminal, peut être importante, même si le nœud est de "petite taille" (i.e. comporte peu d'équations).

7.3.1.2 Test d'intégration

La technique de test unitaire présentée plus haut ne peut s'appliquer qu'à des nœuds dont l'automate associé est de petite taille. Ainsi, des nœuds plus complexes sont testés suivant une technique de test dite d'intégration. Les objectifs de cette technique sont :

- Vérifier les *conditions d'appel*, c'est à dire s'assurer que les appels de nœuds sont faits au bon moment.
- Vérifier que *le calcul des paramètres* d'appel est correct.
- Vérifier que le *résultat fourni* par un nœud appelé sont correctement utilisé par le nœud appelant pour le calcul de ses sorties.

Deux modèles du programme sont utilisés pour le test d'intégration, le *graphe d'appel* et les *automates d'intégration*. Le graphe d'appel est un graphe orienté qui décrit les

dépendances entre nœuds du programme. Chaque sommet de ce graphe est un nœud LUSTRE du logiciel à tester. Si le nœud n_i appelle le nœud n_j, alors dans ce graphe les sommets associés à ces deux nœuds sont reliés par un arc partant du sommet associé à n_i.

L'automate d'intégration d'un nœud *n* est obtenu en simplifiant l'automate produit par le compilateur LUSTRE. Pour cela, on supprime dans ce dernier toutes les informations relatives à la réalisation des nœuds appelés par *n* et qui ont déjà été testés. La technique de test unitaire peut donc s'appliquer à ce nouvel automate.

Tous les nœuds LUSTRE peuvent donc être testés selon cette approche. Le test s'effectue par *niveaux d'intégration*. Le premier niveau comprend les nœuds terminaux (n'appelant aucun autre nœud). Le niveau d'ordre *n* comprend les nœuds qui n'appartiennent pas à des niveaux d'ordre inférieur à *n* et qui ne font appel qu'à des nœuds de niveau inférieur ou égal à *n-1*.

7.3.1.3 Discussion

Bien que le thème de ces travaux soit très proche du nôtre, nous identifions très peu de points communs entre eux. En effet, seules les motivations des deux travaux sont communes, puisqu'elles admettent l'insuffisance de la vérification formelle pour la validation des logiciels synchrones. Les principales différences entre cette approche et notre travail sont les suivantes :

- La taille des modèles utilisés pour la vérification formelle n'est pas ressentie dans ces travaux comme un problème potentiel. La nécessité du test découle du besoin de validation du logiciel au moyen de son exécution. De plus, toutes les techniques proposées sont fondées sur l'utilisation des automates que produit le compilateur (elles supposent donc que la taille de ces automates n'est pas explosive).

  Dans le cadre de notre travail et plus particulièrement des techniques de test fonctionnel, nous représentons les modèles manipulés sous une forme compacte (variables et fonctions booléennes). Cette représentation a été adoptée pour prendre en compte des cas où la taille des modèles rend leur génération impossible ou fastidieuse. En revanche, elle ne permet pas d'appliquer des techniques de test statistique comparables à celle que nous venons d'étudier, puisque il n'est pas possible d'associer de manière simple de probabilités d'occurrence aux transitions.

- L'automate associé à un programme LUSTRE n'est pas à notre avis un modèle de la structure du programme mais un modèle de son *comportement*. Ainsi, bien qu'on puisse appliquer des critères de couverture de cet automate il semble difficile de déterminer leur signification sur le programme LUSTRE original. Pour cette raison, d'ailleurs, la technique de test statistique présentée plus haut se définit comme une "approche de test pour des programmes synchrones".

La technique de test structurel que nous avons proposée s'appuie sur le réseau d'opérateurs du programme. Notre choix, comme nous l'avons expliqué au chapitre 5, est motivé par la possibilité de définir des critères spécifiques au programmes LUSTRE assurant la couverture de leurs composants (opérateurs, arcs, chemins) et d'éviter ainsi que des parties du code ne soient jamais exécutées.

- Nous pouvons reprocher à la technique de test statistique proposée de ne s'intéresser qu'à la couverture de l'automate sans examiner la conformité des données de test utilisées aux contraintes d'environnement. En effet, comme nous l'avons mentionné au chapitre 1, l'automate produit après la compilation ne tient compte que des assertions qui permettent d'optimiser la taille du code engendré. En conséquence, les défaillances éventuelles provoquées par les données de test ainsi engendrées peuvent correspondre non pas à un défaut du logiciel mais à un non respect de la spécification de l'environnement.

  Notre approche peut apporter une solution à ce problème. Cette solution consisterait à utiliser un simulateur d'environnement capables de prendre en compte des profils opérationnels (cf. chapitre 4) et d'affecter aux variables d'entrée les probabilités d'occurrence résultant de la détermination du profil de test.

- Un des points importants traités dans ces travaux, que nous n'avons que très partiellement abordé dans notre travail, est la prise en compte de conditions faisant intervenir des variables numériques. En effet, l'adoption d'une génération de données aléatoire facilite le traitement de telles variables.

Une dernière critique que nous pouvons formuler concerne la définition du niveau unitaire et du niveau d'intégration. En effet, le niveau unitaire est défini non pas sur la taille du nœud LUSTRE concerné mais sur la complexité de l'automate qui lui est associé. Cela peut rendre difficile la planification des activités de test. En effet, l'automate d'un nœud dont la complexité initiale permet le test unitaire peut après une - même très petite - modification (destinée, par exemple, à corriger un défaut découvert lors du test) devenir de taille explosive.

Quant au critère d'intégration, il nécessite la génération de l'automate complet d'un nœud avant de le simplifier en automate d'intégration. Cette démarche écarte, encore une fois, des logiciels dont l'automate est d'une taille explosive.

7.3.2 Test à partir d'une modélisation algébrique de LUSTRE

7.3.2.1 Aperçu des travaux

Pour que l'utilisation d'une technique fondée sur l'existence de spécifications algébriques soit rendue possible, il est nécessaire de montrer que tout programme LUSTRE peut être transformé en une telle spécification. Hsiao [Hsi94] a montré qu'une telle transformation est possible en introduisant une algèbre particulière. Les valeurs de cette algèbre sont les suites de valeurs des expressions LUSTRE (des "flots") tandis que

ses fonctions sont des opérations sur ces suites. Un flot est de longueur finie mais non bornée.

Cela permet de modéliser tout programme LUSTRE par un module de spécifications algébriques. Les axiomes de ce module sont les équations et les assertions du programme LUSTRE.

La modélisation de LUSTRE proposée tient compte de toutes les caractéristiques du langage y compris les variables numériques et les horloges multiples. Ces dernières semblent avoir été une des difficultés principales de la modélisation.

L'étude approfondie de l'adéquation de cette approche à la dérivation de jeux de test étant actuellement en cours [Cal95], nous nous contentons d'en exposer brièvement ici les principes et les résultats qui nous paraissent significatifs.

La génération des jeux de test est effectuée au moyen de l'outil LOFT [BGM91] qui implante la méthode originale de test à partir de spécifications algébriques. Pour chacune des équations du programme LUSTRE (i.e. les axiomes du module) l'outil peut calculer un ensemble de séquences de données d'entrée ainsi que la suite de valeurs que prend la variable en partie gauche de l'équation pour chaque telle séquence. Il est nécessaire pour cela de préciser la longueur des séquences à engendrer.

LOFT est une machine PROLOG spécialisée : il procède à une évaluation symbolique de l'expression en partie gauche de l'équation par substitutions successives des variables y apparaissant en considérant tous les cas de figure possibles. Cette manière de procéder rend impossible l'application directe de la technique pour des longueurs de séquences d'entrées importantes, le nombre de cas à considérer devenant explosif. Il est possible, toutefois, de restreindre le nombre de cas considérés lors de l'analyse de l'expression en insérant judicieusement dans le processus d'analyse des clauses adéquates.

7.3.2.2 Discussion

Cette méthode peut être utilisée pour le test d'un logiciel implanté en un langage autre que LUSTRE mais dont on a une spécification LUSTRE. En disposant des données d'entrée et des valeurs des variables de sortie associées calculées par LOFT, on peut vérifier que l'implantation sous test produit cette même sortie.

L'intérêt de la méthode est cependant moins clair quand le logiciel exécutable testé est issu de la compilation du programme LUSTRE servant à la génération de données. Une application de cette technique serait la vérification de la correction du compilateur LUSTRE.

