Test de conformité de contrôleurs logiques spécifiés en grafcet

Julien Provost

EDSP
École Doctorale
Sciences Pratiques

ENS
CACHAN

ENSC-2011 n°296

THÈSE DE DOCTORAT

DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

présentée par

Monsieur Julien PROVOST

pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Domaine :

Électronique, Électrotechnique, Automatique

Sujet de la thèse :

TEST DE CONFORMITÉ DE CONTRÔLEURS LOGIQUES SPÉCIFIÉS EN GRAFCET

Thèse présentée et soutenue à Cachan, le 8 juillet 2011, devant le jury composé de :

Janan ZAYTOON	Professeur, Université de Reims – CReSTIC	Président
Hassane ALLA	Professeur, Université de Grenoble – GIPSA-Lab	Rapporteur
Thierry JÉRON	Directeur de recherche, INRIA Rennes	Rapporteur
Franck CORBIER	Docteur, Ingénieur R&D, Geensoft/Dassault Systèmes	Examinateur
Jean-Marc FAURE	Professeur, SUPMECA Paris – LURPA	Directeur de thèse
Jean-Marc ROUSSEL	Maître de conférences, ENS Cachan – LURPA	Encadrant

Laboratoire Universitaire de Recherche en Production Automatisée
ENS de Cachan / EA 1385 / Université de Paris Sud 11
61, avenue du Président Wilson – 94235 Cachan Cedex

Remerciements

C'est avec grand plaisir que j'exprime en avant-propos à ce mémoire mes remerciements à toutes les personnes qui ont contribué de près ou de loin à l'élaboration de cette thèse.

Les travaux de recherche présentés dans ce mémoire de thèse ont été réalisés au sein du Laboratoire Universitaire de Recherche en Production Automatisée (LURPA) de l'École Normale Supérieure de Cachan.

J'adresse tout d'abord mes sincères remerciements à Monsieur le Professeur Jean-Marc FAURE pour avoir assumé la direction de mes travaux ainsi qu'à Monsieur Jean-Marc ROUSSEL pour son encadrement. Merci à tous les deux pour vos précieux conseils et remarques, merci de m'avoir fait profiter de votre expérience et de vos compétences. Merci à Jean-Marc ROUSSEL pour son aide au quotidien, aussi bien lors de discussion autour d'une équation que lors de développement logiciel.

Merci aux membres du jury pour le temps qu'ils m'ont consacré. Merci à Monsieur le Professeur Janan ZAYTOON qui me fait l'honneur de présider mon jury de thèse. Merci à Monsieur le Professeur Hassane ALLA et à Monsieur Thierry JÉRON, Directeur de recherche à l'INRIA Rennes, pour leur lecture approfondie de ce mémoire et leur remarques précises et détaillés. Merci à Monsieur Franck CORBIER pour l'intérêt qu'il a porté à mes travaux.

Merci également à l'ensemble des membres du projet TESTEC pour les échanges enrichissants que nous avons pu avoir durant ces trois années.

Merci à tous les membres du LURPA, anciens et actuels, pour les bons moments partagés au laboratoire durant ces années. Merci à mes compagnons de galères durant ces dernières longues soirées de printemps passées au laboratoire, Robin et David. Bon courage à ceux qui sont dans la dernière ligne droite, Dom, Pierre, Alain...

Enfin, merci à ma famille pour avoir essayé de comprendre ce à quoi j'occupais mes journées. Merci également pour leur soutien et pour leur présence le jour de la soutenance. Merci à Irène pour m'avoir encouragé et supporté durant ces trois années.

Résumé

Les travaux présentés dans ce mémoire de thèse s'intéressent à la génération et à la mise en œuvre de séquences de test pour le test de conformité de contrôleurs logiques. Dans le cadre de ces travaux, le Grafcet (IEC 60848 (2002)), langage de spécification graphique utilisé dans un contexte industriel, a été retenu comme modèle de spécification. Les contrôleurs logiques principalement considérés dans ces travaux sont les automates programmables industriels (API). Afin de valider la mise en œuvre du test de conformité pour des systèmes de contrôle/commande critiques, les travaux présentés proposent :

- Une formalisation du langage de spécification Grafcet. En effet, l'application des méthodes usuelles de vérification et de validation nécessitent la connaissance du comportement à partir de modèles formels. Cependant, dans un contexte industriel, les modèles utilisés pour la description des spécifications fonctionnelles sont choisis en fonction de leur pouvoir d'expression et de leur facilité d'utilisation, mais ne disposent que rarement d'une sémantique formelle.
- Une étude de la mise en œuvre de séquences de test et l'analyse des verdicts obtenus lors du changement simultané de plusieurs entrées logiques. Une campagne d'expérimentation a permis de quantifier, pour différentes configurations de l'implantation, le taux de verdicts erronés dus à ces changements simultanés.
- Une définition du critère de SIC-testabilité d'une implantation. Ce critère, déterminé à partir de la spécification Grafcet, définit l'aptitude d'une implantation à être testée sans erreur de verdict. La génération automatique de séquences de test minimisant le risque de verdict erroné est ensuite étudiée.

Mots-clés : Test de conformité, Grafcet, Contrôleurs logiques, Automate Programmable Industriel, Transformation de modèle, SIC-testabilité

Abstract

The works presented in this PhD thesis deal with the generation and implementation of test sequences for conformance test of logic controllers. Within these works, Grafcet (IEC 60848 (2002)), graphical specification language used in industry, has been selected as the specification model. Logic controllers mainly considered in these works are Programmable Logic Controllers (PLC). In order to validate the carrying out of conformance test of critical control systems, this thesis presents :

- A formalization of the Grafcet specification language. Indeed, to apply usual verification and validation methods, the behavior is required to be expressed through formal models. However, in industry, the models used to describe functional specifications are chosen for their expression power and usability, but these models rarely have a formal semantics.
- A study of test sequences execution and analysis of obtained verdicts when several logical inputs are changed simultaneously. Series of experimentation have permitted to quantify, for different configurations of the implantation under test, the rate of erroneous verdicts due to these simultaneous changes.
- A definition of the SIC-testability criterion for an implantation. This criterion, determined on the Grafect specification defines the ability of an implementation to be tested without any erroneous verdict. Automatic generation of test sequences that minimize the risk of erroneous verdict is then studied.

Keywords : Conformance test, Grafcet, Logic controllers, Programmable Logic Controller, Model transformation, SIC-testability

Table des matières

Liste des figures

Liste des tableaux

Liste des acronymes

API	Automate Programmable Industriel . . . 122
APIdS	Automate Programmable Industriel dédié à la Sécurité . . . 87
ALS	Automate des Localités Stables . . . 123
BIST	Built-In Self Test . . . 117
DFL	Diagramme Fonctionnel Logique . . . 19
ECU	Electronic Control Unit . . . 10
GSA	Graphe des Situations Accessibles . . . 57
IOLTS	système de transitions étiquetées à entrée/sortie (Input/Output Labelled Transition System) . . . 91
LTS	système de transitions étiquetées (Labelled Transition System) . . . 33
méthode TT	méthode du Tour de Transition . . . 122
MaxC-SIC	Maximum Consecutive Single Input Change . . . 122
MIC	Multiple Input Change . . . 123
min-MIC	Minimum Multiple Input Change . . . 122
PCO	Point de Contrôle et d'Observation . . . 100
POU	Program Organization Unit . . . 53
SCC	Système de Contrôle/Commande . . . 43
SDH	Système Dynamique Hybride . . . 24
SED	Système à Évènements Discrets . . . 24
SIC	Single Input Change . . . 122
TOR	Tout Ou Rien . . . 100
V&V	Vérification et Validation . . . 15
VLSI	à haut niveau d'intégration (Very-Large-Scale Integration) . . . 29

Introduction

Les systèmes de contrôle/commande industriels sont aujourd'hui de plus en plus souvent commandés par des contrôleurs industriels programmables. Ceci concerne également les applications critiques tels que l'automobile, le transport ferroviaire ou la production d'électricité, où les contrôleurs utilisés peuvent être des calculateurs embarqués ou des automates programmables industriels. Pour ces applications critiques, il est impératif de pouvoir garantir le bon comportement des contrôleurs les pilotant.

Afin de répondre à ce besoin industriel et sociétal, différents moyens sont mis en œuvre pour garantir le niveau de sûreté exigé pour ce type de systèmes, aussi bien pendant la phase de développement de ces systèmes (méthodes hors ligne) que pendant leur phase d'opération (méthodes en ligne).

Cette thèse s'inscrit dans le projet ANR TESTEC [1] dans lequel le LURPA travaille en collaboration avec l'INRIA Rennes, le LaBRI à Bordeaux et l'I3S à Sophia Antipolis et deux partenaires industriels : EDF R&D [2] et Geensoft [3]. Ce projet étudie l'application des techniques de test de conformité aux systèmes de grande taille, ainsi que l'apport des méthodes de vérification pour le test de conformité.

L'objectif de cette thèse est de proposer une méthode de réalisation du test de conformité des contrôleurs logiques utilisés dans les systèmes de contrôle/commande critiques. Le test de conformité est une méthode opérant sur une réalisation du contrôleur, appelée implantation. Son objectif est de garantir que le comportement de l'implantation est conforme au comportement défini par sa spécification. Les travaux réalisés durant cette thèse s'intéressent plus particulièrement aux implantations réalisées par un automate programmable industriel (API) établies à partir de spécifications définies en Grafcet.

Ce mémoire de thèse est composé de quatre chapitres qui nous permettront de

1. TESTEC : TEst des Systèmes Temps réel Embarqués Critiques. Projet ANR (ANR-07 TLOG 022)
2. EDF R&D : Electricité De France, Recherche & Développement
3. Geensoft : Éditeur de logiciels dédiés au développement, au test et à la validation de systèmes embarqués et d'automatismes industriels.

présenter le contexte scientifique des travaux, de proposer une méthode de formalisation du langage de spécification Grafcet et de génération de séquences de test, puis d'exposer l'approche expérimentale qui a été conduite concernant l'exécution de ces séquences de test et les nouvelles contributions formelles qui en découlent.

D'une manière plus générale, le premier chapitre présente tout d'abord les architectures des systèmes de contrôle/commande industriels. Cette partie nous permettra de préciser la composition et le fonctionnement des contrôleurs logiques utilisés. La seconde partie de ce chapitre présente les différents moyens de vérification et de validation de systèmes de contrôle/commande et permet de positionner le test de conformité par rapport aux autres méthodes de vérification et de validation. La troisième partie est consacrée aux méthodes relatives au test de conformité, tandis que la quatrième partie présente les modèles formels couramment utilisés dans ce contexte.

Ce chapitre permet d'identifier l'écart entre les nombreux travaux académiques abordant le problème du test de conformité à partir de système formel et les besoins et pratiques industriels. En effet, certaines phases de vérification et de validation restent mal maîtrisées ou effectuées manuellement. L'importance de cette problématique explique l'implication d'EDF et de Geensoft dans le projet TESTEC.

Partant de ce constat, notre première contribution est présentée dans le deuxième chapitre. La méthode proposée permet la formalisation d'une spécification définie dans un langage normalisé largement utilisé dans l'industrie pour la modélisation de comportement séquentiel. La méthode de formalisation proposée a été développée dans l'objectif d'une utilisation pour le test de conformité réalisé en boîte noire, le concept de localité d'une spécification Grafcet étend le concept de situation pour permettre la prise en compte des sorties logiques émises.

Le troisième chapitre est fortement orienté "expérimentation". Des expérimentations ont été menées pour étudier les verdicts obtenus en appliquant les techniques de génération de séquence de test existantes pour les machines de Mealy. Ces expérimentations ont pour objectif d'étudier l'applicabilité des techniques usuelles au test de contrôleurs logiques utilisés dans un contexte industriel, en particulier pour des automates programmables industriels.

Enfin, le dernier chapitre s'intéresse au développement d'une méthode de génération de séquences de test en prenant en compte le comportement de l'implantation vis-à-vis de la lecture des entrées logiques. Cette prise en compte de la spécificité du comportement

des contrôleurs logiques permet ainsi la génération de séquences de test garantissant l'absence d'erreur de verdict lors de leur exécution. Un *taux de couverture garanti sans erreur de verdict* permet de quantifier le niveau de confiance du test de conformité effectué.

En conclusion, une synthèse des principaux résultats obtenus est effectuée, et quelques perspectives de ces travaux sont proposées.

Chapitre 1

Contexte et positionnement du problème

Sommaire

Introduction

Le travail de recherche réalisé durant cette thèse s'intéresse au test de conformité de contrôleurs logiques dont la spécification comportementale est définie par un Grafcet.

Dans ce chapitre, le contexte de ces travaux ainsi qu'une analyse bibliographique seront présentés afin de préciser le cadre scientifique de ces travaux.

Ce chapitre se décompose en quatre parties, organisées comme suit :

- la première section présente les Systèmes de Contrôle/Commande (SCC) industriels et détaille la composition et le fonctionnement d'un Automate Programmable Industriel (API), composant de ces systèmes qui sera utilisé lors de nos expérimentations ;
- la deuxième section présente les techniques de vérification et de validation permettant d'assurer la sûreté des SCC. Cette partie permettra de préciser la place du test de conformité dans le cycle de développement d'un SCC ;
- la troisième section se focalise davantage sur le test de conformité de contrôleurs logiques, et propose une approche comparative entre le test de circuits électroniques intégrés et le test de logiciels ;
- la quatrième section présente les modèles formels couramment utilisés pour le test de conformité et précise le choix du formalisme retenu ;
- enfin, la dernière section précise les objectifs des travaux effectués durant cette thèse et leur positionnement par rapport aux travaux existants.

1 Les systèmes de contrôle/commande industriels

1.1 Architectures des systèmes de contrôle/commande

Les figures 1.1 et 1.2 présentent deux types d'architecture caractéristiques de SCC contrôlant des processus critiques. La première illustre le découpage en niveaux d'un SCC d'une centrale électrique, tandis que la seconde illustre la découpage modulaire d'une architecture de contrôle/commande d'un véhicule automobile. Dans les deux cas, les systèmes assurent des fonctions séquentielles et des fonctions continues (régulation, asservissement) ainsi que des fonctions de communication et de dialogue au travers d'interfaces homme/machine. La communication entre les différents éléments architecturaux de ces SCC est assurée par différents réseaux de communication.

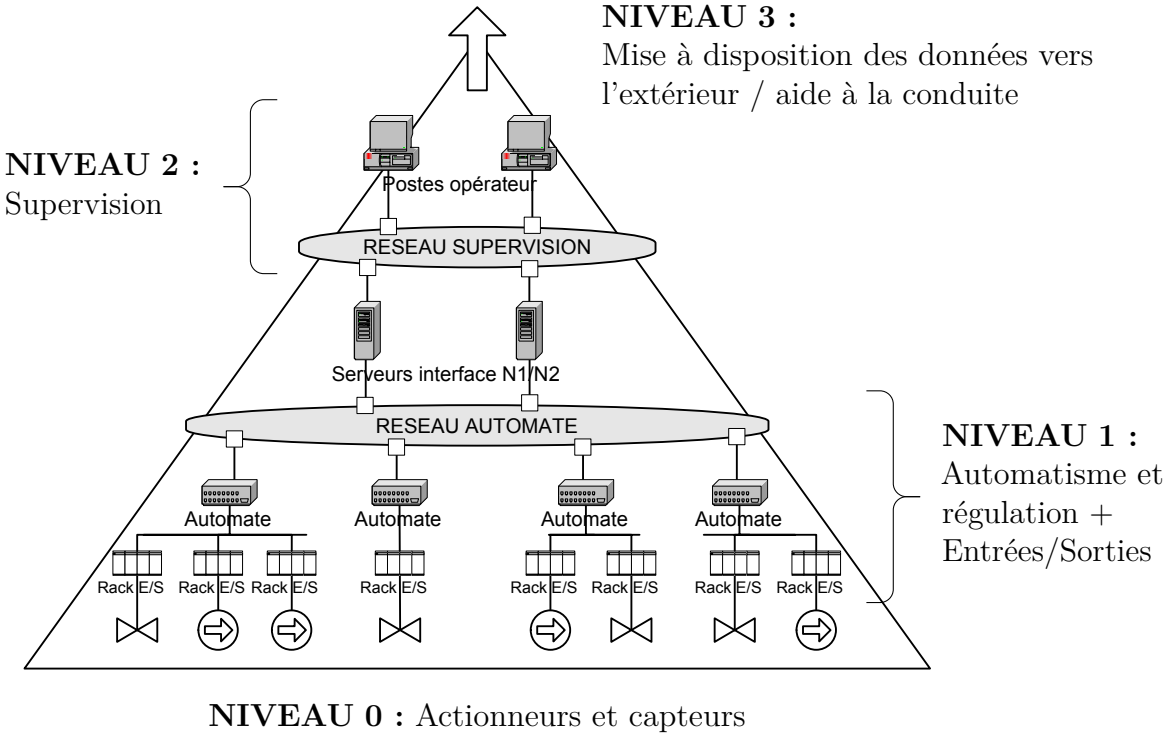


Figure 1.1 – Architecture type d'un système de contrôle/commande d'une centrale électrique (Source : EDF R&D)

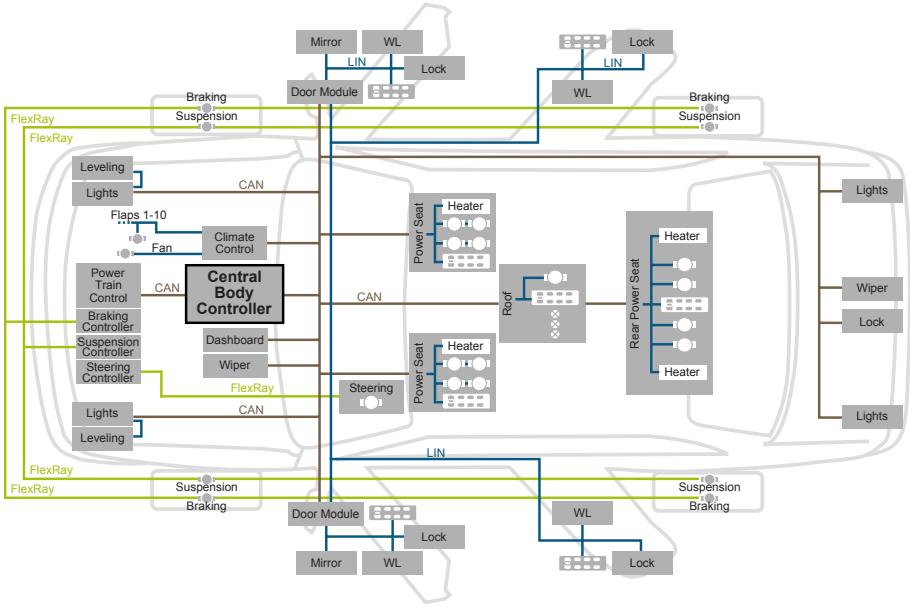


Figure 1.2 – Architecture type d'un système de contrôle/commande automobile (Source : Freescale Semiconductor)

Le découpage hiérarchique proposé pour l'architecture du SCC d'une centrale électrique permet de séparer les niveaux selon le type de fonctions assurées par les composants de chaque niveau. Ainsi, le niveau 0 est constitué des composants physiques en interaction directe avec le système physique piloté (la partie opérative) ; ces composants de niveau 0 permettent d'acquérir les informations du système et d'agir sur celui-ci, il s'agit des capteurs, détecteurs, pré-actionneurs et actionneurs. Le niveau 1 est constitué de contrôleurs logiques et de cartes d'entrées/sorties permettant d'assurer les fonctions logiques et continues du SCC ; ces composants permettent de calculer les ordres à envoyer aux actionneurs en fonction des mesures issues des capteurs et détecteurs. Le niveau 2 est constitué de composants permettant les fonctions de conduite et de supervision ; les grandeurs mesurées au niveau 1 sont transmises aux opérateurs par le biais d'interface homme/machine et permettent ainsi aux opérateurs de transmettre des ordres de démarrage ou de changement de séquence d'opération. Le niveau 3 traite le suivi et l'optimisation de la conduite ainsi que la maintenance du SCC. Il est composé d'outils d'aide à la décision.

