Génération automatique de test pour les contrôleurs logiques programmables synchrones

Mouna Tka

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Mouna Tka

Thèse dirigée par **Ioannis Parissis** et
co-encadrée par Christophe Deleuze

préparée au sein du **Laboratoire de Conception et d'Intégration des Systèmes**
dans l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique: ED MSTII**

Génération automatique de test pour les contrôleurs logiques programmables synchrones

Thèse soutenue publiquement le **2 juin 2016**, devant le jury composé de :

Mme Lydie du Bousquet
Professeur, Univ. Grenoble-Alpes, Président
Mr Yves Le Traon
Professeur, Université du Luxembourg, Rapporteur
Mme Virginie Wiels
Chercheur (HDR), ONERA Toulouse, Rapporteur
M. Serge Zaninotti
Ingénieur, Thales Valence, Examinateur
Mr Vincent List
Innovation and Adv. R&D Manager, InnoVista Sensors Valence, Invité
Mr Ioannis Parissis
Professeur, Grenoble INP, Directeur de thèse
Mr Christophe Deleuze
Maître des Conférences, Grenoble INP, Co-encadrant de thèse

Dédicaces

À mon fils Haroun, soleil de ma vie, source de tendresse et d'amour,

À mon mari Bader pour son aide, son soutien, son amour et sa compréhension,

À mes parents Salah et Fatma pour leurs sacrifices durant mes années d'étude et leurs encouragements continuels,

À toute ma famille.

Remerciements

Cette thèse s'est déroulée au sein du Laboratoire de Conception et d'Intégration des Systèmes. Je tiens á exprimer ma reconnaissance á son directeur Michel Occello, pour m'avoir accueillie et m'avoir fourni des conditions de travail exceptionnelles.

Je tiens á exprimer mes plus sincères remerciements á :

Yves Le Traon, Professeur á l'Université du Luxembourg et Virginie Wiels, Chercheur (HDR) à ONERA Toulouse, pour avoir accepté de juger ce travail en tant que rapporteurs;

Lydie du Bousquet, Professeur à l'Université Grenoble-Alpes, Vincent List, Innovation and Adv. R&D Manager à InnoVista Sensors Valence et Serge Zaninotti, Ingénieur à Thales Valence, pour m'avoir fait l'honneur de participer au jury de cette thèse;

Ioannis Parissis, Professeur à Grenoble INP et Christophe Deleuze, Maître des Conférences à Grenoble INP, pour m'avoir dirigée, conseillée et guidée ainsi que pour avoir fait preuve de tant de confiance, de patience, d'amitié et de bienveillance durant ces années de thèse.

Tous les membres du projet Bluesky pour les échanges fructueux lors des réunions de travail:

L'équipe de InnoVista Sensors à savoir : Jean Baptiste Gning, Guillaume Marie et Jackie Launay pour leur collaboration, les matériaux et les exemples d'applications em4 qu'ils nous ont fournis ;

Les équipes de LCIS et d'INRIA Grenoble.

Tous mes collègues et amis du laboratoire et de l'ESISAR qui se reconnaitront ici.

Enfin, un grand merci à mes parents, dont les encouragements m'ont aidée malgré la distance qui nous sépare, à mon mari Bader pour m'avoir toujours soutenue et à mon fils Haroun, qui égaye ma vie et me fait oublier la dure épreuve du travail par son sourire rayonnant.

Résumé

Ce travail de thèse, effectué dans la cadre du projet FUI Minalogic Bluesky, porte sur le test fonctionnel automatisé d'une classe particulière de contrôleurs logiques programmables (em4) produite par InnoVista Sensors. Ce sont des systèmes synchrones qui sont programmés au moyen d'un environnement de développement intégré (IDE). Les personnes qui utilisent et programment ces contrôleurs ne sont pas nécessairement des programmeurs experts. Le développement des applications logicielles doit être par conséquent simple et intuitif. Cela devrait également être le cas pour les tests. Même si les applications définies par ces utilisateurs ne sont pas nécessairement très critiques, il est important de les tester d'une manière adéquate et efficace. Un simulateur inclus dans l'IDE permet aux programmeurs de tester leurs programmes d'une façon qui reste à ce jour informelle et interactive en entrant manuellement des données de test. En se basant sur des recherches précédentes dans le domaine du test des programmes synchrones, nous proposons un nouveau langage de spécification de test, appelé SPTL (Synchronous Programs Testing Language) qui rend possible d'exprimer simplement des scénarios de test qui peuvent être exécutés à la volée pour générer automatiquement des séquences d'entrée de test. Il permet aussi de décrire l'environnement où évolue le système pour mettre des conditions sur les entrées afin d'arriver à des données de test réalistes et de limiter celles qui sont inutiles. SPTL facilite cette tâche de test en introduisant des notions comme les profils d'utilisation, les groupes et les catégories. Nous avons conçu et développé un prototype, nommé "Testium", qui traduit un programme SPTL en un ensemble de contraintes exploitées par un solveur Prolog qui choisit aléatoirement les entrées

de test. La génération de données de test s'appuie ainsi sur des techniques de programmation logique par contraintes. Pour l'évaluer, nous avons expérimenté cette méthode sur des exemples d'applications EM4 typiques et réels. Bien que SPTL ait été évalué sur em4, son utilisation peut être envisagée pour la validation d'autres types de contrôleurs ou systèmes synchrones.

Abstract

This thesis work done in the context of the FUI project Minalogic Bluesky, concerns the automated functional testing of a particular class of programmable logic controllers (em4) produced by InnoVista Sensors. These are synchronous systems that are programmed by means of an integrated development environment (IDE). People who use and program these controllers are not necessarily expert programmers. The development of software applications should be as result simple and intuitive. This should also be the case for testing. Although applications defined by these users need not be very critical, it is important to test them adequately and effectively. A simulator included in the IDE allows programmers to test their programs in a way that remains informal and interactive by manually entering test data. Based on previous research in the area of synchronous test programs, we propose a new test specification language, called SPTL (Synchronous Testing Programs Language) which makes possible to simply express test scenarios that can be executed on the fly to automatically generate test input sequences. It also allows describing the environment in which the system evolves to put conditions on inputs to arrive to realistic test data and limit unnecessary ones. SPTL facilitates this testing task by introducing concepts such as user profiles, groups and categories. We have designed and developed a prototype named "Testium", which translates a SPTL program to a set of constraints used by a Prolog solver that randomly selects the test inputs. So, generating test data is based on constraint logic programming techniques. To assess this, we experimented this method on realistic and typical examples of EM4

applications. Although SPTL was evaluated on EM4, its use can be envisaged for the validation of other types of synchronous controllers or systems.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

Les contrôleurs logiques programmables (CLP) sont de plus en plus utilisés dans l'industrie et dans la vie quotidienne pour l'automatisation de différents processus. Cette thèse porte sur le test fonctionnel automatisé d'une classe spécifique de ces contrôleurs synchrones : les contrôleurs produits par l'entreprise "InnoVista Sensors" leader du projet FUI "BlueSky" impliquant plusieurs entreprises et laboratoires de la région de Grenoble et Valence. Dans ce chapitre introductif, nous commencerons par présenter brièvement le contexte général de nos travaux de recherche, à savoir le projet "BlueSky" et le test des systèmes synchrones, pour continuer ensuite avec la description des motivations et des contributions de cette thèse.

1.1 Contexte des recherches

1.1.1 Test des systèmes synchrones

Les logiciels sont omniprésents dans notre vie quotidienne. Nous y sommes confrontés à chaque utilisation d'un système informatisé. Ils sont de plus en plus complexes et parfois critiques. Comme tout objet complexe, un logiciel doit être testé avant d'être mis en service. Les tests font partie des procédés de la Vérification et la Validation (V&V) d'un logiciel. D'après les définitions de la norme ISO9000:2015, le processus de vérification et de validation permet de s'assurer qu'une conception respecte bien les spécifications. La vérification confirme par des preuves objectives que les exigences spécifiées ont été satisfaites. La validation confirme par des preuves objectives que les exigences pour une utilisation spécifique ou une application prévues ont été satisfaites [GD15]. Généralement, le test est défini par l'exécution du système sous test avec l'intention d'y trouver des erreurs [Mat08]. Le but est de réparer ces dernières pour améliorer la qualité et la sûreté du logiciel testé.

Un système réactif est un système répondant constamment aux sollicitations de son environnement en produisant des actions sur celui-ci. Les systèmes réactifs synchrones ont la particularité d'avoir la propriété fondamentale suivante : la réaction du logiciel doit être assez rapide (théoriquement instantanée) de sorte que n'importe quel changement de ce dernier puisse être pris en compte. L'approche synchrone [BB02] a connu un grand développement depuis une vingtaine d'années, en particulier grâce à l'apparition de langages qui lui sont propres, tels que Lustre [HCRP91], Esterel [BdS91] et Signal [PLGM91]. Pour ces systèmes synchrones qui assez souvent font partie d'applications critiques (aéronautique, réseau ferroviaire ...) la phase de test est primordiale pour éviter des défaillances graves et parfois catastrophiques. Cependant, les tests sont coûteux autant en termes de temps que d'argent. L'automatisation du processus de test peut être une solution à ce problème. En effet, en automatisant ce processus, on peut exécuter des milliers de

cas de test complexes et divers pendant chaque exécution. Ces tests fournissent ainsi une couverture impossible à obtenir avec des tests manuels. Les méthodes de test automatisé peuvent ainsi permettre d'économiser beaucoup d'effort humain. Cela vaut particulièrement pour les systèmes réactifs synchrones qui nécessitent d'être testés de manière approfondie. Le choix de données de test pertinentes est aussi important pour augmenter la capacité à révéler des fautes. Les méthodes de test ont évolué au fil du temps pour rendre ce choix plus pertinent en fonction des types de fautes recherchées et des particularités des logiciels testés

De nombreux travaux de recherche ont été menés sur les méthodes de génération automatique de test des logiciels. On distingue deux grandes catégories de techniques de test. Les techniques dites en "boîte noire", exécutent le logiciel avec des données de test sélectionnées au hasard ou bien construits à partir d'une spécification du logiciel. Ces techniques ne prennent pas en compte la manière dont le logiciel a été implanté (langage utilisé, code produit etc.). La deuxième catégorie concerne les techniques dites en "boîte blanche" qui s'appuient sur une analyse du code réalisant le logiciel. Elles considèrent des séquences de test dont l'exécution assure l'activation de ce code de différentes manières. Parmi les outils de génération de test pour les systèmes synchrones connus, nous nous sommes particulièrement intéressés à Lutess [SP06, PSP09], Lurette [JRB06] (test fonctionnel ou boite noire) et Gatel [MB05](test en boite blanche). Dans les travaux de cette thèse, nous avons eu un intérêt particulier pour le test à base de contraintes introduit en 1991 [DO91]. Cette méthode vise à générer automatiquement des données de tests en utilisant une modélisation en un problème de satisfaction de contraintes, ou CSP (Constraint Satisfaction Problem). Nous reviendrons à toutes ces méthodes plus en détails dans les chapitres 2, 3 et 4 pour étudier leurs points forts et leurs points faibles ainsi que leur applicabilité dans notre contexte industriel à savoir le projet BlueSky.

1.1.2 Projet BlueSky

Comme un grand nombre de technologies modernes, le premier Contrôleur Logique Programmable (CLP) est né d'une exigence de l'industrie automobile et plus précisément d'une proposition formulée en 1968 par la division des transmissions automatiques de General Motors. Au cours de ces dernières années, diverses améliorations clés des CLP se sont succédées, souvent générées par de nouvelles demandes des marchés. Nous sommes passés d'automatismes simples surveillant une grandeur physique et capable d'activer un relais, à l'intégration de fonctionnalités de plus en plus complexes. Ainsi sont apparus les automates programmables qui ont bénéficié de nombreuses avancées technologiques (microprocesseurs et autres ...) afin de les rendre de plus en plus performants et de moins en moins couteux. Pour répondre aux besoins de petits installateurs d'équipements de toutes sortes, voire de particuliers qui ne possèdent pas nécessairement les compétences requises pour la mise en œuvre de tels outils, les Contrôleurs Logiques Programmables comme nous les connaissons aujourd'hui sont apparus. Il n'est plus nécessaire d'être un automaticien pour utiliser ces CLP. Em4, le Contrôleur Logique Programmable de "InnoVista Sensors" fait partie de ces matériels qui allient simplicité de mise en œuvre et performance. Les systèmes de communication ont connu aussi un grand développement technologique : de la transmission analogique à la transmission numérique puis à l'intégration de services (voix, données, images ...). La fibre optique et les technologies IP ont par la suite permis le déploiement à grande échelle du haut débit et des services Internet. En parallèle, nous avons vécu la grande révolution des systèmes cellulaires numériques avec l'apparition des réseaux 3 ème et 4 ème génération. La convergence de ces deux grandes évolutions a donné naissance à la notion M2M(MACHINE TO MACHINE) : technologie qui permet de rendre les machines communicantes avec des serveurs sur Internet et cela de façon automatique, sans intervention humaine. Ainsi le M2M avec les protocoles internet ont fait prospérer ce que nous appelons par

l'Internet des objets qui représente l'extension d'Internet à des choses (machines) du monde physique.

1.1.2.1 Objectifs

Avec la nouvelle génération des automates connectés EM4, InnoVista Sensors vise à faire converger le monde de l'automatisme avec celui de l'Internet des objets. Ces automates peuvent faire directement appel aux ressources d'Internet : des données et informations du Web pourront être utilisées comme éléments de capteurs virtuels. Cependant, la complexité de ces technologies rajoute des contraintes de sûreté de fonctionnement. Les techniques actuellement disponibles ne permettent pas une validation effective de ce type de système.
Les défis du projet sont de concevoir un système satisfaisant aux exigences suivantes :

- Garantir et maintenir un haut niveau de déterminisme des systèmes automatiques.
- Proposer une mise en œuvre simplifiée des nouvelles solutions de communication.
- Garantir la sécurisation des flux d'informations transmis.
- Proposer des méthodologies et outils de validation de systèmes d'automatismes distribués et non-distribués.

1.1.2.2 Partenaires

Le Projet BlueSky fait participer plusieurs partenaires industriels et académiques. Comme le montre la figure 1.1, le projet est divisé en sous-programmes. Chacun des partenaires est responsable d'au moins un sous-programme. Les partenaires dans ce projet sont : InnoVista Sensors, Vertical M2M, MOOTWIN et les laboratoires INRIA et LCIS. Le sous-projet SP2, qui concerne les laboratoires INRIA et LCIS, vise le dernier objectif cité dans la section 1.1.2.1, à savoir la proposition de services de validation de programmes applicatifs métiers.

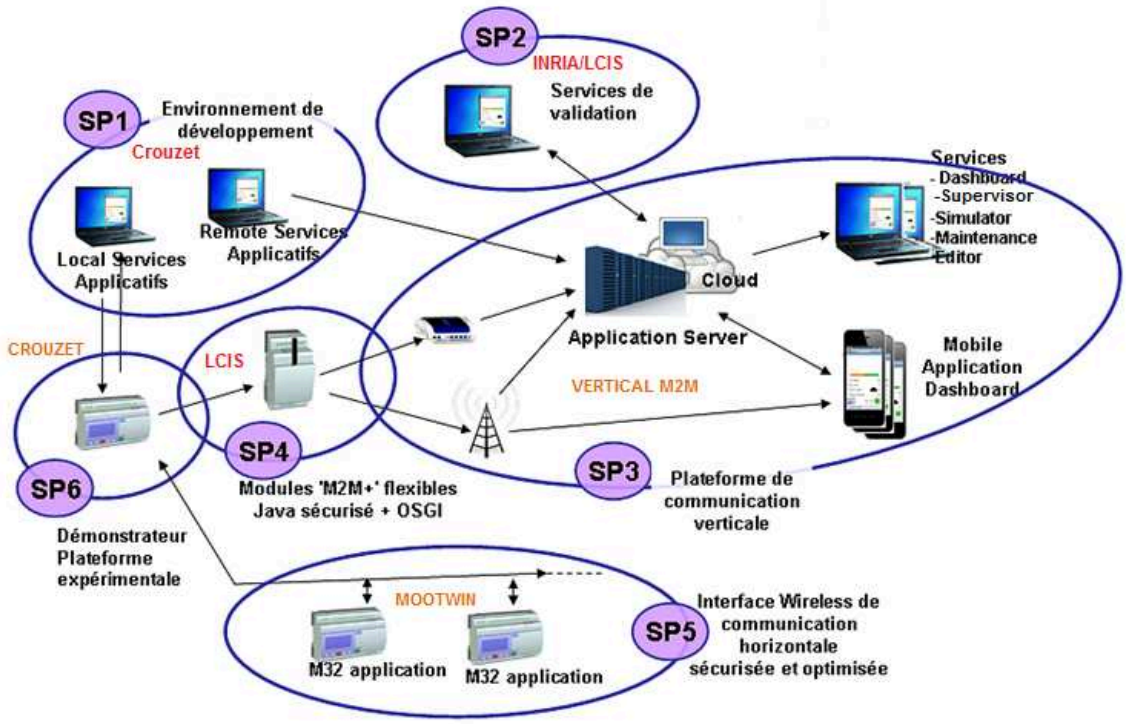


Figure 1.1 Représentation graphique des sous-programmes

Source: Annexe technique projet BLUESKY

1.2 Problèmes et motivations

Les utilisateurs et les programmeurs des contrôleurs logiques programmables ne sont pas forcément des experts. C'est pour cela que InnoVista Sensors fournit un environnement de développement intégré (IDE) qui permet de concevoir des applications d'une qualité professionnelle sur em4 simplement au moyen de composants prédéfinis. Bien que les applications développées sur ces PLC ne soient pas nécessairement critiques, leur test reste important et doit être efficace et adéquat. Actuellement, un simulateur intégré à l'atelier de développement "em4 Soft" permet ce test d'une

manière informelle en entrant manuellement les données de test et en voyant directement les sorties correspondantes. Ce moyen de test ne valide pas ces applications d'une manière rigoureuse et ne permet pas non plus d'écrire des scénarios de test.
La problématique de ce travail s'inscrit donc autour du mécanisme du test de ces CLP. Elle consiste à trouver une méthode qui permettra de tester automatiquement des applications programmées sur les contrôleurs logiques programmables synchrones de façon intuitive. Cette méthode devra permettre aussi à l'utilisateur de décrire l'environnement où évolue le système afin d'avoir des jeux de test réalistes et d'écrire des scénarios de test. Nous abordons dans ce travail de thèse cette problématique, en se basant sur des recherches précédentes dans le domaine de test des systèmes synchrones réactifs et en s'appuyant sur la Programmation Logique avec Contraintes (PLC) [JL87], souvent suggérée à des fins de génération de tests en particulier dans le cas des systèmes réactifs [PL01].

1.3 Contributions

L'objectif principal de cette thèse est d'aboutir à une approche de test fonctionnel automatisé pour les contrôleurs logiques programmables synchrones et de mettre ainsi à disposition des utilisateurs non spécialistes des technologies performantes de test réservées jusqu'à présent à des experts pour la validation de leurs systèmes.
Nous proposons une méthode permettant la génération automatique de données de test, à partir d'un modèle. Ce modèle est écrit dans un langage de spécification de test : Synchronous Programs Testing Language (SPTL). Avec ce langage, nous visons la simplicité d'utilisation. Il propose des opérateurs proches du langage de programmation du contrôleur à tester. Il permet de cerner les intervalles possibles des données de test afin de réduire les valeurs inutiles et ceci en décrivant l'environnement d'utilisation du système au moyen de contraintes tout en étant guidé par le profil d'utilisation. Nous détaillerons toutes ces notions dans les chapitres qui suivent. L'utilisateur peut aussi rédiger des scénarios toujours sous formes de

contraintes. Les algorithmes de génération se basent sur les principes de résolution des problèmes à contraintes pour affecter des valeurs aux entrées du SST (Système Sous Test).
À travers ce travail, nous proposons :

- La définition du langage de description de test SPTL (sa syntaxe et sa sémantique). Le but de ce langage est de répondre aux exigences de simplicité et de donner aux utilisateurs la possibilité d'exprimer les contraintes de l'environnement du système et d'exprimer des scénarios de test d'une manière intuitive.

- La conception et la réalisation d'un prototype d'outil de génération de test "Testium" basé sur le langage SPTL et la mise en œuvre de la programmation logique par contraintes pour la génération des données de test. Ce prototype a été évalué par des études de cas réels et typiques.

1.4 Structure du document

Ce manuscrit est composé de 8 chapitres organisés comme suit :
Après ce premier chapitre introductif, le chapitre 2 est dédié aux contrôleurs logiques programmables synchrones, leurs caractéristiques et leurs langages de programmation et plus généralement les langages de programmation des systèmes réactifs synchrones.
Par la suite, le chapitre 3 introduit les notions fondamentales concernant le test logiciel en général et le test des systèmes synchrones en particulier. Les outils existants de génération automatique de test pour les systèmes synchrones sont exposés dans ce chapitre.
Le chapitre 4 détaille la problématique, présente les lacunes des outils existants et expose l'approche globale de l'outil de test "Testium".
Dans le chapitre 5, nous présentons les notions de base du langage SPTL, sa syntaxe et sa sémantique.
Au chapitre 6, nous décrivons le composant Prolog de l'outil "Testium" et la manière

dont il traduit un programme SPTL en un programme Prolog.

Le chapitre 7 est consacré quant à lui à l'évaluation du prototype à travers une expérimentation sur un exemple réel et typique d'une application programmée sur em4.

Enfin, le dernier chapitre conclut ce travail de thèse et présente ses perspectives.

Chapitre 2

Les contrôleurs logiques programmables synchrones

2.1 Introduction du chapitre

Nous présentons pour commencer les contrôleurs logiques programmables (CLP) qui sont au cœur de notre étude. Ce chapitre introduit également un exemple d'application programmée sur le contrôleur qui sera repris par la suite dans les autres chapitres pour décrire notre contribution. Nous nous intéressons également aux langages de programmation synchrone et plus en détails à Lustre et à Esterel.

2.2 Les contrôleurs Logiques programmables

Les automatismes industriels et domestiques sont fréquemment mis en œuvre au moyen de contrôleurs logiques programmables (CLP), qui exécutent en mode synchrone des applications embarquées interagissant avec leur environnement. Histori-

quement, les automates programmables industriels sont apparus à la fin des années soixante, à la demande de l'industrie automobile américaine (GM), qui réclamait plus d'adaptabilité des systèmes de commande. Des relais électromagnétiques et des systèmes pneumatiques étaient utilisés pour la réalisation des parties commandes (logique câblée). Cette méthode était chère et n'était pas flexible. La solution était donc d'utiliser plutôt des systèmes à base de microprocesseurs permettant une modification aisée des systèmes automatisés (logique programmée). Trois caractéristiques fondamentales distinguent un contrôleur logique programmable des autres outils informatiques :

- il peut être directement connecté aux capteurs et pré-actionneurs grâce à ses entrées/sorties industrielles,
- il est conçu pour fonctionner dans des ambiances industrielles sévères (température, vibrations, micro-coupures de la tension d'alimentation, parasites, etc.),
- enfin, sa programmation à partir de langages spécialement développés pour le traitement de fonctions d'automatisme fait en sorte que sa mise en œuvre et son exploitation nécessitent en général peu de connaissances en informatique.

Ces contrôleurs logiques programmables font partie de la famille des systèmes réactifs synchrones. Il est alors nécessaire de voir de plus près les caractéristiques qui rassemblent ces systèmes.

La notion de systèmes réactifs a été introduite par David Harel et Amir Pnueli [HP85]. Le terme désigne couramment maintenant les systèmes en interaction permanente avec leurs environnements, à l'opposé des systèmes transformationnels qui eux produisent des sorties en fonction d'entrées selon un processus de calcul indépendant de leurs environnements.

Les systèmes synchrones [BB02] représentent une classe de systèmes réactifs qui sont basés sur l'hypothèse de synchronisme : ils sont censés être assez rapides pour prendre en compte tout événement externe. Autrement dit, l'hypothèse synchrone

suppose qu'une réaction ne prend pas de temps ; les entrées et les sorties du système sont quasiment simultanées.

Dans le paradigme synchrone [ABS03], l'exécution d'un programme repose sur le calcul d'instants successifs. Nous parlons alors de système à événements discrets où chaque instant est défini par un calcul des sorties et du prochain état du système à partir des entrées et de l'état courant du système. Son comportement peut être représenté par une séquence de réactions à des stimuli comme l'illustre la figure 2.1 : à chaque impulsion d'horloge, il réagit instantanément à toute action extérieure. Par exemple à l'instant t_0, il reçoit une entrée i_0, il répond par la sortie o_0, puis répond par o_1 suite à l'entrée i_1, ainsi de suite.
Il faut noter aussi que la même suite d'entrées produit la même suite de sorties (déterminisme).