7.3.3 Test systématique et vérification formelle pour la validation des logiciels synchrones

7.3.3.1 Aperçu des travaux

Selon la méthodologie exposée [MHM+95], le test devrait être utilisé dans le but d'acquérir rapidement une certaine confiance dans les spécifications ainsi que dans le programme à valider. Une fois cette confiance acquise, la vérification formelle est utilisée pour prouver la validité des propriétés de sûreté. Cette vue de la validation nous conforte dans notre travail dont les motivations sont similaires et confirme, par ailleurs, la complémentarité entre le test et la vérification formelle (cf. paragraphe 7.2).

Le principe de vérification formelle basée sur l'utilisation de LESAR ayant été présenté en détail au chapitre 1, nous nous contentons d'étudier ici la partie des travaux portant sur le test qui nous intéresse plus particulièrement.

La technique de test utilisée est de type "boîte noire". Elle utilise pour la sélection des jeux de test une spécification comportementale du logiciel donnée sous la forme d'un automate d'états finis. Contrairement à l'approche de test statistique présentée précédemment, cet automate n'est pas celui construit par le compilateur mais est développé de manière indépendante.

Suite au problème de l'explosion du nombre de ses états, cet automate ne représente qu'une abstraction du comportement réel du logiciel. Plus précisément, seules les informations relatives au contrôle interviennent à la construction des états. Les transitions sont étiquetées par deux informations :

- Une *condition d'activation*, qui est un prédicat portant sur les entrées du logiciel. Ce prédicat doit être satisfait pour que la transition soit exécutée.
- Un *calcul*, qui correspond à l'ensemble des calculs effectués par le logiciel au moment de l'exécution de la transition.

Cet automate est utilisé à la fois de base formelle pour la dérivation des jeux de test et comme oracle pour la détection des comportements erronés du logiciel.

Pour la constitution de l'ensemble de jeux de test plusieurs critères de sélection définis sur l'automate sont considérés tels que couverture des états et des transitions. L'ensemble des prédicats présents sur les transitions peuvent également donner lieu à la définition de critères tels que la couverture des conditions multiples (consistant à considérer les conditions élémentaires dans une condition composée) ou le test aux limites. Ce dernier consiste à exécuter le logiciel pour des valeurs de variables numériques se situant à la frontière des intervalles rendant respectivement vraie ou fausse une condition. Enfin, un critère plus original est celui des *entrées simultanées* qui cherche à détecter des défauts dans la prise en compte par le logiciel de la modification simultanée de deux variables d'entrée.

La sélection des jeux d'essai est systématique mais leur génération n'est pas faite de manière automatique. Ceci est en particulier dû à la multitude des critères de sélection utilisés et à l'existence de séquences de transitions impossibles à exécuter.

7.3.3.2 Discussion

Bien que ces travaux soient animés par des motivations similaires aux nôtres, la démarche proposée présente une différence significative. En effet, l'approche de test systématique proposée est fondée sur l'existence d'une spécification du comportement du logiciel autre que son implantation. Cela implique un effort de spécification supplémentaire qui, dans la plupart des cas, n'est pas nécessaire dans nos techniques. En contrepartie, l'existence d'un modèle précis du comportement du logiciel présente des avantages incontestables quant à la sélection des jeux de test, puisque la recherche des défauts peut se situer à des niveaux beaucoup plus fins que la simple recherche de violations de propriétés de sûreté.

Une autre différence réside dans l'utilisation de critères de sélection destinés à la recherche de défauts particuliers. Dans le cadre de notre travail nous n'ayons par étudié de nombreux critères; nous voyons donc dans cette perspective une amélioration intéressante de nos techniques.

Notons, enfin, que l'environnement du logiciel réactif n'est pas pris en compte par la méthode de test, la spécification utilisée modélisant uniquement le comportement du logiciel.

7.3.4 Bilan synthétique

Le point commun des approches que nous venons de survoler est qu'il s'agit d'adaptations au cas des logiciels synchrones de techniques conçues à l'origine pour des logiciels ne possédant pas cette caractéristique. De ce fait, les deux premières techniques procèdent à la transformation des programmes LUSTRE en des modèles abstraits adaptés à leurs besoins tandis que la dernière repose sur l'existence d'une spécification supplémentaire.

Il s'agit là de la première différence avec notre démarche qui n'a pas consisté à étudier l'adaptation d'une méthode existante mais de concevoir des nouvelles techniques qui tirent profit des caractéristiques particulières de LUSTRE (langage de programmation - logique temporelle).

Une autre originalité de notre approche, et plus particulièrement des techniques de génération aléatoire sous contraintes, est que la génération des jeux de test est dynamique : les entrées fournies au logiciel sont calculées au moment de l'exécution de ce dernier et en fonction de ses réactions. Il serait de ce fait intéressant d'étudier les moyens d'effectuer du test statistique en définissant des profils opérationnels de la même manière que les profils de test.

Bien que le test des propriétés de sûreté définit un critère de sélection en orientant le test vers la détection des défauts particuliers liés à la sûreté, il est clair que d'autres critères doivent être étudiés et évalués de manière expérimentale.

De plus, la prise en compte de programmes comportant des variables numériques est le défi principal pour notre travail et constitue un de ces points faibles vis à vis des autres approches présentées ici.

7.4 TEST DE PROTOCOLES

Le test de protocoles est assimilable à un test de type fonctionnel et est formulé en général sous la forme d'une question :

"Un logiciel L est-il conforme à une spécification S ?"

Pour donner un sens précis à cette question il faut définir formellement la notion de conformité. Ce problème dépend des modes d'expression du programme et des spécifications.

Dans le modèle le plus commun - le modèle comportemental- l'implantation et les spécifications sont représentées chacune soit par une expression qui décrit ses comportements observables soit directement par un modèle opérationnel de comportement. Les expressions sont fournies dans des langages comme Estelle, Lotos ou LDS tandis que les modèles opérationnels sont souvent des automates d'états finis ou des systèmes de transitions.

La conformité est alors définie par une relation, dite d'implantation, entre le modèle de l'implantation et celui des spécifications. Le choix de cette relation est arbitraire, bien que généralement il se porte sur une des relations bien adaptées au formalisme retenu. Dans le cas du formalisme des machines d'entrées-sorties, le plus proche du nôtre, diverses relations d'implantation sont proposées. On peut vouloir s'assurer qu'un logiciel réalise au moins toutes les spécifications (c'est à dire qu'il ne la contredit jamais) sans se soucier de ses réactions pour des comportements non prévus dans ces spécifications. On peut également souhaiter montrer que l'implantation réalise exactement les spécifications. C'est ce dernier cas (connu comme "l'équivalence de traces") qui est le plus souvent retenu.

Pour décider de la conformité, on s'appuie sur des données de test qui sont des séquences d'entrées-sorties correspondant aux échanges d'interactions entre le logiciel sous test et son environnement. Ces séquences sont produites sans disposer du logiciel et stipulent des comportements du logiciel, à partir d'un point de vue externe à ce dernier. Elles consistent en des comportements autorisés par la spécification et d'autres qui ne lui sont pas conformes. En effet, le but du test étant de dévoiler des défauts, la mise en évidence de comportements incorrects est également recherchée. Le verdict de conformité est prononcé au vu des résultats des exécutions du logiciel pour ces différentes données de test. Les séquences d'entrées-sorties sont construites

par composition d'un testeur et d'un objectif de test, éventuellement de manière automatique. Le testeur est lui-même dérivé automatiquement ou manuellement des spécifications qui explicitent toutes les émissions et réceptions. Il consiste en un automate qui fournit une image miroir de tous les comportements prévus par la spécification. Il est ensuite complété par le rajout des transitions de détection d'erreurs et d'un état caractérisant l'échec du test; lorsque le testeur reçoit une interaction qui n'est pas censée arriver dans un état donne, il passe dans cet état d'échec. L'objectif de test correspond à des situations plus ou moins abstraites, énoncées par les responsables du test, qu'il faut examiner. Un objectif de test peut donc être aussi bien un scénario particulier qu'une propriété plus générale (de service par exemple). L'objectif de test sert à réduire la taille de l'ensemble des comportements à analyser pour émettre un verdict.

