HAL
archives-ouvertes.fr

# Test de logiciels synchrones avec la PLC

Besnik Seljimi

▶ **To cite this version:**

Besnik Seljimi. Test de logiciels synchrones avec la PLC. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2009. Français. <tel-00408225>

**HAL Id: tel-00408225**
**https://tel.archives-ouvertes.fr/tel-00408225**

Submitted on 29 Jul 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.


Université Joseph Fourier - Grenoble 1

# Test de logiciels synchrones avec la PLC

## Thèse

*présentée par*

**Besnik SELJIMI**

*en vue de l'obtention du grade de*

Docteur de l'Université Joseph Fourier Grenoble 1

Discipline Informatique

*soutenue le 02 juillet 2009 devant le jury composé de :*

| | | |
|---|---|---|
| Jean-Claude Fernandez | Prof. Université de Grenoble | **Président** |
| Michel Rueher | Prof. Université de Nice | **Rapporteur** |
| Fabrice Bouquet | Prof. Université de Besançon | **Rapporteur** |
| Bernard Botella | Ingénieur CEA Saclay | **Examinateur** |
| Laurent Trilling | Prof. Université de Grenoble | **Examinateur** |
| Ioannis Parissis | Prof. Institut Polytechnique de Grenoble | **Examinateur** |

# Test de logiciels synchrones avec la PLC

## Résumé

Ce travail porte sur le test fonctionnel, basé sur les spécifications et complètement automatisé des logiciels synchrones. Nous proposons une extension des techniques de test proposées par l'outil LUTESS afin de prendre en compte des logiciels qui comportent des entrées/sorties numériques. La génération de données de test est abordée en s'appuyant sur les techniques de programmation par contraintes.

Nous avons redéfini les méthodes de guidage de la génération afin de les adapter à ce nouveau contexte numérique. Ainsi, nous proposons, en plus de la génération aléatoire respectant les propriétés invariantes de l'environnement, le guidage du test basé sur des probabilités conditionnelles ou sur des propriétés de sûreté. Des connaissances partielles sur le logiciel, que nous appelons hypothèses de test, peuvent être intégrées dans le processus de génération et contribuer à l'amélioration du pouvoir de détection de fautes du guidage par propriétés de sûreté. Enfin, nous permettons l'utilisation conjointe de plusieurs techniques de guidage dans une même spécification.

Une implémentation de ces méthodes de test a été réalisée dans une nouvelle version de l'outil, que nous appelons LUTESS V2. L'applicabilité de ces méthodes dans un contexte plus réaliste a été évaluée sur une étude de cas significative d'un contrôleur de niveau d'eau dans une chaudière.

# Testing synchronous software with CLP


## Abstract

This work deals with functional, specification-based and fully automated testing of synchronous software. We propose an extension of the testing techniques proposed by the LUTESS tool in order to consider programs with numerical inputs/outputs. The test data generation is now based on constraint programming techniques.

We have redefined the generation methods in order to adapt them in this new context. Thus, we propose, in addition to the random generation with respect to the invariant properties of the environment, test guidance based on conditional probabilities or safety properties. Partial knowledge about the software, called test hypotheses, can be integrated in the generation process and improve the fault detection ability of safety property guided testing. Finally, we make it possible to use simultaneously several guidance techniques in the same specification.

An implementation of these testing methods has resulted in a new version of the tool, called LUTESS V2. The applicability of these methods for testing more realistic programs has been evaluated on a significant case study of a water level controller in a steam boiler.

# Remerciements

Cette thèse a débuté au sein du Laboratoire Logiciels, Systèmes et Réseaux et s'est poursuivie au sein du Laboratoire d'Informatique de Grenoble. Je tiens à exprimer ma reconnaissance à leurs directeurs respectifs, Farid Ouabdesselam et Brigitte Plateau, ainsi que toute l'équipe administrative pour m'avoir accueilli et m'avoir fourni des conditions de travail exceptionnelles.

Je tiens à exprimer mes sincères remerciements à mes deux responsables Ioannis Parissis et Laurent Trilling de m'avoir dirigé, guidé et conseillé ainsi que pour avoir fait preuve de tant de confiance, de patience, d'amitié et de bienveillance durant ces années de thèse.

Je remercie également Michel Rueher et Fabrice Bouquet pour avoir accepté de juger ce travail, Jean-Claude Fernandez qui m'a fait l'honneur de présider ce jury et Bernard Botella pour y avoir apporté ses compétences et son expérience.

Je souhaite également remercier tous les membres du projet RNTL DANOCOPS pour les échanges fructueux lors des réunions de travail.

Merci aussi à tous mes collègues et amis de longue date du laboratoire qui se reconnaitront ici. Je remercie tout particulièrement Abdesselam, Laya, Virginia et Amal pour leur amitié et pour toutes les discussions que nous avons eu durant ces années de thèse.

Pour finir, je remercie mes parents ainsi que toute ma famille pour toute leur affection et leur soutien incoditionnel dans mes choix de carrière.

*à toute ma famille*

# Table des matières

# Table des figures

# Liste des tableaux

# Liste des algorithmes

# Chapitre 1

# Introduction

Cette thèse porte sur le test fonctionnel, basé sur des spécifications et complètement automatisé des logiciels synchrones. Nous présentons une méthode de construction de générateurs de données de test à partir de spécifications formelles. Le paradigme de la programmation par contraintes est utilisé pour modéliser ces générateurs. Les données de test satisfaisant les spécifications résultent des solutions de problèmes à contraintes.

Ce chapitre introductif commence par présenter brièvement le contexte actuel du test des logiciels synchrones, pour continuer par une présentation des motivations et des contributions de notre travail.

## 1.1 Contexte de recherches

Aujourd'hui les logiciels sont présents dans des domaines aussi variés que sensibles de la vie quotidienne. Ils sont de plus en plus complexes, tant au niveau de leur taille que des fonctionnalités qu'ils offrent. S'assurer que le logiciel réalise correctement les services pour lesquels il a été conçu devient aussi de plus en plus difficile. Si les pannes dues à une défaillance logicielle sont devenues habituelles dans nos ordinateurs personnels, elles peuvent provoquer de réelles catastrophes s'il s'agit de systèmes qui mettent en jeu des vies humaines ou d'importantes sommes d'argent. Ces systèmes, dits *critiques*, nécessitent une confiance accrue dans les services qu'ils réalisent.

Pour permettre aux utilisateurs d'avoir la confiance nécessaire dans un logiciel, il faut s'assurer que les risques d'une panne sévère sont limités (idéalement, nuls). Les

différents moyens de *vérification et validation* ont pour but la maîtrise de ce risque, qui par ailleurs n'est pas une tâche facile si ces moyens ne sont pas clairement identifiés et définis formellement. Selon Boehm [8], l'activité de vérification est 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".

Des visions très tranchées de la problématique de vérification et validation entre le monde industriel et académique ont par le passé donné naissance à deux approches distinctes, l'une fondée sur la *preuve* mathématique et l'autre sur le *test* des logiciels. Contrairement à la preuve qui consiste à démontrer mathématiquement la correction d'un logiciel, le test vise à découvrir des défauts dans son implantation [31]. Les industriels, en pratique, utilisaient le plus souvent le test, alors que la recherche ce concentrait sur la vérification formelle. La recherche en génie logiciel tente de rapprocher ces deux approches qui aujourd'hui sont considérées comme complémentaires et partagent au moins la nécessité de fondements théoriques.

## Test des logiciels synchrones

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. En pratique, il existe un très grand nombre, voir une infinité, de valeurs possibles pour les entrées et leur exécution dans la totalité n'est simplement pas envisageable. Le choix de données de test pertinentes est donc crucial pour augmenter la capacité du test à révéler des fautes. Les méthodes de test se spécialisent pour rendre ce choix plus pertinent en fonction des types de fautes recherchées et des particularités des logiciels testés.

L'automatisation des différentes étapes de test permet de diminuer l'effort nécessaire et d'augmenter ainsi considérablement le nombre de tests pouvant être effectués. L'intérêt principal de l'utilisation de spécifications formelles dans le test est de permettre une meilleure automatisation, que ce soit pendant la phase de génération de données ou pour décider de la réussite du test. Elles permettent également de

réduire l'intervention du facteur humain et les erreurs qu'il peut induire involontairement.

Nous nous intéressons en particulier aux logiciels réactifs synchrones [9, 10, 26] qui, par la nature de leur utilisation dans des systèmes critiques, requièrent plus de vérification et de validation. Un programme réactif 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 hypothèse, dite de *synchronisme*, caractérise l'approche synchrone qui a connu un grand essor depuis plus d'une vingtaine d'années, en particulier grâce à des langages dédiés comme ESTEREL [5], SIGNAL [26] et LUSTRE [10, 19]. Ces langages sont particulièrement bien adaptés à la description de logiciels dans les systèmes du type contrôle/commande où le logiciel doit agir sur son environnement pour éviter tout dysfonctionnement susceptible de provoquer des catastrophes. Ce type de logiciels est utilisé dans les parties critiques des applications dans des domaines comme l'aéronautique, l'énergie et les transports.

LUSTRE [10] est un langage déclaratif à flots de données synchrone munis d'opérateurs temporels permettant de se référer à des valeurs passées des données. Il a été conçu pour la spécification et la programmation des logiciels réactifs synchrones dont le comportement est cyclique. Un flot de données est une suite finie ou infinie de valeurs associées à des instants discrets d'une horloge globale. Un programme LUSTRE décrit des fonctions entre des flots en sortie et des flots en entrée.

Les travaux présentés ici, proposent une extension de LUTESS [14], un outil de génération de données de test fonctionnelles pour les logiciels synchrones. Ils se sont déroulés dans le cadre du projet RNTL DANOCOPS (Détection Automatique de NOn-COnformités entre un Programme et sa Spécification), regroupant des partenaires universitaires (Laboratoire d'Informatique de Grenoble, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis et le Laboratoire d'Informatique de Franche-Comté) et industriels (THALES Systèmes Aéroportés et AxLog) autour d'une problématique commune : l'utilisation de la Programmation par Contraintes pour la vérification des logiciels.

## 1.2 Problèmes et motivations

L'utilisation du langage LUSTRE et de son environnement SCADE dans des applications nécessitant un niveau élevé de criticité fait que les activités de vérification et de validation revêtent un intérêt particulier dans le processus de développement. Les phases de vérification et validation sont donc très développées et leur outillage est indispensable. Elles ont été abordées dans le passé principalement au moyen de la preuve formelle basée sur le "model-checking" [20] ainsi que par le biais du test logiciel en particulier bien fondé mathématiquement [14, 28, 32, 36, 47].

Les travaux sur le test des logiciels synchrones ayant abouti au développement de l'environnement LUTESS [12, 37] avaient pour principale motivation la volonté d'apporter un moyen de vérification complémentaire à la preuve formelle par "model-checking" [20]. Cette dernière se limitant essentiellement à la vérification de propriétés définies sur des signaux booléens du programme, cette même limitation s'est retrouvée dans les outils de test développés. La possibilité de prendre en compte des spécifications incluant des relations entre valeurs et expressions numériques a toujours été considérée comme une extension indispensable, les logiciels synchrones dans la vie réelle ne se limitant pas à des entrées/sorties booléennes.

La problématique de ce travail s'inscrit dans la continuité des travaux menés autour du test des logiciels synchrones [14, 32, 36, 55, 62]. Elle consiste en la génération de données de test pour des spécifications de logiciels synchrones ayant des entrées/sorties numériques. Nous abordons ici cette extension en s'appuyant sur la Programmation Logique avec Contraintes (PLC) [23], souvent suggérée à des fins de génération de tests en particulier dans le cas des systèmes réactifs [43]. Ce choix est motivé par la richesse des environnements de résolution de contraintes qui permet d'envisager cette extension de manière simple et efficace sans avoir à développer une solution "ad hoc".

Cette nouvelle représentation nécessite aussi l'adaptation et la redéfinition formelle des différentes techniques de test développées autour de l'outil LUTESS [14, 36, 55, 62] de deux points de vue distincts. D'une part, l'utilisation de contraintes de nature numérique pose en des termes nouveaux le problème de la sélection de données d'entrée pertinentes. En particulier, la sélection de données de manière équitable,

assurée dans le cadre de la version booléenne de LUTESS par une étude exhaustive de l'arbre décrivant l'ensemble des solutions, ne peut être assurée de la même manière. De plus, « l'équité » est un concept qui, dans le cadre numérique, appelle à une autre définition. D'autre part, la traduction en contraintes des spécifications LUSTRE enrichies utilisées par les différentes stratégies de LUTESS nécessite une définition précise de la transformation à effectuer et la démonstration du principe de génération de données dans ce nouveau modèle de calcul.

## 1.3 Contributions de la thèse

L'objectif principal de cette thèse est donc la prise en compte de logiciels synchrones numériques dans l'approche de test fonctionnel proposée par LUTESS. Nous proposons, en conséquence, une méthode permettant la construction automatique de générateurs de test à partir de spécifications formelles. Ces spécifications sont réalisées dans un langage de spécification basé sur LUSTRE et contenant un ensemble d'opérateurs spécifiques au test. Cette spécification est ensuite modélisée sous la forme d'une machine à états finis. Les relations associées à cette machine sont représentées au moyen de contraintes. Les algorithmes de génération se basent sur les principes de la résolution de problèmes à contraintes pour affecter des valeurs aux entrées du logiciel sous test.

Sur le plan théorique, nous proposons :

- La génération de données de test fonctionnel respectant des invariants d'environnement qui incluent, en plus de la logique booléenne, des variables entières et des relations entre ces variables.
- Le guidage de la génération vers la violation des propriétés de sûreté dans ce nouveau contexte. Elle consiste à rechercher, au voisinage d'un état atteint lors de l'exécution, les états susceptibles de violer la propriété et générer des entrées qui mènent vers un tel état.
- L'introduction dans le processus de test de connaissances sur le logiciel, même partielles, sous la forme d'hypothèses de test. Elles peuvent améliorer le guidage par les propriétés de sûreté, en introduisant un lien entre les entrées et les sorties du logiciel. De plus, si le programme entier est introduit en hypothèse,

ce guidage s'approche d'une preuve locale. En effet, si l'algorithme de résolution termine dans une situation donnée, on peut soit conclure que la propriété ne peut pas être violée dans le voisinage de l'état courant, soit qu'elle peut être violée et les entrées choisies mènent forcement vers une telle situation.

- La possibilité d'utiliser des expressions quelconques dans la génération guidée par les probabilités conditionnelles, contrairement à la précédente version de l'outil qui ne permet que les variables d'entrées booléennes du logiciel sous test. Cette augmentation d'expressivité permet la définition de profils d'utilisation évolués du logiciel sous test.
- L'utilisation conjointe de l'ensemble des opérateurs de test dans une même spécification. En effet, dans la version booléenne de l'outil, une seule technique de guidage était autorisée. Le pouvoir d'expression du testeur se trouve ainsi considérablement amélioré.

Sur le plan pratique, nous avons implémenté ces techniques de génération dans la nouvelle version de l'outil LUTESS. Nous illustrons par une étude de cas réaliste, l'exemple bien connu d'un contrôleur du niveau d'eau dans une chaudière, l'utilisation de l'outil pour générer des données de test. Nous proposons aussi, sur le plan de la méthodologie de test, des idées sur le processus de modélisation, applicables dans un cadre plus général.

## 1.4 Structure du manuscrit

Ce manuscrit s'articule autour de trois parties principales.

La **première partie**, constitué des chapitres 2 et 3, est dédiée à la présentation de l'état de l'art dans le domaine du test des logiciels synchrones :

- Le chapitre 2 présente l'état de l'art du test des logiciels informatiques et en particulier celui des logiciels synchrones, ainsi que les principes de la programmation par contraintes.
- Le chapitre 3 est dédié à une présentation de l'outil LUTESS et des techniques de test qui lui sont associées.

Dans la **deuxième partie**, qui se compose des chapitres 4 et 5, nous exposons nos travaux permettant de générer des données de test pour des logiciels synchrones

numériques :

- Le chapitre 4 introduit les différentes techniques de test que nous proposons. Il présente la méthode de test ainsi que la sémantique informelle des différents opérateurs de test.
- Le chapitre 5 est consacré à la définition formelle de la génération de données de test en utilisant la programmation par contraintes.

Les chapitres 6 et 7 de la **troisième partie**, présentent la mise en oeuvre de ces méthodes de test ainsi que leur évaluation :

- Le chapitre 6 présente les aspects d'implémentation de cette génération dans Lutess V2.
- Le chapitre 7 consiste en une expérimentation de l'outil sur une étude de cas.

Enfin, le chapitre 8 conclut ce document par un bilan ainsi que des perspectives et questions ouvertes de ce travail.

# Première partie

# Test de logiciels synchrones

# Chapitre 2

# Test des logiciels synchrones

Dans ce chapitre, nous abordons les domaines dans lesquels se situent ces travaux de thèse, celui du test logiciel, particulièrement dans le cadre des logiciels synchrones et celui de la programmation par contraintes.

Nous commençons par un exposé synthétique, dans la section 2.1, des principaux travaux théoriques et empiriques sur le test. Ensuite, nous présentons les particularités des logiciels synchrones ainsi que les travaux spécifiques sur le test de ce type de logiciels dans la section 2.2. Enfin, puisque nous nous appuyons sur la programmation par contraintes pour générer des données de test, la section 2.3 présente les caractéristiques de ce paradigme de programmation.

## 2.1 Le test des logiciels

La qualité d'un logiciel est un ensemble de propriétés et caractéristiques d'un produit ou service qui lui confèrent l'aptitude à satisfaire des besoins exprimés ou implicites (ISO 8402) et la sûreté de fonctionnement [25], est une de ces propriétés les plus importantes. Le sûreté de fonctionnement (la crédibilité selon [39]) exprime la confiance qu'on peut apporter sur le service délivré par un logiciel. L'établissement de cette confiance se fait par la vérification et la validation de l'exactitude du logiciel, entre autres, par un processus de recherche et d'élimination des *fautes*. Une faute est définie comme étant à l'origine d'une *erreur* [25], où celle-ci représente un état du logiciel susceptible de provoquer une *défaillance*. Les moyens proposées pour l'élimination des fautes vont de la preuve formelle au test logiciel.

Par opposition aux moyens statiques comme l'inspection et les revues de code

[31], le test logiciel est une activité dynamique de vérification et de validation de programmes. Le *test pour la vérification* suppose l'existence d'une spécification explicite et complète et vise à montrer que l'implémentation lui est conforme. Par exemple, le test de conformité [53] dans le domaine des protocoles est un moyen de vérification. Son but est de s'assurer qu'une implantation d'un protocole est conforme aux exigences formulées dans sa spécification. Le *test pour la validation*, en revanche, repose sur une spécification partielle voire implicite et vise, tout autant, à provoquer des défaillances qu'à conclure que le programme se comporte bien conformément à ce qu'on attend de lui.

Selon la définition de l'IEEE, le test est "*l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus*". Cette définition met en évidence deux objectifs du test : celui de montrer que le logiciel remplit bien ses fonctions et celui de recherche de fautes. Seulement le deuxième de ces objectifs est relevé par Myers [31], qui définit le test comme *"l'activité d'exécution d'un programme dans l'intention d'y trouver des erreurs*[1]*"*. La différence d'approche entre ces deux définitions se retrouve aussi dans la notion de réussite ou échec de l'exécution d'une donnée de test. Selon la définition de l'IEEE, l'exécution sera réussie si le résultat est conforme à la spécification tandis que, selon Myers, elle est réussie seulement si une erreur est mise en évidence.

### 2.1.1 Processus de test

Le test consiste donc à exécuter un logiciel en lui fournissant des données en entrée ainsi qu'à décider si sa réaction est celle attendue. Cette définition, ainsi que la nécessité de décider quand l'activité de test est complétée, mettent en évidence quatre étapes dans un processus de test :

#### Sélection

L'étape de sélection de données de test consiste à choisir des valeurs pour les entrées du logiciel. Nous considérons, dans le cadre de ce travail, qu'une *donnée de*

1. Testing is the process of executing a program with the intent of finding errors

*test* est une instance du produit cartésien des domaines des paramètres d'entrée du programme et une *séquence de test* est une suite de données de test consécutives.

La constitution manuelle de données de test efficaces s'avère être une activité très difficile, coûtante en temps et dépendante de l'erreur humaine. La sélection automatique réduit les efforts du testeur, qui devront se porter non plus sur la conception des données de test, mais sur la description des objectifs du test. Le terme de *génération* est utilisé pour parler de la sélection automatique de données de test.

#### Exécution

Cette étape consiste à exécuter le programme sous test, en lui soumettant les données de test choisies pendant l'étape de sélection. Le test est qualifié de *statique* si l'ensemble des données de test est constitué avant l'exécution du programme. Si, au contraire, les données sont constituées au fur et à mesure de l'exécution du programme, le test est qualifié de *dynamique*.

#### Verdict

Après l'exécution du programme avec des données de test, il est nécessaire de décider de la réussite ou non du test en comparant les résultats obtenus avec ceux attendus. La distinction entre un comportement erroné et un comportement correct est du rôle de l'*oracle*. L'oracle peut être produit manuellement en faisant correspondre les sorties aux entrées ou obtenu automatiquement à partir d'une spécification.

#### Évaluation

La décision d'*arrêt du test* nécessite l'évaluation de la qualité du test effectué, une notion fortement liée à l'objectif fixé pour le test.

### 2.1.2 Sélection des données de test

Idéalement, pour s'assurer du "bon fonctionnement" d'un logiciel dans tous les cas, il faudrait tester tous les comportements possibles de ce dernier. La notion d'*exhaustivité* suppose que le logiciel est complètement déterminé par les entrées qui lui sont fournies, et revient à exécuter le logiciel avec l'ensemble des entrées

possibles. Cette notion est alors synonyme d'une qualité maximale du jeu de test et l'intuition veut que plus le nombre de tests effectuées augmente plus la qualité du test augmente.

Mais généralement, le test exhaustif n'est pas réalisable en pratique car le nombre d'entrées possibles peut être très grand, sinon infini. Par exemple, un logiciel comportant comme entrées seulement deux entiers 32 bits devrait s'exécuter avec $(2^{32})^2 \approx 1.8 \times 10^{19}$ données de test différentes, alors qu'il existe une infinité de programmes qui peuvent être soumis à un compilateur.

L'un des enjeux du test logiciel est de maximiser la qualité du test avec un nombre réduit, voire minimal de données de test. Les testeurs souvent se basent sur leur intuition et l'expérience pour construire un jeu de tests fini et abordable. La formalisation de ces pratiques s'appuie sur les *hypothèses de réduction*, formulées de manière à garantir la préservation de la pertinence d'un jeu de tests. D'autres hypothèses font un lien entre un ensemble de données de test et la classe de fautes qu'il est susceptible de détecter. Par exemple, l'exécution de toutes les instructions d'un programme est supposée garantir que toutes les fautes relatives aux instructions soient détectées.

L'exigence d'exécution de toutes les instructions du programme ou l'obligation de révéler certaines classe de fautes par un jeu de tests constituent *des critères* de test.

### 2.1.3 Critères de test

Un critère est un ensemble de propriétés et de conditions qu'un ensemble de données de test doit satisfaire pour atteindre un objectif. Selon que ces propriétés et conditions portent sur la sélection ou l'évaluation des données de test, un critère peut être :

- un *critère de sélection*, permettant de construire, *a priori*, les données de test ou
- un *critère d'adéquation*, permettant d'évaluer, *a posteriori*, la qualité des données de tests ; il est appelé *critère d'arrêt* lorsqu'il est utilisé pour décider de l'arrêt du test.

Un critère de sélection de données de test peut être défini à partir d'une information venant du programme ou d'une spécification de celui-ci. Quand les données de test sont constituées à partir d'un modèle qui est une abstraction du programme sous test, par exemple le graphe de contrôle ou de données, on parle de *test structurel* ou "boîte blanche". Dans le test boîte blanche, la structure interne (code) du programme est connue et utilisé pour définir un critère de sélection. A l'opposé, on parle de *test fonctionnel* ou "boîte noire" quand la structure interne est inconnue ou ignorée pour la sélection. Les données de test sont alors obtenues à partir de spécifications précisant le domaine d'entrée, les fonctions accomplies et/ou des propriétés du programme sous test.

Les travaux présentés dans [56, 57] portent sur la formalisation des différents critères structurels, basé sur le flot de contrôle d'un programme, en utilisant la notation Z. La formalisation inclue les critères bien connus [31] qui sont basés sur la couverture des instructions, décisions, conditions et leurs combinaisons ainsi que d'autres, moins connus, comme la couverture des prédicats. Cette formalisation des critères est importante pour enlever les ambiguïtés liées à une description informelle et permet, entre autres, de définir précisément l'inclusion d'un critère dans un autre. Ces relations d'inclusion peuvent être utilisées comme une mesure quantitative de la qualité du test effectué avant de décider de l'arrêt du test et facilitent l'introduction de nouveaux critères.

### 2.1.4 Test "boîte noire"

Dans le test boîte noire, la structure interne du programme est inconnue ou ignorée pendant la sélection. Les données de test sont alors choisies par tirage aléatoire ou alors en se basant sur des spécifications du logiciel.

Le *test aléatoire* [22] est une technique de test consistant à sélectionner les données de test par tirage aléatoire sur le produit cartésien des domaines de définition des entrées du programme sous test. Si les usages futurs du logiciels sont inconnus, ce tirage est réalisé selon une distribution uniforme. Autrement, dans le *test statistique*, le tirage est effectué en se basant sur un profil opérationnel [30]. Un profil opérationnel caractérise quantitativement les usages possibles du logiciel en définissant une distribution opérationnelle statistique du domaine des entrées. Le but principal du

test statistique est de s'assurer que les fonctionnalités les plus testées correspondent à celles qui sont les plus utilisées.

Une décomposition plus fine que le test purement aléatoire des domaines de définition des entrées du programme caractérise la technique des *catégories et partitions* [17]. La décomposition est faite par partitionnement en classes d'équivalence, selon la capacité des entrées à détecter des fautes.

La technique des *valeurs aux limites* [31] peut être considéré comme une variante de la technique précédente en considérant, non seulement les valeurs à l'intérieur de la classe d'équivalence, mais aussi aux limites de chaque classe. Elle vise en particulier, la détection d'une classe de fautes généralement introduites par l'utilisation erronée des opérateurs de comparaison.

### 2.1.5 Test à partir de spécifications

Généralement, les spécifications définissent le domaine d'entrée du logiciel ainsi que des relations liant les entrées et les sorties et déterminent les comportements possibles ou attendues du logiciel. Les techniques de test à partir des spécifications se basent sur un critère de sélection de données sur ces dernières. Ensuite, des données de test sont générées plus ou moins automatiquement pour satisfaire le critère de sélection défini sur la spécification.

#### 2.1.5.1 Test à partir de spécifications algébriques

Les travaux sur le test à partir de spécifications algébriques [6], visent à produire des données de test équivalentes au test exhaustif. Les auteurs proposent d'introduire le concept de contexte de test incluant des hypothèses sur le logiciel sous test, l'ensemble des données de test et un oracle. L'idée est de réduire ce contexte par raffinements successifs en gardant des données de test *non biaisés* (ne rejetant pas de programme correct) et *valides* (n'acceptant que des programmes corrects).

Les hypothèses de réduction, qui permettent de réduire l'ensemble de données de test sélectionnées, se divisent en deux catégories :

- Les hypothèses de *régularité* stipulent que les comportements partageant de grandes séquences d'instructions sont similaires. Ainsi, il est inutile de tester tous les comportements incluant une même boucle, il suffit d'exécuter quelques

itérations de cette boucle.

- Les hypothèses d'*uniformité* admettent que si le logiciel se comporte correctement pour un élément d'un ensemble, il se comportera également correctement pour tous les autres éléments.

Dans le prolongement de ces travaux, les spécifications algébriques bornées [3] sont utilisées pour générer automatiquement des données de test aux limites. Ces spécifications sont constituées de deux parties :

- la "partie idéale" permet de définir les structures de données et les opérateurs associés à l'aide des spécifications algébriques telles qu'elles sont utilisées dans [6] et
- la "partie bornée" permet de définir les bornes pour les structures de données définies dans la partie idéale.

#### 2.1.5.2 Test à partir de spécifications en B

Les langages de spécification comme B ou Z décrivent les opérations d'un logiciel par l'intermédiaire de *pre* et *post* conditions. Une méthode de génération de données de test aux limites, basée sur les spécifications formelles en B, a été proposée dans [27]. Les spécifications B sont utilisées comme une spécification pour le test et non dans le cadre du développement du logiciel.

Les spécifications sont traduites en différents ensembles de contraintes, permettant de représenter l'évolution de la machine B par un automate dont l'état est constitué des variables de la machine B [44]. Le domaine des variables est partitionné en fonction des prédicats qui les définissent : le domaine des variables entrant dans l'expression de la condition d'une construction IF-THEN-ELSE est partitionné en fonction du fait qu'elles rendent vraie ou fausse cette condition. Cette partition permet la définition d'un ensemble de comportements à couvrir.

La stratégie de couverture fonctionnelle est complétée par la définition d'une génération de données de test aux bornes de chaque partition créée précédemment. Un état de la machine est considéré aux limites si au moins une des variables le constituant est évaluée avec une valeur limite. L'approche proposée dans [44] se décompose en trois parties :

- construction pour chaque état aux limites $q$ d'une séquence de données de test

permettant de mener la machine B dans l'état $q$.

- une fois que la machine se trouve dans $q$, le test d'une opération est effectué par l'appel de cette dernière.
- les résultats obtenus sur la spécification sont comparés aux résultats obtenus sur l'implémentation du système.

#### 2.1.5.3 Test à partir de spécifications booléennes

Dans [60], les auteurs utilisent une spécification booléenne liant les entrées aux sorties du logiciel et proposent différentes stratégies de génération automatique de données de test à partir de cette spécification. Le principe de base de ces stratégies est de sélectionner des données de test de telle sorte que la modification de la valeur d'une seule variable entraîne la modification de la valeur de vérité de la formule booléenne utilisée comme spécification.

Une spécification booléenne peut aussi se présenter sous la forme d'un graphe causes-effets [16,31]. Ces graphes associent les causes (conditions d'entrée) aux effets (conditions de sortie). Un noeud est associé à chaque cause et chaque effet et les noeuds sont reliés entre eux par un des quatre opérateurs (*identité*, *non*, *et*, *ou*). Il est possible d'utiliser des conditions intermédiaires entre les causes et les effets en associant également un noeud à chaque condition. De plus, il est possible de spécifier des contraintes sur les causes (par exemple deux causes $a$ et $b$ ne peuvent se produire en même temps).

Un outil pour la génération automatique de données de test à partir de graphes causes-effets a été proposée dans [4]. Ces graphes sont construits à partir d'un langage de spécification et les données de test sont ensuite dérivées en trois étapes :

- l'ensemble des combinaisons d'entrées possibles est déterminée par un parcours du graphe ;
- les entrées sont regroupées en classe d'équivalence selon la partie du graphe causses-effets activée et
- seule une donnée de test est choisie dans chaque classe d'équivalence.

Les graphes causes-effets sont également utilisées pour représenter les spécifications de test dans les travaux présentés dans [52, 58, 59]. Les auteurs commencent par convertir les graphes causes-effets en formules booléennes. Chaque formule boo-

léenne est construite à partir de plusieurs prédicats reliés les uns aux autres par les opérateurs booléens *and* et *or*. Un prédicat peut être soit une variable booléenne, soit une expression relationnelle. Il définissent ensuite différentes classes de fautes possibles sur les formules booléennes et les critères que doit respecter un ensemble de données de test afin qu'il soit efficace pour la détection de la classe de fautes considérée. Plusieurs classes de fautes sont caractérisées dans ces travaux, parmi lesquelles les auteurs s'intéressent particulièrement au trois classes de fautes sur les formules booléennes :

- les *fautes sur les opérateurs booléens* considèrent une mauvaise utilisation des opérateurs booléens, par exemple le remplacement d'un *and* par un *or* ou une utilisation abusive de la négation ;
- les *fautes sur les opérateurs de relation* considèrent une mauvaise utilisation des opérateurs de comparaison ;
- les *fautes de parenthèse* considèrent les parenthèses mal placées ou manquantes.

Une autre approche, présentée dans [61], spécifie les comportements du logiciel sous test par un automate déterministe. La génération de données de test est effectuée en deux phases consistant à :

- amener le logiciel sous test dans un état cible de la spécification
- tester dans cet état la validité d'une propriété contre la spécification

L'algorithme proposé dans ces travaux permet de construire une séquence d'entrées menant le logiciel sous test dans un état de l'automate déterministe.

L'ensemble des travaux de sélection de données de test à partir de spécifications que nous avons présentés, montrent que l'aptitude à détecter des fautes est augmenté significativement par rapport à la génération aléatoire lorsque un critère de sélection est appliqué. Cependant, les mesures d'efficacité de ces critères sont liées au type de fautes recherchées. En conséquence, il est nécessaire de bien identifier les types d'erreurs afin d'appliquer le critère adéquat. L'avantage de la génération aléatoire reste dans le fait qu'il est facile de produire un grand nombre de données de test qui, par ailleurs, sont indépendantes d'un type particulier de fautes.

## 2.2 Test des logiciels synchrones

### 2.2.1 Logiciels réactifs

Les logiciels informatiques classiques, dits transformationnels, recupèrent l'ensemble des données à l'initialisation, réalisent une transformation et retournent les résultats avant de terminer leur exécution. A l'opposé, les logiciels qui interagissent avec leur environnement, généralement fonctionnent d'une manière cyclique, chaque cycle correspondant à un échange actions-réactions. Selon la cadence de ces échanges, on distingue deux classes de logiciels : les logiciels *conversationnels* et les logiciels *réactifs*.

Les logiciels *conversationnels* imposent leur temps de réponse à l'environnement, en répondant (ou non) aux requêtes de l'environnement. La communication avec l'environnement se fait d'une manière asynchrone et souvent non-déterministe. Des exemples de logiciels conversationnels que nous utilisons tous les jours sont les logiciels de traitement de texte ou encore les systèmes d'exploitation.

A l'opposé, un logiciel *réactif* doit répondre à toute sollicitation de l'environnement et doit évoluer à la vitesse imposée par son environnement, c'est-à-dire qu'il doit réagir plus rapidement que ce dernier. La communication avec l'environnement se fait d'une manière synchrone et le déterminisme est une propriété souvent voulue pour les logiciels réactifs.

### 2.2.2 Logiciels réactifs synchrones

Un programme est dit *synchrone*, s'il vérifie l'hypothèse de synchronisme qui stipule que le calcul des sorties du programme à partir de ses entrées est instantané. De la même façon, les temps de communication entre le logiciel, ses différents composants et l'environnement doivent aussi être nuls. En supposant que le temps est divisé en des instants discrets définis par une horloge globale, un programme synchrone, à un instant $t$, lit ses entrées $i_t$ et calcule ses sorties $o_t$. L'hypothèse synchrone assure que le calcul et l'émission de $o_t$ est fait instantanément, au même instant $t$ (figure 2.1).

En pratique, on considère qu'un logiciel a un comportement synchrone s'il réagit à son environnement avant toute évolution de ce dernier. Ainsi, si à l'instant $t$ le logiciel

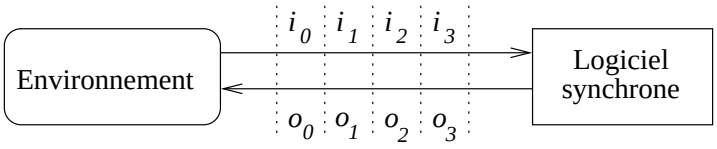


FIGURE 2.1 – Fonctionnement d'un logiciel synchrone

reçoit les entrées $i_t$ depuis son environnement externe, il émet les sorties $o_t$ avant qu'une nouvelle entrée $i_{t+1}$ soit disponible. Cette propriété permet de s'abstraire des problèmes de temporalité et de prendre en compte l'obligation de réactivité d'un logiciel.