Le découpage proposé pour l'architecture du SCC d'un véhicule automobile est un découpage type induit depuis la mise en place des réseaux de communication dans les SCC de ces véhicules. Cette architecture modulaire permet au SCC d'être adapté en fonction de la gamme et des options du véhicule. Chaque module assure une fonction bien précise et délimitée par ses entrées/sorties (module d'ouverture et de fermeture à distance, module ABS, module contrôle de la pression des pneus...). Cette décomposition des fonctions à assurer en modules permet en outre de faciliter la sous-traitance et la production de ces modules en grande série.

Les travaux présentés dans ce mémoire traitent du *test de conformité de contrôleurs logiques*. Ces travaux s'intéressent plus particulièrement à l'étude des *systèmes non temporisés*. Cette limitation aux systèmes non temporisés, restrictive certes, ne diminue pas pour autant l'intérêt et la portée de ces travaux car la première préoccupation des ingénieurs lors du test de conformité d'une implantation est la validation de la conformité des fonctions critiques logiques ; seule la conformité du comportement non temporisé de l'implantation par rapport à sa spécification est validée. La validation de la conformité des contraintes temporelles est effectuée dans une seconde phase, après validation des contraintes logiques. De plus, pour beaucoup d'installations industrielles, les traitements logiques (mise en service, verrouillage, déclenchement d'alarme) ont un degré de criticité

beaucoup plus important que les opérations de régulation faisant intervenir des variables numériques ou analogiques et le temps. Par exemple, un système régulé peut évoluer dans une plage dite de bon fonctionnement autour d'un point de fonctionnement optimal. Des écarts modérés sont donc tolérés ; en revanche, le dépassement des bornes de la plage de bon fonctionnement est beaucoup plus problématique et nécessite une correction plus rapide. Par exemple, s'il est important de réguler la vitesse de rotation d'une turbine, il est encore plus important de gérer le dépassement des seuils de sa vitesse de rotation. Par conséquent, seuls des signaux logiques sont considérés, ceux-ci pouvant désigner soit l'état 1 ou 0 d'un détecteur soit un seuil d'un capteur analogique ou numérique.

Les travaux présentés ont pour cible d'application des contrôleurs logiques tels que ceux utilisés pour assurer les fonctions logiques de niveau 1 d'un SCC d'une centrale électrique (API) ou d'un module d'un SCC d'un véhicule automobile (ECU). Les technologies des API et des Electronic Control Units (ECU) étant différentes, les expérimentations ont été réalisées sur des API, décrits dans la section suivante, mais une démarche similaire pourrait être appliquée aux ECU.

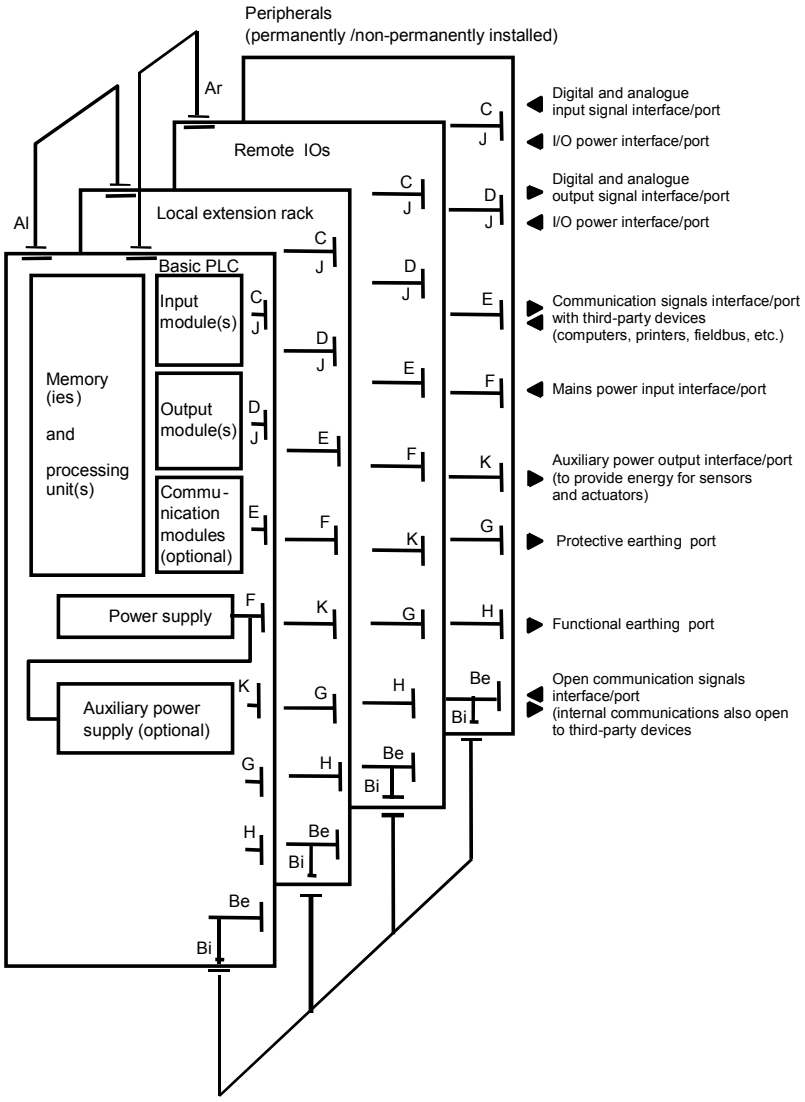
1.2 Les Automates Programmables Industriels (API)

Les SCC de production sont souvent pilotés par des API. Les API font partie des contrôleurs les plus répandus en automatisme, dans différents secteurs : industrie manufacturière, transport ferroviaire, production d'énergie, agriculture, gestion d'accès (parking, bâtiment).

Les normes internationales IEC 61131 (IEC 61131-1 (2003) ; IEC 61131-2 (2007) ; IEC 61131-3 (2003) ; IEC 61131-4 (2004) ; IEC 61131-5 (2000) ; IEC 61131-6 (2010) ; IEC 61131-7 (2000) ; IEC 61131-8 (2000)) définissent de nombreux aspects matériels, logiciels et de communications des API. Les sections suivantes ont pour objectifs de présenter succinctement la structure, le fonctionnement et la programmation de ces API. Dans le cadre de ce mémoire, nous nous focaliserons davantage sur la norme IEC 61131-1 (2003) qui décrit la structure et le fonctionnement d'un API et sur la norme IEC 61131-3 (2003) qui décrit les langages de programmation des API.

1.2.1 Structure d'un automate programmable industriel

La figure 1.3 donne une illustration de la structure type d'un API modulaire.



Key

AI	Communication interface/port for local I/O
Ar	Communication interface/port for remote I/O station
Be	Open-communication interface/port also open to third-party devices (for example, personal computer used for programming instead of a PADT)
Bi	Internal communication interface/port for peripherals
C	Interface/port for digital and analogue input signals
D	Interface/port for digital and analogue output signals
E	Serial or parallel communication interfaces/ports for data communication with third-party devices
F	Mains power interface/port. Devices with F ports have requirements on keeping downstream devices intelligent during power-up, power-down and power interruptions.
G	Port for protective earthing
H	Port for functional earthing
J	I/O power interface/port used to power sensors and actuators
K	Auxiliary power output interface/port

Figure 1.3 – Structure et interfaces d'un API modulaire (IEC 61131-2 (2007))

Un API modulaire est constitué des principaux modules suivants :

1) Un module de base, assurant les fonctions principales de l'API :
   - stockage du programme embarqué et des variables d'entrée et de sortie ;
   - exécution du programme embarqué ;
   - communication avec l'atelier de programmation et avec les autres modules ;

   Les fonctions de stockage sont assurées par différents types de mémoires (EEPROM pour le stockage du programme, RAM pour le stockage des variables d'entrée et de sortie, ROM pour le moniteur d'exécution).

   L'exécution du programme est assurée par un processeur ; ce processeur permet d'effectuer aussi bien des opérations simples sur des bits (AND, OR, NOT), des opérations de comptage et de temporisation que des opérations plus complexes telles que des fonctions mathématiques (SQRT, SIN, COS) ou des fonctions de corrections (PID).

   Différents modules de communication permettent d'une part les communications avec l'atelier de programmation pour l'édition et la mise au point du programme utilisateur, et d'autre part les communications avec les modules d'entrées/sorties locaux ainsi que les modules d'entrées/sorties déportés. Ces communications sont effectuées au travers de réseaux différents utilisant des protocoles tels que InterBus, Modbus ou Ethernet industriel.

   De plus, le module de base peut comporter quelques entrées et sorties afin de permettre un fonctionnement autonome.

2) Des modules (ou cartes) d'entrées, assurant l'interface avec les capteurs et détecteurs de la partie opérative.

   Les modules d'entrées permettent de mesurer des grandeurs de différents types : logiques, numériques ou analogiques. En pratique, afin d'améliorer les performances de l'API, chaque module d'entrées est dédié à un type d'entrées ; par exemple, un API peut être relié à un module de 16 entrées logiques et à un module de 4 entrées analogiques. La tension d'entrée la plus répandue est la tension continue de 24V.

   Nous reviendrons plus en détail sur le comportement de ces modules d'entrées dans le chapitre 3, lors de la mise en œuvre des séquences de test sur un API.

3) Des modules (ou cartes) de sorties, assurant l'interface avec les préactionneurs de la partie opérative.

   Tout comme les modules d'entrées, chaque module de sorties permet de piloter

un ensemble de sorties du même type : logiques, numériques ou analogiques ; la tension de sortie la plus répandue est également la tension continue de 24V.

Afin de protéger les circuits internes des modules d'entrée/sortie et du module de base des parasites et des surcharges électriques, l'utilisation d'optocoupleurs permet d'assurer le découplage entre les circuits électriques internes à l'API et les circuits électriques de la partie opérative. L'emploi de pré-actioneurs permet également d'adapter les tensions entre le circuit de commande et le circuit de puissance.

Dans le cas de SCC de grande taille (d'un point de vue géographique), l'emploi de modules d'entrée/sorties déportés permet d'y connecter des entrées et des sorties réparties à différents endroits du système piloté et d'échanger les valeurs de ces entrées et sorties avec l'API via le réseau et le module de communication du module de base.

En cas de besoins plus spécifiques, d'autres modules complémentaires, tels que des modules de comptage rapide ou des modules de commandes d'axes numériques, peuvent également être connectés au module de base afin d'en accroitre les possibilités.

Dans le cadre de ces travaux, portant sur le test de conformité d'une implantation composée d'un API exécutant un programme embarqué, nous nous intéresserons plus particulièrement à l'étude des entrées et sorties logiques ainsi qu'au module principal (Basic PLC) et aux modules secondaires locaux (Local extension rack).

1.2.2 Fonctionnement d'un automate programmable industriel

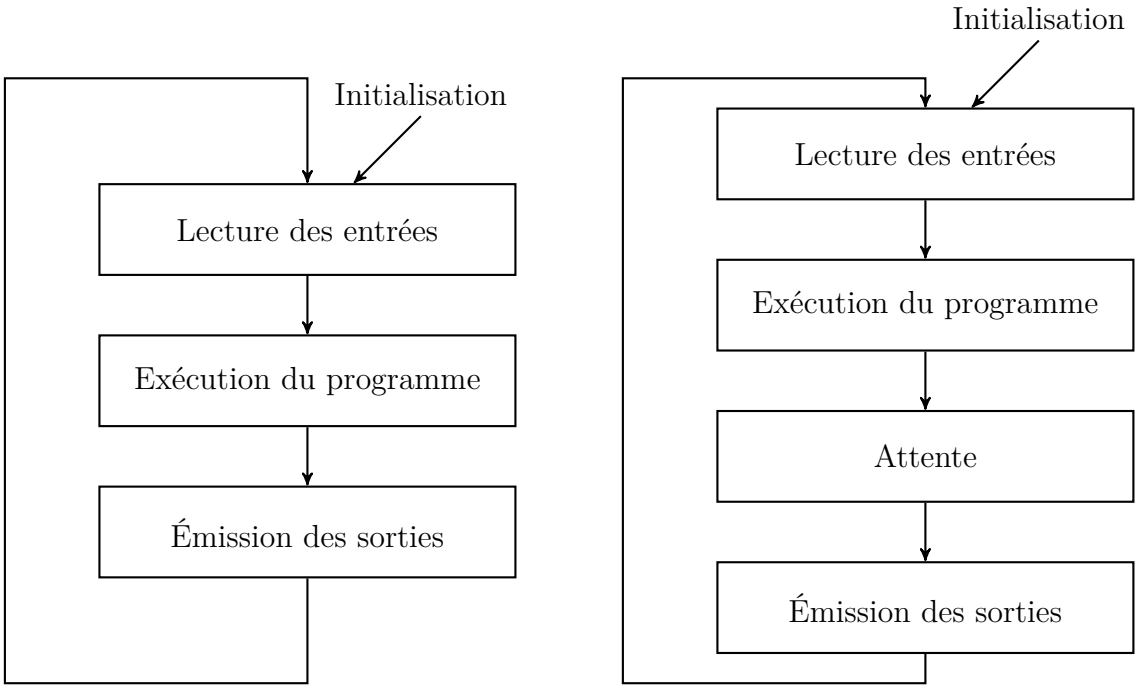
Le fonctionnement d'un API consiste en l'exécution *séquentielle* de plusieurs opérations. L'exécution de ces opérations est contrôlée par un *moniteur d'exécution* temps-réel qui peut être mono-tâche ou multi-tâches ; dans le cadre de ces travaux, seul le cas d'un API avec un moniteur d'exécution mono-tâche est considéré.

Les deux modes de fonctionnement principaux d'un API sont le mode cyclique et le mode périodique. En mode de fonctionnement cyclique, l'API exécute le programme puis recommence dès que celui-ci est terminé, tandis qu'en mode de fonctionnement périodique, l'API exécute le programme puis attend que la période définie pour son temps de cycle soit écoulée avant d'effectuer une nouvelle exécution du programme. Le mode d'exécution cyclique permet de garantir un meilleur temps de réponse, tandis que le mode d'exécution périodique assure la régularité d'exécution du programme.

Dans le cas du mode d'exécution cyclique, le cycle d'un API est constitué de trois opérations principales, exécutées séquentiellement dans l'ordre de présentation ci-dessous :

- Lecture (ou scrutation) des entrées : lecture des informations des capteurs de la partie opérative et mise en mémoire de ces informations via une table des variables. Les valeurs des variables d'entrée sont conservées jusqu'à la prochaine phase de lecture des entrées.
- Traitement du programme utilisateur : exécution du programme embarqué. Le programme en mémoire est exécuté une seule fois. Cette exécution conduit à la mise à jour des variables internes et des variables de sortie en fonction des valeurs des variables d'entrée.
- Écriture (ou mise à jour) des sorties : écriture des valeurs des variables de sorties sur les borniers de sortie de l'API à destination de la partie opérative. Les valeurs de ces sorties sont conservées et restent émises jusqu'à la prochaine phase d'écriture des sorties.

Lorsqu'un cycle est terminé, l'API reprend alors un nouveau cycle. Dans le cas d'un fonctionnement périodique, une phase d'attente est insérée entre la phase d'exécution du programme et la phase d'écriture des sorties. La figure 1.4 illustre les cycles API suivant ces deux modes d'exécution .



(a) Exécution cyclique (b) Exécution périodique

Figure 1.4 – Cycles du moniteur d'exécution d'un API

Ce type de fonctionnement caractérise donc un moniteur à scrutation cyclique ou périodique des entrées, dont le temps de cycle (temps entre deux mises à jour consécutives

des sorties) doit être adapté à l'application. Selon les applications et les performances de l'API, le temps de cycle varie de quelques millisecondes à quelques dizaines de millisecondes.

Le fonctionnement global d'un API dépend donc du code utilisateur (applicatif) programmé dans l'API et du mode d'exécution du moniteur temps-réel à scrutation cyclique.

Pour les applications les plus critiques, les SCC sont pilotés par des Automates Programmables Industriels dédiés à la Sécurité (APIdS). Le fonctionnement de ces contrôleurs est semblable à celui des API standard, mais la structure matérielle des APIdS est composée d'éléments redondants afin d'accroitre la fiabilité de ces contrôleurs. Les expérimentations réalisées durant cette thèse ont été effectuées sur des API standard, mais les résultats obtenus peuvent être étendus aux APIdS. Ce point sera détaillé dans la section 7 du chapitre 3.

2 Développement des systèmes de contrôle/commande : vérification, validation et test

Comme pour tout produit, le cycle de vie d'un SCC peut être découpé en plusieurs phases, de la définition des besoins à son arrêt et recyclage.

La figure 1.5 donne une représentation condensée du cycle de vie d'un SCC. Ce cycle de vie peut être séparé en deux grandes phases : une phase hors ligne et une phase en ligne. La phase hors ligne, aussi appelée phase de développement, est constituée des différentes phases élémentaires de spécification et de réalisation du SCC, tandis que la phase en ligne est constituée des phases de suivi, maintenance et mise à jour.

Dans le cadre de ces travaux, le test de conformité s'inscrit plutôt dans la phase de développement hors-ligne, même si ce type de test peut être utilisé lors de mises à jour ou de mises à niveau d'une partie d'un SCC. Les sections suivantes précisent la composition du cycle de développement et la place du test de conformité dans la phase de développement d'un SCC ainsi que sa complémentarité par rapport aux autres méthodes de Vérification et Validation (V&V).

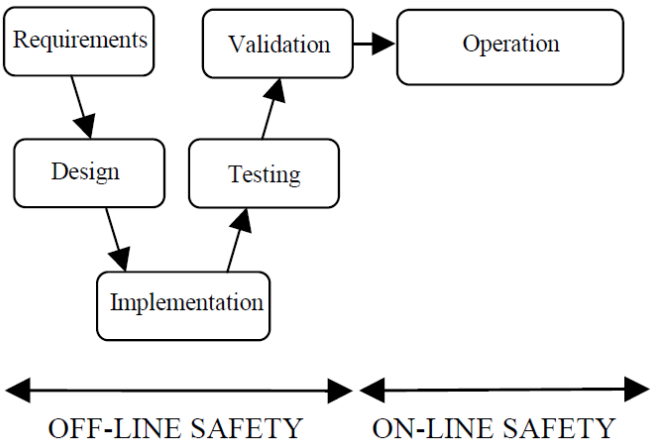


Figure 1.5 – Cycle de vie d'un SCC (Faure et Lesage (2001))

2.1 Cycle de développement des systèmes de contrôle/commande

Le cycle de développement d'un SCC correspond aux premières phases — phases hors ligne — du cycle de vie de ce SCC. Le cycle de développement est constitué des différentes phases depuis la définition des besoins jusqu'à la mise en service du SCC. Le découpage du cycle de développement en plusieurs phases permet la définition de jalons intermédiaires, jalons pouvant être utilisés à la fois pour définir le calendrier du développement de ce produit, mais également pour définir les frontières de chaque phase de V&V.

Ce découpage en phases et la définition de jalons intermédiaires sont également dus au constat suivant : "plus les erreurs sont introduites tôt et détectées tard, plus leur coût sur le produit final est élevé". La figure 1.6 illustre le coût de correction d'un défaut en fonction de ses moments d'introduction et de détection. La définition du cycle de développement et le découpage du cycle de développement en plusieurs phases auxquelles sont associées des méthodes de V&V a donc pour objectif de permettre la détection au plus tôt des erreurs afin de maîtriser la qualité du produit, les délais de réalisation ainsi que les coûts associés.