Sur le plan formel, ce type de test revient à réaliser la composition parallèle avec communication synchrone d'un automate testeur (éventuellement réduit par l'objectif de test) déterministe et de l'automate du logiciel sous test. Il présente l'avantage de pouvoir être mis en pratique immédiatement lorsque l'on teste en un temps fini des implantations déterministes. Les principaux problèmes de mise en oeuvre demeurent la construction des données de test, la caractérisation du verdict, et la sélection d'un nombre raisonnable de données de test. Dans [Du 96] une étude a été menée sur les possibilités de construire des données de test pour les applications synchrones sur le modèle des testeurs dans le domaine des protocoles. Elle répond au besoin de pouvoir assurer le test quand certains composants, les spécifications ou les objectifs de test, ne sont pas fournis en LUSTRE mais dans un formalisme à base d'automates, plus précisément les automates d'entrées-sorties dans lesquels les transitions portent distinctement une entrée ou une sortie. Le but assigné au test est de révéler les violations des propriétés de sûreté, la spécification comprenant les invariants d'environnement et les propriétés de sûreté. Les applications sont supposées n'opérer que sur des booléens.

La relation d'implantation retenue est l'inclusion des comportements du logiciel dans ceux de la spécification. Le principe de construction du testeur s'appuie sur un automate de comportements qui représente les interactions entre l'environnement et le logiciel, sous le point de vue du logiciel. Les réactions de l'implantation sont limitées à celles satisfaisant les propriétés de sûreté; aucune autre spécification des fonctions du logiciel n'est fournie. L'automate de comportements est bi-parti pour distinguer les entrées des sorties ainsi que les états de l'environnement de ceux du logiciel à tester. Il peut être construit automatiquement à partir d'une spécification en LUSTRE. Le testeur est obtenu à partir de cet automate en associant à chaque état la transition inverse de l'automate de comportement et en créant les états d'échec nécessaires. Le testeur représente bien alors les comportements autorisés, vus de l'environnement.

Cette étude a mis en évidence des similitudes et des différences entre le test de protocoles et le test des applications synchrones. On remarque d'abord que la possibilité de production de données de test sans disposer de l'implantation est similaire aux techniques d'animation de spécifications et plus particulièrement à l'interaction entre un simulateur d'environnement et un simulateur de logiciel sûr. Toutefois, les séquences ainsi produites sont difficilement utilisables pour la validation de l'implantation,

puisque les spécifications animées n'offrent qu'une description non déterministe du logiciel. L'utilisation d'un oracle est le seul moyen automatique pour vérifier la conformité du comportement du logiciel à ces propriétés. Enfin, la prise en compte d'objectifs de test formulés en LUSTRE n'introduit aucune difficulté : il faut adjoindre ces dernières expressions aux propriétés de sûreté pour qu'elles soient prises en compte dans le simulateur de logiciel sûr.

Conclusion

RAPPEL DU CONTEXTE ET DES OBJECTIFS

Le travail que nous venons de présenter s'inscrit dans le cadre du développement de méthodes et outils formels pour la spécification, la programmation, la vérification et la validation des programmes réactifs à l'aide du langage synchrone LUSTRE. Il vient enrichir une famille d'outils accompagnant ce langage et comprenant un compilateur efficace et un outil de preuve de propriétés de sûreté. Ces outils constituent un cadre rigoureux de développement qui favorise la prévention des fautes, grâce à la définition formelle de la sémantique du langage, et l'élimination de fautes, plus précisément de celles concernant les propriétés de sûreté.

Nous avons voulu renforcer ce cadre formel en lui adjoignant de nouveaux outils pour étendre les capacités d'élimination de défauts. L'approche qui nous a semblé la plus appropriée pour cela est le test des logiciels. Notre préférence pour ce type de vérification et validation est justifiée par le fait qu'il est le moyen de recherche de défauts le plus largement utilisé, en particulier en milieu industriel.

Ce travail a été réalisé sans négliger la multitude d'approches de test existantes (formelles ou informelles, automatiques, systématiques ou empiriques) pour en concevoir des nouvelles, adaptées à notre cadre de développement rigoureux. Ces nouvelles techniques doivent permettre en particulier la vérification et validation automatique, une de leurs fonctions étant de compléter ou de se substituer à la preuve. Nous avons, de ce fait, attaché une importance particulière à leur définition formelle, d'autant plus que nous tenions à pouvoir les comparer avec celles de preuve.

CONTRIBUTION

Nous avons commencé ce document par l'étude approfondie des techniques de compilation et de preuve des programmes LUSTRE. Nous avons ensuite replacé le problème de la vérification et de la validation des logiciels synchrones par rapport à celui, plus général, de l'élimination des défauts.

Cette étude a montré qu'une approche de vérification et de validation exclusivement fondée sur la preuve présente deux types d'insuffisances :

- Les insuffisances liées à la validation des spécifications utilisées : la preuve repose sur des spécifications de l'environnement et des propriétés de sûreté dont l'exactitude et la complétude demandent à être établies.
- Les insuffisances concernant le processus de preuve : fondé sur l'exploration de modèles de taille importante, il ne peut s'intéresser, d'une part, qu'à un nombre restreint de défauts liés aux propriétés de sûreté et reste exposé, d'autre part, au danger de l'éventuelle explosion du nombre d'états.

Partant de ces deux constatations, nous avons cherché à définir un cadre méthodologique qui prend en compte et pallie ces insuffisances. Nous avons ainsi abordé le problème du test avec deux motivations : la validation des spécifications et la recherche des défauts.

- La validation des spécifications a deux objectifs. D'une part renforcer la confiance des concepteurs dans leurs spécifications avant la preuve, et d'autre part, assister ces derniers dans le travail complexe de rédaction d'une spécification correcte.
- La recherche de défauts vise à compléter la preuve, en s'intéressant à des types de défauts que cette dernière ignore, mais aussi à la remplacer quand son application s'avère impossible.

Les techniques que nous avons développées sont variées comme les besoins qu'elles doivent satisfaire. Il s'agit, d'une part, de techniques fonctionnelles qui permettent la génération automatique de données à partir des spécifications de l'environnement et des propriétés de sûreté et, d'autre part de techniques structurelles, fondées sur la structure du programme LUSTRE.

Deux des techniques fonctionnelles, la simulation de l'environnement et la simulation d'un logiciel sûr, permettent de procéder à l'animation des spécifications en cours de rédaction et, par observation, à leur validation. Elles améliorent de ce fait la prévention des défauts en fournissant un cadre plus riche et plus simple pour le développement des spécifications.

La première de ces techniques (simulation de l'environnement) a aussi pour objectif la recherche des défauts, puisqu'elle peut produire des données de test pour le logi-

ciel. Cette technique définit le critère de sélection de jeux de test le plus élémentaire pour les logiciels réactifs.

Afin de permettre une meilleure détection des défauts liés au non respect des propriétés de sûreté du logiciel, nous avons développé une dernière technique, le test des propriétés de sûreté. Il s'agit d'un critère de sélection plus restrictif que la simulation d'environnement simple, puisqu'il met l'accent sur un type particulier de défaut tout en respectant les contraintes d'environnement.

L'étude de techniques de type structurel a été motivée par la difficulté pour ces techniques fonctionnelles d'assurer la détection de défauts liés à l'activité de programmation. Le réseau d'opérateurs nous a semblé être le modèle le plus naturel de la structure d'un programme LUSTRE. Des critères de couverture des différents éléments du réseau ont été définis, au même titre que pour le graphe de contrôle utilisé dans le cas des langages séquentiels.

Enfin, nous nous sommes intéressés à deux autres moyens de la sûreté de fonctionnement, la prévision de fautes et la tolérance aux fautes qui sont absents du cadre actuel de développement. Nous avons ainsi montré qu'un simulateur d'un logiciel sûr peut être utilisé afin d'améliorer la tolérance aux fautes d'un logiciel réactif. De plus, nous avons étudié les moyens de reproduire les comportements de l'environnement du logiciel les plus proches de la réalité. Cela peut s'obtenir en associant des probabilités d'occurrence aux entrées du logiciel et en les intégrant au processus de simulation d'environnement. Cette prise en compte de probabilités est une condition nécessaire à l'évaluation de la fiabilité du logiciel.

Toutes ces techniques ont fait l'objet d'une définition formelle. En particulier, les techniques fonctionnelles ont été définies sur un modèle semblable à celui utilisé dans le cadre de la vérification formelle, ce qui facilite la comparaison entre les deux approches. La définition de ce tel modèle a également permis l'expression claire et concise des principes des techniques.

La dernière activité de ce travail était de nature pratique et expérimentale. Elle a consisté en la mise en œuvre d'une partie des techniques développées ainsi qu'en la réalisation d'une application synchrone qui a servi à leur évaluation. Cette application, bien que restreinte, a partiellement comblé l'absence d'application à un logiciel de type industriel et a permis de réaliser des observations dont certaines nous ont amené à des conclusions intéressantes. En particulier, nous avons pu nous convaincre de l'utilité des techniques servant à l'animation des spécifications mais aussi de la nécessité de définir un plus grand nombre de critères de sélection pour une meilleure détection de défauts.