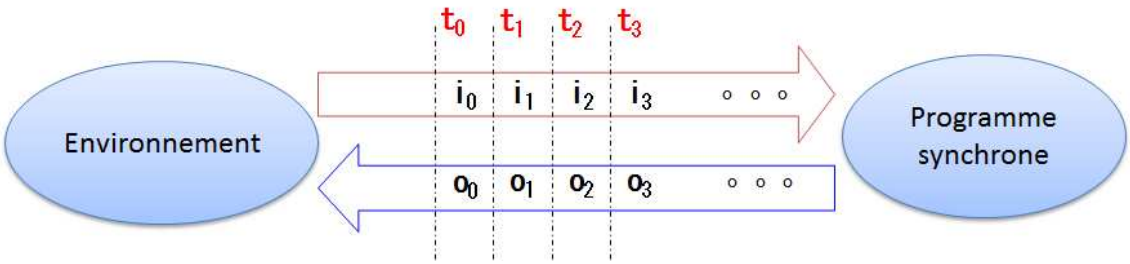


Figure 2.1 Système réactif synchrone

Examinons maintenant les langages qui permettent de programmer les CLP afin de connaitre au mieux les langages auxquels sont habitués leurs programmeurs qui sont leurs potentiels testeurs.
Il existe quatre langages de programmation des automates normalisés au niveau mondial (norme CEI 61131-3) [HUB] :

- Langage LD : le langage Ladder est un langage graphique [Bou04]. Il peut être utilisé pour écrire des diagrammes de relais. Il est adapté pour les programmes combinatoires. Ladder est le mot anglais pour "échelle".

- Langage FBD : le logigramme FBD (Function Block Diagram) permet une programmation graphique basée sur l'utilisation de blocs fonctionnels prédéfinis [WSL15].

- Langage IL : le langage IL (Instruction List) [Huu05] est un langage textuel de bas niveau semblable à de l'assembleur. Il est particulièrement adapté aux applications de petite taille. Les instructions opèrent toujours sur un résultat courant (ou registre IL). L'opérateur indique le type d'opération à effectuer entre le résultat courant et l'opérande. Le résultat de l'opération est stocké à son tour dans le résultat courant.

- Langage ST : le langage ST (Structured Text) [Bol15] est un langage de haut niveau et sa structure rappelle les langages Ada et Pascal. Ce langage est principalement utilisé pour décrire les procédures complexes d'automatisation. C'est le langage par défaut pour la programmation des actions dans les étapes et des conditions associées aux transitions du langage SFC.

Hors normalisation internationale le langage SFC (Sequential Function Chart) [Bol15], ou GRAFCET, est un langage graphique de description de systèmes séquentiels qui est utilisé par certains constructeurs d'automates européens. Le procédé est représenté comme une suite connue d'étapes, reliées entre elles par des transitions, une condition booléenne est attachée à chaque transition. Les actions dans les étapes sont décrites avec les langages ST, IL, LD ou FBD.

Dans le cadre de nos recherches pour l'automatisation du test fonctionnel des contrôleurs logiques programmables synchrones, nous nous intéressons particulièrement aux automates "em4"[1] produits par InnoVista Sensors [2].
La facilité de programmation et le prix relativement peu élevé de ces automates ne restreignent pas leur utilisation aux professionnels. Au début de ce travail de thèse,

[1] `http://www.em4-remote-plc.com`
[2] `http://www.innovistasensors.com`

l'em4 n'était pas encore disponible sur le marché ; seule la version précédente qui est le "Millénium 3" était disponible. Le "Millénium 3" commande un ensemble d'actionneurs en fonction de l'état des capteurs, de l'évolution du temps et du programme élaboré à l'aide du logiciel "M3 Soft". Le logiciel "M3 Soft" est téléchargeable gratuitement sur internet. Il permet de concevoir simplement le programme d'une installation, de le tester manuellement et d'une façon ponctuelle grâce à une simulation et de dialoguer avec l'application à travers le mode monitoring.
En effet cet atelier de programmation possède 3 modes de fonctionnement :

- Mode d'édition : le mode d'édition est utilisé pour écrire des programmes en FBD.
- Mode de simulation : en mode simulation, le programme est exécuté hors ligne directement dans l'atelier de programmation (simulé sur le PC). Chaque action sur le graphe du programme (changement d'état d'une entrée, forçage d'une sortie) met à jour les fenêtres de simulation.
- Mode monitoring : en mode monitoring le programme est exécuté sur le contrôleur et l'atelier de programmation est connecté au contrôleur. Dans ce mode, nous pouvons voir le fonctionnement de la machine en temps réel sur le PC. Les différentes fenêtres sont mises à jour cycliquement.

`em4` est une évolution de `Millénium 3`. Le nouveau CLP et son logiciel de programmation "em4 soft" possèdent toutes les fonctionnalités de la gamme `Millénium` améliorées en y ajoutant la possibilité d'accès au monde de "l'Internet des objets". L'utilisateur peut gérer à distance ses installations et les fonctions distantes sont gérées directement comme de simples blocs de fonctions. Les données sont maintenant accessibles sur un smart-phone ou un PC.
Le logiciel "em4 soft" garde donc le même principe que l'atelier de "Millénium 3". Le langage de programmation reste aussi le même : FBD. Regardons de plus près ce langage : le FBD ressemble étroitement au câblage électrique d'éléments. Il permet un câblage de blocs fonctionnels qui effectuent, par exemple, des calculs

ou permettent l'accès à des variables. Le câblage réalise l'échange de données entre ces éléments. Les blocs fonctionnels ont des entrées et des sorties appelées ports. Chaque bloc est associé à certaines fonctionnalités et les données sont échangées via les ports et leurs connexions [BB13].
Le langage FBD dans l'atelier "em4 soft" propose plusieurs fonctions sous forme de blocs dans sa barre à outils. Nous pouvons citer quelques éléments de ce langage :

- Les éléments d'entrée : les entrées de type TOR (Tout Ou Rien), les entrées réseaux et les entrées de type analogique (vu de l'automate ce sont des entiers sur 16 bits).
- Les éléments de sortie : les sorties de type TOR (accessible à partir de la fenêtre IN/OUT) et PWM (modulation de largeur d'impulsion : Ceci permet de commander une grandeur entre 0 et 100% de sa valeur maximale.)
- Les fonctions de contrôle : Comparaison, programmateur horaire, chronomètre...
- Les fonctions de calcul : opérations arithmétiques, conversions décimale et binaire...
- Les fonctions logiques : les opérateurs logiques classiques NOT, AND, OR, NAND, NOR, XOR.
- Les fonctions PROG : elles regroupent les fonctions qui ne rentrent pas dans les autres catégories comme : la mémorisation (mémorise une valeur ou stocke plusieurs valeurs), la conversion heures/minutes, la fonction RANDOM (fournit une valeur aléatoire) ...

Pour mieux voir la simplicité de l'utilisation de ces blocs fonctionnels nous allons présenter un exemple d'une application typique.

Exemple : Éclairage intérieur/extérieur d'une habitation

L'em4 propose des systèmes de commande adaptés à plusieurs domaines d'application. L'exemple que nous présentons ici est une application d'une installation capable

de gérer seule l'éclairage d'une cage d'escalier et d'une entrée extérieure accédant à l'habitation.

- Éclairage extérieur : le circuit est rendu actif tous les ans du 1er juin au 1er octobre, et la nuit grâce à un interrupteur crépusculaire (l'entrée I_5 dans la figure 2.2). Un capteur (l'entrée I_1 dans la figure 2.2) détecte tout passage et active l'éclairage extérieur durant 2 minutes.

- Éclairage intérieur : deux boutons poussoirs sont disposés dans la cage d'escalier ; l'un dans le hall d'entrée, l'autre en haut de l'escalier. Leurs fonctions sont identiques.

  L'utilisateur peut contrôler la luminosité de l'éclairage grâce à un potentiomètre : c'est l'entrée I_5 dans la figure 2.2. Il y a deux modes d'éclairage :

- L'éclairage temporisé (2 minutes) est provoqué par une brève pression sur un des boutons (l'entrée I_3 ou I_4 de la figure 2.2). La minuterie peut être inhibée par une nouvelle action sur l'un d'eux.

- L'éclairage permanent est activé si un bouton est maintenu pressé durant au moins 2 secondes. Il est stoppé par une brève pression.

Comme le montre la figure 2.2, les sorties de ce programme sont connectées à un relais qui servira d'intermédiaire entre la partie commande et les lampes d'éclairage. Ce composant permettra d'alimenter la lampe si l'éclairage est activé avec la luminosité désirée et indiquée par l'utilisateur à l'entrée I_5.
La partie gauche de la figure montre les constituants de l'atelier "em4 Soft". Le programme est composé d'abord des entrées et des sorties représentées par des blocs de fonctions "IN/OUT" déposés verticalement à gauche et à droite de la la surface de câblage (en blanc). Entre les deux, nous trouvons un réseau de blocs fonctionnels reliés par des connexions. Tous ces blocs sont glissés simplement de la barre de fonctions que nous pouvons voir en haut de l'image.

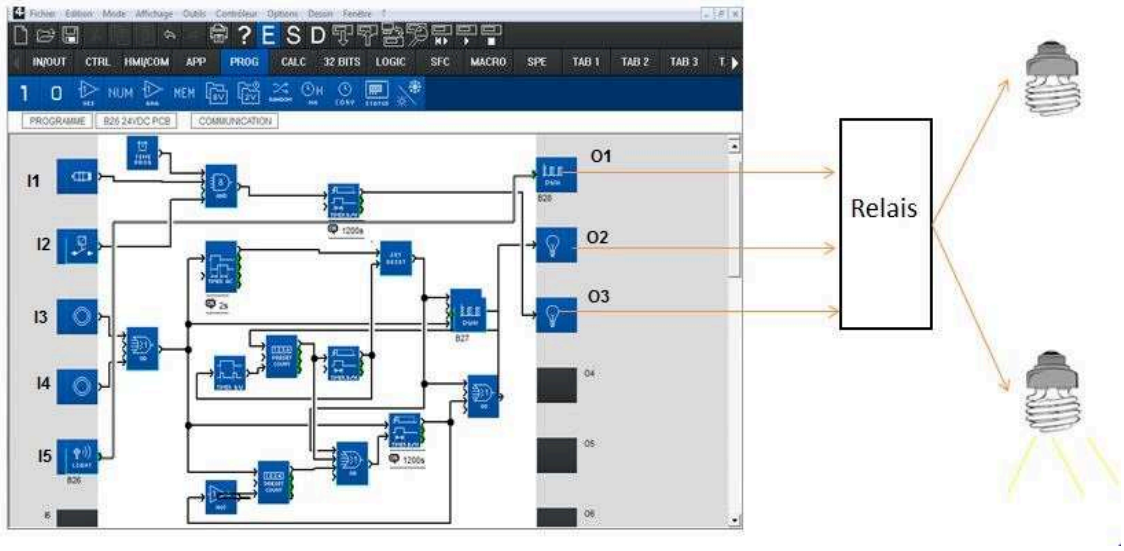


Figure 2.2 Programme d'éclairage sur em4

Par exemple, les entrées I_3 et I_4 qui représentent les deux boutons, sont reliées à un bloc fonctionnel logique "OR", qui à son tour est relié à d'autres blocs logiques et de contrôle, jusqu'à arriver à la sortie O_2.

2.3 Les langages de programmation synchrone

Le langage de programmation de em4 appartient à la catégorie des langages synchrones. Ces langages permettent de décrire, simuler, vérifier des systèmes réactifs et de compiler du code ou du matériel garantissant le même comportement que le système décrit, si l'hypothèse synchrone est vérifiée. Dans la littérature, nous trouvons plusieurs familles de langages qui sont fondés sur le modèle synchrone. On distingue principalement deux paradigmes de programmation :

- La programmation orientée flots de données comme Signal [PLGM91, BGJ91], Lustre [CPHP87] et Lucid Synchrone [CHP07]. Dans ce type de programmation, le traitement des données est constant au cours du temps et les données passent à travers un réseau d'opérateurs à un rythme préétabli. Ces langages permettent de spécifier le système par un ensemble d'équations de suites. Ces

suites, ou flots, servent à représenter les communications entre les divers composants du système.

– La programmation orientée contrôle comme le langage Esterel [BG92]. Cette programmation est particulièrement adaptée à la description des systèmes événementiels dont le comportement change fréquemment (et en fonction des données).

Comme déjà mentionné, la base commune de ces langages c'est le modèle zéro délai, qui permet au moment de la spécification, de raisonner en temps logique sans tenir compte des temps réels des calculs.
L'hypothèse synchrone permet de concevoir des langages de programmation de haut niveau offrant des constructions de concurrence mais pouvant être compilés vers du code séquentiel. On parvient ainsi à réconcilier les descriptions de haut niveau et l'efficacité d'une implémentation séquentielle.

2.3.1 Lustre

Lustre est un langage synchrone à flux de données [HCRP91]. Il a été inventé dans les années 80 par les chercheurs du laboratoire Verimag (Grenoble).

Ce langage compose les paradigmes flux de données et synchronisme. Il propose une syntaxe simple et intègre la notion de temps discret. Toutes ces caractéristiques font de lui un très bon langage pour la modélisation et la conception de systèmes de contrôle dans plusieurs domaines industriels.

Lustre a été industrialisé à l'origine par la société Verilog et actuellement par ANSYS sous le nom de SCADE et a servi pour le contrôle des centrales nucléaires, des logiciels de commande de vol et des logiciels de contrôle-commande de nombreux systèmes ferroviaires (trains, métros...).

Les programmes Lustre peuvent être compilés vers différents langages cibles : C, VHDL ... Il convient très bien ainsi pour le test automatique et la vérification.

2.3.1.1 Principaux concepts

Lustre est un langage déclaratif, c'est à dire que les objets manipulés sont définis par des équations spécifiant leurs propriétés. En effet, un programme Lustre est structuré en nœuds. Un nœud comporte un ensemble d'équations non ordonnées qui définissent chacun des paramètres de sortie en fonction des paramètres d'entrée.
Lustre est aussi un langage à flux de données, il manipule des séquences infinies de valeurs (des flux d'entrée ou de sortie). La suite de ces valeurs peut être vue comme une succession d'évènements et définit donc une notion de temps.
Le code ci-dessous représente un nœud Lustre.

```
node Moyenne(X, Y : int) returns (A : int);
    var
        S : int;      --@ variable locale
    let
        A = S / 2;        --@  équations
        S = X + Y;        --@ (l'ordre n'a pas d'importance)
    tel
```

Il permet de calculer la moyenne de deux entiers X et Y qui sont les données. L'entier A contient le résultat.

2.3.1.1.1 Types et opérateurs

En plus des constantes (par exemple : 3 ≡ 3, 3, 3, 3, ...), il y a plusieurs types de base de variables en Lustre qui sont : le type booléen(bool), entier (int) et flottant (real). Cependant, le langage Lustre étant destiné à être compilé vers des langages procéduraux, étendre les types abstraits lui permet d'accepter tout type correctement défini dans le langage hôte. Lustre propose tous les opérateurs arithmétiques et

logiques classiques. Il possède en outre deux opérateurs originaux de manipulation de flux :

- Opérateur de mémorisation : pre ("précédent") : Cet opérateur permet de se référer au cycle n à la valeur d'un flux au cycle (n-1) : si A = (a_1, a_2,..., a_n ,...) est un flux, pre(A) est le flux (nil, a_1, a_2,.., a_{n-1},...). Sa première valeur est la valeur indéfinie nil, et pour tout n$>$ 1, sa nième valeur est la (n-1)ème valeur de A.

- Opérateur d'initialisation : $\longrightarrow$ ("suivi de") Cet opérateur permet d'initialiser un flux. Si A = (a_1, a_2,..., a_n,...) et B = (b_1, b_2,..., b_n,...) sont deux flux du même type, puis "A $\longrightarrow$ B" est le flux (a_1, b_2,..., b_n,...), égal à A au premier instant, et pour toujours égal à B. En particulier, cet opérateur permet de masquer la valeur "nil" introduite par l'opérateur "pre".

2.3.1.1.2 Expressions et équations

Une expression Lustre est un terme construit avec des constantes, des variables, des opérateurs sur les valeurs (opérateurs algébriques) et des opérateurs temporels (pre et $\longrightarrow$). Une équation permet de définir une variable. Elle a pour forme syntaxique : Var =expression. Il y a synonymie complète entre la variable et l'expression apparaissant dans une équation, et donc, dans tout contexte on peut remplacer la variable par l'expression correspondante et réciproquement. Le comportement d'une variable X définie par une équation X = E, est complètement déterminé par cette équation et par le comportement des variables apparaissant dans l'expression E.

2.3.2 Esterel

Esterel, inventé par G. Berry à l'INRIA [3] appartient à la famille orientée contrôle de langages de spécification formelle spécialisée pour les systèmes réactifs.

[3] `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`

En raison de la nature textuelle de l'Esterel (par opposition à la graphique) et de sa facilité de composition, il est relativement facile de rédiger des spécifications pour les systèmes compacts avec des machines d'états très complexes. Un système avec des milliers d'états peut généralement être spécifié par un programme Esterel de seulement quelques centaines de lignes. C'est un langage impératif et **modulaire** qui permet l'expression simple du **parallélisme**. Il manipule les signaux et permet aussi une manipulation aisée du temps.

2.3.2.1 Parallélisme

Esterel fournit l'opérateur || pour une composition parallèle de ses programmes. Si P1 et P2 sont deux programmes Esterel alors P1 || P2 est également un programme Esterel, avec les caractéristiques suivantes :

- Toutes les entrées reçues de l'environnement sont disponibles pour P1 et P2.
- Toutes les sorties générées par P1 (ou P2) sont disponibles dans le même instant de P2 (ou P1).
- P1 et P2 continuent leur exécution en parallèle et la déclaration P1 || P2 se termine lorsque les deux se terminent.
- Aucune donnée ou variable ne peuvent être partagées par P1 et P2.

2.3.2.2 Modules

Une application Esterel est un ensemble de modules. Un module dans Esterel définit une unité réutilisable de comportement. Un module est comme un sous-programme avec ses propres données et ses comportements locaux. Cependant, les modules sont très différents des sous-programmes par la façon de les utiliser. Il n'y a pas d'"appel à un module" dans Esterel. Un module est utilisé comme une macro en C ; utiliser un module signifie simplement une substitution textuelle de son code en entier à l'emplacement de "l'appel". Il y a d'autres différences significatives entre un module

et un sous-programme. Par exemple, les définitions de modules récursifs ne sont pas possibles, et ainsi de suite.
Un module comporte une interface et un corps définissant le comportement. Un module Esterel a la forme suivante :

```
% Ceci est un commentaire de ligne
module nom-module:
déclarations et directives de compilation
% signaux, les variables locales, etc.
corps
.
% fin du corps du module
```

Chaque module est indépendant et Esterel fournit plusieurs constructions permettant de combiner des modules pour construire des systèmes réactifs complexes. Nous pouvons considérer un programme Esterel comme un réseau inter-connecté de modules.

2.3.2.3 Instructions temporelles

En plus des instructions classiques telles que la boucle, la conditionnelle, les affectations, les déclarations etc, Esterel propose des instructions temporelles comme par exemple :

- Émission de signaux : "emit S" = diffusion instantanée d'un signal S.
- Attente de signaux : "await S" = attendre la prochaine occurrence du signal S dans le futur strict.

Les instructions Esterel sont instantanées (affectation, émission de signaux, etc) sauf les instructions utilisant explicitement le temps (await, abort .. when , etc). Par exemple, l'instruction :

```
await 30 MILLISECOND
```

dure exactement 30 millisecondes et l'instruction :

```
await every 60 SECOND do
emit MINUTE
end
```

signifie qu'un signal MINUTE est envoyé exactement toutes les 60 occurrences du signal SECOND.

2.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté les contrôleurs logiques programmables, dont l'utilisation ne cesse de s'étendre. Nous avons donné un aperçu sur les types de CLP produits par InnoVista Sensors et l'atelier de leur programmation. Nous avons consolidé ceci par un exemple de programme qui peut être utilisé par des particuliers. Nous avons aussi décrit d'autres langages de programmation synchrones dans le même contexte et parlé des systèmes réactifs synchrones en général. Cette étude nous permettra de mieux les comprendre afin de bien cibler notre recherche dans le domaine de l'automatisation de test. Dans le chapitre suivant nous allons aborder le test des systèmes synchrones car une évolution des outils de programmation synchrone implique nécessairement des nouveaux besoins en terme de test pour la validation du fonctionnement et la détection des fautes.

Chapitre 3

Test de logiciels synchrones

3.1 Introduction du chapitre

Comme nous l'avons vu au chapitre 2, les contrôleurs logiques programmables sont très utilisés de nos jours dans l'automatisation de divers processus. Ces applications connaissent une grande évolution ce qui rajoute de l'importance à la phase de test. En outre, l'utilisateur doit pouvoir tester son programme, aussi simplement qu'il l'a conçu, afin de valider son fonctionnement et garantir le respect des spécifications. Dans ce chapitre, nous commençons par nous intéresser au test logiciel en général : ses concepts, son processus ainsi que les principales techniques. Puis, nous montrerons comment la programmation par contraintes a été utilisée dans le domaine du test. Pour finir, nous présenterons trois outils académiques de génération automatique de test pour les systèmes synchrones.

3.2 Test logiciel

D'après la définition donnée par Meyer : "Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts" [Mye79]. Un défaut est une imperfection dans un composant ou un système qui peut conduire à ce qu'un composant ou un système n'exécute pas les fonctions requises, par exemple une instruction ou une définition de données incorrecte. Un défaut, si rencontré lors de l'exécution, peut causer la défaillance d'un composant ou d'un système [SE07]. Le test du logiciel est une activité obligatoire dans l'ensemble du processus de développement [Ber01] et c'est l'une des pratiques logicielles du "V&V". D'après la norme ISO9000:2015 :

- La vérification (la première V) est le processus d'évaluation d'un système ou d'un composant pour déterminer si les produits d'une phase du développement satisfont aux conditions imposées au début de cette phase. Globalement, le processus de vérification confirme si le logiciel répond à ses spécifications techniques.
- La validation (la deuxième V) est le processus d'évaluation d'un système ou d'un composant pendant ou à la fin du processus de développement afin de déterminer s'il répond aux exigences opérationnelles spécifiées.

Les objectifs du test logiciel sont mis en évidence par la définition donnée par IEEE qui consiste à dire que "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".

3.3 Le processus de test

Le test logiciel n'est pas une seule activité mais plutôt un processus. Le processus de test, pour qu'il soit de qualité, doit suivre les mêmes principes que tous les autres processus de réalisation ou de création [Cha00]. Il commence par la planification

des tests, puis passe à la conception de cas de test, ensuite par la préparation pour l'exécution et l'évaluation de l'état jusqu'à arriver à la fermeture de test. Nous pouvons donc diviser les activités au sein d'un processus fondamental de test en 3 étapes :

- la conception des tests : cette étape a pour but de concevoir tous les tests à mener qu'ils soient manuels ou automatiques. Selon la stratégie de test employée, les tests couvrent tout ou une partie des exigences du logiciel à tester.
- l'implémentation et l'exécution des tests : l'exécution des tests est l'étape où le système est testé suivant des procédures de tests et où le résultat obtenu est comparé avec le résultat attendu afin d'identifier les éventuelles anomalies.
- l'évaluation et la production des rapports : l'objectif de cette étape est de rédiger un rapport détaillé des résultats des tests (tests réalisés, environnement de test, résultats, anomalies, recommandations).

La figure 3.1 montre les étapes fondamentales du processus du test logiciel.

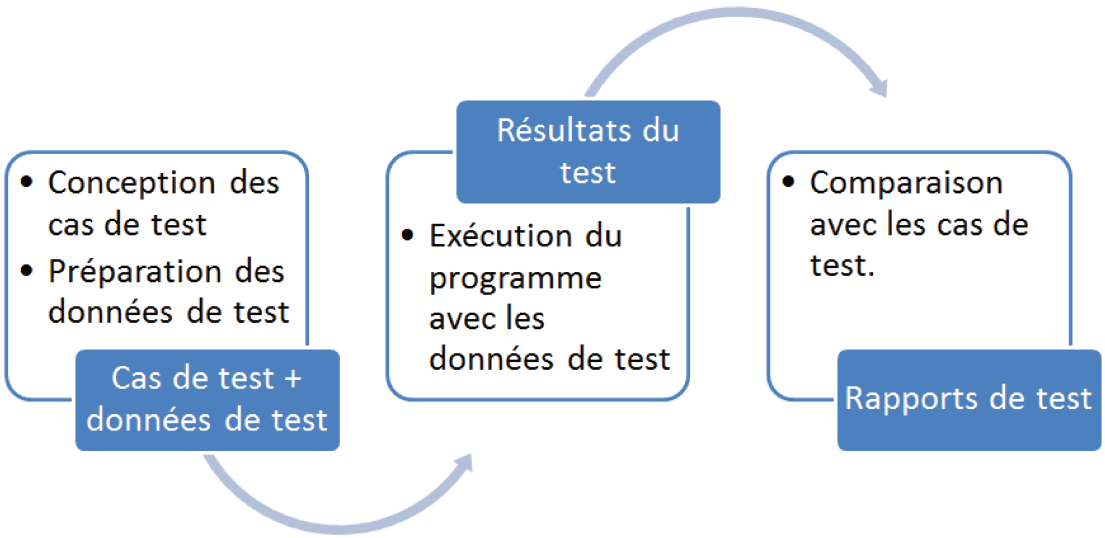


Figure 3.1 Processus du test logiciel

3.4 Les techniques de test

Essentiellement il y a deux techniques basiques de test : les tests fonctionnels (ou boîte noire) et les tests structurels (ou boîte blanche).

3.4.1 Test "boîte blanche"

La définition donnée par le glossaire de l'ISTQB (International Software Testing Qualifications Board) est la suivante : "Les tests boîte blanche sont des tests basés sur une analyse de la structure interne du composant ou système" [SE07]. Ainsi, l'objectif majeur des tests boîte blanche est de se concentrer sur la structure interne du programme pour découvrir les erreurs internes du programme. Une faute est la cause adjugée ou supposée d'une erreur ; une erreur est la partie de l'état du système qui est susceptible d'entraîner une défaillance. Cette dernière survient lorsque le service dévie de l'accomplissement de la fonction du système [AL95]. Dans le test "boîte blanche", la structure interne du programme est donc connue. Les scénarios de test en tiennent compte. Une première approximation consiste à estimer que le test est exhaustif si toutes les combinaisons de chemins de contrôle sont exécutées lors du test. Le test "boîte blanche" est également connu comme test "structurel", test "boîte transparente" ou test "boîte de verre" [Bei90]. Les connotations de "boîte transparente" et "boîte de verre" indiquent de manière appropriée une visibilité complète du fonctionnement interne du produit logiciel, en particulier, la logique et la structure du code.