Le modèle de cycle en V est un des modèles proposés pour représenter le cycle de développement d'un SCC. D'autres modèles (cascade, spirale...) sont présentés dans Patterson, Jr. (2009), la représentation par un modèle en V est ici choisie car elle permet de positionner plus facilement chacune des phases usuelles de V&V. La figure 1.7 donne une représentation synthétique de ce cycle en V dans un cadre général, tandis que la

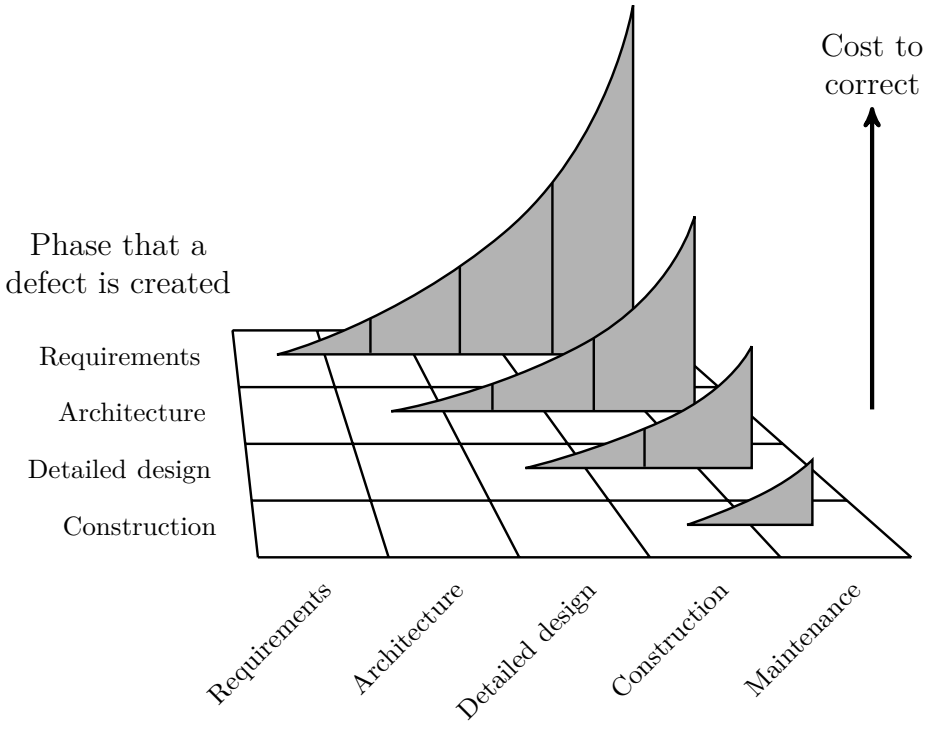


Figure 1.6 – Évolution du coût de correction d'un défaut (McConnell (1997))

figure 1.8 illustre de façon détaillée les différentes phases du développement d'un SCC.

Dans le modèle de cycle de développement en V, les phases descendantes — phases de la branche gauche — représentent la spécification, la conception et la réalisation des composants, tandis que les phases ascendantes — phases de la branche droite — représentent les phases d'intégration et de test progressifs.

La première des phases de la partie descendante est la définition des besoins attendus du produit. Cette définition du besoin, très générale, indique les attentes de l'utilisateur final : il veut un produit dont il sera satisfait par les performances. En appliquant une démarche d'"Analyse de la Valeur", des critères subjectifs sont d'abord qualifiés et quantifiés afin de distinguer les fonctions de service du système (utilités) de ses fonctions techniques (technologies). Puis, la définition de ces fonctions de service et techniques donneront naissance aux propriétés attendues du produit. Ensuite, ces propriétés serviront de support à la définition de spécifications générales puis détaillées.

Comme l'illustre la méthodologie de conception et d'implantation de contrôleurs logiques proposée dans Zaytoon (1996), les langages et formalismes utilisés pour définir les exigences fonctionnelles (propriétés, ... , spécifications détaillées) diffèrent en fonction du positionnement dans le cycle de développement. Ces langages et formalismes sont choisis

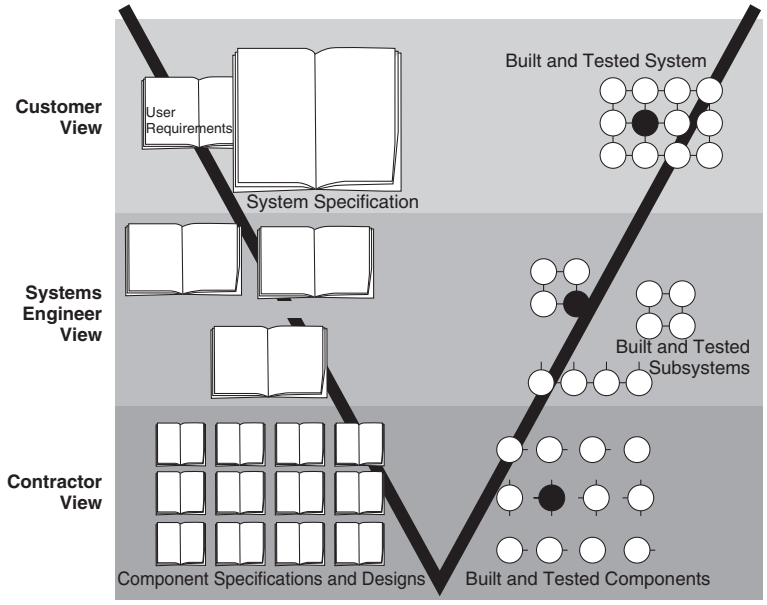


Figure 1.7 – Cycle en V et outils de développement d'un SCC (Patterson, Jr. (2009))

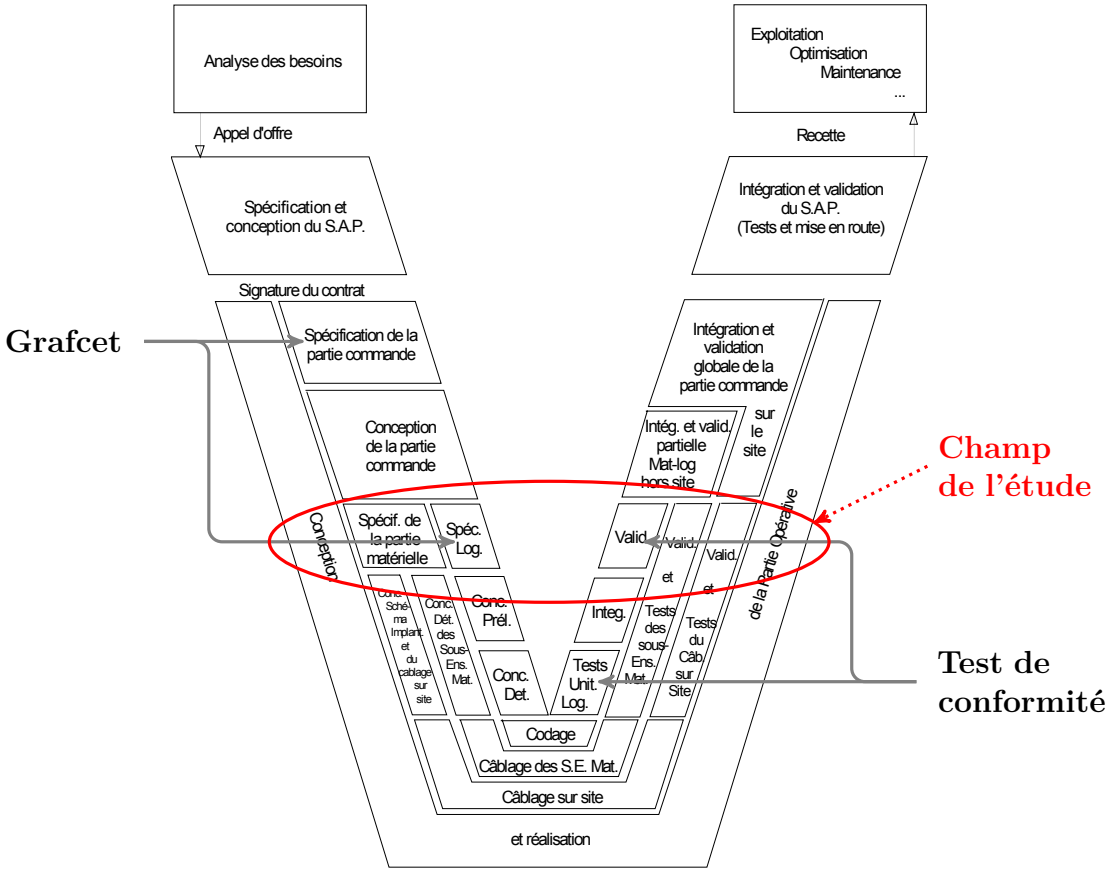


Figure 1.8 – Cycle en V détaillé du développement d'un SCC (Denis (1994))

en fonction de leur capacité d'expression et d'analyse et doivent être compréhensibles par les ingénieurs et techniciens automaticiens. C'est pourquoi, les formalismes choisis dans ces phases de conception sont aussi appelés "langages métier". Aussi, les langages métier choisis dépendent de la phase de conception ; aucun langage n'étant universel, chaque langage est choisi en fonction de son pouvoir d'expression et de sa facilité d'utilisation et de compréhension pour une phase donnée. Parmi les langages métier utilisés, on peut citer les exemples suivants :

- UML ou SysML : les diagrammes états-transitions et d'activité peuvent être utilisés pour décrire le fonctionnement des SCC ;
- Les matrices de sécurité : elles permettent d'exprimer des conditions d'activation d'action de protection (ou déclenchement d'alarme). Les lignes de la matrice correspondent aux causes, et les colonnes aux actions correctives à entreprendre. La relation de cause à effet est désignée par les intersections lignes/colonnes.
- Le Grafcet : c'est un langage de spécification graphique permettant de décrire le comportement séquentiel d'un système. Ce langage est basé sur la succession d'étapes et de transitions.
- Les Diagrammes Fonctionnels Logiques (DFL) : utilisés par EDF, ces diagrammes utilisent une représentation graphique de portes logiques. Il permet de décrire simplement des conditions d'activation d'action de protection.

Le Grafcet (IEC 60848 (2002)) est un modèle de spécification qui peut être utilisé pour définir aussi bien des spécifications générales que détaillées ; différents concepts de structuration seront détaillés dans la section 2.4 du chapitre 2.

Enfin, à partir des spécifications détaillées, les composants élémentaires peuvent être choisis (choix des composants matériels) ou réalisés (codage des programmes). Comme pour la conception des spécifications, des règles métier peuvent également être définies afin de faciliter la vérification des programmes codés, par exemple le document Cnomo (2003) énonce les règles de conception et de réalisation des systèmes automatisés pour le groupe PSA Peugeot Citroën.

Dans le cadre de ces travaux, le composant retenu pour réaliser la commande du SCC est un API, le programme exécuté par cet API sera généré, depuis la spécification Grafcet, dans un des langages de la norme IEC 61131-3 (2003).

Ces phases descendantes peuvent être automatisées ou manuelles. En effet, pour certaines tâches telles que le codage du programme, des outils permettent la génération

automatique de ce code à partir de description de plus haut niveau. En revanche, pour certaines tâches telles que la définition des propriétés, le choix de l'architecture du SCC ou la répartition des composants matériels et logiciels, une approche experte est nécessaire afin de résoudre les nombreux compromis.

Les composants élémentaires étant choisis et réalisés, les phases de la partie ascendante permettent l'intégration successive de ces composants en modules, sous-systèmes, puis l'installation et la mise en route du SCC global.

Le modèle de cycle en V met en vis-à-vis les phases descendante et ascendante ; une phase ascendante Y a pour but de produire un artefact conforme au modèle défini à la phase descendante X en vis-à-vis. Selon le niveau de ces phases, la conformité peut être définie par rapport aux données, au comportement ou à l'organisation. Ainsi, la définition des frontières de chacune des phases descendantes permet de préciser les attendus des futures phases ascendantes.

2.2 Vérification et validation

2.2.1 Quelques définitions

Avant de comparer les différents objectifs et méthodes des phases de vérification, validation et test, nous rappelons les définitions de ces termes de vocabulaire, tels que définis dans la norme IEEE 1012 (2004) pour le développement de logiciel, mais également applicables dans le cas d'un SCC :

Vérification : The verification process provides objective evidence whether the software and its associated products and processes :

a) Conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance)

b) Satisfy standards, practices, and conventions during life cycle processes

c) Successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly)

Validation : The validation process provides evidence whether the software and its associated products and processes :

a) Satisfy system requirements allocated to software at the end of each life cycle activity

b) Solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions)

c) Satisfy intended use and user needs

Test : Test is a set of one or more set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

En résumé, et pour reprendre la définition des termes vérification et validation proposée dans Boehm (1979), la *vérification* est une méthode qui permet de s'*assurer que l'on fait bien le produit*, tandis que la *validation* est une méthode qui permet de s'*assurer que l'on fait le bon produit*. Le test est une méthode particulière permettant de valider ou de vérifier un produit.

Par exemple, pour le système de commande des 2 pompes présenté figure 1.9, le processus de vérification permet de garantir l'absence de blocage de type deadlock du système, tandis que le processus de validation permet de garantir qu'une demande en eau déclenche toujours le démarrage d'une des pompes.

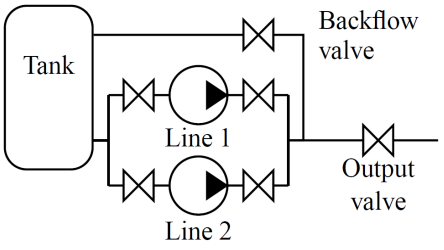


Figure 1.9 – Système de commande de 2 pompes (Roussel et Denis (2002))

2.2.2 Vérification et validation dans le cycle de développement

Les phases descendantes sont basées sur la manipulation de modèles qui sont de plus en plus détaillés selon l'avancement de la conception du SCC. À l'opposé, les phases ascendantes représentent la réalisation du SCC, il s'agit alors de modules, sous-systèmes, systèmes réels. De ce fait, si le cycle en V met en évidence les phases de validation des phases ascendantes au regard de leurs attendus définis lors des phases descendantes, il est tout aussi important de préciser que des vérifications doivent (ou peuvent) être effectuées entre deux étapes de conception ou de réalisation successives (figure 1.10).

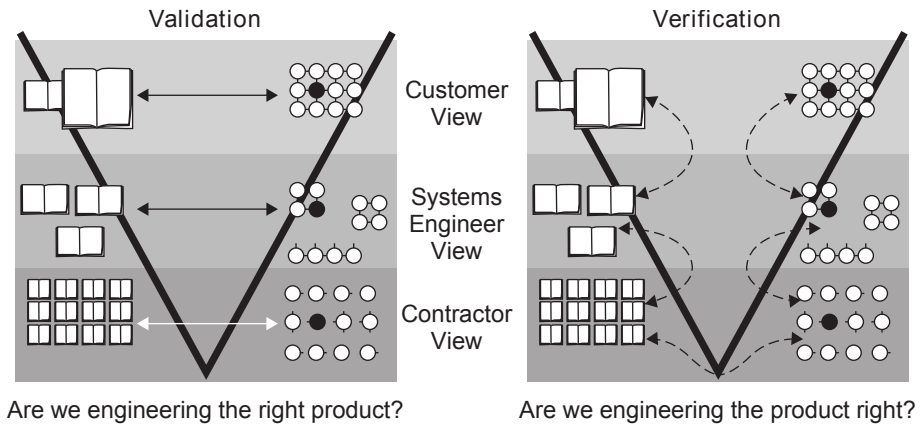


Figure 1.10 – Vérification et validation dans le cycle de développement (Patterson, Jr. (2009))

La validation, effectuée entre deux phases en vis-à-vis, permet de s'assurer de la conformité des attendus sur une réalisation. À l'opposé, la vérification, effectuée entre deux phases successives, permet de s'assurer que le raffinement de modèles ou l'intégration de modules conserve les propriétés obtenues à la phase précédente. Les phases de validation entre deux phases en vis-à-vis permettent donc de détecter des erreurs sur la réalisation, tandis que les phases de vérification entre deux phases successives permettent de détecter des erreurs, au plus tôt (dans l'idéal, aussitôt leur introduction). Les techniques de validation transversales sont communément appelés "tests". Les différentes techniques de test couramment utilisées seront détaillées dans la section suivante.

Le recours aux techniques de V&V entre deux phases descendantes est d'autant plus nécessaire lorsque plusieurs phases de raffinement successives sont nécessaires à la conception du SCC. De plus, lors du passage par des phases de conception "manuelles", la correction du SCC final vis-à-vis des besoins initiaux ne peut être garantie sans utilisation des différents techniques de V&V.

2.2.2.1 Vérification et validation dans la phase descendante

Afin de valider les attentes à l'issue de ces différentes phases de raffinement successives, des méthodes complémentaires de V&V ont été développées, parmi lesquelles on peut citer, par exemple :

- la relecture de code (par un tiers) ;
- l'analyse statique de code ;
- la vérification formelle (par model-checking par exemple) ;

– la simulation.

Ces différentes techniques de V&V permettent de s'assurer de la correction des modèles conçus ; ou plutôt de s'en persuader au regard des hypothèses effectuées, aussi bien lors de la définition des besoins que pour la mise en œuvre de ces différentes techniques de V&V. La complémentarité entre ces différentes techniques de V&V permet donc de vérifier par une technique A que les hypothèses nécessaires à l'application d'une technique B sont respectées. Par exemple, une analyse statique de code permet de vérifier les dépendances entre entrées et sorties de ce code, ce qui permet de garantir que le domaine de variation des entrées étudié lors d'une phase de vérification inclut bien l'ensemble du domaine auquel ce code est sensible.

Ces activités représentent une part importante de l'effort fourni et du temps passé pour le développement du produit (50% dans le cas d'un projet automobile (Awedikian (2009))).

Les étapes de relecture ou de confrontation peuvent être placées à différents niveaux. Ces étapes permettent de repérer certaines erreurs grossières et sont l'occasion d'apporter un œil extérieur sur ce qui a été réalisé. Cette relecture "manuelle" est complémentaire aux approches automatiques dans la mesure où certains types d'erreurs sont plus facilement détectables par un être humain que par un programme, et vice-versa.

La vérification formelle par model-checking permet de prouver qu'un modèle formel, représenté par un système de transitions ou par un réseau d'automates communicants par exemple, satisfait les propriétés attendues pour ce modèle. Les propriétés à vérifier peuvent être des propriétés d'atteignabilité, de sûreté, de vivacité et d'équité.

La simulation repose sur l'étude du parcours de l'espace d'états d'un modèle en suivant un scenario défini par une séquence d'occurrences d'évènements d'entrées. La simulation permet de vérifier si les propriétés dans le cahier des charges sont vérifiées lors de la simulation du scenario.

Les propriétés vérifiées par vérification formelle par model-checking ou par simulation peuvent porter sur le comportement intrinsèque du produit conçu, ou bien sur son comportement extrinsèque (comportement vis-à-vis de ses entrées/sorties). La vérification par model-checking est en général utilisée pour vérifier des propriétés sur l'ensemble de l'espace d'états décrit par le modèle et permet, par exemple de vérifier une propriété d'atteignabilité ou d'absence de blocage de type deadlock, tandis que la simulation repose sur la simulation de l'exécution d'un scenario sur un modèle.

De nombreux travaux ont été effectués au LURPA dans le domaine de l'amélioration de la sûreté de fonctionnement des SCC. Ces travaux traitent des différentes méthodes de V&V, de la définition de propriétés formelles (Barragan Santiago (2007)) à l'évaluation des performances d'architectures de SCC en réseau (Ruel (2009)) ; certains s'intéressent plus particulièrement à la formalisation de modèle de défaillance (Merle (2010)), à la synthèse algébrique de contrôleurs (Hietter (2009)), à la validation de code d'implantation (Rossi (2003)), ou à la vérification de propriétés d'un Système Dynamique Hybride (SDH) en boucle fermée (Juárez-Orozco (2008)), ainsi qu'à leur application dans un contexte industriel (Gourcuff (2007) ; Limal (2009)).