En récapitulant, notre contribution peut se résumer en trois points :

- Définition d'un cadre méthodologique de vérification et de validation de logiciels réactifs spécifiés en LUSTRE compatible avec la preuve formelle. Le concepteur peut dans un premier temps utiliser les techniques de test pour vérifier et valider l'ensemble de spécifications développées avant de procéder à la preuve :

la confiance dans le résultat de cette dernière ne saurait qu'être renforcée. De plus, en cas d'impossibilité d'application de la preuve, les techniques de test permettent d'obtenir des informations précieuses sur la qualité du logiciel développé.

- Les techniques développées ont fait l'objet d'une définition formelle qui a permis, en particulier, de caractériser les spécifications utilisées et de mettre en évidence les similitudes et les différences avec la vérification formelle ainsi qu'avec d'autres techniques de test de logiciels réactifs.
- Enfin, bien que de manière succincte, nous avons mené une expérience qui nous a permis d'identifier certains des points forts et faibles de nos techniques et de mieux définir les améliorations à apporter. En particulier, nous avons pu nous rendre compte de la difficulté de spécifier l'environnement d'un logiciel et ses propriétés de sûreté qui rend leur animation encore plus utile.

CRITIQUES ET PERSPECTIVES

La principale critique que nous pouvons formuler concerne l' évaluation pratique des techniques de test. L'expérience que nous avons menée n'a pas été assez détaillée, en particulier pour le test structurel, dont le pouvoir de détection de défauts devrait être évalué de manière plus précise. De plus, une véritable évaluation ne peut s'opérer qu'à travers plusieurs études de cas, incluant en particulier des applications industrielles.

Bien que limitée, l'expérience que nous avons conduite a mis en évidence un certain nombre de faiblesses de nos techniques : la simulation d'environnement et le test des propriétés de sûreté sont dans l'incapacité de détecter certains types de défauts. De plus, les critères de couverture structurelle semblent être insuffisants.

Enfin, les extensions de la génération aléatoire sous contraintes aux profils opérationnels et aux contraintes de type numérique n'ont pas été évaluées.

Ces critiques nous dictent un premier ensemble d'actions qui constituent les perspectives à court terme à l'issue de ce travail. En particulier :

- L'ensemble des techniques proposées doivent être implantées et évaluées sur l'exemple de logiciel de contrôle d'ascenseur. La technique de mutation, que nous avons très partiellement appliquée, devrait permettre d'évaluer le pouvoir de détection de défauts global de ces techniques.
- Nos techniques pourront être également évaluées dans le cadre de l'étude de la *validation incrémentale de services téléphoniques* que nous allons mener dans le cadre d'un contrat de recherche avec le Centre National d'Etudes des Télécommunications. Le but de cette étude et d'explorer la possibilité d'exprimer en LUSTRE des services pour ensuite pouvoir les valider au moyen des techniques d'animation de spécifications que nous avons développées.

Ce travail a également mis en évidence plusieurs directions concrètes de recherche future qui portent aussi bien sur le cadre méthodologique que sur la définition et la mise en œuvre des techniques de test :

- La méthodologie proposée permet d'envisager l'intégration de l'ensemble des techniques de vérification et de validation (preuve formelle et test) au sein d'un environnement commun de développement de logiciels synchrones. L'outil interactif que nous sommes en train de développer devrait faciliter cette intégration.
- Le développement de nouvelles techniques de test est nécessaire. Ce besoin est plus ressenti pour le test structurel pour lequel les critères définis n'ont pas convaincu par leur efficacité. La prise en compte d'autres types de propriétés (par exemple de vivacité) est également une voie à explorer.
- Enfin, l'implantation des techniques de génération aléatoire doit être améliorée afin de minimiser le temps nécessaire au calcul des fonctions composant les modèles manipulés.

Enfin, plusieurs points n'ont pas été abordés dans cette étude :

- Nous avons constaté que le développement des spécifications sous forme d'invariants est une opération délicate à cause de la forte probabilité de spécification excessive ou insuffisante. De plus, en fonction de leur utilisation future (intégration dans un oracle, preuve, simulation d'environnement ou test de propriétés de sûreté), les invariants doivent être exprimés de manière différente. Bien que nous ayons énoncé certains principes empiriques pour le cas de la simulation d'environnement, nous n'avons pas défini de manière formelle la manière dont ces spécifications doivent être développées.
- Les défauts recherchés ont été simplement classés en défauts de spécification et défauts d'implantation. Or, il est possible de les caractériser de manière plus fine afin de concevoir des techniques adaptées à leurs particularités. Une telle classification nécessite, en particulier, l'étude expérimentale de plusieurs exemples de programmes LUSTRE afin de se rendre compte des erreurs les plus souvent commises lorsqu'on utilise ce langage.
- L'identification et la correction des fautes dont la présence est détectée par le test est une activité essentielle pour l'élimination des défauts. Il est nécessaire, donc, d'intégrer dans un outil de test des fonctionnalités facilitant ce travail.

En récapitulant, nous estimons avoir par ce travail répondu aux objectifs initiaux, en définissant le cadre nécessaire au développement du test des applications synchrones, en posant les fondements nécessaires au développement de nouvelles techniques de test qui doivent, indéniablement, être développées et en montrant que le test est une technique parfaitement complémentaire à la preuve formelle.

Bibliographie

[ABF90] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[AG93] J. Atlee and J. Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, pages 24–40, Janvier 1993.

[AHLW95] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault Localization using Execution Slices and Dataflow Tests. In *6th International Symposium on Software Reliability Engineering*, Toulouse, France, Octobre 1995.

[Ake78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516, Juin 1978.

[AW85] E.A. Ashcroft and W.W. Wadge. *LUCID, the data-flow programming language*. Academic Press, 1985.

[BB91] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, september 1991.

[BDS91] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, september 1991.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[Ben89] A. Benveniste. Les langages synchrones : des logiciels pour la spécification et la conception des systèmes temps-réel de traitement de l'information. Rapport de recherche , INRIA/IRISA et groupement C2A, Rennes, France, Mai 1989.

[Ber86] J-L. Bergerand. LUSTRE : Un langage déclaratif pour le temps réel.

Thèse, INPG, Grenoble, France, Janvier 1986.

[BFH90] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation. In *Workshop on Computer-Aided Verification*, Rutgers (N.J.), Juin 1990.

[BGM91] G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6:387–405, 1991.

[BM94] A. Bertolino and M. Marré. Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. *IEEE Transactions on Software Engineering*, pages 885–899, Décembre 1994.

[Boe81] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, pages 667–692, Août 1986.

[BS87] V. Basili and R. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, 13:1278–1296, 1987.

[Cal95] R. Calippe. Principes d'une méthode de test à partir de textes lustre. In *Journés du GDR Programmation*, Grenoble, France, Novembre 1995.

[Cho78] T. Chow. Testing Software Design Modeled By Finite State Machines. *IEEE Transactions on Software Engineering*, pages 178–187, Mars 1978.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL 87), Munich*, pages 178–188. ACM, 1987.

[Cou91] O. Coudert. SIAM : Une Boîte à Outils pour la Preuve Formelle des Systèmes Séquentiels. Thèse, Ecole Nationale Supérieure des Télécommunications, Paris, France, Octobre 1991.

[CPRZ89] L. Clarke, A. Podgurski, D. Richardson, and S. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, pages 1318–1331, Novembre 1989.

[DKM+94] L. Dillon, G. Kutty, P. Melliar-Smith, L. Moser, and Y. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, Avril 1994.

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection : Help for the practicing programmer. *Computer, 11*, pages 34–41, 1978.

[DN84] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Trans-*

actions on Software Engineering, 10(4):438–444, Juillet 1984.

[Du 96] L. Du Bousquet. Caractérisation du test des applications synchrones. Rapport de DEA, DEA d'Informatique de Lyon, Ecole Normale Supérieure, Lyon, France, Juin 1996.

[Duc93] M. Ducassé. A pragmatic survey of automated debugging. In *Automated and Algorithmic Debugging. First International Workshop*, Linkoping, Sweden, Mai 1993. Springer Verlag.

[DY94] L. Dillon and Q. Yu. Oracles for Checking Temporal Properties of Concurrent Systems. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–153, New Orleans, USA, Décembre 1994.

[GG75] J. Goodenough and S. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, pages 156–173, Juin 1975.

[Glo89] A-C. Glory. Vérification de propriétés de programmes flots de données synchrones. Thèse, Université Joseph Fourier, Grenoble, France, Décembre 1989.