En fonction de l'étape du cycle du développement, nous avons les types de tests suivants : d'unité, d'intégration, fonction/système, d'acceptation, de régression et bêta. Le test "boîte blanche" est utilisé pour trois de ces six types :

- Les tests unitaires, qui sont des tests d'unités ou de groupes de matériels ou logiciels. Une unité est un composant logiciel qui ne peut être subdivisé en d'autres composants [glo90]. Les tests unitaires sont importants pour assurer

que le code soit solide avant d'être intégré avec un autre code. Une fois que le code est intégré, la cause d'une défaillance observée est plus difficile à trouver. En outre, puisque l'ingénieur logiciel qui écrit et dirige l'unité est aussi celui qui teste, les entreprises ont souvent du mal à détecter les erreurs par les tests unitaires. Environ 65% de tous les bugs peuvent être trouvés dans les tests unitaires [Bei90].

- Les tests d'intégration, dans lesquels les composants du logiciel, le matériel des composants, ou les deux combinés sont testés pour évaluer l'interaction entre eux [glo90]. Les cas de test sont écrits pour examiner explicitement les interfaces entre les différentes unités.

- Les tests de régression, sont des tests sélectifs d'un système ou d'un composant pour vérifier que des modifications n'ont pas causé des effets indésirables et que le système ou le composant reste toujours conforme à ses exigences spécifiées [glo90]. Dans certains cas, les tests unitaires et d'intégration peuvent être sauvegardés et repris dans le cadre de tests de régression.

3.4.2 Test "boîte noire"

Le test "boîte noire" permet de s'assurer que le logiciel respecte sa spécification. Le programme est regardé comme une boîte noire dont on ignore la structure interne. Le testeur ne considère que les spécifications de sa boîte, et doit déterminer si les sorties sont correctes en fonction des entrées. Cette technique est adaptée principalement à la détection de défauts dus à une mauvaise compréhension de la spécification du logiciel. Par exemple : une fonctionnalité incorrecte ou manquante, des erreurs de comportement ou de performances... [Pre92]. Cependant, il est important de noter qu'aucune quantité de tests ne peut démontrer sans équivoque l'absence de défauts du code. Il est préférable que la personne qui planifie et exécute les tests de la boîte noire ne soit pas le programmeur du code et qu'il ne sache rien de la structure du code. Les programmeurs du code sont susceptibles de tester que le programme fait ce

qu'ils l'ont programmé à faire. Alors que les tests doivent assurer que le programme fait ce que le client veut qu'il fasse. En conséquence, la plupart des entreprises ont des testeurs indépendants pour effectuer des tests "boîte noire". Les testeurs devraient simplement être en mesure de comprendre et de spécifier ce que le résultat souhaité devrait être pour une entrée donnée dans le programme[ND12].
Le test "boîte noire" suit le processus de test classique qui s'effectue donc en trois étapes : le testeur va d'abord sélectionner des tests. Ensuite, il va exécuter ces tests sur le programme et observer son comportement. Finalement, il va comparer les comportements observés avec les comportements attendus. Cette dernière étape est réalisée au moyen d'un oracle. Un oracle de tests par définition est une source utilisée pour déterminer les résultats attendus à comparer avec les résultats obtenus de l'application en cours de tests. Un oracle peut être le système existant (comme point de référence), un manuel utilisateur, ou la connaissance spécialisée d'un individu, mais ne devrait pas être le code [SE07].

3.5 Programmation par contraintes et utilisation pour le test

3.5.1 Concepts de base

La programmation par contraintes (PPC), aussi appelée "Constraint Programming" (CP), est une approche de programmation qui consiste à écrire un ensemble de conditions (contraintes). Ces conditions seront par la suite résolues par des méthodes générales ou des **solveurs** [Apt03]. Les méthodes générales sont des techniques de réduction de l'espace de la recherche. En revanche, les solveurs sont généralement fournis sous la forme d'algorithmes spécifiques. Un exemple typique de solveurs de contraintes est un programme qui permet de résoudre des systèmes d'équations linéaires. Cette approche s'intéresse donc à la modélisation et à la résolution de problèmes décrits à l'aide de variables et de domaines qui leur sont associés et de contraintes qui restreignent les valeurs conjointement admissibles de certaines variables. L'efficacité de ce paradigme repose sur de puissants algorithmes de propaga-

tion de contraintes qui éliminent du domaine des variables les valeurs qui engendrent des solutions irréalisables. Une solution non réalisable est une solution pour laquelle au moins une contrainte est violée. Si la propagation de contraintes ne suffit pas à elle seule à établir une solution réalisable, une recherche arborescente est entreprise afin de réduire davantage le domaine des variables définissant le problème. Nous présenterons dans ce qui suit les 3 concepts centraux de la PPC à savoir : la modélisation du problème, la propagation des contraintes et la recherche des solutions.

3.5.1.1 Modélisation

La modélisation consiste à représenter un problème dans un formalisme particulier. En PPC, le formalisme cible est celui des problèmes de satisfaction de contraintes (CSP) [Apt03]. Un CSP est un triplet <X, D, C>, où :

- X est un ensemble des variables $x_1, ..., x_n$;
- D est un ensemble des domaines $D_{x_1}, ..., D_{x_n}$ (ensembles des valeurs possibles) pour ces variables ;
- C est un ensemble de contraintes $Ci(x_{i_1}, ..., x_{n_i})_{i\epsilon|C|}$. Chaque contrainte Ci restreint les valeurs que les variables $x_{i_1}, ..., x_{n_i}$ peuvent prendre simultanément.

Cette démarche constitue la principale difficulté pratique de la programmation par contraintes, car une fois représenté dans le formalisme voulu, un problème peut être résolu automatiquement à l'aide d'un solveur adapté (même si en pratique cette résolution peut être coûteuse dans certains cas). Le solveur déterminera si le problème est consistant et, le cas échéant, fournira une solution. Un CSP est dit consistant s'il admet au moins une solution. Dans le cas contraire, il est dit inconsistant.

Prenons l'exemple simple du problème de coloriage d'une carte : pour le représenter, nous devons premièrement identifier toutes les composantes du CSP, à savoir les variables, les contraintes et les domaines. Dans ce problème, nous devons colorier chaque région de la carte avec une couleur spécifique de telle façon que deux régions

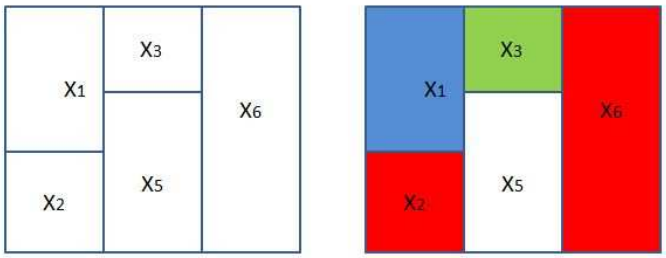


Figure 3.2 Un exemple de problème de coloriage de carte et une solution possible

adjacentes n'aient pas la même couleur. La figure 3.2 montre un exemple de problème de coloriage. Dans ce cas, les composants sont ainsi définis :

- Variables : les variables sont les choix à faire dans le problème, elles correspondent aux cinq régions à colorier X = x_1, x_2, x_3, x_4, x_5.
- Domaines : les domaines sont les options disponibles pour chaque variable. Ici, le domaine de chaque variable est l'ensemble de couleurs : blue(b), red(r), green(g) et white(w) et $D_1 = D_2 = D_3 = D_4 = D_5 = b, r, g, w$. L'assignation d'une valeur du domaineD_i à une variable x_i correspond à un choix de couleurs pour x_i.
- Contraintes : les contraintes sont les relations entre les variables qui expriment des combinaisons valides ou non-valides. L'ensemble des contraintes restreint l'ensemble de toutes les combinaisons possibles de choix de couleurs à un petit ensemble satisfaisant les conditions d'une solution du problème de coloriage. Dans notre exemple, pour chaque paire de régions voisines, une contrainte binaire les relie.

  Ainsi, nous avons :

  C = $\{c1 : x_1 \neq x_2, c2 : x_1 \neq x_3, c3 : x_1 \neq x_4, c4 : x_2 \neq x_4, c5 : x_3 \neq x_5, c6 : x_3 \neq x_4,$

  $c7 : x_4 \neq x_5\}$

  La figure 3.2 nous montre une solution possible à notre problème de coloriage

où chaque variable x_i a une valeur v $\in D_i$ respectant l'ensemble de contraintes C.

3.5.1.2 Propagation

La propagation de contraintes est le fait de réduire le domaine d'une variable afin de maintenir l'ensemble des valeurs possibles cohérent avec les contraintes du problème. Les techniques de propagation de contraintes sont utilisées pour réduire la taille de l'espace de recherche lors de la résolution d'un problème de satisfaction de contraintes par un algorithme de recherche arborescente.
Ce moteur de résolution diffuse à travers l'ensemble des contraintes les résultats de raisonnements locaux à chacune d'entre elles, obtenus par des algorithmes de filtrage. Chaque type de contrainte possède son propre algorithme de filtrage pour éliminer du domaine des variables les valeurs localement incohérentes, ne pouvant pas mener à une solution. La communication entre les différents algorithmes de filtrage se fait à travers les domaines des variables. Par exemple, pour la contrainte (x $<$ y) avec D(x)=[30, 40], D(y)=[0, 35], un algorithme de filtrage pourra supprimer les valeurs de 35 à 40 de D(x) et les valeurs de 0 à 30 de D(y).

3.5.1.3 Recherche

Comme le montre la figure 3.3, après la modélisation du problème et la propagation vient l'étape de recherche pour finir le processus. En effet, après la propagation il se peut que certaines variables ne soient pas toutes instanciées : certains domaines peuvent encore contenir plusieurs valeurs. Nous avons recours alors à des méthodes de recherche. En général, ces méthodes sont basées sur l'ajout dynamique de contraintes et la génération d'un arbre de recherche où chaque feuille représente une solution réalisable.
Par exemple, la figure 3.4 représente l'arbre de recherche du problème de coloriage. Pour des raisons de simplicité, seules les branches contenant une solution sont étendues. Les nœuds représentant une solution sont entourés.

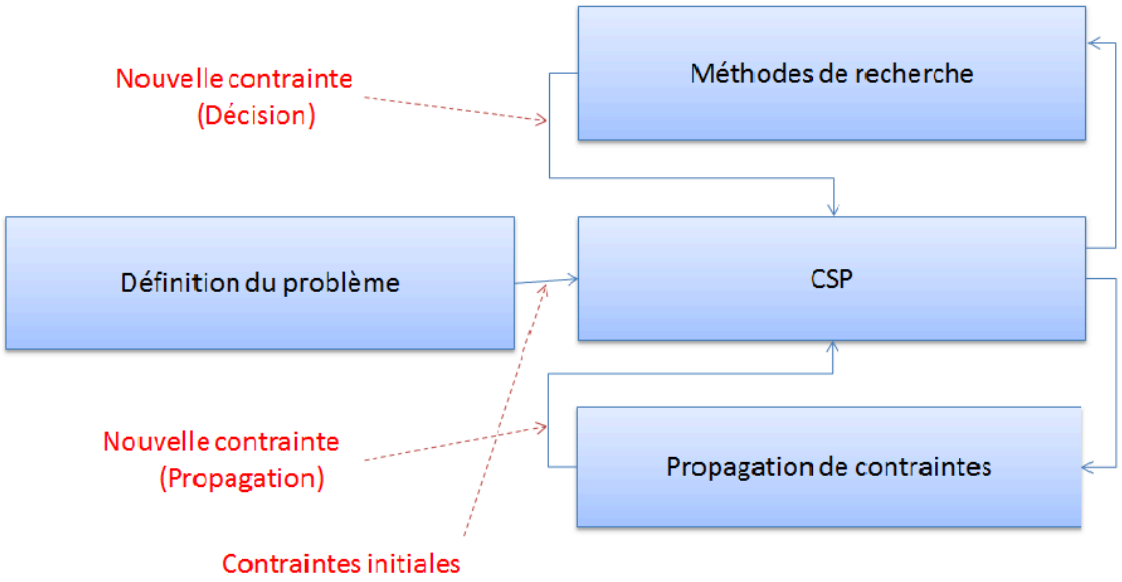


Figure 3.3 Mécanisme de la PPC

Source: from Baptiste et al.

3.5.2 De la PL vers la PLC : Programmation Logique par Contraintes

La programmation logique (PL) est le résultat des recherches menées par R. Kowalski [Kow74] et A. Colmerauer [Col90] sur un sous ensemble de la logique des prédicats du premier ordre. La programmation logique constitue un paradigme de programmation simple et déclaratif. Afin de combler la principale lacune de la PL, qui est la limitation du champ d'application, la PL a été étendue dans le but de supporter d'autres domaines que ceux que permettaient d'exprimer les formules logiques. Cette extension venait après une étude théorique, faite par Joxan Jaffar et Jean Louis Lassez en 1986, qui a donné naissance au concept général de la "Programmation Logique par Contraintes" (CLP) [JL87]. Le besoin de combiner au processus de déduction logique des algorithmes incrémentaux efficaces de résolution des contraintes appelés "solveurs de contraintes", a permis d'appliquer la CLP dans

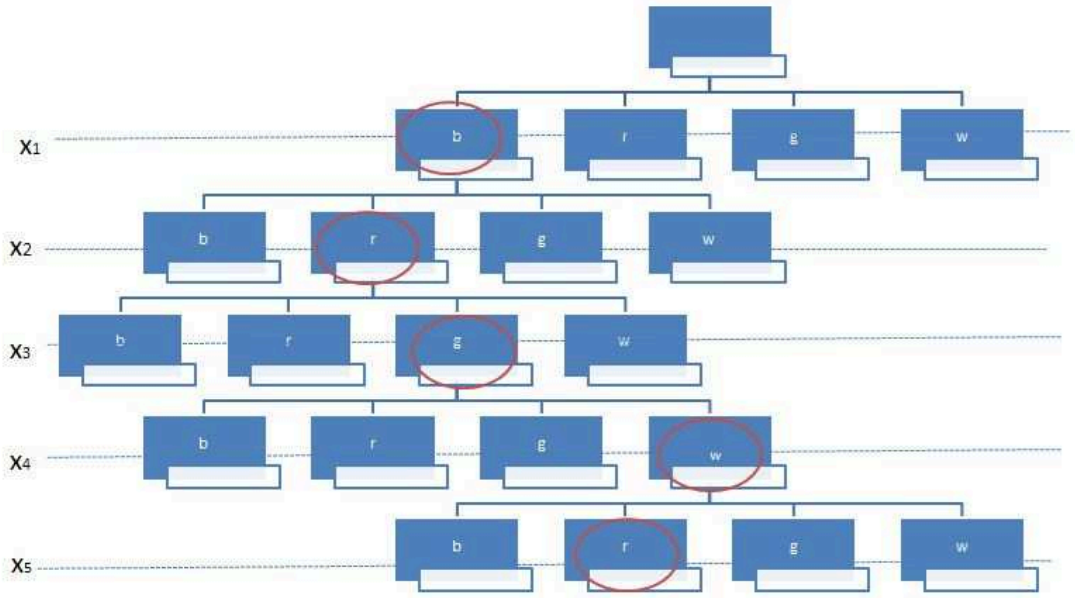


Figure 3.4 L'arbre de recherche associé au problème de coloriage

plusieurs grands domaines, tels que la recherche opérationnelle, l'intelligence artificielle, le calcul formel...

Dans ce qui suit nous allons présenter quelques notions importantes de la logique du premier ordre afin de mieux comprendre la CLP. Un langage de premier ordre est caractérisé par :

- un ensemble de variables (notées souvent x,y,z,...)
- un ensemble de constantes (notées souvent a,b,c,...)
- un ensemble de fonctions d'arités strictement positives (les constantes peuvent être vues comme des fonctions d'arité 0)
- termes : toute variable ou constante est un terme, si f est un symbole de fonction d'arité n, et t_1, .. , t_n sont des termes, f(t_1,..,tn) est un terme
- un ensemble de prédicats (relations) d'arités positives ou nulles
- atomes : expressions de la forme p(t_1,..,t_n) où p est un symbole de prédicat n-aire et t_1,.., t_n sont des termes

- formules du premier ordre : formées à partir des atomes, des constantes logiques 0 et 1 et clos par les connecteurs $\neg, \Rightarrow, \vee, \wedge, \forall$ et $\exists$. Les quantificateurs $\forall$ et $\exists$ ne portent que sur des variables.

Les programmes logiques avec contraintes ressemblent aux programmes logiques usuels, sauf qu'ils peuvent contenir des contraintes dans la définition des règles. Les langages de programmation logique avec contraintes sont une généralisation du langage Prolog. En effet après l'avènement de Prolog, à partir de ce dernier furent développés d'autres langages comme par exemple Mercury [HCS+96] ou Visual Prolog [Sco10]. Le projet japonais d'ordinateurs de $5^{\text{ème}}$ génération fut à l'origine de nombreux langages de programmation logique concurrente, tels que CS Prolog [KOM87] ou plus récemment Actor Prolog [MP15].

3.5.3 Notions de base de Prolog

Prolog est un langage de Programmation Logique par Contraintes basé sur la logique du premier ordre, il a été inventé au début des années 70 par Alain Colmerauer [Col90] à Marseille. D'ailleurs le nom Prolog est l'acronyme de PROgrammation LOGique. Sa syntaxe et son principe de fonctionnement sont radicalement différents de langages impératifs tels que C ou Java. En voici quelques notions basiques : un programme Prolog est composé d'une suite de faits et de relations entre ces faits exprimées par des règles de la forme suivante :

```
P :- Q1, Q2, ... , Qn.
```

qu'on interprète comme :

```
P est vrai si Q1, Q2,...,Qn sont vrais.
```

L' unité de base de Prolog est le prédicat. P est appelé la tête et Q1, Q2,...,Qn le corps de la règle. Chacun de ces éléments est appelé un littéral composé d'un symbole de prédicat et de paramètres ou arguments entre parenthèses séparés par des virgules. Par exemple :

```
pere(marie, pierre).
```

Les paramètres sont des termes composés de variables dénotant des objets à définir plus tard, des atomes ou des termes composés.
Une règle de la forme :

```
P.
```

est appelée un fait car il n'y a aucune condition pour sa véracité. Un programme Prolog est au moins constitué d'un ou plusieurs fait(s) car c'est à partir de lui (d'eux) que Prolog va pouvoir rechercher des preuves pour répondre aux requêtes de l'utilisateur.
Les variables sont indiquées en utilisant un ensemble de lettres, chiffres et caractères de soulignement et débutant par une majuscule ou un souligné. Par exemple :

```
Y Type _compteur
```

Les atomes sont les textes constants. Un atome est ordinairement noté soit par des nombres ou des identificateurs débutant par une minuscule. Pour introduire un atome non alphanumérique, on l'entoure d'apostrophes : ainsi '+' est un atome, + un opérateur). Par exemple :

```
16 féminin singulier
```

Prolog ne peut représenter des données complexes que par des termes composés. **Un terme composé** est noté par un foncteur et des paramètres qui sont eux-mêmes des termes. Le nombre de paramètres, nommé arité du terme, est en revanche significatif. Un terme composé est identifié par sa tête et son arité, et habituellement écrit comme foncteur/arité.
Exemples de termes composés :

```
aime(romeo, juliette)
```

Le foncteur est aime et l'arité est 2, le terme composé s'écrit aime/2.
Tout langage a besoin d'un moyen pour gérer des collections d'objets et Prolog ne

fait pas exception. **Une liste** en Prolog est une structure de données de base. Elle peut être sous l'une des formes suivantes :

- [] est la liste vide
- $[Tte|Queue]$ est la liste où le premier élément est Tte et le reste de la liste est Queue
- $[a, b, c, 3, 'Coucou']$ est une liste de constantes

Prolog est avant tout un langage de programmation logique par contraintes. Rappelons qu'une contrainte est une formule logique. Elle dénote un ensemble de solutions (les solutions de la formule) pour une interprétation logique fixée d'avance. Exemple : La contrainte X + Y = 1 dénote les deux solutions X= 0, Y = 1 et X = 1, Y = 0 si le domaine d'interprétation est N.

Exemple de résolution d'un problème de satisfaction de contraintes (CSP) avec Prolog Considérons le problème suivant : Quels sont les quatre entiers positifs dont la somme de leurs carrés est égale à un entier n donné ?
Une requête Prolog pour répondre à cette question peut être :

```
-trouver(10,L).
```

avec "trouver" un prédicat qui peut être défini dans un fichier prolog "exemple.pl" de la manière suivante (l'exemple est écrit avec Swi prolog) :

```
:- use_module(library(clpfd)).


trouver(N,L) :-
        L = [A,B,C,D],
        L ins 1..sup,
        N #= A*A + B*B + C*C + D*D,
        labeling([ ],L).
```

L étant la liste recherchée. Chaque élément de L est entre 1 et la borne supérieure du domaine. Pour rechercher une affectation de valeurs à une liste de variables, nous utilisons le prédicat "labeling". En exécution nous obtenons :

```
  ?- [exemple].
true.

?- trouver(10,L).
L = [1, 1, 2, 2] ;
L = [1, 2, 1, 2] ;
L = [1, 2, 2, 1] ;
L = [2, 1, 1, 2] ;
L = [2, 1, 2, 1] ;
L = [2, 2, 1, 1].
```

Si nous voulons que les quatre nombres soient différents, nous pouvons rajouter la contrainte :

```
all_distinct(L),
```

Ensuite, en rechargeant le fichier Prolog nous obtenons :

```
?- [exemple].
true.

?- trouver(30,L).
false.

?- trouver(30,L).
L = [1, 2, 3, 4] ;
L = [1, 2, 4, 3] ;
L = [1, 3, 2, 4] ;
```

```
L = [1, 3, 4, 2] ;
L = [1, 4, 2, 3] ;
L = [1, 4, 3, 2] ;
L = [2, 1, 3, 4] ;
L = [2, 1, 4, 3] ;
L = [2, 3, 1, 4] .
```

Nous pouvons aussi supprimer toutes ces solutions qui sont juste des réarrangements en rajoutant les contraintes suivantes :

```
A #< B, B #< C, C #< D,
```

Et nous obtenons ainsi une seule solution :

```
? - [exemple].
vrai.
? - trouver(30, L).
L = [1, 2, 3, 4].
```

Pour résumer, une fois le CSP défini, en déclarant les domaines des variables et en posant des contraintes sur ces variables, nous pouvons demander à Prolog de le résoudre, c'est-à-dire de déterminer s'il existe une ou plusieurs solutions, et le cas échéant de donner les valeurs des variables correspondantes. Prolog résout un CSP en énumérant les différentes affectations possibles de valeurs aux variables jusqu'à en trouver une qui satisfasse toutes les contraintes.

3.5.4 Utilisation de la CLP dans l'automatisation du test

La génération de données de test est le processus de génération automatique d'entrées de test pour des critères de test bien déterminés. Dans ce contexte, la programmation logique avec contraintes a souvent été utilisée.

Une approche pour générer des cas de test statiques est d'effectuer une exécution symbolique du programme [Cla76] où le contenu des variables sont des expressions plutôt que des valeurs concrètes. L'exécution symbolique produit un système de contraintes sur les variables d'entrée comprenant des conditions pour exécuter les différents chemins. La conjonction de ces contraintes représente la classe d'équivalence des entrées qui prendraient ce chemin. Les outils de test peuvent ensuite tester la fonctionnalité d'une application en exécutant ces entrées de test et en vérifiant que la sortie est comme prévue.

Ce type d'outils a fait objet de plusieurs recherches : citons les travaux de [Sel09], qui réalisent une extension des techniques de test proposées par l'outil Lutess, qui sera présenté plus en détails dans le paragraphe suivant, afin de prendre en compte des logiciels à tester qui comportent des entrées/sorties numériques. La génération de données de test est abordée en s'appuyant sur les techniques de programmation par contraintes.

Citons aussi l'outil Genetta [opa08], dans le cadre de la construction automatique de générateurs pour le test structurel statistique, qui met en œuvre une bibliothèque de résolution de contraintes probabilistes et l'utilise pour la sélection uniforme des chemins exécutables dans un graphe de contrôle, et ceci pour des programmes C.

Dans le même cadre que Lutess, c'est-à-dire des programmes synchrones, nous trouvons l'outil de test GATeL [MA00] que nous allons aussi présenter en détails ultérieurement. GATeL 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.

En 2001, Pretschner a développé l'outil de génération de cas de test orienté modèle AutoFocus [Pre01]. Comme GATeL, il est basé sur la PLC. Il permet de modéliser un système en une collection de composants communicants, qui peuvent être

décomposés hiérarchiquement en des sous-réseaux de composants synchrones. L'environnement de test d'Autofocus donne la possibilité de traduire ces modèles dans un langage de PLC et d'exécuter symboliquement ces modèles transformés.

3.6 Outils de génération automatique de données de test pour les systèmes synchrones

Le premier outil parmi ceux qui servent à la génération automatique de données qui nous a particulièrement inspiré dans nos travaux de recherche est "Lutess".