Les logiciels réactifs synchrones sont donc particulièrement bien adaptés à la réalisation des *systèmes critiques* [42]. Un système est généralement dit critique lorsqu'une défaillance de celui-ci est susceptible de mettre en jeu des vies humaines ou d'importantes sommes d'argent. En effet, ces logiciels sont généralement utilisés dans des systèmes de type contrôle/commande où le logiciel doit repondre à toute évolution de l'environnement pour éviter une catastrophe. Le modèle synchrone est notamment utilisé par *Airbus* pour les logiciels de contrôle du vol de ses avions.

La conception de logiciels synchrones ne pouvant pas être réalisé facilement à l'aide de langages traditionnels, des langages plus spécialisés ont été introduits, parmi lesquelles compte le langage LUSTRE [10, 19].

### 2.2.3 Le langage LUSTRE

LUSTRE [10, 19] est un langage synchrone à flot de données, dédié à la spécification et la programmation des logiciels réactifs synchrones. Il constitue un langage exécutable même s'il est souvent utilisé comme langage de spécification. Un programme LUSTRE peut être considéré comme un réseau d'opérateurs constitué d'un ensemble de noeuds (les opérateurs du réseau) connectés par des canaux de communication, les données étant traités en traversant ce réseau.

LUSTRE est muni d'opérateurs temporels permettant de faire référence à des valeurs passées d'un flot. Le calcul de sortie peut dépendre, à un instant donné, aussi bien des valeurs courantes 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. De ce fait, LUSTRE permet d'exprimer des propriétés de

logique temporelle du passé.

### Syntaxe d'un programme LUSTRE

La figure 2.2 donne la structure syntaxique d'un programme LUSTRE. Les programmes sont structurés en *noeuds*. Un *noeud* comporte un ensemble d'équations non ordonnées qui définissent chacun des paramètres de sortie en fonction des paramètres d'entrée. Des variables locales peuvent être définies pour faciliter l'écriture des programmes en renommant certains flots. Une fois défini, chaque noeud peut être ensuite utilisé dans d'autres noeuds comme tout autre opérateur.

```
node Programme(<entrées>) returns (<sorties>)
var
    <variables locales>
let
    <sorties> = f(<entrées>,<variables locales>);
tel
```

FIGURE 2.2 – Structure syntaxique d'un programme LUSTRE

### Exemple d'un noeud LUSTRE

Le noeud *implies* de la figure 2.4 définit l'implication logique qui ne dispose pas d'opérateur spécifique dans LUSTRE.

```
node implies(a, b : bool) returns (imp : bool)
let
    imp = not(a) or b;
tel;
```

FIGURE 2.3 – Exemple de noeud LUSTRE définissant l'opérateur implies

### Opérateurs temporels

En plus des opérateurs arithmétiques et logiques habituels, LUSTRE admet des opérateurs spécifiques pour exprimer les événements du passé :

- L'opérateur *précédent* (noté *pre*) mémorise la valeur d'un flot d'un cycle à l'autre, introduisant ainsi un décalage du flot d'une unité de temps. Si $X$ correspond au flot $(x_0, x_1, x_2, ...)$, l'expression *pre(X)* définit le flot : $(\perp, e_0, e_1, ...)$, ou $\perp$ est une valeur non définie, comparable à une valeur non initialisée dans les langages impératifs.

- L'opérateur *suivi par* (noté '->') permet l'initialisation des flots non définis au premier cycle, habituellement après un décalage. Ainsi, si $X$ et $Y$ correspondent respectivement aux flots$(x_0, x_1, x_2, ...)$ et $(y_0, y_1, y_2, ...)$, l'expression *X->pre(Y)* définit le flot $(x_0, y_0, y_1, ...)$
- Les opérateurs *when* et *current* permettent d'échantillonner un flux en changeant de valeur seulement quand une valeur booléenne est vraie. Dans la suite, nous ne considérons que les programmes ayant une seule horloge. Ces simplifications sont habituelles pour la vérification, le fait qu'un programme LUSTRE avec plusieurs horloges se ramène à un programme avec une seule horloge ayant été établi pour la compilation des programmes LUSTRE [46].

**Exemple d'un noeud LUSTRE utilisant des opérateurs temporels**

L'opérateur *edge* décrit le front montant d'un flot booléen, retournant la valeur *vraie* lorsque le flot booléen donné en paramètre passe de *faux* à *vrai*. Cet opérateur est implémenté avec le noeud LUSTRE suivant de la figure 2.4.

```
node edge (X : bool) returns (Y : bool)
let
    Y = false -> X and not(pre X) ;
tel
```

FIGURE 2.4 – Exemple d'un programme LUSTRE

Si on considère que le flot à l'entrée de l'opérateur *edge* est $X = (x_0, x_1, x_2, ...)$, ce noeud définit le flot de sortie $Y = (y_0, y_1, y_2, ...)$ par la fonction suivante :

$$y_t = \begin{cases} false & si\ t = 0 \\ x_t \wedge \neg x_{t-1} & si\ t > 0 \end{cases}$$

#### 2.2.3.1 Exemple d'un programme LUSTRE

Considérons un exemple de programme réactif contrôlant un climatiseur dont la fonction est de commander les différents dispositifs du système de climatisation (émission d'air chaud ou froid en fonction de la température). Le programme prend en entrée un signal *Bouton*, actif si l'utilisateur a appuyé sur le bouton marche/arrêt, et 3 signaux (*Tinf*, *Tok* et *Tsup*) indiquant respectivement que la température est

inférieure, égale ou supérieure à celle fixée par l'utilisateur. Le vecteur de sortie $o$ est constitué des signaux *En_ marche* indiquant si le climatiseur est en marche, *Froid* et *Chaud* indiquant si le climatiseur fonctionne en mode chauffage ou refroidissement et *Inactif* si aucun des deux modes n'est actif (quand la température visée est atteinte). Dans la figure 2.5 on donne une réalisation possible de ce contrôleur dans le langage LUSTRE.

```
node Clim(Bouton, Tinf, Tok, Tsup : bool)
returns (En_marche, Froid, Inactif, Chaud : bool)
let
   En_marche = Bouton -> pre En_marche and not(Bouton)
      or not(pre En_marche) and Bouton ;
   Froid = En_marche and Tsup ;
   Inactif = En_marche and Tok ;
   Chaud = En_marche and Tinf ;
tel
```

FIGURE 2.5 – Contrôleur du climatiseur en LUSTRE

Le tableau 2.1 donne la trace d'une exécution illustrant le fonctionnement du climatiseur. Remarquons que pour les mêmes entrées, le logiciel peut produire des sorties différentes (instants $t_1$ et $t_7$), due à l'état du logiciel. En effet, le climatiseur reste en marche entre deux appuis successifs sur le bouton, mémorisant ainsi le fait qu'on a déjà appuyé sur le bouton dans le passé. Notons que les signaux *Froid*, *Inactif* et *Chaud* ne peuvent pas être vrais si le climatiseur n'est pas en marche.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ |
|---|---|---|---|---|---|---|---|---|---|
| Bouton | 0 | **1** | 0 | 0 | 0 | 0 | 0 | **1** | 0 |
| Tinf | 0 | **0** | 1 | 1 | 0 | 0 | 0 | **0** | 1 |
| Tok | 0 | **1** | 0 | 0 | 1 | 0 | 0 | **1** | 0 |
| Tsup | 1 | **0** | 0 | 0 | 0 | 1 | 1 | **0** | 0 |
| En_marche | 0 | **1** | 1 | 1 | 1 | 1 | 1 | **0** | 0 |
| Froid | 0 | **0** | 0 | 0 | 0 | 1 | 1 | **0** | 0 |
| Inactif | 0 | **1** | 0 | 0 | 1 | 0 | 0 | **0** | 0 |
| Chaud | 0 | **0** | 1 | 1 | 0 | 0 | 0 | **0** | 0 |

TABLE 2.1 – Chronogramme représentant une exécution possible du climatiseur

### Machine à états finis associée à un programme LUSTRE

L'opérateur *pre* constitue une mémoire de la valeur à l'instant précédent de l'expression associée. L'ensemble fini des utilisations de l'opérateur *pre* définit un

état du programme LUSTRE. Cet état représente en effet une abstraction des valeurs passées des variables d'entrée et de sortie du logiciel. Les sorties d'un logiciel à un instant donné sont alors complètement définies en fonction de l'état du programme et des entrées à ce même instant. En conséquence, en associant une variable d'état à chaque opérateurs *pre* ainsi qu'une variable supplémentaire pour distinguer l'instant initial, on peut associer à chaque programme LUSTRE une machine à états finis $M = (Q, q_{init}, E, S, f, t)$ ou :

- $Q$ est l'ensemble des états du logiciel (ensemble de valeurs des variables d'état) ;
- $q_{init} \in Q$ est l'état initial de l'environnement ;
- $e$ et $s$ étant les vecteurs des paramètres d'entrée et de sortie du logiciel sous test, $E$ et $S$ sont leurs ensembles de valeurs associées ;
- $f : Q \times E \rightarrow S$ est la fonction calculant la valeur des paramètres de sortie à partir de l'état et des valeurs des paramètres d'entrée et
- $t : Q \times E \times S \rightarrow Q$ et la rélation de transition définissant l'état suivant du logiciel après chaque cycle d'exécution du logiciel synchrone.

### 2.2.4 Vérification de logiciels synchrones écrits en LUSTRE

#### 2.2.4.1 Propriétés temporelles en LUSTRE

Le langage LUSTRE, muni de ses opérateurs temporels, peut etre considéré comme une logique temporelle du passé [41]. Des opérateurs temporels peuvent être définis par l'utilisateur. Les opérateurs suivants [2] sont souvent utilisés :

- *after(A)* est vrai s'il y a eu au moins une occurence de A par le passé.
- *always_since(A,B)* est vrai si A a été toujours vrai depuis la derniere occurence de B.
- *once_since(A,B)* est vrai si A a été vrai au moins une fois depuis la derniere occurence de B.
- *always_from_to(A,B,C)* est vrai si A a été vrai dans tous les instants entre l'instant ou B a été vrai et l'instant ou C a été vrai.
- *once_from_to(A,B,C)* est vrai si A a été vrai au moins pendant un instant entre l'instant ou B a été vrai et l'instant ou C a été vrai.

2. Le lecteur pourra trouver dans l'annexe A une implémentation de ces opérateurs en LUSTRE

#### 2.2.4.2 Propriétés de sûreté

Les aspects de logique temporelle du langage rendent facile la définition de propriétés décrivant les comportements attendus d'un logiciel ou des assertions sur son environnement d'exécution. Ainsi, il est possible d'écrire en LUSTRE les propriétés invariantes du logiciel qui doivent être vérifiées dans tous les états de ce dernier. Ces propriétés, dites *de sûreté*, définissent les comportements sûrs du programme. Dans l'exemple du climatiseur que nous avons traité précédement, nous pouvons définir les propriétés de sûreté suivantes.

- Le climatiseur qui est en marche doit émettre de l'air chaud quand il fait froid :

  `p1 = implies(En_marche and Tinf, Chaud) ;`
- Le climatiseur qui est en marche doit être inactif si la température voulue est atteinte :

  `p2 = implies(En_marche and Tok, Inactif) ;`
- Le climatiseur qui est en marche doit émettre de l'air froid quand il fait chaud :

  `p3 = implies(En_marche and Tsup, Froid) ;`

Ces propriétés peuvent être écrits dans un noeud séparé, dont les entrées sont faites des entrées et sorties du logiciel et la seule sortie correspond à la valeur de vérité des propriétés de sûreté (cf. figure 2.6). Le programme ainsi consititué peut servir d'oracle décidant si le logiciel respecte les propriétés de sûreté.

```
node PropClim(Bouton, Tinf, Tok, Tsup, En_marche, Froid, Inactif,
Chaud : bool)
returns (ok : bool)
let
   p1 = implies(En_marche and Tinf, Chaud) ;
   p2 = implies(En_marche and Tok, Inactif) ;
   p3 = implies(En_marche and Tsup, Froid) ;
   ok = p1 and p2 and p3 ;
tel
```

FIGURE 2.6 – Propriétés de sûreté du contrôleur du climatiseur

#### 2.2.4.3 Assertions sur l'environnement

De la même manière que les propriétés de sûreté qui portent sur le logiciel, les propriétés de l'environnement d'exécution du logiciel peuvent être décrites à l'aide des assertions. Une assertion est spécifiée à l'aide du mot clef `assert`. Les assertions

sont une partie intégrante du procéssus de développement de logiciels en LUSTRE. Elles sont utilisées aussi pour l'optimisation des programmes pendant la compilation. Dans l'exemple du climatiseur, on peut par exemple spécifier les assertions suivantes :

- La température doit être soit inférieure (`Tinf`), soit égale (`Tok`), soit supérieure (`Tsup`) à celle voulue par l'utilisateur :
  `assert(Tok or Tinf or Tsup) ;`
- Au plus un des signaux `Tinf`, `Tok`, `Tsup` peut être vrai à la fois :
  `assert(#(Tok, Tinf, Tsup)) ;`
- La température ne peut pas passer de `Tinf` à `Tsup`, sans passer par `Tok` *:*
  `assert(once_from_to(Tok, Tinf, Tsup)) ;`
- La température ne peut pas passer de `Tsup` à `Tinf`, sans passer par `Tok` :
  `assert(once_from_to(Tok, Tsup, Tinf)) ;`

#### 2.2.4.4 Vérification

Etant donné un programme P, des suppositions sur le comportement de l'environnement par l'assertion A et une propriété de sûreté S, un nouveau programme V peut être constitué englobant P, S et l'assertion A, comme dans la figure 2.7. Les assertions et les entrées de V sont idéntiques à celles de P alors qu'il contient une sortie unique *ok* représentant la valeur de vérité des propriétés de sûreté à vérifier. Ainsi, pour vérifier les propriétés du programme P, il faut vérifier que la sortie *ok* du programme V est vraie pendant toute exécution du programme dans un environnement qui sattisfait l'assertion A. Ce principe est connu sous le nom d'*observateur synchrone*.

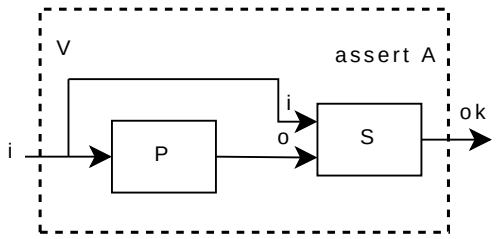


FIGURE 2.7 – Principe de l'observateur synchrone

Le noeud *VerifClim* de la figure 2.8 montre le programme de vérification créé pour l'exemple du climatiseur de la figure 2.5 et les propriétés de sûreté et assertions que nous avons spécifiées précédemment.

```
node VerifClim(Bouton, Tinf, Tok, Tsup : bool)
returns (ok : bool)
var
   En_marche, Froid, Inactif, Chaud : bool
let
   (En_marche, Froid, Inactif, Chaud) = Clim(Bouton, Tinf, Tok,
Tsup) ;
   ok = PropClim(Bouton, Tinf, Tok, Tsup, En_marche, Froid, Inactif,
Chaud) ;
   assert(Tok or Tinf or Tsup) ;
   assert(#(Tok, Tinf, Tsup)) ;
   assert(once_from_to(Tok, Tinf, Tsup)) ;
   assert(once_from_to(Tok, Tsup, Tinf)) ;
tel
```

FIGURE 2.8 – Programme de vérification des propriétés de sûreté du climatiseur

#### 2.2.4.5 Vérification formelle de propriétés

**LESAR** [20, 45] est un outil de vérification formelle par *model checking* développé pour le langage LUSTRE. Il se base sur le principe des observateurs synchrones (voir figure 2.7). Il commence par construire l'automate associé au programme de vérification. Il parcourt ensuite tous les états où l'assertion est vérifiée : dans chacun de ces états, LESAR s'assure que la propriété est toujours évaluée à *vrai*. Cependant, ce type d'approche souffre du problème de l'explosion combinatoire du nombre d'états à explorer : ce nombre croît (au pire) de manière exponentielle avec le nombre $n$ de variables booléennes d'état contenues dans le programme [3] : $nb_{state} \leq 2^n$.

#### 2.2.4.6 Test structurel en LUSTRE

Les critères de couverture utilisés pour le test des logiciels séquentiels, tels que la couverture des instructions, ne sont pas pertinents pour les logiciels écrits en LUSTRE : les données traversant l'ensemble des opérateurs du programme à chaque instant, on peut dire que les instructions sont couvertes avec une seule donnée de test.

L'approche proposée dans [29] ne définit de critère de test structurel pour le langage LUSTRE à proprement parler, mais sur l'automate de contrôle construit à partir du programme. Plus précisément, l'automate est transformé en un modèle

3. LESAR travaille uniquement avec des données booléennes.

stochastique de comportement du logiciel en associant une probabilité à chaque transition. Une séquence d'entrées est ensuite calculée de sorte à ce que la probabilité de couverture des états, des transitions ou des séquences de deux transitions soit supérieure à un seuil fixé au préalable.

Les travaux présentés dans [24] définissent une hierarchie de critères structurels adaptés au paradigme flot de données du langage LUSTRE. Ces critères sont définis sur le réseau d'opérateurs et permettent de mésurer la couverture des chemins, basé sur le calcul symbolique des conditions d'activation de ces chemins. Il a été montré dans [7] que la génération de données de test à partir de ces critères pourrait être envisageable.

#### 2.2.4.7 Test fonctionnel en LUSTRE

**LUTESS** [14] est un outil de test fonctionnel en *boîte noire* des logiciels synchrones. Il se base sur une spécification formelle de l'environnement du logiciel dans un langage basé sur LUSTRE, pour construire automatiquement un générateur efficace de données de test. Ce générateur se comporte comme un simulateur de l'environnement fournissant des données d'entrée au logiciel sous test. En plus de la génération aléatoire équiprobable, LUTESS implémente un certain nombre de stratégies de test qui peuvent être utilisées. En revanche, il est limité seulement aux logiciels ayant des entrées/sorties booléennes. Puisque les travaux présentés ici portent sur l'évolution de cet outil, le chapitre suivant est consacrée à une présentation plus détaillée de celui-ci.

**LURETTE** [47] est un autre outil de test fonctionnel de logiciels synchrones écrits en LUSTRE. De la même façon que LUTESS, il permet une simulation aléatoire du comportement de l'environnement du logiciel sous test. LURETTE construit automatiquement une séquence de test en générant et soumettant au logiciel sous test un vecteur d'entrées respectant des contraintes d'environnement ; chacune des entrées composant le vecteur pouvant être de type entier ou booléen.

Contrairement à LUTESS qui connecte le simulateur d'environnement à une version exécutable du logiciel sous test, LURETTE construit un programme exécutable

à partir du code C [4] du logiciel sous test, des assertions (environnement) et d'un oracle. Cette approche basée sur le modèle des observateurs [21] offre la possibilité de connaître complètement l'état du système et de l'environnement : cette information permet en particulier d'évaluer, dans chaque état du système, les différentes évolutions possibles du système en fonction de l'entrée choisie avant de lui soumettre. De même, on ne retrouve pas dans LURETTE les stratégies de génération automatique de LUTESS mais plutôt des langages de spécification de scénarios de test.

**GATeL** [28] permet de générer des séquences de test en se basant d'une part sur des objectifs de test écrits en LUSTRE étendu et d'autre part une description sous forme de noeuds LUSTRE du programme sous test (dans le cas du test fonctionnel) ou le programme LUSTRE lui-même (dans le cas du test structurel). Intuitivement, GATeL propose une solution permettant de générer automatiquement une séquence de test menant vers un objectif défini par le testeur.

L'approche proposée dans GATeL se décompose en trois parties :

1. l'outil commence par construire la séquence d'entrées qui va permettre d'atteindre l'objectif de test fixé par le testeur ;
2. ensuite, l'outil doit exécuter la séquence de test précédemment construite ;
3. enfin, l'oracle doit décider si le test est un succès ou non. Ici, deux cas sont possibles : soit nous sommes dans le cadre du test fonctionnel et la description LUSTRE du programme peut être assimilée à une spécification exécutable du logiciel sous test permettant de déterminer des valeurs de sortie à comparer à celles fournies par le logiciel sous test ; soit nous sommes dans le cadre du test structurel et l'oracle doit s'appuyer sur des propriétés annexes pour pouvoir décider du succès du test.

GATeL traduit le programme sous test et ses spécifications en une représentation équivalente en contraintes, dont la résolution vise la satisfaction d'un objectif défini par l'utilisateur. GATeL a été utilisé pour tester des programmes réalistes. Ses prin-

4. Ce code est en général obtenu par la compilation d'un langage de plus haut niveau dédié au développement des logiciels réactifs synchrones.

cipales différences avec LUTESS résident dans le traitement des spécifications et du programme (GATeL les combine en une seule représentation tandis que LUTESS les traite comme des composants distincts) et l'absence dans GATeL de stratégies de génération élaborées (les objectifs de test sont en effet définis par l'utilisateur).

## 2.3 Programmation par Contraintes et utilisation pour le test

La Programmation par Contraintes (PC) est une forme de programmation déclarative. Au lieu de spécifier une suite d'actions à exécuter, elle consiste à modéliser un problème par des relations mathématiques et ensuite appliquer des algorithmes standards pour rechercher une solution. Ainsi, de nombreux problèmes de planification et de gestion de ressources peuvent être facilement décrits comme des problèmes de satisfaction de contraintes. En général, la formulation et la résolution de problèmes combinatoires est un champ de prédilection pour la programmation par contraintes.

La représentation sous forme de relations permet de raisonner plus facilement avec des données qui sont partiellement connues. Les problèmes de génération de données de test à partir de spécifications s'inscrivent très naturellement dans cette optique, le but étant souvent d'instancier les variables d'entrée afin qu'elles vérifient une spécification donnée.

### 2.3.1 Introduction à la Programmation par Contraintes

La PC consiste à poser un problème sous forme de relations logiques (propriétés qui doivent être vérifiées), appelées *contraintes*, entre différentes inconnues, appelées *variables*, chacune prenant des valeurs dans un ensemble donné, appelé *domaine*. En fonction du problème modélisé, les variables, leur domaine de variation ainsi que les contraintes associées peuvent être de nature différente. Ainsi, le domaine des variables peut être fini (ensemble, intervalle d'entiers) ou continu (variables réelles).

Chaque contrainte définit une relation sur un sous-ensemble de variables restreignant les valeurs que peuvent prendre simultanément les variables. Par exemple, la contrainte "$2 \cdot x + y = 10$" restreint les valeurs que l'on peut affecter simultanément aux variables $x$ et $y$ aux celles qui satisfont cette équation. A la différence d'une

fonction qui définit la valeur d'une variable en fonction des valeurs des autres variables, les contraintes sont relationnelles et permettent de déduire n'importe quelle variable à partir des autres variables. Ainsi, la contrainte "$2 \cdot x + y = z$" permet de déduire $z$ à partir des valeurs de $x$ et $y$, mais aussi la valeur de $x$ (resp. $y$) en fonction des valeurs de $y$ et $z$ (resp. $x$ et $z$).

La recherche d'une solution au problème à contraintes se fait en utilisant un résolveur de contraintes. Un résolveur consiste en un algorithme décidant, quand cela est théoriquement possible, s'il existe ou non une affectation de toutes les variables telle que l'ensemble des contraintes du problème soient satisfaites. Dans ce cas, le problème à contraintes est dit *satisfiable*. Dans le cas contraire, il est dit *non satisfiable*.

Dans la suite de cette section nous décrivons les principes de résolution de contraintes sur les domaines finis. Cette restriction est due au fait que seul ce type de résolveur de contraintes a été utilisé dans nos travaux de thèse.

### 2.3.2 Contraintes sur les domaines finis

Quand le domaine de variation des variables d'un problème a une taille finie, on parle de problème à contraintes sur domaines finis. Un problème arithmétique avec des variables à valeur dans un ensemble fini d'entiers est un exemple de problème à contraintes sur les domaines finis. Dans ce cas, on peut appliquer des algorithmes spécifiques pour la recherche de solutions. Pour présenter les méthodes de résolution de ce type de contraintes, il est nécessaire d'introduire quelques définitions. Pour commencer, le terme problème de satisfaction de contraintes (CSP - de l'anglais "Constraint Satisfaction Problem") est utilisé pour décrire un problème à contraintes sur les domaines finis.

**Définition 2.3.1 (Problème de satisfaction de contraintes)** *Un CSP est un triplet $\langle X, D, C \rangle$ où :*

- *$X$ est un ensemble de variables ;*
- *$D$ est une fonction qui associe à chaque variable de $X$ son domaine fini de variation et*
- *$C$ est l'ensemble des contraintes.*

*On note par $dom(x)$ le domaine de la variable $x$, par $min(dom(x))$ (resp. $max(dom(x))$) la valeur minimale (resp. maximale) du domaine de la variable $x$ et par $vars(c)$ l'ensemble des variables associées à la contrainte $c$.*

Étant donné un CSP $\langle X, D, C\rangle$, sa résolution consiste à affecter des valeurs aux variables, de telle sorte que toutes les contraintes soient satisfaites. L'opération d'affectation d'une variable par une valeur dans son domaine est appelée *instanciation* :

**Définition 2.3.2 (Instanciation)** *L'instanciation d'une variable $x \in X$ du CSP $\langle X, D, C\rangle$ est une fonction de $X$ à valeurs dans $D$ qui associe à $x$ une valeur $v \in dom(x)$.*

A tout moment, une instanciation $\{x_i = v_i, x_j = v_j, ...\}$ définit un état du problème. Selon qu'une partie ou la totalité des variables d'un CSP est instanciée, on parle d'instanciation *partielle* ou *totale* du CSP. Si cette instanciation satisfait l'ensemble des contraintes, elle est dite *consistante*. Une instanciation totale et consistante représente une solution du système de contraintes.

**Définition 2.3.3 (Solution d'un CSP)** *Une solution d'un CSP $\langle X, D, C\rangle$ est une instanciation de toutes les variables $x \in X$ de celui-ci telle que chaque contrainte $c \in C$ soit satisfaite.*

**Exemple 2.3.1** Considérons le CSP suivant : $\langle\{x, y\}, \{0..10, 0..10\}, \{2 \cdot x \leq 10, x + y = 8\}\rangle$ où 0..10 dénote l'ensemble des entiers entre 0 et 10. Les solutions de ce problème consistent à instancier les variables $x$ et $y$ par des valeurs dans 0..10 telles que les deux contraintes $2 \cdot x \leq 10$ et $x + y = 8$ soient satisfaites. Par exemple, l'instanciation $\{x = 3, y = 5\}$ est une solution de ce CSP.

### 2.3.3 Résolution d'un problème à contraintes sur les domaines finis

Dans le cas ou les domaines des variables sont finis, il est en théorie possible d'énumérer toutes les valeurs possibles et de vérifier si elles violent ou non les contraintes. Cependant, en raison de la combinatoire, cette approche s'avère impraticable dès que le nombre de variables commence à augmenter. La difficulté de la résolution d'un CSP réside dans la mise en place d'un algorithme efficace permettant d'obtenir une solution à celui-ci ou prouver qu'il n'en a aucune. Dans le pire des cas, trouver une solution à un CSP nécessite une énumération sur l'ensemble des instanciations. Ce problème est NP-Complet [54].

Pour éviter l'énumération exhaustive, des algorithmes de *propagation* sont mis en oeuvre lors de la résolution. Le but de ces algorithmes est de déduire à partir des contraintes, les valeurs qui ne peuvent pas faire partie d'une solution et de les enlever du domaine des variables concernées. Le paragraphe 2.3.4 explique plus

en détail ces algorithmes. Cependant, la propagation seule ne permet pas toujours d'instancier toutes les variables du problème. L'*énumération* consiste alors à scinder le problème en plusieurs parties (par exemple en instanciant une variable à chacune de ses valeurs possibles) et relancer la propagation sur ces parties et ce, de manière récursive jusqu'à obtenir l'instanciation de toutes les variables. Nous présentons le concept d'énumération dans le paragraphe 2.3.6.

### 2.3.4 La propagation de contraintes

La propagation a pour objectif d'améliorer la recherche de solution par un algorithme moins coûteux que l'énumération. L'idée implicite est de détecter les valeurs qui ne peuvent pas prendre part à une solution et de les enlever du domaine des variables concernées, afin de réduire le coût de la recherche de solution par énumération. Cette opération de détection de ces valeurs et d'élimination du domaine est appelée *filtrage*. Le CSP réduit obtenu par filtrage a un domaine plus petit que le CSP de départ, mais reste équivalent au précédent dans le sens ou l'ensemble des solutions du CSP réduit est celui des solutions du CSP initial. Le domaine obtenu après filtrage est appelée *domaine réduit*.

A titre d'illustration, considérons deux entiers $x, y \in (1..3)$ et la contrainte $x < y$. Dans ce cas $y = 1$ ne peut pas faire partie d'une solution et la valeur 1 est enlevée du domaine de $y$. La propagation des contraintes permet dans ce cas de réduire les domaines des variables $x \in (1..2)$ et $y \in (2..3)$.

Le filtrage est effectué en raisonnant sur une contrainte à la fois et non pas sur l'ensemble des contraintes du système. Cette opération est appelée filtrage *local* contrairement au filtrage *global* consistant à considérer plusieurs ou l'ensemble des contraintes. Chaque réduction du domaine par un raisonnement local est ensuite propagée aux autres contraintes du CSP. Cette propagation prend la forme du réveil de contraintes pour lesquelles de nouveaux raisonnements locaux peuvent amener à des réductions de domaine. Ces réveils sont effectués jusqu'à ne plus obtenir de réduction de domaine. Le but de l'algorithme de propagation est d'organiser les différents réveils des contraintes.

**Exemple 2.3.2** Reprenons le CSP $\langle\{x, y\}, \{0..10, 0..10\}, \{2 \cdot x \leq 10, x + y = 8\}\rangle$, présenté dans l'exemple 2.3.1. En considérant la contrainte $2 \cdot x \leq 10$, on peut déduire

que $x$ ne peut être supérieur à 5 et réduire son domaine en 0..5. De cette nouvelle information, on peut en déduire une réduction du domaine de $y$ en considérant la contrainte $x + y = 10$. En effet, on déduit que $y$ doit être supérieur ou égal à 3 pour respecter cette contrainte. Comme aucune autre réduction ne peut être obtenue, l'algorithme de propagation s'arrête résultant au CSP réduit $\langle\{x, y\}, \{0..5, 3..10\}, \{2 \cdot x \leq 10, x + y = 8\}\rangle$.

Pour caractériser la précision des algorithmes de filtrage local, il est nécessaire de définir la notion de cohérence :

**Définition 2.3.4 (Cohérence de domaine)** *Soit un CSP $\langle X, D, C\rangle$, une contrainte $c \in C$ est cohérente de domaine ssi pour chaque variable $x$ de $vars(c)$ et pour chaque valeur possible de $dom(x)$, il existe une instanciation des variables restantes de $vars(c)$ telle que $c$ soit satisfaite. Un CSP est cohérent de domaine ssi chacune de ses contraintes est cohérente de domaine.*

Établir la cohérence de domaine d'un CSP peut s'avérer coûteux lorsque le domaine des variables est grand. Dans ce cas, le résolveur de contraintes assure une cohérence plus faible : la cohérence des bornes.

**Définition 2.3.5 (Cohérence des bornes)** *Soit un CSP $\langle X, D, C\rangle$, une contrainte $c$ est cohérente des bornes ssi pour chaque variable $x$ de $vars(c)$ et pour $min(dom(x))$ (resp. $max(dom(x))$) il existe une instantiation des variables restantes de $vars(c)$ telle que $c$ soit satisfaite. Un CSP est cohérent de bornes ssi chacune de ses contraintes est cohérente de bornes.*

### 2.3.5 Contraintes globales

En raisonnant sur une contrainte à la fois, on ne peut pas toujours détecter l'insatisfiabilité d'un système de contraintes. Par exemple, si $x, y, z \in \{1, 2\}$, le système de contraintes $x \neq y \wedge x \neq z \wedge y \neq z$ est cohérent de domaine mais n'admet pourtant pas de solution. Or, un simple raisonnement sur la cardinalité de l'ensemble des valeurs possibles permettrait de déduire qu'il ne peut y avoir de solution pour ce système de contraintes. Ces raisonnements qui portent sur l'ensemble des variables sont implémentée par des contraintes dites *globales*.

Le terme de contraintes globales est introduit par opposition à la décomposition d'une contrainte n-aire, comme par exemple $alldifferent(\{x_1, ..., x_n\})$ imposant que tous ses éléments soient différents, en une conjonction de contraintes binaires, comme $\bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} x_i \neq x_j$. L'intérêt d'utiliser une contrainte globale réside soit dans la représentation de contraintes n-aires qui sont indécomposables, soit dans la possibilité

d'implémenter des algorithmes de propagation plus efficaces que ceux obtenus par décomposition. Ainsi, la contrainte globale *alldifferent* est représentée par un graphe biparti dans [48], ayant pour sommets les variables et l'ensemble des valeurs de leurs domaines et pour arrêtes les associations permises. Le filtrage se fait alors par un algorithme linéaire par rapport au nombre d'arêtes du graphe.

De nombreuses contraintes globales ont été développées depuis les années 90. Souvent elles reposent sur un changement de représentation, faisant appel à la théorie des graphes et à l'adaptation d'algorithmes connus, en des algorithmes incrémentaux de test de satisfiabilité et de réduction des domaines.

### 2.3.6 L'énumération

L'énumération est généralement utilisée en dernier recours car coûteuse. Elle est lancée lorsque l'algorithme de propagation ne permet plus d'obtenir de réduction de CSP. Elle se base sur la séparation du CSP en au moins deux CSP plus simples, ayant des solutions dans des ensembles disjoints, et la recherche des solutions de ces nouveaux problèmes. La séparation est obtenue par l'ajout de contraintes au CSP initial. En prenant le CSP $\langle X, D, C\rangle$ et une variable $x \in X$ , une séparation classique consiste à diviser le problème en deux : $\langle X, D, C \cup \{x = min(dom(x))\}\rangle$ et $\langle X, D, C \cup \{x > min(dom(x))\}\rangle$.

Une fois la séparation effectuée, l'algorithme de résolution est appliqué à ces deux nouveaux problèmes menant à d'autres énumérations possibles. Cette série de découpages du problème peut être représentée sous forme d'un arbre. Le but de la recherche est de parcourir cet arbre (en le construisant au fur et à mesure) jusqu'à trouver une solution au problème tandis que le filtrage consiste à « élaguer » cet arbre en supprimant toutes les parties n'aboutissant qu'à des contradictions.

Le choix de la séparation du problème en plusieurs problèmes est crucial pour permettre la résolution du CSP en un temps raisonnable. Cette "bonne séparation" dépend souvent du problème rencontré. Le choix d'une "bonne séparation" se fait donc parmi un ensemble d'heuristiques. Celles-ci portent classiquement sur l'ordre dans lequel les variables du CSP doivent être instanciées et l'ordre dans lequel le domaine de variation de chacune des variables est parcouru.

#### 2.3.6.1 Choix de l'ordre d'instantiation des variables

L'ordre d'intanciation des variables influe sur l'efficacité des algorithmes de recherche de solutions et l'ordre dans lequel ces solutions sont trouvées. En fonction des objectifs, cet ordre peut être arbitraire ou basé sur des heuristiques appropriées au problème. Généralement, les heuristiques sont constituées à partir du domaine des variables et du nombre de contraintes qui leur sont associées. Pour améliorer l'efficacité des algorithmes, généralement les variables les plus "critiques" sont instanciées en premier, c'est-à-dire celles qui interviennent dans beaucoup de contraintes et/ou qui ne peuvent prendre que très peu de valeurs.