[Gou83] J. Gourlay. A MAthematical Framework for the Investigation of Testing. *IEEE Transactions on Software Engineering*, pages 686–709, Novembre 1983.

[Hal94] N. Halbwachs. BAC: A Boolean Automaton Checker (Version 1). Rapport interne , Vérimag, Grenoble, France, 1994.

[Ham94] D. Hamlet. Foundations of Software Testing : Dependability Theory. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, New Orleans, USA, Décembre 1994.

[Ham95] D. Hamlet. Software Quality, Software Process and Software Testing. *Advances in Computers*, 1995.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, september 1991.

[Hen80] K. Heninger. Specifying software requirements for complex systems : New techniques and their aplications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, Janvier 1980.

[Hil89] R. Hilal. Test d'une application décrite à l'aide du langage de programmation LUSTRE. Rapport de DEA, Institut National Polytechnique de Grenoble, Grenoble, France, Juin 1989.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*,

pages 785–793, Septembre 1992.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente*. Workshops in Computing, Springer Verlag, Juin 1993.

[How75] W. Howden. Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers*, pages 554–559, Mai 1975.

[How76] W. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):37–44, Septembre 1976.

[How87] W. Howden. *Functional Program Testing & Analysis*. McGraw-Hill Series In Software Engineering And Technology, 1987.

[HPOG89] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A-C. Glory. Specifying, Programming and Verifying Real-Time Systems, using a synchronous declarative language. In *Workshop on automatic verification methods for finite state systems, LNCS 407*, Grenoble, France, Juin 1989. Springer Verlag.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating Efficient Code From Data-Flow Programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), august 1991.

[Hsi94] N-C. Hsiao. Sélection de tests de propriétés de sûreté à partir d'une modélisation algébrique de programmes LUSTRE. Thèse, Université Paris-Sud, Orsay, France, Octobre 1994.

[HT90] D. Hamlet and R. Taylor. Partition Analysis Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, pages 1402–1411, Décembre 1990.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Hua75] J.C. Huang. An Approach to Program Testing. *Computing Surveys*, 7(3):113–128, Septembre 1975.

[ISO91] ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International standard, IS-9646, 1991.

[JM86] F. Jahanian and A. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, pages 890–904, Septembre 1986.

[JPVO95] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Safety Property Verification of Esterel Programs and Applications to Telecommunica-

tions Software. In *7th Conference on Computer-Aided Verification*, Juillet 1995.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*, Amsterdam, 1974.

[Kor90] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, pages 870–879, Août 1990.

[Lap92] J.C. (Ed.) Laprie. *Dependability : Basic Concepts and Terminology*. Springer-Verlag L.N.C.S, 1992.

[Lap93] J.C. Laprie. Informatique Sûre de Fonctionnement: Des Concepts Aux Limites. In *International Seminar held at LAAS-CNRS for its 25th anniversary*, pages 43–54, Toulouse, France, Mai 1993.

[Lap95] J.C. (Ed.) Laprie. *Guide de la sûreté de fonctionnement*. Cépaduès, 1995.

[Lev86] N.G. Leveson. Software Safety: Why, What, and How. *Computing Surveys*, 18(2), Juin 1986.

[Lev90] N.G. Leveson. The Challenge of Building Process-Control Software. *IEEE Software*, Novembre 1990.

[Lev91] N.G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2), Février 1991.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, september 1991.

[LR95] Y. Le Traon and C. Robach. Towards a Unified Approach to the Testability of Co-Designed Systems. In *6th International Symposium on Software Reliability Engineering*, Toulouse, France, Octobre 1995.

[Maz94] C. Mazuet. Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre. Thèse, Institut National Polytechnique de Toulouse, Toulouse, France, Décembre 1994.

[MHM+95] M. Müllerburg, L. Holenderski, O. Maffeis, A. Merceron, and M. Morley. Systematic Testing and Formal Verification to Validate Reactive Programs. *Software Quality Journal*, 4(4), 1995.

[Mül94] M. Müllerburg. Why Systematic Testing is Difficult : The Problem of the Sample. In *7th International Software Quality Week*, San Francisco, USA, Mai 1994.

[Mus93] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, Mars 1993.

[Mye79] G. Myers. *The Art Of Software Testing*. Wiley-Interscience, 1979.

[Nta88] S. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE*

Transactions on Software Engineering, pages 868–874, Juin 1988.

[OP94a] F. Ouabdesselam and I. Parissis. Testing Safety Properties of Synchronous Reactive Software. In *7th International Software Quality Week*, San Francisco, USA, Mai 1994.

[OP94b] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, pages 239–248, Monterey, USA, Novembre 1994.

[OP95a] F. Ouabdesselam and I. Parissis. Constructing Operational Profiles for Synchronous Critical Software. In *6th International Symposium on Software Reliability Engineering*, Toulouse, France, Octobre 1995.

[OP95b] F. Ouabdesselam and I. Parissis. Testing Techniques for Data-Flow Synchronous Programs. In *2nd International Workshop on Automated and Algorithmic Debugging*, Saint Malo, France, Mai 1995.

[Par96] I. Parissis. A Tool For Testing Synchronous Critical Software. In *Third International Conference on Achieving Quality in Software*, Florence, Italy, Janvier 1996.

[PB88] E. Pilaud and J-L. Bergerand. SAGA : A software Development Environment for Dependability Automatic Control. In *SAFECOMP'88*, Fulda, West Germany, 1988. IFAC.

[PH88] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In *Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Warwick, Septembre 1988. Springer Verlag.

[Pnu86] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems : a survey of current trends. *Current Trends in Concurrency, LNCS, Springer-Verlag*, 224:510–584, 1986.

[PO96a] I. Parissis and F. Ouabdesselam. Specification-based Testing of Synchronous Software. In *ACM SIGSOFT Fourth Symposium on the Foundations of Software Engineering*, San Francisco, USA, Octobre 1996.

[PO96b] I. Parissis and F. Ouabdesselam. Techniques de Test pour des Logiciels Réactifs Synchrones. In *Modélisation des Systèmes Réactifs*, Brest, France, Mars 1996.

[PvSPK90] David Parnas, John van Schouwen, and Shu Po Kwan. Evaluation of Safety-Critical Software. *Communications of the ACM*, 33(6):636–648, Juin 1990.

[PZ93] A. Parrish and S. Zweben. Clarifying Some Fundamental Concepts in Software Testing. *IEEE Transactions on Software Engineering*, pages 742–746, Juillet 1993.

[Rat92] C. Ratel. Définition et réalisation d'un outil de vérification formelle de

programmes Lustre: Le système Lesar. Thèse, Université Joseph Fourier, Grenoble, France, Juin 1992.

[Ray91] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : le générateur de code LUSTRE-V3. Thèse, Institut National Polytechnique de Grenoble, Grenoble, France, Novembre 1991.

[RC85] D. Richardson and L. Clarke. Partition Analysis : A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, pages 1477–1490, Décembre 1985.

[RLO92] D. Richardson, S. Leif Aha, and T. O'Malley. Specification-based Test Oracles for Reactive Systems. In *14th Int'l Conf.on Software Engineering*, pages 105–118, Melbourne, Australia, Mai 1992.

[ROT89] D. Richardson, O. O'Malley, and C. Tittle. Approaches to Specification-Based Testing. In *ACM/SIGSOFT'89(TAV3)*, pages 86–96, Décembre 1989.

[RRSV87] J-L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. *Xesar: a Tool for Protocol Validation - User Manual - 1.2 edition*. Laboratoire de Génie Informatique, Grenoble, France, Sep. 1987.

[RW85] S. Rapps and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, pages 367–375, Avril 1985.

[RW87] P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. CONTROL AND OPTIMIZATION*, 25(1):206–230, Janvier 1987.

[Sha38] C.E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions AIEE*, 57:305–316, 1938.

[Thé89] P. Thévenod-Fosse. Software validation by means of statistical testing : retrospect and future direction. In *1st Working Conference on Dependable Computing for Critical Applications*, pages 15–22, Santa Barbara, USA, Août 1989.

[Wae93] H. Waeselynck. Vérification de logiciels critiques par le test statistique. Thèse, Institut National Polytechnique de Toulouse, Toulouse, France, Janvier 1993.

[Wey86] E. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, pages 1128–1138, Décembre 1986.

[Wey88] E. Weyuker. The Evaluation of Program-based Software Test Data Adequacy Criteria. *Communications of the ACM*, pages 668–675, Juin 1988.