3.6.1 Lutess

Lutess [dBORZ99, SP06] est un outil de test pour la validation des programmes réactifs synchrones qui est basé sur le langage Lustre [CPHP87]. Il réalise un test fonctionnel en "boîte noire". La génération automatique de tests nécessite trois éléments : le SST, un oracle et une spécification de l'environnement (fig 3.5).

Le programme sous test est donné sous sa forme exécutable et supposé avoir un

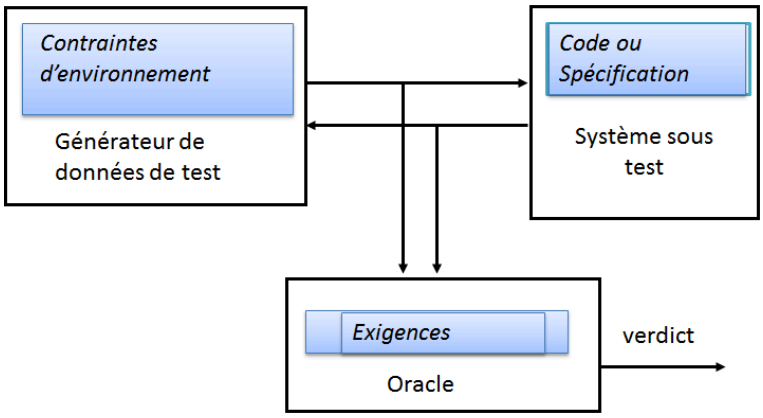


Figure 3.5 Principe de Lutess

comportement synchrone. La génération des séquences de test se fait à l'aide de spécifications formelles, dans un langage similaire à Lustre. À partir de ces spécifications, Lutess permet de générer automatiquement et dynamiquement des données

de test qui respectent les contraintes de l'environnement dans lequel le système évolue. La description de l'environnement est fournie sous la forme d'un nœud spécial (testnode) comme le montre la figure fig 3.5. Son rôle est d'abord d'exclure les comportements irréalistes de l'environnement (opérateur environment) qui ne doivent pas être pris en compte pendant la génération de données de test. Il a pour but aussi de guider la génération de tests au moyen de directives ajoutées par le testeur [Par07]. Les entrées de ce nœud sont les sorties du SST et ses sorties sont les entrées du SST. Lutess fonctionne en cycles "action-réaction" comme suit : Les sé-

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

Figure 3.6 Le nœud testnode et l'opérateur environment

quences de test sont générées à la volée tandis que le SST est exécuté. D'abord, le générateur fournit un vecteur d'entrée initiale pour le SST. Alors, le SST et le générateur calculent de manière alternée des vecteurs de sortie et des vecteurs d'entrée, respectivement. L'oracle est alimenté à la fois, du flux d'entrée et du flux de sortie, et calcule le verdict.

Lutess a été développé entre 1996 et 2009, au cours de 5 thèses différentes [Par96, dB99, Zua, Vas04, Sel09]. Une première version de Lutess repose sur les travaux successifs de [Par96, dBORZ99, Vas04]. Le générateur de données de test s'appuie sur des techniques de BDD (Binary Decision Diagram). Cette version ne traite que des entrées booléennes. Différentes stratégies de guidage sont proposées au fil des thèses,

de façon à rendre la génération aléatoire plus performante et adaptée aux besoins des utilisateurs comme par exemple les schémas comportementaux qui permettent de conduire la génération vers une situation précise, en indiquant les situations intermédiaires dans lesquelles l'environnement doit se trouver. Ils peuvent être vus comme un ensemble de spécifications de probabilités conditionnelles. Dans sa thèse, Seljimi B. [SP06] propose de prendre en compte les données entières. Pour cela, il propose une nouvelle version de Lutess (Lutess-V2), dans laquelle la génération de données de test se fait à l'aide de la PLC. Dans cette version, les stratégies de guidage sont plus limitées que pour la première version. Elle utilise toujours la probabilité conditionnelle mais il n'y a plus de schémas comportementaux. Mais à la différence de Lutess-V1, où les probabilités ne portaient que sur les variables d'entrée du logiciel, en Lutess-V2 elles peuvent porter sur une expression quelconque.

3.6.2 Lurette

L'approche de Lurette[1] [JRB06] est globalement comparable à Lutess car elle est également basée sur le langage Lustre et se construit à partir des mêmes 3 éléments : la description de l'environnement, le SST et l'oracle. En outre, les deux outils construisent leurs générateurs de séquences de test à partir d'une description de l'environnement écrite en Lustre tandis que le SST est testé comme une boîte noire. Cependant, Lurette exige que le SST soit donné comme un fichier C qui met en œuvre un ensemble prédéfini de procédures.

La méthode de test de Lurette suit le principe suivant : la description de l'environnement est utilisée pour exprimer des séquences de tests pertinents et intéressants. Le terme "pertinence" se réfère à ces propriétés qui limitent l'environnement lui-même et le terme "intérêt" se réfère à l'objectif de test. Cette approche est basée sur le modèle des observateurs [HLR93] qui offre la possibilité de connaître complètement l'état du système et de l'environnement. La description de l'environnement et l'objectif de test sont écrits dans le même observateur. Cet observateur est alimenté par

[1] http ://www-verimag.imag.fr/Lurette,107.html

les entrées et les sorties du SST. Une séquence de test est générée par Lurette uniformément et d'une façon aléatoire. L'outil construit et soumet au logiciel sous test un vecteur d'entrées respectant les contraintes se trouvant dans l'observateur. Chacune des entrées composant le vecteur peut être de type entier ou booléen. Lurette doit changer la description de l'environnement pour calculer un nouveau vecteur d'entrée pour le SST, en se basant sur l'état actuel de l'environnement et la dernière sortie du SST. Considérons par exemple ce code Lustre d'un observateur tiré de [RWNH98] :

```
relevant = (X = 0) -> if A then (B or (X > Y))
                              else (X + Y <= 10) and (X * pre Y < 12);
```

A et B sont des entrées booléennes et X et Y sont des entrées numériques. Nous supposons que nous sommes dans un état non initial où la valeur de pre Y est 3. La fonction qui réalise le choix des entrées identifie 3 contraintes linéaires :

```
C1 = (X > Y)
C2 = (X + Y <= 10)
C3 = (X < 4)
```

Générer les entrées revient à chercher des valeurs d'entrée telles que la fonction booléenne suivante soit vraie : $A \wedge (B \vee C1) \vee \neg A \wedge C2 \wedge C3$
L'oracle est également alimenté par les entrées et les sorties du SST afin d'évaluer l'exactitude de la séquence. Le résultat de l'oracle est soit une réussite soit un échec.

Comme pour Lutess, il y a eu deux versions : La première (Lurette-V1) est décrite dans le paragraphe précédent et la seconde (Lurette-V2) a été aussi réalisée au sein de Verimag et dont la particularité réside dans l'apparition du langage Lutin [RRJ08]. L'objectif principal de ce dernier est la description des systèmes réactifs. Il présente d'autres objectifs comme de pouvoir décrire des scénarios de test, en ajoutant le non déterminisme sur les solutions, mais il a aussi été utilisé pour la simulation. Les programmes écrits en Lutin sont compilés dans un format interne à l'outil Lurette. Ce format appelé Lucky [JHJR04] est basé sur un automate explicite. Lutin sert comme

un langage de haut niveau pour concevoir les scénarios de test. Le langage Lutin est basé sur les expressions régulières. Les littéraux sont des contraintes exprimées dans une syntaxe très ressemblante à Lustre qui relie des entrées, sorties et des valeurs mémorisées (pre). Les contraintes définissent l'ensemble possible de sorties pour un instant logique. Les instants peuvent être enchaînés en utilisant les opérateurs de concaténation (fby, de l'anglais "followed by") et de boucle (loop).

Comme dans Lustre, les programmes Lutin sont structurés en nœuds qui peuvent être exécutés en parallèle, et sur le flux d'instants logiques [JHR13]. Voici un exemple de nœud LUTIN qui illustre le non déterminisme non seulement dans les données mais aussi dans le contrôle :

```
node random() returns (x:int) =
loop [10,15] 0 <= x and x <= 50
```

Ce nœud va générer n fois un entier x compris entre 0 et 50, avec n lui-même compris entre 10 et 15. Ci-dessous un autre exemple illustrant l'utilisation de l'opérateur "fby" :

```
node sn_gen() returns ( sn : int ) =
loop [10 ,20] sn =1 fby
loop [20 ,30] sn =2
```

Ce nœud génère une séquence finie d'entier, sans utiliser d'entrées. Il utilise d'abord une contrainte atomique qui affecte `sn` à 1, durant 10 à 20 cycles. Ensuite, il affecte 2 à `sn` durant un nombre de cycles entre 20 et 30, puis s'arrête de s'exécuter [JDDML14]. Le testeur peut également mettre une certaine condition pour passer d'une contrainte à une autre. Cette condition est exprimée sous forme d'une contrainte aussi comme dans le code qui suit où nous remarquons que dans la première boucle `loop` : dès que le système devient stable ("is_stable" est à vraie), le générateur passera aux contraintes suivantes (celles après `fby`) :

```
-- (1) Attendre la stabilisation
```

```
loop { not is_stable and O = pre(O) }
fby -- (2) choisir une nouvelle valeur de la variable O
{  O > pre(O) }
fby -- attendre que le changement soit pris en compte
loop { is_stable and O = pre(O) }
```

3.6.3 GATeL

GATeL [MB05], développé par le commissariat à l'énergie atomique et aux énergies alternatives (CEA), est un outil de test en "boîte blanche". Il traduit en un programme Lustre la spécification de l'environnement du SST dans une représentation Prolog équivalente puis calcule une séquence d'entrées en essayant de trouver une solution qui satisfait à la fois les contraintes et un objectif de test. L'approche Gatel est assez différente de celles des deux autres outils basés sur Lustre (Lutess et Lurette) : ces deux derniers génèrent les séquences de test à la volée, à savoir, à chaque étape, les sorties du SST sont utilisées pour calculer une nouvelle entrée pour le SST sur la base de la description de l'environnement et l'objectif de test. Ce processus est itéré soit jusqu'à ce qu'un verdict de test négatif est produit, ou jusqu'à ce que la longueur maximale de la séquence soit atteinte. En revanche, GATeL commence avec l'état final à atteindre par la séquence de test. Ainsi la génération de données de test est effectuée avant que le système sous test ne soit exécuté (et non lors de son exécution).

GATeL exige le code Lustre du programme en cours de test, la description de son environnement ainsi qu'un objectif de test fixé par le testeur. Un objectif de test peut être une propriété de sureté ou un prédicat de chemin. Les propriétés invariantes qui doivent tenir dans chaque étape de la séquence de test générée sont exprimées avec la directive "ASSERT", par exemple :

```
assert true -> not ( signal and pre(signal) )
```

Cette "assertion" exige que "signal" ne soit pas vrai dans deux cycles consécutifs. Les propriétés qui doivent être satisfaites au moins une fois sont évaluées grâce à la directive "REACH", par exemple le franchissement d'un seuil :

```
reach (pre(Val) < seuil and Val >= seuil)
```

À partir de système sous test et de la description de l'environnement, GATeL essaie de trouver une séquence de test qui satisfait les deux propriétés exprimées dans les directives "ASSERT" et "REACH". Si un tel cas de test est trouvé, il est exécuté sur le système sous test et la sortie générée est comparée à l'une correspondante prévue à partir du cas de test pré-calculé. Si ces deux correspondent, alors le cas de test passe, sinon il échoue.

Chapitre 4

Testium : L'approche globale

4.1 Introduction du chapitre

Après avoir étudié des approches pour l'automatisation du test de systèmes réactifs synchrones, nous commençons ce chapitre par l'étude de leur adéquation à notre contexte de recherche. Cette étude nous permet de définir notre approche dont nous donnons un aperçu de l'architecture globale et qui préfigure un nouvel outil de test baptisé "Testium", orienté test en boîte noire et basé sur la PLC.

4.2 Rappel des besoins

Nous avons vu que l'atelier "em4 Soft", présenté au chapitre 2, permet aux utilisateurs de em4 de programmer des applications pour les automates em4 en toute simplicité en activant le mode d'édition. Il met à disposition des programmeurs plusieurs blocs fonctionnels spécifiques et prédéfinis pour concevoir leurs applications.

Les deux autres modes disponibles, qui sont le mode de simulation et le mode monitoring, permettent quant à eux de tester manuellement ces applications. En effet, l'utilisateur peut changer les valeurs des entrées et voir le résultat s'afficher au niveau des sorties. Il est clair que l'utilisateur ne peut pas réaliser un test rigoureux de l'application et cela demande aussi beaucoup de temps.

- Vue l'importance de la phase de test avant la mise en marche du système contrôlé par le CLP (critique ou pas), le besoin qui apparaît ici est surtout d'avoir un moyen de générer automatiquement les données de test pour plus d'efficacité.
- Le deuxième besoin que nous dénotons est la simplicité d'utilisation ; tester les applications des CLP doit être aussi simple que leur programmation et à la portée de tout le monde. L'outil de test souhaité doit être conçu avec cet objectif.
- Troisièmement, l'utilisateur doit pouvoir écrire des scénarios de tests fonctionnels de la manière la plus naturelle et intuitive possible c'est à dire comme s'il racontait une histoire. Par exemple pour le système d'éclairage, il pourrait dire qu'il veut appuyer sur le bouton de marche un certain temps jusqu'à ce que le mode permanent s'active. Une fois activé, il appuie de nouveau. Par la suite, il passe devant la maison puis repasse 2 fois successives après un moment, etc. Cette fonctionnalité serait aussi avantageuse pour gagner en temps et en rigueur. L'outil doit permettre aussi l'obtention de données de test reflétant l'interaction réelle du contrôleur avec son environnement dans lequel il est utilisé. L'utilisateur doit pouvoir générer facilement des données de test qui traduiront les conditions dans lesquelles le système va être placé et la manière avec laquelle il sera utilisé généralement, sauf dans le cas où il veut le tester dans des conditions extrêmes et il doit aussi pouvoir le faire.

- Certains outils existants proposent le non déterminisme dans la génération automatique de test. Il serait intéressant de l'introduire aussi dans notre approche car il permet d'obtenir des résultats non triviaux et minimise l'intervention du testeur dans le processus de test.

- Dans l'utilisation des contrôleurs logiques programmables, il y a souvent recours à des opérateurs de gestion de temps à savoir : de planification, des temporisateurs ...
  Il est donc nécessaire d'introduire d'une certaine manière la notion du temps dans l'outil de génération de test souhaité.

- Il est intéressant aussi qu'un oracle soit intégré dans l'outil de génération de test. Cet oracle permettra de détecter les erreurs et d'indiquer en plus des degrés de gravité de ces erreurs. La définition des spécifications sur lesquelles se basera l'oracle devrait aussi être simple.

Dans la section suivante, nous analysons les approches existantes afin d'en identifier les limites, par rapport aux besoins énoncés mais également pour en extraire les caractéristiques intéressantes.

4.3 Limites des outils existants

Les approches présentées dans le chapitre 3 sont spécifiquement conçues pour la programmation synchrone. Ils sont soit des prototypes de recherche soit des outils de l'industrie destinés à être utilisés par les experts du domaine et du test.
Avec Lutess, nous ne pouvons pas exprimer des scénarios de test et décrire ainsi une évolution dans le temps de certains éléments qui interagissent avec le système sous test. Dans les deux versions de Lutess, la notion du temps n'intervient pas. Il est important de pouvoir faire apparaître le temps dans les scénarios, d'autant plus que l'atelier d'InnoVista offre des composants permettant de manipuler des variables en temps logique. Lutess est surtout basé sur l'expression d'invariants de l'environne-

ment écrits dans un pseudo-Lustre. Par exemple si nous voulons générer une entrée température avec la contrainte d'un intervalle nous aurons le code suivant :

```
environment(
Temp>=-20 and Temp=<60
)
```

Imaginons maintenant le cas où nous pouvons manipuler la température et voulons imposer une certaine évolution de cette température dans le temps, autrement dit écrire un scénario comme suit : partir d'une température égale à 10, augmenter la température à chaque top d'horloge jusqu'à atteindre la valeur 15 ° puis la fixer puis la diminuer etc...
La version actuelle de Lutess ne permet pas d'exprimer d'une manière naturelle ce scénario. Il est possible d'écrire par exemple dans le nœud environment :

```
environment(
            Temp = if (pre(Temp)>= 15 and (not att))
            then 16 else pre(Temp) + 1
            att = false -> if (pre(Temp)>= 16)
            then true else false )
```

Ce code en Lutess représente juste une partie d'un éventuel scénario. Plus le scénario est long, plus il se complique et utilise plus de variables auxiliaires comme ici la variable "att". C'est clair aussi qu'il faut être un vrai connaisseur du langage Lustre pour pouvoir l'écrire.

Pour la première version de Lurette, le comportement de l'environnement du SST est décrit exclusivement en Lustre. Les observateurs sont constitués d'un ensemble de contraintes (linéaires) en plus de variables booléennes et numériques. Le but de Lurette est de résoudre ces contraintes et d'en tirer une valeur parmi les solutions pour produire un vecteur d'entrée pour le SST. Mais, d'un point de vue du langage,

Lustre n'est pas très pratique, en particulier pour exprimer des séquences de différents scénarios de test ou avoir un certain contrôle sur la distribution probabiliste des solutions. Certains essais ont été réalisés pour améliorer l'outil Lurette afin de remédier à ce point comme avec l'apparition du langage Lutin. Ce dernier exprime des réactions atomiques du système à travers des relations entre les sorties et les entrées. Il est apparu comme une extension stochastique de Lustre. Il a cependant évolué récemment pour permettre plus de flexibilité quant à l'écriture des scénarios de test(cf. Chap 3.6.2) mais il parait toujours complexe et nécessite une bonne connaissance du langage académique Lustre.

L'approche Gatel est tout à fait différente de celles des deux outils connexes basés sur Lustre (Lurette et Lutess). En effet, il nécessite le SST ou une spécification complète du SST, une description de l'environnement et un objectif de test. Ces trois éléments doivent être fournis en code source Lustre et impliqués dans le processus de génération de données de test. De plus, il nécessite une grande intervention humaine durant le déroulement du test. En effet, la stratégie de sélection des données de test est totalement interactive : Gatel divise le système de contraintes de sorte que chaque sous-système caractérise une classe particulière de comportement pour atteindre un objectif. Cette séparation est traitée par le testeur en appliquant des décompositions prédéfinies d'opérateurs sur les expressions Lustre qui représentent le système de contraintes. Et c'est aussi au testeur de décider d'arrêter ou pas cette division.
Le tableau de la figure 4.1 montre une comparaison entre les différents outils de génération de test que nous avons étudiés.
Dans notre contexte de recherche, nous visons à avoir un outil de génération automatique de test qui indépendamment du code du système à tester, permettra à des non-experts d'écrire des scénarios de tests et de produire facilement des entrées qui répondent aux exigences de l'environnement et aux objectifs du testeur. Alors suite à notre étude, nous avons décidé de concevoir un nouvel outil, que nous avons ap-

	Type du test	Génération basée sur :	scénario	Notion du temps	Particularités
Lutess V1	Boîte noire	BDD	Pas de notion de scénario à proprement parler mais des schémas comportementaux	non	Basé sur Lustre, l'utilisation est complexe pour un non - expert
Lutess V2	Boîte noire	PLC	Pas de scénarios	non	
Lurette V1	Boîte noire	BDD	Pas de scénarios	non	Exige une compilation du SST vers un programme C.
Lurette V2	Boîte noire	BDD	Scénarios avec Lutin	non	Lutin est une évolution stochastique de Lustre: donc nécessite toujours une bonne connaissance du langage.
Gatel	Boîte blanche	PLC	Pas de scénarios	non	• Nécessite le code SST en Lustre • Importante intervention du testeur. • La génération de données de test est effectuée avant l'exécution du système

Figure 4.1 Tableau comparatif entre les outils étudiés

pelé "Testium", et ceci en essayant de surmonter les limitations des outils existants tout en s'inspirant des points positifs qui pourraient nous aider dans notre approche.

4.4 Vers une nouvelle approche : l'outil Testium

Pour essayer de répondre à nos besoins précédemment détaillés et élargir notre champ de réflexion et de recherche, nous avons décidé de concevoir un nouveau langage dédié à la description des tests à générer. Afin que ce langage baptisé "SPTL" (Synchronous Programs Testing Language) respecte le critère de simplicité d'utilisation, nous avons pensé à le doter d'une syntaxe relativement simple.

En effet, les utilisateurs des contrôleurs logiques programmables ne seront pas dépaysés avec le nouveau langage de test étant donné qu'il ressemble au langage de programmation de em4 dans le sens où :

- il manipule des variables de mêmes types,
- il possède des opérateurs semblables à ceux d'em4,
- il introduit la notion du temps.

Nous avons aussi réfléchi à un moyen qui permettra de guider les testeurs non-experts dans leurs tâches. Il est évident que l'environnement du système à tester est un grand facteur dans le choix des tests. Afin d'en tenir compte, nous avons introduit la notion de "Profil d'utilisation" dans le langage SPTL. Un profil apparaît sous la forme de contraintes sur les données de test à générer. Ce qui évitera d'avoir des valeurs inutiles. Nous avons aussi défini le concept de "Catégories" dans le but d'aider le testeur dans la conception des profils. Ces catégories engloberont les contraintes caractéristiques de chaque environnement d'utilisation. Ces catégories peuvent être prédéfinies pour obtenir à la fin des profils totalement ou partiellement prédéfinis. Le testeur n'aura qu'à choisir le profil qui lui convient sans avoir à l'écrire lui même.

En plus, nous nous sommes inspirés de Lutess dans l'ajout du non déterminisme à travers l'utilisation d'un opérateur pour exprimer la probabilité dans les contraintes. Nous avons ajouté la possibilité d'écrire des scénarios de test d'une manière intui-

tive : le testeur peut faire succéder des évènements, imposer des conditions d'arrêt etc ... Ceci n'était pas possible dans les outils existants comme Lutess et les premières versions de Lurette. Nous avons remarqué cependant que Lutin a évolué récemment d'une manière assez semblable à ce que nous voulons avoir dans nos scénarios comme : la succession de contraintes, la possibilité de fixer un nombre de cycles pour appliquer des contraintes données... (cf. Chap 3.6.2)

Les programmes synchrones ont un comportement cyclique (voir la figure 4.2) : à chaque cycle, toutes les entrées sont lues, traitées et toutes les sorties sont émises simultanément et théoriquement instantanément. Nous avons pensé alors que le processus de test de notre outil devrait consister lui aussi en une séquence d'échanges entre le générateur et le système sous test d'une manière cyclique.

L'idée globale qui nous a guidé pour le développement du nouvel outil baptisé "Testium" c'est donc qu'à partir d'un modèle écrit dans un langage simple d'utilisation qui permet de décrire l'environnement du système à l'aide des profils et d'écrire des scénarios de test sous forme de contraintes, nous pourrons générer des données de test qui respectent ces contraintes.

"Testium" enverra ces données au SST qui à son tour envoie ses sorties vers le générateur de test pour qu'il en tienne compte dans son calcul des entrées suivantes, et ainsi de suite. Il est intéressant qu'un oracle, qui observe les entrées et les sorties échangées pour détecter les défaillances, soit intégré dans l'outil de test.

Nous avons vu à une étape de ce processus la nécessité d'avoir un mécanisme pour faciliter la résolution des contraintes écrites en SPTL et générer les données de test. Nous nous sommes inspirés de Lutess V2 dans l'automatisation de la génération de test, à savoir l'utilisation de la Programmation Logique avec Contraintes [JMMJ98] qui a prouvé son efficacité auparavant dans le domaine de génération automatique de données de test.

Par conséquent, l'outil "Testium" est ainsi basé sur deux grandes parties qui seront décrites plus en détails dans les chapitres qui suivent : le programme SPTL et le générateur en Prolog. Il est donc possible pour les testeurs d'écrire un pro-

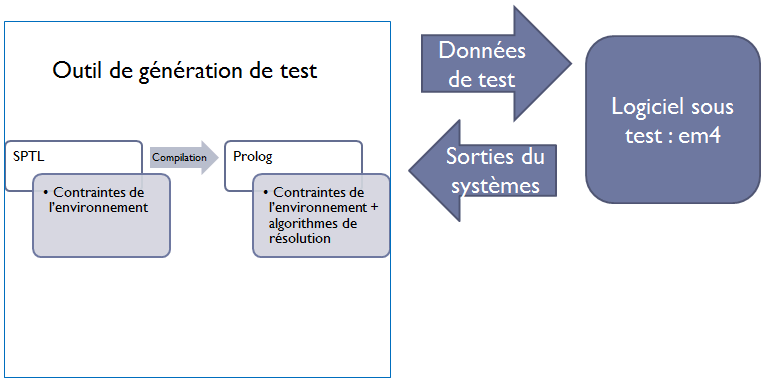


FIGURE 4.2 Processus de génération de test

gramme SPTL qui contiendra des contraintes invariantes sur la façon avec laquelle les entrées du programme sous test se comportent. Ce programme peut également contenir des scénarios dynamiques correspondant aux objectifs du test toujours sous forme de contraintes. Les cas de test générés doivent répondre aux contraintes définies [TMDP13, MDP14]. Par exemple, pour le système présenté dans le chapitre 2 (voir 2.2), une contrainte réaliste peut être la suivante : la nuit, le commutateur sensible à la lumière aura la valeur "true". Plus de contraintes complexes peuvent traiter le comportement temporel des variables d'entrée ou prendre en compte les sorties du programme en cours de test. Par exemple, tant que le mode "permanent" n'est pas encore activé, on garde le bouton de marche enfoncé c'est à dire son entrée correspondante reste à "true" durant un certain temps.