Quelques méthodes connues de sélection des variables dans une liste consistent à :

- Sélectionner la première variable de la liste. Cette stratégie est utilisée par défaut par la plupart des implémentations de résolveurs de contraintes sur domaines finis.
- Sélectionner la première variable ayant **la plus petite valeur minimale**.
- Sélectionner la première variable ayant **la plus grande valeur maximale**.
- La première variable ayant le **plus petit domaine** réduit. Cette stratégie, communément connue sous le nom de "first-fail", suppose que le choix d'une variable de petit domaine va impliquer moins de séparations du CSP pour arriver jusqu'à son instantiation.
- La variable la **plus contrainte**, consistant à choisir la variables ayant le plus grand nombre de contraintes qui lui sont associées. L'idée derrière ce type d'énumération est que la réduction du domaine de cette variable va être propagée sur le plus de contraintes possibles, augmentant ainsi l'efficacité du filtrage.
- La variable de **plus petit domaine** la **plus contrainte**. Cette méthode, connue sous le nom "first-fail-constrained", consiste à choisir parmi les variables ayant le plus petit domaine, celle qui a le plus de contraintes qui lui sont associées.

Dans le cas ou ces stratégies ne sont pas adaptées au problème, dans la plupart des implémentations de résolveur de contraintes, l'utilisateur peut fournir sa propre

méthode [5] de choix de la variable.

#### 2.3.6.2 Choix de parcours du domaine de la variable

Il existe plusieurs façons de parcourir le domaine d'une variable et on ne peut parler de *bonne stratégie* que dans le contexte d'un problème particulier. Cependant, on peut lister certaines méthodes "classiques", consistant pour une variable $x \in X$ du CSP $\langle X, D, C\rangle$ à :

- Choisir d'abord la valeur *minimale* du domaine : $l = min(dom(x))$. Le CSP est ensuite séparé en deux : $\langle X, D, C \cup \{x = l\}\rangle\langle X, D, C \cup \{x \neq l\}\rangle$. Cette stratégie explore les valeurs de $x$ dans l'ordre croissant en commençant par la plus petite valeur permise.
- Choisir d'abord la valeur *maximale* du domaine : $l = max(dom(x))$. Le CSP est ensuite séparé en deux : $\langle X, D, C \cup \{x = l\}\rangle\langle X, D, C \cup \{x \neq l\}\rangle$. Cette stratégie explore les valeurs de $x$ dans l'ordre décroissant en commençant par la plus grande valeur permise.
- Séparer le domaine en deux en se basant sur la valeur de milieu $m$ du domaine, et séparer ensuite le CSP en deux : $\langle X, D, C \cup \{x \leq m\}\rangle\langle X, D, C \cup \{x > m\}\rangle$. Cette stratégie, connue sous le nom de "domain splitting", est intéressante quand très peu de solutions sont attendues alors que les domaines des variables restent assez grands.
- Une valeur *aléatoire* dans le domaine : $v \in dom(x)$. Cette méthode est surtout utile quand on ne s'intéresse qu'à une seule et non pas l'ensemble des solutions. Elle permet d'obtenir aléatoirement une des solutions du CSP, sans avoir à parcourir toute l'espace des solutions.

Selon la nature du problème, d'autres énumérations sont possibles, obtenues notamment en combinant les méthodes précédentes. En général, les résolveurs de contraintes proposent à l'utilisateur de fournir sa propre stratégie par un algorithme censé réduire le domaine de la variable.

5. Un prédicat est utilisé dans le cas des langages logiques.

### 2.3.7 Programmation logique avec contraintes

La Programmation Logique avec Contraintes (PLC) a été introduite par Jaffar et Lassez en 1987 [23]. Ils ont observé que les égalités et inégalités des termes de Prolog II étaient une forme spécifique de contraintes et ont généralisé cette idée pour des langages arbitraires de contraintes. Les langages de programmation logique avec contraintes sont une généralisation du langage Prolog dans laquelle on peut considérer n'importe quel système de contraintes, en plus des contraintes d'égalité sur le domaine de Herbrand. Il s'agit donc d'une classe de langages de programmation paramétrées par le système de contraintes $\mathcal{C}$, que l'on note CLP($\mathcal{C}$).

Les programmes logiques avec contraintes ressemblent aux programmes logiques usuels, sauf qu'ils peuvent contenir des contraintes dans la définition des règles. Par exemple, la contrainte `X>0` est inclue dans la règle suivante :

```
a(X,Y) :- X>0, b(Y).
```

Ainsi, pour prouver un but `a(X,Y)`, en plus de prouver le but `b(Y)`, il faut que la valeur choisie pour `X` soit strictement positive.

L'exécution d'un programme logique avec contraintes se fait par un interpréteur qui évalue les requêtes en essayant de prouver un but. A la différence d'un programme logique, des contraintes peuvent faire partie du but à prouver. Les contraintes qui sont rencontrées pendant la preuve d'un but sont placées dans un ensemble, appelé *store*. Si une insatisfiabilité de l'ensemble des contraintes est détectée, l'interpréteur fait un retour en arrière, en essayant d'utiliser d'autres règles pour prouver le but. Ces mécanismes de retour en arrière, incorporés dans l'interpréteur, permettent aussi de contrôler les méthodes de recherche d'une solution (énumérations) lorsque les méthodes standard ne sont pas adaptées.

### 2.3.8 Utilisation de la PLC dans le test

La programmation logique avec contraintes a été souvent suggérée pour la génération de données de tests.

Dans le cadre du test structurel, l'outil INKA [18] utilise la PLC pour générer des données de test qui permettent d'atteindre un point précis du graphe de contrôle du programme. Ceci permet l'augmentation de la couverture structurelle

du programme, ainsi que la détection de chemins non exécutables.

La programmation par contraintes est aussi utilisée pour la construction automatique de générateurs pour le test structurel statistique dans [40]. Ce travail, propose une librairie de résolution de contraintes probabilistes et son utilisation pour la sélection uniforme des chemins exécutables dans un graphe de contrôle. L'outil GENETTA mets en oeuvre cette méthode pour des programmes C.

Dans le cadre général des systèmes réactifs, notons en particulier l'utilisation de la PLC pour la génération des tests basé sur des modèles, implémentée dans l'outil AUTOFOCUS [43], un outil de spécification formelle graphique de systèmes réactifs. Les modèles et les spécifications de contraintes de test sont automatiquement traduites en contraintes, et leur résolution donne des traces d'entrée/sorties qui représentent des séquences de test.

Dans le cadre des programmes synchrones, l'outil de test des GATeL [28], que nous avons présenté précédemment, traduit un programme LUSTRE en une représentation équivalente en contraintes et l'utilise pour la construction des séquences de test. La génération commence à partir d'un objectif de test, représentant un état particulier du programme, et fait une recherche en arrière pour trouver un chemin jusqu'à un état initial. Les transitions possibles sont conditionnées par les contraintes représentant le programme sous test.

# Chapitre 3

# Test des logiciels synchrones avec LUTESS

Les contributions présentées dans cette thèse portent sur le test des logiciels réactifs synchrones. Plus précisément, nous souhaitons améliorer les techniques de génération de données de test qui sont développées autour de l'outil LUTESS [14,36, 55,62] que nous présentons, naturellement, dans ce chapitre.

Nous présentons d'abord l'approche de LUTESS pour automatiser le test des logiciels synchrones et les formalismes associés (3.1), pour ensuite introduire les différentes techniques de génération de données implantées dans l'outil (3.2). Ce chapitre se termine par une présentation des limitations des différentes techniques de test (3.5), principalement dues au fait que LUTESS ne traite que des logiciels ayant des entrées/sorties booléennes, et qui définissent la problématique principale de notre travail.

## 3.1 Introduction à l'outil LUTESS

LUTESS est un outil de test pour la validation des programmes réactifs synchrones à entrées/sorties booléennes. L'outil LUTESS fournit un environnement permettant de réaliser du test fonctionnel en "*boîte noire*" à l'aide de spécifications formelles, écrites dans le langage LUSTRE. A partir de ces spécifications, LUTESS permet de générer automatiquement et dynamiquement des données de test qui respectent les contraintes de l'environnement du logiciel. En plus de la génération aléatoire, les données de test peuvent être choisies selon différents critères afin de guider le test vers des situations jugées intéressantes, comme par exemple, celles correspondant à

des comportements dangereux.

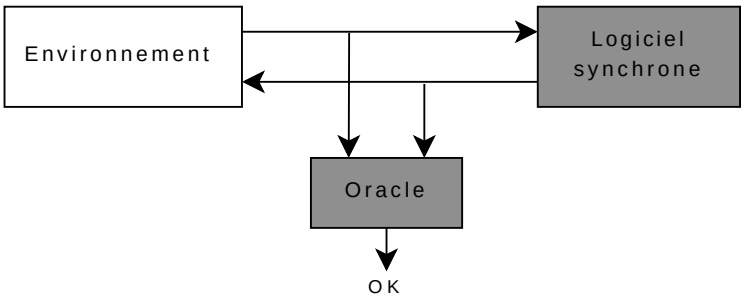


FIGURE 3.1 – Principe de fonctionnement de LUTESS

Le principe de base de LUTESS est de simuler l'environnement du logiciel sous test en lui fournissant des données d'entrée. Il intègre (fig. 3.1) la génération des jeux de tests et leur exécution. La génération de tests nécessite trois éléments : le logiciel sous test, un oracle et une spécification de l'environnement. Le programme sous test est donné sous sa forme exécutable et est supposé avoir un comportement synchrone. L'oracle est un autre programme exécutable examinant la séquence d'entrées-sorties du logiciel afin de donner un verdict sur leur concordance avec les propriétés spécifiées par le testeur. L'environnement est spécifié dans un *noeud de test*, utilisé ensuite pour produire un générateur de données de test.

### 3.1.1 Spécification de l'environnement du logiciel

L'ensemble des contraintes d'environnement qui permettent à LUTESS de générer des données de test est exprimé dans un *noeud de test*. La syntaxe d'un noeud de test, repéré par le nouveau mot clef *testnode*, correspond à une extension de la grammaire de LUSTRE. La forme générale d'un noeud de test est présentée dans la figure 3.2.

```
testnode Environnement(<sorties du logiciel>)
    returns (<entrées du logiciel>)
var
    <variables locales>
let
    environment(I1,...,In) ;
    <définition des variables locales>
tel
```

FIGURE 3.2 – Structure syntaxique d'un noeud de test

Un noeud de test prend comme paramètres d'entrée les sorties du logiciel sous test et comme paramètres de sortie les entrées à fournir au logiciel. Les mots clés *let* et *tel* encadrent le corps du noeud de test : l'opérateur *environment* permet d'exprimer les contraintes invariantes d'environnement $I_1, \ldots, I_n$. Chacune de ces propriétés est exprimée en fonction des variables d'entrée à l'instant courant ainsi que les valeurs précédentes des variables d'entrée et sortie. Les variables de sortie du logiciel ne peuvent pas être utilisées car elles ne sont pas connues au moment de la génération.

Un ensemble de déclarations de variables locales nécessaires à l'expression des contraintes d'environnement peuvent être définies à l'aide du mot clé *var*. Chacune de ces variables locales doit être définie par une équation LUSTRE. Dans un noeud de test, les variables locales offrent la possibilité de définir des valeurs intermédiaires permettant de simplifier l'écriture des contraintes d'environnement : leur utilisation est facultative.

On note que ce noeud ne contient aucune équation pour définir les valeurs de ses sorties. En effet, les valeurs émises par un noeud de test sont choisies de manière non-déterministe parmi toutes celles satisfaisant les contraintes d'environnement.

### 3.1.2 Exemple du contrôleur d'un climatiseur

Afin d'illustrer le fonctionnement de LUTESS, nous reprénnons l'exemple du contrôleur d'un climatiseur que nous avons traité dans le chapitre précédent. Afin de remplir sa fonction, quand le climatiseur est en marche, il doit émettre de l'air chaud quand il fait froid et inversement, émettre de l'air froid quand il fait chaud. Dans la figure 3.3, une telle propriété est utilisée pour spécifier en LUSTRE un oracle pour l'exemple du climatiseur ; cet oracle vérifie, si le climatiseur émet de l'air chaud quand il fait froid.

```
node Oracle(Bouton, Tinf, Tok, Tsup, En_marche, Froid, Inactif,
    Chaud : bool) returns (ok : bool)
let
    ok = implies(En_marche and Tinf, Chaud) ;
tel
```

FIGURE 3.3 – Réalisation de l'oracle en LUSTRE

Pour générer des données de test avec l'outil LUTESS, il faut spécifier les contraintes de l'environnement dans lequel ce climatiseur est censé s'exécuter. Le noeud de test de la figure 3.4 montre un exemple de spécification stipulant que la température ne peut pas passer directement de *Tinf* (froid) à *Tsup* (chaud), sans passer par la température fixée par l'utilisateur (*Tok*).

```
testnode Env_clim(En_marche, Froid, Inactif, Chaud : bool)
    returns (Bouton, Tinf, Tok, Tsup : bool)
let
    environment( true -> (not(Tsup) or not(pre Tinf)) ) ;
tel
```

FIGURE 3.4 – Exemple de spécification de l'environnement du climatiseur en LUSTRE

### 3.1.3 Machine à états finis associée à une spécification

Un noeud de test est transformé en un *simulateur d'environnement* qui est une machine à états finis.

**Définition 3.1.1** *Simulateur d'environnement*
*Un simulateur d'environnement* $M = (Q, q_{init}, E, S, trans, env)$ *est défini comme suit :*

- $Q$ *est l'ensemble des états de l'environnement (ensemble de valeurs des variables d'état q) ;*
- $q_{init} \in Q$ *est l'état initial de l'environnement ;*
- *e et s étant les vecteurs des paramètres d'entrée et de sortie du logiciel sous test,* $E$ *et* $S$ *sont leurs ensembles de valeurs associées ;*
- *La relation de transition trans :* $Q \times E \times S \rightarrow Q$ *définit l'état suivant de l'environnement après chaque échange d'entrées-sorties avec le logiciel synchrone et*
- *La relation env :* $Q \times E \rightarrow \{false, true\}$ *définit à tout moment la conformité de l'ensemble des valeurs d'entrée à la spécification de l'environnement.*

#### Illustration

Pour donner une idée plus précise de cette transformation en une machine à états finis, nous reprenons l'exemple précédent de l'environnement du climatiseur. La figure 3.5 présente une version dans laquelle nous introduisons deux variables locales supplémentaires : la variable *q0* permet de distinguer l'état initial et *q1* fait référence à la valeur précédente de *Tinf*. Cette version, sémantiquement équivalente à la précédente, met en évidence l'obtention des variables d'état (en l'occurrence *q0* et *q1*) de la machine à états finis associée.

```
testnode Env_clim(En_marche, Froid, Inactif, Chaud : bool)
    returns (Bouton, Tinf, Tok, Tsup : bool)
var
    q0,q1 : bool ;
let
    q0 = true -> false ;
    q1 = pre Tinf ;
    environment( if q0 then true else (not(Tsup) or not(q1)) ) ;
tel
```

FIGURE 3.5 – Réécriture de l'environnement pour mettre en évidence les variables d'état

Formellement, à l'exemple précédent correspond la machine à états finis $M = (Q, q_{init}, E, S, trans, env)$ :

- État : $Q = \{(q_0, q_1)\}$ , état initial $q_{init} \in \{(q_0, q_1) \in Q \mid q_0 = true\}$
- Paramètres d'entrée : $e = \{Bouton, Tinf, Tok, Tsup\}$
- Paramètres de sortie : $s = \{En\_marche, Froid, Inactif, Chaud\}$
- Fonction de transition $trans$ : $q_0 = false \wedge q_1 = Tinf$
- Fonction de l'environnement $env$ : $q_0 \vee \neg q_0(\neg Tsup \vee \neg q_1)$

### 3.1.4 Génération de données de test

Le simulateur d'environnement agit comme un générateur de tests dont le coeur consiste en une animation des spécifications de l'environnement. En effet, une séquence de test correspond à un parcours de la machine $M$, en partant de l'état initial. L'algorithme de génération peut se résumer en une boucle consistant à chaque instant $t$ à :

1. choisir un vecteur d'entrées $e_t$ dans l'ensemble des valeurs possibles $E$ qui dans l'état courant $q_t$ respecte les contraintes d'environnement, c'est à dire $\{e \in E \mid env(q_t, e) = true\}$ ;
2. envoyer les entrées $e_t$ au logiciel ;
3. récupérer les sorties $s_t$ du logiciel et
4. calculer l'état suivant de l'environnement à l'aide de la fonction de transition : $q_{t+1} = trans(q_t, e_t, s_t)$.

## 3.2 Stratégies de génération de LUTESS

A chaque instant de la génération, un ensemble de vecteurs d'entrée, dits *valides*, satisfont les contraintes d'environnement. Les stratégies de génération de LUTESS portent sur les critères de sélection des vecteurs d'entrées parmi l'ensemble des vecteurs valides.

### 3.2.1 Génération aléatoire équiprobable

La génération aléatoire équiprobable consiste à choisir un vecteur d'entrées d'une manière équitable. Ainsi, chaque vecteur d'entrées valide a la même probabilité d'être choisi. La représentation judicieuse des formules booléennes à satisfaire, permet à LUTESS d'assurer cette équiprobabilité de tirage d'une manière très efficace. Nous discuterons les détails de cette représentation dans la section 3.4, consacrée à l'implémentation de l'outil.

### 3.2.2 Génération guidée par les propriétés de sûreté

Les propriétés de sûreté sont un élément important dans le développement des logiciels réactifs synchrones. Généralement, elles définissent l'absence de comportement du logiciel pouvant amener à une situation critique (dysfonctionnement aux conséquences graves). Ces propriétés peuvent constituer une base pour définir un oracle de test. De plus, dans le cas où elles sont formellement spécifiés en LUSTRE, elles peuvent être utilisées dans la génération de tests [35,55]. En effet, en détectant les valeurs d'entrée qui ne peuvent pas violer les propriétés, le générateur de tests peut éviter de les choisir, augmentant ainsi la probabilité de détecter des cas de violation de ces propriétés.

Cette technique de test vise à générer les tests qui sont susceptibles de détecter la violation d'une propriété [38]. Étant donné une propriété, le but est d'éviter de produire des entrées qui à l'évidence ne peuvent pas violer celle-ci. Par exemple, si $i$ et une entrée et $o$ une sortie d'un programme devant satisfaire la propriété $i \Rightarrow o$, seulement l'entrée $i = vrai$ permet la violation de la propriété (sinon, la propriété sera vraie quelle que soit la valeur de $o$). Le guidage par les propriétés de sûreté favorise la génération de l'entrée $i = vrai$, à condition que les contraintes

d'environnement le permettent.

Les propriétés de sûreté qui vont guider la sélection des vecteurs d'entrées sont décrites dans le noeud de test de manière similaire aux contraintes d'environnement et identifiées par l'opérateur *safety*. Considérons l'exemple du climatiseur et la propriété de sûreté suivante : "S'il fait froid et le climatiseur est en marche, il doit émettre de l'air chaud". Le test peut être guidé par cette propriété en spécifiant :

```
safety(implies(Tinf and En_marche, Chaud)) ;
```

Dans les séquences générées, LUTESS va favoriser l'apparition de `Tinf=true`, car c'est une condition nécessaire pour violer cette propriété.

De même, puisqu'une propriété de sûreté peut faire référence à des valeurs passées, si on considère la propriété $true \rightarrow \mathbf{pre}\ i \Rightarrow o$, il faut engendrer l'entrée $i = true$ à l'instant courant pour que la propriété puisse être violée à l'instant suivant. D'une manière générale, ce type de guidage consiste à engendrer, à un instant $t$, des entrées qui peuvent mener le logiciel dans une situation où la propriété peut être violée à un instant $t + k$, $k$ étant le nombre maximum d'instants à considérer.

La définition du simulateur d'environnement défini plus haut est étendue de telle sorte que les propriétés de sûreté y soient intégrées. La différence principale avec le simulateur d'environnement réside dans le calcul de l'ensemble des vecteurs *pertinents*. Ce sont les vecteurs qui respectent l'environnement et sont à l'origine d'un chemin de longueur k, sur lequel la propriété de sûreté est susceptible d'être violée . Le calcul se fait en représentant dans un BDD l'ensemble des conditions caractérisant un tel chemin. Selon les condition de violation qui sont posées, trois stratégies de calcul différentes sont définies dans [55] :

- la *stratégie d'union* considère tous les chemins de longueur $1..k$ qui mènent le logiciel à une situation susceptible de violer la propriété de sûreté (autrement dit *état suspect*) ;
- la *stratégie d'intersection* ne considère que les vecteurs d'entrée qui font partie de tous les chemins de longueur $1..k$ qui mènent à un état suspect ;
- la *stratégie paresseuse* choisit le vecteur qui mène le plus rapidement à un état suspect.

### 3.2.3 Génération basée sur les profils opérationnels

Le guidage basée sur les profils opérationnels [13] permet de rendre l'opération de test proche d'un profil d'utilisation du logiciel [30]. Cette méthode vise à garantir que les fonctionnalités les plus utilisées seront celles qui seront le plus testées. Les probabilités conditionnelles sont pratiques pour définir des profils opérationnels d'une manière implicite.

La construction syntaxique `prob((`$e_1$`,`$p_1$`,`$c_1$`),...,(`$e_n$`,`$p_n$`,`$c_n$`))` permet de définir, dans un noeud de test, une liste de $n$ probabilités. Chacun des triplets `(`$e_i$`,`$p_i$`,`$c_i$`)` pour $i = 1..n$, spécifie la probabilité d'occurrence $p_i$ d'un paramètre d'entrée du logiciel $e_i$, qui ne sera effective que si la condition $c_i$ est vérifiée dans l'état courant.

A titre d'illustration, considérons de nouveau l'exemple du climatiseur. La génération purement aléatoire fait que *Bouton* est vrai, en moyenne, une fois sur deux, ce qui ne permet pas d'observer une exécution du logiciel pendant laquelle le climatiseur resterait en marche longtemps. Pour s'approcher d'une exécution plus réaliste, on peut spécifier que la probabilité d'appuyer sur *Bouton* est faible (resp. élevée) si $pre\ En\_marche = true$ (resp. $pre\ En\_marche = false$) :

```
proba(
    (Bouton, 0.1, pre En_marche),
    (Bouton, 0.9, not pre En_marche)
) ;
```

Les probabilités portant sur une même variable sont évaluées dans l'ordre de spécification et seulement si les conditions des probabilités précédentes ne sont pas vérifiées. Ceci permet de garantir une cohérence de l'ensemble des probabilités, dans le cas ou les conditions ne serait pas exclusives entre elles.

### 3.2.4 Génération à l'aide de schémas comportementaux

La génération guidée par les schémas comportementaux a été introduite dans [62]. Le but de cette méthode est de permettre à l'utilisateur de guider plus précisément la génération vers une situation précise, en indiquant les situations intermédiaires dans lesquelles l'environnement doit se trouver. Un schéma se présente comme une succession de conditions portant alternativement sur des instants et sur des intervalles. Les conditions d'instants portent sur les entrées qu'on souhaite voir apparaître et les

conditions d'intervalle sur les invariants à maintenir entre deux instants successifs. Le guidage se fait en augmentant la probabilité d'occurrence de ces conditions, sans pour autant forcer leur apparition. Théoriquement, un schéma comportemental peut aussi être ramenée à un ensemble de spécifications de probabilités conditionnelles.

## 3.3 Arrêt du test

Le problème d'arrêt du test dans LUTESS se pose de deux manières :

- déterminer la longueur de la séquence qui est générée,
- déterminer le nombre de séquences à générer.

La longueur de la séquence est arbitraire : le processus de génération est déconnecté de la décision d'arrêt du test.

En ce qui concerne le nombre de séquences qu'il faut générer avant de décider que l'activité de test est complétée, il dépend étroitement de la nature du logiciel. Cependant, pour les logiciel réalisées en LUSTRE, des critères formels de couverture structurelle spécifiques, définis dans [24], pourraient être utilisés à cette fin.

## 3.4 Implémentation

### 3.4.1 Représentation avec un graphe de décision binaire

La fonction de l'environnement *env* étant une fonction booléenne, elle peut être représentée efficacement par un Graphe de Décision Binaire (*BDD - Binary Decision Diagram*) [2]. C'est le choix de représentation qui a été fait dans la première version de LUTESS et ses évolutions ultérieures, qui ne traitent que les programmes à entrées et sorties booléennes. Un Graphe de Décision Binaire est une représentation optimisée de l'arbre de Shannon pour les formules booléennes. Il est obtenu en éliminant les noeuds redondants et en partageant les sous-arbres isomorphes.

On notera par la suite $\Delta_f$ le BDD représentant la fonction booléenne $f$. Chaque noeud de ce BDD correspond à une variable de la fonction $f$. Chaque arc sortant représente la valeur que peut prendre la variable du noeud. Par convention, la branche de gauche correspond à la valeur *faux* et la branche de droite à *vrai*. Un chemin de la racine à une feuille correspond à une valuation de toutes les variables de la formule. Les feuilles de l'arbre sont valuées avec les constantes booléennes *vrai* et *faux*

et correspondent à la valeur de la formule pour cette valuation. Une valuation est *valide* par rapport à $f$ si la feuille désignée porte la valeur *vrai*.

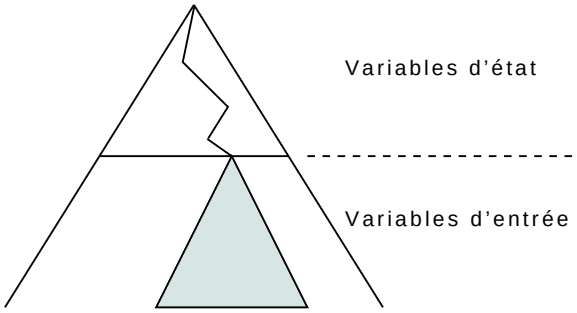


FIGURE 3.6 – Représentation de l'environnement par un BDD

La fonction $env : Q \times E \rightarrow \{false, true\}$ que nous représentons par le BDD $\Delta_{env}$ est une formule booléenne dont les variables sont les variables d'état et les variables d'entrée. A chaque pas de la génération, les variables d'état sont complètement connues et la génération consiste à choisir les valeurs des entrées telles que $env = true$. Afin d'optimiser la génération, LUTESS place les variables d'état en haut de l'arbre (voir fig. 3.6). Ainsi, un simple parcours permet d'accéder au sous-arbre représentant les valeurs que les entrées peuvent prendre dans l'état courant. Une affectation valide du sous-arbre correspondant représente des valeurs d'entrée valides.

### 3.4.2 Étiquetage des noeuds du BDD

Les algorithmes de génération de LUTESS sont basés sur l'étiquetage des noeuds du BDD, effectué une seule fois lors de sa construction. Chaque noeud correspondant à une variable d'entrée $x$ est étiqueté par un couple d'entiers $(v_0, v_1)$, représentant le nombre de chemins distincts menant à des feuilles évaluées à *vrai,* quand $x$ vaut respectivement faux ou vrai.

Considérons par exemple la fonction booléenne $f(e_0, e_1, e_2) = e_0 \vee (\neg e_1 \wedge e_2)$. La figure 3.7 représente le BDD et l'arbre étiqueté associé.

LUTESS utilise cet étiquetage des noeuds pour assurer l'équiprobabilité de tirage pendant la génération. Si on suppose que l'étiquette d'un noeud est $(v_0, v_1)$, la

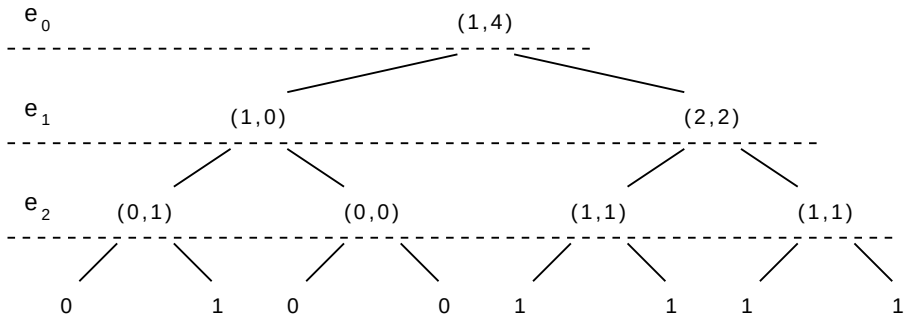


FIGURE 3.7 – BDD étiqueté

variable correspondante $e$ prend sa valeur en fonction des probabilités suivantes :

$$\mathcal{P}(e = false) = \frac{v_0}{v_0 + v_1} \text{ et } \mathcal{P}(e = true) = \frac{v_1}{v_0 + v_1}$$

On peut facilement vérifier sur l'exemple de la figure 3.7, que la probabilité pour choisir chacun des 5 vecteur valides est égale à 1/5.

Les différentes méthodes de guidage de LUTESS se basent aussi sur une représentation des différentes formules booléennes par des BDD et l'étiquetage des noeuds, en construisant d'autres BDD si nécessaire. Par exemple, la génération basée sur les profils opérationnels crée un étiquetage supplémentaire pour représenter les probabilités d'occurrence alors que le guidage par des schémas comportementaux crée 3 BDD pour chacune des conditions spécifiées.

## 3.5 Problématique de la thèse

Si le choix de représenter les contraintes par des BDD permet de générer des données de test d'une manière efficace, il limite néanmoins l'utilisation de LUTESS à un cadre strictement booléen. Ainsi, les nombreux logiciels qui comportent des entrées/sorties numériques ne peuvent pas être testés avec cet outil.

Les différentes techniques de test qui ont été développées dans LUTESS subissent, elles aussi, cette limitation. De plus, chacune de ces techniques a été développée séparément, si bien que les différentes techniques de test ne peuvent pas être combinées entre elles.

Les limitations de LUTESS sont donc de deux natures :

– soit elles sont liées au cadre strictement booléen et au choix d'implémentation

au moyen de BDD (cf. section 3.5.1).

- soit elles sont méthodologiques, dues à un manque de synthèse des différentes techniques (cf. section 3.5.2).

### 3.5.1 Limitations dues aux choix d'implémentation

#### Insuffisance pour applications numériques

Les logiciels réactifs réels ne se limitent pas aux données booléennes et contiennent souvent des valeurs numériques comme paramètres d'entrées/sorties. Pour élargir sensiblement le champ d'application des techniques de test de LUTESS, il a toujours été considéré comme indispensable de l'étendre afin de permettre la génération de données de test à partir de spécifications numériques [36].

Pour modéliser efficacement un noeud de test pour ces logiciels, il est nécessaire de pouvoir utiliser des variables numériques et d'exprimer des relations sur ces variables incluant, en plus des opérateurs booléens, les opérateurs relationnels et arithmétiques du langage LUSTRE. La spécification résultant de cette extension ne peut pas être représentée efficacement par des BDD. Sa prise en compte nécessite donc une représentation et un mode de calcul différents. De même, les différentes stratégies de génération étant fortement liées à la représentation en BDD et au cadre booléen, il est nécessaire de les redéfinir afin de les adapter à ce nouveau contexte.

#### Probabilités conditionnelles portant sur les variables de sortie uniquement

Dans le cadre de la définition de profils opérationnels, l'utilisateur peut définir un ensemble de probabilités conditionnelles, chacune portant sur une variable de sortie booléenne [15, 33]. Cette manière de définir les profils opérationnels est trop restrictive. Ceci est encore plus vrai avec l'introduction de paramètres d'entrée numériques qui rendent utile la possibilité d'exprimer des probabilités sur la validité d'une expression quelconque (par exemple la probabilité qu'un entier appartienne à un intervalle donné, la probabilité que deux entiers soient égaux, etc). Nous considérons donc nécessaire que les probabilités conditionnelles puissent porter sur toute expression LUSTRE à résultat booléen (et non pas seulement sur des variables booléennes).

#### Complexité du guidage par les propriétés de sûreté

Le guidage par les propriétés de sûreté recherche des chemins d'une longueur arbitraire $k$ pouvant violer la propriété. Chaque transition contenue dans un tel chemin introduit une nouvelle occurrence des variables d'état, d'entrée et de sortie. Les conditions caractérisant ce chemin, exprimées en fonction de toutes ces variables, sont représentées par un BDD. Du fait que la taille de l'arbre est exponentielle par rapport au nombre de variables impliquées, la taille du BDD ainsi que le temps nécessaire pour le construire deviennent rapidement très importants, dès que $k$ commence à augmenter. Pour faire face à ce problème, des solutions d'approximation sont proposées dans [55], comme par exemple, ignorer les invariants de l'environnement pour cette recherche. Ces approximations réduisent la capacité de détection d'erreurs pour ce guidage.

### 3.5.2 Limitations méthodologiques

#### Test en "boîte noire" et test en "boîte grise"

Si le logiciel sous test est considéré comme une "boîte noire", les sorties pouvant être calculées par ce dernier ne sont pas connues. Dans le guidage par propriétés de sûreté, LUTESS doit considérer tout le domaine des sorties comme une réaction possible du logiciel. En conséquence, un grand nombre de vecteurs d'entrée considérés pertinents par rapport à cette recherche, ne le sont pas en réalité, car les sorties associées ne sont pas possibles.

Pour améliorer ce guidage, plusieurs perspectives ont été proposées dans [55], comme par exemple l'apprentissage du logiciel. Dans le cas général, il est nécessaire de considérer l'intégration dans le processus de test d'une connaissance quelconque sur le logiciel, afin de rendre plus pertinentes les données de test produites. Avec l'introduction d'une telle connaissance, le logiciel ne serait plus considéré comme une boîte "noire" mais plutôt comme une boîte "grise" et s'approchant d'une boîte "blanche" avec l'augmentation de la quantité de connaissances introduites.

#### Impossibilité de combiner les différentes techniques

Les différentes techniques de génération abordées ici ont été développées indépendamment et leur intégration n'a pas été considérée auparavant. L'utilisateur ne

peut pas spécifier conjointement plusieurs techniques de guidage. Par exemple, on ne peut pas spécifier un profil d'utilisation en même temps que nous guidons le test par les propriétés de sûreté. Du point de vue méthodologique, il n'y a aucune raison de se limiter à une seule technique de test.

Reprenons l'exemple du climatiseur que nous avons traité précédemment et le guidage par la propriété de sûreté suivante :

```
safety(implies(Tinf and En_marche, Chaud)) ;
```

Pour violer cette propriété, entre autres, il est nécessaire que la sortie `En_marche` soit vraie. Or, le guidage par propriétés de sûreté ne peut influer directement sur la valeur des sorties. Par ailleurs, cette même sortie est souvent vraie lorsque le profil opérationnel suivant est spécifié :

```
proba(
    (Bouton, 0.1, pre En_marche),
    (Bouton, 0.9, not pre En_marche)
) ;
```

On constate que si LUTESS permettait l'utilisation de plusieurs techniques de test, le guidage par la propriété de sûreté serait rendu plus efficace avec l'introduction d'un profil opérationnel. En réalité, plus les comportements du logiciel sont complexes, plus il devient nécessaire de pouvoir utiliser plusieurs techniques de guidage dans la même séquence générée.

## 3.6 Conclusion

En conclusion, l'outil LUTESS permet de tester les logiciels synchrones booléens d'une manière dynamique et automatisée. Le test est basé sur des spécifications formelles de l'environnement ainsi que sur des stratégies de guidage du test. Grâce à l'utilisation ingénieuse des graphes de décisions binaire (BDD), la génération de test se fait d'une manière très rapide et les données de test sont générées de façon équiprobable. Un ensemble de méthodes de guidage de la génération sont proposées pour gérer des comportements plus complexes.

Mais, l'outil n'est pas directement utilisable sur un grand nombre de logiciels de la vie réelle, du fait de sa limitation exclusive aux logiciels booléens et ce bien que les logiciels de type contrôle-commande, qui sont la cible principale de l'outil,

comportent une partie significative booléenne. Il est nécessaire donc d'étendre cette approche, afin de prendre en compte les programmes comportant des entrées/sorties numériques.

Pour envisager cette extension, il est nécessaire de pouvoir représenter, en plus des formules booléennes, des relations complexes entre des données de type numérique. Les BDD n'étant pas adaptés à ce contexte, il est nécessaire de considérer une représentation et un mode de calcul différents. De même, les différentes techniques de test, doivent être adaptées à cette représentation est leur sémantique redéfinie dans le cadre numérique.

Dans le chapitre suivant, nous présentons les extensions que nous apportons à l'outil LUTESS afin de faire évoluer ces techniques de test dans le but de lever ces limitations.