[WF89] R.D. Wallace and R.U. Fujii. Verification and Validation : Techniques to Assure Reliability. *IEEE Software*, Mai 1989.

[WHH80] M. Woodward, D. Hedley, and M. Hennell. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, SE-6(3):278–286, Mai 1980.

[Whi81] L. J. White. Basic Mathematical Definitions and Results in Testing. *Proc. computer program testing, North Holland*, pages 13–24, 1981.

[Whi92] J. Whittaker. Markov chain techniques for software testing and reliability analysis. Thesis, University of Tenessee, Mai 1992.

[WO80] E. Weyuker and T. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, Mai 1980.

[Woi93] D. Woit. Specifying Operational Profiles for Modules. In *International Symposium on Software Testing and Analysis*, pages 2–10, Cambridge, Massachusetts, USA, Juin 1993.

[WWH91] E. Weyuker, S. Weiss, and D. Hamlet. Comparison of Program Testing Stategies. In *Symposium on Testing, Analysis and Verification (TAV)*, Victoria, British Columbia, Octobre 1991.

[ZG89] S. Zweben and J. Gourlay. On the Adequacy of Weyuker's Test Data Adequacy Axioms. *IEEE Transactions on Software Engineering*, pages 496–500, Avril 1989.

ANNEXE A

Format de description des automates booléens (BAC)

PRÉSENTATION DE L'OUTIL BAC

BAC (*"Boolean Automaton Checker"*) [Hal94] est un outil de vérification d'automates booléens (i.e. dont les entrées sont exclusivement des signaux booléens). Les états d'un automate booléen sont décrits au moyen d'un vecteur de variables (*variables d'état*) : un état correspond à une valeur particulière des variables du vecteur.

Avant d'être traité par l'outil, un automate booléen doit être décrit dans la syntaxe donnée dans la figure A-1. Suivant cette syntaxe, il y a quatre types de variables dans un automate booléen :

- Les *variables d'état* servent à la définition des états de l'automate.
- Les *variables d'entrée* sont les signaux booléens correspondant aux stimuli externes reçus par l'automate.
- Les *variables de sortie* sont des signaux booléens émis par l'automate.
- Les *variables locales* servent aux calculs intermédiaires nécessaires à la définition des variables d'entrée et de sortie ainsi qu'à la définition de la fonction de transition.

L'*état initial* de l'automate est défini au moyen d'une expression logique portant sur ses variables d'état. La valeur de ces variables est telle que l'expression est vérifiée.

La fonction de transition est définie sous forme de fonctions de transitions partielles : pour chaque variable d'état *v*, une équation de la forme *v'* = <*expression*> définit la valeur que prend cette variable à l'état suivant. Ainsi, la fonction de transition est composée par l'ensemble de ces équations.

<automate> ::= *<déclarations> <initial> <équations>*
[<assertion>] [<invariant>]
<déclarations> ::= *<décl_état> [<décl_entrées>]*
[<décl_local>] [<décl_sorties>]
<décl_état> ::= ***state*** *<liste_idf>* ***;***
<décl_entrées> ::= ***input*** *<liste_idf>* ***;***
<décl_local> ::= ***localstate*** *<liste_idf>* ***;***
<décl_sorties> ::= ***output*** *<liste_idf>* ***;***
<liste_idf> ::= *<idf>* | *<liste_idf>* ***;*** *<idf>*
<equations> ::= ***transition*** *<liste_transitions>*
*[**definitions** <liste_définitions>]*
<liste_transitions>::=<transition> | *<liste_transitions> <transition>*
<transition> ::= *<idf>* ***'*** **=** *<expression>* ***;***
<liste_définitions>::=<définition> | *<liste_définitions> <définition>*
<définition> ::= *<idf>* **=** *<expression>* ***;***
<expression> ::= ***0*** | ***1*** | *<idf>* | ***(*** *<expression* ***)*** | ***not*** *<expression>* |
<expression> <op_bin> <expression> |
if *<expression>* ***then*** *<expression>* ***else*** *<expression>* ***fi***
<op_bin>::= ***or*** | ***and*** | ***xor*** | ***eq***
<initial> ::= ***initial*** *<expression>* ***;***
<assertion> ::= ***assertion*** *<expression>* ***;***
<invariant> ::= ***invariant*** *<expression>* ***;***

Figure A-1 : Syntaxe de l'outil BAC

Les variables de sortie de l'automate sont également définies au moyen d'équations ainsi que les variables locales éventuellement utilisées.

Nous remarquons enfin que la syntaxe proposée permet la définition d'une *assertion* et d'un *invariant* qui sont des expressions logiques portant sur les variables de l'automate. L'assertion spécifie une relation entre les variable qui est supposée invariante pendant toute exécution de l'automate tandis que l'invariant est une relation analogue mais dont la validité est à prouver.

La fonction de l'outil est de vérifier que l'invariant est toujours vérifié par cet automate (i.e. quelle que soit son exécution) sous l'hypothèse que l'assertion soit satisfaite. BAC propose également le calcul des états accessibles de l'automate. Le résultat de ce calcul est une expression logique portant sur les variables de l'auomate et qui correspond à la fonction caractéristique de l'ensemble des états accessibles.

TRANSFORMATION D'UN PROGRAMME LUSTRE EN UN AUTOMATE BOOLÉEN

Le passage d'un programme LUSTRE n'utilisant que des variables booléennes à un automate booléen décrit dans le format BAC est automatiquement réalisée à l'aide du programme *lus2ba* du laboratoire Vérimag. Les principes de cette transformation sont les suivants :

- A chaque variable d'entrée, de sortie et locale du programme est associée une variable de même type dans l'automate booléen.
- Une variable d'état de l'automate est associée à chaque expression du programme à laquelle est appliquée l'opérateur ***pre***.
- Les équations définissant les variables d'état, d'entrée et de sortie sont directement déduites du programme LUSTRE.
- La conjonction des assertions figurant dans le programme LUSTRE correspond à la partie *assertion* de l'automate booléen.
- Enfin, la première sortie du programme LUSTRE doit être booléenne. L'expression lui étant associée par l'équation la définissant est correspond à la partie *invariant* de l'automate booléen.