Comme le montre la figure 4.2, le programme SPTL est ensuite traduit en un générateur écrit en Prolog. Beaucoup de travaux de recherche ont montré l'efficacité de la PLC dans la génération automatique des données de test. Par exemple, nous pouvons citer ATGen [Meu01] qui est un outil de génération automatique de test basé sur la PLC et destiné essentiellement aux programmes écrits en SPARK Ada[1]. Nous pouvons noter aussi les travaux de [SF12], qui ont expérimenté quelques exemples d'approches de génération automatique de test de la littérature pour montrer que

[1] `http://www.adaic.org/advantages/spark-ada/`

l'utilisation des contraintes permet d'écrire des générateurs plus simples et plus efficaces que les autres.

Dans notre prototype d'outil, le programme SPTL est traduit en un ensemble de prédicats Prolog. Le générateur de séquences de test exécute une boucle pour un nombre donné de cycles. A chaque cycle, les contraintes de l'environnement et de l'étape actuelle du scénario sont traitées. Toute solution du système des contraintes est un vecteur d'entrée valide. De ce fait, pour générer un vecteur d'entrée de test, il faut trouver une solution du système des contraintes. Seule une solution entre toutes les solutions possibles est choisie et transmise au SST.

La première composante de notre outil "Testium" est SPTL. Dans le chapitre suivant nous allons donner la syntaxe et la sémantique du langage à travers l'exemple de programme em4 présenté dans le chapitre 2.

Chapitre 5

Testium : spécification avec SPTL

5.1 Introduction du chapitre

Comme nous l'avons évoqué au chapitre précédent, l'objectif principal de ce travail de thèse est de concevoir un environnement de test pour les programmes synchrones qui soit efficace et facile à utiliser par des non-experts. Les utilisateurs doivent être en mesure de générer des séquences d'entrées de test en conformité avec des spécifications du fonctionnement du système ainsi que de l'environnement extérieur. Comme première étape, nous avons défini un langage de description de test : SPTL. Dans ce chapitre, nous commençons par donner les concepts de base du langage, en particulier les profils et les scénarios. Par la suite, nous proposons une définition formelle de SPTL. Nous présentons la syntaxe et la sémantique à travers l'exemple du système d'éclairage déjà introduit au chapitre 2.

5.2 Concepts de base

5.2.1 Profil

En se basant sur l'idée que la performance et le fonctionnement du système sont dépendants significativement de l'environnement dans lequel il opère, nous introduisons la notion de "Profil". Un profil représente une utilisation possible du système et il est lié au contexte dans lequel il évolue.
Nous trouvons cette notion dans ZigBee [ZBM+07], par exemple, où les profils définissent les formats de messages et les actions correspondantes visant à permettre aux appareils de demander, d'émettre des données et de savoir comment les interpréter.
Plus généralement, la notion de "profil opérationnel" a été introduite par John Musa [Mus93]. Le profil a été défini comme une caractérisation quantitative de la façon avec laquelle le système sera utilisé. Nous pensons reprendre cette notion de profil. Un profil dans SPTL correspond à un certain nombre de contraintes d'utilisation qui va nécessairement affecter le test. Nous proposons des moyens pour aider à la rédaction des profils. Il y a deux cas : soit l'utilisateur est un connaisseur de l'application et il va caractériser l'environnement de différents points de vue et une combinaison de ces contraintes écrites vont constituer les profils ou bien ces profils seront prédéfinis par le fabricant du contrôleur ou du système en général pour les testeurs non-experts, comme un point de départ pour générer des données de test. Ces profils peuvent être bien sûr modifiés ou enrichis.

5.2.1.1 Catégories et groupes

Le programmeur doit définir certaines catégories de contraintes générales. Comme le montre la figure 5.1, chaque catégorie contient certains groupes de contraintes. Les contraintes de chaque groupe définissent les limites liées à un environnement bien déterminé ou notamment à des usagers. Un profil est une combinaison de groupes de différentes catégories (voir la figure **??**). Elle est représentée par l'union des

contraintes de ces groupes. Les groupes et les catégories peuvent être prédéfinis, ainsi ces notions permettent non seulement de faciliter la création des profils mais aussi d'obtenir des profils partiellement ou totalement prédéfinis.

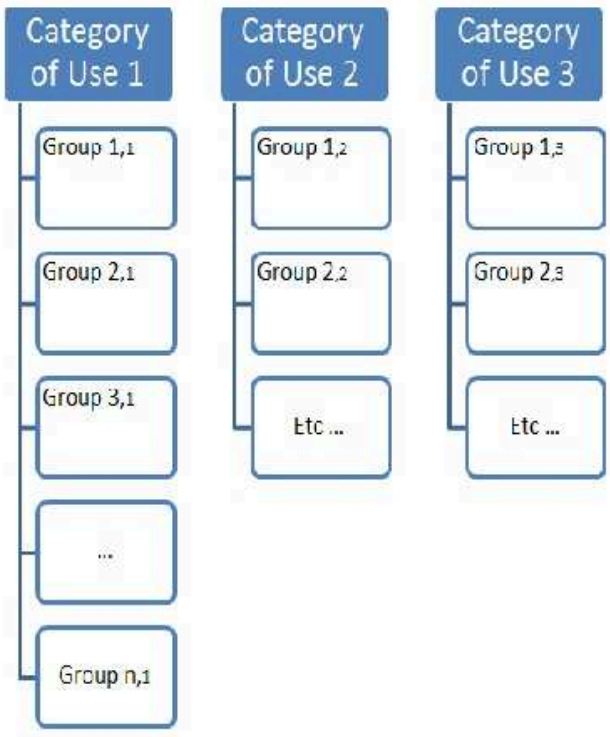


Figure 5.1 Categories

Group 1,1 = { C1, C2, C3 , C4 } , Group2,1 = { C5, C6 } ...

avec Ci : contrainte $\forall i \in N$

Par exemple nous pouvons avoir un profil composé de Group1,1, Group3,2 and Group 2,3 :

Profil1 = Group1,1 $\cup$ Group3,2 $\cup$ Group 2,3

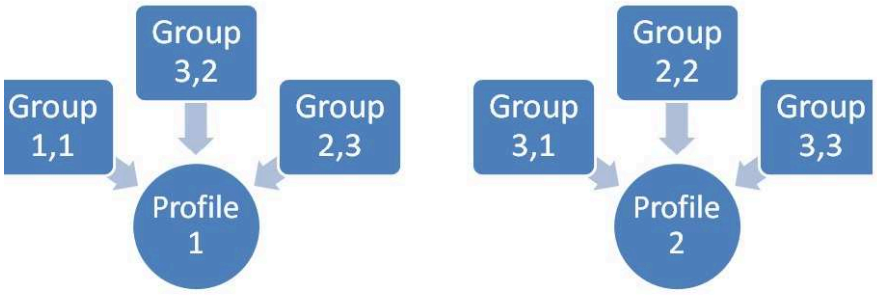


Figure 5.2 Profiles

5.2.2 Scénario

Lors du test des systèmes réactifs, nous avons besoin d'exprimer comment ils sont contraints par le contexte et l'activité de l'utilisateur. Une approche directe et explicite est de fournir au processus de test les activités typiques et significatives de l'utilisateur. Ces descriptions, souvent appelées "scénarios", raisonnent en se basant sur des situations d'utilisation.

Un scénario peut être considéré comme une histoire [Car00]. Par exemple pour le système d'éclairage, nous pouvons écrire cette suite d'évènements : au début tous les boutons du système sont au repos puis l'utilisateur appuie sur le premier bouton rapidement, ensuite il appuie sur le second et continue à appuyer jusqu'à ce que le mode d'éclairage permanent s'établisse, etc. Un scénario permet de prendre en compte des objectifs de test pendant la génération des données de test.

Il est composé de deux types d'éléments :

- Une étape ponctuelle est un ensemble de contraintes qui doit être satisfait dans un cycle de génération de données de test.
- Une étape étendue est un ensemble de contraintes qui doit rester vérifié jusqu'à ce qu'une condition (une expression booléenne spécifiée dans le scénario), soit satisfaite.

À la fin de chaque étape d'un scénario, nous pouvons ajouter une ou plusieurs expressions booléennes en guise d'oracle pour décider si le système fonctionne bien selon les spécifications indiquées. Un exemple simple est donné dans la section suivante.

5.3 Syntaxe du langage

5.3.1 La partie "Déclaration"

Pour écrire un programme SPTL, nous devons d'abord déclarer les entrées et les sorties du système sous test et éventuellement les initialiser. Cette partie de déclaration peut contenir en plus des variables une déclaration de constantes. Une déclaration d'une variable dans l'entête associe un type avec un identifiant. Chaque identifiant est unique. Les types de base qui sont considérés sont les entiers, le type de temps, les booléens, et les chaînes de caractère (pour afficher éventuellement des messages). Pour l'exemple de la figure 2.2 nous pouvons déceler les entrées suivantes :

- "Bouton1" et "Bouton2" représentent l'appui respectivement sur les deux boutons d'éclairage Bouton 1 et Bouton 2.
- "Pot" représente un potentiomètre pour contrôler la luminosité de l'éclairage.
- "CaptMvt" et "CaptCrep" pour indiquer respectivement l'état du capteur de mouvement et le capteur crépusculaire.

Pour les sorties nous avons :

- Lint pour indiquer l'état de l'éclairage interne ;
- Lext pour indiquer l'état de l'éclairage externe ;
- Pwd, une sortie Pwm (modulation de largeur d'impulsion) qui permettra en la reliant à un relais de commander la luminosité des lampes.

La figure 5.3 donne une idée sur le branchement des entrées et des sorties directement sur le contrôleur em4. Ainsi la partie déclaration des variables dans le code SPTL

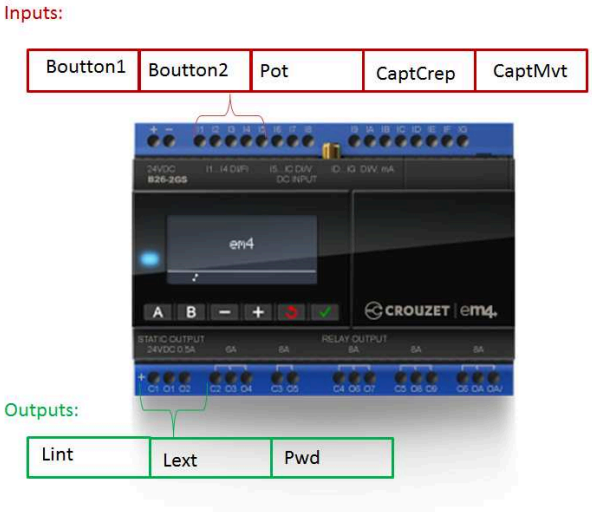


Figure 5.3 Les entrées et les sorties du programme d'éclairage

```
var
  input bool Bouton1
  input bool Bouton2
  input bool CaptCrep
  input bool CaptMvt
  input int Pot=5
  output bool Lint
  output bool Lext
  output int Pwd
```

Figure 5.4 Déclaration des variables en SPTL

sera de la forme donnée par la figure 5.3.1

Ici l'initialisation de `Pot` à 5 est liée à l'opérateur `pre` décrit ultérieurement.

5.3.2 Spécification des Catégories

La deuxième étape dans la spécification d'un programme SPTL est la définition des catégories et des groupes de contraintes en se basant sur la spécification du fonctionnement du système et les connaissances sur son environnement externe. À titre d'exemple, dans une "maison de retraite", les habitants appuient sur les boutons d'éclairage beaucoup moins souvent que dans un pensionnat pour les jeunes, chose qui peut être exprimée par l'utilisation de groupes différents. Comme défini en 5.2.1.1, un profil est généré par la combinaison de différents groupes de différentes catégories choisies par le testeur. Par exemple, parmi les catégories nous pouvons obtenir un profil composé de contraintes des groupes Pensionnat, Nuit et Ville :

```
categ HabitatType
{
    group Pensionnat{
        Bouton1 = prob(true,0.1) ;
        Bouton2 = prob(true,0.2)}

    group MaisonRetraite{
        Bouton1 = prob(true,0.03);
        Bouton2 = prob(true,0.03)}
}

categ Periode
{
    group Jour{ CaptCrep = false;Pot<6;Pot>0}
    group Nuit{ CaptCrep = true;Pot>4; Pot<10}
}
```

```
categ Place
{
    group Ville{CaptMvt = prob(true,0.3) }
    group Campagne{CaptMvt = prob(true,0.0001)}
}
```

Une contrainte implique une ou plusieurs variables et limite les valeurs que peuvent prendre simultanément ces variables. Ces relations reflètent l'environnement du système et le profil d'utilisation. Une "Expression" est définie en combinant les sous-expressions via des opérateurs arithmétiques, logiques et temporels (cf annexe A). Afin de guider la génération de données de test, l'opérateur "prob" définit les probabilités qui peuvent être associées avec toute expression. Par exemple, la contrainte CaptMvt = prob(true,0.3) signifie que la variable CaptMvt a la valeur "vrai" avec une probabilité de "0.3", sinon elle obtient une valeur par hasard avec une distribution uniforme.

5.3.3 Spécification des Scénarios

En plus des contraintes du profil choisi, SPTL propose la possibilité d'ajouter des scénarios et par conséquent prendre les objectifs de test en compte pendant la production de données de test. Comme nous l'avons évoqué à la section 5.2, un scénario est une succession d'étapes de deux types :

- L'étape ponctuelle, délimitée syntaxiquement par : "{" et "}", est un ensemble de contraintes à valider durant un cycle unique.
- L'étape étendue, encadrée par "[" et "]", est un ensemble de contraintes qui doit rester valable jusqu'à ce qu'une condition écrite entre "(" et ")", soit satisfaite. Cette étape peut être donc considérée sur plusieurs cycles.

Considérons le scénario simple du code ci-dessous toujours pour l'exemple de l'éclairage.

```
scenario Lightening
var
time tvariable1
time tvariable2
time tvariable3
begin
{ CaptMvt = true; Bouton1 = true;  tvariable1.start } |
[ Bouton1 = true  ( tvariable1 > 2 ) ] |
{ Bouton1 = false;  Bouton2 = true; Pot= 5;tvariable2.start} |
[ CaptMvt = false ; Bouton1 = false;
Bouton2 = true; Pot= 5 (tvariable2 > 1) ] |
{Bouton1 = false; Bouton2 = false;tvariable3.start} |
[  Bouton1 = true; Pot = pre(Pot) + 1 ( tvariable3 > 6 ) ]
end
```

Ce scénario utilise des variables de temps. Les valeurs des variables temps sont des nombres positifs pour indiquer un certain nombre de cycles. Un cycle dans le programme testé est le temps nécessaire pour une exécution complète. Cette dernière compte la lecture des entrée, le calcul des valeurs de sortie et la génération des sorties. `tvariable1.start` est une pseudo-contrainte ayant pour effet de déclencher le compteur du temps dans `tvariable1`. Dans l'exemple, `tvariable1` est utilisée dans la condition d'une étape étendue. La dernière étape du scénario utilise l'opérateur temporel `pre`. Cet opérateur est utilisé pour obtenir la valeur d'une variable (ou plus généralement une expression) au cycle précédent. Pour que cela ait un sens, `Pot` doit avoir une valeur avant le premier cycle. Une telle valeur initiale doit être spécifiée dans la partie "déclaration des variables".
Notons que chaque variable de sortie doit apparaître à l'intérieur de l'opérateur `pre` étant donné que dans le calcul des valeurs de test nous considérons les sorties du système générées dans le cycle précédent.

Revoyons ce scénario : la première étape initialise `CaptMvt` et `Bouton1` à `vrai` et déclenche le compte dans `tvariable1`. Dans la deuxième étape, `Bouton1` reste à `vrai` jusqu'à ce que `tvariable1` atteigne une valeur supérieure à 2 cycles. Le but de cette étape étendue est d'activer le mode d'éclairage permanent.
Puis, dans la deuxième étape ponctuelle, `Bouton1` est mise à `false`, `Bouton2` à true, `Pot` à 5 et le compte `tvariable2` est démarré. Dans l'étape suivante prolongée, les valeurs des variables `CaptMvt` et `Bouton1` restent à `faux`, `Bouton2` demeure `vrai` et le potentiomètre `Pot` reste à 5 jusqu'à ce que `tvariable2` atteigne une valeur supérieure à 1. L'objectif de cette étape consiste à tester le système sous le mode temporisé d'éclairage. La cinquième étape met `Bouton1` et `Bouton2` à `faux` et déclenche le compte dans la variable temps `tvariable3`.
Dans la dernière étape étendue, la valeur de la variable `Bouton1` reste à `vrai` et Pot augmente de 1 à chaque cycle jusqu'à ce que `tvariable3` atteigne une valeur supérieure à 6.

5.3.4 L'oracle

Une tâche très importante dans l'activité de test consiste à observer l'exécution du logiciel pour permettre de détecter les comportements du système relevant d'une défaillance. Parfois un observateur humain suffit pour accomplir cette fonction. Dans d'autres cas, il est nécessaire d'avoir recours à des moyens automatisés pour détecter les défaillances : des oracles automatisés. Un oracle peut être par exemple un programme extérieur qui a pour but de comparer les sorties du système aux sorties attendues selon les spécifications. Étant extérieur au système sous test et au générateur, ce programme peut être dans n'importe quel langage.
Une autre solution, qui nous a paru plus pratique pour le testeur dans notre cas, c'est d'intégrer l'oracle dans le langage SPTL. Nous proposons d'ajouter la possibilité de rajouter une liste d'expressions booléennes à toute étape du scénario. Cette liste est vérifiée à la fin de l'exécution de l'étape. Ces expressions représentent des propriétés que le système sous test devrait respecter dans un fonctionnement normal. Grâce

à l'opérateur `verdict`, nous pouvons écrire une spécification du comportement correct. Par exemple nous pouvons avoir un scénario simple en y rajoutant l'oracle écrit de la manière suivante :

```
scenario LighteningOracle
var
time tvariable1
begin
{ CaptMvt = true; Bouton1 = true;  tvariable1.start
:: verdict ( Lext= true; Lint = true ) } |
[ Bouton1 = true;  ( tvariable1 > 2 ):: verdict ( Lint = true ) ] |
{ Bouton1 = false;  Bouton2 = true:: verdict ( Lint = false )
```

5.4 Sémantique du langage

Dans cette section, nous donnons une description de la sémantique du langage SPTL. Tout en évitant d'être complètement formel, nous exposerons la sémantique d'un programme SPTL à partir de la sémantique de ses trois constructions de base : les expressions, les contraintes et les scénarios.

5.4.1 Expressions

La valeur d'une expression dépend des valeurs des variables du programme. Nous appelons *environnement* que nous notons σ le mappage des variables à leurs valeurs. Ainsi, $\sigma(x)$ est la valeur de la variable x dans l'environnement σ.

Notons $[\![e]\!]$ la fonction sémantique de l'expression e. La sémantique (dénotationnelle) de la plupart des expressions est plutôt simple comme le montre la figure 5.5. La sémantique d'une valeur constante est cette valeur même, quel que soit l'environnement ; la sémantique d'une variable est la valeur de cette variable dans l'envi-

ronnement considéré ; les opérateurs booléens et arithmétiques correspondent aux opérateurs mathématiques associés.

$$
\begin{aligned}
[\![cst]\!](\sigma) &= cst \\
[\![id]\!](\sigma) &= \sigma(id) \\
[\![e_1 + e_2]\!](\sigma) &= [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma) \\
[\![e_1 \texttt{ and } e_2]\!](\sigma) &= [\![e_1]\!](\sigma) \wedge [\![e_2]\!](\sigma) \\
&\ldots \\
[\![\texttt{prob}(v,p)]\!](\sigma) &= \begin{cases} v & \text{avec la probabilité } p \\ w & \text{avec la probabilité } 1-p, \quad w \neq v \end{cases} \\
[\![\texttt{pre}(e)]\!](\sigma) &= [\![e]\!](prev(\sigma)) \\
\multicolumn{2}{r}{\text{où } prev(\sigma) = \lambda \texttt{v}. \begin{cases} \sigma(\texttt{v'}) & \text{si v' existe} \\ \bot & \text{sinon} \end{cases}}
\end{aligned}
$$

Figure 5.5 Sémantique des expressions

L'opérateur probabiliste introduit le non déterminisme : l'expression `prob(`v`,`p`)` a la valeur v avec la probabilité p, ou une valeur w choisie au hasard parmi les autres valeurs possibles de ce type.

L'opérateur temporel `pre` permet de se référer à la valeur d'une variable (ou plus généralement une expression) dans le cycle précédent. Pour le définir, nous avons étendu l'environnement pour inclure les *variables mémoire* : une telle variable `v'` a toujours la valeur qu'a la variable `v` au cycle précédent.

Comme indiqué dans la section précédente, les valeurs initiales doivent être fournies pour les variables apparaissant dans un `pre` de sorte que les variables mémoire soient initialisées. Nous montrerons par la suite comment ces variables sont mises à jour.

L'ensemble des variables mémoire est fini et peut être construit à partir d'une analyse statique du programme. Commençons avec un ensemble σ_m vide. À chaque occurrence de l'opérateur `pre` dans le programme, nous considérons l'expression dans son

argument `e` : pour chaque variable `v` survenant dans `e`, nous ajoutons `v'` à σ_m si elle ne la contient pas déjà. Les expressions `pre` imbriquées peuvent également être facilement traitées par l'ajout de variables mémoire d'un second ordre (exemple : `v''` pour la variable `v'`).
Nous définissons *prev* la fonction de correspondance d'un environnement σ_1 à un environnement σ_2 où chaque variable `v` a la valeur de la variable mémoire correspondante `v'` dans σ_1 si elle existe. La sémantique de `pre(`e`)` dans l'environnement σ est la sémantique de e dans l'environnement $prev(\sigma)$.

5.4.2 Contraintes

Pour traiter les contraintes, nous devons d'abord distinguer les différents types de variables qui existent dans un programme SPTL :

- *les variables de sortie* ont leurs valeurs définies par le système sous test,
- *les variables d'entrée* doivent être calculées par le programme SPTL,
- *les variables mémoire* gardent les valeurs des variables d'entrée ou de sortie du cycle précédent.

Étant donné que les variables de sortie ne peuvent apparaître que dans un opérateur `pre`, une contrainte ne peut contenir que des occurrences de variables d'entrée et de variables mémoire[1]. Elle a la forme d'une expression booléenne, mais sa sémantique est très différente : une contrainte définit les valeurs possibles de ses variables d'entrée, selon les valeurs de ses variables mémoire.

Partitionnons l'environnement σ en un environnement de variables mémoire σ_m et un environnement de variables d'entrée σ_i. Nous notons $\sigma = \sigma_m \oplus \sigma_i$. Une contrainte définit les valeurs possibles de σ_i à partir d'une valeur donnée de σ_m.

[1] Bien sûr, les variables mémoire n'apparaissent pas explicitement dans le programme. Nous appelons ici variable mémoire tout occurrence d'une variable d'entrée ou de sortie dans une expression `pre`.

$$\begin{aligned} \langle\!\langle c \rangle\!\rangle(\sigma_m) &= \{\sigma_i, [\![c]\!](\sigma_m \oplus \sigma_i) = true\} \\ \langle\!\langle c_1; ...; c_n \rangle\!\rangle(\sigma_m) &= \langle\!\langle c_1 \rangle\!\rangle(\sigma_m) \cap ... \cap \langle\!\langle c_n \rangle\!\rangle(\sigma_m) \end{aligned}$$

Figure 5.6 Sémantique des contraintes

Notons par $\langle\!\langle c \rangle\!\rangle$ la fonction sémantique de la contrainte c. La figure 5.6 montre que les σ_i possibles sont ceux qui, lorsqu'ils sont associés avec σ_m, forment un environnement complet σ dans lequel l'expression booléenne a la valeur "vrai". La figure montre aussi la règle pour la composition de la contrainte : l'ensemble des valeurs possibles pour les variables d'entrée est l'intersection des ensembles de valeurs possibles pour chaque contrainte.

5.4.3 Scénarios

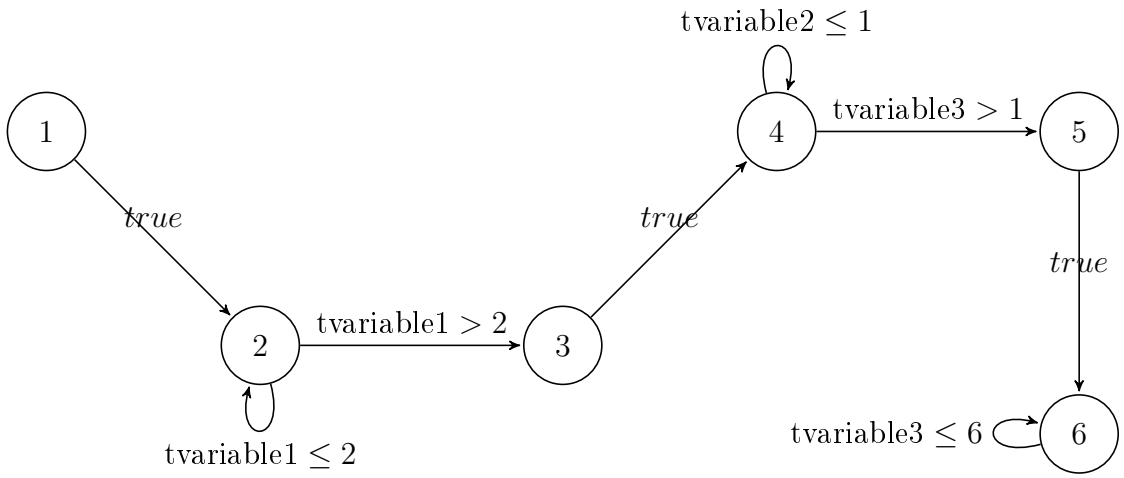
Rappelons qu'un scénario est une séquence de :

- *étapes ponctuelles* : elles définissent des contraintes applicables en un cycle ;
- *étapes étendues* : elles définissent des contraintes applicables jusqu'à ce qu'une certaine *condition de sortie* soit vérifiée.