Les travaux présentés dans ce mémoire sont les premiers travaux effectués au LURPA dans le domaine du test de conformité de contrôleurs logiques et élargissent ainsi le spectre de compétences du LURPA dans le domaine de la sûreté de fonctionnement de Systèmes à Évènements Discrets (SED). Le positionnement de ces travaux par rapport aux travaux existants dans le domaine du test de conformité sera présenté dans les sections suivantes.

2.2.2.2 Vérification et validation dans la phase ascendante

Parmi les nombreuses méthodes de V&V, le test reste une méthode de validation incontournable dans le cas de systèmes critiques. En effet, les méthodes de V&V présentées précédemment s'effectuent sur des modèles. Ces modèles, qui peuvent être construits par une approche experte, permettent de représenter un SCC connecté à un environnement nominal incluant éventuellement des modèles de fautes importantes ou fréquentes. En revanche, la vérification ne s'effectue pas sur un SCC réel, connecté dans un environnement pouvant comporter des erreurs peu probables, non prévues dans le modèle de fautes utilisé pour la vérification. Seules les phases de validation transversales permettent de garantir la conformité du produit **réalisé** au besoin exprimé. Pour les systèmes les plus critiques, dans le nucléaire, la Règle Fondamentale de Sûreté RFS II.4.1.a pour les logiciels (Autorité de Sûreté Nucléaire (2000)) précise, à propos de la preuve à apporter qu'un logiciel répond à ses spécifications : "Cet objectif ne peut pas être atteint uniquement par des essais [1], en dépit de son importance pour les systèmes programmés classés de sûreté. En effet, ceux-ci devraient solliciter le logiciel suivant son profil opérationnel, c'est-à-dire pour toutes les séquences de combinaisons de valeurs issues des spécifications du logiciel.

1. Ici, le terme "essais" est à interpréter dans le sens "tests"

En pratique, ceci est rarement réalisable, ce qui conduit à élargir la vérification à [...] l'évitement d'erreurs [...] l'élimination des erreurs [...] la tolérance des erreurs résiduelles éventuelles". Cette recommandation précise bien que les étapes de vérifications viennent en complément au test, et non pas l'inverse ; dans tous les cas le recours au test est nécessaire. De même, la norme NF EN 60880 (2010) propose une sélection de méthodes de vérification et de test, en mentionnant que ces méthodes sont complémentaires. Le plan de V&V du produit est élaboré sous la responsabilité de l'industriel fabriquant le produit. Ce plan de V&V est élaboré suivant son expérience et son savoir-faire, mais également selon certains critères obligatoires imposés par une autorité. Le plan de V&V doit expliciter les techniques et outils utilisés ainsi que les critères (et leur niveau) qui ont été choisis.

2.2.3 Présentation succincte des différentes techniques de test

Si le terme générique "test" désigne *un moyen de validation opérant sur une réalisation et non sur un modèle*, il existe de nombreuses techniques de test qui peuvent être classées suivant différents critères.

La figure 1.11, illustre la décomposition de l'ensemble de ces techniques de test suivant 3 axes. L'objectif de cette section est de présenter les principales caractéristiques de ces différentes techniques de test et de préciser le positionnement de nos travaux.

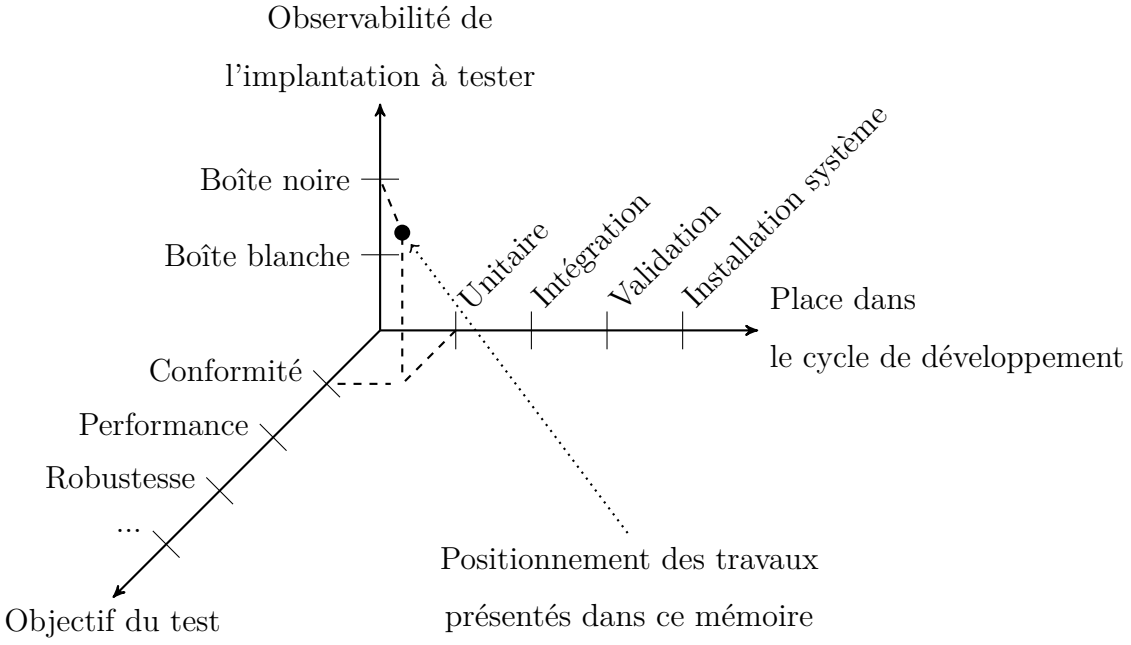


Figure 1.11 – Les 3 axes de définition du test (Tretmans (2010))

2.2.3.1 Place du test dans le cycle de développement

Selon le niveau où l'on se place dans la phase ascendante du cycle de développement, l'objectif des techniques de test employées ainsi que la réalisation qu'elles permettent de tester sont différents.

Du début de la phase ascendante à la fin de cette phase, on retrouve les phases de test suivantes :

- *test unitaire* : permet de tester la réalisation d'un module du plus bas niveau ;
- *test d'intégration* : permet de tester le comportement d'un sous-système suite à l'intégration d'un module ;
- *test de validation* : permet de tester l'ensemble d'un sous-système ;
- *test d'installation système* : permet de tester le système global, ses procédures de mise en route... Il s'agit du dernier test avant livraison du SCC, il permet de valider le respect des besoins initialement définis.

Afin de détecter les erreurs au plus tôt, il est impératif de réaliser des phases de test dès le plus bas niveau de la phase de réalisation. La réalisation de ces tests nécessite l'utilisation de plateformes (ou bancs) de test, ce qui engendre un coût supplémentaire, mais permet d'effectuer des opérations de test hors-ligne, avant de réaliser un essai sur site avec les risques que cela impliquerait. De plus, la mise en place de plateforme de test permet de tester des modules élémentaires, et par conséquent de réduire la taille du comportement à tester.

2.2.3.2 Objectif du test

Parmi les différents objectifs des techniques de test, on peut citer, par exemple, les objectifs suivants :

- *le test de conformité* : ce test vise à valider la conformité d'une implantation vis-à-vis du comportement fonctionnel décrit par sa spécification ;
- *le test de performances* : ce test vise à valider plus précisément les performances d'un SCC ;
- *le test de robustesse* : ce test vise à valider la robustesse d'un SCC, par exemple son comportement aux limites du comportement nominal.

2.2.3.3 Observabilité de l'implantation à tester

Enfin, le test peut être réalisé en boîte blanche ou en boîte noire. La différence entre ces deux critères de test réside dans le niveau d'observabilité de l'implantation. Si l'implantation peut être entièrement observée, le terme de test en boîte blanche est alors utilisé ; le terme de test en boîte noire désigne un test pour lequel le comportement de l'implantation ne peut être connu que par observation des entrées et sorties de cette implantation. Lorsqu'une partie de l'implantation peut être observée, par exemple afin de vérifier les hypothèses de mise en œuvre du test, le terme de test en boîte grise est alors utilisé.

Le test en boîte blanche, aussi appelé test structurel, permet de valider la structure ou architecture d'un programme, module ou système (connaissance des composants, des algorithmes). Le test en boîte noire, aussi appelé test fonctionnel, permet de valider le respect des besoins exprimés dans les spécifications ou propriétés. Pour reprendre les définitions des termes "vérification" et "validation" proposées précédemment (partie 2.2.1), le test en boîte blanche se rapproche davantage des méthodes de vérification tandis que le test en boîte noire fait partie des méthodes de validation. Un test structurel permet de vérifier ce qu'une implantation fait et non pas ce qu'elle doit faire, tandis qu'un test fonctionnel permet de valider ce qu'une implantation doit faire et non pas ce qu'elle fait.

Test en boîte blanche et test en boîte noire doivent être combinés car les tests en boîte blanche (tests structurels) ne permettent pas de détecter les divergences par rapport aux spécifications et les tests en boîte noire (tests fonctionnels) ne permettent pas de détecter les défauts dus à la mauvaise structure d'un programme ou d'une carte électronique par exemple. Cette combinaison permet de tester ce que l'implantation fait et sa conformité par rapport à ce qu'elle doit faire.

Une méthode de test donnée peut donc recevoir différents qualificatifs selon les critères de classification donnés ci-dessus, par exemple : test de conformité unitaire en boîte noire, test de robustesse en boîte grise lors de l'intégration.

2.2.3.4 Positionnement des travaux présentés

Ce mémoire s'intéresse à l'étude du test de conformité unitaire en boîte noire de composants de SCC industriels de type API ou ECU.

3 Test de conformité de contrôleurs logiques et test de circuits intégrés et de logiciels

À notre connaissance, peu de travaux se sont intéressés au test de conformité de contrôleurs logiques. En revanche, de nombreux travaux ont permis de développer des techniques de test de conformité dans deux domaines distincts : le test de circuits intégrés d'une part, et le test de logiciel ou de protocoles de communication d'autre part. Si le principe et les objectifs globaux des techniques de test de conformité restent les mêmes, il est cependant important d'insister sur les points de divergence entre ces deux domaines. En effet, dans le cadre de cette thèse, nous nous intéressons au test de conformité de contrôleurs logiques, parmi lesquels nous étudierons plus particulièrement le cas des automates programmables industriels (API).

Comme précisé dans la section 1.2, les API sont composés de modules d'entrées/sorties physiques permettant d'observer et de contrôler des signaux électriques d'entrées et de sorties, et d'un module de base intégrant un moniteur temps-réel et exécutant un programme embarqué. De ce fait, le test d'une implantation réalisée par un API se situe à l'interface entre les deux domaines du test logiciel et du test de circuits intégrés. Il existe cependant peu de travaux sur le test de conformité d'API.

3.1 Modèles de fautes

La première différence entre le test de circuits intégrés et le test de logiciel réside dans la définition des modèles de fautes. Dans le cas de composants physiques comme les circuits intégrés les fautes potentielles sont des *fautes aléatoires*, alors que dans le cas de logiciels les fautes potentielles sont des *fautes systématiques*.

Afin de valider le comportement d'un circuit intégré, le test doit être effectué individuellement sur chaque circuit (test unitaire). Les modèles de fautes les plus courants associés à la technologie électronique sont les suivants (Galiay *et al.* (1980) ; Hulgaard *et al.* (1995)) :

- collages ("stuck-at-1" et "stuck-at-0") ;
- court-circuits.

Les court-circuits sont dus à l'interconnexion électrique entre deux zones proches d'un circuit, tandis que les collages sont dus à un dysfonctionnement (permanent ou

non) d'un composant électronique conduisant au blocage de celui-ci à un niveau 0 ou 1. Ces défauts peuvent être présents dès la réalisation du circuit ou apparaitre avec le vieillissement de ce circuit, suite à des perturbations électromagnétiques, à un choc...

Dans le cas de la production de circuits intégrés à haut niveau d'intégration (Very-Large-Scale Integration) (VLSI) en grande série, le test unitaire de chacune des puces se révèle très chronophage, c'est pourquoi des techniques d'auto-test (Built-In Self Test (BIST)) (David *et al.* (2002)) ont été développées afin de remédier à ce problème. Outre l'automatisation de l'exécution du test (effectuée par la puce elle même), ces techniques d'auto-test permettent d'effectuer des tests plus régulièrement (à chaque remise sous tension, par exemple).

À l'opposé, le comportement d'un logiciel ne nécessite pas l'exécution d'un test unitaire approfondi (une vérification de l'intégrité d'un logiciel copié ou téléchargé peut être effectué rapidement par comparaison avec l'empreinte d'origine, checksumMD5 par exemple). Afin de valider le comportement d'un logiciel ou d'un protocole, plusieurs modèles de fautes ont été définis ; la définition de ces modèles de fautes dépend du logiciel ou du protocole à tester. Parmi les différents modèles de fautes existants, on peut citer, par exemple pour le test de protocoles de communication, les fautes suivantes :

- perte de message ;
- duplication de message ;
- envoi de message erroné.

Cependant, la difficulté à définir un modèle de fautes représentatif de la réalité conduit souvent à la réalisation du test sans modèle de fautes détaillé ; le test est alors guidé par un modèle du logiciel (modèle structurel ou fonctionnel) (Daran (1996)). Deux modèles de fautes générales peuvent alors être retenus :

- fautes de transfert : une évolution de l'implantation à tester la conduit dans un état différent de celui défini par la spécification ;
- fautes de sortie : suite à une évolution de l'implantation à tester, celle-ci émet une sortie différente de celle définie par la spécification.

3.2 Test structurel et test fonctionnel

Les différentes techniques existantes peuvent être classées en deux grandes classes : le test structurel et le test fonctionnel.

Le test structurel s'intéresse à la structure de l'implantation et au respect des règles de spécification et d'implantation, tandis que le test fonctionnel s'intéresse davantage au comportement de l'implantation et au respect des propriétés fonctionnelles.

Dans le cas du test de circuits intégrés, une première phase de test permet de valider la définition de l'architecture interne du circuit à tester, (appelé masque du circuit). Cette phase de test structurel, en boîte blanche, permet de vérifier l'application de règles métier ainsi que les choix et connexions entre les différentes puces de ce circuit. L'architecture est ensuite considérée comme correcte. La seconde phase de test est le test fonctionnel unitaire effectué sur chaque circuit réalisé.

Dans le cas du test logiciel, le test structurel, effectué en boîte blanche, a pour objectif de vérifier la structure du logiciel (condition, boucle, syntaxe...). La mise en œuvre de ce test structurel nécessite de sélectionner des données d'entrée permettant de déclencher l'exécution de certains des chemins reliant le point d'entrée au point de sortie d'un graphe de contrôle ou d'un flot de données du code. Le test fonctionnel en boîte noire d'un logiciel consiste à tester le comportement d'une implantation de ce logiciel lors de son exécution afin de valider sa conformité par rapport à sa spécification. La nécessité de tester un logiciel en boîte noire est due au fait que pour des raisons de confidentialité, par exemple, les développeurs de logiciels ne fournissent en général pas le code de ce logiciel mais seulement son exécutable. Dans ce cas, le comportement de l'implantation logicielle ne peut être connu que par l'observation qui en est faite lors de son exécution.

3.3 Observabilité et contrôlabilité

L'avantage des méthodes de test structurel par rapport au test fonctionnel réside dans la réduction de la taille des séquences de vecteurs de test à utiliser pour atteindre un critère de couverture donné. En effet, le test exhaustif d'un circuit séquentiel nécessite $2^{(i+m)}$ vecteurs de test, où i est le nombre d'entrées et m le nombre de mémoires logiques du circuit. En revanche, le test structurel nécessite la connaissance de la structure de l'implantation. Or, dans la plupart des cas, il est impossible de contrôler et d'observer totalement le comportement interne du composant. L'observabilité d'une implantation est définie par sa capacité à être observée au travers de ses sorties ; sa contrôlabilité est définie par sa capacité à être contrôlée via ses entrées. Brièvement, plus l'implantation possède de sorties observables, plus le comportement de cette implantation est observable,

de même plus elle possède d'entrées contrôlables, plus elle est contrôlable.

Afin de répondre à ces problèmes d'observabilité et de contrôlabilité, des travaux ont été effectués sur l'ajout de Points de Contrôle et d'Observation (PCO) (Hayes et Friedman (1974) ; Gundlach et Muller-Glaser (1990)). La mise en place d'un PCO consiste à relier physiquement une connexion interne (sortie de bascule ou entrée de porte logique) à une des bornes du circuit intégré VLSI. L'ajout de PCO permet ainsi d'avoir accès à des variables internes au circuit et ainsi de faciliter la génération et la mise en mise en œuvre des séquences de test. Cependant, l'usage de PCO est limité car il augmente l'encombrement des circuit intégrés VLSI, ce qui va à l'encontre des objectifs de miniaturisation. Des travaux ont été effectués sur l'optimisation du nombre et de l'emplacement de ces points de contrôle et d'observation (Williams et Angell (1973) ; Schotten et Meyr (1995)). L'ajout de PCO combiné aux techniques d'auto-test (BIST) permet d'accéder plus facilement aux variables internes du circuit.

Le problème d'observabilité et de contrôlabilité des variables internes semble moins problématique dans le cadre de logiciel ou de protocoles. En effet, il est plus facile de rajouter des sorties (sur écran, dans des fichiers de log...), tout du moins pour le développeur du logiciel.

3.4 Critères de couverture

Il existe plusieurs objectifs de test à atteindre pour ces différentes techniques de test. Ces objectifs de test peuvent être définis par des critères de couverture.

Dans le cas d'un test structurel, ces critères de couverture reposent sur une quantification des éléments structurels testés par rapport à l'ensemble de ces éléments. Ces critères de couverture peuvent être classés par catégories et par niveau d'exigence croissant en fonction du nombre de cas de tests à réaliser.

Par exemple, pour le test d'un logiciel, les critères suivants peuvent être retenus :

- couverture des conditions (branchements, enchaînements) ;
- couverture des chemins.

Pour le test d'un circuit électronique intégré, les critères suivants peuvent être retenus :

- couverture des sorties ;
- couverture des portes logiques ;

Dans le cas d'un test fonctionnel, la définition de ces critères est plus difficile à définir ;

on peut par exemple définir les critères suivants :

- couverture des états décrits par la spécification ;
- couverture des entrées ;
- couverture de l'ensemble des séquences d'entrées depuis tous les états (complétude).

Sans hypothèse restrictive, l'exhaustivité d'un test fonctionnel est impossible à atteindre pour des systèmes de taille non triviale. Afin de réduire la taille des sous-systèmes à tester et pour permettre la vérification des hypothèses des techniques telles que le partitionnement d'une spécification en fonction de ses dépendances entrées/sorties (Cohen *et al.* (1984)), le pairwise-testing (test de toutes les paires d'entrées logiques) (Cohen *et al.* (1996)) ou bien le test de vecteurs adjacents peuvent être utilisées pour améliorer le taux de défauts détectés (David *et al.* (2002)).

3.5 Caractéristiques retenues pour le test de conformité de contrôleurs logiques

Les caractéristiques qui ont été retenues pour le développement d'une méthode de génération et d'exécution d'une séquence de test sont les suivantes :

- les fautes sont systématiques (algorithme de commande implanté dans l'API erroné) ;
- le test effectué est un test fonctionnel ;
- l'implantation à tester est observable au travers de ses sorties logiques et contrôlable via ses entrées logiques ;
- deux taux de couverture seront utilisés : la *complétude* par rapport à la spécification, et le *taux de couverture SIC* garantissant l'absence d'erreur de verdict lors de l'exécution du test.

La technique de test développée pour tester un contrôleur logique pilotant un SCC se situe donc à l'interface entre les techniques de test des deux types d'implantation présentés ci-dessus. En effet, si la contrôlabilité et l'observabilité d'un API sont comparables à celles d'un circuit intégré VLSI, un API exécute un programme embarqué qui, dû aux erreurs de programmation, est la première source de non-conformité du SCC par rapport à sa spécification.

La définition du premier taux de couverture (complétude) est défini uniquement par rapport à la spécification, tandis que le second (taux de couverture garanti sans erreur

de verdict) a été développé suite à des résultats d'expérimentations.