# Deuxième partie

# Évolution des techniques de test

# Chapitre 4

# Test de logiciels synchrones numériques

Les évolutions que nous apportons aux techniques de génération de l'outil LUTESS sont de plusieurs natures :

- Des extensions de la syntaxe du langage de spécification utilisé dans LUTESS sont proposées. Cette nouvelle syntaxe permet, en plus des variables booléennes, l'utilisation de variables de type numérique et la définition de relations entre ces variables à l'intérieur des opérateurs de test.
- La sémantique associée aux opérateurs de test existants est redéfinie en conséquence et de nouveaux opérateurs de test sont introduits. La génération de tests, en plus d'être guidée par les propriétés de sûreté, peut inclure des *hypothèses de test* afin d'améliorer la pertinence des résultats. Les probabilités conditionnelles, affectées à des expressions de toute sorte, permettent la définition de profils opérationnels. Le lecteur remarquera l'utilisation de nouveaux mots clef pour les opérateurs de test ayant une sémantique qui diffère des travaux précédents.
- De plus, toutes ces différentes directives de guidage peuvent être utilisées conjointement dans une même spécification de test.
- Enfin, la réalisation du générateur de tests se base maintenant sur la programmation par contraintes.

Dans ce chapitre, nous présentons par des exemples, les extensions que nous apportons à la syntaxe du langage de spécification des générateurs de tests et la sémantique informelle associée. La définition formelle de cette sémantique fait l'objet du cha-

pitre 5, où l'on modélise notamment le problème de la génération de test comme un problème de résolution de contraintes. Les aspects liés à la réalisation de la nouvelle version de LUTESS sont, quant à eux, présentés au chapitre 6.

## 4.1 Motivations & principes

Considérons une nouvelle version de l'exemple du contrôleur de climatiseur présenté dans 3.1.2. Dans cette nouvelle version, au lieu des résultats de la comparaison entre les températures, le contrôleur manipule directement des entiers représentant les valeurs de ces températures.

```
node Clim(Bouton : bool ; Tamb, Tutil : int)
returns (En_marche : bool ; Tsort : int) ;
```

FIGURE 4.1 – Interface du climatiseur numérique en LUSTRE

La figure 4.1 décrit l'interface du nouveau contrôleur [1]. Il prend en entrée :

- la valeur booléenne *Bouton* qui sera vraie lorsque l'utilisateur appuie sur le bouton de mise en marche/arrêt du climatiseur ;
- l'entier *Tamb* qui représente la valeur de la température ambiante ;
- l'entier *Tutil* qui représente la valeur de la température voulue par l'utilisateur.

et calcule les sorties :

- le booléen *En_ marche* qui détermine si le climatiseur est en marche ;
- l'entier *Tsort* représentant la température de l'air sortant de la soufflerie du climatiseur.

Le rôle du contrôleur est de calculer la température de l'air sortant de la soufflerie *(Tsort)* de manière à faire évoluer la température ambiante *(Tamb)* vers celle voulue par l'utilisateur *(Tutil)*.

Un générateur de test pour ce contrôleur de climatiseur doit produire des entrées qui sont contraintes par les propriétés physiques de l'environnement dans lequel celui-ci va évoluer. Pour décrire précisément l'environnement de ce programme, on a besoin d'utiliser des variables de type numérique et d'exprimer des relations entre ces variables. Par exemple, on doit pouvoir spécifier que la température ambiante

1. Une implémentation de ce contrôleur en LUSTRE peut être trouvé dans l'annexe A

ne peut varier que de peu entre deux instants successifs, ou alors que les différentes températures sont comprises dans des intervalles donnés.

Pour exprimer de telles contraintes invariantes, il est nécessaire d'augmenter le langage d'entrée de LUTESS avec les variables de type numérique et l'ensemble des opérateurs relationnels et arithmétiques du langage LUSTRE. Cette extension est présentée et illustrée par un exemple dans la section 4.2.

En plus de la satisfaction des contraintes de l'environnement, nous permettons le guidage de la génération de données de test selon plusieurs stratégies. La section 4.3 est consacrée à une présentation de ces stratégies. Le guidage par rapport aux propriétés de sûreté dans ce nouveau contexte numérique est présentée dans le paragraphe 4.3.1. Nous introduisons aussi le concept d'*hypothèse de test* dans le paragraphe 4.3.2, dans le but d'améliorer la pertinence des résultats du guidage par propriétés de sûreté. Une hypothèse est exprimée sous la forme d'une relation entre les entrées et sorties du logiciel et permet d'intégrer dans la génération des connaissances partielles ou des suppositions sur le comportement du logiciel. La spécification de probabilités conditionnelles est présentée dans le paragraphe 4.3.3. A la différence de la version booléenne, qui ne permet que les variables d'entrée du logiciel, ces probabilités peuvent porter sur une expression quelconque.

Enfin, nous présentons dans la section 4.4 un exemple d'utilisation conjointe, dans une même spécification, de l'ensemble de ces techniques de test.

## 4.2 Spécification de l'environnement

### 4.2.1 Structure syntaxique

Syntaxiquement, la spécification d'un générateur de test se fait toujours dans un noeud de test (figure 4.2) et les invariants de l'environnement $I_1, ..., I_n$ sont spécifiées à l'aide de l'opérateur *environment*. A la différence de la version booléenne de LUTESS, des variables de type numérique peuvent être spécifiées comme paramètre d'entrée, paramètre de sortie ou variable locale. Les différentes expressions peuvent exprimer des relations entre ces variables numériques à l'aide des opérateurs arithmétiques et de comparaison du langage LUSTRE.

```
testnode Environnement(<sorties du logiciel>)
    returns (<entrées du logiciel>)
var
    <variables locales>
let
    <définition des variables locales>
    environment(I_1,...,I_n) ;
tel
```

FIGURE 4.2 – Structure syntaxique d'un noeud de test

```
testnode Env_clim(En_marche : bool ; Tsort : int)
    returns(Bouton : bool ; Tamb, Tutil : int) ;
var dTamb : int ;
let
    dTamb = 0 -> Tamb - pre Tamb ;                                    (1)
    environment(
        Tamb >= -20 and Tamb <= 60,                                   (2)
        Tutil >= 10 and Tutil <= 40,                                  (3)
        dTamb >= -1 and dTamb <= 1                                    (4)
    ) ;
tel
```

FIGURE 4.3 – Description de l'environnement du climatiseur

### 4.2.2 Illustration

Reprenons l'exemple du climatiseur numérique vu précédemment (fig. 4.1). Le noeud de test de la figure 4.3 spécifie 3 propriétés invariantes :

- La température ambiante *Tamb* doit toujours être comprise entre -20 et 60 degrés (2) ;
- La température choisie par l'utilisateur *Tutil* est comprise entre 10 et 40 degrés (3) et
- La température ambiante varie d'au plus 1 degré entre deux instants successifs (4).

Notons l'utilisation de la variable locale *dTamb* pour définir la différence de la température ambiante entre l'instant courant *Tamb* et l'instant précédent *pre Tamb* (1). Par la suite, cette variable est utilisée pour spécifier la propriété invariante $-1 \leq dTamb \leq 1$.

Les contraintes contenues dans l'opérateur *environment* sont des propriétés invariantes que les entrées du logiciel doivent vérifier à chaque instant. Chacune des

affectations des variables d'entrée satisfaisant ces contraintes constitue un *vecteur d'entrées valide*. Une séquence de test qui respecte l'environnement est une suite de vecteurs d'entrées valides à l'état courant.

Dans le tableau 4.1, nous montrons un exemple de séquence de test respectant la spécification de test de la fig. 4.3. Dans cette séquence, les températures *Tamb* et *Tutil* sont comprises dans les intervalles $(-20, 60)$ et $(-10, 40)$ respectivement. De même, on peut remarquer que la température *Tamb* change au plus d'un degré entre deux instants successifs.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ |
|---|---|---|---|---|---|---|---|---|---|
| Bouton | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| Tamb | **-12** | **-13** | **-14** | **-14** | **-14** | **-14** | **-15** | **-15** | **-14** |
| Tutil | **16** | **17** | **30** | **24** | **25** | **33** | **21** | **24** | **34** |
| En_marche | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| Tsort | 25 | 27 | 44 | 36 | 38 | 48 | 33 | 37 | 50 |

TABLE 4.1 – Une séquence de test respectant l'environnement numérique

## 4.3 Guidage de la génération

La modélisation d'un générateur de données de test qui respectent les contraintes d'environnement permet de limiter le domaine des tests générés à des situations valides, en excluant celles dans lesquelles le logiciel n'est jamais censé se trouver pendant son exécution. Or, un testeur humain peut avoir un objectif précis qui rend certaines situations valides plus intéressantes que d'autres. Le guidage de la génération de tests est introduite pour permettre au générateur de prendre en compte ces objectifs.

La technique de guidage à laquelle nous nous intéresserons en premier considère que le testeur souhaite valider la capacité du logiciel à respecter des propriétés de sûreté (paragraphe 4.3.1). Cette technique est renforcée par l'introduction d'hypothèses de test (paragraphe 4.3.2) qui permettent d'utiliser toute connaissance sur le logiciel pour améliorer la pertinence du guidage. Enfin, nous montrerons l'utilisation de probabilités pour simuler des comportements plus réalistes de l'environnement (paragraphe 4.3.3).

### 4.3.1 Guidage par les propriétés de sûreté

Cette technique de test vise à générer les tests qui sont les plus susceptibles de détecter la violation d'une propriété de sûreté. Pour faire cela, nous cherchons les entrées qui mènent le logiciel vers une situation suspecte, en évitant celles qui à l'évidence, ne peuvent pas mener à une violation de la propriété de sûreté spécifiée. Cette approche de guidage étant défini pour les logiciels booléens, nous souhaitons étudier comment elle peut être adaptée dans le nouveau contexte comprenant des spécification de type numérique.

#### 4.3.1.1 Structure syntaxique

Une propriété de sûreté peut être définie dans un noeud de test à l'aide du mot clé *safeprop*[2] (fig. 4.4). Comme pour l'opérateur *environment*, l'utilisateur peut utiliser des variables locales pour exprimer $P_S$. Une seule occurrence de l'opérateur *safeprop* est autorisée dans un noeud de test en admettant qu'il peut contenir la conjonction de plusieurs propriétés.

```
testnode Environnement(<sorties du logiciel>)
    returns (<entrées du logiciel>)
var
    <variables locales>
let
    <définition des variables locales>
    environment(I_1,...,I_n) ;
    safeprop(P_S) ;
tel
```

FIGURE 4.4 – Spécification de propriétés de sûreté

A la différence des propriétés invariantes qui s'expriment en fonction des paramètres d'entrée et des valeurs précédentes seulement, une propriété de sûreté s'exprime aussi en fonction des paramètres de sortie à l'instant courant qui ne sont pas connus au moment de la génération.

2. *De l'anglais "**safe**ty **prop**erty"*

#### 4.3.1.2 Illustration

Nous reprenons l'exemple du climatiseur afin d'illustrer la syntaxe utilisée dans le cadre du guidage par les propriétés de sûreté. Considérons la propriété $P_S = En\_marche \wedge Tamb < Tutil \implies Tsort > Tamb$, qui signifie que si le climatiseur est en marche et il fait froid (la température ambiante *Tamb* est plus basse que celle choisie par l'utilisateur *Tutil*) alors le climatiseur chauffe nécessairement la pièce (la température de l'air qui sort du climatiseur *Tsort* est plus élevée que *Tamb*).

Le noeud de test correspondant à un générateur avec guidage par cette propriété de sûreté est spécifié dans l'exemple de la fig. 4.5.

```
testnode Env_clim(En_marche : bool; Tsort : int)
    returns(Bouton : bool; Tamb, Tutil : int);
var dTamb : int;
let
    dTamb = 0 -> Tamb - pre Tamb;
    environment(
        Tamb >= -20 and Tamb <= 60,
        Tutil >= 10 and Tutil <= 40,
        dTamb >= -1 and dTamb <= 1
    );
    safeprop(implies(En_marche and Tamb<Tutil, Tsort>Tutil));
tel
```

FIGURE 4.5 – Guidage par propriétés de sûreté pour le climatiseur

L'objectif est de tester le logiciel en favorisant les situations dans lesquelles cette propriété peut ne pas être vérifiée. Puisque *En_ marche* et *Tsort* sont des sorties du logiciel, nous ne pouvons pas contrôler leurs valeurs et toute sortie peut être émise. Par contre, nous pouvons générer de telles valeurs pour *Tamb* et *Tutil* pour qu'il vérifient *Tamb*$<$*Tutil*. A l'évidence, on ne peut jamais violer la propriété $P_S$ en ayant *Tamb*$\geq$*Tutil*. Le tableau 4.2 montre une séquence de test respectant cette spécification.

#### 4.3.1.3 Sémantique de l'opérateur *safeprop*

L'opérateur *safeprop* permet de spécifier une propriété quelconque que le logiciel est censé respecter. L'idée du guidage est de chercher les situations ou cette propriété est violée.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ |
|---|---|---|---|---|---|---|---|---|---|
| Bouton | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| Tamb | **8** | **7** | **6** | **5** | **4** | **4** | **4** | **5** | **6** |
| Tutil | **16** | **13** | **34** | **11** | **25** | **14** | **38** | **14** | **27** |
| En_marche | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| Tsort | 18 | 15 | 43 | 13 | 32 | 17 | 49 | 17 | 34 |

TABLE 4.2 – Séquence de valeurs générées en utilisant le guidage par des propriétés de sûreté

Dans le cas basique, on cherche simplement à rendre vraie la négation de la propriété. Considérons la négation de la propriété $P_S$ de l'exemple précédent :

$$\neg P_S = En\_marche \wedge (Tamb < Tutil) \wedge (Tsort \leq Tamb)$$

Puisque le test s'effectue en boîte noire, il est supposé que toutes les valeurs pour les sorties *En_ marche* et *Tsort* sont possibles. En contraignant les entrées à satisfaire $\neg P_S$, nous imposons la condition $Tamb < Tutil$ portant seulement sur les entrées qui sont générées.

Dans d'autre cas, en plus des variables à l'instant courant, une propriété de sûreté peut faire référence aussi aux valeurs passées de ces variables. C'est pour cela que pour pouvoir violer une propriété, il faut être capable de construire automatiquement le passé nécessaire, afin d'amener le logiciel dans une situation où il est susceptible de violer la propriété de sûreté. Considérons une propriété simple : $i_{t-1} \Rightarrow o_t$ ou $i$ et $o$ sont une entrée et une sortie du logiciel, respectivement. Pour pouvoir violer cette propriété à l'instant $t$, il faut qu'à l'instant $t-1$ nous ayons générée la valeur *vrai* pour l'entrée $i$.

### 4.3.2 Prise en compte d'hypothèses sur le programme

Le guidage par les propriétés de sûreté tel qu'il est défini plus haut présente un inconvénient important : le programme sous test étant considéré comme une boîte noire, certaines réactions du logiciel considérés possibles peuvent, en réalité, ne jamais se produire. En effet, ne disposant pas de description ou de spécification du programme sous test (boîte noire) nous sommes obligés de considérer que ce dernier peut produire beaucoup plus de valeurs de sortie qu'il n'en est réellement capable.

L'extension que nous proposons ici vise à introduire dans le processus de génération des hypothèses sur le programme sous test. Le sens qu'on donne à une hypothèse est celle d'une connaissance partielle sur le programme. Ces hypothèses peuvent être de nature différente : elles peuvent être issues d'une analyse du programme, dans le cas où le code source ou une spécification sont disponibles, être issues de l'analyse des résultats de tests préalablement effectués (qui ont montré la validité de certaines propriétés) ou bien être le simple fruit de l'expérience du concepteur des tests. Quelle que soit leur origine, ces hypothèses peuvent fournir une spécification partielle du programme sous test et leur exploitation peut améliorer la pertinence des vecteurs de test engendrés pour le test guidé par les propriétés. En effet, au moment de la sélection de ces vecteurs, le générateur peut tenir compte de ces hypothèses et exclure ceux qui, bien qu'initialement considérés comme pouvant mener à une situation suspecte, donnent lieu à une réaction du programme rendant la violation des propriétés impossible.

#### 4.3.2.1 Structure syntaxique

L'opérateur *hypothesis* permet de spécifier un ensemble d'hypothèses de test sous la forme d'une liste d'expressions LUSTRE.

```
testnode Environnement(<sorties du logiciel>)
    returns (<entrées du logiciel>)
var
    <variables locales>
let
    <définition des variables locales>
    environment(I_1,...,I_n) ;
    safeprop(P_S) ;
    hypothesis(H_1,...,H_m) ;
tel
```

FIGURE 4.6 – Introduction d'hypothèses

#### 4.3.2.2 Illustration

Dans le tableau 4.2, nous avons montré une séquence qui vise la violation de la propriété de sûreté $En\_marche \wedge Tamb < Tutil \Rightarrow Tsort > Tamb$. Les entrées générées tentent de rendre vraie la partie gauche de l'implication, condition pour

que la propriété de sûreté soit violée. Or, *En_ marche* est une sortie du logiciel et le générateur ne peut pas agir sur elle puisqu'il n'a aucune connaissance sur le lien entre les entrées et la sortie *En_ marche*.

Supposons maintenant que le générateur ait connaissance du fait qu'appuyer sur *Bouton* change la valeur de *En_ marche*. Cette hypothèse introduit une relation entre l'entrée *Bouton* et la sortie *En_ marche*, donnant ainsi la possibilité de contrôler la valeur de la sortie *En_ marche* par l'intermédiaire de *Bouton*. Dans la figure 4.7 nous spécifions cette propriété comme une hypothèse de test.

```
testnode Env_clim(En_marche : bool ; Tsort : int)
    returns(Bouton : bool ; Tamb, Tutil : int) ;
var dTamb : int ;
let
    dTamb = 0 -> Tamb - pre Tamb ;
    environment(
        Tamb >= -20 and Tamb <= 60,
        Tutil >= 10 and Tutil <= 40,
        dTamb >= -1 and dTamb <= 1
    ) ;
    safeprop(implies(En_marche and Tamb<Tutil, Tsort>Tutil)) ;
    hypothesis( true -> (Bouton = En_marche<>pre(En_marche)) ) ;
tel
```

FIGURE 4.7 – Introduction d'hypothèses sur le climatiseur

Le tableau 4.3, montre un exemple de séquence de test qui respecte cette spécification. On peut remarquer que *Bouton* est appuyé seulement une fois lorsque le climatiseur n'est pas en marche afin de produire une séquence avec *En_ marche=true*. La violation de la propriété de sûreté ne dépend alors que de la valeur de *Tsort*.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ |
|---|---|---|---|---|---|---|---|---|---|
| Bouton | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| Tamb | 13 | 14 | 13 | 12 | 11 | 12 | 11 | 10 | 10 |
| Tutil | 36 | 24 | 33 | 28 | 26 | 32 | 25 | 17 | 22 |
| En_marche | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| Tsort | 43 | 27 | 39 | 33 | 31 | 38 | 29 | 19 | 26 |

TABLE 4.3 – Illustration du guidage par des propriétés de sûreté avec prise en compte d'hypothèses

### 4.3.3 Génération guidée par les probabilités conditionnelles

Ce type de guidage de la génération vise à simuler plus précisément certains comportements de l'environnement du logiciel, en permettant à l'utilisateur de définir des profils d'utilisation du logiciel. Ces profils peuvent être définis par l'introduction dans une spécification d'une ou plusieurs probabilités conditionnelles.

La version booléenne de LUTESS permet de définir la probabilité qu'une entrée booléenne ait la valeur vraie, quand une condition est vérifiée. Nous souhaitons donner la possibilité à l'utilisateur de spécifier cette probabilité, non pas seulement sur les entrées, mais sur une expression quelconque.

#### 4.3.3.1 Structure syntaxique

Les probabilités conditionnelles sont définies à l'intérieur d'un noeud de test à l'aide du mot-clé *prob*. La forme générale de la syntaxe utilisé est donné dans la fig. 4.8.

```
testnode Environnement(<sorties du logiciel>)
    returns (<entrées du logiciel>)
var
    <variables locales>
let
    <invariants>
    <propriété de sûreté>
    <hypothèses>
    prob(C1,E1,P1) ;
    ...
    prob(Cm,Em,Pm) ;
    <définition des variables locales>
tel
```

FIGURE 4.8 – Spécification de probabilités

Ainsi, une probabilité est définie par une expression *prob(C,E,P)* où :

- $C$ est une condition portant sur les valeurs passées des paramètres d'entrée/sortie,
- $E$ est une expression LUSTRE à résultat booléen,
- $P$ est une constante réelle dans l'intervalle (0.0..1.0).

La signification d'une telle probabilité est : Si la condition $C$ est vérifiée ($C$=*vrai*) alors il y a une probabilité $P$ pour que $E$ soit aussi vraie. Dans le cas ou la condition

n'est pas vérifiée, cette probabilité ne produit aucun effet.

#### 4.3.3.2 Illustration

Dans l'exemple du climatiseur, l'entrée *Bouton* permet de mettre en marche ou d'arrêter le climatiseur. L'affectation de valeurs aléatoires à *Bouton* amène le climatiseur à passer d'un état de marche à un état d'arrêt, et inversement, trop souvent, si bien qu'on ne peut jamais observer une exécution réaliste, i.e. avec des longues sous-séquences où le climatiseur est en marche. La première spécification de probabilité conditionnelle de l'exemple de la figure 4.9, diminue la probabilité que l'utilisateur appuie sur le bouton dans le cas ou le climatiseur est en marche. La deuxième probabilité conditionnelle exprime le fait que si le climatiseur est en marche et la température de l'air émis est supérieure à la température ambiante, alors cette dernière va probablement augmenter (dans 80% des cas).

```
testnode Env_clim(En_marche :bool ; Tsort : int)
    returns(Bouton : bool ; Tamb, Tutil : int) ;
var dTamb : int ;
let
    dTamb = 0 -> Tamb - pre T ;
    environment(
        Tamb >= -20 and Tamb <= 60,
        Tutil >= 10 and Tutil <= 40,
        dTamb >= -1 and dTamb <= 1
    ) ;
    prob(false -> pre En_marche, Bouton, 0.1) ;
    prob(false -> pre(En_marche and Tsort > Tamb),
        true -> (Tamb > pre Tamb), 0.8) ;
tel
```

FIGURE 4.9 – Description de l'environnement du climatiseur avec des probabilités

Le tableau 4.4 ci-dessous, montre une séquence de test respectant la spécification de la figure 4.9. On peut remarquer que le bouton n'est pas appuyé souvent lorsque le climatiseur est en marche, ce qui permet d'obtenir des sous-séquences plus longues avec *En_ marche=vrai* (de $t_1$ à $t_{11}$ dans ce cas). Dans ces conditions, on pourra tester d'autres comportements du contrôleur qui ne sont effectifs que quand le climatiseur est en marche. De même, grâce à la deuxième probabilité conditionnelle, on peut remarquer que lorsque le climatiseur est en marche et *Tsort* est supérieur à *Tamb*,

la température ambiante généralement augmente (instants $t_2 - t_5, t_8, t_{10} - t_{12}$) alors que $t_9$ montre que cette augmentation n'est pas systématique.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ | $t_9$ | $t_{10}$ | $t_{11}$ | $t_{12}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bouton | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Tamb | 14 | 14 | 15 | 16 | 17 | 18 | 17 | 18 | 19 | 18 | 19 | 20 | 19 |
| Tutil | 37 | 28 | 33 | 27 | 27 | 15 | 14 | 27 | 28 | 36 | 37 | 25 | 26 |
| En_marche | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| Tsort | 44 | 32 | 39 | 30 | 30 | 14 | 13 | 30 | 31 | 42 | 43 | 26 | 28 |

TABLE 4.4 – Une séquence de test respectant les probabilités

## 4.4 Combinaisons des techniques de test

Nous avons présentées séparément les techniques de guidage par rapport aux propriétés de sûreté et par les probabilité conditionnelles. Mais, du point de vue méthodologique, rien n'empêche l'utilisation de ces deux techniques simultanément. Ainsi, on peut spécifier des probabilités à respecter par les données générées, même si la génération est guidée par rapport à une propriété de sûreté. De cette façon, le testeur à une plus grande liberté pour spécifier plus précisément le comportement auquel il s'intéresse.

L'exemple de spécification de la figure 4.10 illustre l'utilisation conjointe de toutes ces techniques dans une même spécification :

- L'opérateur *environment* définit les propriété invariantes $Tamb_t \in (-20, 60)$, $Tutil_t \in (-10, 40)$ et $-1 \leq Tamb_t - Tamb_{t-1} \leq 1,\ t > 0$ ;
- Le test est guidé par la propriété de sûreté : $En\_marche_t \wedge Tamb_t < Tutil_t \Rightarrow Tsort_t > Tamb_t$ ;
- L'hypothèse $Bouton_t \Leftrightarrow En\_marche_t \neq En\_marche_{t-1},\ t > 0$ introduit le fait que l'appui sur le *Bouton* change la valeur de *En_ marche* entre deux instants ;
- Et enfin, à l'aide de l'opérateur *prob* on introduit le fait que si le climatiseur émettait de la chaleur ($Tsort_{t-1} > Tamb_{t-1},\ t > 0$) alors probablement (dans 80% des cas) la température va augmenter ($Tamb_t > Tamb_{t-1},\ t > 0$) et son corollaire quand le climatiseur émet du froid.

Le tableau 4.5 montre une séquence de test respectant cette spécification qui uti-

```
testnode Env_clim(En_marche :bool ; Tsort : int)
    returns(Bouton : bool ; Tamb, Tutil : int) ;
var dTamb : int ;
let
    dTamb = 0 -> Tamb - pre Tamb ;
    environment(
        Tamb >= -20 and Tamb <= 60,
        Tutil >= 10 and Tutil <= 40,
        dTamb >= -1 and dTamb <= 1
    ) ;
    safeprop(implies(En_marche and Tamb<Tutil, Tsort>Tutil)) ;
    hypothesis( true -> (Bouton = En_marche<>pre(En_marche)) ) ;
    prob(false -> pre(Tsort>Tamb), true -> (Tamb>pre Tamb), 0.8) ;
    prob(false -> pre(Tsort<Tamb), true -> (Tamb<pre Tamb), 0.8) ;
tel
```

FIGURE 4.10 – Une spécification avec plusieurs techniques de test

lise plusieurs techniques de guidage. On peut remarquer que, les températures sont dans les intervalles spécifiés par les invariants de l'environnement, et que la température ambiante entre deux instants successifs ne change pas de plus d'un degré. De même on peut facilement remarquer que, le guidage par propriété de sûreté, fait que la température ambiante est inférieure à celle choisie par l'utilisateur et en conjonction avec l'hypothèse de test, le climatiseur reste en marche plus longtemps. De plus, due à la première probabilité, quand la température de sortie est plus grande que celle ambiante, la température ambiante généralement augmente. Remarquons que la condition de la deuxième probabilité ($Tsort<Tamb$) n'est jamais vérifiée à cause de l'effet de la propriété de sûreté qui ne pourrait pas être violée dans ce cas et cette probabilité n'a aucun effet sur la séquence de test générée.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ | $t_9$ | $t_{10}$ | $t_{11}$ | $t_{12}$ | $t_{13}$ | $t_{14}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bouton | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Tamb | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| Tutil | 15 | 24 | 21 | 20 | 21 | 27 | 22 | 33 | 35 | 24 | 33 | 37 | 35 | 35 | 29 |
| En_marche | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Tsort | 15 | 27 | 22 | 21 | 22 | 29 | 22 | 37 | 39 | 24 | 36 | 41 | 38 | 38 | 29 |

TABLE 4.5 – Exemple d'une séquence utilisant plusieurs techniques de guidage

## 4.5 Conclusion

Dans ce chapitre nous avons présenté les extensions que nous apportons à l'approche de test fonctionnel automatique à l'aide de l'outil LUTESS. Nous prenons en compte les logiciel synchrones qui ont des sorties numériques et introduisons de nouveaux éléments dans le langage de spécification de test. En plus des propriétés invariantes de l'environnement du logiciel sous test, on peut guider la génération vers des objectifs de test plus précis avec les propriétés de sûreté. L'introduction d'hypothèses permet d'utiliser les connaissances sur le logiciel pour améliorer le guidage. Enfin, les données peuvent être générées selon des probabilités spécifiées.

L'ensemble de ces techniques de test, ainsi que leur utilisation conjointe dans la même spécification, est censé constituer un outil pour la spécification formelle de générateurs de données de test. Le chapitre suivant montre comment à partir de ces spécifications on produit des données de test d'une manière automatique.

# Chapitre 5

# Génération de tests à l'aide de contraintes

Nous présentons dans ce chapitre notre approche pour la génération automatique de données de test à partir des spécifications introduites dans le chapitre précédent. Cette approche est basée sur les principes de la Programmation par Contraintes.

## 5.1 Propositions

L'objectif principal est la génération de données de test pour des logiciels synchrones comprenant des entrées/sorties numériques. Les spécifications pour ces logiciels comprennent, en plus des formules booléennes, des relations de comparaison et des calculs sur des données numériques. La génération de données de test qui sattisfont ces spécifications nécessite donc un formalisme de représentation adéquat pour ces relations. Nous proposons de les modéliser par des contraintes et d'utiliser les méthodes de résolution associées pour éffectuer la génération de tests.

Nous commençons, dans la section 5.2, par la définition d'un modèle formel pouvant représenter une spécification de test. Tout naturellement, ce modèle reste celui d'une machine à états finis dont les états représentent la mémoire. Chacune des expressions contenues dans les opérateurs de test peut alors être représentée en fonction des variables d'état et les paramètres d'entrée et de sortie du logiciel sous test. La transformation d'une spécification en machine à états finis est définie dans le paragraphe 5.2.1. Par la suite, cette machine est représentée par un ensemble de contraintes équivalentes. Les principes de cette représentation sont donnés dans le paragraphe 5.2.2.

Dans la section 5.3, nous montrons comment les contraintes de cette machine sont utilisées pour la génération de données de test. Une séquence de test consiste à parcourir la machine à états finis et correspond à un chemin partant de l'état initial. Les transitions entre les états sont définies en fonction des entrées qui sont générées et les sorties correspondantes, calculées par le logiciel. A chaque état, les entrées pouvant être produites sont conditionnées par un ensemble de contraintes correspondant aux différentes directives de test. Un algorithme de génération d'une séquence de test qui respecte les invariants de l'environnement, réalisé à l'aide de contraintes, est donné dans le paragraphe 5.3.1. Le guidage par rapport aux propriétés de sûreté, l'algorithme de génération ainsi que les contraintes permettant ce guidage sont définis dans le paragraphe 5.3.2. Une nouvelle définition de ce guidage, permettant la prise en compte des hypothèses de test, est présentée dans le paragraphe 5.3.3. Enfin, le paragraphe 5.3.4 complète notre algorithme de génération avec l'introduction des probabilités conditionnelles.

Une fois que l'ensemble des contraintes à respecter a été défini, il faut affecter des valeurs aux variables d'entrée du logiciel tout en respectant l'ensemble de ces contraintes. Les différentes techniques pour choisir ces valeurs sont développées dans la section 5.4.

## 5.2 Modélisation d'un générateur de test

Comme pour chaque programme écrit en LUSTRE, on peut associer à un noeud de test une machine à états finis. Les états de cette machine sont constitués de la mémoire du logiciel, introduite par l'opérateur *pre* qui mémorise la valeur d'un flux à l'instant précédent.

### 5.2.1 Représentation par une machine à états finis

Pour transformer une spécification en une machine d'états finis, on associe une variable d'état $q_i$ à chaque opérateur *pre*. Une variable supplémentaire $q_0$ permet de distinguer l'état initial, introduit par l'opérateur *"suivi de"* ->.

**Définition 5.2.1** *Machine à états finis associée à une spécification*
*Ainsi, on obtient la machine à état finis* $M = (Q, q_{init}, E, S, trans)$ *où :*
*–* $Q$ *est l'ensemble des états de la machine ;*

- $q_{init} \in Q$ *est l'état initial;*
- $E$ *est l'ensemble des valeurs des paramètres d'*entrée *du programme sous test;*
- $S$ *est l'ensemble des valeurs des paramètres de* sortie *du programme sous test;*
- $trans : Q \times E \times S \rightarrow Q$ *est la fonction de transition calculant les nouvelles valeurs des variables d'état. Elle peut être présenté par un vecteur de fonctions* $(trans_i)_{i=0,n}$*, ou chaque fonction* $trans_i$ *calcule la nouvelle valeur de la variable d'état* $q_i$*.*

#### 5.2.1.1 Machine génératrice associée à une spécification

En remplaçant chacune des expressions *pre* $E$ par les variables d'états correspondantes, toutes les expressions à l'intérieur d'un noeud de test peuvent s'exprimer en fonction des variables d'état et des paramètres d'entrée et de sortie du logiciel à l'instant courant. On peut représenter alors les expressions à l'intérieur des opérateurs spécifiques au noeud de test par des fonctions à résultat booléen.

**Représentation des invariants de l'environnement**

Soit `environment(`$I_1, I_2, ..., I_n$`)` la définition de $n$ invariants d'environnement. Chacune des ces expressions ne dépend que des variables d'état et des paramètres d'entrée du logiciel. On peut les représenter comme un vecteur $(env)_{i=1,n}$ de $n$ fonctions à résultat booléen telles qu'une fonction $env_i : Q \times E \rightarrow \{0, 1\}$ est associée à chaque invariant $I_i$. Puisque nous nous intéressons à la satisfaction de l'ensemble des propriétés invariantes, nous définissons leur conjonction par $env = env_1 \wedge env_2 \wedge ... \wedge env_n$.

**Représentation de la propriété de sûreté**

Si une propriété de sûreté est spécifiée avec l'expression `safeprop(`$P_S$`)`, elle peut être représentée par une fonction $prop : Q \times E \times S \rightarrow \{0, 1\}$. A la différence des invariants de l'environnement, en plus des variables d'état et les paramètres d'entrée, une propriété de sûreté peut dépendre aussi des paramètres de sortie du logiciel sous test.

**Représentation des hypothèses**

Soit `hypothesis(`$H_1, H_2, ..., H_m$`)` la définition de $m$ hypothèses sur le comportement du logiciel. De manière semblable aux invariants, on peut représenter les hypothèses comme un vecteur $(hyp_i)_{i=1,m}$ de $m$ fonctions à résultats booléen telles

qu'une fonction $hyp_i : Q \times E \times S \rightarrow \{0,1\}$ est associée à chaque hypothèse $H_i$. Nous définissons la conjonction de toutes ces hypothèses comme $hyp = hyp_1 \wedge ... \wedge hyp_m$.

#### Représentation des probabilités

Une série de $v$ probabilités conditionnelles $prob(C_1, E_1, P_1)$ ; ... $prob(C_v, E_v, P_v)$ ; peut être représenté par un vecteur de triplets $((c_i, e_i, p_i))_{i=1,v}$ tels que :

- $c_i : Q \rightarrow \{0,1\}$ représente la condition $C_i$ de la probabilité
- $e_i : Q \times E \rightarrow \{0,1\}$ représente l'expression $E_i$
- $p_i \in \mathbb{R} \wedge p_i \in (0..1)$ est la probabilité pour que l'expression $E_i$ soit vraie.

#### Machine génératrice

En ajoutant les fonctions correspondant aux opérateurs de test à la machine à états finis, on obtient une nouvelle machine.

**Définition 5.2.2** *Machine génératrice*
*Une machine génératrice est définie comme*
$M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ *où :*
- $Q$ *est l'ensemble des états de la machine ;*
- $q_{init} \in Q$ *est l'état initial ;*
- $E$ *est l'ensemble des valeurs des paramètres d'*entrée *du programme sous test ;*
- $S$ *est l'ensemble des valeurs des paramètres de* sortie *du programme sous test ;*
- $trans : Q \times E \times S \rightarrow Q$ *est la fonction de transition.*
- $env : Q \times E \rightarrow \{0,1\}$ *est la fonction représentant la conjonction des invariants de l'environnement ;*
- $prop : Q \times E \times S \rightarrow \{0,1\}$ *représente la propriété de sûreté éventuelle ;*
- $hyp : Q \times E \times S \rightarrow \{0,1\}$ *représente la conjonction des hypothèses et*
- $prob = ((c_i, e_i, p_i))_{i=1,v}$ *représente la définition de v probabilités dans la spécification avec* $c_i : Q \rightarrow \{0,1\}, e_i : Q \times E \rightarrow \{0,1\}$ *et* $p_i \in \mathbb{R} \wedge p_i \in (0..1)$.