ANNEXE B

Spécification détaillée du logiciel de contrôle d'ascenseur

IMPLANTATION

```
const NBETAGES = 4;

-- ************************************************************
-- Nœud Ascenseur
-- C'est le nœud principal de l'application
-- Il calcule les signaux de sortie en fonction des signaux d'entrée envoyés
-- par l'environnement
-- ************************************************************

node Ascenseur(
    -- signaux de requêtes mises à l'intérieur de la cabine
    requete : bool^NBETAGES;

    -- signaux indiquant l'état des portes
    portes_ouvertes, portes_fermees : bool;

    -- signaux relatifs à la position de la cabine
    etage, dessus, dessous : bool^NBETAGES;

    -- signaux d'appel de l'ascenseur depuis les étages
    appel_montee, appel_descente : bool^NBETAGES)
returns (
    -- allumage des boutons à l'intérieur de la cabine
    allumer_interieur : bool^NBETAGES;
```

```
    -- commandes d'ouverture et de fermeture des portes
    ouvrir_portes, fermer_portes : bool;

    -- commandes du moteur
    monter, descendre, stop : bool;

    -- allumage des boutons d'appel des étages
    allumer_haut, allumer_bas : bool^NBETAGES);
var
    -- Indicateurs de la direction courante de la cabine
    -- (en train de monter ou de descendre)
    direction_haut, direction_bas : bool;
let
    -- appel du nœud concernant les boutons de la cabine
    allumer_interieur = BoutonsCabine(requete, etage, portes_ouvertes);

    -- appel du nœud concernant l'ouverture et la fermeture des portes
    (ouvrir_portes, fermer_portes) =
        PortesCabine(portes_ouvertes, portes_fermees,
                etage, allumer_haut, allumer_bas,
                allumer_interieur,
                direction_haut, direction_bas);

    -- appel du nœud calculant les commandes du moteur
    (descendre, monter, stop, direction_haut, direction_bas) =
        Moteur(portes_ouvertes, portes_fermees,
            etage, dessus, dessous,
            allumer_interieur, allumer_haut, allumer_bas,
            ouvrir_portes);

    -- appel du nœud concernant les boutons aux étages
    (allumer_haut, allumer_bas) =
        EtagesBoutons(portes_ouvertes, portes_fermees,
                etage,
                appel_montee, appel_descente,
                direction_haut, direction_bas);
tel

-- ***************************************************************
-- Nœud BoutonsCabine
-- Calcule la valeur du tableau des signaux allumer_interieur à partir de celle
-- des tableaux requete, etage.
-- Les signaux de sortie sont vrais à partir du moment où une requête valide
```

```
-- a été emise (bouton appuyé à un moment où l'ascenseur ne se trouve pas
-- déjà arrêté à l'étage demandé).
-- Il restent vrais tant que la cabine ne s'est pas arrêtée à l'étage demandé.
-- ***************************************************************

node BoutonsCabine(
    requete, etage: bool^NBETAGES;
    portes_ouvertes : bool;)
returns (
    allumer_interieur:bool^NBETAGES);
let
    allumer_interieur[0..NBETAGES-1] =
        not (etage[0..NBETAGES-1] and portes_ouvertes^(NBETAGES)) and
        requete[0..NBETAGES-1] ->
        not (etage[0..NBETAGES-1] and portes_ouvertes^(NBETAGES)) and
        (requete[0..NBETAGES-1] or pre allumer_interieur[0..NBETAGES-1]);
tel

-- ***************************************************************
-- Nœud PortesCabine
-- Une commande d'ouverture est envoyée quand :
--  a. l'ascenseur est arrêté à un étage
--  b. une requête ou appel sont en attente pour l'étage en question
--  c. les portes sont fermées
-- Une commande de fermeture est envoyée quand les portes sont ouvertes
-- ***************************************************************

node PortesCabine(
    portes_ouvertes, portes_fermees : bool;
    etage : bool^NBETAGES;
    allumer_haut, allumer_bas : bool^NBETAGES;
    allumer_interieur : bool^NBETAGES;
    direction_haut, direction_bas : bool;)
returns(
    ouvrir_portes, fermer_portes : bool);
var
    -- indique quand l'ascenseur est arrete à un etage
    -- et une requete est emise pour cet etage qui
    -- doit etre prise en compte immediatement.
    arrete_sollicite:bool^NBETAGES;
let
    arrete_sollicite[0] =
        etage[0] and
        ((allumer_haut[0] and pre direction_haut) or allumer_interieur[0]);
```

```
    arrete_sollicite[1..NBETAGES-2] =
        etage[1..NBETAGES-2] and
        ((allumer_haut[1..NBETAGES-2] and
            (pre direction_haut^(NBETAGES-2))) or
            (allumer_bas[1..NBETAGES-2] and
                (pre direction_bas^(NBETAGES-2))) or
            allumer_interieur[1..NBETAGES-2]);

    arrete_sollicite[NBETAGES-1] =
        etage[NBETAGES-1] and
            ((allumer_bas[NBETAGES-1] and pre direction_bas) or
                allumer_interieur[NBETAGES-1]);

    ouvrir_portes = portes_fermees and OR(NBETAGES, arrete_sollicite);

    fermer_portes = portes_ouvertes;
tel

-- ****************************************************************
-- Nœud EtagesBoutons
-- Enregistre les appels depuis les étages et calcule la valeur
-- des signaux allumant les boutons associes
-- ****************************************************************
node EtagesBoutons(
    portes_ouvertes, portes_fermees : bool;
    etage : bool^NBETAGES;
    appel_montee, appel_descente : bool^NBETAGES;
    direction_haut, direction_bas : bool;)
returns(
    allumer_haut, allumer_bas : bool^NBETAGES);
let
    allumer_haut[0] =
        not(etage[0] and portes_ouvertes) and appel_montee[0]->
        not(etage[0] and portes_ouvertes) and
            (appel_montee[0] or pre allumer_haut[0]);

    allumer_bas[0] = false;

    allumer_haut[1..NBETAGES-2] =
        not(etage[1..NBETAGES-2] and (portes_ouvertes^(NBETAGES-2))) and
        appel_montee[1..NBETAGES-2] ->
        not(etage[1..NBETAGES-2] and (portes_ouvertes^(NBETAGES-2)) and
        (pre direction_haut^(NBETAGES-2))) and
```

```
        (appel_montee[1..NBETAGES-2] or pre allumer_haut[1..NBETAGES-2]);

    allumer_bas[1..NBETAGES-2] =
        not(etage[1..NBETAGES-2] and (portes_ouvertes^(NBETAGES-2)))
        and appel_descente[1..NBETAGES-2] ->
        not(etage[1..NBETAGES-2] and (portes_ouvertes^(NBETAGES-2)) and
            (pre direction_bas^(NBETAGES-2))) and
        (appel_descente[1..NBETAGES-2] or pre allumer_bas[1..NBETAGES-2]);

    allumer_haut[NBETAGES-1] = false;

    allumer_bas[NBETAGES-1] =
        not(etage[NBETAGES-1] and portes_ouvertes) and
        appel_descente[NBETAGES-1] ->
        not(etage[NBETAGES-1] and portes_ouvertes) and
        (appel_descente[NBETAGES-1] or pre allumer_bas[NBETAGES-1]);
tel

-- ***************************************************************
-- Nœud Moteur
-- Envoie les commandes de déplacement de la cabine en fonction des
-- requêtes mises et la direction courante de la cabine.
-- ***************************************************************

node Moteur(
    portes_ouvertes, portes_fermees : bool;
    etage, dessus, dessous : bool^NBETAGES;
    allumer_interieur : bool^NBETAGES;
    allumer_haut, allumer_bas : bool^NBETAGES;
    ouvrir_portes : bool;)
returns(
    descendre, monter, stop : bool;
    direction_haut, direction_bas : bool);
var
    -- max_req (min_req) est l'étage maximum (minimum) où la cabine doit
    -- s'arrêter
    max_req, min_req : int;

    -- max_haut (min_bas) est l'étage maximum (minimum) pour lequel
    -- une requête de montée (descente) a été émise
    max_haut, min_bas : int;

    -- etage_courant est l'étage où se trouve la cabine
    etage_courant : int;
```

```
    -- mem_req contient l'ensemble des requêtes
    mem_req : bool^NBETAGES;

    -- mem_haut et mem_bas contiennent l'ensemble des  requêtes de montée
    -- et de descente
    mem_haut, mem_bas : bool^NBETAGES;

    -- pret_a_partir est vrai quand l'ascenseur est prêt à partir c'est à dire quand
    -- il est arrêté à un etage les portes fermées et aucun signal d'ouverture
    -- n'est envoyé.
    pret_a_partir : bool;
let
    direction_haut = true ->
        monter or
        (not descendre and ((etage_courant = min_req and
                                etage_courant <> min_bas) or
                            etage_courant = 0 or
                            min_req = -1 or
                            pre direction_haut)) ;

    direction_bas = true ->
        descendre or
        (not monter and ((etage_courant = max_req and
                                etage_courant <> max_haut) or
                           etage_courant = NBETAGES - 1 or
                           max_req = -1 or
                           pre direction_bas)) ;

    mem_req = allumer_interieur or allumer_haut or allumer_bas;
    mem_haut =  allumer_interieur or allumer_haut;
    mem_bas = allumer_interieur or allumer_bas;

    max_req = MAX_T(NBETAGES, mem_req);
    min_req = MIN_T(NBETAGES, mem_req);
    max_haut = MAX_T(NBETAGES, allumer_haut);
    min_bas = MIN_T(NBETAGES, allumer_bas);

    etage_courant = MAX_T(NBETAGES, etage);

    pret_a_partir =
        etage_courant <> -1 and portes_fermees and not ouvrir_portes;

    descendre = false ->
```

```
        not monter and pre direction_bas and pret_a_partir and
        min_req < etage_courant and min_req <> -1;

    monter = false->
        pre direction_haut and pret_a_partir and max_req > etage_courant and
        max_req <> -1;

    stop = false ->
        (pre direction_haut and (OR(NBETAGES, mem_haut and dessous)) or
            (mem_bas[NBETAGES-1] and dessous[NBETAGES-1])) or
        (pre direction_bas and (OR(NBETAGES, mem_bas and dessus)) or
            (mem_haut[0] and dessus[0]));
tel

-- ************************************************************
-- Nœuds récursifs
--
-- OR réalise la disjonction des éléments d'un tableau de booléens
-- tandis que AND en réalise la conjonction.
-- AuPlusUn est l'équivalent de l'opérateur #.
-- MAX_T retourne l'indice du dernier élément vrai du tableau
-- MIN_T retourne l'indice du premier élément vrai du tableau
-- ************************************************************

node OR(const n: int; A: bool^n) returns (LIN_OR: bool)
let
    LIN_OR = with n=1 then A[0]
        else A[0] or OR(n-1, A[1..n-1]);
tel

node AND(const n: int; A: bool^n) returns (LIN_AND: bool)
let
    LIN_AND = with n=1 then A[0]
         else A[0] and AND(n-1, A[1..n-1]);
tel

node AuPlusUn(const n: int; A: bool^n) returns (APU: bool)
let
    APU = with n=1 then true
         else ((A[0] and not OR(n-1, A[1..n-1])) or
          (not A[0] and AuPlusUn(n-1, A[1..n-1])));
tel

node MAX_T(const n : int; A : bool^n) returns (MAX :int)
```

```
let
    MAX = with n=1 then  if A[0] then 0
                         else -1
          else   if A[n-1] then n-1
                 else MAX_T(n-1, A[0..n-2]);
tel

node MIN_T(const n:int; A : bool^n) returns (MIN :int)
var B:bool^n;
   max:int;
let
    B[n-1..0] = A[0..n-1];
    max = MAX_T(n,B);
    MIN =   if max = -1 then -1
            else n - max -1;
tel
```