4 Les modèles formels utilisés pour le test de conformité

Comme mentionné dans la section 1, pour beaucoup d'installations industrielles, les traitements logiques (mise en service, verrouillage, déclenchement d'alarme) ont un degré de criticité beaucoup plus important que les opérations de régulation. De plus, la première préoccupation lors de la validation de la correction du système vis-à-vis de ses propriétés est la validation des propriétés fonctionnelles logiques, la validation des propriétés relatives aux aspects temporisés arrivant en second plan, une fois les propriétés fonctionnelles non-temporisées vérifiées.

Par conséquent, nous nous limiterons à l'étude des systèmes *logiques*, *non temporisés*.

Les modèles utilisés pour le test de conformité d'un contrôleur logique doivent permettre la prise en compte d'entrées et de sorties car le test de conformité étant un test en boîte noire, seules les entrées et les sorties du contrôleur logiques sont respectivement contrôlables et observables.

Les deux modèles formels les plus utilisés pour cette classe de systèmes sont les systèmes de transitions étiquetées et les machines de Mealy (Broy *et al.* (2005)). Des travaux plus récents ont étudié la génération automatique de séquence de test à partir de diagrammes UML (state-charts) (Massink *et al.* (2006)) ou d'une classe particulière de réseaux de Petri (réseaux de Petri k-saufs) (von Bochmann et Jourdan (2009)).

4.1 Les systèmes de transitions étiquetées à entrée/sortie

Les systèmes de transitions étiquetées à entrée/sortie (Input/Output Labelled Transition Systems) (IOLTS) sont une classe d'automates utilisés pour décrire un processus séquentiel (le déroulement d'un calcul ou un protocole de communication, par exemple) (Tretmans (1996) ; Jéron (2004)). Le modèle des IOLTS est un raffinement du modèle des systèmes de transitions étiquetées (Labelled Transition Systems) (LTS) (Arnold et Nivat (1982) ; Brookes *et al.* (1984)), afin de permettre la prise en compte d'évènements externes d'entrée et de sortie. Les LTS sont des modèles qui permettent de définir un modèle déterministe ou non-déterministe : une même entrée peut être étiquetée sur une

ou plusieurs transitions partant d'un même état. Une bibliographie annotée sur le test à partir de LTS est disponible dans Brinksma et Tretmans (2000). Les IOLTS sont des systèmes de transitions qui, en plus des actions internes, sont dotés d'actions externes d'entrée et de sortie.

Ce modèle peut être décrit par un graphe orienté (figure 1.12), où les sommets et les arcs sont respectivement appelés états et transitions. Des noms sont associés aux états, tandis que des étiquettes sont associées aux transitions.

Formellement, un système de transition à entrée/sortie $IOLTS$ est un 4-uplet $(S, A, \rightarrow, s_{Init})$ tel que :

- S est un ensemble fini d'états ;
- $s_{Init} \in S$ est l'état initial ;
- A est l'alphabet des actions, partitionné en trois ensembles disjoints A_I, A_O et I où :
  - A_I est l'alphabet des actions d'entrée ;
  - A_O est l'alphabet des actions de sortie ;
  - I est l'alphabet des actions internes.
- $\rightarrow \subseteq S \times A \times S$ est la relation de transition.

Un IOLTS est déterministe si et seulement si il n'a aucune transition étiquetée par une action interne, et que dans chaque état s pour chaque action a au plus une transition issue de s est étiquetée a. Il est non-déterministe sinon.

La différence entre l'alphabet des actions d'entrée et l'alphabet des actions de sortie est fondamentale dans le cas du test. En effet, une action d'entrée permet de solliciter l'implantation à tester, tandis qu'une action de sortie est un évènement émis par cette implantation. Pour faciliter la représentation graphique, et afin de distinguer les différents alphabets, une action d'entrée $i \in A_I$ est notée $?i$ tandis qu'une action de sortie $o \in A_O$ est notée $!o$ et une action interne $x \in I$ est notée τ_x.

Une action d'entrée est contrôlable et une action de sortie est observable, tandis qu'une action interne n'est ni contrôlable, ni observable. L'ensemble des actions d'entrée et de sortie (actions contrôlables ou observables) définit l'ensemble A_{vis} des actions visibles ($A_{vis} = A_I \cup A_O$).

Plusieurs IOLTS peuvent interagir en parallèle ; il est possible de définir une composition parallèle de ces IOLTS en effectuant une synchronisation sur les actions communes.

La trace d'un IOLTS est une séquence finie d'actions visibles de cet IOLTS ; elle est

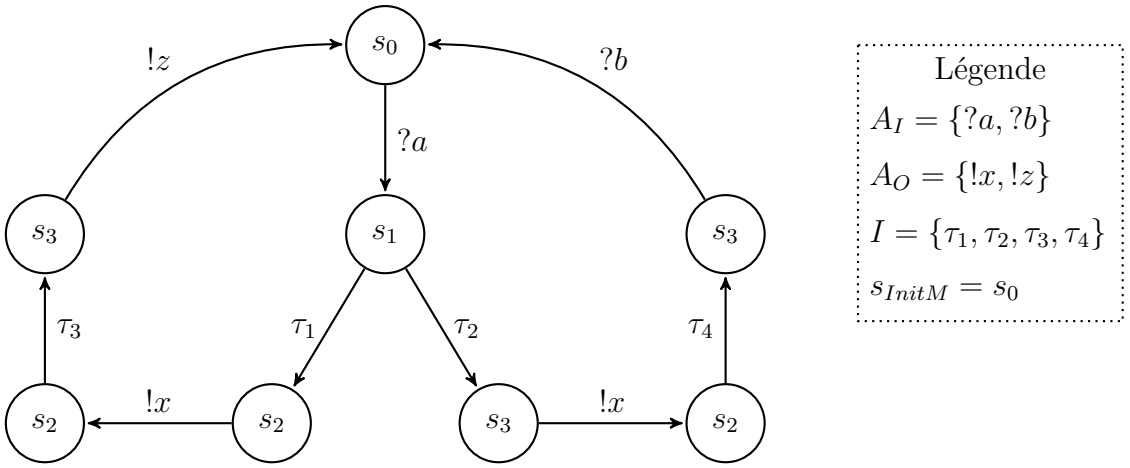


Figure 1.12 – Exemple d'IOLTS (Jéron (2004))

obtenue par projection des séquences d'actions sur les actions visibles.

Un IOLTS permet de représenter le comportement global d'une spécification ou d'une implantation : comportement intrinsèque, tandis que la trace d'un IOLTS permet de représenter le comportement visible d'une spécification ou d'une implantation : comportement extrinsèque.

Une relation de conformité permet de définir l'ensemble des implantations conformes à une spécification donnée. La définition d'une relation de conformité entre IOLTS s'appuie sur le comportement visible de l'implantation et de la spécification, elle peut donc être définie à partir d'une relation d'inclusion entre les traces des implantations *impl* et les traces de la spécification *spec*.

Une relation de conformité simple est le pré-ordre de trace $\leq_{tr}$ (Tretmans (1996)). Selon cette relation de conformité, une implantation *impl* est conforme à une spécification *spec* si l'ensemble des traces de l'implantation est inclus dans l'ensemble des traces de la spécification ($impl \leq_{tr} spec \Leftrightarrow traces(impl) \subseteq traces(spec)$).

La relation de conformité la plus couramment utilisée pour le test de conformité est la relation **ioco**, proposée par Tretmans (1996). Contrairement à la relation précédente, cette relation de conformité définit une relation d'inclusion, non plus entre les traces d'une implantation et de la spécification, mais entre l'ensemble des actions de sortie produits par une implantation et ceux de la spécification après une trace suspendue. Le lecteur pourra trouver une définition formelle de cette relation dans Tretmans (1996, 2008).

4.2 Les machines de Mealy

Les machines de Mealy sont des automates à états finis pour lesquels chaque transition est étiquetée par une entrée et une sortie. Nous présenterons ici les définitions de base sur les machines de Mealy. Plus de détails peuvent être trouvés dans la synthèse et la bibliographie annotée suivantes : Lee et Yannakakis (1996) et Petrenko (2000).

Formellement, une machine de Mealy déterministe M est un 6-uplet $(S_M, s_{InitM}, I_M, O_M, \delta_M, \lambda_M)$ tel que :

- S_M est un ensemble fini d'états ;
- $s_{InitM} \in S_M$ est l'état initial ;
- I_M est un alphabet fini d'entrées ;
- O_M est un alphabet fini de sorties ;
- $\delta_M : S_M \times I_M \to S_M$ est la fonction de transfert ;
- $\lambda_M : S_M \times I_M \to O_M$ est la fonction de sortie.

Contrairement aux systèmes de transitions à entrées/sorties, dont les transitions sont soit des entrées, soit des sorties, soit des actions internes, les transitions des machines de Mealy portent à la fois une entrée et une sortie. La figure 1.13 illustre une machine de Mealy minimale.

Les machines de Mealy sont déterministes si δ_M et λ_M sont définies par des fonctions, et sont complètement définies si δ_M et λ_M sont définies pour tout élément de $S_M \times I_M$. Comme pour les autres classes d'automates à états finis, il existe une notion d'équivalence entre deux états : deux états s_i et s_j d'une machine de Mealy M sont équivalents si et seulement si pour toute séquence d'entrée $\sigma \in I_M^*$, ils produisent les mêmes séquences de sorties. Une machine est dite minimale si et seulement si elle ne contient pas de paire d'états équivalents. Deux machines M_1 et M_2 ayant les mêmes alphabets sont dites équivalentes si et seulement si pour tout état de M_1 il existe un état équivalent dans M_2 et réciproquement.

Formellement :

- deux états s_i et s_j d'une machine de Mealy M sont équivalents si et seulement si :

$$\forall \sigma \in I_M^*, [\lambda_M^*(s_i, \sigma) = \lambda_M^*(s_j, \sigma)] \tag{1.1}$$

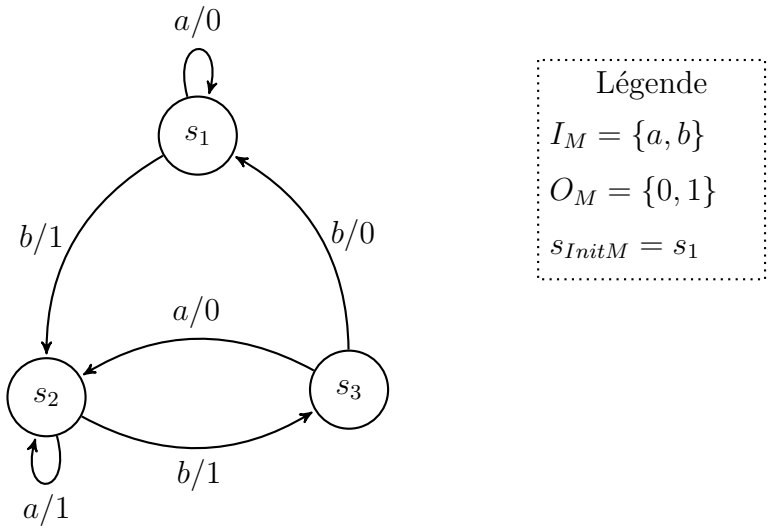


Figure 1.13 – Exemple de machine de Mealy minimale (Lee et Yannakakis (1996))

– une machine de Mealy M est minimale si et seulement si :

$$\forall (s_i, s_j) \in S_M^2, i \neq j,\ \forall \sigma \in I_M^*\ [\lambda_M^*(s_i, \sigma) \neq \lambda_M^*(s_j, \sigma)] \tag{1.2}$$

Si une machine de Mealy M n'est pas minimale, il est possible de construire une machine minimale équivalente, de façon similaire à la minimisation des automates (Hopcroft (1971)).

– deux machines de Mealy M_1 et M_2, de mêmes alphabets d'entrée I_M et de sortie O_M, sont équivalentes si et seulement si :

$$\begin{cases} \forall s_1 \in S_{M_1},\ \exists s_2 \in S_{M_2},\ \forall \sigma \in I_M^*\ [{\lambda_{M_1}}^*(s_1, \sigma) = {\lambda_{M_2}}^*(s_2, \sigma)] \\ \forall s_2 \in S_{M_2}),\ \exists s_1 \in S_{M_1},\ \forall \sigma \in I_M^*\ [{\lambda_{M_1}}^*(s_1, \sigma) = {\lambda_{M_2}}^*(s_2, \sigma)] \end{cases} \tag{1.3}$$

4.3 Choix d'un formalisme

Le modèle retenu pour ces travaux est le modèle des machines de Mealy. Ce choix est justifié par plusieurs aspects relatifs au test de conformité d'API.

Comme indiqué précédemment, les API sont des contrôleurs logiques opérant sur des valeurs de variables logiques (d'entrée et de sortie) obtenues à partir de signaux électriques. Ces signaux électriques sont reçus par la carte d'entrée et émis par la carte de sortie. La notion d'évènement n'est donc ici pas naturelle. En effet, un évènement, par exemple le front montant de l'entrée logique a (noté $\uparrow a$), est calculé par l'API à

partir des valeurs de la variable associée, ici l'entrée logique a, en début de cycle ($a[k]$) et en début de cycle précédent ($a[k-1]$). Par conséquent il est préférable d'utiliser un modèle permettant de manipuler les valeurs de ces signaux logiques plutôt que des évènements construits à partir de ces signaux. Cependant, les deux modèles formels présentés, majoritairement utilisés par la communauté informatique, manipulent des évènements, il sera donc nécessaire dans les deux cas d'effectuer une transformation permettant d'utiliser ces modèles.

Le fonctionnement cyclique (ou périodique) d'un API et l'absence d'éléments temporisés dans l'implantation impliquent qu'une variation des signaux de sortie est toujours la conséquence d'une variation des signaux d'entrée. L'utilisation d'une machine de Mealy permet donc de simplifier la représentation du comportement extrinsèque d'une implantation par API. Avec une machine de Mealy, le changement d'état d'un implantation suite à une lecture des entrées et l'émission des sorties correspondantes peut être représenté en utilisant une seule transition, à laquelle est associée le couple entrée/sortie correspondant. Avec un IOLTS ce même changement d'état nécessite l'utilisation d'une transition étiquetée par une action d'entrée, d'un état intermédiaire représentant l'exécution du programme, et d'une autre transition étiquetée par une action de sortie.

De plus, si le pouvoir d'expression des IOLTS est nettement supérieur à celui des machines de Mealy (prise en compte d'actions internes), ce pouvoir d'expression ne serait pas utilisé dans le cadre de ces travaux. En effet, le test de conformité étant effectué en boîte noire, seul le comportement extrinsèque de l'implantation est étudié.

La modélisation du comportement des spécifications et implantations par des machines de Mealy, bien que plus pauvre en terme de pouvoir d'expression, est donc simple à mettre en œuvre dans le cadre de ces travaux.

5 Objectifs des travaux présentés dans le mémoire

Comme indiqué précédemment, lors des phases de conception, les langages métier utilisés pour définir les exigences fonctionnelles (propriétés, ... , spécifications détaillées) sont choisis en fonction de leur capacité d'expression et doivent être compréhensibles par les ingénieurs et techniciens automaticiens. Cependant, ces langages métier ne sont pour la plupart pas dotés d'une sémantique formelle. La formalisation de ces langages est donc nécessaire afin de pouvoir appliquer les différentes méthodes de vérification

et de validation, aussi bien pour l'application de la vérification par model-checking que pour l'automatisation de la génération et de l'exécution de séquences de test. Or si le besoin industriel en terme de test est important, l'utilisation de spécifications formelles n'est pas toujours évidente. Certaines pratiques visent à persuader le milieu industriel de l'apport des méthodes formelles, il est également envisageable de s'adapter aux pratiques industrielles et de proposer des moyens permettant de formaliser les spécifications couramment utilisées. Cela permet ainsi de conserver les processus de développement habituels ainsi que le savoir-faire et l'expertise associés.

Le langage métier retenu dans le cadre de cette thèse est le langage Grafcet. Ce langage présente l'avantage d'être normalisé et ses règles d'interprétation sont clairement définies dans la norme IEC 60848 (2002). Contrairement au DFL, il est également plus largement utilisé. Quant à l'utilisation d'UML, elle est pour l'instant surtout développée dans le domaine informatique et dans une moindre mesure dans le domaine de l'automatique.

Les travaux présentés dans ce mémoire traiteront aussi bien d'aspects théoriques que d'aspects expérimentaux. En effet, nous ne pouvons concevoir de développer une théorie sans confronter les résultats obtenus à la réalité. C'est pourquoi des expérimentations ont été effectuées sur un banc de test relié à un API.

Le chapitre 2 de ce mémoire présente une méthode de formalisation du langage de spécification Grafcet. Ce chapitre propose une formalisation d'une spécification Grafcet sous forme d'un automate, dénommé Automate des Localités Stables (ALS) puis une méthode de traduction de cet automate en une machine de Mealy, plus adaptée à la génération automatique de séquence de test. L'originalité de cette méthode de formalisation réside dans la définition du concept de *localité stable* d'une spécification Grafcet. Le concept de localité permet de décrire le comportement de cette spécification d'un point de vue externe, nécessaire pour le test de conformité en boîte noire, en déterminant les valeurs des sorties pouvant être émises pour chaque état interne (situation) de la spécification.

Après avoir présenté les techniques de test de machine de Mealy, le chapitre 3 aborde la génération de séquence de test de façon automatique à partir de la machine de Mealy équivalente au Grafcet de départ. Ensuite, ce chapitre s'intéresse plus particulièrement à la mise en œuvre de ces séquences de test pour effectuer le test de conformité d'un API, et présente le banc de test développé pour la réalisation d'une campagne d'expérimentations mettant en évidence que des verdicts erronés peuvent être obtenus avec des séquences de

test construites par les approches conventionnelles.

Enfin, le chapitre 4 propose la définition d'une nouvelle propriété, qui peut être vérifiée sur une spécification Grafcet, et permet de qualifier la capacité d'une implantation de cette spécification à être testée sans erreurs de verdicts de test. Différentes méthodes de génération de séquences de test, permettant d'améliorer le taux de couverture de l'implantation par des pas de test garantis sans erreurs de verdict, sont enfin proposées afin d'améliorer le niveau de confiance d'une séquence de test.

Chapitre 2

Formalisation du comportement d'une spécification Grafcet

Sommaire

Introduction

Comme indiqué dans le chapitre précédent, les spécifications des Systèmes de Contrôle/-Commande (SCC) industriels sont souvent définies à partir de langages métier dont fait partie le langage de spécification Grafcet. Ces langages métier ne sont pour la plupart pas dotés d'une sémantique formelle. La formalisation du comportement d'une spécification Grafcet est donc nécessaire afin de pouvoir appliquer les différentes méthodes de vérification et de validation, et plus particulièrement de test de conformité.

Ce chapitre présente notre première contribution : la définition d'une méthode de *formalisation du langage de spécification Grafcet* et sa transcription en un *modèle adapté au test de conformité*.

Les travaux présentés dans ce chapitre ont fait l'objet de communications en conférences nationale et internationales (Provost *et al.* (2009b,a, 2011a)) ainsi qu'en revue internationale (Provost *et al.* (2011b)).

Ce chapitre est organisé comme suit :

- la première section présente les notations utilisées pour représenter et définir les variables logiques manipulées tout au long de ce mémoire ;
- la deuxième section rappelle les informations générales sur le Grafcet nécessaires à la compréhension de la démarche proposée ;
- la troisième section précise les hypothèses de travail utilisées et esquisse la méthode de formalisation d'une spécification Grafcet ;
- la quatrième et principale partie détaille la méthode de formalisation, après avoir donné les définitions formelles du Grafcet et de l'Automate des Localités Stables (ALS) ;
- enfin, la cinquième partie présente la méthode de transcription de cet ALS en une machine de Mealy, plus adaptée à la génération de séquences de test de conformité.