Nous pouvons décrire un scénario par un automate où chaque étape du scénario est un état :

- L'état d'une étape ponctuelle a une seule transition vers l'état suivant marquée à *true*.
- L'état d'une étape étendue a une transition vers l'autre état, marquée par sa condition de sortie, et une transition vers lui-même, marquée par la négation de sa condition de sortie.

Le scénario décrit dans la section 5.3.3 serait ainsi représenté par l'automate suivant :



Ce scénario montre l'utilisation des *variables temps* (variables de type `time`). Les valeurs de ce type de variables sont calculées en nombre de cycles `c`.

5.4.4 Programme

Rappelons qu'un ensemble de contraintes, le profil, a été produit à partir d'une combinaison choisie de groupes de différentes catégories. Le programme dont nous voulons définir la sémantique est ainsi composé de :

- déclarations de variables,
- contraintes du profil,
- un scénario.

La figure 5.8 montre une description informelle de la sémantique opérationnelle du programme. L'idée de base est qu'à chaque cycle un ensemble courant de contraintes *cset* est en vigueur (étape 1) : il est la combinaison des contraintes du profil et de l'ensemble des contraintes de l'étape actuellement active dans le scénario (`curscen`). Pour résoudre *cset* dans l'environnement σ_m, les valeurs possibles pour les variables d'entrée sont calculées. Une de ces valeurs σ_i est sélectionnée de façon aléatoire (2) et injectées aux entrées du système sous test (3).

Quand les sorties du système sont lues (4), l'étape active du scénario est modifiée (5) si sa condition de sortie est *true* (évaluée dans l'environnement $\sigma = \sigma_m \oplus \sigma_i$). [2] L'évaluation du verdict se fait à chaque cycle (6). La sémantique de l'oracle peut être définie comme le montre la figure 5.7 : La valeur logique de l'oracle est calculée donc

$$[\![verdict(e_1; ..; e_n)]\!](\sigma) \quad = \quad [\![e_1]\!](\sigma) \wedge .. \wedge [\![e_n]\!](\sigma)$$

Figure 5.7 Sémantique de l'oracle

par la conjonction $\wedge$ entre les différentes valeurs des expressions constituant l'oracle.

Les variables de mémoire sont mises à jour (7). Le test se termine si le scénario se termine, sinon le prochain cycle commence (8).

Initialement, `curscen` est mis à 0, la première étape du scénario. Les valeurs initiales des variables mémoire sont données avec la déclaration des variables (0).
Pour plus de clarté, nous n'avons pas inclus dans la figure le traitement des variables de temps. Une variable temps compte un certain nombre de cycles exécutés, ainsi elle possède une valeur entière initialisée à 0. Nous expliquerons dans le chapitre suivant comment ce type de variables est géré et comment résoudre la pseudo-contrainte `t.start` en Prolog.

5.5 Conclusion du chapitre

Dans ce chapitre, nous avons proposé un langage pour générer des séquences de test pour les systèmes réactifs synchrones. Il est basé sur la description à la fois des contraintes de l'environnement et des profils des utilisateurs du programme sous test. L'un des principaux objectifs de sa conception est de permettre à des non-experts à faire des tests réalistes. Cela nous a conduit à concevoir un langage dont

[2] Une étape ponctuelle peut être définie comme une étape étendue avec une condition de sortie marquée par *true*

(0) `curscen` $\leftarrow$ `0`
initialiser σ_m

1. `cset` $\leftarrow$ profil $\cup$ `contraintes[curscen]`
2. calculer σ_i en résolvant `cset` (utilisant σ_m)
3. appliquer σ_i aux entrées du système
4. lire les sorties du système en σ_o
5. évaluer la condition de sortie de l'étape du scénario, incrémenter `curscen` si *true*
6. calcul du verdict par l'oracle
7. mettre à jour σ_m (de σ_i et σ_o)
8. si fin de scénario alors sortie sinon aller à (1)

Figure 5.8 Exécution d'un cycle d'un programme SPTL

les structures de base sont très proches de celles du langage utilisé pour programmer les contrôleurs de InnoVista Sensors qui serviront à expérimenter notre outil. En outre, les notions de catégories d'utilisation, des groupes et des profils permettront de réduire le travail nécessaire pour préparer les tests. Certains profils standards peuvent être prédéfinis, par les ingénieurs du fabricant ou par un tiers, afin d'aider l'utilisateur dans sa conception des test. Exprimer des scénarios de test permet aussi de prendre en compte les objectifs de test lors de la génération. Dans le chapitre suivant, nous allons nous concentrer sur la seconde partie du prototype de générateur de données de test, à savoir la traduction de SPTL vers Prolog et les algorithmes de génération des entrées de test.

Chapitre 6

Testium : le composant Prolog

6.1 Introduction du chapitre

La deuxième étape importante après la définition de la syntaxe et la sémantique de SPTL, est la compilation. Dans cette étape, nous vérifions la justesse du code écrit et nous le traduisons en un autre langage qui nous permettra de résoudre les contraintes décrites dans le programme source. Nous avons choisi le langage de programmation logique Prolog comme langage cible. Nous avons conçu un compilateur pour assurer la traduction SPTL-Prolog. Le moteur Prolog utilise un algorithme de sélection qui choisit un vecteur d'entrée valide, selon la description de l'environnement. Dans ce chapitre, nous décrivons les différentes parties de cette traduction, présentons les principaux algorithmes de résolution et expliquons le fonctionnement de l'outil réalisé à travers l'exemple présenté au chapitre 2.

6.2 Compilation d'un programme SPTL

SPTL permet d'exprimer les contraintes de l'environnement du système sous test d'une manière purement déclarative. Pour résoudre ces contraintes, nous avons pensé aux langages de programmation logique par contraintes et plus précisément Prolog, pour profiter des avantages qu'ils proposent et que nous avons évoquées au chapitre 3, à savoir : les solveurs, l'exécution rapide, la facilité d'interaction avec d'autres langages etc.

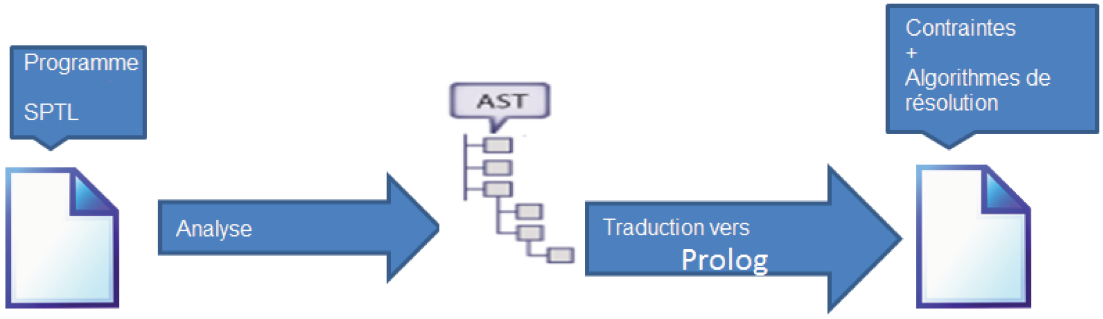


Figure 6.1 Traduction de SPTL vers Prolog

Une partie importante de tout compilateur est la détection et le rapport des erreurs. Comme représenté par la figure 6.1, l'explorateur analyse le code du programme source pour détecter d'éventuelles erreurs. Nous avons réalisé un compilateur nommé "SPTL Grammar Explorer" pour vérifier la justesse du programme SPTL.
Une fois le code validé, le compilateur produit un arbre abstrait correspondant au programme SPTL. En exploitant cet arbre et en parcourant ses nœuds "Testium" récupère les éléments nécessaires pour écrire les contraintes correspondantes en Prolog. Le moteur Prolog peut ainsi exécuter le programme cible et générer des solutions.

6.3 Composants du programme Prolog cible

6.3.1 Algorithme principal de génération de test

Le prédicat étant l'unité de base de Prolog, l'idée est de traduire les contraintes du profil et du scénario en des prédicats. Le processus de test se base sur une boucle d'échanges entre le générateur de données de test et le SST. L'algorithme principal se basera aussi par conséquent sur une boucle. Principalement, il effectue un appel récursif au prédicat `path`. Chaque appel correspond à un cycle de test qui représente un couple d'action-réaction. Le générateur produit un vecteur d'entrée et envoie au système sous test ; ce dernier réagit avec un vecteur de sortie envoyé vers le générateur. Le vecteur de sortie est utilisé pour calculer les nouvelles entrées pour le cycle suivant, et ainsi de suite. Au cours de chaque cycle, les contraintes invariantes et les contraintes associées à l'étape courante du scénario doivent être résolues pour calculer un vecteur d'entrée de test pour le SST.
L'algorithme 1 écrit en `SWI Prolog` génère des données de test pour `P` cycles d'exécution. Le processus commence avec le prédicat `read` qui permet de lire le nombre de cycles P entré par le testeur. Ensuite, le prédicat `init` effectue les initialisations.
À chaque appel à `path` nous avons les étapes suivantes :

- récupération des valeurs des variables mémoire (getMemory)
- ajout des contraintes des invariants du profil (environment)
- ajout des contraintes avec les probabilités qui traduisent les expressions utilisant l'opérateur prob (probability)
- ajout des contraintes de l'étape courante du scénario (scenario)
- choix aléatoire des données à envoyer au SST (choose)
- mise à jour des variables mémoire (setMemory)
- envoi de la solution choisie parmi les solutions possibles (writeln)

Algorithm 1 Algorithme pour la génération d'une séquence de test en SWI Prolog

Data :{ P(Index du cycle courant), counter(Index de l'étape du scénario), M(Liste des variables mémoire), T(Liste des variables temps), I(Liste des entrées), O(Liste des sorties) }

```
main:-read(P),init(I,O),path(P).
path(0).
path(P):- P>0,
        getMemory(M),
        nb_getval(counter, CounterValue),
        environment(I,O,M),
        probability(I),
        scenario(CounterValue,I,O,M,T,P),
        choose(I),
        setMemory(I),
        writeln(I),
        read(O),
        verdict(CounterValue,I,O),
        P1 is P-1 ,
        path(P1).
```

– lecture des sorties (read)

– calcul du verdict de l'oracle (verdict)

Il y a bien d'autres prédicats utilisés que ceux apparus dans cet algorithme. Certains servent à spécifier les entrées et les sorties et leurs types. D'autres gèrent les variables de temps etc. Dans les sections suivantes nous allons présenter plus de détails sur les différents prédicats mis en œuvre pour la construction du générateur de test.

6.3.2 Contraintes de type

Dans SPTL, chaque variable d'entrée (représentant une entrée du système) a un type donné. Dans un sens, le type est une contrainte implicite appliquée à la valeur de la variable. Comme déjà dit, en Prolog, nous pouvons exprimer cette contrainte par un prédicat. Par exemple, les types int et bool sont représentés par les prédicats suivants :

```
type_int(X): - X in -32768 32767 ...
type_bool(X): - X in 0..1.
```

Le prédicat correspondant aux entrées spécifie le vecteur d'entrée comme un ensemble de variables d'entrée du système synchrone et il les force à prendre que les valeurs autorisées par leurs types. De même, le prédicat des sorties spécifie les variables de sortie du système sous test.

```
Inputs(I): -
   I = [Bouton1, Pot, Bouton2, CaptCrep, CaptMvt],
   type_bool(Bouton1), type_int(Pot),
   type_bool(Bouton2),
   type_bool(CaptCrep), type_bool(CaptMvt).
Outputs(O): -
   O = [Lint, Lext, Pwd],
   type_int(Pwd),
   type_bool(Lint), type_bool(Lext).
```

6.3.3 Initialisation

Le prédicat `init` définit les variables d'entrée et de sortie, assigne des valeurs initiales selon les déclarations SPTL, met à jour les variables de mémoire, et enfin initialise les variables de temps.

```
init(I, O): - Inputs(I), Outputs(O),
              I = [Bouton1, Pot, Bouton2, CaptCrep, CaptMvt],
              O = [Lint, Lext, Pwd],
              Pot = 5, setMemory (I, O),
              nb_setval (counter, 0),
              nb_setval (tvariable1, 0),
              nb_setval (tvariable2, 0),
              nb_setval (tvariable3, 0).
```

6.3.4 Gestion des variables mémoire

L'opérateur `pre` est un opérateur très important du langage SPTL. Il permet d'utiliser la valeur qu'avait une expression dans le cycle précédent. En `Prolog`, nous mettons en œuvre ceci avec une liste `M` de variables mémoire. Nous créons des variables globales en utilisant les prédicats `nb_setval(nom,valeur)` et `nb_getval(nom,valeur)`. Une variable globale est définie pour chaque expression `pre` figurant dans le code SPTL. Les variables mémoire sont affectées (pour une utilisation dans le cycle suivant) avec le prédicat `setMemory` et lues avec le prédicat `getMemory`. Pour exemple, dans SPTL nous avons : `Pot = pre(Pot) + 1`. En Prolog, nous définissons la variable globale `preExp1` que nous enregistrons dans la liste `M` avec la clé `PreExp1`. Nous utilisons aussi les "variables anonymes" de Prolog pour stocker certains calculs intermédiaires. Ce sont des variables commençant par un trait de soulignement comme par exemple _preExp1. Contrairement à d'autres variables, elles ne représentent pas la même valeur partout où elles apparaissent.

```
setMemory (I, O): - I = [Bouton1, Pot, Bouton2, CaptCrep, CaptMvt],
                    O = [Lint, Lext, Pwd], _preExp1 is Pot1,
                    nb_setval (preExp1, _preExp1).
                    getMemory (M): - M = [PreExp1],
                    nb_getval (preExp1, PreExp1).
```

6.3.5 Profils

Le prédicat `environnement` représente les propriétés invariantes des entrées du système. Pour notre exemple, nous avons :

```
environment(I,O,M) :- inputs(I),
   outputs(O), M = [PreExp1],
   I = [Bouton1,Pot,Bouton2, CaptCrep, CaptMvt],
   O = [Lint, Lext, Pwd],
   Pot #> 5, Pot #< 10, CaptCrep is 1.
```

Prolog propose plusieurs opérateurs logiques et arithmétiques comme "supérieur" ou "infénieur" utilisés dans les faits ci-dessus. Les contraintes exprimées avec des probabilités utilisant l'opérateur `prob` sont traduites en Prolog avec le prédicat `probability`. Ce dernier appelle pour chaque contrainte le prédicat `prob` comme suit :

```
ok(P):-random(X),!,X<P.
prob(P,Y,Z):-(ok(P) -> Y =Z ; !).
probability(I):-inputs(I),
                I = [Bouton1,Pot,Bouton2, CaptCrep, CaptMvt],
                prob(0.1,Bouton1,1),
                prob(0.2,Bouton2,1), prob(0.3,CaptMvt,1).
```

Afin d'introduire la probabilité, nous utilisons le prédicat `ok(P)` qui ne réussit qu'avec une probabilité P en utilisant le prédicat `random` défini dans une biblio-

thèque `random.pl` fournie par `SWI Prolog`. `random(X)` retourne un nombre au hasard différent dans l'intervalle [0, 1) à chaque appel. Si ok(P) réussit, la variable Y sera unifiée avec la variable Z : Bouton1 par exemple aura la valeur 1 avec la probabilité 0,1.

6.3.6 Scenarios

Une autre partie importante de l'algorithme de génération de test est associée à des scénarios. Chaque étape dans un scénario sera représentée en Prolog par le prédicat `scenario`. Par example, pour le scénario "Lightening" donné au chapitre précédent :

- L'étape numéro 0 est une étape ponctuelle :

```
scenario(0, I, O, M, T, P): -
        I = [Bouton1, Pot1, Bouton2, CaptCrep, CaptMvt]
        O = [Lint, Lext, Tempo, permanent],
        M = [PreExp1],
        T = [Tvariable1, Tvariable2, Tvariable3],
        CaptMvt is  1, Bouton1 is  1,
        nb_setval(tvariable1,P),
        getTimes(T,P), nextStep.
```

- l'étape numéro 1 est une étape étendue :

```
scenario(1, I, O, M, T, P): -
        I = [Bouton1, Pot1, Bouton2, CaptCrep, CaptMvt]
        O = [Lint, Lext, Tempo, permanent],
        M = [PreExp1],
        T = [Tvariable1, Tvariable2, Tvariable3],
        Bouton1 is  1, getTimes(T,P),
```

```
(Tvariable1 #< 2 -> !; nextStep).
```

La variable globale `counter` contient le numéro de l'étape courante du scénario (commençant par 0). Le prédicat `nextStep` est utilisé pour passer à l'étape suivante du scénario, simplement en incrémentant ce compteur. Dans une étape étendue, `nextStep` n'est appelée que lorsque la condition de fin d'étape est vraie.

```
nextStep :- nb_getval(counter, C),
   CNew is C + 1,nb_setval(counter, CNew).
```

Dans l'algorithme principal, la valeur du compteur est récupérée avant l'appel des prédicats `environment` et `scenario`.
Si le scénario contient des variables de temps, une liste `T` de ces variables sera créée. La valeur d'une variable de temps va commencer à augmenter seulement si elle est activée par l'opération `.start` associée, comme un déclenchement d'un chronomètre. Toutes les valeurs des variables de temps sont mises à jour grâce au prédicat `getTimes` dont le code est le suivant :

```
getTimes(T,P) :-
  T=[Tvariable1,Tvariable2,Tvariable3],
  nb_getval(tvariable1,C1),
  nb_getval(tvariable2,C2),
  nb_getval(tvariable3,C3),
  Tvariable1 is C1- P + 1,
  Tvariable2 is C2- P + 1,
  Tvariable3 is C3- P + 1.
```

Par exemple, à l'étape 0 du scénario donné précédemment, l'opération `.start` associée à Tvariable1 est traduite par l'appel au prédicat `nb_setval(tvariable1,P)`, qui donnera par conséquence la valeur de 1 à `Tvariable1` à l'appel du prédicat `getTimes`. `Tvaribale1` va s'incrémenter de 1 à chaque cycle.

Pour chaque étape du scénario, nous pouvons rajouter un oracle pour déterminer si le système sous test réagit correctement aux données de test ou s'il y a une défaillance. Côté Prolog, cet oracle est traduit par le prédicat `verdict`. Par exemple, si nous avons rajouté un oracle à l'étape 1 du scénario, nous aurions en prolog le code suivant :

```
verdict(1, I, O): - I = [Bouton1, Pot1, Bouton2, CaptCrep, CaptMvt]
                    O = [Lint, Lext, Tempo, permanent],
                   (Lint is 1 -> writeln(1) ; writeln(0)).
```

Le résultat de l'oracle est donc soit une réussite(1) soit un échec (0).

6.3.7 Sélection des données de test

L'étape de la sélection des données de test consiste à sélectionner au hasard, à chaque cycle, un vecteur d'entrée système parmi les solutions des contraintes. Pour cela, nous utilisons les prédicats suivants :

```
choose(X,L):- fd_inf(X,DMIN), fd_sup(X,DMAX), random(L,DMIN,DMAX).

random(L,DMIN,DMAX):- B is DMAX - DMIN+1,
                      A is random(B), L is A + DMIN.
```

Le prédicat `choose` fixera X à une valeur choisie au hasard dans la liste `L` des valeurs. Prolog propose des prédicats prédéfinis qui traitent les domaines de variables. Par exemple avec `SWI Prolog` nous trouvons les prédicats : `fd_inf` et `fd_sup` dans la bibliothèque `clpfd`. `fd_inf` trouve `DMIN` la borne inférieure du domaine en cours de la variable X tandis que `fd_sup` trouve la borne maximale. La méthode de sélection consiste donc à affecter une valeur dans ce domaine réduit, généralement d'une manière aléatoire et équiprobable grâce au prédicat `random`.

6.4 Conclusion du chapitre

Nous avons proposé une méthode et un outil pour générer automatiquement des séquences de test pour les contrôleurs synchrones basés sur l'utilisation du langage SPTL pour décrire à la fois des propriétés invariantes et l'évolution dynamique de l'environnement. Les contraintes écrites en SPTL et expliquées dans le chapitre précédent, doivent être résolues et pour ce faire, nous recourons à la compilation du programme SPTL vers un langage de programmation logique qui est Prolog dans notre cas. Dans ce chapitre, nous avons exposé les différents prédicats cibles ainsi que l'algorithme principal pour la génération des données de test. Pour valider le prototype, nous avons expérimenté "Testium" sur un exemple de programme utilisé par les clients de InnoVista Sensors que cette dernière nous a fourni. Dans le chapitre suivant, nous allons exposer ce cas, le tester et analyser les résultats.

Chapitre 7

Implémentation et Expérimentation

7.1 Introduction du chapitre

Dans les chapitres précédents, nous avons présenté les différentes étapes de la génération de test avec "Testium", en se basant sur un exemple simple d'un contrôleur d'éclairage d'une maison. Ce chapitre porte sur l'évaluation de cette méthode sur une étude de cas plus réaliste et utilisé actuellement par les clients de InnoVista Sensors : un système d'irrigation. Une implémentation en FBD de la spécification de cette application a été fournie par InnoVista Sensors. Cet exemple caractérise bien les logiciels de contrôle-commande synchrones, qui sont la cible principale de l'outil "Testium". Les objectifs que nous nous posons pour cette étude sont multiples, elle vise à voir dans quelle mesure :

- les méthodes de génération proposées sont applicables à l'échelle réaliste ;
- écrire des spécifications de test et des scénarios en SPTL est facile.

Dans ce chapitre nous allons commencer par présenter l'environnement technique de la réalisation de l'outil "Testium". Par la suite, nous allons étudier l'exemple de l'irrigation partant de la modélisation SPTL, passant par la traduction vers Prolog et enfin arrivant à l'exécution de quelques scénarios de test et l'analyse des résultats.

7.2 Environnement technique du développement de l'outil Testium

L'outil "Testium" a été réalisé sur la plateforme .net V4.5 avec Microsoft Visual C# 2013. Pour la partie compilation du programme, nous avons mis en œuvre le kit de développement "Irony". Et pour la programmation logique par contraintes, nous avons eu recours au moteur SWI Prolog.

7.2.1 Irony

Irony[1]est un kit de développement pour l'implémentation de langages sur la plateforme .NET. Contrairement à la plupart des solutions existantes comme yacc /lex, **Irony** n'utilise pas d'analyseur qui génère du code à partir des spécifications de la grammaire écrite dans un méta-langage spécialisé. Dans Irony, la grammaire du langage est codée directement en C# en utilisant la surcharge d'opérateurs pour exprimer des constructions grammaticales. Les modules de l'analyseur d'Irony utilisent la grammaire codée comme une classe C# pour contrôler le processus d'analyse.
Pour créer une nouvelle grammaire, il suffit de créer un nouveau fichier de classe et d'étendre la classe de grammaire de base trouvée dans le kit d'Irony. Nous avons ainsi défini la grammaire de SPTL (cf. annexe A) sous forme de classes en C#.

7.2.2 Swi Prolog

La version de Prolog que nous avons utilisée est SWI-Prolog, développée à l'Institut Suédois de l'Informatique. La principale raison de ce choix réside dans la richesse des bibliothèques de Swi-Prolog. Une autre raison majeure en est que l'environnement Swi-Prolog est librement disponible, un critère qui permet une distribution plus fa-

[1] `http://blogs.msdn.com/b/kirillosenkov/archive/2009/10/31/irony.aspx`

cile de l'outil. L'environnement SWI-Prolog est un système interactif. Il possède un débuggeur graphique ainsi que plusieurs solveurs de contraintes. Notons que la bibliothèque (clpfd) : "Constraint Logic Programming over Finite Domains" [Tri12] de l'environnement SWI Prolog, que nous avons utilisée pour la réalisation du prototype "Testium", est un solveur qui résout les problèmes qui impliquent des ensembles de variables, où les relations entre les variables doivent être satisfaites. C'est un formalisme déclaratif pour décrire des problèmes combinatoires comme l'ordonnancement, la planification et les tâches d'allocation [JL87]. Cette bibliothèque est tout à fait suffisante pour les types de contraintes que nous pouvons avoir en SPTL. En effet dans notre contexte de travail, nous n'avons pas besoin d'écrire des contraintes avec des expressions mathématiques complexes.
Dans notre projet nous n'avons pas utilisé l'interface graphique de SWI Prolog, mais plutôt son moteur exécutable dont le nom est souvent "swipl". Cet exécutable permet de compiler le programme Prolog que nous avons généré. Ainsi notre programme "Testium", communiquera avec cet exécutable en redirigeant les entrées/sorties de ce dernier.

7.2.3 Communication avec l'EM4 soft

Après la compilation du programme SPTL et le choix du profil d'utilisation à tester, le testeur a le choix de lancer le test en mode "Step by step" c'est à dire cycle par cycle ou exécuter toutes les étapes du scénario de test en une seule fois en mode "automatic". Il sélectionne le programme à tester ainsi que le nombre de cycles à exécuter et lance l'exécution. La figure 7.1 représente l'interface de "Testium" permettant le lancement de la génération de test.