SPÉCIFICATION DE L'ENVIRONNEMENT

```
-- ************************************************************
-- Nœud de test Env
-- ************************************************************

testnode Env(
    allumer_interieur : bool^NBETAGES;
    ouvrir_portes, fermer_portes : bool;
    monter, descendre, stop : bool;
    allumer_haut, allumer_bas : bool^NBETAGES)
returns(
    requete : bool^NBETAGES;
    portes_ouvertes, portes_fermees : bool;
    etage, dessus, dessous : bool^NBETAGES;
    appel_montee, appel_descente : bool^NBETAGES );
let
    environment(
    -- PROPRIETES INSTANTANEES

    -- PROPRIETES SUR LES PORTES

    -- Les portes ne sont jamais ouvertes et fermées à la fois

     not (portes_fermees and portes_ouvertes) and

        -- PROPRIETES SUR LA POSITION DE LA CABINE
```

-- A l'instant initial la cabine est arrêtée à un étage

(OR(NBETAGES, etage) ***->*** ***true****)* ***and***

-- La cabine ne dépasse jamais les limites physiques

*(****not*** *(dessous[0]* ***or*** *dessus[NBETAGES-1]))* ***and***

-- La cabine est à un et un seul endroit à la fois

((AuPlusUn(NBETAGES, etage) ***and*** *AuPlusUn(NBETAGES, dessus)* ***and***
AuPlusUn(NBETAGES, dessous)) ***and***

#(OR(NBETAGES, etage), OR(NBETAGES, dessus),
OR(NBETAGES, dessous)) ***and***

(OR(NBETAGES, etage) ***or*** *OR(NBETAGES, dessus)* ***or***
OR(NBETAGES, dessous))) ***and***

-- PROPRIETES SUR LES BOUTONS DES ETAGES

-- Le bouton de demande de descente (montée) est inactif au RC (dernier étage)

*(****not*** *(appel_montee[NBETAGES-1]* ***or*** *appel_descente[0]))* ***and***

-- PROPRIETES TEMPORELLES

-- PROPRIETES SUR LES PORTES

-- Pas d'ouverture ou fermeture spontanée des portes

always_from_to(portes_fermees, portes_fermees, ***pre*** *ouvrir_portes)* ***and***
always_from_to(portes_ouvertes, portes_ouvertes, ***pre*** *fermer_portes)* ***and***

-- SPECIFICATION DU DEPLACEMENT DE LA CABINE

-- La cabine ne quitte pas spontanément un étage

(AND(NBETAGES, always_from_to(etage, etage,
pre *(monter* ***or*** *descendre)^(NBETAGES)))* ***and***

-- Continuité du mouvement : Si la cabine quitte une position
-- c'est pour passer à la position immédiatement au dessus

-- ou au dessous.
-- Ces propriétés spécifient également que quand la cabine
-- commence à monter (descendre) elle continue à monter (descendre)
-- jusqu'au prochain arrêt

AND(NBETAGES-2, always_from_to(etage[1..NBETAGES-2],
etage[1..NBETAGES-2],
dessus[0..NBETAGES-3] ***or*** *dessous[2..NBETAGES-1]))* ***and***

always_from_to(etage[0], etage[0], dessous[1]) ***and***

always_from_to(etage[NBETAGES-1], etage[NBETAGES-1],
dessus[NBETAGES-2]) ***and***

AND(NBETAGES-2, always_from_to(dessus[1..NBETAGES-2],
dessus[1..NBETAGES-2],
etage[1..NBETAGES-2] ***or*** *dessus[0..NBETAGES-3]))* ***and***

always_from_to(dessus[0], dessus[0], etage[0]) ***and***

AND(NBETAGES-2, always_from_to(dessous[1..NBETAGES-2],
dessous[1..NBETAGES-2],
etage[1..NBETAGES-2] ***or*** *dessous[2..NBETAGES-1]))* ***and***

always_from_to(dessous[NBETAGES-1], dessous[NBETAGES-1],
etage[NBETAGES-1]) ***and***

-- Cette propriété spécifie que si un stop a été émis la cabine
-- s'arrêtera au premier étage suivant sa position courante.

AND(NBETAGES-2,
*once_from_to(****pre****(etage[1..NBETAGES-2]),* ***pre****(stop^(NBETAGES-2)* ***and***
dessous[1..NBETAGES-2]), dessous[2..NBETAGES-1])) ***and***

AND(NBETAGES-2,
*once_from_to(****pre****(etage[1..NBETAGES-2]),* ***pre****(stop^(NBETAGES-2)* ***and***
dessus[1..NBETAGES-2]), dessus[0..NBETAGES-3])) ***and***

-- La cabine ne s'arrête pas spontanément :
-- elle ne s'arrête que suite à une commande stop

*AND(NBETAGES, always_from_to(****not*** *etage,* ***not*** *etage,*
pre *stop^(NBETAGES)))* ***and***

-- La cabine monte (descend) seulement après réception d'un signal monter
-- (descendre)

*AND(NBETAGES-1, once_from_to(**pre**(monter^(NBETAGES-1)),*
*   **pre** etage[0..NBETAGES-2], dessous[1..NBETAGES-1])) **and***

*AND(NBETAGES-1, once_from_to(**pre**(descendre^(NBETAGES-1)),*
*   **pre** etage[1..NBETAGES-1], dessus[0..NBETAGES-2]))*
);
***tel**;*

Table des matières

Résumé : Ce travail s'inscrit dans le cadre des méthodes formelles et des outils pour la spécification, la programmation, la vérification et la validation des logiciels réactifs à l'aide du langage synchrone LUSTRE. Nous avons étendu l'environnement de développement actuel de ces logiciels en lui adjoignant des outils de validation de spécifications et de test de programmes.

Nous proposons une technique de construction automatique d'un simulateur aléatoire de l'environnement externe du logiciel ainsi que d'un simulateur des comportements sûrs de ce dernier. Exécutés conjointement, ces deux simulateurs permettent, par observation, la validation des spécifications de l'environnement et des propriétés de sûreté. La simulation de l'environnement constitue également un moyen de test aléatoire du logiciel. Une variante de cette technique est le test des propriétés de sûreté. Ce type de test favorise l'exécution des comportements de l'environnement du logiciel qui peuvent mettre en évidence avec une plus grande probabilité des violations des propriétés de sûreté. Enfin, nous définissons des critères de couverture pour le test structurel en s'appuyant sur le réseau d'opérateurs associé à un programme LUSTRE. Les conditions d'exécution des composants du réseau satisfaisant les critères sont calculées automatiquement ce qui permet la génération automatique de données de test ainsi que la mesure du taux de couverture obtenu.

Toutes ces techniques sont formellement définies. Une illustration de leur application sur un exemple de logiciel synchrone que nous avons développé à cet effet complète notre étude.

Mots-clés : Logiciels synchrones, test, vérification et validation de logiciels.

Abstract : The specification, implementation, verification and validation of reactive software by means of the LUSTRE synchronous language is performed in a formal framework. This work aims at enhancing the present development environment by adding new tools which allow us to validate formal specifications and to test the software.

We have devised a technique for randomly simulating both the software environment and the software behaviors preserving the safety properties. The concurrent execution of these simulators make possible the validation of the software specifications. The environment simulation provides a means to randomly test the software. Moreover, this technique can be improved in order to force the simulation to preferably generate test data revealing safety property violations. Finally, we have defined structure-based coverage criteria on the operator net associated with a LUSTRE program. The execution conditions of the net components satisfying a criterion are automatically computed. This allows us to automatically generate test data and to assess the effective coverage.

All these techniques are formally defined. Their use is illustrated on a software example specifically developed for this study.

Keywords : Synchronous software, software test, verification and validation.