1 Notation et représentation des valeurs des signaux logiques

Dans ce mémoire, il sera souvent fait référence aux valeurs de signaux logiques d'une implantation ou aux valeurs des variables d'entrée et de sortie des modèles. Cette section a pour objectif la définition des différentes notations utilisées pour représenter ces signaux

et variables.

Une spécification Grafcet permet de spécifier le comportement de la partie commande d'un système au travers d'un modèle manipulant des variables logiques d'entrée et de sortie. Un Automate Programmable Industriel (API) est une implantation sollicitée par des signaux électriques via ses cartes d'entrées et qui émet des signaux électriques via ses cartes de sorties. Dans le cadre de ces travaux, les variables d'entrée et de sortie sont des variables logiques, il en est de même pour les signaux manipulés par l'API. La figure 2.1 illustre les notions de variables et de signaux logiques d'entrée et de sortie.

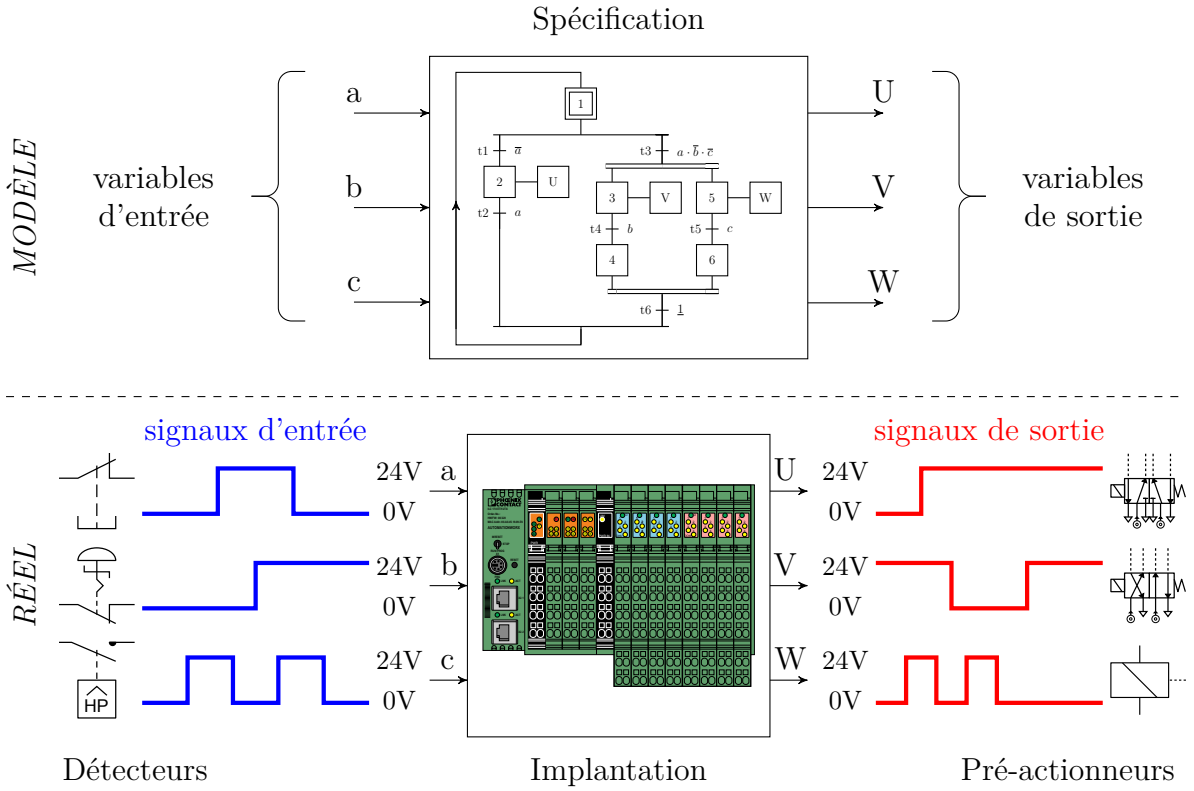


Figure 2.1 – Illustration de variables et de signaux logiques d'entrée et de sortie

Les valeurs des variables d'entrée de la spécification Grafcet étant l'image des valeurs logiques des signaux d'entrée de l'implantation (API), le terme "valuation d'entrée" sera utilisé pour désigner soit l'ensemble des valeurs des variables d'entrée de la spécification (Grafcet) ou des autres modèles (automates des localités stables, machine de Mealy), soit l'ensemble des valeurs des signaux logiques d'entrée de l'implantation (API) à un instant t. De la même manière, le terme "valuation de sortie" sera utilisé pour désigner soit l'ensemble des valeurs des variables de sortie du modèle, soit l'ensemble des valeurs des signaux logiques de sortie. Une valuation d'entrée sera notée v_I, une valuation de

sortie v_O.

En fonction des modèles utilisés, la manipulation des valuations d'entrée et de sortie nécessite différentes représentations. Ainsi, une valuation d'entrée ou de sortie pourra être représentée par un symbole, un minterme ou un ensemble des variables vraies, définis comme suit :

symbole : chaque valuation d'entrée v_I ou de sortie v_O peut être représentée par un symbole, noté $i_{indice}(v_I)$ ou $o_{indice}(v_O)$ avec $indice \in \mathbb{N}^+$. L'entier naturel *indice* est calculé à partir du poids affecté à chaque variable ou signal logique d'entrée. La première variable déclarée représente le poids fort (MSB : Most Significant Bit), la dernière le poids faible (LSB : Least Significant Bit). Cette représentation présente l'avantage d'être condensée et simple à écrire, mais est plus difficile à interpréter par un être humain.

minterme : chaque valuation d'entrée v_I ou de sortie v_O peut être représentée par un minterme, noté $m(v_I)$ ou $m(v_O)$. Un minterme est une conjonction de variables et de leur complément (expression booléenne utilisant uniquement l'opérateur de conjonction ET, noté $\cdot$, et l'opérateur complément, noté $\overline{\ }$). Ce minterme est construit à partir des variables dont la valeur est vrai (1 logique) et de la négation des variables dont la valeur est faux (0 logique).

ensemble des variables vraies : chaque valuation d'entrée v_I ou de sortie v_O peut être représentée par l'ensemble des variables vraies pour cette valuation, noté $\mathbb{1}(v_I)$. Cet ensemble ne contient que les variables dont la valeur est vrai (1 logique).

En utilisant ces définitions, et dans un souci de compréhension, une valuation (d'entrée ou de sortie) sera indifféremment représentée par son symbole, son minterne ou son ensemble des variables vraies. Par exemple, la définition d'un minterme peut être étendu au symbole et à l'ensemble des variables vraies d'une valuation :

$$m(v_I) \Leftrightarrow m(i_{indice}(v_I)) \Leftrightarrow m(\mathbb{1}(v_I))$$

Le tableau 2.1 donne les équivalences entre ces différentes représentations pour les valuations d'entrée de l'exemple illustré figure 2.1.

Valeurs des variables d'entrée			Représentation des valuations d'entrée		
a	b	c	symbole(v_I)	minterme(v_I)	$\mathbb{1}(v_I)$
0	0	0	i_0	$\overline{a} \cdot \overline{b} \cdot \overline{c}$	{}
0	0	1	i_1	$\overline{a} \cdot \overline{b} \cdot c$	{c}
0	1	0	i_2	$\overline{a} \cdot b \cdot \overline{c}$	{b}
0	1	1	i_3	$\overline{a} \cdot b \cdot c$	{b,c}
1	0	0	i_4	$a \cdot \overline{b} \cdot \overline{c}$	{a}
1	0	1	i_5	$a \cdot \overline{b} \cdot c$	{a,c}
1	1	0	i_6	$a \cdot b \cdot \overline{c}$	{a,b}
1	1	1	i_7	$a \cdot b \cdot c$	{a,b,c}

Tableau 2.1 – Équivalence entre les différentes représentations des valuations d'entrée

2 Informations générales sur le Grafcet

Le Grafcet est un langage de spécification normalisé (IEC 60848 (2002)) permettant de décrire le comportement de systèmes logiques séquentiels. Ce langage est très répandu dans divers domaines industriels, tels que le transport ferroviaire, la production d'électricité ou les systèmes manufacturiers. Le Grafcet permet de spécifier le comportement attendu d'un système de contrôle/commande relié à un système physique envoyant des signaux logiques au système de contrôle/commande et recevant les signaux logiques générés en retour. Le Grafcet a d'abord été normalisé en France au début des années 1980 (NF C03-190 (1982)), puis a fait l'objet d'une norme internationale en 1988 (IEC 60848 (1988)). Depuis, plusieurs extensions ont été proposées afin d'accroître ses possibilités de modélisation. Ces améliorations sont incluses dans la dernière version de la norme, révisée en 2002 (IEC 60848 (2002)). Une présentation détaillée des caractéristiques de chaque version de la norme Grafcet est disponible dans David (1995) et Guéguen et Bouteille (2001). Nous rappelons aussi que la norme IEC 60848 est bien différente de la norme IEC 61131-3, même si ces deux normes font référence, en anglais, au même terme "SFC" pour "Sequential Function Chart" et si leur représentation graphique semble similaire. Les différences entre ces deux langages normalisés résident aussi bien dans leur syntaxe que dans leur sémantique. De plus, ces deux normes ne sont également pas destinées au même usage, la norme IEC 60848 décrit un langage de spécification, tandis que la norme

IEC 61131-3 décrit un langage d'implantation. Les principales différences entre ces deux langages sont détaillées dans la section 2.5. Dans la suite de ce mémoire, comme nous traiterons de la formalisation de spécification, nous utiliserons donc le terme Grafcet et la norme associée IEC 60848.

Le Grafcet a été développé à partir de résultats issus de la communauté scientifique sur les réseaux de Petri (RdP) et plus particulièrement sur des travaux relatifs aux réseaux de Petri interprétés (RdPI) (Moalla (1981)). Une présentation détaillée des concepts et formalismes liant ces deux modèles a été proposée par David et Alla (David et Alla (1992a), David et Alla (1992b)). Une syntaxe et une sémantique spécifiques ont été définies pour le Grafcet, afin de prendre en compte les besoins spécifiques des ingénieurs en charge de la conception (au niveau de la définition de la spécification fonctionnelle) de systèmes séquentiels complexes. Les points clés concernant la syntaxe et la sémantique du Grafcet sont rappelés ci-après.

2.1 Syntaxe du langage Grafcet décrit dans la norme IEC 60848 (2002)

Une spécification Grafcet permet de décrire le comportement souhaité d'un contrôleur logique. Un Grafcet est composé d'étapes, représentées graphiquement par des carrés, et de transitions, représentées par des segments horizontaux. Une étape ne peut être liée qu'à des transitions, et une transition qu'à des étapes. Les liaisons entre étapes et transitions sont orientées. Par défaut, les liaisons orientées sont représentées de haut en bas. Lorsque la liaison est orientée vers le haut, une flèche doit être ajoutée sur cette liaison. Une flèche peut également être ajoutée sur toute liaison afin de faciliter la compréhension de la spécification.

Une étape définit un état partiel du système ; une étape peut être active ou inactive. De plus, une variable booléenne, appelée "variable d'activité" de l'étape peut être définie pour chaque étape. Des actions peuvent être associées à une étape ; une action associée à une étape est réalisée uniquement lorsque cette étape est active. Une action agit sur les variables de sortie. Une condition de transition doit être associée à chaque transition ; cette condition est une expression booléenne définie à partir des variables d'entrée, des variables d'activité d'étapes et de conditions temporisées.

Les concepts de structuration tels que les macro-étapes, les étapes encapsulantes et

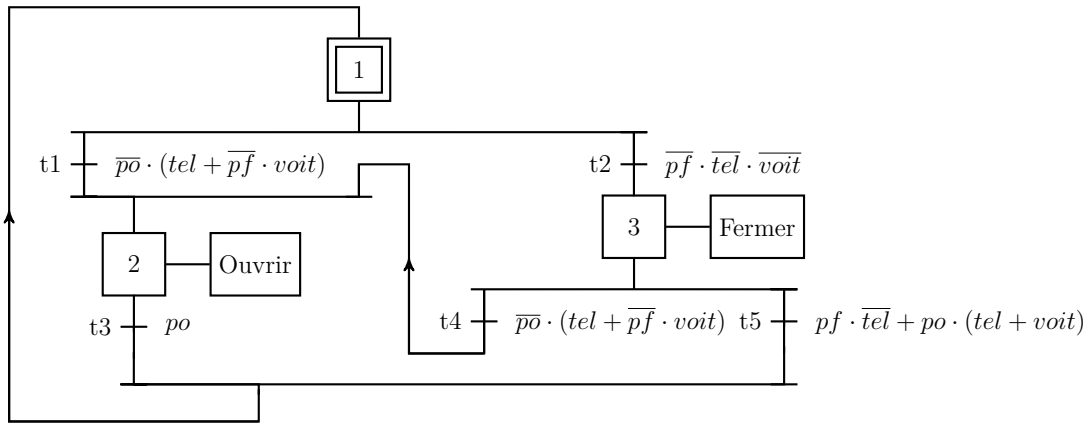


Figure 2.2 – Exemple de spécification Grafcet (Exemple A)

le forçage d'étapes ainsi que des concepts liés aux modes de sortie tels que les actions conditionnelles et mémorisées seront détaillés après la définition des règles d'évolution du Grafcet.

2.2 Règles d'évolution du langage Grafcet d'après la norme IEC 60848 (2002)

La définition complète du comportement de toute spécification Grafcet peut être obtenue par l'application des cinq règles d'évolution suivantes :

Règle 1 : *La situation initiale, choisie par le concepteur, est la situation à l'instant initial.*

À l'initialisation, toutes les étapes initiales (les étapes de la situation initiale, indiquées par un carré double) sont actives ; toutes les autres étapes sont inactives.

Règle 2 : *Une transition est dite validée lorsque toutes les étapes immédiatement précédentes reliées à cette transition sont actives. Le franchissement d'une transition se produit :*

– *lorsque la transition est validée,*

– *et que la réceptivité associée à cette transition est vraie.*

Une transition validée et dont la réceptivité associée est vraie est immédiatement franchie.

Règle 3 : *Le franchissement d'une transition entraîne simultanément l'activation de*

toutes les étapes immédiatement suivantes et la désactivation de toutes les étapes immédiatement précédentes.

Règle 4 : *Plusieurs transitions simultanément franchissables sont simultanément franchies.*

Règle 5 : *Si, au cours du fonctionnement, une étape active est simultanément activée et désactivée, alors elle reste active.*

Cette description textuelle des règles d'évolution du Grafcet telles que définies dans la norme IEC 60848 est suffisante pour la compréhension du comportement décrit et permet l'implantation d'algorithme d'interprétation, mais ne constitue pas une définition formelle. Une définition formelle du comportement d'une spécification Grafcet est nécessaire pour appliquer les méthodes automatisées de vérification, ainsi que pour le test. En effet, sans définition formelle, la description du comportement d'un Grafcet laisse beaucoup de place à l'interprétation, par exemple pour la recherche ou non de stabilité et pour la gestion des modes de sorties. C'est pourquoi, ce chapitre propose une méthode permettant la formalisation de ce langage, ce qui permettra de définir sans ambiguïté le comportement attendu d'une implantation lors de la réalisation du test de conformité.

Ces règles d'évolution montrent que l'état global d'une spécification Grafcet, appelé *situation* et caractérisé par l'ensemble des étapes actives d'un Grafcet à un instant donné, est défini par l'ensemble de toutes les étapes simultanément actives ; la situation initiale d'un Grafcet est définie par l'ensemble des étapes initiales du Grafcet. Par exemple, pour le Grafcet donné figure 2.7, la situation initiale est {A1,1}. Une évolution d'une situation vers une autre situation correspond au franchissement simultané des transitions franchissables, en accord avec les règles 2 à 4. Cette nouvelle situation peut être *instable* ou *stable*. Une situation est instable si au moins une des transitions du Grafcet peut être franchie depuis cette situation sans changement des variables d'entrées, et stable sinon. Ainsi, l'état global du Grafcet évolue de situation stable en situation stable, en passant éventuellement par des situations instables. Une évolution menant à une situation instable est qualifiée d'*évolution fugace*. Une évolution entre deux situations stables du Grafcet correspond à une séquence (éventuellement réduite à un seul élément) de franchissement d'ensembles de transitions simultanément franchissables ; cette évolution est instantanée.

Des exemples d'évolutions seront détaillés par la suite, après la définition formelle du comportement d'une spécification Grafcet.

2.3 Actions

Une action est représentée graphiquement par un rectangle relié au symbole de l'étape à laquelle elle est associée. Plusieurs types d'actions peuvent être utilisées dans une spécification Grafcet pour définir les valeurs des sorties. Ces actions peuvent être classées en deux catégories : les actions continues et les actions mémorisées.

- Une action continue définit la valeur courante de la sortie associée à l'action ; la sortie est alors dite assignée. Si une condition d'assignation est associée à une action continue, alors la sortie est assignée uniquement si cette condition est vraie (voir figure 2.3b). Tout comme pour les réceptivités associées aux transitions, une condition d'assignation peut dépendre des variables d'entrées et des variables d'état des étapes du Grafcet.
- Une action mémorisée décrit l'allocation d'une variable de sortie à une variable booléenne, en accord avec une règle d'allocation. Les règles d'allocation sont définies par des flèches situées sur la gauche du rectangle d'action. Une flèche montante indique que la variable de sortie sera allouée lors de l'activation de l'étape (voir figure 2.3c) ; une flèche descendante indique que le variable de sortie sera allouée lors de la désactivation de l'étape. Une action mémorisée peut également être déclenchée par un évènement tel qu'un ou plusieurs fronts de variables d'entrée (voir figure 2.3d).

Il est important de souligner que les actions continues ne sont effectuées que si la situation courante est stable, tandis que les actions mémorisées sont effectuées même lors du passage par une situation instable lors d'une évolution fugace.

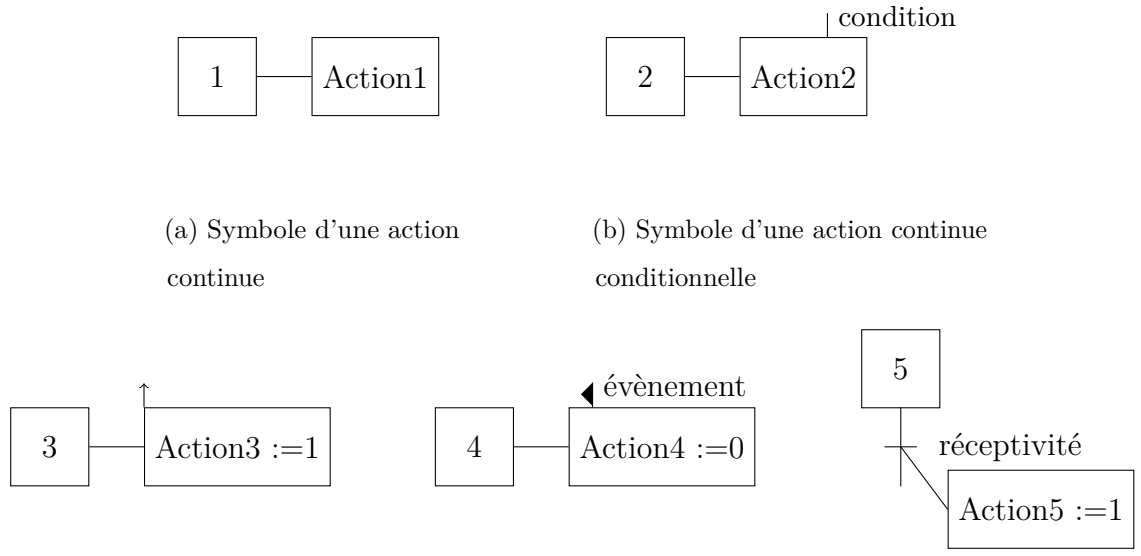
Une action peut également être reliée au symbole de la transition à laquelle elle est associée. Dans ce cas, il s'agit d'une action mémorisée, puisqu'elle est associée à un évènement (le franchissement de la transition). Elle est représentée par un rectangle complété d'un trait oblique la reliant à la transition (voir figure 2.3e).