### 5.2.1.2 Illustration avec l'environnement du climatiseur

Nous reprenons l'exemple du climatiseur (fig. 5.1) du chapitre précédent pour illustrer comment on peut représenter un noeud de test sous la forme d'une machine à états finis.

```
testnode Env_clim(En_marche :bool ; Tout : int)
        returns(Bouton : bool ; Tamb, Tutil : int) ;
var dTamb : int ;
let
    dTamb = 0 -> Tamb - pre Tamb ;
    environment(
        Tamb >= -20 and Tamb <= 60,
        Tutil >= 10 and Tutil <= 40,
        dTamb >= -1 and dTamb <= 1
    ) ;
    safeprop( implies(En_marche and Tamb<Tutil, Tsort>Tutil) ) ;
    hypothesis( true -> Bouton = En_marche<>pre(En_marche) ) ;
    prob(false -> pre(Tsort>Tamb), true -> (Tamb>pre Tamb), 0.8) ;
    prob(false -> pre(Tsort<Tamb), true -> (Tamb<pre Tamb), 0.8) ;
tel
```

FIGURE 5.1 – Description de l'environnement du climatiseur

Dans la figure 5.2, nous montrons un noeud de test dans lequel on introduit des variables locales pour renommer les flux correspondant à l'utilisation de l'opérateur *pre*. Une variable supplémentaire *q0* permet de définir l'état initial. Ainsi, toute expression du type `E->F` peut être écrite par `if q0 then E else F`.

```
testnode Env_clim(En_marche : bool ; Tsort : int)
        returns(Bouton : bool ; Tamb, Tutil : int) ;
var dTamb : int ;
q0,q2,q3,q4 : bool ;
q1 : int ;
let
    q0 = true -> false ;
    q1 = pre Tamb ;
    q2 = pre En_marche ;
    q3 = pre(Tsort>Tamb) ;
    q4 = pre(Tsort<Tamb) ;
    dTamb = if q0 then 0 else Tamb - q1 ;
    environment(
        Tamb >= -20 and Tamb <= 60,
        Tutil >= 10 and Tutil <= 40,
        dTamb >= -1 and dTamb <= 1
    ) ;
    safeprop( implies(En_marche and Tamb<Tutil, Tsort>Tutil) ) ;
    hypothesis( if q0 then true else (Bouton = En_marche<>q2) ) ;
    prob(if q0 then false else q3, if q0 then true else Tamb>q1,
0.8) ;
    prob(if q0 then false else q4, if q0 then true else Tamb<q1,
0.8) ;
tel
```

FIGURE 5.2 – Description de l'environnement du climatiseur après réécriture

La spécification que nous obtenons en renommant des expressions est sémantiquement équivalente à la précédente, mais il montre clairement les variables d'état de la machine à états finis ainsi que la fonction de transition associée :

- Les états de la machine correspondante sont déterminés par les valeurs des variables d'état $q_0, ..., q_4$, l'état initial étant déterminé par la condition $q_0 = 1$.
- La fonction de transition découle directement de l'expression définissant chacune des variables d'état. Puisque chacune des variables fait référence a la valeur précédente d'une expression, à l'état suivant elles auront pour valeur celle de l'expression à l'état courant.
- De même, on peut remarquer que toutes les expressions définies à l'intérieur du noeud de test, s'expriment en fonction seulement des variables d'état et des paramètres d'entrée et de sortie.

La figure 5.3 donne la machine complète correspondant à cet exemple.

```
inputs
    Bouton : bool ; Tamb, Tutil : int ;
outputs
    En_marche : bool ; Tsort : int ;
state
    q0 : bool ; q1 : int ; q2,q3,q4 : bool ;
initial
    q0 = true ;
var
    dTamb : int ;
local
    dTamb = if q0 then 0 else Tamb - q1 ;
transition
    q0' = false ;
    q1' = Tamb ;
    q2' = En_marche ;
    q3' = Tsort>Tamb ;
    q4' = Tsort<Tamb ;
environment
    Tamb>=-20 and Tamb<=60
    and
    Tutil>=10 and Tutil<=40
    and
    dTamb>=-1 and dTamb<=1 ;
safeprop
    not(En_marche and Tamb<Tutil) or (Tsort>Tutil) ;
hypothesis
    if q0 then true else (Bouton = En_marche <> q2) ;
probabilities
    (c1, e1, p1) = (if q0 then false else q3,
        if q0 then true else Tamb>q1, 0.8) ;
    (c2, e2, p2) = (if q0 then false else q4,
        if q0 then true else Tamb<q1, 0.8) ;
```

FIGURE 5.3 – Représentation d'un *testnode* sous la forme d'une machine à états finis

### 5.2.2 Représentation d'une machine par des contraintes sur domaine fini

Nous souhaitons utiliser la programmation par contraintes pour générer des données à partir d'une machine génératrice. Dans cette optique, nous représentons d'abord cette machine par des contraintes, en associant un système de contraintes à chacune des fonctions de cette machine. Ces contraintes sont obtenues simplement par décomposition de l'arbre syntaxique de l'expression définissant la fonction. Les noeuds de l'arbre syntaxique sont constitués des opérateurs LUSTRE alors

que les feuilles sont des variables ou constantes. En associant à chaque opérateur la contrainte correspondante, on obtient un système de contraintes équivalent à l'expression.

#### 5.2.2.1 Machine génératrice sous forme de contraintes

On peut représenter une machine génératrice $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ par un ensemble de contraintes sur domaine fini, ou chacune de ses caractéristiques est représentée par une contrainte. Nous illustrons par la suite cette représentation en utilisant l'exemple précédent de l'environnement du climatiseur.

**Contraintes de type**

Les variables LUSTRE sont fortement typés et les valeurs que peuvent prendre les variables sont déterminées par leur type. Nous considérons seulement les variables booléennes (*bool*) et les entiers signés (*int*) ou non (*uint*). Les variables de type flottant nécessitent un solveur adéquat, même si les principes décrits à la suite restent valables.

Nous utilisons les contraintes suivantes pour définir le type des variables :

$$\begin{aligned} bool(x) &\Leftrightarrow x \in [0..1] \\ int(x) &\Leftrightarrow x \in [-2^{31}..2^{31}-1] \\ uint(x) &\Leftrightarrow x \in [0..2^{32}-1] \end{aligned}$$

**Paramètres d'entrée et de sortie**

La contrainte *inputs*, permet de spécifier un vecteur d'entrée comme un ensemble de variables d'entrée du logiciel ainsi que les contraindre à prendre seulement des valeurs permises par leur type. De même pour la contrainte *outputs*, qui spécifie les variables de sortie du logiciel.

```
inputs
    Bouton : bool ; Tamb, Tutil : int ;
```

$$inputs(e) \Leftrightarrow e = \{Bouton, Tamb, Tutil\} \wedge bool(Bouton) \wedge int(Tamb) \wedge int(Tutil)$$

```
outputs
    En_marche : bool ; Tsort : int ;
```

$$outputs(s) \Leftrightarrow s = \{En\_marche, Tsort\} \wedge bool(En\_marche) \wedge int(Tsort)$$

### Variables d'état et état initial

La contrainte *state* définit les variables d'état et le type de chacune d'entre elles.

```
state
    q0 : bool ; q1 : int ; q2,q3,q4 : bool ;
```

$$\begin{aligned} state(q) \Leftrightarrow q = \{&q_0, q_1, q_2, q_3, q_4\} \\ &\wedge bool(q_0) \wedge int(q_1) \wedge bool(q_2) \wedge bool(q_3) \wedge bool(q_4) \end{aligned}$$

La contrainte *initial* spécifie que la seule condition pour qu'un état soit initial est que la variable $q_0$ soit vraie. Cette condition sera la même pour toute spécification.

```
initial
    q0 = true ;
```

$$\begin{aligned} initial(q) \Leftrightarrow &state(q) \wedge q = \{q_0, q_1, q_2, q_3, q_4\} \\ &\wedge q_0 = 1 \end{aligned}$$

### Fonction de transition

La fonction de transition définit l'état suivant $q'$ en fonction de l'état courant $q$, des entrées $e$ et des sorties $s$ du logiciel. La contrainte *transition* définit cette relation par les contraintes correspondant à chacune des variables d'état.

```
transition
    q0' = false ;
    q1' = Tamb ;
    q2' = En_marche ;
    q3' = Tsort>Tamb ;
    q4' = Tsort<Tamb ;
```

$$
\begin{aligned}
transition(q, e, s, q') \Leftrightarrow\ & state(q) \wedge q = \{q_0, q_1, q_2, q_3, q_4\} \\
& \wedge inputs(e) \wedge e = \{Bouton, Tamb, Tutil\} \\
& \wedge outputs(s) \wedge s = \{En\_marche, Tsort\} \\
& \wedge state(q') \wedge q' = \{q'_0, q'_1, q'_2, q'_3, q'_4\} \\
& \wedge q'_0 = 0 \\
& \wedge q'_1 = Tamb \\
& \wedge q'_2 = En\_marche \\
& \wedge q'_3 = Tsort > Tamb \\
& \wedge q'_4 = Tsort < Tamb
\end{aligned}
$$

### Invariants de l'environnement

La contrainte *environment* définit les propriétés invariantes en fonction de l'état $q$ et des entrées $e$.

```
var
    dTamb : int ;
local
    dTamb = if q0 then 0 else Tamb - q1 ;
environment
    Tamb>=-20 and Tamb<=60
    and
    Tutil>=10 and Tutil<=40
    and
    dTamb>=-1 and dTamb<=1 ;
```

$$
\begin{aligned}
environment(q, e) \Leftrightarrow\ & state(q) \wedge q = \{q_0, q_1, q_2, q_3, q_4\} \\
& \wedge inputs(e) \wedge e = \{Bouton, Tamb, Tutil\} \\
& \wedge int(dTamb) \wedge dTamb = if\_then\_else(q_0, 0, Tamb - q_1) \\
& \wedge Tamb \geq -20 \wedge Tamb \leq 60 \\
& \wedge Tutil \geq 10 \wedge Tutil \leq 40 \\
& \wedge dTamb \geq -1 \wedge dTamb \leq 1
\end{aligned}
$$

Notons l'utilisation de la contrainte *if_ then_ else* pour représenter la construction fonctionnelle *if* du langage LUSTRE. La sémantique de cette contrainte est

définie par :

$$x = if\_then\_else(b, y, z) \Leftrightarrow (b \wedge x = y) \vee (\neg b \wedge x = z)$$

### Propriété de sûreté et hypothèses

Les contraintes définissant la propriété de sûreté (*safeprop*) et les hypothèses (*hypothesis*), sont obtenues de manière similaire aux invariants.

```
safeprop
    not(En_marche and Tamb<Tutil) or (Tsort>Tutil) ;
```

$$\begin{aligned} safeprop(q, e, s) \Leftrightarrow\ & state(q) \wedge q = \{q_0, q_1, q_2, q_3, q_4\} \\ & \wedge inputs(e) \wedge e = \{Bouton, Tamb, Tutil\} \\ & \wedge outputs(s) \wedge s = \{En\_marche, Tsort\} \\ & \wedge (\neg(En\_marche \wedge Tamb < Tutil) \vee (Tsort > Tutil)) \end{aligned}$$

```
hypothesis
    if q0 then true else (Bouton = En_marche <> q2) ;
```

$$\begin{aligned} hypothesis(q, e, s) \Leftrightarrow\ & state(q) \wedge q = \{q_0, q_1, q_2, q_3, q_4\} \\ & \wedge inputs(e) \wedge e = \{Bouton, Tamb, Tutil\} \\ & \wedge outputs(s) \wedge s = \{En\_marche, Tsort\} \\ & \wedge if\_then\_else(q_0, true, (Bouton = (En\_marche \neq q_2))) \end{aligned}$$

### Probabilités

Les probabilités s'expriment en fonction des variables d'état $q$ et des entrées $e$ seulement. La contrainte *probabilities* définit les conditions qui relient les différentes expressions de chacune des probabilités.

```
probabilities
    (c1, e1, p1) = (if q0 then false else q3,
        if q0 then true else Tamb>q1, 0.8);
    (c2, e2, p2) = (if q0 then false else q4,
        if q0 then true else Tamb<q1, 0.8);
```

$$
\begin{aligned}
probabilities(q, e, \{l_1, l_2\}) \Leftrightarrow\ & state(q) \wedge q = \{q_0, q_1, q_2, q_3, q_4\} \\
& \wedge inputs(e) \wedge e = \{Bouton, Tamb, Tutil\} \\
& \wedge l_1 = (c_1, e_1, p_1) \wedge c_1 = if\_then\_else(q_0, false, q_3) \\
& \wedge e_1 = if\_then\_else(q_0, true, Tamb > q_1) \wedge p_1 = 0.8 \\
& \wedge l_2 = (c_2, e_2, p_2) \wedge c_2 = if\_then\_else(q_0, false, q_4) \\
& \wedge e_2 = if\_then\_else(q_0, true, Tamb < q_2) \wedge p_2 = 0.8
\end{aligned}
$$

## 5.3 Génération de données de test à l'aide de contraintes

Dans la section 5.2, nous avons montré les principes de la génération des contraintes représentant la machine à états finis associée à une spécification de test. Nous utiliserons ces contraintes pour générer des données de test qui respectent cette même spécification.

### 5.3.1 Génération d'une séquence de test respectant les invariants

Une séquence de test correspond à un parcours de la machine à états finis en démarrant de l'état initial. Ce parcours constitue un chemin qui est une suite d'états liés par la fonction de transition. Les transitions possibles d'un état à l'autre sont conditionnées par la satisfaction de la spécification et la longueur $n$ d'un chemin correspond au nombre de transitions parcourues.

**Définition 5.3.1** *Chemin dans la machine génératrice*
*Un chemin de longueur $n$ issu de $q_0$ dans la machine $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ est une suite d'états $(q_0, ..., q_n) \in Q^{n+1}$ tel que :*

$$\exists (e_0, ..., e_{n-1}) \in E^n,\ \exists (s_0, ..., s_{n-1}) \in S^n \text{ telles que}$$
$$\forall i \in [0..n-1]\ env(q_i, e_i) \wedge q_{i+1} = trans(q_i, e_i, s_i)$$

Intuitivement, un chemin de longueur $n$ issu de l'état initial $q_{init}$ correspond à une séquence de test de longueur $n$ qui respecte les invariants de l'environnement.

#### 5.3.1.1 Principe de génération d'une séquence de test

Le principe de génération d'une séquence de test est défini par l'algorithme 5.1. La génération d'une séquence démarre à partir de l'état initial, en affectant un état initial à l'état courant (2). La génération d'une séquence de longueur $n$, consiste à générer $n$ fois des vecteurs d'entrées (3-4) respectant la spécification, effectuant autant de transitions dans la machine. La génération d'une donnée de test s'effectue en deux pas : la détermination de l'ensemble des entrées *valides* à l'état courant par rapport à la spécification $V \leftarrow \{x \in E \mid env(q,x)\}$(3) et le choix d'un élément dans cet ensemble à l'aide de la méthode *choose* (4). Une fois les données de test sélectionnées, elles sont envoyées au logiciel sous test (5) et les sorties, calculées par ce dernier, sont récupérées (6). Les entrées et les sorties sont ensuite utilisées pour calculer l'état suivant à l'aide de la fonction de transition (7).

**Algorithm 5.1** Algorithme de génération d'une séquence de test respectant les invariants de l'environnement

**paramètres**
$n \in \mathbb{N}$, $M_g = (Q, q_{init}, E, S, t, env, prop, hyp, prob)$ ;
**variables**
$q : Q$ ; $e : E$ ; $s : S$ ; $V \subseteq E$ (1)
**début**
$q \leftarrow q_{init}$ ; (2)
**faire** $n$ **fois**
$V \leftarrow \{x \in E \mid env(q,x)\}$ ; (3)
$e \leftarrow$`choose`$(V)$ ; (4)
`write`$(e)$ ; (5)
`read`$(s)$ ; (6)
$q \leftarrow trans(q,e,s)$ ; (7)
**finfaire** ;
**fin**.

#### 5.3.1.2 Réalisation de l'algorithme de génération avec des contraintes

Pour réaliser l'algorithme 5.1, nous utiliserons les contraintes associées à la machine génératrice générés précédemment :

- Pour contraindre l'état $q$ à prendre des valeurs dans $Q$ on poste la contrainte $state(q)$ et pour $e : E$ et $s : S$ on poste les contraintes $inputs(e)$ et $outputs(q)$ respectivement (1).
- L'état initial $q \leftarrow q_{init}$ (2) est déterminé par la contrainte $initial(q)$.

- L'ensemble des entrées $e$ qui respectent les invariants de l'environnement dans un état donné $q$ (3), peut être déterminé à l'aide de la contrainte $environment(q, e)$.
- La méthode `choose` va être déterminé par la méthode de recherche d'une solution du système à contraintes (4). Nous discuterons ceci plus en détail dans la section 5.4.
- La contrainte $transition(q, e, s, q')$ sera utilisée pour déterminer complètement $q'$ comme l'état suivant de $q$ quand $e$ et $s$ sont connus (7).

### 5.3.2 Guidage par les propriétés de sûreté

Le but de ce type de guidage est de générer des données de test qui favorisent la détection d'une violation pour une propriété de sûreté, spécifiée à l'aide de l'opérateur *safeprop*. L'idée de base de ce guidage consiste à éviter de générer, dans un état donné, les entrées qui rendent impossible la violation de cette propriété. Cependant, en plus des paramètres d'entrée et de sortie à l'instant courant, une propriété de sûreté peut faire référence aux valeurs passées de ces paramètres et donc dépendre de l'état. C'est pour cela que pour pouvoir violer une propriété, il faut être capable de construire automatiquement le passé nécessaire, afin d'amener le logiciel dans l'état voulu.

#### 5.3.2.1 États suspects de la machine génératrice

Informellement, le guidage par les propriétés de sûreté consiste à chercher les états dans lesquels une entrée valide peut être émise et tels qu'il existe au moins une sortie possible du logiciel qui viole la propriété de sûreté. On appelle un tel état, un *état suspect*.

**Définition 5.3.2** *État suspect*
*Soit* $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ *une machine génératrice. Un état* $q \in Q$ *de cette machine est dit* suspect *par rapport à la propriété de sûreté ssi*

$$\exists e \in E, \exists s \in S \text{ tel que } env(q, e) \wedge \neg prop(q, e, s)$$

La définition d'un état suspect permet de caractériser formellement une situation critique. En effet, dans un état suspect, il existe des entrées permises par les invariants de l'environnement telles que la propriété de sûreté ne dépend plus que

des sorties du logiciel sous test. Au contraire, dans un état non suspect, la propriété ne peut pas être violée. Pour pouvoir violer une propriété de sûreté, le chemin correspondant à une séquence de test doit impérativement comporter au moins un état suspect.

#### 5.3.2.2 Recherche d'un état suspect

Pour favoriser l'exécution des chemins qui comportent des états suspects, à chaque instant de la génération, l'idée est de rechercher les états suspects dans le voisinage de l'état courant. Pour cela, nous considérons l'ensemble des sous-chemins possibles d'une longueur prédéfinie et qui sont issus de l'état courant.

**Définition 5.3.3** *Sous-chemin faisable*
*Soit* $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ *une machine génératrice. Soit* $q_t \in Q$ *l'état courant dans cette machine. Un sous-chemin* $(q_t, q_{t+1}, ..., q_{t+k})$ *issu de* $q_t$ *et de longueur* $k > 0$ *est* faisable *(noté* $fais((q_t, q_{t+1}, ..., q_{t+k}), (b_i)_{i=1,k})$ *) dans la machine* $M_g$ *ssi*

$$\begin{gathered}\forall i = [1..k],\ \exists e_{t+i-1} \in E,\ \exists s_{t+i-1} \in S,\\ \exists b_i \in \{0,1\},\ b_i = prop(q_{t+i-1}, e_{t+i-1}, s_{t+i-1})\\ \wedge env(q_{t+i-1}, e_{t+i-1}) \wedge q_{t+i} = trans(q_{t+i-1}, e_{t+i-1}, s_{t+i-1})\end{gathered}$$

*et sa* signature *par rapport à prop est* $(b_i)_{i=1,k}$

**Remarque.** Intuitivement, un chemin est faisable si chacune de ses transitions est permise par les invariants de l'environnement. Autrement dit, un chemin est faisable si la première transition est possible et si la suite du chemin est aussi faisable :

$$fais((q_t, q_{t+1}, ..., q_{t+k}), (b_i)_{i=1,k}) = \begin{cases} true & k = 0 \\ env(q_t, e_t) & \\ \wedge b_1 = prop(q_t, e_t, s_t) & k > 0 \\ \wedge q_{t+1} = trans(q_t, e_t, s_t) & \\ \wedge fais((q_{t+1}, ..., q_{t+k}), (b_i)_{i=2,k}) & \end{cases}$$

La signature caractérise la valeur de vérité de la propriété de sûreté dans les états du sous-chemin considéré. Une valeur fausse dans cette signature correspond à une violation de la propriété de sûreté et, en conséquence, à un état suspect. Un vecteur

d'entrées est *pertinent* par rapport à la propriété de sûreté s'il est à l'origine d'un chemin comportant au moins un état suspect.

**Définition 5.3.4** *Vecteur d'entrées pertinent*
*Soit* $(q_t, q_{t+1}, ..., q_{t+k})$ *un* sous-chemin faisable *dans la machine* $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ *et* $(b_i)_{i=1,k}$ *sa signature par rapport à* prop *où* $q_t \in Q$ *est l'état courant. Un vecteur d'entrées* $e \in E$ *est dit* pertinent *d'ordre* $k$ *dans l'état* $q$ *(noté* $pert_k(q, e)$*) par rapport à la propriété de sûreté ssi*

$$\exists s \in S \text{ tel que } q_{t+1} = trans(q_t, e, s) \wedge (\neg b_1 \vee ... \vee \neg b_k)$$

Remarquons qu'un vecteur pertinent ne mène pas forcément à un état suspect, car les transitions possibles sont conditionnées par les sorties que le logiciel va fournir.

#### 5.3.2.3 Algorithme de génération

**Algorithm 5.2** Algorithme de génération d'une séquence de test guidé par propriété de sûreté

**paramètres**
$n \in \mathbb{N}$, $M_g = (Q, q_{init}, E, S, t, env, prop, hyp, prob)$, $k \in \mathbb{N}$ ; (1)
**variables**
$q : Q$ ; $e : E$ ; $s : S$ ; $V \subseteq E$ ;$V_P \subseteq E$ ;
**début**
$q \leftarrow q_{init}$ ;
**faire** $n$ **fois**
$V \leftarrow \{x \in E \mid env(q, x)\}$ ;
$V_P \leftarrow \{x \in V \mid pert_k(q, x)\}$ ; (2)
$e \leftarrow$`choose`$(V_P)$ ;
**si** $e = \emptyset$ **alors** (3)
$e \leftarrow$`choose`$(V)$ ; (4)
**finsi**
`write`$(e)$ ;
`read`$(s)$ ;
$q \leftarrow trans(q, e, s)$ ;
**finfaire**
**fin**.

Le but étant de générer un chemin comportant des états suspects, nous allons générer des vecteurs pertinents à chaque fois que c'est possible. L'algorithme de génération 5.2, considère d'abord les vecteurs pertinents par rapport à la propriété de sûreté (2). Si aucun vecteur pertinent n'est trouvé à l'état courant (3), l'algorithme continue la génération avec un vecteur satisfaisant seulement les invariants de l'environnement (4). Notons aussi l'introduction d'un nouveau paramètre $k$, représentant

la longueur des sous-chemins recherchés dans la définition des vecteurs pertinents (1).

#### 5.3.2.4 Détermination des vecteurs pertinents avec des contraintes

Pour réaliser l'algorithme précédent, il faut déterminer l'ensemble des vecteurs pertinents. Nous utilisons la définition récursive des chemins faisables pour réaliser la contrainte *feasible*. Cette contrainte définit les conditions reliant un sous-chemin faisable de longueur $k$ et issu de $q$ et sa signature $b = \{b_1, ..., b_k\}$. La variable $vars_k$ représente la liste des variables supplémentaires introduite par cette contrainte.

$$feasible(k, q_1, e_1, b, vars_k) \Leftrightarrow \begin{cases} b = \emptyset \wedge vars_k = \emptyset & k = 0 \\ b = \{b_1, .., b_k\} & \\ \wedge environment(q_1, e_1) & \\ \wedge safeprop(q_1, e_1, s_1) = b_1 & k > 0 \\ \wedge transition(q_1, e_1, s_1, q_2) & \\ \wedge vars_k = \{s_1, e_1, q_2\} \cup vars_{k-1} & \\ \wedge feasible(k-1, q_2, e_2, \{b_2, ..., b_k\}, vars_{k-1}) & \end{cases}$$

Ensuite, on définit l'ensemble des vecteurs d'entrées pertinents $V \leftarrow \{x \in E \mid pert_k(q, x)\}$, en postant la contrainte *pertinent*, définie comme suit :

$$pertinent(k, q, e, vars) \Leftrightarrow feasible(k, q, e, \{b_1, .., b_k\}, vars) \wedge (\neg b_1 \vee ... \vee \neg b_k)$$

#### 5.3.2.5 Stratégies de recherche d'états suspects

La définition d'un vecteur pertinent que nous avons donnée plus haut, très générale, considère l'ensemble des sous-chemins qui contiennent au moins un état suspect. Cependant, dans le cas où $k > 1$, la capacité de détection d'un défaut peut changer en fonction du nombre d'états suspects que contient un sous-chemin, ainsi que de leur proximité. Plusieurs stratégies de recherche d'états suspects ont été proposées dans [55] pour le cas booléen : l'*union*, l'*intersection* et la stratégie *paresseuse*. Nous pouvons réaliser l'ensemble de ces stratégies avec des contraintes, en posant des conditions sur la signature associée à un chemin suspect :

- La stratégie *d'union* correspond à l'ensemble des sous-chemins de longueur $k$ comportant au moins un état suspect. Elle correspond à la condition sur la signature $\neg b_1 \vee ... \vee \neg b_k$, que nous avons posé plus haut.
- La stratégie *d'intersection* ne considère que les sous-chemins de longueur $k$ dont tous les états sont suspects. Elle est obtenue en posant la condition $\neg b_1 \wedge ... \wedge \neg b_k$.
- La stratégie *paresseuse* introduit un ordre des sous-chemins considérant d'abord ceux qui ont l'état suspect le plus proche. Elle peut être réalisée en cherchant les sous-chemins qui vérifient, dans l'ordre, les conditions : $\neg b_1$, $b_1 \wedge \neg b_2$, ..., $b_1 \wedge ... \wedge b_{k-1} \wedge \neg b_k$.

#### 5.3.2.6 Longueur minimale des sous-chemins

Le guidage par propriétés de sûreté utilise un paramètre $k$ représentant la longueur du sous-chemin à considérer pendant la recherche de vecteurs pertinents. Considérons une propriété :

$$true \rightarrow pre(i) \Rightarrow o$$

ou $i$ est une entrée et $o$ une sortie du logiciel sous test. Évidemment, à l'instant courant $t$ nous ne pouvons violer cette propriété si la valeur de $pre(i)$ n'était pas vraie à l'instant $t-1$. Pour arriver à un état suspect à l'instant $t+1$, il faut générer $i = vrai$ à l'instant $t$. Il faut donc considérer pour cette propriété au moins deux instants successifs. La recherche de vecteur pertinent doit se faire sur des chemins de longueur $k \geq 2$.

S'il est facile de voir dans cette propriété le nombre minimum d'instant à considérer, dans le cas général il peut être très grand ou alors impossible à déterminer car il peut être lié à un calcul. Considérons par exemple un compteur $x$ dont la valeur est remise à zéro quand un signal *reset* et reçu :

$$x = \text{if } reset \text{ then } 0 \text{ else } 1 \rightarrow pre(x) + 1$$

Pour une propriété de sûreté $x > 1000 \Rightarrow o$, ce paramètre est $k \geq 1001$, alors que pour une propriété $(pre(reset) \wedge x > 1000) \Rightarrow o$, aucun des états suspects n'est accessible.

L'utilisateur doit donc choisir judicieusement ce paramètre en fonction de la propriété considérée.

### 5.3.3 Prise en compte d'hypothèses sur le programme

Les hypothèses représentent des relations entre les entrées et le sorties du logiciel. Cette information n'est utilisée que pendant le guidage par propriétés de sûreté où l'anticipation des sorties du logiciel peut réduire considérablement l'espace de recherche, améliorant ainsi les résultats du guidage par les propriétés de sûreté. Une hypothèse réduit le nombre d'états suspects en éliminant ceux pour lesquels il n'existe pas de sortie du logiciel pouvant violer la propriété.

Nous introduisons une nouvelle définition de l'état suspect :

**Définition 5.3.5** *État suspect sous hypothèse*
*Un état* $q \in Q$ *de la machine génératrice* $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ *est dit* suspect sous hypothèse *si*

$$\exists e \in E,\ \exists s \in S,\ env(q, e) \wedge hyp(q, e, s) \wedge \neg prop(q, e, s)$$

Dans ce cas, un chemin est faisable si l'hypothèse est vérifiée tout au long du sous-chemin recherchée.

**Définition 5.3.6** *Sous-chemin faisable sous hypothèse*
*Soit* $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ *une machine génératrice. Soit* $q_t \in Q$ *l'état courant dans cette machine. Un sous-chemin* $(q_t, q_{t+1}, ..., q_{t+k})$ *issu de* $q_t$ *et de longueur* $k$ *est* faisable sous hypothèse *(noté* $fais_H((q_t, q_{t+1}, ..., q_{t+k}), (b_i)_{i=1,k})$ *) dans la machine* $M_g$ *ssi*

$$fais_H((q_t, q_{t+1}, ..., q_{t+k}), (b_i)_{i=1,k}) = \begin{cases} true & k = 0 \\ env(q_t, e_t) & \\ \wedge b_1 = prop(q_t, e_t, s_t) & k > 0 \\ \wedge hyp(q_t, e_t, s_t) & \\ \wedge q_{t+1} = trans(q_t, e_t, s_t) & \\ \wedge fais_H((q_{t+1}, ..., q_{t+k}), (b_i)_{i=2,k}) & \end{cases}$$

*et sa* signature *par rapport à* prop *est* $(b_i)_{i=1,k}$

Un vecteur pertinent étant défini par rapport à cette signature, nous gardons la

même définition et la génération de données de test se fait en utilisant l'algorithme 5.2.

#### 5.3.3.1 Détermination des chemin faisables sous hypothèse par des contraintes

A la différence de la version précédente de la contrainte *feasible*, pour prendre en compte les hypothèses sur le programme, nous introduisons la définition de la contrainte *hfeasible* comprenant aussi les contraintes sur l'hypothèse :

$$hfeasible(k, q_1, e_1, b, vars_k) \Leftrightarrow \begin{cases} b = \emptyset \wedge vars_k = \emptyset & k = 0 \\ \\ b = \{b_1, .., b_k\} & \\ \wedge environment(q_1, e_1) & \\ \wedge safeprop(q_1, e_1, s_1) = b_1 & \\ \wedge hypothesis(q_1, e_1, s_1) & k > 0 \\ \wedge transition(q_1, e_1, s_1, q_2) & \\ \wedge vars_k = \{s_1, e_1, q_2\} \cup vars_{k-1} & \\ \wedge hfeasible(k-1, q_2, e_2, \{b_2, ..., b_k\}, vars_{k-1}) & \end{cases}$$

En utilisant cette contrainte, on obtient une nouvelle définition pour la contrainte *pertinent*, prenant en compte les hypothèses :

$$pertinent(k, q, e, vars) \Leftrightarrow hfeasible(k, q, e, \{b_1, .., b_k\}, vars) \wedge (\neg b_1 \vee ... \vee \neg b_k)$$

#### 5.3.3.2 Discussion

Si le logiciel sous test entier est introduit comme hypothèse de test, l'existence d'un état suspect suppose l'existence d'une transition qui nécessairement viole la propriété de sûreté. La recherche explore l'ensemble des sous-chemins de longueur $k$ et peut être vue comme une preuve locale. Dans chacun des états sur le chemin d'exécution du logiciel, on cherche dans tous les sous-chemins de longueur $k$ qui partent de l'état courant si la propriété de sûreté est violée. Si un état suspect est trouvé dans le voisinage de l'état courant, cet état est nécessairement accessible et le guidage mène à une violation de la propriété de sûreté.

### 5.3.4 Génération guidée par les probabilités conditionnelles

Les probabilités conditionnelles permettent d'exprimer la probabilité qu'une expression soit vraie si une condition est vérifiée. L'ensemble des probabilités d'une spécification est représenté dans la machine génératrice par le vecteur $((c_i, e_i, p_i))_{i=1,v}$. Chacune des conditions $c_i : Q \rightarrow \{0, 1\}$, qui ne dépend que des variables d'état, détermine un sous-ensemble d'états de la machine dans lesquels cette condition est vérifiée. Dans les états où cette condition n'est pas vérifiée, la probabilité ne doit avoir aucun effet.

Dans les états qui vérifient la condition, la probabilité que $e_i(q, e) = 1$ est égale à $p_i$. Autrement dit, si $r_i$ est une variable aléatoire, la sémantique d'une expression de probabilité est $c_i \Rightarrow (r_i \leq p_i \Leftrightarrow e_i)$.

**Définition 5.3.7** *Vecteur d'entrée respectant les probabilités*
*Soit la machine génératrice* $M_g = (Q, q_{init}, E, S, trans, env, prop, hyp, prob)$ *avec* $prob = ((c_i, e_i, p_i))_{i=1,v}$, *des probabilités conditionnelles. Soit* $(r_i)_{i=1,v}$ *un vecteur de variables aléatoires dans* (0..1). *Un vecteur d'entrées* $x \in E$ *respecte les probabilités dans l'état* $q \in Q$ *ssi :*

$$\forall i \in (1..v),\ c_i(q) \Rightarrow (r_i \leq p_i \Leftrightarrow e_i(q, x))$$

#### 5.3.4.1 Algorithme de génération

Pour prendre en compte les probabilités, dans l'algorithme de génération 5.3, nous introduisons une variable aléatoire (1). Pour chaque probabilité spécifiée (2), nous ajoutons des contraintes supplémentaires sur les vecteurs d'entrée valides (3-5) :

- $c_i$ est la condition pour que la probabilité soit effective, $e_i$ est l'expression et sa probabilité est $p_i$ (3),
- $r$ prend une valeur aléatoire (4),
- Nous contraignons les vecteurs valides à satisfaire la condition $c_i(q) \Rightarrow (r \leq p_i \Leftrightarrow e_i(q, x))$ (5).