Pour communiquer avec l'atelier "Em4 Soft", nous avons eu recours à un exécutable "SimuEm4Soft.exe" qui permet d'ouvrir l'atelier de programmation en mode simulation et d'injecter nos données de test et de récupérer par la suite les sorties du programme comme le montre la figure 7.2.

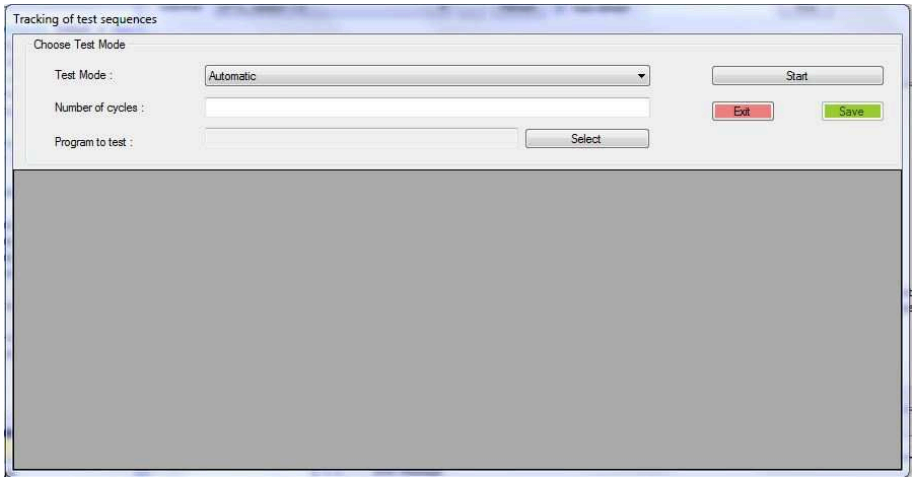


FIGURE 7.1 Interface pour le lancement de la génération et d'exécution des tests

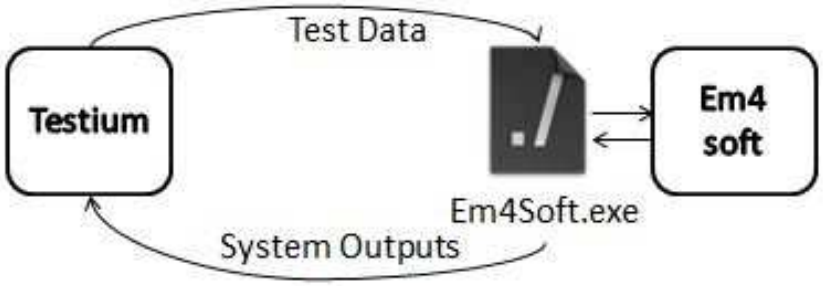


FIGURE 7.2 Communication avec l'atelier EM4 Soft

"SimuEm4Soft" lance l'atelier dans le premier appel en ouvrant le programme sélectionné. Par la suite, il injecte nos données de test au programme en cours d'exécution et récupère les sorties que "Testium" récupère à son tour pour le calcul des données de test pour le cycle suivant et ainsi de suite jusqu'à ce que le scénario atteigne sa fin.

Nous pouvons voir le changement des valeurs des entrées et des sorties sur le simulateur en cours d'exécution du test comme le montre la figure 7.3 et elles sont affichées au fur et à mesure dans la grille de données dans "Testium". La figure 7.4 montre un exemple d'affichage de résultats de test.

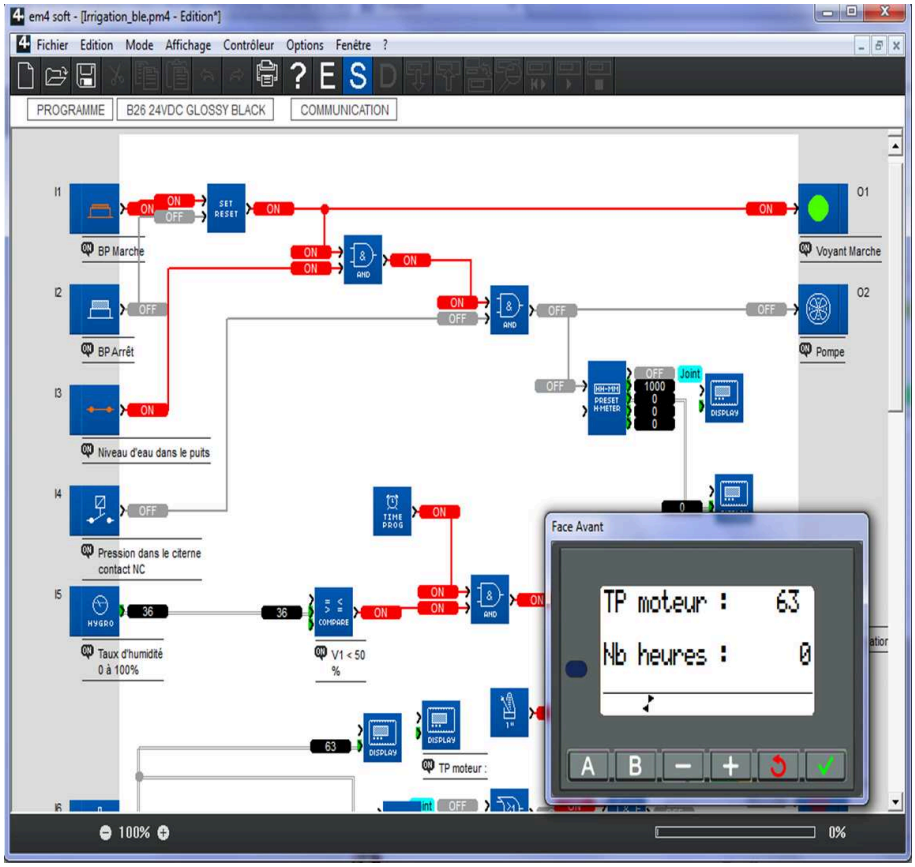
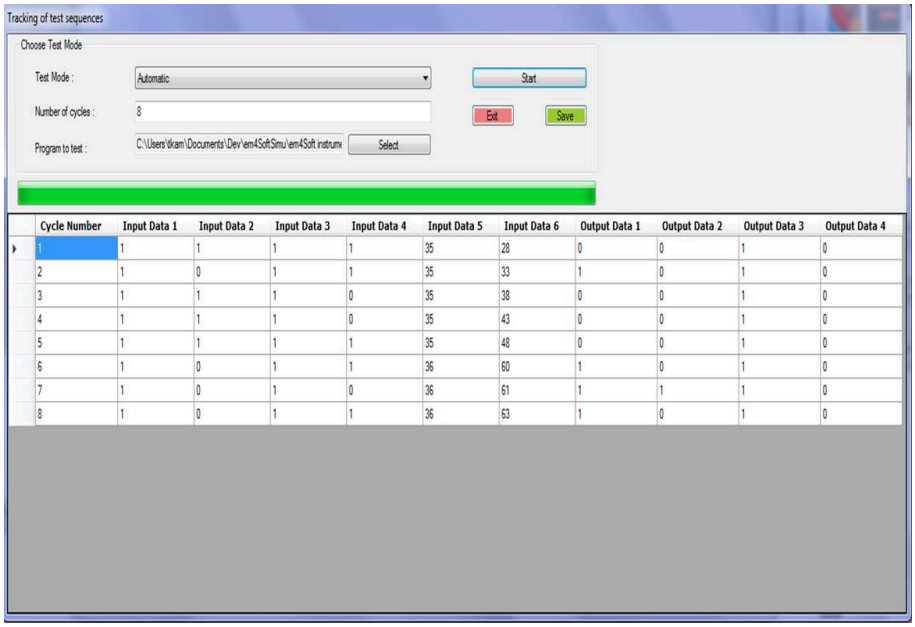


Figure 7.3 Simulation d'exécution

Le générateur produit des données de test jusqu'à ce que le nombre de cycles soit atteint, le scénario exécuté ait atteint sa fin ou qu'il n'y ait plus de solutions aux contraintes imposées. Nous pouvons enregistrer ces données, à la fin de l'exécution, sous un format .CSV ou autre pour pouvoir exploiter les résultats ultérieurement.

7.3 Présentation de l'exemple à tester

Les séries MILLENIUM et EM4 ont été conçues pour être employées dans des domaines domestiques, médicaux et industriels. Chaque module permet de gérer l'ensemble des capteurs et actionneurs de l'installation. Nous avons choisi un programme fourni par InnoVista Sensors et utilisé par un de ses clients : un système d'irrigation.



Cycle Number	Input Data 1	Input Data 2	Input Data 3	Input Data 4	Input Data 5	Input Data 6	Output Data 1	Output Data 2	Output Data 3	Output Data 4
1	1	1	1	1	35	28	0	0	1	0
2	1	0	1	1	35	33	1	0	1	0
3	1	1	1	0	35	38	0	0	1	0
4	1	1	1	0	35	43	0	0	1	0
5	1	1	1	1	35	48	0	0	1	0
6	1	0	1	1	36	60	1	0	1	0
7	1	0	1	0	36	61	1	1	1	0
8	1	0	1	1	36	63	1	0	1	0

Figure 7.4 Affichage des résultats

7.3.1 Description du système à tester

7.3.1.1 Composants

Comme le montre la figure 7.5 :

- L'eau est fournie par un puits d'eau.
- Une citerne d'eau reliée à une pompe.
- Le niveau de l'eau dans le puits et la pression de l'eau dans la citerne sont surveillés par des capteurs.
- Une vanne d'irrigation.
- Le contrôleur s'occupe de l'irrigation basé sur des horaires quotidiens prédéfinis.
- Une alimentation électrique pour alimenter le contrôleur.
- Un capteur de température pour le moteur.

- Une sonde d'humidité.
- Des boutons marche/arrêt, reliés à une bascule RS à "RESET" prioritaire : si les deux boutons sont appuyés, c'est le bouton arrêt qui l'emporte.
- Un voyant lumineux qui s'allume quand le système est en marche.

Le bon fonctionnement du système repose sur l'hypothèse du synchronisme. En particulier, nous admettons que tous les signaux en provenance des capteurs ou du système sont transmis d'une manière instantanée. La durée du cycle pour ce système est de 0.01 secondes.

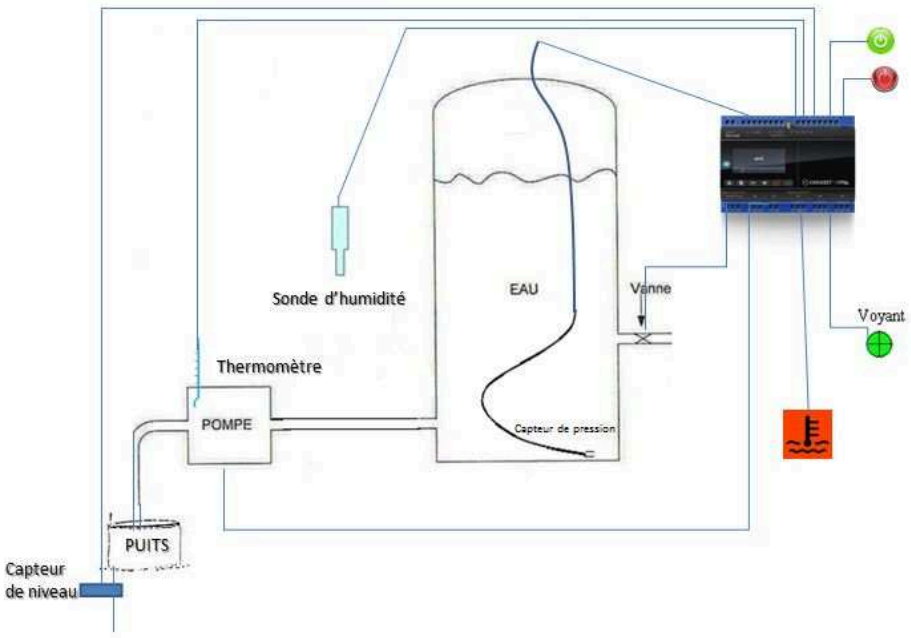


Figure 7.5 Schéma fonctionnel du système d'irrigation

7.3.1.2 Fonctionnement

Le système d'irrigation, généralement de champs de mais ou de blé, a pour objectifs les points qui suivent :

1. Remplir la citerne

- Surveiller le niveau d'eau dans le puits,
- Contrôler la pression d'eau dans la citerne.

2. Irrigation : ouvrir la vanne à une heure précise et ce pendant un certain temps à condition que le taux d'humidité du sol soit $<$ à 50

3. Maintenance préventive

   - Surveiller la température du moteur, alarme si t°$>=$ à 90°C et afficher la température
   - Maintenance du moteur : joint à changer toutes les 1000 heures

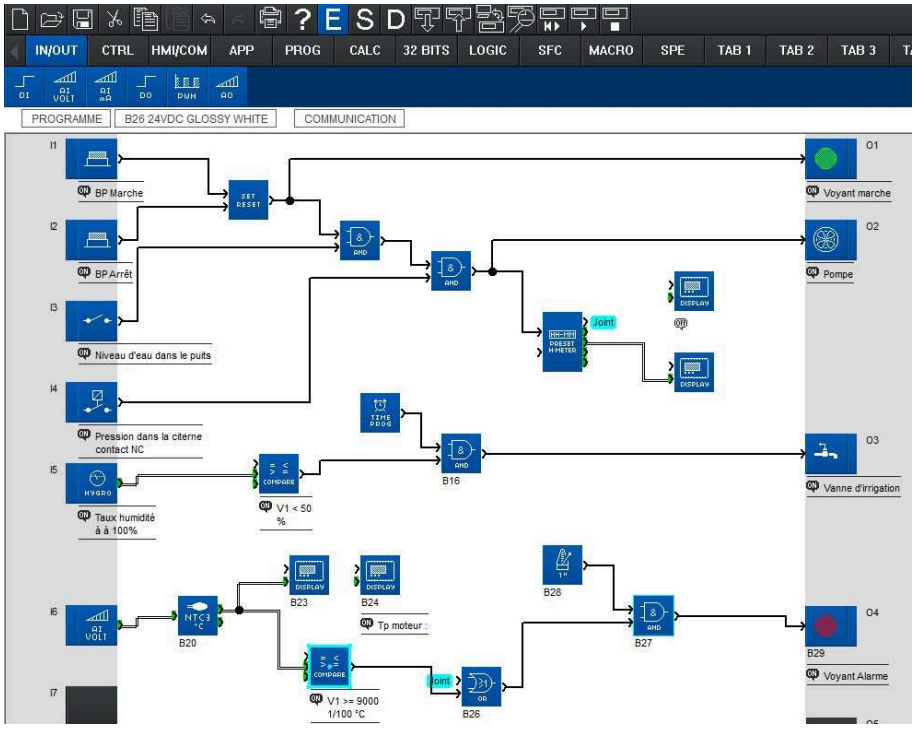


Figure 7.6 Système d'irrigation dans l'atelier de programmation de em4

7.3.2 Définition des entrées et des sorties du système et des profils d'utilisation

La figure 7.6 montre l'application d'irrigation dans l'atelier em4 soft. Une déclaration possible des entrées et des sorties du système en SPTL est la suivante :

```
var
 input bool BoutonOn
 input bool BoutonOff
 input bool NiveauEau
 input bool Pression
 input int Humid
 input int Temp
 output bool Voyant
 output bool Pompe
 output bool Vanne
 output bool Alarme
```

Le booléen `Pression` indique si oui ou non il y a suffisamment de l'eau dans la citerne pour ouvrir la vanne. De même, le booléen `NiveauEau` indique si le niveau d'eau dans le puits est convenable pour procéder au pompage d'eau. Le domaine d'une entrée est l'ensemble des valeurs possibles de cette entrée d'après sa spécification. Ces domaines doivent être définis avant la génération des données de test. Nous utilisons les invariants suivants pour définir le domaine de chaque entrée entière du contrôleur :

- Humidité du sol (`Humid`) : en général, en France elle varie entre 30% et 60%
- Température du moteur de la pompe (`Temp`) : pour nos tests cette température sera entre 25°et 100°.

Nous remarquons ainsi que les valeurs de ces entrées sont fortement liées au climat (Température, précipitations, humidité...). Nous pouvons alors définir dans notre Programme SPTL les catégories "Saison" et "Climat". Voici quelques profils par exemple qui peuvent être générés :

- **Profil Méditerranéen/été**
- **Profil Méditerranéen/hiver**
- **Profil Montagnard/hiver**
- **Profil Montagnard/été**

7.4 Simulation de profils et de scénarios spécifiques

7.4.1 Simulation avec les contraintes du profil uniquement

Commençons par simuler un profil sans scénarios et donc générer des données de test d'une manière aléatoire mais conditionnée par les contraintes générales du profil choisi. Le profil `Méditerranéen/été` est composé à titre d'exemple des groupes de contraintes suivants :

```
   group ete
{
Temp = prob(95, 0.1);
BoutonOn = prob(true, 0.1);
NiveauEau = true;
Pression = prob(true,0.9)}
}
  group mediterranien{
Humid >= 30;
Humid =< 40;
Temp >= 25;
```

```
Temp =< 100
}
```

Le tableau 7.1 représente un extrait de la trace d'exécution d'une séquence de test avec ce profil.

	BoutonOn	BoutonOff	NiveauEau	Pression	Humid	Temp	Voyant	Pompe	Vanne	Alarme
t_0	1	1	1	1	30	53	0	0	1	0
t_1	1	1	1	0	37	95	0	0	1	1
t_2	1	0	1	1	34	94	1	0	1	0
t_3	1	0	1	1	38	95	1	0	1	0
t_4	1	0	1	0	34	84	1	1	1	0
t_5	1	1	1	1	31	96	0	0	1	0
t_6	1	0	1	1	40	49	1	0	1	0
t_7	1	0	1	1	40	100	1	0	1	1
t_8	1	1	1	1	38	89	0	0	1	0
t_9	1	0	1	1	35	25	1	0	1	0

Table 7.1 Extrait d'une séquence de test dans le profil `Méditerranéen/été`

À ce point, le modèle peut être utilisé directement pour générer des données de test répondant à des contraintes invariantes. Pourtant, comme nous pouvons le constater dans le tableau 7.1, ces séquences ne sont pas très concluantes. Cette simulation ne reflète pas exactement l'environnement du système et toutes les entrées ne présentent pas réellement un tel comportement aléatoire. Un taux d'humidité par exemple ne saute pas d'une valeur à une autre de cette manière, de même pour la température du moteur. C'est pour cela qu'il est nécessaire de définir précisément la dynamique de l'environnement, afin d'observer des exécutions plus réalistes grâce aux scénarios.

7.4.2 Simulation avec des scénarios

7.4.2.1 Scénario dans un état normal du système d'irrigation

Essayons maintenant de tester sur l'exemple de l'irrigation avec le profil `Méditerranéen/hiver`. Nous allons ajouter un scénario qui simulera un fonctionnement normal du système d'irrigation lors de son démarrage. Voici un exemple de scénario :

```
scenario Demarrage
var
   time t1
   time t2
begin
   { Humid = 35; Temp=28; t1.start } |
   [ Humid =35; Temp=(pre(Temp)+5) (t1>5) ] |
   { Humid=36; Temp=60; t2.start } |
   [ Humid=36; Temp>60; Temp<65 (t2>5) ]
end
```

Le tableau 7.2 montre un extrait d'une séquence de test résultant du modèle contenant le scénario `Démarrage`.

	BoutonOn	BoutonOff	NiveauEau	Pression	Humid	Temp	Voyant	Pompe	Vanne	Alarme
t_0	1	0	1	1	35	28	1	0	1	0
t_1	1	1	1	1	35	33	0	0	1	0
t_2	1	1	1	0	35	38	0	0	1	0
t_3	1	0	1	0	35	43	1	1	1	0
t_4	1	1	1	1	35	48	0	0	1	0
t_5	0	1	1	1	36	60	0	0	1	0
t_6	1	0	1	1	36	62	1	0	1	0
t_7	1	0	1	0	36	64	1	1	1	0
t_8	1	1	1	0	36	62	0	0	1	0
t_9	1	1	1	1	36	61	0	0	1	0

Table 7.2 Extrait d'une séquence de test dans le profil `Méditerranéen/hiver` avec le scénario `Démarrage`

Cette séquence simule un fonctionnement normal du système d'irrigation. On peut remarquer qu'à l'instant t_0, le contrôleur demande l'ouverture de la vanne, à cause du niveau faible de l'humidité du sol. La vanne est maintenue ouverte. Par la suite, la température du moteur augmente de 5 (puisque le système démarre) tant que le compte T1 n'atteint pas 5 cycles. Ensuite, une fois que le moteur est chauffé, Temp sera comprise entre 60 ° et 65 ° , durant 5 cycles aussi. Nous remarquons que le voyant d'alarme n'a jamais été allumé.

7.4.2.2 Tests intensifs : cas extrêmes

Les séquences de test générées précédemment sont obtenues en ajoutant les invariants qui définissent le fonctionnement correct du système d'irrigation autrement dit une exécution normale. Mais il est important que l'on puisse aussi mettre le système dans un scénario d'exécution anormale et voir ce qui se passe.

7.4.2.2.1 Simulation d'un changement excessif des valeurs des entrées

"Testium" permet de simuler différentes pannes du système en usant des probabilités, des scénarios ou des deux. Un usage intensif des boutons Marche/Arrêt à titre d'exemple peut créer un problème au niveau du système. Nous pouvons aussi jouer sur la valeur de la température du moteur en la faisant osciller entre deux bornes durant une période de temps. Et pour couronner tout ça, nous pouvons rajouter un changement brutal d' l'humidité du sol pour ouvrir et fermer la vanne intensivement. Pour ceci nous pouvons imaginer une catégorie "TypeTest" dans laquelle nous trouvons un groupe "normal" et un groupe "intensif". Dans ce dernier, nous pouvons écrire les deux contraintes suivantes :

```
BoutonOff = prob(true, 0.4);
BoutonOn = prob(true, 0.4);
```

avec un scénario comme suit :

```
scenario oscillation
var
   time t1
begin
   { Temp=28; Humid = 35; t1.start } |
   [ Humid = pre(Humid) + pre(Temp); Temp = -pre(Temp) (t>10) ]
end
```

Dans le profil **Montagnard/hiver** et en exécutant le scénario oscillation, nous pouvons aboutir à des séquences de test dont le tableau 7.3 représente un extrait. Nous remarquons que le contrôleur réagit correctement à notre scénario de test. En effet, il ouvre et referme la vanne à chaque changement brutal d'humidité du sol. De même pour le voyant de marche etc.

7.4.2.2.2 Simulation d'une surchauffe du moteur

Un autre cas extrême envisageable et qui peut avoir des conséquences graves sur le matériel est celui du surchauffe du moteur. Pour simuler ce dernier, un scénario assez simple peut s'appliquer :

```
scenario surchauffe
begin
     { BoutonOn = true; BoutonOff = false; Pression = false;
     Temp = 60; Humid = 35} |
     [ BoutonOn = true; BoutonOff = false; Pression = false;
     Humid = 35; Temp = pre(Temp) + 6 (Temp <100) ]
end
```

Une partie des données qui en résultent est classée dans le tableau 7.4.

	BoutonOn	BoutonOff	NiveauEau	Pression	Humid	Temp	Voyant	Pompe	Vanne	Alarme
t_0	1	1	1	0	35	28	0	0	1	0
t_1	1	0	1	1	63	-28	1	0	0	0
t_2	0	1	1	1	35	28	0	0	1	0
t_3	1	1	1	1	63	-28	0	0	0	0
t_4	1	0	1	0	35	28	1	1	1	0
t_5	1	1	1	1	63	-28	0	0	0	0
t_6	1	1	1	0	35	28	0	0	1	0
t_7	1	1	1	1	63	-28	0	0	0	0
t_8	1	0	1	0	35	28	1	1	1	0
t_9	1	1	1	0	63	-28	0	0	0	0

Table 7.3 Extrait d'une séquence de test dans le profil `Montagnard/hiver` avec le scénario `oscillation`

	BoutonOn	BoutonOff	NiveauEau	Pression	Humid	Temp	Voyant	Pompe	Vanne	Alarme
t_0	1	0	1	0	35	50	1	1	1	0
t_1	1	0	1	0	35	54	1	1	1	0
t_2	1	0	1	0	35	58	1	1	1	0
t_4	1	0	1	0	35	62	1	1	1	0
t_5	1	0	1	0	35	66	1	1	1	0
t_8	1	0	1	0	35	82	1	1	1	0
t_9	1	0	1	0	35	86	1	1	1	0
t_{10}	1	0	1	0	35	90	1	1	1	1
t_{11}	1	0	1	0	35	94	1	1	1	1
t_{12}	1	0	1	0	35	98	1	1	1	1

Table 7.4 Extrait d'une séquence de test dans le profil `Méditerranéen/été` avec le scénario `surchauffe`

Le test guidé par le scénario nous permet d'orienter la variation des données de test dans le sens qui servira notre objectif, qui est dans ce cas le test de l'alarme de surchauffe et voir la réaction du système globalement. Nous avons fixé les valeurs de toutes les entrées et joué uniquement sur la variation de la température du moteur. Comme le montre le tableau 7.4, l'alarme a bien été déclenchée.

7.5 Observations

En testant des exemples d'applications et notamment l'application typique du système d'irrigation, nous voulons évaluer notre prototype "Testium" d'un point de vue applicabilité, complexité de la génération et difficulté d'utilisation.