2.4 Éléments de structuration du Grafcet

La représentation d'une spécification Grafcet peut être structurée et hiérarchisée à l'aide de Grafcet partiels, du forçage, de macro-étapes ou d'étapes encapsulantes.

La norme IEC 60848 définit les Grafcet connexes et partiels ainsi :

- *Un Grafcet connexe est une structure de Grafcet telle qu'il existe toujours une suite*



(a) Symbole d'une action continue

(b) Symbole d'une action continue conditionnelle

(c) Symbole d'une action mémorisée sur activation d'étape

(d) Symbole d'une action mémorisée sur évènement

(e) Symbole d'une action mémorisée sur franchissement d'une transition

Figure 2.3 – Symboles des différentes actions continues (a et b) et mémorisées (c, d et e) (IEC 60848 (2002))

de liens (alternance d'étapes et de transitions) entre deux éléments quelconques, étape ou transition, de ce Grafcet.

– *Constitué d'un ou plusieurs grafcets connexes, un Grafcet partiel résulte d'une partition, selon des critères méthodologiques, du Grafcet global décrivant le comportement de la partie séquentielle d'un système.*

Le forçage d'un Grafcet partiel est un moyen de structuration permettant d'imposer (forcer) un Grafcet partiel dans une situation spécifique, à partir d'un autre Grafcet partiel hiérarchiquement supérieur. L'ordre de forçage est prioritaire sur l'application des règles d'évolution. Tant que l'ordre de forçage est effectif, le Grafcet forcé ne peut pas évoluer. L'ordre de forçage est représenté par un double rectangle associé à une étape.

Une macro-étape représente de manière synthétique une partie de la spécification, en mentionnant la fonction à remplir mais sans exprimer les détails superflus pour la compréhension globale du comportement de la spécification. La description détaillée de cette partie est appelée "expansion" de la macro-étape. Cette expansion, composée d'un ensemble d'étapes et de transitions, commence et se termine par des étapes spécifiques, appelées étapes d'entrée et de sortie de la macro-étape. L'étape d'entrée est activée lorsque l'une des transitions amont de la macro-étape est franchie. Les transitions aval de

la macro-étape ne sont validées que lorsque l'étape de sortie est active ; le franchissement d'une de ces transitions conduit à la désactivation de l'étape de sortie.

Une macro-étape est nommée $M*$ et ses étapes d'entrée et de sortie $E*$ et $S*$, respectivement. Une représentation d'une macro-étape est donnée figure 2.4a.

Une étape encapsulante est utilisée pour structurer de manière hiérarchique un Grafcet. L'encapsulation associée à une étape encapsulante est un Grafcet partiel qui ne peut évoluer que lorsque l'étape encapsulante est active. Lors de l'activation d'une étape encapsulante, les étapes encapsulées associées à un lien d'activation sont activées ; lors de la désactivation de l'étape encapsulante, toutes les étapes encapsulées sont désactivées. Une représentation d'une étape encapsulante est donnée figure 2.4b.

(a) Symbole d'une macro-étape

(b) Symbole d'une étape encapsulante

Figure 2.4 – Symboles d'une macro-étape et d'une étape encapsulante (IEC 60848 (2002))

Les macro-étapes et les étapes encapsulantes ont un comportement bien différent ; nous rappellerons ici les différences principales. Une macro-étape peut être considérée comme une simplification de la représentation graphique du comportement ; c'est-à-dire qu'il est toujours possible de remplacer directement une macro-étape par l'ensemble étapes/transitions défini dans son expansion. Au contraire, une étape encapsulante ne peut pas être remplacée par l'ensemble de ses séquences d'étapes encapsulées. En effet, dans le cas d'une macro-étape, seule l'étape de sortie de son expansion valide les transitions aval de la macro-étape, tandis que pour une étape encapsulante les transitions aval peuvent être franchies indépendamment des étapes actives dans les encapsulations.

2.5 Différences entre Grafcet et SFC

La comparaison suivante est basée sur les deux normes relatives au Grafcet et au SFC, respectivement IEC 60848 (2002) et IEC 61131-3 (2003). Dans ce qui va suivre, les passages extraits de ses normes seront indiqués en italique.

Le Grafcet est un *langage de spécification [...] pour la description fonctionnelle du comportement de la partie séquentielle des systèmes de commande.* À l'opposé, le SFC est utilisé pour structurer une unité d'organisation de programmes (Program Organization Unit (POU)) ; ses élements sont définis en utilisant un ou plusieurs des quatre langages de programmation décrits dans la norme IEC 61131-3 (LD, ST, IL, FBD) [1]. Une spécification Grafcet est donc utilisée pour spécifier un comportement (vue externe d'un système de contrôle/commande), alors qu'un SFC décrit la structure, ou une partie de la structure, du logiciel exécuté dans un automate programmable industriel réalisant ce comportement (vue interne du système de contrôle/commande). La norme 60848 précise que *les méthodes de réalisation d'une spécification utilisant le GRAFCET ne font pas partie du domaine d'application de cette norme. Une méthode possible est l'utilisation du langage "SFC" décrit dans la norme IEC 61131-3, qui définit un ensemble de langages de programmation destinés aux automates programmables.*

Si leurs syntaxes sont assez semblables, les sémantiques du Grafcet et du SFC présentent des différences notables. Par exemple, dans le cas d'une sélection de séquence, dans un Grafcet, *l'exclusion entre les séquences n'est pas structurelle. Pour l'obtenir, le spécificateur doit s'assurer soit de l'incompatibilité mécanique ou temporelle des réceptivités, soit de leur exclusion logique.* À l'opposé, dans un SFC, toute sélection de séquence est exclusive, il ne peut pas y avoir de franchissement simultané de plusieurs transitions d'une même sélection de séquences ; afin de s'assurer de cela, l'utilisateur peut définir des priorités entre les différentes séquences [2]. Un Grafcet peut contenir une ou plusieurs étapes initiales (au moins une étape initiale par cycle de séquence), alors qu'un SFC ne peut et ne doit comporter qu'une seule étape initiale par diagramme SFC. Un SFC ne comporte ni étape encapsulante, ni macro-étape, ni action conditionnelle ; en revanche, les actions peuvent également être déclenchées à l'activation ou la désactivation

1. d'après IEC 61131-3 : *Sequential Function Chart (SFC) elements are defined for structuring the internal organization of programmable controller programs and function blocks*
2. d'après IEC 61131-3 : *any sequence selection is exclusive, [...] it cannot have crossing simultaneous transitions in a sequence selection ; to do this, the user can define priorities between branches at the divergence of sequence selection*

de l'étape à laquelle elles sont associées, et la combinaison de la structure SFC avec un autre langage de la norme 61131-3 permet, en prenant certaines précautions par rapport aux différences mentionnées ci-avant, d'implanter un comportement équivalent à celui défini par une specification Grafcet.

La principale différence sémantique réside dans la définition des règles d'évolution. Dans une spécification Grafcet, les évolutions entre situations stables sont dues à un changement des variables d'entrées ; alors que dans un SFC, les évolutions sont régies par le cycle de lecture des entrées du contrôleur exécutant le modèle. Lorsque la valeur d'une variable d'entrée d'un Grafcet change, ce modèle évolue vers une nouvelle situation stable ; cette évolution est instantanée afin d'éviter de manquer une variation des variables d'entrée, même si des évolutions fugaces sont nécessaires pour atteindre une situation stable. À l'opposé, dans un modèle SFC, le délai de franchissement d'une transition peut être considéré, d'un point de vue théorique, aussi court que souhaité, mais il ne peut jamais être considéré comme nul [3] ; en pratique, ce temps de franchissement est égal au temps de cycle de l'API exécutant ce SFC. Par conséquent, un seul ensemble de transitions simultanément franchissables peut être franchi à chaque cycle d'exécution. Ainsi, les notions de situations stables et instables n'ont pas de sens dans un modèle SFC ; une situation est stable pendant au moins un temps de cycle de l'API.

En conclusion, dans le cadre de ces travaux sur le test de conformité (validation d'une implantation par rapport à sa spécification), le langage Grafcet sera utilisé pour décrire la spécification, le SFC étant seulement une des solutions envisageables pour implanter cette spécification. À partir d'une spécification Grafcet, une séquence de test peut être générée et utilisée pour tester une implantation qui peut tout aussi bien être un API programmé avec un SFC et un des autres langages de la norme IEC 61131-3, ou un contrôleur embarqué programmé en C ; quelle que soit la cible d'implantation, le même Grafcet reste la référence pour les différentes implantations.

3. d'après IEC 61131-3 : *of a transition may theoretically be considered as short as one may wish, but it can never be zero*

3 Formalisation d'une spécification Grafcet : hypothèses et méthode

3.1 Hypothèses de travail

Dans le cadre de cette thèse, les spécifications étudiées devront être définies en accord avec les hypothèses de travail données ci-après.

HYPOTHÈSES DE TRAVAIL :

- Le Grafcet ne contient pas d'élément temporisé (transitions, actions).
- Le Grafcet ne contient pas de réceptivité ou de condition incluant des fronts de variables logiques d'entrée.
- Le Grafcet ne contient pas de compteur.
- Le Grafcet ne contient pas d'action associée aux transitions.
- Le Grafcet ne contient pas d'étape encapsulante.
- Le Grafcet ne contient pas d'ordre de forçage.
- Le Grafcet ne contient pas d'étape source, ni d'étape puits, afin d'assurer la réatteignabilité de l'ensemble des étapes.
- Le Grafcet peut être composé de plusieurs Grafcet partiels.
- Chaque Grafcet partiel peut être composé d'étapes classiques et de macro-étapes.
- Les actions peuvent être continues, conditionnelles ou non, ou mémorisées.

Les contraintes de sécurité logiques devant être testées avant les contraintes temporelles, la technique de test de conformité proposée traite uniquement le test non temporisé d'une implantation ; c'est pourquoi les éléments temporisés, les fronts ainsi que les compteurs sont exclus des spécifications étudiées. Les limitations relatives aux actions associées aux transitions, aux étapes encapsulantes et au forçage restreignent le domaine des spécifications qui peuvent être traitées automatiquement, mais cette limitation ne constitue pas une limite théorique forte. En effet, une majorité des spécifications Grafcet peut être traitée avec les concepts pris en compte. De plus, si une spécification utilise un des concepts non pris en compte (fronts de variables logiques d'entrée, action associée aux transitions, étapes encapsulantes ou ordre de forçage), il est possible de la modifier afin d'exprimer le même comportement à l'aide des modèles pris en compte. Par exemple,

une action mémorisée associée au franchissement d'une transition peut être traitée en remplaçant la transition par une séquence transition-étape-transition (voir figure 2.5) ; l'étape intermédiaire est instable mais l'action mémorisée est effectuée.

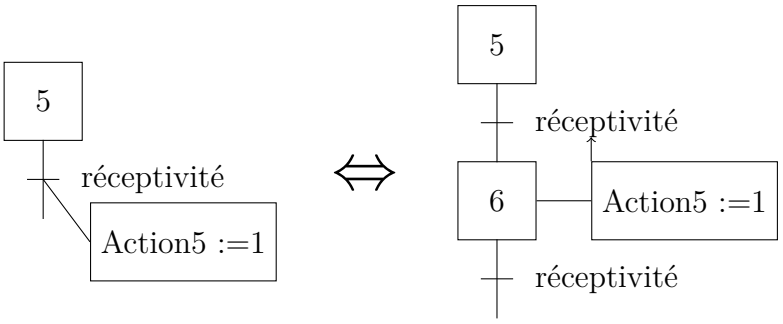


Figure 2.5 – Transformation d'une action mémorisée associée à une transition

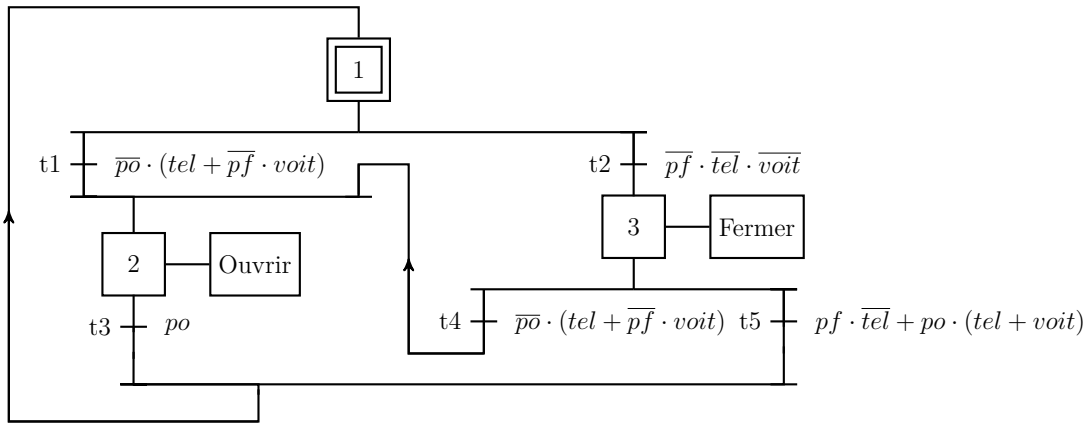
Les figures 2.6 et 2.7 décrivent deux spécifications Grafcet qui serviront d'illustrations pour la suite de ce chapitre.

La première spécification est tirée de Roussel et Faure (2006) ; elle ne contient que des étapes classiques auxquelles sont associées des actions continues non conditionnelles. Ce premier exemple simple comporte 4 entrées logiques et 2 sorties logiques ; il sera appelé exemple A par la suite.

La seconde spécification est adaptée d'un exemple utilisé dans les normes IEC 60848 (2002) et IEC 61131-3 (2003) ; elle contient également des macro-étapes ($M20$, $M30$ et $M40$), des actions conditionnelles (associées aux étapes $F1$, $E20$, 21 et 22) et des actions mémorisées (à l'activation : associée à l'étape $E30$ et à la désactivation : associée à l'étape 32). Cette spécification illustre les deux types de parallélisme : le parallélisme structurel par l'activation de séquences parallèles en aval de la transition $t4$, et le parallélisme interprété entre deux Grafcet partiels par l'utilisation des variables d'activités $X1$ et $XF1$ dans l'expression de la condition d'affectation de l'action associée à l'étape $F1$ ainsi que dans l'expression de la réceptivité associée à la transition $t4$. Ce deuxième exemple comporte 9 entrées logiques et 10 sorties logiques ; il sera appelé exemple B par la suite.

3.2 Description de la méthode de formalisation

La méthode proposée permet de définir une transformation de modèle, sans perte de sémantique, entre une spécification Grafcet et une machine de Mealy. Cette méthode



Entrées				Sorties	
po	Portail ouvert	tel	Télécommande	Fermer	Fermer le portail
pf	Portail fermé	voit	Présence voiture	Ouvrir	Ouvrir le portail

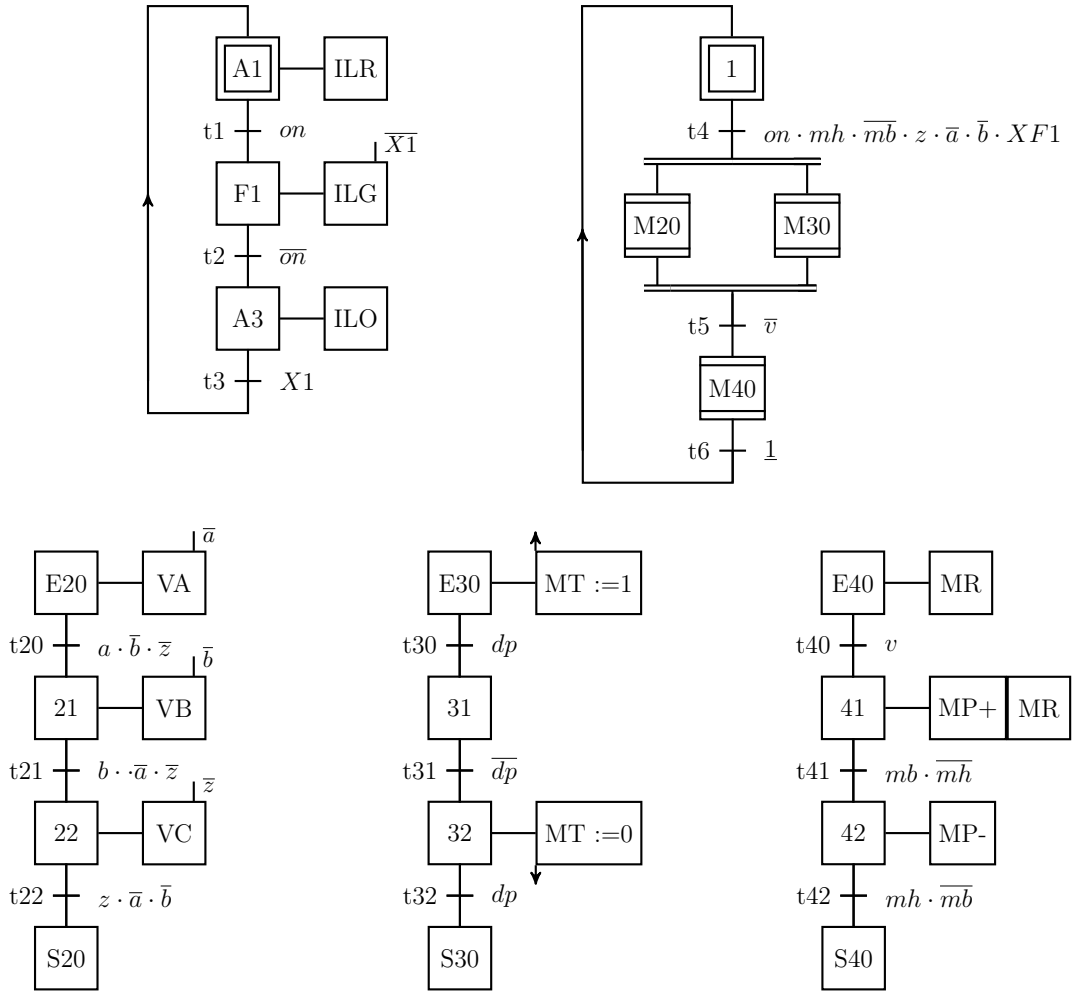
Figure 2.6 – Exemple de spécification Grafcet simple

peut être décomposée en deux phases (voir figure 2.8) :

- La construction d'un automate, appelé ALS, représentant de manière formelle tous les états stables du comportement d'un système logique décrit par une spécification Grafcet ainsi que toutes les évolutions entre ces états.
- La traduction de cet automate en une machine de Mealy équivalente, modèle plus adapté à la génération de séquences de test.

L'état global d'un système logique dont le comportement est décrit par une spécification Grafcet, appelé *localité*, est défini par des étapes simultanément actives de cette spécification Grafcet (situation) et par l'ensemble des sorties émises.