#### 5.3.4.2 Les contraintes correspondant aux probabilités

La contrainte *probabilities* générée définit les relations entre l'état $q$, les entrées courantes $e$ et la liste de probabilités $l_1, ..., l_v$ spécifiées. En supposant que $r_1, ..., r_v$ sont $v$ valeurs aléatoires entre 0 et 1, on peut imposer le respect de chacune des probabilités en posant les contraintes suivantes :

**Algorithm 5.3** Algorithme de génération d'une séquence de test avec des probabilités

**paramètres**
$n \in \mathbb{N}$, $M_g = (Q, q_{init}, E, S, t, env, prop, hyp, prob)$, $k \in \mathbb{N}$ ;
**variables**
$q : Q$ ; $e : E$ ; $s : S$ ; $V \subseteq E$ ; $V_P \subseteq E$ ; $r \in \mathbb{R}$ ; (1)
**début**
$q \leftarrow q_{init}$ ;
**faire** $n$ **fois**
$V \leftarrow \{x \in E \mid env(q, x)\}$ ;
**pour** $i = 1..|prob|$ **faire** (2)
$(c_i, e_i, p_i) \leftarrow prob_i$ ; (3)
$r \leftarrow$`random()` ; (4)
$V \leftarrow \{x \in V \mid c_i(q) \Rightarrow (r \leq p_i \Leftrightarrow e_i(q, x))\}$ ; (5)
**finpour**
$V_P \leftarrow \{x \in V \mid pert_k(q, x)\}$
$e \leftarrow$`choose`$(V_P)$ ;
**si** $e = \emptyset$ **alors**
$e \leftarrow$`choose`$(V)$ ;
**finsi**
`write`$(e)$ ;
`read`$(s)$ ;
$q \leftarrow trans(q, e, s)$ ;
**finfaire**
**fin**.

$$probabilities(q, e, [(c_1, e_1, p_1), ..., (c_v, e_v, p_v)]) \wedge \bigwedge_{i=1}^{v} c_i \Rightarrow (r_i \leq p_i \Leftrightarrow e_i)$$

#### 5.3.4.3 Discussion sur la cohérence des probabilités

Il est possible que l'ensemble des contraintes ainsi exprimées, comprenant aussi les contraintes d'environnement, ne puissent pas être satisfaites. Cette incohérence peut apparaître dans une seule probabilité conditionnelle, entre une probabilité conditionnelle et les invariants de l'environnement ou alors entre deux ou plusieurs probabilités conditionnelles.

Une incohérence peut apparaître dans une probabilité conditionnelle si l'expression, à chaque fois que la condition est vérifiée, ne peut pas prendre à la fois la valeur *faux* et *vrai*. Par exemple, considérons une entrée $x$ et une probabilité conditionnelle simple :

```
prob(true, pre(x), 0.8) ;
```

où la valeur de *pre(x)* est déjà connue et ne peut pas donc être affectée à une autre valeur, quelque soit la valeur des entrées choisies. Une condition nécessaire pour éviter cette incohérence est que l'expression comporte au moins une variable d'entrée. Mais, cette condition n'est pas suffisante, comme on peut le voir dans l'exemple suivant :

```
prob(true, x and pre(x), 0.8) ;
```

car si *pre(x)=faux*, l'expression sera fausse quelle que soit la valeur de x.

Une incohérence peut aussi apparaître en conséquence d'une contradiction entre une probabilité conditionnelle et les invariants de l'environnement. Considérons l'exemple suivant :

```
environment(x) ;
prob(true, x, 0.8) ;
```

où l'invariant de l'environnement empêche $x$ d'avoir la valeur *faux*, alors que la probabilité stipule que cela doit arriver dans 20% des cas.

Enfin, deux ou plusieurs probabilités conditionnelles peuvent être contradictoires entre elles. Considérons la spécification suivante :

```
prob(true, x, 0.8) ;
prob(true, not(x), 0.2) ;
```

qui est incohérente, car les deux probabilités sont indépendantes ainsi que les valeurs aléatoires correspondantes. Ainsi, il peut arriver que $r_1 \leq 0.8$ et $r_2 \leq 0.2$ (16%), ou alors que $r_1 > 0.8$ et $r_2 > 0.2$ (16%), et dans ces cas il faut satisfaire à la fois $x$ et $not(x)$. Cette incohérence peut apparaître seulement dans le cas où les conditions des deux probabilités conditionnelles peuvent être vérifiées simultanément dans un même état. Si l'ensemble des probabilités conditionnelles ont des conditions deux à deux disjointes, on est assuré que ces probabilités ne seront jamais considérés dans le même état évitant ainsi une contradiction éventuelle.

Généralement, le probabilités $prob = ((c_i, e_i, p_i))_{i=1,v}$ d'une machine génératrice $M_g = (Q, q_{init}, E, S, env, prop, hyp, prob)$ sont cohérentes dans un état $q \in Q$ ssi :

$$\forall (l_1, ..., l_v) \in \{0,1\}^v,\ \exists x \in E,\ env(q,x) \wedge \bigwedge_{i=1}^{v} c_i(q) \Rightarrow (l_i \Leftrightarrow e_i(q,x))$$

Notons qu'il n'est pas nécessaire que les probabilités soit cohérentes dans tous les états de la machine mais seulement dans ceux qui sont accessibles. Or, outre la

complexité que ça représente le problème d'accessibilité dans une machine à états finis, les transitions de cette machine ne sont pas connues car dépendant de la valeur des sorties du logiciel sous test. En conséquence, on ne peut pas vérifier statiquement si les probabilités exprimés sont cohérentes.

A défaut de pouvoir vérifier à priori, nous pouvons générer des données de test et détecter dynamiquement le situations ou une incohérence se présente, dans quel cas, nous avons deux choix :

- Considérer que la génération doit s'arrêter en signalant à l'utilisateur qu'il doit rectifier sa spécification de probabilités.
- Considérer, au contraire, que l'utilisateur exprime ces probabilités comme un moyen simple de guider la génération, sans se soucier de leur distribution effective et de leur cohérence.

Dans le deuxième cas on envisage la possibilité de continuer la génération en rejetant certaines probabilités. Une méthode simple que nous avons réalisé pour faire ceci, consiste à supposer que les probabilités spécifiées en premier sont plus importantes que les autres, et rejete ainsi les probabilités en commençant par la dernière et jusqu'à obtenir un système de contraintes comportant une solution. Nous laissons le choix à l'utilisateur, par le moyen de configuration, d'utiliser ou non cette possibilité.

## 5.4 Choix des données concrètes de test

Les contraintes invariantes et les éventuelles directives de guidage réduisent les séquences de test possibles en un sous-ensemble qui respecte la spécification, définissant ainsi ce qu'on peut appeler une *donnée de test abstraite*. Afin de générer des données de test effectives, il faut choisir dans cet sous-ensemble des valeurs concrètes pour les variables d'entrée du logiciel (méthode `choose`). Autrement dit, il faut trouver une solution particulière du système à contraintes associé. La propagation des contraintes réduit le domaine des variables d'entrée et la recherche de cette solution se fait par énumération de ce domaine réduit. Il existe plusieurs types d'énumération, dont les plus connus ont été présentés dans 2.3.6.

Le choix de la méthode d'énumération peut déterminer le type d'erreur qu'une séquence de test est susceptible de détecter. Nous présentons par la suite la généra-

tion par énumération aléatoire, qui a le mérite de considérer l'ensemble des vecteurs d'entrées valides.

### 5.4.1 Génération aléatoire

La génération aléatoire consiste à rechercher une solution du système à contraintes par une énumération aléatoire. Ce type d'énumération, choisit pour chaque variable une valeur aléatoire dans son domaine réduit, et ceci jusqu'à obtenir une affectation complète qui satisfait l'ensemble des contraintes.

Notons que, contrairement à la version booléenne de LUTESS, il n'est pas évident de garantir l'équiprobabilité entre les différentes solutions possibles (l'ensemble des vecteurs d'entrée conformes à l'environnement), même si aucune des solutions n'est totalement exclue. De plus, le tirage équiprobable entre les vecteurs d'entrée valides n'a pas forcement la même pertinence quand les domaines des variables sont de tailles très différentes. Considérons, par exemple, une condition simple $b \Rightarrow x = 0$. Dans une sélection équiprobable entre toutes les solutions, la probabilité de tirer une solution telle que $\{b \leftarrow 1, x \leftarrow 0\}$ est quasi nulle ($\frac{1}{2^{32}+1}$), alors que cette solution a manifestement un intérêt particulier.

Par ailleurs, choisir aléatoirement les valeurs pour les variables dans leur domaine de solutions peut donner une distribution différente selon l'ordre des variables selon lequel cette sélection est effectuée. Ainsi, si on choisit d'abord la variable $x$, on obtient une probabilité de $\frac{1}{2^{33}}$ d'avoir la solution $\{b \leftarrow 1, x \leftarrow 0\}$ alors que si on choisit d'abord la variable $b$, la probabilité d'avoir cette même solution est de $\frac{1}{2}$.

Plusieurs ordres pour l'énumération des entrées sont envisageables dans ce cas. Parmi les méthodes classiques, que nous avons présentées dans 2.3.6, nous considérons l'ordre consistant à choisir d'abord la variable ayant le plus petit domaine réduit. Cette méthode, communément connue sous le nom de "first-fail", en plus d'être plus efficace, généralement résulte en une meilleure distribution du point de vue du test, car elle considère d'abord les variables booléennes. Ceci est particulièrement vrai dans les logiciels de contrôle-commande dont la partie numérique est d'une nature secondaire par rapport à la partie contrôle, généralement implémentée avec des variables booléennes.

### 5.4.2 Autres aspects

A part l'énumération aléatoire, on peut envisager la génération aux bornes qui consiste à énumérer aléatoirement une des bornes (le maximum ou le minimum) de chaque variable d'entrée. Cette génération vise à découvrir des défauts particuliers, généralement liées à l'utilisation imprudente des opérateurs de comparaison.

En plus des variables de l'instant courant, d'autres variables sont introduites pendant la recherche d'un chemin suspect. Afin de s'assurer que l'ensemble des contraintes introduites avec ce guidage sont satisfiables, il faut trouver au moins une affectation de ces variables satisfaisant les contraintes associées. Le domaine de ces variables est énuméré dans un ordre croissant, en commençant par la valeur minimale.

## 5.5 Conclusion

Nous avons défini la représentation d'une spécification de test par une machine à états finis, appelée *machine génératrice*. Les expressions contenues dans les opérateurs de test de la spécification correspondent à des fonctions dans cette machine et une séquence de test correspond à un parcours de celle-ci. Pour réaliser ce parcours, nous avons besoin de déterminer, dans chaque état, l'ensemble des transitions possibles. A cette fin, nous utilisons la programmation par contraintes et nous avons donné les principes de la représentation de cette machine par un ensemble de contraintes.

Nous avons ensuite défini formellement la signification de chaque opérateur de test, informellement introduit dans le chapitre précédent. Cette définition est faite en utilisant les fonctions de la machine génératrice. Nous précisons l'algorithme de génération utilisé, en introduisant les différents opérateurs de test d'une manière incrémentale. Les différents éléments de cet algorithme sont réalisés à l'aide des contraintes représentant la machine génératrice.

A chaque instant, l'ensemble des vecteurs d'entrées valides par rapport à la spécification est déterminé par un ensemble de contraintes. Afin de choisir une valeur concrète à envoyer au logiciel sous test, nous utilisons des énumérations classiques de la programmation par contraintes.

Suite à la définition théorique de l'ensemble des algorithmes de génération ainsi que des contraintes qu'il faut satisfaire, nous avons réalisé une implémentation de celles-ci dans la nouvelle version de l'outil, que nous appelons LUTESS V2. Les aspects pratiques et les choix que nous avons fait sont développés dans le chapitre suivant.

# Troisième partie

# Mise en oeuvre (Lutess V2)

# Chapitre 6

# Réalisations

A la suite de la définition théorique des spécifications et des moyens de génération de données à l'aide des contraintes, présentées dans le chapitre précédent, nous avons réalisé une implémentation des méthodes de génération dans la nouvelle version de l'outil, LUTESS V2.

Nous présentons dans ce chapitre certains des aspects techniques et des choix que nous avons fait pour cette implémentation. Parmi ces aspects, l'architecture et le fonctionnement de l'outil font l'objet de la section 6.1, les caractéristiques des contraintes générées pour représenter la machine à états finis correspondant à une spécification sont présentées dans la section 6.2 et la section 6.3 présente les algorithmes de génération et les prédicats utiles que nous utilisons pour implémenter ces algorithmes.

## 6.1 Architecture de LUTESS V2

LUTESS est un outil de génération dynamique et automatisée de données de test qui permet la définition de plusieurs spécifications de test pour un même logiciel. Pour chacune de ces spécifications, un grand nombre de séquences aléatoires peuvent être produites, en utilisant un germe différent pour l'initialisation du générateur de nombre aléatoires.

### 6.1.1 Composants de l'outil

La figure 6.1 montre la composition fonctionnelle de l'outil LUTESS. Le processus de test implique 3 composants interconnectés :

– le logiciel sous test dans une forme exécutable,

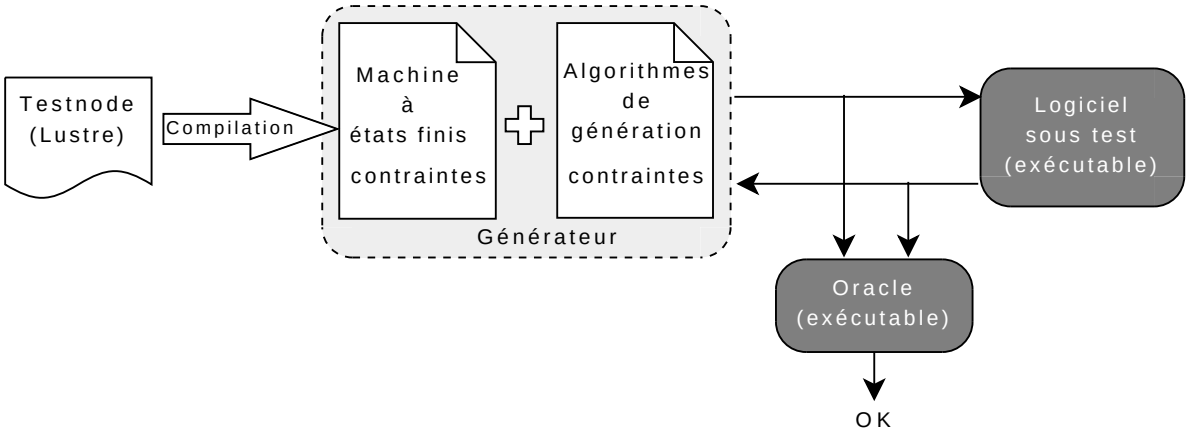


FIGURE 6.1 – Architecture de l'outil LUTESS

- un oracle aussi exécutable et
- un générateur de données de test.

Le processus de test consiste en une séquence d'échanges entre le générateur et le logiciel : les entrées qui sont produites par le générateur sont envoyées au logiciel sous test et les sorties calculées par ce dernier sont récupérées pour le calcul des entrées suivantes. Pendant chacun de ces échanges, l'ensemble des entrées et des sorties est utilisé par l'oracle pour décider si le logiciel se comporte correctement.

**Le logiciel sous test** est un programme exécutable ayant un comportement synchrone.

**L'oracle** est aussi un logiciel exécutable qui fonctionne sous la forme d'un observateur synchrone. Il prend en entrée l'ensemble des entrées et des sorties du logiciel sous test et calcule une sortie qui représente la valeur de vérité des propriétés qui sont observées.

**Le générateur de tests** est créé automatiquement à partir de la spécification de l'environnement, contenue dans un *testnode*.

### 6.1.2 Compilation de la spécification

Une spécification est d'abord compilée afin d'obtenir une représentation sous la forme d'une machine à états finis. Cette machine est représentée dans un fichier par un ensemble de prédicats Prolog, qui spécifient les contraintes associées. La section

6.2 est consacrée aux aspects pratiques de cette réprésentation. En plus de cette machine, un autre fichier contient l'algorithme de génération qui est aussi réalisé entièrement en Prolog. Cet algorithme est présenté dans la section 6.3. Le générateur résulte de la combinaison de ces deux fichiers.

### 6.1.3 Environnement de programmation logique par contraintes

Les contraintes présentées dans le chapitre 5 sont indépendantes, du point de vue théorique, d'un solveur particulier. Nous avons réalisé ce générateur en utilisant l'environnent ECLiPSe [1] et sa librairie hybride de contraintes sur les intervalles d'entiers/réels *ic* [2].

La libraire *ic* implémente les contraintes et les algorithmes standard d'une librairie de résolution de contraintes sur les domaines finis. La raison principale de ce choix se situe dans l'aptitude de cette librairie à gérer tous les entiers de taille 32 bits. Une autre raison majeure de ce choix est le fait que l'environnement ECLiPSe est librement disponible, un critère qui permet une distribution plus facile de l'outil.

## 6.2 Représentation de la machine en contraintes

La spécification contenue dans un *testnode* est compilée en un fichier de prédicats ECLiPse, permettant de poster les contraintes décrivant la machine à états finis associée. Le lecteur pourra trouver en annexe B l'ensemble des contraintes générées pour l'exemple du climatiseur que nous avons traité dans le chapitre 5. Nous donnons ici quelques aspects de cette génération, tels qu'ils sont implémentés dans la nouvelle version de Lutess.

Nous utilisons des prédicats pour poster les différentes contraintes. Un prédicat est généré pour chacune des contraintes que nous avons spécifiées dans le chapitre précédent (*inputs*, *outputs*, *state*, *initial*, *transition*, *environment*, *safeprop*, *hypothesis* et *probabilities*). Par exemple, l'état de la machine est spécifié à l'aide du prédicat suivant :

1. http ://www.eclipse-clp.org
2. Interval Constraint

```
state(S) :- S = [_Q0,_Q1,_Q2,_Q3,_Q4], type_bool(_Q0),
type_int(_Q1), type_bool(_Q2), type_bool(_Q3), type_bool(_Q4).
```

où `type_bool/1` et `type_int/1` spécifient respectivement les contraintes pour les types booléen et entier signé pour les variables.

Donnons en exemple le prédicat `environment` pour le climatiseur, utilisé pour définir les contraintes correspondant aux invariants de l'environnement :

```
environment(S,I) :-
     state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4],
     inputs(I), I = [_Bouton,_Tamb,_Tutil],
     _dTamb #= if_then_else(_Q0,0,_Tamb - _Q1),
     and(_Tamb #>= -20, _Tamb #=< 60),
     and(_Tutil #>= 10, _Tutil #=< 40),
     and(_dTamb #>= -1, _dTamb #=< 1).
```

Premièrement, nous définissons `S` comme un état et `I` comme un vecteur d'entrées. Ensuite, la variable locale `dTamb` est définie en utilisant la contrainte `if_then_else`, implémentée selon la sémantique de la construction *if-then-else* fonctionnelle du langage LUSTRE. Enfin, les trois contraintes suivantes correspondent aux invariants de l'environnement.

D'une façon similaire aux invariants de l'environnement, la propriété de sûreté est spécifiée par le prédicat `safeprop(S,I,O,B)` reliant l'état `S`, les entrées `I` et les sorties `O` avec la valeur de vérité de la propriété `B`. Le prédicat suivant est généré pour la propriété de sûreté $En\_marche \wedge Tamb < Tutil \Rightarrow Tsort > Tutil$ de l'exemple du climatiseur :

```
safeprop(S,I,O,B) :-
     state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4],
     inputs(I), I = [_Bouton,_Tamb,_Tutil],
     outputs(O), O = [_En_marche,_Tsort],
     type_bool(B),
     B #= or( neg( and( _En_marche, _Tamb #< _Tutil ) ),
          _Tsort #> _Tutil ).
```

Remarquons la spécification comme dernier argument de la variable `B` pour représenter la valeur de vérité de la propriété. L'environnement ECLiPSe permet l'utilisation d'un prédicat comme toute autre contrainte, la dernière valeur représentant le contexte d'utilisation. Ainsi, nous pouvons spécifier simplement `neg(safeprop(S,I,O))` pour contraindre la propriété à être fausse.

#### Utilisation des opérateurs optionnels

Notons que l'utilisation des opérateurs *safeprop*, *hypothesis* et *prob* est optionnelle. Pour avoir un algorithme de génération uniforme, même si un opérateur n'est pas utilisé, nous générons un prédicat.

Dans le cas où aucune propriété n'est spécifiée, il suffit de contraindre `B` à être de type booléen de sorte que la propriété de sûreté puisse être soit vraie, soit fausse :

```
safeprop(S,I,O,B) :- type_bool(B).
```

Nous assimilons aussi le cas où aucune hypothèse n'est spécifiée à l'ajout de l'hypothèse *vrai*. De même, le prédicat `probabilities(S,I,L)` définit une liste `L` de probabilités qui sera simplement vide si aucune probabilité n'est spécifiée.

## 6.3 Algorithmes de génération

Nous avons aussi réalisé l'algorithme de génération de données de test par un ensemble de prédicats ("framework"). Le code complet de ce framework est donné en annexe C. Ils contient, outre les algorithmes de génération de données de test et les prédicats utilisés par ces derniers, les librairies utilisées, les prédicats de configuration ainsi que des contraintes pour les opérateurs spécifiques du langage Lustre. Nous expliquons dans cette section quelques particularités de cette implémentation.

### 6.3.1 Prédicats de configuration

Nous avons discuté, dans le chapitre précédent, certaines alternatives pour la génération. Nous avons laissé une possibilité pour que l'utilisateur puisse configurer ces alternatives. La version actuelle de Lutess, prend en compte les configurations suivantes :

- `safeprop_length` permet de spécifier la longueur du sous-chemin pour la recherche des états suspects. La valeur doit être un entier strictement positif.
- `safeprop_strategy` permet de spécifier la stratégie de recherche des états suspects. Nous avons défini les stratégies suivantes : `intersection, union` et `lazy.`
- `probability_reject` permet de spécifier si dans le cas de conflit des probabilités, il faut continuer la génération en ignorant certaines probabilités. Les valeurs possibles sont :
  - `none` : signifie que la génération doit s'arrêter et
  - `backtrack` : signifie que la génération continue en rejetant les probabilités une à une, en commençant par la dernière et jusqu'à ce qu'une solution soit trouvée,
- `choose_method` : permet de spécifier la méthode de choix des entrées. Nous avons défini 3 méthodes pour ce choix :
  - `random` : choisit une valeur aléatoire dans le domaine réduit de la variable, dans l'ordre d'apparence des variables,
  - `random_smallest_domain` : choisit une valeur aléatoire dans le domaine réduit de la variable, en choisissant celle ayant le plus petit domaine et
  - `random_bound` : choisist aléatoirement une des bornes inférieure ou supérieure du domaine réduit de la variable, dans l'ordre d'apparence des variables

L'exemple suivant configure la génération pour rechercher une violation de la propriété de sûreté sur les sous-chemins de longueur 3 par la stratégie d'intersection, arrêtant la génération quand il y a un conflit des probabilités et générant les entrées aléatoires en commençant par les variables ayant le plus petit domaine réduit :

```
config(safeprop_length, 3).
config(safeprop_strategy, intersection).
config(probability_reject, none).
config(choose_method, random_smallest_domain).
```

### 6.3.2 Algorithme de génération d'une séquence

La génération d'une séquence de longueur `N` est définie par le prédicat `sequence(N, Seed)` donné dans l'algorithme 6.1, comme un chemin de même longueur partant de l'état initial. La valeur `Seed` correspond au germe initialisant le générateur aléatoire.

**Algorithm 6.1** Algorithme de génération d'une séquence de test en ECLiPSe Prolog

```
sequence(N, Seed) :- seed(Seed), initial(S), path(S,N).
path(_S,0).
path(S,N) :- N>0,
    environment(S,I),
    post_probabilities(S,I),
    ( pertinent(S,I,Vars) ; Vars = [] ),                    (1)
    config(choose_method, Method),
    choose(I, Method),
    enumlist(Vars),!,
    write_inputs(I),
    read_outputs(O),
    transition(S,I,O,Sp), enum(Sp),
    N1 is N-1, path(Sp,N1).
```

Notons la création d'un point de choix lors de la recherche des vecteurs pertinents (1) et l'utilisation de la recherche en arrière en cas d'échec comme une alternative si aucun tel vecteur n'est trouvé. Le prédicat `enumlist` est utilisé pour énumérer les variables supplémentaires, introduites pour la recherche d'un vecteur pertinent, afin de s'assurer qu'il y a au moins une solution.

### 6.3.3 Poster les contraintes pour les probabilités

L'implémentation du prédicat pour poster les contraintes sur les probabilités est donné par l'algorithme 6.2. Nous définissons d'abord la liste des probabilités en utilisant le prédicat `probabilities/3` et ensuite, pour chacune de ces probabilités, on utilise le prédicat `proba(C,E,P,Option)` pour poster la contrainte $C \Rightarrow (R \leq P \Leftrightarrow E)$ où $R$ est une valeur aléatoire.

**Algorithm 6.2** Prédicat pour poster les contraintes sur les probabilités

```
post_probabilities(S,I) :-
    probabilities(S,I,L),
    (foreach(X,L) do
        (
            config(probability_reject, Option),
            X = [C,E,P],
            prob(C,E,P,Option)
        )
    ).
```

Le dernier argument du prédicat `prob/4` sert à spécifier si la génération doit continuer en rejetant des probabilités dans le cas ou une incohérence empeche la satisfaction du système de contraintes. Nous utilisons les mécanismes de retour en arrière en cas d'échec et créons un point de choix alternatif pour chaque probabilité.

### 6.3.4 Poster les contraintes pour la propriété de sûreté et les hypothèses

Nous utilisons la définition récursive d'un sous-chemin faisable sous hypothèse pour réaliser la contrainte `hfeasible/5` de l'algorithme 6.3. Un sous-chemin faisable de longueur $k > 1$ est constitué d'une première transition $(q, e, s, q')$ suivie d'un sous-chemin faisable de longueur $k - 1$. Les entrées/sorties de cette transition doivent satisfaire les invariants de l'environnement et l'hypothèse, la valeur de vérité de la propriété étant associée au premier élément de la signature. Le dernier élément de cette contrainte récupère la liste des variables supplémentaires qui sont introduites. Une énumération de ces variables est nécessaire pour s'assurer qu'il existe une solution pour ces contraintes, une fois que les données d'entrée sont choisies aléatoirement.

**Algorithm 6.3** Définition récursive d'un sous-chemin faisable sous hypothèse.

```
hfeasible(0,_S,_I,[],[]).
hfeasible(K,S,I,[B|Signature],[I,O,Sp|Vars]) :- K > 1,
    environment(S,I),
    safeprop(S,I,O) #= B,
    hypothesis(S,I,O),
    transition(S,I,O,Sp),
    K1 is K - 1, hfeasible(K1,Sp,_Ip,Signature,Vars).
```

Un vecteur pertinent d'entrées fait partie d'un sous-chemin faisable et il est

recherché par la méthode choisie dans la configuration (alg. 6.4).

**Algorithm 6.4** Définition d'un vecteur pertinent

```
pertinent(S,I,Vars) :-
    config(safeprop_length, K),
    config(safeprop_strategy, Strategy),
    hfeasible(K,S,I,Signature,Vars),
    search_strategy(Signature, Length, Strategy).
```

Les 3 méthodes de recherche d'un vecteur pertinent sont définies à l'aide de la signature du sous-chemin (alg. 6.5). Les deux premières sont définies à l'aide de contraintes supplémentaires alors que la dernière se fait par une énumération de cette signature.

**Algorithm 6.5** Définition des stratégies de recherche d'un vecteur pertinent.

```
search_strategy(Signature, Length, union) :-
    M is Length - 1, atmost(M,Signature,1).
search_strategy(Signature, Length, intersection) :-
    ( foreach(B,Signature) do B#=0 ).
search_strategy(Signature, Length, lazy) :-
    search(Signature, 0, input_order, indomain_min, complete, []).
```

### 6.3.5 La méthode de sélection de données concrètes de test

Le prédicat `choose/2` implémente les différentes méthodes pour la sélection de données de test concrètes que nous avons discutées dans le chapitre précédent. Nous avons fait remarquer que, pendant l'énumeration, l'ordre de choix des variables ainsi que la manière de parcourir leur domaine affecte la distribution des données de test qui sont générées. Les trois alternatives du prédicat `choose/2` de l'algorithme 6.6 implémentent les différentes méthodes discutées :

- La méthode `random` énumère les variables dans l'ordre selon lequel elles apparaissent dans la liste, qui est celui des variables d'entrée dans la spécification. Les variables sont instanciées avec une valeur aléatoire, prise dans leur domaine réduit,
- La méthode `random_smallest_domain` énumère les variables en commençant par celle qui à le plus petit domaine réduit. De même, les variables sont instanciées avec une valeur aléatoire, prise dans leur domaine réduit.

– Enfin, la méthode `random_bound` énumère les variables dans l'ordre d'apparition, mais utilise un prédicat pour choisir aléatoirement d'instancier avec une des bornes inférieure ou supérieure du domaine réduit de la variable.

**Algorithm 6.6** Définition des 3 methodes alternatives de sélection de données concrètes.

```
choose(L, random) :-
    search(L, 0, input_order, indomain_random, complete, []).
choose(L, random_smallest_domain) :-
    search(L, 0, most_constrained, indomain_random, complete, []).
choose(L, random_bound) :-
    search(L, 0, first_fail, min_max_random, complete, []).
min_max_random(X) :- (frandom(R), (R =< 0.5) -> get_min(X,X) ;
get_max(X,X) ).
```

### 6.3.6 Contraintes pour les opérateurs LUSTRE

Nous prenons en compte les différents opérateurs de LUSTRE en associant une contrainte à chacun d'entre eux. Parmi eux, on trouve les opérateurs usuels binaires : logiques, arithmétiques et de comparaison. Pour ces opérateurs, il existe des contraintes spécifiques dans la libraire *ic* que nous faisons correspondre. Pour les autres opérateurs du langage, nous réalisons des contraintes spécifiques implémentant la sémantique de l'opérateur.

Nous avons implémenté une contrainte pour l'opérateur booléen n-aire "#", spécifique au langage LUSTRE, ayant la signification qu'au plus une seule de ces opérandes est vraie. Une implémentation efficace de cet opérateur est réalisé à l'aide de la contrainte globale `atmost`.

```
sharp(L,1) :- boolean(L), atmost(1,L,1).
sharp(L,0) :- boolean(L), length(L,N), M is N-2, atmost(M,L,0).
```

De plus, pour prendre en compte facilement la construction *if-then-else* fonctionnelle du langage, nous implémentons le prédicat `if_then_else`, transformant une contrainte $x = if\_then\_else(b, y, z)$ en $(b \wedge x = y) \vee (\neg b \wedge x = z)$ :

```
if_then_else(Cond, Exp1, Exp2, Result) :-
C#=eval(Cond), E1 #= eval(Exp1), E2#=eval(Exp2), type_bool(C),
R#=eval(Result), and(C#=1,R #= E1) or and(C#=0,R #= E2).
```

## 6.4 Conclusion

Nous avons présenté les aspects d'implémentation de la génération en utilisant la programmation logique par contraintes. Cette implémentation est réalisée dans la nouvelle version de l'outil, appelée Lutess V2. Le chapitre suivant montre une expérimentation avec Lutess V2 sur une étude de cas plus réaliste que l'exemple jouet du climatiseur que nous avons traité jusque là.

# Chapitre 7

# Expérimentations

Dans les chapitres précédents, nous avons présenté les différentes méthodes de génération de test, en se basant sur un exemple simple d'un contrôleur de climatiseur. Ce chapitre porte sur l'évaluation de ces méthodes sur une étude de cas plus réaliste, le contrôleur du niveau d'eau dans une chaudière.

L'exemple du contrôleur du niveau d'eau dans la chaudière a été utilisé par le passé comme une étude de cas commune pour l'évaluation de plusieurs méthodes de spécification formelle [1]. Une implémentation en LUSTRE de la spécification de ce logiciel a été proposée dans [11]. Nous utilisons pour cette étude de cas [34], une version que nous avons complétée avec les noeuds manquants, afin d'obtenir un code exécutable. Cet exemple caractérise bien les logiciels de contrôle-commande, qui sont la cible primaire de l'outil LUTESS. De par la taille et la complexité, ce logiciel est proche d'un logiciel qu'on s'attend à rencontrer dans des applications réelles.

Les objectifs que nous nous fixons pour cette étude sont multiples :

- montrer que les méthodes de génération proposées peuvent passer à l'échelle d'un exemple plus réaliste ;
- préciser les idées sur le processus de spécification et les difficultés d'une telle activité ;
- proposer une méthodologie de modélisation, qui serait applicable dans le cas général.

Nous présentons dans la section 7.1 le contrôleur et sa spécification, tandis que la section 7.2 décrit le processus de modélisation des tests et leur génération. Enfin, nous résumons la méthodologie de test dans la section 7.3.

## 7.1 Présentation du contrôleur de la chaudière

Selon les spécifications informelles données dans [1], le système physique est composé de quatre pompes fournissant de l'eau à une chaudière. Cette eau est transformée en vapeur à la sortie de la chaudière. Pour éliminer les risques que présente une panne ou une situation anormale, le niveau d'eau doit être maintenu à l'intérieur des limites de sûreté. Le but du logiciel est donc de contrôler le niveau d'eau dans la chaudière.

### 7.1.1 Composants de la chaudière

Le système de la chaudière, comme le montre la figure 7.1, est composé des unités physiques suivantes :

- **La chaudière** qui est caractérisée par les éléments suivants :
  - Sa capacité totale $C$ (indiquée en litres).
  - Les limites minimum et maximum $M_1$ et $M_2$ de la quantité d'eau (en litres). En dehors de ces limites, le système va présenter un danger après cinq secondes, suite à un manque ou un débordement d'eau.
  - Les quantités normales $N_1$ et $N_2$ d'eau (en litres) à maintenir dans la chaudière pendant le fonctionnement normal ($M_1 < N_1$ et $N_2 < M_2$).
  - La quantité maximale $W$ de vapeur (en litres/sec) en sortie de la chaudière.
- **Une valve** sert à évacuer l'eau résiduelle de la chaudière au début de son fonctionnement.
- **Quatre pompes** fournissent de l'eau à la chaudière. Chaque pompe est caractérisé par sa capacité ($p$ litres/sec) et son état, "open" ou "closed". Même si une pompe peut être arrêtée instantanément, au démarrage elle a besoin d'un cycle complet pour s'ouvrir à cause du temps nécessaire pour que la pression soit équilibrée.
- **Quatre contrôleurs**(un par pompe) vérifient le bon fonctionnement des pompes (s'il y a écoulement d'eau vers la chaudière ou non).
- **Une unité de mesure de quantité d'eau** mesure la quantité d'eau ($q$) dans la chaudière, en litres.
- **Une unité de mesure d'écoulement** mesurant la quantité du flux ($v$) à la

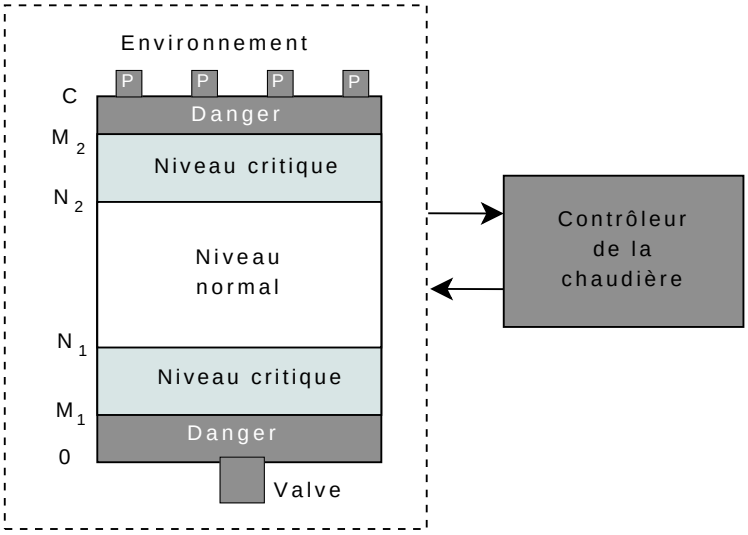


FIGURE 7.1 – Le système de contrôle du niveau d'eau dans la chaudière.

sortie de la chaudière, en litres/sec.

### 7.1.2 Communication entre le système physique et le contrôleur

Le système physique communique avec le contrôleur, un logiciel qui contrôle son fonctionnement, via des messages. Le système physique peut être vu comme l'environnement externe du contrôleur avec lequel il interagit continuellement. Pendant chaque cycle, dont la durée est de cinq secondes, les actions suivantes sont effectuées :

1. le contrôleur reçoit en entrée des messages de l'environnement ;
2. il analyse l'information reçue et calcule les sorties ;
3. et envoie les messages, contenant les sorties calculées, à son environnement - le système physique.

### 7.1.3 Modes de fonctionnement

Selon les messages susceptibles d'être envoyés par l'environnement et les fautes pouvant être détectées, le contrôleur peut opérer dans différents modes.