Les tests ont été effectués sur une machine avec Windows 7 Pro comme système d'exploitation et un processeur Intel Core i3 2,3 GHz avec 8 Go de mémoire RAM. Bien qu'il soit difficile de quantifier le temps nécessaire pour la génération des données, nous avons constaté qu'il faut approximativement moins de 30 secondes pour générer une séquence de cent cycles, pour chacun des modèles SPTL que nous avons utilisés. Il faut noter que ce temps d'exécution augmente avec le nombre et la complexité des contraintes imposées dans le modèle SPTL et avec le nombre d'entrées à générer.

Globalement, l'expérimentation que nous avons effectuée sur des petites applications de em4 et spécialement l'exemple de l'irrigation a montré que les modèles SPTL pour le contrôleur em4 ne sont pas difficiles à construire. La modélisation de l'environnement du contrôleur du système d'irrigation a nécessité un jour de travail pour quelqu'un qui ne connaît rien du domaine de l'irrigation et voulant avoir des valeurs réalistes. Ce temps peut être considérablement réduit pour un test totalement aléatoire. En général, il faut bien préciser que, bien que l'activité de modélisation de test avec SPTL pourrait être effectuée d'une manière relativement rapide, l'effort nécessaire pour construire des modèles de test pour une opération complète de test

est assez difficile à évaluer avec précision. Actuellement, cela dépend de la rigueur souhaitée des séquences de test qui peuvent conduire le testeur à écrire plusieurs scénarios correspondant à différentes situations. Cependant, la construction d'un nouveau scénario peut se faire rapidement en modifiant des scénarios déjà enregistrés et exécutés. Cette expérience suggère que la méthodologie et l'outil sera plus efficace avec des profils prédéfinis par des constructeurs ou les experts de chaque domaine d'application. L'utilisateur final n'a qu'à choisir le profil souhaité et exécuter les tests générés. Il reste totalement libre de modifier les profils ou d'en rajouter des nouveaux.

Il faut noter aussi que l'indépendance de la méthode du code source fait d'elle tout à fait adaptable à n'importe quel système réactif synchrone et pas uniquement pour les contrôleurs logiques programmables.

7.6 Conclusion du chapitre

Dans ce chapitre, nous avons présenté l'environnement de développement du prototype de l'outil "Testium". Nous avons expérimenté ce dernier avec un exemple typique d'application utilisée avec em4 à travers laquelle nous avons explicité les étapes de la procédure de test partant de la définition des catégories et des groupes et arrivant à l'exécution. Le simulateur de em4 Soft a été utilisé pour réaliser les tests. L'expérience montre que l'outil :

- permet d'écrire simplement des scénarios reflétant de nombreuses situations d'utilisation possibles.
- permet d'avoir des données de test adéquates et réalistes.

L'utilisation réelle de ce prototype avec les profils prédéfinis réduirait fortement l'effort demandé

Chapitre 8

Conclusion et perspectives

Le travail que nous venons de présenter s'inscrit dans le cadre de développement des techniques pour le test des systèmes réactifs synchrones. Nous avons effectué une étude sur différents travaux traitant la génération automatique de test pour les systèmes synchrones [JRB06, MB05, Sel09]. Les outils qui résultent de ces travaux se limitent à des applications réservées à des académiques ou des experts. Ce travail a pour but de répondre à des besoins spécifiques des utilisateurs et testeurs des contrôleurs logiques programmables synchrones qui ne sont pas des automaticiens. Nous nous sommes intéressés à une classe particulière de ces CLP produite par InnoVista Sensors : l'em4. Nous proposons une méthode de génération automatique de test basée sur un nouveau langage de spécification de test : le langage SPTL. En plus d'être simple d'utilisation, ce langage permet d'écrire des scénarios de test, éventuellement non déterministes et d'avoir des jeux de test assez réalistes, et ceci en imposant des contraintes sur les données de test qui reflètent l'environnement

réel d'utilisation du système sous test. Nous avons défini la syntaxe et la sémantique du langage SPTL [MDP14, TMDP13]. Ensuite, nous avons donné les principes de la traduction d'une spécification de test sous forme d'un ensemble de contraintes écrites en SPTL vers un programme en langage Prolog.

Ce programme contient, en plus de spécifications de test, des algorithmes s'appuyant sur la programmation par contraintes pour déterminer les entrées valides par rapport à la spécification. Le choix des entrées à envoyer au SST parmi les solutions valides se fait d'une façon totalement aléatoire.

Le langage SPTL est assez proche du langage de programmation des contrôleurs em4, dans le sens où ils manipulent les mêmes types de variables et possèdent des opérateurs semblables. La notion de temps a été aussi introduite dans SPTL (l'opérateur "pre", les variables de type temps). Ces caractéristiques vont aider les testeurs de em4 dans leur tâche. Une autre notion très importante que nous trouvons dans SPTL : la notion de profils. En effet, les utilisateurs peuvent profiter de profils d'utilisation prédéfinis ou en construire des nouveaux à partir d'éléments prédéfinis (catégories et groupes). Un profil contient des contraintes caractérisant un environnement bien déterminé. Ils peuvent aussi y rajouter facilement des scénarios de test. Ces scénarios peuvent être non déterministes. Le non-déterminisme n'est pas réalisé uniquement grâce aux scénarios (les étapes étendues à travers lesquels l'utilisateur exprime ses objectifs de test) mais aussi via l'opérateur de probabilité (prob).

Le langage SPTL est la base d'un outil que nous avons implémenté au cours de ces travaux et que nous avons appelé "Testium" [TM15]. Ce dernier transforme les spécifications de test écrites en SPTL en contraintes qui seront utilisées par des algorithmes de génération de test, développés en programmation logique par contraintes. Cet outil a été utilisé pour mener une expérimentation sur une application typique implémentée sur em4, dans le but de l'évaluer. Même si d'autres expérimentations sont nécessaires, les exemples d'applications testés avec "Testium" montrent sa pertinence, son applicabilité et surtout sa facilité d'utilisation.

Perspectives

Les approches proposées dans cette thèse contribuent à l'amélioration des techniques de test des systèmes synchrones et surtout en ce qui concerne les contrôleurs logiques programmables mais d'autres évolutions sont envisagées et de nombreuses questions nécessitent des études plus approfondies.

Au niveau du langage SPTL :

- Tout d'abord, par rapport à l'oracle : actuellement SPTL offre l'option au testeur de rajouter des spécifications sur les sorties du système par rapport aux jeux de test. Le verdict est donné sous forme de "réussite" ou "échec".

  Une amélioration serait de permettre au testeur d'écrire aussi des degrés ou des niveaux pour caractériser les erreurs toujours dans le fichier SPTL de départ. Ceci permettra d'afficher des messages d'erreurs plus clairs et de savoir si l'erreur nécessite une correction immédiate ou pas. C'est une sorte de distinction entre des messages d'avertissement et des messages pour les erreurs les plus conséquentes. Nous pouvons par exemple introduire un nouveau mot clef `oracleMSG` et ajouter dans la spécification SPTL le code suivant :

  ```
  oracleMSG(1,"alerte erreur");
  oracleMSG(2,"attention risque de mal fonctionnement");
  ```

  Les degrés 1 et 2 seront ajoutés par la suite avec chaque opérateur verdict. À l'affichage des résultats donnés par l'oracle, ces degrés seront traduites par les messages correspondants avec couleurs d'affichage spécifiques par exemple.

- Parmi les autres points qui nécessitent plus d'études, citons l'aspect modulaire que nous pourrions donner au langage SPTL. Une modularité qui peut être au

niveau des scénarios en ajoutant la possibilité de les décomposer et de garder les composantes pour une utilisation ultérieure. Par exemple, nous pourrions donner des noms aux étapes d'un scénario pour pouvoir les enregistrer d'une manière indépendante. Il sera possible ainsi de constituer des scénarios à partir d'anciens "sous-scénarios".

La modularité peut être aussi dans l'appel à des sous-programmes internes ou externes depuis le programmes SPTL. Les sous-programmes internes (ou fonctions) peuvent contenir quelques calculs utiles par exemple. Ceci présente des avantages en terme de structuration du programme et de réutilisation. L'appel à des sous-programmes externes (en FBD par exemple) serait plus difficile car il impliquerait de doter SPTL d'interfaces avec d'autres langages. Mais il permettra aux utilisateurs du système sous-test de réutiliser des modules qu'ils ont déjà développés.

Au niveau de l'exécution des tests :

- Nous pouvons penser à la génération interactive des données de test c'est-à-dire donner la possibilité à l'utilisateur de choisir des entrées qui valident les contraintes à la place d'une génération purement automatique. L'utilisateur guidera ainsi le choix des données de test pour concrétiser son objectif de test. Une idée de présentation serait par exemple de proposer deux modes de test : totalement automatique ou interactif. Une fois le mode interactif activé, le testeur pourrait appliquer des filtres sur les données satisfaisant les contraintes spécifiées au lieu d'utiliser le prédicat de choix aléatoire (choose). À la fin de chaque cycle, il peut changer ses préférences de choix. Ainsi, il a plus de contrôle sur les données générées.

Bibliographie

[ABS03] Stephen A. Edwards Nicolas Halbwachs Paul Le Guernic Albert Benveniste, Paul Caspi et Robert De Simone. The synchronous languages 12 years later. *In : Proceedings of the IEEE*, p. 91(1).

[AL95] J. Arlat et J.C. Laprie. *Guide de la sûreté de fonctionnement.* Cépaduès-Editions, 1995.

[Apt03] Krzysztof Apt. *Principles of Constraint Programming.* New York, NY, USA, Cambridge University Press, 2003.

[BB02] Albert Benveniste et Gérard Berry. Readings in hardware/software co-design. chap. The Synchronous Approach to Reactive and Real-time Systems, pp. 147–159. Norwell, MA, USA, Kluwer Academic Publishers, 2002.

[BB13] Jan Olaf Blech et Sidi Ould Biha. On formal reasoning on the semantics of PLC using coq. *CoRR*, vol. abs/1301.3047, 2013.

[BdS91] F. Boussinot et R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, vol. 79, n9, 1991, pp. 1293–1304.

[Bei90] Boris Beizer. *Software Testing Techniques (2Nd Ed.).* New York, NY, USA, Van Nostrand Reinhold Co., 1990.

[Ber01] A. Bertolino. Ieee swebok trial version 1.00. chap. Chapter 5: Software Testing. May 2001.

[BG92] Gérard Berry et Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, vol. 19, n2, novembre 1992, pp. 87–152.

[BGJ91] Albert Benveniste, Paul Le Guernic et Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, vol. 16, n 2, 1991, pp. 103 – 149.

[Bol15] W. Bolton. Chapter 6 - il, sfc, and ST programming methods. *In : Programmable Logic Controllers (Sixth Edition)*, éd. par W. Bolton, pp. 151 – 185. Boston, Newnes, sixth edition édition, 2015.

[Bou04] M.N. Bouteille. Procédé de programmation d'un automatisme utilisant un langage graphique de schémas à contacts, septembre 22 2004. EP Patent App. EP20,040,075,862.

[Car00] J.M. Carroll. Five reasons for scenario-based design. *Interacting with computers*, vol. 13, n1, 2000, pp. 43–60.

[Cha00] P. Charpentier. Comment construire les tests d'un logiciel. Cahiers de notes documentaires - Hygiène et sécurité du travail - N: 181, 2000.

[CHP07] Paul Caspi, Grégoire Hamon et Marc Pouzet. Lucid synchrone: un langage pour la programmation des systèmes réactifs. *In : Systèmes temps réel.* Lavoisier, 2007.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, vol. 2, n3, mai 1976, pp. 215–222.

[Col90] Alain Colmerauer. An introduction to prolog III. *Commun. ACM*, vol. 33, n7, 1990, pp. 69–90.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs et J. A. Plaice. Lustre: A declarative language for real-time programming. *In : Proceedings of the 14th*

ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 178–188. New York, NY, USA, 1987.

[dB99] L. du Bousquet. *Test fonctionnel statistique de systèmes spécifiés en Lustre*. Grenoble, France, Thèse de PhD, Université Joseph Fourier, September 1999.

[dBORZ99] L. du Bousquet, F. Ouabdesselam, J.-L. Richier et N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. *In: 21st International Conference on Software Engineering (ICSE'99)*. pp. 267–276. Los Angeles, May 1999.

[DO91] Richard A. DeMillo et A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, vol. 17, n9, septembre 1991, pp. 900–910.

[GD15] D.L. Goetsch et S. Davis. *Quality Management for Organizational Excellence: Introduction to Total Quality*. Pearson Education, 2015.

[glo90] *IEEE Standard Glossary of Software Engineering Terminology*. Rapport technique, 1990.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond et Daniel Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, vol. 79, n9, sep 1991, pp. 1305–1320.

[HCS+96] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor et Chris Speirs. *The Mercury Language Reference Manual*. Rapport technique, 1996.

[HLR93] N. Halbwachs, F. Lagnier et P. Raymond. Synchronous observers and the verification of reactive systems. *In: Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, éd. par M. Nivat, C. Rattray, T. Rus et G. Scollo. Twente, June 1993.

[HP85] D. Harel et A. Pnueli. Logics and models of concurrent systems. chap. On the Development of Reactive Systems, pp. 477–498. New York, NY, USA, Springer-Verlag New York, Inc., 1985.

[HUB] D. HUBERT. *Transmission de Puissance Pneumatique*. Ed. Techniques Ingénieur.

[Huu05] Ralf Huuck. Semantics and analysis of instruction list programs. *Electronic Notes in Theoretical Computer Science*, vol. 115, 2005, pp. 3 – 18. Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004)Semantic Foundations of Engineering Design Languages 2004.

[JDDML14] Erwan Jahier, Simplice Djoko-Djoko, Chaouki Maiza et Eric Lafont. Environment-model based testing of control systems: Case studies. *In : International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014), Held as Part of ETAPS 2014.* Grenoble, France, April 2014.

[JHJR04] Anil K. Jain, Lin Hong, Sharath PankantiErwan Jahier et Pascal Raymond. *The lucky language reference manual.* Rapport technique, 5104 CNRS - INPG - UJF, Unitťe Mixte de Recherche, 2004.

[JHR13] Erwan Jahier, Nicolas Halbwachs et Pascal Raymond. Engineering functional requirements of reactive systems using synchronous languages. *In : International Symposium on Industrial Embedded Systems, 2013. SIES'13.* Porto, Portugal, 06 2013.

[JL87] J. Jaffar et J.-L. Lassez. Constraint logic programming. *In : Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. pp. 111–119. New York, NY, USA, 1987.

[JMMJ98] J. Jaffar, M. J. Maher, K. Marriott et P. J.Stuckey. The Semantics of Constraint Logic Programs. *Logic Programming*, vol. 37, n1-3, 1998, p. 1–46.

[JRB06] Erwan Jahier, Pascal Raymond et Philippe Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer*, vol. 8, n6, 2006, pp. 517–530.

[KOM87] Toshio Kawamura, Hayato Ohwada et Fumio Mizoguchi. Cs-prolog: A generalized unification based constraint solver. *In : LP*, éd. par Koichi Furukawa, Hozumi Tanaka et Tetsunosuke Fujisaki. pp. 19–39. Springer.

[Kow74] Robert A. Kowalski. Predicate logic as programming language. *In : IFIP Congress*, pp. 569–574.

[MA00] Bruno Marre et Agnès Arnould. Test sequences generation from LUSTRE descriptions: Gatel. *In : The Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11-15, 2000*, p. 229.

[Mat08] Aditya P. Mathur. *Foundations of Software Testing:*. Delhi :, Pearson Education,, 2008.

[MB05] Bruno Marre et Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electr. Notes Theor. Comput. Sci.*, vol. 111, 2005, pp. 93–111.

[MDP14] Mouna Tka Mnad, Christophe Deleuze et Ioannis Parissis. Synchronous programs testing language (SPTL). *In : Computational Science and Its Applications - ICCSA 2014 - 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part I*, pp. 683–695.

[Meu01] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing Verification and Reliability*, vol. 11, n2, 2001, pp. 81–96.

[MP15] Alexei A. Morozov et Alexander F. Polupanov. Development of the logic programming approach to the intelligent monitoring of anomalous human behaviour. *In : IMTA*, éd. par Igor B. Gurevich, Heinrich Niemann, Ovidio Salvetti et Bernd Radig. pp. 5–13. SciTePress.

[Mus93] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, vol. 10, n2, mars 1993, pp. 14–32.

[Mye79] Glenford J. Myers. *Art of Software Testing*. New York, NY, USA, John Wiley & Sons, Inc., 1979.

[ND12] Srinivas Nidhra1 et Jagruthi Dondeti. Black box and white box testing techniques ?a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, vol. Vol 2, 2012.

[opa08] *Test statistique structurel par résolution par contraintes de choix probabiliste*. Thèse de PhD, Rennes 1, 2008. Thèse doctorat : Informatique.

[Par96] Ioannis Parissis. *Test de logiciels synchrones spécifiés en LUSTRE*. Thèse de PhD, Grenoble 1, 1996. Th. : informatique.

[Par07] Ioannis Parissis. *Méthodes et outils pour le test des logiciels*. Grenoble, France, Habilitation à diriger les recherches, Université Joseph Fourier, December 2007.

[PL01] Alexander Pretschner et Heiko Lötzbeyer. Model based testing with constraint logic programming: First results and challenges. *In : in: proceedings 2nd ICSE intl. workshop on automated program analysis, testing and verification.*

[PLGM91] Michel Le Borgne Paul Le Guernic, Thierry Gautier et Claude Le Maire. Programming real time applications with signal. *In : Proc. IEEE, Special Issue*, p. 79 :1321?1336.

[Pre92] Roger S. Pressman. *Software Engineering (3rd Ed.): A Practitioner's Approach*. New York, NY, USA, McGraw-Hill, Inc., 1992.

[Pre01] Alexander Pretschner. Classical search strategies for test case generation with constraint logic programming. *In : In Proc. Formal Approaches to Testing of Software*. pp. 47–60. BRICS.

[PSP09] Virginia Papailiopoulou, Besnik Seljimi et Ioannis Parissis. Revisiting the Steam-Boiler Case Study with LUTESS : Modeling for Automatic Test Generation. *In : Proceedings of the 12th European Workshop on Dependable Computing, EWDC 2009*. p. 8 pages. Toulouse, France, mai 2009.

[RRJ08] Pascal Raymond, Yvan Roux et Erwan Jahier. Specifying and Executing Reactive Scenarios With Lutin. *Electr. Notes Theor. Comput. Sci.*, vol. 203, n4, 2008, pp. 19–34.

[RWNH98] P. Raymond, D. Weber, X. Nicollin et N. Halbwachs. Automatic testing of reactive systems. *In : 19th IEEE Real-Time Systems Symposium*. Madrid, Spain, décembre 1998.

[Sco10] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog*. Outskirts Press, 2010.

[SE07] Dorothy Graham Julian Harty David Hayman Juha Itkonen Vipul Kocher Fernando Lamas de Oliveira Tilo Linz Peter Morgan Thomas Müller Avi Ofer Dale Perry Horst Pohlmann Meile Posthuma Erkki Pöyhönen Maaret Pyhäjärvi Andy Redwood Stuart Reid Hans Schaefer Jurriën Seubers Dave Sherrat Mike Smith Andreas Spillner Richard

Taylor Geoff Thompson Matti Vuori Stephanie Ulrich Pete Williams Sigrid Eldh, Isabel Evans. *Glossaire CFTL/ISTQB des termes utilisés en tests de logiciels*. Erik van Veenendaal, Décembre 2007.

[Sel09] Besnik Seljimi. *Testing synchronous software with CLP*. Theses, Université Joseph-Fourier - Grenoble I, juillet 2009.

[SF12] V. Senni et F. Fioravanti. Generation of test data structures using constraint logic programming. *In : 6th International Conference on Tests and Proofs (TAP 2012)*.

[SP06] Besnik Seljimi et Ioannis Parissis. Using clp to automatically generate test sequences for synchronous programs with numeric inputs and outputs. *In : Proceedings of the 17th International Symposium on Software Reliability Engineering*. pp. 105–116. Washington, DC, USA, 2006.

[TM15] Deleuze C. Parissis I. TKA Mnad, M. Testium: outil de génération automatique de données de test pour les systèmes synchrones. *In : Approches Formelles dans l'Assistance au Développement de Logiciels AFADL'2015*. Bordeaux, France, 2015.

[TMDP13] Mouna TKA Mnad, Christophe Deleuze et Ioannis Parissis. Synchronous Programs Testing Language (SPTL). MSR 2013 - Modélisation des Systèmes Réactifs, 2013.

[Tri12] Markus Triska. The finite domain constraint solver of SWI-Prolog. *In : FLOPS*, pp. 307–316.

[Vas04] Jérôme Vassy. *Génération automatique de cas de test guidée par des propriétés de sûreté*. Grenoble, France, Thèse de PhD, Université Joseph Fourier, October 2004.

[WSL15] M. Wcislik, K. Suchenia et M. Łaskawski. Programming of sequential control systems using functional block diagram language. *IFAC-*

PapersOnLine, vol. 48, n4, 2015, pp. 330 – 335. 13th IFAC and IEEE Conference on Programmable Devices and Embedded SystemsPDES 2015.

[ZBM+07] F.L. Zucatto, C.A. Biscassi, F. Monsignore, F. Fidelix, S. Coutinho et M.L. Rocha. Zigbee for building control wireless sensor networks. *In : Microwave and Optoelectronics Conference, 2007. IMOC 2007. SBMO/IEEE MTT-S International*, pp. 511–515.

[Zua] Nicolas Zuanon. Modular feature integration and validation in a synchronous context. pp. 213–231.

Appendices

Chapitre A

Grammaire de SPTL

```
ModelTest ::= Declaration Categs | Scenario

Declaration ::= {{ DeclConsts }} DeclVars

DeclVars ::= var DeclVar ( ";" Decl Var )*

DeclVar ::= {{ output | input }} {{ random}} Type
            Ident {{ "=" Value }}

DeclConsts ::= const DeclConst ( ";" DeclConst )*

DeclConst ::= Type Ident "=" Value

IDENT ::= [a-zA-Z0-9]*
```

```
Type ::= int {{ in "[" Integer ":" Integer"]" }} | time
        |  bool | string

Value ::= Bool | Time | Integer | Symbol

Time ::= Integer

Categs ::= Category ( ";" Category)*

Category  ::= categ Ident  "{"  Groups   "} "

Equation ::= Ident = Exp;

Groups ::= Group ( ";" Group)*

Group :: =  group Ident " { " {{ DeclVars }}
          (Constraint ";")* Constraint" } "

Constraint ::= Exp

Exp ::=   Exp OpBin Exp | OpUni Exp | CondExp

| " ( " Exp " ) " | Prob"(" Exp,ProbFloat ")"

| Value | Ident |Function | Ident.start | Ident.stop

Function ::= Ident" ( " Exp ( ";" Exp )* " ) "

ProbFloat ::= int.int                          // in [0.0..1.0]

OpUni ∈ { - , not, pre\}

OpBin ∈ { +, -,*,div , mod, and, or, =,
```

```
 /=,  > , <, =< , =>, nor, nand, xor }

CondExp ::=  if Exp then Exp else Exp

Scenario ::= {{ Constraint* }} {{ DeclVars}}

 begin ( PoncConstraint (:: verdict ( Exp (; Exp)* ))* |
  IntervConstraint (:: verdict ( Exp(; Exp)* ))* )( " |"
  ( PoncConstraint (:: verdict ( Exp (; Exp)* ))* |
   IntervConstraint (:: verdict ( Exp (; Exp)* ))* ) )*

 end

IntervConstraint :: = "[" Constraint +

"(" Condition ")" "]"

PoncConstraint ::= "{" Constraint + "}"

Condition ::= Constraint

Entier  ∈  Z

Bool ∈ { true, false }

Symbol ∈ [a–zA–Z0–9]*
```

Résumé

Ce travail de thèse, effectué dans la cadre du projet FUI Minalogic Bluesky, porte sur le test fonctionnel automatisé d'une classe particulière de contrôleurs logiques programmables (em4) produite par InnoVista Sensors. Ce sont des systèmes synchrones qui sont programmés au moyen d'un environnement de développement intégré (IDE). Les personnes qui utilisent et programment ces contrôleurs ne sont pas nécessairement des programmeurs experts. Le développement des applications logicielles doit être par conséquent simple et intuitif. Cela devrait également être le cas pour les tests. Même si les applications définies par ces utilisateurs ne sont pas nécessairement très critiques, il est important de les tester d'une manière adéquate et efficace. Un simulateur inclus dans l'IDE permet aux programmeurs de tester leurs programmes d'une façon qui reste à ce jour informelle et interactive en entrant manuellement des données de test. En se basant sur des recherches précédentes dans le domaine du test des programmes synchrones, nous proposons un nouveau langage de spécification de test, appelé SPTL (Synchronous Programs Testing Language) qui rend possible d'exprimer simplement des scénarios de test qui peuvent être exécutés à la volée pour générer automatiquement des séquences d'entrée de test. Il permet aussi de décrire l'environnement où évolue le système pour mettre des conditions sur les entrées afin d'arriver à des données de test réalistes et de limiter celles qui sont inutiles. SPTL facilite cette tâche de test en introduisant des notions comme les profils d'utilisation, les groupes et les catégories. Nous avons conçu et développé un prototype, nommé "Testium", qui traduit un programme SPTL en un ensemble de contraintes exploitées par un solveur Prolog qui choisit aléatoirement les entrées de test. La génération de données de test s'appuie ainsi sur des techniques de programmation logique par contraintes. Pour l'évaluer, nous avons expérimenté cette méthode sur des exemples d'applications EM4 typiques et réels. Bien que SPTL ait été évalué sur em4, son utilisation peut être envisagée pour la validation d'autres types de contrôleurs ou systèmes synchrones.