1. **startup** : Ce mode correspond au démarrage du logiciel, ou aucune erreur critique ou demande d'arrêt n'est détectée. Dans ce mode, le programme attend le message approprié du système physique indiquant qu'il est prêt à fonctionner.

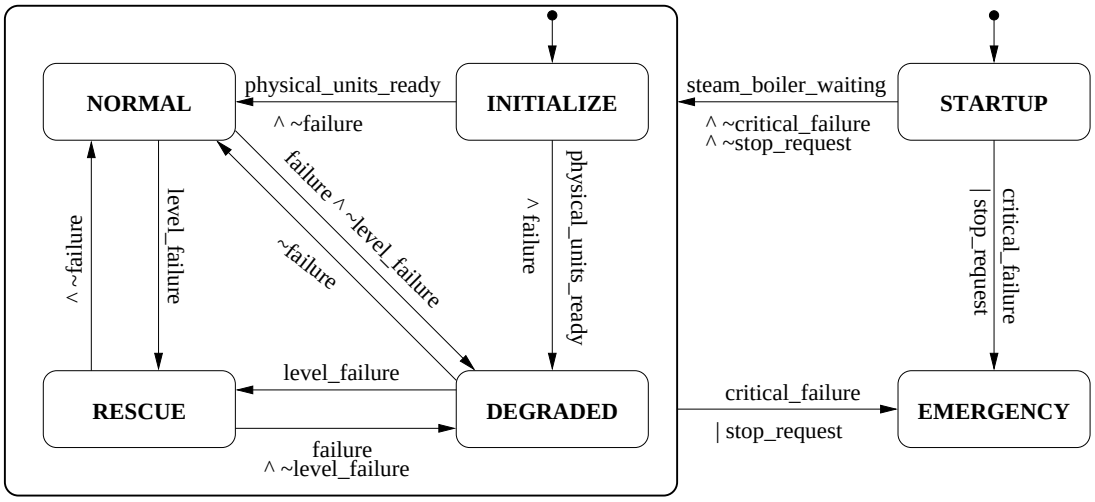


Figure 7.2: Modes de fonctionnement du système.

Si une erreur critique est détectée ou il y a une demande d'arrêt, le programme passe au mode *emergency*.

2. **initialization** : Dès que le système physique est prêt à fonctionner, le programme entre dans un mode d'initialisation. Dans ce mode, le programme envoie continuellement à l'environnement un message indiquant qu'il est prêt à fonctionner ; ceci se passe jusqu'à ce qu'il reçoive une réponse positive de l'environnement. A la réception de ce message, si aucune défaillance n'est détectée, le programme entre dans son mode *normal* de fonctionnement ; autrement, c.-à-d. si une défaillance est détectée dans l'unité physique, le programme entre dans le mode *degraded*.

3. **normal** : Dans ce mode, le programme essaie de maintenir la quantité d'eau à l'intérieur des limites normales, $N_1$ et $N_2$, avec toutes les unités physiques opérant correctement. Si une défaillance dans l'unité d'eau est détectée, le programme entre dans le mode *rescue*, tandis que pour toute autre défaillance, le programme passe au mode *degraded*.

4. **degraded** : Dans ce mode, le programme essaie de maintenir la quantité d'eau à un niveau satisfaisant, malgré le non fonctionnement d'une unité physique autre que celle mesurant le niveau d'eau. Si cette panne est réparée, le programme retourne au mode *normal*. Si une panne de l'unité d'eau est détectée, le programme passe au mode *rescue*.

5. **rescue** : Dans ce mode, le programme essaie de maintenir la quantité d'eau à un niveau satisfaisant, malgré le non fonctionnement de l'unité d'eau. La quantité d'eau dans la chaudière est estimée, à partir de la quantité d'eau qui sort et la quantité d'eau qui rentre par les pompes. Quand la panne est réparée, le programme retourne au mode *normal*, sauf si une panne dans une autre unité physique est détectée, auquel cas le programme passe au mode *degraded*.

6. **emergency** : Le programme entre dans ce mode obligatoirement quand une panne critique est détectée ou alors quand il y a une demande d'arrêt du contrôleur. Une fois entrée dans ce mode, le programme y reste pour toujours.

### 7.1.4 Caractéristiques du contrôleur

Le programme de la chaudière requiert l'échange d'un nombre important de messages entre le contrôleur et les unités physiques du système. Il prend en entrée 27 signaux booléens et 7 valeurs entières et calcule 30 signaux booléens et 8 valeurs entières en sortie. Le noeud principal se compose d'une trentaine de fonctions internes et fait, à plat, 686 lignes de code LUSTRE.

Le tableau 7.1 donne la liste des constantes utilisées pour caractériser les unités physiques et le contrôleur de la chaudière.

| Constante | Valeur | Description |
|---|---|---|
| N_pump | 4 | Le nombre de pompes de la chaudière |
| C | 1000 | La capacité totale de la chaudière |
| M1 | 150 | La limite de sûreté minimale |
| N1 | 400 | La limite normale minimale |
| N2 | 600 | La limite normale maximale |
| M2 | 850 | La limite de sûreté maximale |
| V | 10 | La quantité de flux en sortie de la chaudière |
| W | 25 | La quantité maximale d'eau qui peut sortir de la chaudière |
| P | 15 | La quantité d'eau qui entre par une pompe |
| NB_stop | 3 | Nombre de messages *stop* nécessaires pour arrêter la chaudière |
| Dt | 5 | Durée d'un cycle en secondes |
| startup | 1 | Modes de fonctionnement du contrôleur |
| initialize | 2 | |
| normal | 3 | |
| degraded | 4 | |
| rescue | 5 | |
| emergency | 6 | |
| open | 1 | La valve ou la pompe est ouverte |
| closed | 0 | La valve ou la pompe est fermée |

TABLE 7.1 – Les constantes caractérisant la chaudière

| Paramètre d'entrée | Type | Description |
|---|---|---|
| stop | bool | Permet l'arrêt du contrôleur si vraie pendant 3 instants successifs |
| steam_boiler_waiting | bool | Signale que les unités physiques sont en attente du contrôleur |
| physical_units_ready | bool | Signale que les unités physiques sont prêtes à fonctionner |
| level | int | Représente le niveau d'eau mesurée dans la chaudière |
| steam | int | Représente la quantité de vapeur mesurée à la sortie de la chaudière |
| valve_status | int | Représente l'état d'ouverture de la valve |
| pump_state | int^4 | Représente l'état de chacune des 4 pompes |
| pump_control_state | bool^4 | Signale l'ouverture ou non des pompes tel que observé par leur contrôleurs |
| pump_repaired | bool^4 | Signale la réparation des pompes après la constatation d'une panne |
| pump_control_repaired | bool^4 | Signale la réparation des contrôleurs de pompe après la constatation d'une panne |
| level_repaired | bool | Signale la réparation de l'unité de mesure du niveau d'eau |
| steam_repaired | bool | Signale la réparation de l'unité de mesure de la quantité de vapeur |
| pump_failure_acknowledgement | bool^4 | Signale la réception du message de détection d'une panne dans une pompe |
| pump_control_failure _acknowledgement | bool^4 | Signale la réception du message de détection d'une panne dans un contrôleur |
| level_failure_acknowledgement | bool | Signale la réception du message de détection d'une panne dans l'unité de niveau |
| steam_failure_acknowledgement | bool | Signale la réception du message de détection d'une panne dans l'unité de vapeur |

TABLE 7.2 – Paramètres d'entrée du contrôleur de la chaudière

Le tableau 7.2 donne la liste des entrées du contrôleur de la chaudière.

| Paramètre de sortie | Type | Description |
|---|---|---|
| program_ready | bool | Signale que le contrôleur est prêt à fonctionner |
| mode | int | Représente le mode d'opération du contrôleur |
| valve | bool | Commande l'ouverture/fermeture de la valve |
| q | int | Représente le niveau d'eau supposée dans la chaudière |
| v | int | Représente la quantité de vapeur supposée à la sortie de la chaudière |
| p | int^4 | Représente la quantité d'eau supposée passer par les pompes |
| n_pumps | int | Représente le nombre de pompes qui doivent être ouvertes |
| open_pump | bool^4 | Commande l'ouverture des pompes |
| close_pump | bool^4 | Commande la fermeture des pompes |
| pump_failure_detection | bool^4 | Signale la détection d'une panne des pompes |
| pump_control_failure _detection | bool^4 | Signale la détection d'une panne des contrôleurs de pompe |
| level_failure_detection | bool | Signale la détection d'une panne de l'unité de mesure du niveau d'eau |
| steam_outcome_failure _detection | bool | Signale la détection d'une panne de l'unité de mesure de la quantité de vapeur |
| pump_repaired _acknowledgement | bool^4 | Signale la réception du message de réparation des pompes |
| pump_control_repaired _acknowledgement | bool^4 | Signale la réception du message de réparation des contrôleurs de pompe |
| level_repaired _acknowledgement | bool | Signale la réception du message de réparation de l'unité d'eau |
| steam_outcome_repaired _acknowledgement | bool | Signale la réception du message de réparation de l'unité de vapeur |

TABLE 7.3 – Paramètres de sortie du contrôleur de la chaudière

Le tableau 7.3 donne la liste des sorties du contrôleur.

## 7.2 Génération de données de test pour la chaudière

La fonction principale du contrôleur de la chaudière est de maintenir le niveau d'eau entre les limites données, en se basant sur les entrées reçues de l'environne-

ment. C'est pourquoi, dans le but de tester le contrôleur dans son fonctionnement normal, nous considérons d'abord que toutes les unités se comportent correctement et fournissent les bonnes entrées au contrôleur. Dans un deuxième temps, nous simulons différents pannes des unités de mesure qui sont tolérées par le contrôleur, dans le but de tester sa réaction.

### 7.2.1 Définition du domaine des entrées

Le domaine d'une entrée est l'ensemble des valeurs qui ont un sens et pour lesquelles l'entrée est conçue. Les entrées de type entier sont souvent utilisées dans les contrôleurs pour représenter un état, consistant en un sous-ensemble limité d'entiers, comme dans le cas des variables *valve_status* ou *pump_state*. Dans d'autres cas, comme *level*, elles représentent un intervalle d'entiers. Ces domaines doivent être définis avant que des données de test soient générées.

Nous utilisons les invariants suivants pour définir le domaine de chaque entrée entière du contrôleur [1] (entre parenthèses nous rappelons les valeurs des constantes utilisées) :

- Le niveau d'eau doit être entre 0 et la capacité maximale de $C$ (1000) litres :
  `0 <= level and level <= C`
- Le flux d'eau sortant doit être entre 0 et la capacité maximale de $W$ (25) litres/seconde :
  `0 <= steam and steam <= W`
- La valve doit être fermée (`closed=0`) ou ouverte (`open=1`) :
  `closed <= valve_status and valve_status <= open`
- L'état de chaque pompe doit être fermé ou ouvert :
  `AND(N_pump, closed <= pump_state and pump_state <= open)`
  où *N_pump* (4) est le nombre des pompes ; *AND* est un opérateur booléen appliquant l'expression précédente à la totalité du tableau *pump_state* et résultant en la conjonction des valeurs correspondantes [2].
- L'état de chaque contrôleur de pompe peut être fermé ou ouvert :

1. Ces invariants ont un sens seulement sous la supposition que toutes les unités fonctionnent correctement.
2. Le lecteur pourra trouver en annexe A le noeud définissant l'opérateur AND.

```
AND(N_pump, closed <= pump_controller_state
and pump_controller_state <= open)
```

TABLE 7.4 – Une séquence de test complètement aléatoire.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ |
|---|---|---|---|---|---|---|---|---|---|
| stop | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| steam_boiler_waiting | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| physical_units_ready | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| level | 288 | 218 | 836 | 699 | 626 | 740 | 72 | 985 | 248 |
| steam | 9 | 23 | 19 | 8 | 12 | 19 | 15 | 3 | 12 |
| valve_status | open | open | open | closed | closed | closed | open | open | closed |
| pump_state[0..3] | 1,0,0,1 | 0,0,1,0 | 0,0,0,0 | 1,1,1,1 | 0,0,0,1 | 0,1,0,1 | 0,1,0,1 | 1,0,0,1 | 1,1,1,1 |
| pump_control_state[0..3] | 1,0,0,0 | 0,1,0,1 | 1,1,0,0 | 0,0,1,0 | 0,0,1,0 | 0,0,0,0 | 1,0,0,1 | 1,1,0,0 | 1,0,0,0 |
| level_failure_acknowledgement | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| program_ready | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mode | start | emerg. | emerg. | emerg. | emerg. | emerg. | emerg. | emerg. | emerg. |
| valve | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| q | 288 | 218 | 836 | 699 | 626 | 740 | 72 | 985 | 248 |
| v | 9 | 23 | 19 | 8 | 12 | 19 | 15 | 3 | 12 |
| open_pump[0..3] | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 |
| close_pump[0..3] | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 |

A ce point, le modèle peut être utilisé directement pour générer des données de test aléatoires. Pourtant, comme on peut le constater dans le tableau 7.4, ces séquences ne sont pas très concluantes. Le contrôleur ne peut pas gérer le comportement aléatoire de toutes les unités, et rapidement après le démarrage passe en mode *emergency*. La raison est simple : soit une demande d'arrêt a été envoyée (*stop* étant vrai pour 3 pas consécutifs), soit il y a eu une faute pendant l'initialisation. Notons que, selon la spécification, n'importe quel message reçu quand il n'est pas attendu est considéré comme une faute. C'est pour cela qu'il est nécessaire de définir précisément la dynamique de l'environnement, afin d'observer des exécutions plus réalistes.

### 7.2.2 Définition de la dynamique de l'environnement

La dynamique de l'environnement peut être exprimée par des relations temporelles entre les valeurs courantes et passées des entrées et sorties. Nous pouvons dériver, directement à partir de la spécification, les propriétés qui identifient le comportement correct de toutes les unités.

#### 7.2.2.1 La transmission de messages de contrôle

Au démarrage, dans le mode *startup*, les unités physiques envoient le message *steam_ boiler_ waiting* au contrôleur et ce message ne peut être envoyé que dans ce mode. L'invariant suivant définit ce comportement :

```
steam_boiler_waiting = (false -> (pre mode = startup))
```

Une fois que le contrôleur est prêt à fonctionner, il envoie le message *program_ ready*. La réception de ce message doit être notifiée par les unités physiques, en envoyant le message *physical_ units_ ready* :

```
physical_units_ready = (false -> (pre program_ready))
```

#### 7.2.2.2 L'unité de mesure de la quantité de vapeur

Au démarrage de la chaudière, le robinet est censé être fermé est la quantité de vapeur qui passe supposée nulle :

```
implies(true -> pre(mode=startup), steam = 0)
```

#### 7.2.2.3 Les pompes

Le fonctionnement correct des pompes est caractérisé par leur réaction en conformité avec les demandes d'ouverture et de fermeture. Chaque pompe doit être ouverte (resp. fermée) après réception du message *open_ pump* (resp. *close_ pump*). Nous allons définir ce comportement dans le noeud *correct_ pump_ state* suivant :

```
node correct_pump_state(open_pump, close_pump:bool; pump_state:int)
     returns (ok:  bool);
let
     ok = (pump_state = closed) ->
          if pre(open_pump) then pump_state = open
          else if pre(close_pump) then pump_state = closed
               else pump_state = pre pump_state;
tel;
```

En utilisant ce noeud, on peut appliquer cet invariant à l'ensemble des pompes, à l'aide de l'opérateur *AND* :

```
AND(N_pump, correct_pump_state(open_pump, close_pump, pump_state))
```

#### 7.2.2.4 Les contrôleurs des pompes

Les contrôleurs des pompes indiquent si effectivement il y a de l'eau qui passe par les pompes. Selon la spécification [1], à cause du temps nécessaire pour que la

pression soit équilibrée, l'eau ne commence à passer qu'un cycle après l'ouverture d'une pompe. Pendant la fermeture, l'eau est arrêtée instantanément. L'invariant suivant, applique ce comportement à l'ensemble des pompes :

```
AND(N_pump, pump_control_state = (pump_state = open^N_pump
     and pre(pump_state = open^N_pump)) )
```

#### 7.2.2.5 La valve

L'état de la valve (*valve_status*) change seulement quand le message *valve* est envoyé par le contrôleur :

```
(valve_status=closed) -> (pre valve = (valve_status<>pre valve_status))
```

#### 7.2.2.6 L'unité de mesure du niveau d'eau

Pour simuler un comportement realiste de la chaudière, il faut que la variation du niveau d'eau tienne compte de la quantité d'eau qui entre ou sort de la chaudière. Ce niveau est calculé par rapport au niveau précédent, auquel on ajoute la quantité de l'eau entrant par les pompes ouvertes et on retire la quantité sortante par le robinet ou la valve. L'equation suivante calcule le niveau d'eau attendu :

```
expected_level = level -> pre(expected_level) + Dt*sum_flow(N_pump,
     pump_control_state) - Dt*steam - Dt*valve_status*V ;
```

où le noeud *sum_flow* calcule la somme du flux passant par toutes les pompes, basée sur l'état du contrôleur de chaque pompe :

```
node sum_flow(const n:  int; ctrl_state:  bool^n) returns (res:  int)
var
     flow0:  int;
let
     flow0 = if ctrl_state[0] then P else 0;
     res = with n=1 then flow0 else
          flow0 + sum_flow(n-1,ctrl_state[1..n]);
tel;
```

Si le niveau attendu est négatif, cela signifie qu'il ne reste plus d'eau dans la chaudière (*level=0*). Si le niveau attendu est plus grand que la capacité totale de la chaudière, cela signifie que la chaudière est pleine et l'eau ne passe plus par les pompes (*level=C*). Si le niveau attendu se trouve entre *0* et *C*, il représente la quantité d'eau actuelle (*level=expected_level*) :

```
true -> if expected_level < 0 then level=0 else
     if expected_level > C then level=C else level=expected_level
```

#### 7.2.2.7 Gestion des pannes

En supposant que toutes les unités se comportent correctement, normalement il ne doit y avoir aucune panne dans les unités physiques. Malgré cela, si jamais le contrôleur suppose qu'il y a une panne et envoie un message de détection de panne, l'unité physique concernée doit répondre correctement en confirmant la réception d'un message et en envoyant un message de réparation.

**Confirmation du message de détection d'une panne :** Quand un message d'erreur est envoyé par le contrôleur, l'unité physique confirme au contrôleur la réception du message. Ce comportement est appliqué à l'ensemble des unités physiques, à l'aide du noeud *ack_failure* suivant :

```
node ack_failure(failure_ack, failure_detection : bool) returns
(ok :bool) ;
let
     ok = failure_ack = (false -> pre failure_detection) ;
tel ;
```

Les messages de détection des pannes dans les pompes sont représentés par le tableau de booléens *pump_failure_ detection*, alors que la confirmation de réception du message par le tableau *pump_failure_ acknowledgement*. L'invariant suivant applique le noeud *ack_failure* à l'ensemble des pompes, à l'aide de l'opérateur *AND* :

```
AND(N_pump, ack_failure(pump_failure_acknowledgement,
pump_failure_detection)),
```

De manière analogue aux pompes, on applique ce comportement aux contrôleurs des pompes avec l'invariant suivant :

```
AND(N_pump, ack_failure(pump_control_failure_acknowledgement,
pump_control_failure_detection)),
```

Il en est de même pour les unités de mésure du niveau d'eau et de la quantité de vapeur en sortie, qu'on définit avec les invariants suivants :

```
ack_failure(level_failure_acknowledgement, level_failure_detection),

ack_failure(steam_failure_acknowledgement,
steam_outcome_failure_detection)
```

**Envoi du message de reparation :** Après la réception d'un message de détection de panne et sa confirmation, chaque unité physique envoie le message de réparation tant que le contrôleur n'a pas confirmé la réception de celui-ci. Ce comportement est implémenté dans le noeud *repair_ until_ ack* :

```
node repair_until_ack(repair, failure_ack, repair_ack : bool)
returns (ok : bool) ;
let
     ok = repair = (false -> pre(repair) and not(pre repair_ack)
          or pre failure_ack) ;
tel ;
```

Les quatre propriétés invariantes qui suivent, appliquent ce comportement respectivement à l'ensemble des pompes, l'ensemble des contrôleurs de pompes, l'unité de mésure du niveau d'eau et la quantité de vapeur en sortie :

```
AND(N_pump, repair_until_ack(pump_repaired,
pump_failure_acknowledgement, pump_repaired_acknowledgement)),

AND(N_pump, repair_until_ack(pump_control_repaired,
pump_control_failure_acknowledgement,
pump_control_repaired_acknowledgement)),

repair_until_ack(level_repaired, level_failure_acknowledgement,
level_repaired_acknowledgement),

repair_until_ack(steam_repaired, steam_failure_acknowledgement,
steam_outcome_repaired_acknowledgement)
```

#### 7.2.2.8 Génération de données de test

Le tableau 7.5 montre un extrait d'une séquence de test résultant de ce modèle. Afin d'éviter un arrêt prématuré du système, nous ajoutons l'invariant `not(stop)`. Cette séquence simule un fonctionnement normal de la chaudière, avec l'ensemble des unités physiques qui fonctionnent correctement. On peut remarquer qu'à l'instant $t_1$,

TABLE 7.5 – Extrait d'une séquence de test simulant l'environnement physique sans pannes.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_6$ | $t_7$ | $t_8$ | $t_9$ | $t_{10}$ | $t_{11}$ | $t_{1000}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| stop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| steam_boiler_waiting | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| physical_units_ready | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| level | 964 | 964 | 859 | 804 | 519 | 434 | 389 | 374 | 329 | 459 | 509 |
| steam | 0 | 0 | 11 | 1 | 13 | 17 | 9 | 3 | 24 | 4 | 22 |
| valve_status | closed | closed | open | open | open | closed | closed | closed | closed | closed | closed |
| pump_state[0..3] | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 1,0,0,0 | 1,1,0,0 | 1,1,1,0 | 0,0,0,0 |
| pump_control_state[0..3] | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 1,0,0,0 | 1,1,0,0 | 0,0,0,0 |
| program_ready | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mode | start | init | init | init | init | init | normal | normal | normal | normal | normal |
| valve | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| q | 964 | 964 | 859 | 804 | 519 | 434 | 389 | 374 | 329 | 459 | 509 |
| v | 0 | 0 | 11 | 1 | 13 | 17 | 9 | 3 | 24 | 4 | 22 |
| open_pump[0..3] | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 0,0,0,0 | 1,0,0,0 | 0,1,0,0 | 0,0,1,0 | 0,0,0,0 | 0,0,0,0 |

le contrôleur demande l'ouverture de la valve, à cause du niveau élevé d'eau. La valve est maintenue ouverte tant que le niveau n'a pas atteint une valeur comprise entre les limites de sécurité ($t_6$). Par la suite, une fois que le message `physical_units_ready` est reçu, le contrôleur passe dans le mode *normal* et continue l'exécution en restant dans ce mode (instants $t_8$ à $t_{1000}$), puisqu'aucune faute n'est signalée par les unités physiques.

### 7.2.3 Simulation avec des scénarios spécifiques

Les séquences de test générées précédemment sont obtenues en ajoutant systématiquement tous les invariants qui définissent le fonctionnement correct des unités physiques de la chaudière. Ceci peut être vue comme une exécution normale du logiciel. Mais il est souhaitable que l'on puisse aussi mettre le logiciel dans un scénario d'exécution anormale. Dans LUTESS, des scénarios spécifiques d'exécution peuvent être obtenus soit avec des propriétés invariantes, soit en spécifiant des probabilités conditionnelles.

Dans la section précédente, en utilisant une propriété invariante, nous avons forcé l'entrée *stop* à être toujours fausse, ce qui peut être vu comme un scénario simple : "la chaudière n'est jamais arrêtée". Si nous voulons tester si le logiciel se comporte correctement dans le cas d'un arrêt, il est intéressant de permettre "de temps en temps" l'entrée *stop* à être vraie, en spécifiant une probabilité faible :

```
prob(true, stop, 0.05) ;
```

Rappelons que l'expression *prob(C,E,P)* signifie que si la condition $C$ est remplie, alors la probabilité que l'expression $E$ soit vraie est égale à $P$. Avec la spécification de cette probabilité, les séquences obtenues doivent comporter des occurrences rares du message *stop* et généralement le système peut être observé pour quelque temps, avant d'être arrêtée tardivement dans le processus de test.

#### 7.2.3.1 Simulation des pannes

Différentes pannes du système peuvent être simulées par des probabilités. Pour faire cela, les propriétés invariantes spécifiées dans les sections 7.2.1 et 7.2.2 peuvent être remplacées par des expressions *prob*. Par exemple :

- Une panne possible du système peut intervenir dans le mode *initialize*, si on considère qu'il y a une faible probabilité de recevoir le message *physical_ units_ ready*, au moment attendu :

  ```
  prob(true, physical_units_ready = (false -> (pre program_ready)),
  0.2) ;
  ```

- Une autre panne consiste en le changement de l'état de la valve, même si aucune commande n'est envoyée à la valve :

  ```
  prob(true, (valve_status=closed) ->
       (pre valve = (valve_status=pre valve_status)), 0.2) ;
  ```

Evidemment, cette liste n'est pas exhaustive. On peut remplacer la plupart des propriétés invariantes (c.à.d. propriété définies par l'opérateur *environment*) par une expression probabiliste (*prob*). La valeur de la probabilité assignée est déterminée empiriquement par le testeur.

#### 7.2.3.2 Simulation d'une panne dans l'unité de mesure du niveau

Dans ce qui suit, nous montrons une simulation plus avancée, d'une panne non signalé de l'unité de mesure du niveau d'eau. Quand cette unité fonctionne correctement, la valeur mésurée ne peut être négative ni dépasser la capacité totale $C$ de la chaudière. Pour simuler une unité défectueuse, nous laissons une possibilité d'apparition de valeurs quelconques, au moyen de probabilités conditionnelles.

Pour faire cela, nous enlevons d'abord les contraintes sur le domaine de l'entrée *level*. Ensuite, nous considérons l'invariant spécifiée précédemment pour limiter la

valeur courante du niveau :

```
level_inv = true -> if expected_level < 0 then level=0 else
     if expected_level > C then level=C else level = expected_level ;
```

On garde les mêmes conditions tandis que le contrôleur n'est pas dans le mode *normal* :

```
implies(true -> pre(mode)<>normal, 0<=level and level<=C -> level_inv) ;
```

Et, pendant que le contrôleur se trouve dans le mode *normal*, nous introduisons la probabilité suivante :

```
prob( false -> pre(mode)=normal, level_inv, 0.9 ) ;
```

TABLE 7.6 – Extrait d'une séquence de test simulant une unité défectueuse.

| | $t_7$ | $t_8$ | $t_9$ | $t_{10}$ | $t_{11}$ | $t_{12}$ | $t_{13}$ | $t_{14}$ | $t_{15}$ | $t_{16}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| physical_units_ready | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| level | 579 | 519 | 509 | 464 | 429 | 334 | -1069082252 | 389 | 514 | 574 |
| level_repaired | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| level_failure_acknowledgement | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| mode | init | normal | normal | normal | normal | normal | rescue | rescue | normal | normal |
| q | 579 | 519 | 509 | 464 | 429 | 334 | 279 | 389 | 514 | 574 |
| level_failure_detection | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| level_repaired_acknowledgement | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Le tableau 7.6 montre un extrait d'une séquence de test générée avec cette simulation de panne d'unité et la réaction correspondante du contrôleur. A l'instant $t_{13}$, une valeur arbitraire pour *level* a été générée et le contrôleur détecte une faute dans l'unité de mesure du niveau (caractérisé par la valeur *level_failure_detection=1*). On peut remarquer que le mode a changé en *rescue* et le niveau d'eau a été estimé (*q=279*). A l'instant prochain ($t_{14}$), la reception du message de détection d'une panne est confirmée (*level_failure_acknowledgement=1*). Ensuite ($t_{15}$), un message de réparation est envoyé (*level_repaired=1*) et le contrôleur retourne dans le mode *normal*.

### 7.2.4 Test basé sur les propriétés de sûreté

Les propriétés de sûreté sont spécifiées dans un *testnode* à l'aide de l'opérateur *safeprop*. Considérons, par exemple, la propriété : "l'envoi d'un message stop

Table 7.7: Extrait d'une séquence de test guidée par une propriété de sûreté.

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ | $t_9$ | $t_{10}$ | $t_{11}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| stop | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| physical_units_ready | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| mode | start | init | init | init | init | init | normal | normal | normal | emerg. | emerg. | emerg. |

pendant 3 instants consécutifs mène le contrôleur dans le mode *emergency*". Pour exprimer formellement cette propriété, nous utilisons deux variables booléennes locales *pre_stop* et *pre_pre_stop*, se référant à la valeur de *stop* pour les 2 instants passés :

```
pre_stop = false -> pre stop ;
pre_pre_stop = false -> pre pre_stop ;
```

Ensuite, nous introduisons cette propriété avec l'opérateur *safeprop* :

```
safeprop( implies(false -> pre(mode)=normal and stop and pre_stop and
pre_pre_stop, mode=emergency)) ;
```

Le test guidé par les propriétés de sûreté consiste à enlever, quand cela est possible, les valeurs des entrées qui à l'évidence ne peuvent pas violer la propriété. En effet, sa violation requiert au moins que la partie gauche de l'implication soit vraie. Le tableau 7.7 présente l'effet de cette spécification sur la séquence générée. Immédiatement après que le contrôleur soit passé dans le mode *normal*, la valeur de *stop* est mise à vrai pour les 3 instants suivants. Cette séquence est générée avec une recherche des états suspects selon la stratégie d'intersection, sur les sous-chemins de longueur 3.

### 7.2.5 Observations

L'objectif de cette étude de cas était d'évaluer la difficulté de la modélisation des tests ainsi que la complexité de la génération de données de test en utilisant l'outil Lutess V2.

### Spécification de modèles

L'étude a montré que des modèles de test pour le contrôleur de la chaudière n'étaient pas difficiles à obtenir. La modélisation de l'environnement de la chaudière a requis quelques jours de travail. Toutefois, l'effort nécessaire pour une opération de test complète n'est pas facile a évaluer car il dépend de la précision voulue des séquences de test pouvant mener le testeur à l'écriture de plusieurs scénarios correspondant à des situations différentes (et des noeuds de test différents). Il faut remarquer que la création d'un nouveau *testnode* pour la génération d'un nouvel ensemble de séquences de test demande habituellement une modification légère d'un *testnode* précédent. Il est donc facile de générer un grand nombre de séquences de test avec peu d'effort.

Les noeuds de test que nous avons créés comportent environ 20 propriétés invariantes qui modélisent l'environnement de la chaudière. A ceci s'ajoutent, selon le cas, les différents probabilités conditionnelles et propriétés de sûreté. La taille moyenne d'un *testnode*, avec les noeuds intermédiaires, avoisine les 200 lignes de code LUSTRE.

### Contraintes générées

Pour générer des données de test, les spécifications sont d'abord compilées en une machine à états finis, représentée en contraintes. Après la définition de la dynamique de l'environnement (cf. 7.2.2), selon le cas, cette machine comporte entre 63 et 65 variables d'état, dont 7 de type entier. La contrainte *environment*, met en relation ces variables d'état, complètement connues lors de la génération, avec les 34 variables d'entrée. La contrainte *transition*, implique plus de 200 variables et comporte une relation pour la définition de chaque variable d'état. Dans le cas d'une recherche des vecteurs pertinents sur une profondeur de 3 instants lors du guidage par rapport d'une propriété de sûreté (cf. 7.2.4), les variables et les contraintes postées sont triplés. Les fichiers de contraintes générés font entre 300 - 350 lignes de codes Prolog ECLiPSe [3].

3. http ://www.eclipse-clp.org

### Complexité à l'exécution

Le temps nécessaire pour générer une séquence de test n'est pas facile à quantifier généralement. Le temps de génération est linéaire par rapport à la longueur de la séquence. En revanche, le temps nécessaire pour générer un seul vecteur d'entrées dépend étroitement de la complexité de la spécification. Pour les modèles de test que nous avons montré ici, la génération prend moins de 30 secondes pour une séquence d'une centaine de pas (les tests sont fait sur une machine Linux Fedora 9, Intel Pentium 2GHz avec 1GB de mémoire).

## 7.3 Une méthodologie de modélisation pour le test

La génération de séquences de test avec LUTESS requiert une modélisation de l'environnement du système. Pour cette étude de cas, nous avons commencé par définir d'abord les domaines pour les entrées entières du contrôleur. Dans un deuxième temps, nous avons défini les propriétés temporelles de l'environnement du logiciel qui caractérisent un fonctionnement sans pannes des différentes unités physiques du système. Ensuite, nous avons défini des scénarios plus précis, soit dans le but de simuler un comportement réaliste, soit pour simuler des pannes. Enfin, les propriétés de sûreté sont utilisées pour guider la génération avec l'objectif de viser leur violation.

Même si les modèles sont spécifiques à chaque logiciel sous test, le processus de modélisation et de test peut généralement suivre cette même approche incrémentale dont les différentes phases sont :

1. **Définition du domaine** : Cette phase concerne la définition du domaine pour les entrées de type entier, permettant seulement des valeurs significatives. Dans le cas de la chaudière, par exemple, le niveau d'eau ne peut pas être négatif ou dépasser la capacité de la chaudière. On peut définir ces domaines à l'aide de l'opérateur *environment*, utilisant autant d'invariants qu'il en faut. Des séquences aléatoires simples peuvent être générées directement à partir de la spécification résultante, même si leur aptitude à détecter des erreurs est généralement limitée.
2. **Dynamique de l'environnement** : Cette phase consiste en la spécification

des différentes relations temporelles entre les entrées courantes et les valeurs passées des entrées/sorties. Ces relations sont souvent des contraintes physiques de l'environnement. Par exemple, nous avons spécifié que la variation du niveau d'eau dans la chaudière est liée au volumes d'eau entrant et sortant de la chaudière. Les séquences de test qui sont générées à partir de cette spécification correspondent au test aléatoire, sans objectif de test particulier, mais considérant toutes les séquences d'entrées permises par l'environnement.

3. **Scénarios** : En ayant un objectif de test spécifique, le testeur peut spécifier des scénarios plus précis à exécuter. Ces scénarios peuvent être créés en ajoutant des propriétés temporelles invariantes ou en spécifiant des probabilités conditionnelles. L'opérateur *prob* de LUTESS fournit un moyen pour spécifier différents probabilités conditionnelles. Elles peuvent être utilisées soit pour forcer le générateur de données de test à avoir un comportement réaliste, soit pour simuler des pannes. Par exemple, nous avons diminué la probabilité que *stop* soit vraie, afin de générer des séquences de test qui maintiennent le contrôleur en marche plus longtemps. Les séquences de test générées avec cette spécification ont pour but de satisfaire un objectif de test précis.

4. **Test à partir des propriétés de sûreté** : Cette phase utilise des propriétés de sûreté définies formellement pour guider la génération vers la violation de la propriété donnée. La génération résultante va empêcher la génération des entrées qui, étant donnée l'expression de la propriété, à l'évidence ne peuvent pas mener à une violation. Pour être efficace, ce guidage requiert que les propriétés de sûreté soient données comme une relation temporelle entre les entrées et les sorties du logiciel sous test. Les entrées générées ne mènent pas forcement vers une violation de la propriété, car l'ensemble des sorties est considéré comme possible. La connaissance de liens entre les entrées et sorties impliquées dans une propriété de sûreté peut augmenter considérablement l'efficacité de ce guidage. Ces liens peuvent être introduits dans la génération comme des hypothèses de test. Les hypothèses de test peuvent provenir du code du programme ou d'une abstraction de celui-ci ; d'une propriété de sûreté qu'on considère comme étant bien testée et qu'on peut supposer vérifiée

dans le cadre du test incrémental ou elle peuvent être simplement le fruit de l'expérience du testeur.

# Chapitre 8

# Conclusions et perspectives

## 8.1 Bilan de la thèse

Le travail que nous venons de présenter s'inscrit dans le cadre de développement de techniques et outils formels pour la vérification. Il fait suite aux différents travaux sur le test des logiciels synchrones [15,32,36,55,62] autour de l'outil LUTESS, menés dans l'équipe VASCO depuis 1994. Les techniques de test qui résultent de ces travaux se limitent à un cadre strictement booléen. Ce travail a pour but de répondre aux besoins de prendre en compte les logiciels comprenant des entrées/sorties numériques dans les techniques de test de l'outil LUTESS. Il propose une méthode de génération et l'adaptation des différentes techniques de test dans ce nouveau contexte. De plus, il contribue à l'intégration de ces différentes techniques de test, développées séparément par le passé.

Nous avons défini des extensions syntaxiques permettant l'utilisation des variables de type entier et des relations entre ces entiers dans les expressions des opérateurs de test [49,50]. Ensuite, nous avons donné les principes de la transformation d'une spécification en une machine à états finis et la représentation de celle-ci par un ensemble de contraintes. Nous avons par la suite proposée une méthode de génération de tests à partir de ces spécifications, en précisant les algorithmes qui s'appuient sur la programmation par contraintes pour déterminer les entrées valides par rapport à la spécification.

Dans le cadre de cette extension, nous avons adapté la génération guidée pour des propriétés de sûreté incluant des entiers. La notion des vecteurs pertinents a été définie par un ensemble de contraintes et sa recherche se fait en éliminant les entrées

qui ne respectent pas ces contraintes. Les stratégies de recherche sont alors définies en choisissant des valeurs de vérité pour la propriété de sûreté dans tous les états du chemin recherché.

Nous avons introduit dans le processus de génération, la notion d'hypothèses de test qui s'expriment sous la forme de propriétés du logiciel sous test. Ces hypothèses représentent des connaissances ou des suppositions sur le logiciel. Elles permettent de donner des relations liant les entrées aux sorties et peuvent ainsi contribuer à l'augmentation de l'efficacité du guidage par les propriétés de sûreté. Si le logiciel entier est introduit en hypothèse, la recherche de vecteurs pertinents s'apparente à une preuve locale. En effet, si dans ce cas la résolution des contraintes termine, on peut soit trouver des entrées qui mènent à une violation de la propriété de sûreté soit conclure que la propriété ne peut pas être violée dans le voisinage de cet état.

Nous étendons la notion de probabilités conditionnelles pour qu'elles puissent porter sur des expressions quelconques et non seulement sur des entrées booléennes du logiciel sous test. Ainsi, il est possible de spécifier facilement des scénarios d'exécutions évoluées et viser un objectif de test particulier.

Enfin, nous avons défini un modèle englobant l'ensemble des techniques de test proposées dans ce travail, permettant ainsi leur utilisation conjointe dans une même spécification. Malgré le fait que la cohérence de ces spécifications n'est pas vérifiée statiquement, elles donnent au concepteur de tests une liberté supplémentaire pendant la spécification.

Les méthodes de génération que nous avons proposées dans ce manuscrit ont été implémentées dans une nouvelle version de l'outil, appelée Lutess V2 [51]. L'outil transforme automatiquement une spécification en un ensemble de contraintes décrivant la machine à états finis correspondante. Ces contraintes sont ensuite utilisées par les algorithmes de génération de données de test, implémentés en programmation logique par contraintes.

Cet outil a été utilisé pour mener une expérimentation sur une étude de cas connue, le contrôleur du niveau d'eau dans une chaudière, dans le but d'évaluer les méthodes de test dans un contexte plus réaliste [34]. Le processus de construction des modèles de test a été décrit et des recommandations ont été données sur la définition de scénarios d'exécution, l'utilisation des probabilités conditionnelles, ainsi que la

génération de test guidée par les propriétés de sûreté. Suite à cette étude de cas, nous avons suggéré une méthodologie de modélisation par phases, applicable dans un cas plus général de test des applications synchrones.

Même si d'autres études de cas sont nécessaires pour évaluer l'applicabilité et le passage à l'échelle, les expérimentations que nous avons menées suggèrent que la méthodologie, ainsi que l'outil développé, peut être utilisée pour des applications industrielles de la vie réelle.

## 8.2 Évolutions et perspectives

Le principal objectif du travail présenté dans ce manuscrit était de montrer que la génération de données de test peut être réalisée à l'aide de la programmation par contraintes. De nombreuses questions restent cependant ouvertes ; certaines constituent des perspectives immédiates alors que d'autres nécessitent une exploration plus approfondie.

Parmi les questions ouvertes qui nécessitent plus d'études, citons les aspects liés à la vérification statique de la cohérence des spécifications. Nous avons montré qu'une spécification ne peut pas toujours être satisfaite, à cause des incohérences qu'elle peut contenir. Il arrive que ces incohérences soient détectées dynamiquement lorsque dans un état donné, aucune solution à l'ensemble des contraintes ne peut être trouvée. Généralement, il est impossible de vérifier statiquement une spécification, car ceci consisterait à vérifier que dans tous les états accessibles il existe au moins une solution. Les états accessibles ne peuvent pas être déterminés car, outre la complexité que représente le problème d'accessibilité, les transitions de la machine ne sont pas connues. Pourtant, il serait utile, dans certains cas, de vérifier certaines propriétés de la spécification. Par exemple, lorsque plusieurs probabilités sont spécifiées, il serait intéressant de vérifier si les conditions de ces probabilités sont disjointes deux à deux. Ceci consisterait à chercher un état dans lequel les conditions des deux probabilités seraient vérifiées. Notons que si un tel état est trouvé, ceci ne veut pas dire que les deux probabilités sont incohérentes : il faudrait encore que cet état soit accessible.

Les perspectives immédiates peuvent être envisagées dans deux directions : d'une part, la génération interactive de données de test et d'autre part, la prise ne compte

de variables réelles. Dans la suite de cette section, nous donnerons quelques éléments qui permettent d'alimenter la réflexion autour de ces questions.

### 8.2.1 Génération interactive de données de test

La génération interactive de données de test porte sur la manière de détermination des données concrètes de test. L'idée consiste à donner la possibilité à l'utilisateur de choisir des entrées qui respectent la spécification, au lieu qu'elles soient générées automatiquement. Ainsi, le testeur peut se baser sur son intuition pour guider la génération vers un objectif de test précis.

En effet, à chaque instant pendant la génération, un ensemble de contraintes sont postées avant de choisir une donnée de test. Ces contraintes, réduisent les domaines des variables d'entrée et le choix de données de test (méthodes *choose*) consiste à affecter une valeur dans ce domaine réduit, généralement d'une manière aléatoire.

Le processus de génération interactive interviendrait à ce point : au lieu de choisir aléatoirement les valeurs, on donnerait la possibilité à l'utilisateur de choisir la valeur d'une des variables d'entrée dans le domaine réduit de la variable. Ensuite, cette valeur peut être introduite comme une contrainte supplémentaire, réduisant si possible les domaines des autres variables. L'utilisateur pourrait alors choisir la valeur d'une autre variable, ou décider de générer automatiquement les valeurs des variables restantes.

Choisir interactivement les valeurs à tous les instants de la génération peut s'avérer très difficile, alors que seulement certaines situations peuvent être intéressantes pour l'utilisateur. Pour qualifier les états de l'environnement qui correspondent à ces situations, on peut envisager la spécification d'une condition sous laquelle la génération serait interactive. Ainsi, à chaque instant, on pourrait vérifier cette condition et décider si la génération doit se faire interactivement ou automatiquement.

Considérons par exemple, l'introduction d'un nouveau mot clef *interactive* et l'ajout dans la spécification de la directive de test suivante :

```
interactive(false -> pre(x > y)) ;
```

Dans ce cas, la génération serait interactive seulement lorsque à l'instant précédent $x$ était strictement supérieur à $y$. Autrement, les valeurs seraient générées automatiquement.

Si on s'intéresse à tous les états de l'environnement et on veut intervenir à tous les instants, on peut spécifier :

```
interactive(true) ;
```

alors que l'absence de cette directive, signifierait que le test doit se faire d'une manière complètement automatique et serait équivalent à :

```
interactive(false) ;
```

### 8.2.2 Prise en compte des variables réelles

De même manière que les variables entières, des variables de type réel (à virgule flottante) peuvent faire partie de l'interface d'un logiciel. Il serait donc intéressant de pouvoir générer des données de test pour ces logiciels.

Reprenons une dernière fois notre exemple du contrôleur du climatiseur, pour illustrer l'utilisation de variables réelles. Dans la figure 8.1, la température de sortie *Tsort* et la température ambiante *Tamb* sont représentées par des variables réelles, alors que la température choisie par l'utilisateur *Tuser* est toujours représentée avec une valeur entière. Le type *real* du langage LUSTRE, permet de spécifier des variables de type réel à virgule flottante.

```
testnode Env_real(En_marche : bool ; Tout : real)
     returns (Bouton : bool ; Tamb : real ; Tuser : int)
var dTamb : real ;
let
     dTamb = 0.0 -> Tamb - pre Tamb ;
     environment(
          Tamb >= -20.0 and Tamb <= 60.0,
          Tutil >= 10 and Tutil <= 40,
          dTamb >= -1.5 and dTamb <= 1.5
     ) ;
     prob( false -> pre(Tsort > Tamb), true -> Tamb>pre Tamb, 0.8 ) ;
     prob( false -> pre(Tsort < Tamb), true -> Tamb<pre Tamb, 0.8 ) ;
tel
```

FIGURE 8.1 – Exemple du climatiseur avec des variables réelles

Les principes de génération et les contraintes que nous avons spécifiés dans le chapitre 5 restent valables pour ces spécifications. Mais, les variables de type réel et les contraintes associées nécessitent un solveur adéquat, autre que celui des contraintes sur domaine fini. Pour représenter une spécification en contraintes, il faut distinguer

les opérateurs utilisés en fonction du type des opérandes. Par exemple, une comparaison de deux réels ne peut pas être représentée avec la même contrainte que la comparaison de deux entiers.

La différence principale de l'arithmétique à point flottant par rapport à l'arithmétique des nombres réels se situe dans le fait qu'elle peut être seulement approximative. En effet, la représentation en précision finie signifie qu'une valeur réelle peut seulement être représentée par une valeur approximative ; des erreurs d'arrondi peuvent aussi apparaitre pendant les calculs. L'arithmétique des intervalles est une solution possible à ce problème : chaque nombre réel est borné par une paire de valeurs à point flottant. La valeur du nombre réel peut ne pas être connue, mais définitivement elle doit se situer entre ces deux bornes. Les opérations arithmétiques sont alors effectuées sur ces bornes, en élargissant cet intervalle pour inclure les erreurs d'arrondi possibles. L'inconvénient principal de cette approche est qu'on ne peut plus décider simplement si une contrainte est satisfaite ou non. Par exemple, s'il est connu que $x$ se situe dans l'intervalle (1.2..1.4) et $y$ dans (1.1..1.3), on ne peut rien décider sur la contrainte $x \geq y$.

Notons que la libraire *ic* (Interval Constraint) de l'environnement ECLiPSe, que nous avons utilisée pour la réalisation de la nouvelle version de LUTESS, est un solveur hybride de contraintes sur les intervalles d'entiers/réels. L'avantage de cette librairie consiste en le partage des variables entre les deux résolveurs de contraintes. De plus, il représente les valeurs réelles par une paire de nombres réels représentant les bornes minimale et maximale de la valeur réelle, incluant ainsi la marge d'erreur liée aux approximations.

Nous avons donné dans l'annexe D la machine à état finis correspondant à l'exemple de la figure 8.1, ainsi qu'une tentative de représentation de celle-ci en contraintes. Les contraintes portant sur des réels sont préfixées par le signe "$" alors que celles portant sur les entiers par le signe "#". Pour générer des données de test avec cet exemple, il faut décider d'une manière d'affectation aléatoire pour les variables réelles. Nous avons utilisé pour les variables réelles de cet exemple une méthode simple, présentée dans D.3. Dans le cas général, les variables réelles peuvent aussi avoir un domaine infini quand elles ne sont pas suffisamment contraintes. Au delà de ces expérimentations, la prise en compte des variables réelles nécessite une

étude plus approfondie en termes de propagation des contraintes, d'approximations liées aux calculs et d'instanciation aléatoire.

# Bibliographie


[1] Jean-Raymond Abrial. Steam-Boiler Control Specification Problem. In *Formal Methods for Industrial Applications*, pages 500–509, Juin 1995.

[2] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27 :509–516, Juin 1978.

[3] A. Arnould, B. Marre, and P. Le Gall. Génération automatique de test à partir de spécifications de structures de données bornées. *Techniques et Sciences Informatique*, 18(3) :297–321, Mars 1999.

[4] A. M. Beitz and T. P. Rout. A tool for generating test cases. *Australian Software Engineering Conference*, 1993.

[5] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9) :1270–1282, Septembre 1991.

[6] 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.

[7] Benjamin Blanc, Guy Durrieu, Abdesselam Lakehal, Odile Laurent, Bruno Marre, Ioannis Parissis, Christel Seguin, and Virginie Wiels. Automated functional test case generation from data flow specifications using structural coverage criteria. In *3rd European Congress on Embedded Real Time Software (ERTS2006)*, Toulouse, France, Janvier 2006.

[8] B.W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[9] F. Boussinot and R. De Simone. The Esterel Language. *Proceedings of the IEEE*, 79(9) :1293–1304, Septembre 1991.

[10] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A Declarative Language for Programming Synchronous Systems. In *POPL*, pages 178–188, 1987.

[11] T. Cattel and G. Duval. The Steam Boiler problem in Lustre. *Formal Methods for Industrial Applications*, pages 149–164, Juin 1996.

[12] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool Support for System Specification, Development and Verification*, pages 48–61. Advances in Computer Science. Springer Verlag, Juin 1998.

[13] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th IEEE International Symposium on Software Reliability Engineering*, Paderborn, Germany, Novembre 1998.

[14] L. du Bousquet and N. Zuanon. An Overview of Lutess : A Specifiction-based Tool for Testing Sychronous Software. In *14th International Conference on Automated Software Engineering*, pages 208 – 215, Cocoa Beach, USA, Octobre 1999. IEEE.

[15] Lydie Du Bousquet. *Test fonctionnel statistique de systèmes spécifiés en Lustre.* Thèse, Université Joseph Fourier, Grenoble, France, Septembre 1999.

[16] W. R. Elmendorf. Cause-effect graphs in Functional Testing. Technical report, IBM Software Development Division, New York, USA, 1973.

[17] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Trans. Software Eng.*, 1(2) :156–173, 1975.

[18] Arnaud Gotlieb. *Génération automatique de cas de test structurel avec la Programmation Logique par Contraintes.* PhD thesis, Université de Nice-Sophia Antipolis, Nice, France, Janvier 2000.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et Vérification des Systèmes Réactifs : le langage LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[20] 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.

[21] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In *AMAST*, pages 83–96, 1993.

[22] Richard G. Hamlet. Theoretical Comparison of Testing Methods. In *Symposium on Testing, Analysis, and Verification*, pages 28–37, 1989.

[23] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *POPL*, 1987.

[24] Abdesselam Lakehal. *Critères de Couverture Structurelle pour les Programmes Lustre*. Thèse, Université Joseph Fourier, Grenoble, France, Septembre 2006.

[25] Jean-Claude Laprie. *Guide de la Sûreté de Fonctionnement*. Cépaduès, 1995.

[26] 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, Septembre 1991.

[27] B. Legeard and F. Peureux. Generation of Functional Test Sequences from B Formal Specifications - Presentation and Industrial Case-Study. In *16th International Conference on Automated Software Engineering*, San Diego, USA, 2001.

[28] Bruno Marre and Agnès Arnould. Test Sequences Generation from LUSTRE Descriptions : GATeL. In *Automated Software Engineering*, pages 229–237, 2000.

[29] C. Mazuet. *Stratégies de Test pour des Programmes Synchrones - Application au Langage LUSTRE*. Thèse, Toulouse, France, Decembre 1994.

[30] John D. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Softw.*, 10(2) :14–32, 1993.

[31] G. J. Myers. *The Art of Software Testing*. 1979.

[32] F. Ouabdesselam and I. Parissis. Testing Safety properties of Synchronous Critical Reactive Software. In *7th International Software Quality Week*, San Francisco, USA, Mai 1994.

[33] F. Ouabdesselam and I. Parissis. Constructing operational profiles for synchronous critical software. In *6th International Symposium on Software Reliability Engineering*, pages 286–293, Toulouse, France, Octobre 1995.

[34] V. Papailiopoulou, B. Seljimi, and I. Parissis. Revisiting the Steam-Boiler Case Study with Lutess : Modeling for Automatic Test Generation. In *12th European Workshop on Dependable Computing*, Toulouse, France, Mai 2009.

[35] I. Parissis and J. Vassy. Strategies for Automated Specification-Based Testing of Synchronous Software. In *16th International Conference on Automated Software Engineering*, San Diego, USA, Novembre 2001. IEEE.

[36] Ioannis Parissis. *Test de logiciels synchrones spécifiés en Lustre.* Thèse, Université Joseph Fourier, Grenoble, France, Septembre 1996.

[37] Ioannis Parissis and Farid Ouabdesselam. Specification-based Testing of Synchronous Software. In *ACM-SIGSOFT Foundations of Software Engineering*, pages 127–134, 1996.

[38] Ioannis Parissis and Jérôme Vassy. Thoroughness of Specification-Based Testing of Synchronous Programs. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, pages 191–202, Novembre 2003.

[39] David Parnas, John van Schouwen, and Shu Po Kwan. Evaluation of Safety-Critical Software. *Communications of the ACM*, 33(6) :636–648, Juin 1990.

[40] Matthieu Petit. *Test statistique structurel par résolution de contraintes de choix probabiliste.* Thèse, Université de Rennes I, Rennes, France, Juillet 2008.

[41] Daniel Pilaud and Nicolas Halbwachs. From a Synchronous Declarative Language to a Temporal Logic Dealing with Multiform Time. In *FTRTFT*, pages 99–110, 1988.

[42] 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.

[43] Alexander Pretschner, Er Pretschner, and Heiko Lötzbeyer. Model Based Testing with Constraint Logic Programming : First Results and Challenges. In *2nd ICSE Intl. Workshop on Automated Program Analysis, Testing, and Verification (WAPATV'01)*, 2001.

[44] L. Py, B. Legeard, and B. Tatibouët. Évaluation de spécifications formelles en programmation logique avec contraintes ensemblistes - application à l'animation de spécifications B. In *AFADL*, Grenoble, France, Janvier 2000.

[45] C. Ratel. *Définition et réalisation d'un outil de vérification formelle de programmes Lustre : Le système Lesar*. Thèse, Grenoble, France, Juin 1992.

[46] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone :le générateur de code LUSTRE-V3*. Thèse, Grenoble, France, Septembre 1991.

[47] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic Testing of Reactive Systems. In *IEEE Real-Time Systems Symposium*, pages 200–209, 1998.

[48] J. C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *AAAI*, pages 362–367, 1994.

[49] B. Seljimi and I. Parissis. Test de logiciels synchrones : apports de la programmation par contraintes. In *Approches Formelles dans l'Assistance au Développement de Logiciels*, Namur, Belgique, Juin 2007.

[50] Besnik Seljimi and Ioannis Parissis. Using CLP to Automatically Generate Test Sequences for Synchronous Programs with Numeric Inputs and Outputs. In *17th International Symposium on Software Reliability Engineering*, pages 105–116, Raleigh, USA, Novembre 2006.

[51] Besnik Seljimi and Ioannis Parissis. Automatic Generation of Test Data Generators for Synchronous Programs : Lutess V2. In *Workshop on Domain Specific Approaches to Software Test Automation*, pages 8–12, Dubrovnik, Croatia, Septembre 2007.

[52] K. C. Tai, M. P. Vouk, A. Paradkar, and P. Lu. A predicate-based software testing strategy. *IBM Software Journal*, 1994.

[53] Jan Tretmans. A Formal Approach to Conformance Testing. In *Protocol Test Systems*, pages 257–276, 1993.

[54] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Design, Implementation, and Evaluation of the Constraint Language cc(FD). *Constraint Programming*, 910 :293–316, Mai 1995.

[55] Jérôme Vassy. *Génération automatique de cas de test guidée par les propriétés de sûreté*. Thèse, Université Joseph Fourier, Grenoble, France, Octobre 2004.

[56] Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of Software Testing Criteria using the Z Notation. In *COMPSAC*, pages 351–356, 2001.

[57] Sergiy A. Vilkomir and Jonathan P. Bowen. Reinforced Condition/Decision Coverage (RC/DC) : A New Criterion for Software Testing. In *ZB*, pages 291–308, 2002.

[58] M. P. Vouk, K. C. Tai, and A. Paradkar. Test generation for boolean expressions. In *International Symposium on Software Reliability Engineering*, Toulouse, France.

[59] M. P. Vouk, K. C. Tai, and A. Paradkar. Empirical studies of predicate-based software testing. In *International Symposium on Software Reliability Engineering*, pages 55–64, 1994.

[60] E. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, pages 353–363, Mai 1994.

[61] F. Zhang and T. Cheug. Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines. *Transactions on Software Engineering*, 29(1) :1–14, 2003.

[62] Nicolas Zuanon. *Une méthode de test pour la validation de services de télécommunication*. Thèse, Université Joseph Fourier, Grenoble, France, Juin 2000.

# Annexe A

# Définition des noeuds Lustre

Nous donnons dans cette annexe les principaux noeuds utilisés dans ce manuscrit. Trois types de noeuds sont donnés ici.

- Les opérateurs usuels comme l'implication logique et les opérateurs permettant de travailler avec des tableaux.
- Les opérateurs temporels, utiles notamment pour l'écriture de propriétés de sûreté.
- La réalisation en Lustre du climatiseur avec des entrées/sorties entières.

## A.1 Les opérateurs usuels

L'implication logique ne dispose pas d'opérateur standard en Lustre. Cependant, étant donné qu'il est souvent nécessaire et notamment pour définir des propriétés de sûreté, on peut le réaliser avec le noeud *implies* suivant :

```
node implies(a, b : bool) returns (imp : bool)
let
     imp = not(a) or b ;
tel ;
```

L'opérateur *AND* est utile pour définir la conjonction des éléments d'un tableau de booléens :

```
node AND(const n : int ; a : bool^n) returns (res : bool)
let
     res = with n=1 then a[0] else a[0] and AND(n-1,a[1..n]) ;
tel ;
```

Le noeud *sum* calcule la somme des éléments d'un tableau d'entiers :

```
node sum(const n : int; a : int^n) returns (res : int)
let
     res = with n=1 then a[0] else a[0] + sum(n-1,a[1..n]);
tel;
```

## A.2 Les opérateurs temporels

*after(A)* est vrai s'il y a eu au moins une occurrence de A par le passé.

```
node after(a : bool) returns (aft : bool)
let
     aft = false -> pre(aft or a);
tel;
```

*always_since(A,B)* est vrai si A a été vrai dans tous les instants après le premier instant ou B a été vrai.

```
node always_since(a, b : bool) returns (alw_sinc : bool)
let
     alw_sinc = if b then a else
          if after(b) then pre(alw_sinc) and a else true;
tel;
```

*once_since(A,B)* est vrai si A a été vrai au moins pendant un instant après le premier instant ou B a été vrai.

```
node once_since(a, b : bool) returns (once_sinc : bool)
let
     once_sinc = if b then a else
          if after(b) then a or pre(once_sinc) else false;
tel;
```

*always_from_to(A,B,C)* est vrai si A a été vrai dans tous les instants entre l'instant ou B a été vrai et l'instant ou C a été vrai.

```
node always_from_to(a, b, c : bool) returns (alw_frto : bool)
let
     alw_frto = implies( after(b), always_since(a, b) or
once_since(c,b) );
tel;
```

*once_from_to(A,B,C)* est vrai si A a été vrai au moins pendant un instant entre l'instant ou B a été vrai et l'instant ou C a été vrai.

```
node once_from_to(a, b, c : bool) returns (once_frto : bool)
let
     once_frto = implies( after(b) and c, once_since(a,b) ) ;
tel ;
```

## A.3 Le climatiseur numérique

Nous avons donné dans le chapitre 4 des exemples de données de test générées pour le climatiseur avec des entrées/sorties numériques. Le logiciel utilisé pour la génération de ces données a été réalisé avec le noeud LUSTRE suivant :

```
node ClimNum(Bouton : bool ; Tamb, Tutil : int)
     returns (En_marche : bool ; Tout : int) ;
let
     En_marche = Bouton -> pre(En_marche) and not(Bouton)
          or not(pre(En_marche)) and Bouton ;
     Tout = Tamb + (Tutil - Tamb)/3 ;
tel ;
```

# Annexe B

# Représentation en contraintes de l'exemple du climatiseur

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%% THE HEATER TEST SPECIFICATION %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%% Definition of input and output types of the SUT
inputs(I) :- I = [_Bouton,_Tamb,_Tutil],
   type_bool(_Bouton), type_int(_Tamb), type_int(_Tutil).
outputs(O) :- O = [_En_marche,_Tsort],
   type_bool(_En_marche), type_int(_Tsort).

%%%%%% Definition the state variables and their types
state(S) :- S = [_Q0,_Q1,_Q2,_Q3,_Q4], type_bool(_Q0),
   type_int(_Q1), type_bool(_Q2), type_bool(_Q3), type_bool(_Q4).

%%%%%% Definition the initial state
initial(S) :- state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4], _Q0 #= 1.

%%%%%% Definition of the transition constraints
transition(S,I,O,Sp) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   outputs(O), O = [_En_marche,_Tsort],
   state(Sp), Sp = [_Qp0,_Qp1,_Qp2,_Qp3,_Qp4],
   _Qp0 #= 0,
   _Qp1 #= _Tamb,
   _Qp2 #= _En_marche,
   _Qp3 #= #>(_Tsort,_Tamb),
   _Qp4 #= #<(_Tsort,_Tamb).

%%%%%% Definition of the environment invariants
environment(S,I) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   _dTamb #= if_then_else(_Q0,0,_Tamb - _Q1),
   and(_Tamb #>= -20, _Tamb #=< 60),
```

```
   and(_Tutil #>= 10, _Tutil #=< 40),
   and(_dTamb #>= -1, _dTamb #=< 1).

%%%%%% Definition of the safety property
safeprop(S,I,O,B) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   outputs(O), O = [_En_marche,_Tsort],
   type_bool(B),
   B #= or( neg( and( _En_marche, _Tamb #< _Tutil ) ), _Tsort #> _Tutil ).

%%%%%% Definition of the hypotheses
hypothesis(S,I,O) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   outputs(O), O = [_En_marche,_Tsort],
   if_then_else( _Q0, 1, #=( _Bouton, _En_marche #\= _Tutil ) ).

%%%%%% Definition of the probabilities
probabilities(S,I,[L1,L2]) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3,_Q4],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   L1 = [C1,E1,P1], C1 #= if_then_else(_Q0,0,_Q3),
   if_then_else(_Q0,1,#=( E1, _Tamb #> _Q1 )), P1 is 0.8,
   L2 = [C2,E2,P2], C2 #= if_then_else(_Q0,0,_Q4),
   if_then_else(_Q0,1,#=( E2, _Tamb #< _Q1 )), P2 is 0.8.
```

# Annexe C

# Réalisation des algorithmes en ECLiPSe Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%% LIBRARIES %%%%%%%%%%%%%%%%%%%%%%%%%%
:- lib(ic), lib(ic_global), lib(lists), lib(util).

%%%%%%%%%%%%%%%%%%%%%%%% SEQUENCE GENERATION %%%%%%%%%%%%%%%%%%%%%
%%%%%% Generation of a sequence of length N
sequence(N, Seed) :- seed(Seed), initial(S), path(S,N).

%%%%%% Definition of a path of length N
path(_S,0).
path(S,N) :- N>0,
   environment(S,I),
   post_probabilities(S,I),
   ( pertinent(S,I,Vars); Vars = [] ),
   config(choose_method, Method),
   choose(I, Method),
   enumlist(Vars),!,
   write_inputs(I),
   read_outputs(O),
   transition(S,I,O,Sp),
   enum(Sp),!,
   N1 is N-1,
   path(Sp,N1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%% PROBABILITIES %%%%%%%%%%%%%%%%%%%%%%%%

%%%%%% Posting the probabilities
post_probabilities(Q,E) :-
   probabilities(Q,E,L),
   (foreach(X,L) do
      (
         config(probability_reject, Option),
         X = [C,E,P],
         prob(C,E,P,Option)
      )
```

```
   ).

%%%%%% Post a probability constraint, based on the reject option
prob(C,E,P,none) :- frandom(R), B is (R =< P), Cr#=eval(C), Er#=eval(E),
   neg(Cr) or #=(Er,B).
prob(C,E,P,backtrack) :- frandom(R), B is (R =< P), Cr#=eval(C), Er#=eval(E),
   ( neg(Cr) or #=(Er,B) ; neg(Cr) or #\=(Er,B) ).

%%%%%%%%%%%%%%%%%% SAFETY PROPERTIES & HYPOTHESES %%%%%%%%%%%%

%%%%%% Definition of a pertinent input vector (under hypothesis)
pertinent(S,I,Vars) :-
   config(safeprop_length, K),
   config(safeprop_strategy, Strategy),
   hfeasible(K,S,I,Signature,Vars),
   search_strategy(Signature, K, Strategy).

%%%%%% Definition of a feasible sub-path (under hypothesis)
hfeasible(0,_S,_I,[],[]).
hfeasible(K,S,I,[B|Signature],[I,O,Sp|Vars]) :- K > 0,
   environment(S,I),
   safeprop(S,I,O,B),
   hypothesis(S,I,O),
   transition(S,I,O,Sp),
   K1 is K - 1,
   hfeasible(K1,Sp,_Ip,Signature,Vars).

%%%%%% Definition of different search strategies for pertinent vectors
search_strategy(Signature, Length, union) :-
   M is Length - 1, atmost(M,Signature,1).
search_strategy(Signature, _Length, intersection) :-
   ( foreach(B,Signature) do B#=0 ).
search_strategy(Signature, _Length, lazy) :-
   search(Signature, 0, input_order, indomain_min, complete, []).

%%%%%%%%%%%%%%%%%% CHOOSE : TEST DATA SELECTION METHOD %%%%%%%%

% Choose a random value in the domain of variables as they appear in the list
choose(L, random) :-
   search(L, 0, input_order, indomain_random, complete, []).

% Choose a random value starting from the variable with the smallest domain
choose(L, random_smallest_domain) :-
   search(L, 0, most_constrained, indomain_random, complete, []).

% Choose randomly one of the lower or upper bound of the domain, first-fail order
choose(L, random_bound) :-
   search(L, 0, first_fail, min_max_random, complete, []).
min_max_random(X) :- (frandom(R), (R =< 0.5) -> get_min(X,X) ; get_max(X,X) ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%% HELPFUL PREDICATES USED %%%%%%%%%%%

%%%%%% Enumeration of lists of variables
enumlist(L) :- ( foreach(X, L) do enum(X) ).
enum(L) :- search(L, 0, first_fail, indomain_min, complete, []).

%%%%%% Comunication with Lutess
write_inputs(I) :- ( foreach(X,I) do writeln(output,X) ), flush(output).
read_int(X) :- read_line(input,S), number_string(X,S).
read_outputs(O) :- outputs(O), ( foreach(X,O) do read_int(X) ).

%%%%%%%%%%%%%%%%%%%%%%%%% LUSTRE TYPES AND OPERATORS %%%%%%%%%

%%%%%% Define the constraints for variable types
type_uint(X) :- X # : : 0 .. 4294967296.
type_int(X) :- X # : : -2147483648 .. 2147483647.
type_bool(X) :- X # : : 0 .. 1.

%%%%%% Functional if-then-else operator in lustre
if_then_else(Cond, Exp1, Exp2) :- if_then_else(Cond, Exp1, Exp2, 1).
if_then_else(Cond, Exp1, Exp2, Result) :-
   C#=eval(Cond), E1 #= eval(Exp1), E2#=eval(Exp2), R#=eval(Result),
   and(C#=1,R #= E1) or and(C#=0,R #= E2).

%%%%%% The sharp operator
sharp(L) :- sharp(L,1).
sharp(L,1) :- boolean(L), atmost(1,L,1).
sharp(L,0) :- boolean(L), length(L,N), M is N-2, atmost(M,L,0).

boolean(List) :- ( foreach(B,List) do type_bool(B) ).

%%%%%% Natural division and modulo
ndiv(X,Y,R) :- Xs #= eval(X), Ys #= eval(Y), Rs #= eval(R),
   Rs * Ys #= Z, Z #=< Xs, Z + Ys #> Xs.
nmod(X,Y,R) :- Xs #= eval(X), Ys #= eval(Y), Rs #= eval(R),
   Z #>= 0, Rs #>= 0, Rs #< Ys, Z * Ys + Rs #= Xs.
```

# Annexe D

# Exemple du climatiseur avec variables réelles

## D.1 Machine à états finis

```
-- Machine à états finis
inputs
   Bouton : bool ; Tamb : real ; Tutil : int ;
outputs
   En_marche : bool ; Tsort : real ;
state
   q0 : bool ; q1 : real ; q2,q3 : bool ;
initial
   q0 = true ;
var
   dTamb : real ;
local
   dTamb = if q0 then 0.0 else Tamb - q1 ;
transition
   q0' = false ;
   q1' = Tamb ;
   q2' = Tsort>Tamb ;
   q3' = Tsort<Tamb ;
environment
   Tamb>=-20.0 and Tamb<=60.0
   and
   Tutil>=10 and Tutil<=40
   and
   dTamb>=-1.5 and dTamb<=1.5 ;
probabilities
   (c1, e1, p1) = (q2, Tamb>q1, 0.8) ;
   (c2, e2, p2) = (q3, Tamb<q1, 0.8) ;
```

## D.2 Représentation en contraintes

```
%%%%%% Definition of input and output types of the SUT
inputs(I) :- I = [_Bouton,_Tamb,_Tutil],
   type_bool(_Bouton), type_real(_Tamb), type_int(_Tutil).
outputs(O) :- O = [_En_marche,_Tsort],
   type_bool(_En_marche), type_real(_Tsort).

%%%%%% Definition the state variables and their types
state(S) :- S = [_Q0,_Q1,_Q2,_Q3],
   type_bool(_Q0), type_real(_Q1), type_bool(_Q2), type_bool(_Q3).

%%%%%% Definition the initial state
initial(S) :- state(S), S = [_Q0,_Q1,_Q2,_Q3], _Q0 #= 1.

%%%%%% Definition of the transition constraints
transition(S,I,O,Sp) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   outputs(O), O = [_En_marche,_Tsort],
   state(Sp), Sp = [_Qp0,_Qp1,_Qp2,_Qp3],
   _Qp0 #= 0,
   _Qp1 $= _Tamb,
   _Qp2 #= $>(_Tsort,_Tamb),
   _Qp3 #= $<(_Tsort,_Tamb).

%%%%%% Definition of the environment invariants
environment(S,I) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   _dTamb $= if_then_else_real(_Q0, 0.0, _Tamb - _Q1),
   and(_Tamb $>= -20.0, _Tamb $=< 60.0),
   and(_Tutil #>= 10, _Tutil #=< 40),
   and(_dTamb $>= -1.5, _dTamb $=< 1.5).

%%%%%% Definition of the safety property
safeprop(S,I,O,B) :- type_bool(B).

%%%%%% Definition of the hypotheses
hypothesis(S,I,O).

%%%%%% Definition of the probabilities
probabilities(S,I,[L1,L2]) :-
   state(S), S = [_Q0,_Q1,_Q2,_Q3],
   inputs(I), I = [_Bouton,_Tamb,_Tutil],
   L1 = [C1,E1,P1], C1 #= _Q2, #=( E1, _Tamb $> _Q1 ), P1 is 0.8,
   L2 = [C2,E2,P2], C2 #= _Q3, #=( E2, _Tamb $< _Q1 ), P2 is 0.8.
```

## D.3 Affectation des variables réelles à bornes fixés

```
choose_real(L, random) :-
   (foreach(X,L) do
      (
         frandom(R), get_min(X,Min), get_max(X,Max), X is Min + R*(Max-Min)
      )

   ).
```