Nous rappelons en effet que des ensembles de sorties différents peuvent être émis pour une même situation, par exemple lorsque le Grafcet contient des actions mémorisées ou des actions conditionnelles dépendant de valeurs d'entrée. Le concept de situation ne traduit donc pas le comportement extrinsèque (vision en boîte noire) de la spécification, alors que c'est la définition de ce comportement extrinsèque qui est utilisée pour le test de conformité. Par opposition, le concept de localité permet de décrire ce comportement externe, en déterminant les valeurs des sorties pouvant être émises pour chaque état interne (situation) de la spécification. L'ALS est donc un enrichissement du Graphe des Situations Accessibles (GSA) —analogie du graphe des marquages d'un réseau de Petri,



Entrées					
on	En production	mh	Malaxeur haut	a	Poids liquide A atteint
dcy	Départ cycle	mb	Malaxeur bas	b	Poids total atteint
dp	Détection passage	z	Bascule vide	v	Viscosité atteinte

Sorties					
MR	Malaxer	VA	Ouvrir vanne A	ILV	Allumer voy. vert
MP+	Vidanger malaxeur	VB	Ouvrir vanne B	ILO	Allumer voy. orange
MP-	Remonter malaxeur	VC	Ouvrir vanne C	ILR	Allumer voy. rouge
MT	Alimenter tapis				

Figure 2.7 – Exemple de spécification Grafcet utilisant macro-étapes, actions conditionnelles et actions mémorisées (Exemple B)

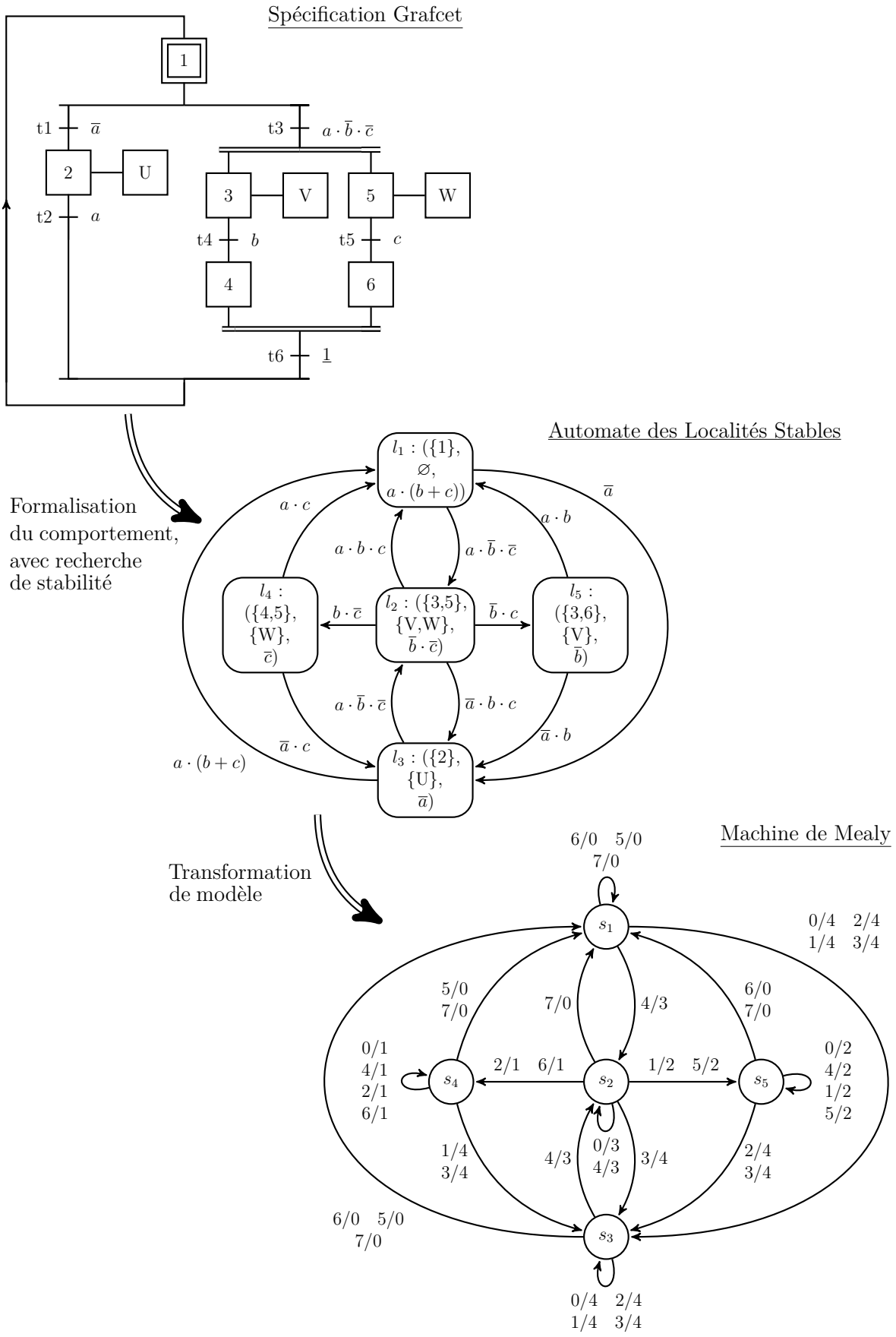


Figure 2.8 – Aperçu de la méthode de formalisation

pour une spécification Grafcet— défini par Blanchard (1979) et formalisé par Roussel (1994). En effet, dans un GSA, une situation représente un ensemble d'étapes actives mais ne tient pas compte des sorties logiques émises.

Comme une situation du Grafcet peut être stable ou instable, une localité peut également être stable ou instable. Les actions continues sont exécutées uniquement lorsque la situation du Grafcet est stable ; ainsi, les sorties commandées par des actions continues sont émises uniquement pour des localités stables. Afin d'observer toutes les sorties, quel que soit le type d'action qui les pilote (continue ou mémorisée), seules les localités stables doivent être conservées. Une condition de stabilité, expression booléenne dépendant des variables d'entrée, est associée à chaque localité stable. Lorsqu'une localité est active[4] et que sa condition de stabilité est vraie, l'ALS conserve sa localité courante active, autrement il évolue vers une nouvelle localité. Chaque évolution depuis une localité stable vers une autre localité stable est due à un changement des variables d'entrée : une expression booléenne, appelée condition d'évolution, est également associée à chaque évolution. Une définition formelle des localités ainsi que des évolutions est détaillée dans la section 4.

Une fois l'ALS construit, l'objectif de la seconde phase est de définir les fonctions de transition et de sortie de la machine de Mealy équivalente à partir des conditions d'évolution de l'ALS. Ceci sera détaillé dans la section 5.

4 Construction de l'automate des localités stables d'une spécification Grafcet

Afin de construire automatiquement l'ALS d'une spécification Grafcet, les sémantiques du Grafcet et de l'ALS doivent préalablement être définies ; ceci est respectivement l'objectif des sections 4.1 et 4.2. La section 4.3 est consacrée à la technique d'obtention d'un ALS à partir d'une spécification Grafcet. La section 4.4 est relative au traitement des exemples.

Nous pouvons rappeler que la formalisation du Grafcet a déjà été étudiée en utilisant différentes approches. Une technique de formalisation du comportement du Grafcet en utilisant un méta-modèle statique associé à un algorithme d'interprétation des règles

4. Par définition, une seule localité est active à chaque instant.

d'évolution a été proposée par Lhoste *et al.* (1993) et Bierel *et al.* (1997). Une formalisation de la sémantique d'un Grafcet réactif pour lequel les réceptivités associées aux transitions sont définies par des évènements (fronts montants ou descendants de variables logiques) a été proposé par Cassez (1997). Cependant cette dernière formalisation ne traite pas du comportement des sorties logiques associées aux étapes du Grafcet. L'ensemble de ces travaux est basé sur la version précédente de la norme IEC 60848, datant de 1988. Les travaux présentés ici considèrent la version de 2002 et sont basés sur une approche algébrique, afin de limiter l'explosion combinatoire.

4.1 Définition formelle d'une spécification Grafcet

Formellement, un Grafcet G est un 4-uplet $(I_G, O_G, C_G, S_{InitG})$ tel que :

- I_G est l'ensemble fini non vide des entrées logiques, (Cardinalité de I_G : $|I_G|$).
- O_G est l'ensemble fini non vide des sorties logiques, (Cardinalité de O_G : $|O_G|$).
- C_G est l'ensemble fini des graphes connexes d'une spécification Grafcet.
- S_{InitG} est l'ensemble fini des étapes initiales.

L'ensemble des graphes connexes C_G peut-être partitionné en deux sous-ensembles, l'ensemble des graphes connexes classiques : C_C, et l'ensemble des graphes connexes représentant les expansions des macro-étapes : C_E.

$$\begin{cases} C_C \cup C_E = C_G \\ C_C \cap C_E = \varnothing \end{cases} \tag{2.1}$$

Un graphe connexe classique $c \in C_C$ est défini par un triplet (S, T, A) tel que :

- S est l'ensemble non vide des étapes s de c.
- T est l'ensemble des transitions t de c.
- A est l'ensemble des actions a de c.

Un graphe connexe représentant l'expansion d'une macro-étape $c \in C_E$ est défini par un 5-uplet $(m, s_I, s_O, S_{oth}, T, A)$ tel que :

- m est la macro-étape à laquelle est associée l'expansion.
- s_I est l'étape d'entrée de l'expansion de la macro-étape.
- s_O est l'étape de sortie de l'expansion de la macro-étape.
- S_{oth} est l'ensemble des autres étapes s de l'expansion de la macro-étape.

– T est l'ensemble des transitions t de l'expansion de la macro-étape.
– A est l'ensemble des actions a associées aux étapes de l'expansion de la macro-étape.

Pour tout graphe connexe représentant l'expansion d'une macro-étape $c \in C_E$, $S(c)$ est l'ensemble de toutes les étapes de ce graphe connexe c.

$$S(c) = \{s_I(c), s_O(c)\} \cup S_{oth}(c) \tag{2.2}$$

L'ensemble de toutes les étapes s d'une spécification Grafcet sera noté S_G.

$$S_G = \bigcup_{c \in C_G} S(c) \tag{2.3}$$

Pour l'exemple A, la spécification Grafcet est composée d'un graphe connexe classique, lui-même composé de 3 étapes (dont une initiale), 5 transitions et 2 actions.

Pour l'exemple B, la spécification Grafcet est composée de deux graphes connexes classiques et de trois graphes connexes représentant l'expansion d'une macro-étape. Le premier graphe connexe classique (en haut à gauche) est composé de 3 étapes (dont une étape initiale), 3 transitions et 3 actions. Le deuxième graphe connexe classique (en haut à droite) est composé de 4 étapes (dont une étape initiale et trois macro-étapes), 3 transitions et aucune action. Les trois autres graphes connexes (en bas) représentent, respectivement, l'expansion des macro-étapes $M20$, $M30$ et $M40$. Ils sont composés chacun de 4 étapes (dont une étape d'entrée et une étape de sortie) et 3 transitions ; et comportent respectivement 3, 2 et 4 actions.

Une variable booléenne d'activité d'étape $X(s)$ est associée à chaque étape s. L'ensemble des étapes initiales S_{InitG} est un sous-ensemble de S_G.

Une transition $t \in T$ d'un graphe connexe donné $c \in C_G$ est défini par un triplet $(S_U, S_D, E_{Cond(I_G,S_G)})$ tel que :

– S_U est l'ensemble des étapes immédiatement précédentes de cette transition, $S_U \subset S(c)$.
– S_D est l'ensemble des étapes immédiatement suivantes de cette transition, $S_D \subset S(c)$.
– $E_{Cond(I_G,S_G)}$ est la réceptivité associée à la transition, expression booléenne dépendant des variables d'entrée logiques et des variables d'activité d'étape.

L'ensemble de toutes les transitions t d'une spécification Grafcet sera noté T_G.

$$T_G = \bigcup_{c \in C_G} T(c) \tag{2.4}$$

Pour l'exemple B, la réceptivité associée à la transition $t4$ utilise 6 variables d'entrées ainsi que la variable d'activité de l'étape $F1$.

L'ensemble des actions A peut-être partitionné en deux sous-ensembles : l'ensemble des actions continues A_C et l'ensemble des actions mémorisées A_S. De façon similaire, l'ensemble des sorties O_G peut-être partitionné en deux sous-ensembles : l'ensemble des sorties assignées (pilotées par une action continue) : O_C, et l'ensemble des sorties affectées (pilotées par une action mémorisée) : O_S.

$$\begin{cases} A_C \cup A_S = A \\ A_C \cap A_S = \varnothing \end{cases} \tag{2.5}$$

$$\begin{cases} O_C \cup O_S = O_G \\ O_C \cap O_S = \varnothing \end{cases} \tag{2.6}$$

Une action continue $a_c \in A_C$ d'un graphe connexe donné c est définie par un triplet $(s, o, E_{Cond(I_G,S_G)})$ tel que :

- s est l'étape à laquelle l'action est associée, $s \in S(c)$.
- o est la sortie assignée par l'action, $o \in O_G$.
- $E_{Cond(I_G,S_G)}$ est la condition d'assignation, expression booléenne dépendant des variables d'entrée logiques et des variables d'activité d'étape.

La valeur de chaque sortie assignée peut ainsi être définie par une expression booléenne dépendant des variables d'entrée logiques et des variables d'activité d'étape, telle que :

$$E_{Emit(I_G,S_G)}(o) = \sum_{\substack{a \in A_C \\ o(a)=o}} \Big(X(s(a)) \cdot E_{Cond(I_G,S_G)}(a) \Big) \tag{2.7}$$

Pour l'exemple B, l'action associée à l'étape $F1$ est une action conditionnelle, sa condition d'assignation est $\overline{X1}$. La sortie ILG n'étant associée qu'à cette action, la condition d'émission de la sortie ILG est $XF1 \cdot \overline{X1}$, c'est-à-dire que la sortie ILG est émise si et seulement si l'étape $F1$ est active et l'étape 1 inactive. De même, l'action

conditionnelle associée à l'étape $E20$ a pour condition d'assignation $\overline{a}$. La sortie VA n'étant associée qu'à cette action, la condition d'émission de cette sortie est $XE20 \cdot \overline{a}$, c'est-à-dire que la sortie VA est émise si et seulement si l'étape $E20$ est active et que l'entrée a est à 0. En revanche la sortie MR est associée à deux actions (associées aux étapes $E40$ et 41), ces deux actions étant des actions non conditionnelles, la condition d'assignation est donc 1 (vrai), et la condition d'émission de la sortie MR est $XE40+X41$, c'est-à-dire que la sortie VA est émise si et seulement si une des deux étapes $E40$ ou 41 est active.

Une action mémorisée $a_s \in A_S$ d'un graphe connexe donné c est définie par un 4-uplet (s,o,op,$inst$) tel que :

- s est l'étape à laquelle l'action est associée, $s \in S(c)$.
- o la sortie affectée par l'action, $o \in O_G$.
- op le type d'affectation, $op \in \{Set, Reset\}$.
- $inst$ est l'instant d'affectation, $inst \in \{Act, Deact\}$, où Act est l'instant correspondant à l'activation de l'étape, et $Deact$ est l'instant correspondant à la désactivation de l'étape.

Ainsi, les valeurs des sorties affectées sont calculées dynamiquement pendant la construction de l'ALS (voir sous-section 4.3.1).

Pour l'exemple B, l'action associée à l'étape $E30$ affecte la sortie MT à 1 lors de l'activation de l'étape $E30$; tandis que l'action associée à l'étape 32 affecte la sortie MT à 0 lors de la désactivation de l'étape 32.

4.2 Définition formelle d'un automate des localités stables

4.2.1 Bases de la définition formelle d'un l'ALS

Formellement, un Automate des Localités Stables ALS est un 4-uplet (I_{ALS}, O_{ALS}, $L, Evol$) tel que :

- I_{ALS} est l'ensemble des entrées logiques de la spécification Grafcet G : $I_{ALS} = I_G$.
- O_{ALS} est l'ensemble des sorties logiques de la spécification Grafcet G : $O_{ALS} = O_G$.
- L est l'ensemble des localités stables l.
- $Evol$ est l'ensemble des évolutions e entre localités stables.

Une localité stable est caractérisée par un ensemble d'étapes simultanément actives, un ensemble de sorties émises et une condition de stabilité, expression booléenne

dépendant des variables d'entrée. Ainsi, une localité stable est définie par le triplet : $(S_{Act}, O_{Emit}, E_{Stab(I_G)})$ tel que :

- S_{Act} est un sous-ensemble des étapes de la spécification Grafcet G, ($S_{Act} \subset S_G$).
- O_{Emit} est un sous-ensemble des sorties de la spécification Grafcet G, ($O_{Emit} \subset O_G$).
- $E_{Stab(I_G)}$ est une expression booléenne dépendant des variables d'entrée de la spécification Grafcet G. Cette expression est vraie uniquement pour les combinaisons d'entrées telle que la localité l est stable.

4.2.2 Prise en compte de l'initialisation

Il est important de préciser que la situation initiale d'un Grafcet peut être instable. En effet, en fonction de la valeur des entrées logiques lors de l'initialisation et de la situation initiale du Grafcet (ensemble des étapes initiales), des transitions peuvent être validées et peuvent donc conduire à un changement de situation. La situation initiale d'une spécification Grafcet représente donc la situation active à l'initialisation avant stabilisation, et non pas la situation active après recherche de stabilité.

Pour prendre en compte le cas particulier de la situation initiale d'une spécification Grafcet, avant et après stabilisation, une localité instable, notée l_{Init} est introduite. Il s'agit de la seule localité instable de l'ALS, cette localité ne pouvant de plus être atteinte que lors de l'initialisation de la spécification (il n'existe aucune évolution permettant de l'atteindre depuis une autre localité). Les détails concernant le comportement de cette localité dans le cadre d'une implantation sur un API sont donnés dans le chapitre 3, section 4.1.

Pour l'exemple A, la situation initiale est $\{1\}$; cette situation initiale est stable si et seulement si aucune des réceptivités associées aux transitions $t1$ et $t2$ n'est vérifiée, c'est-à-dire si et seulement si la condition $pf \cdot \overline{tel} + po \cdot (tel + voit)$ est vérifiée.

Pour l'exemple B, la situation initiale est $\{A1, 1\}$; cette situation initiale est stable si et seulement si aucune des réceptivités associées aux transitions $t1$ et $t4$ n'est vérifiée, c'est-à-dire si et seulement si la condition $\overline{on}$ est vérifiée.

4.2.3 Définition formelle de l'ALS avec initialisation

La définition suivante prend en compte l'instabilité potentielle de la situation initiale d'une spécification Grafcet.

Formellement, un Automate des Localités Stables initialisable ALS est un 5-uplet $(I_{ALS}, O_{ALS}, L, l_{Init}, Evol)$ tel que :

- I_{ALS} est l'ensemble des entrées logiques de la spécification Grafcet G : $I_{ALS} = I_G$.
- O_{ALS} est l'ensemble des sorties logiques de la spécification Grafcet G : $O_{ALS} = O_G$.
- L est l'ensemble des localités stables l.
- l_{Init} est la localité initiale, $l_{Init} \notin L$.
- $Evol$ est l'ensemble des évolutions e entre localités (stables et initiale).

La localité initiale l_{Init}, instable, est caractérisée par le couple : $(S_{InitG}, E_{Stab(I_G)})$ tel que :

- S_{InitG} est l'ensemble des étapes initiales de la spécification Grafcet G.
- $E_{Stab(I_G)} = 0$. La localité initiale étant instable, sa condition de stabilité est toujours fausse.

La localité initiale l_{Init} étant instable, aucune sortie n'est associée à cette localité.

Pour l'exemple A, la localité initiale est définie par le couple $(\{1\}, 0)$. Pour l'exemple B, la localité initiale est définie par le couple $(\{A1, 1\}, 0)$.

Par la suite, la localité stable $l_{exB} = (\{F1, 22, 32\}, \{MT, VC, ILV\}, on \cdot \overline{dp} \cdot \overline{z})$ de l'exemple B sera utilisée pour illustrer les différents points-clés de la méthode.

Une évolution e de $Evol$ est définie pour représenter soit une évolution entre deux localités stables, soit une évolution depuis la localité initiale vers une localité stable. Chacune de ces évolutions peut être définie formellement par le triplet : $(l_U, l_D, E_{Evol(I_G)})$, tel que :

- l_U est la localité amont, $l_U \in L \cup l_{Init}$.
- l_D est la localité aval, $l_D \in L$.
- $E_{Evol(I_G)}$ est la condition d'évolution, expression booléenne dépendant des variables d'entrée de la spécification Grafcet G. Cette expression est vraie uniquement pour les combinaisons d'entrées telles que l'ALS évolue de la localité amont l_U à la localité aval l_D.

Un ALS est correctement défini s'il satisfait les huit propriétés suivantes :

Propriété 1 : Unicité des localités :

$$\forall (l_1, l_2) \in L^2, \quad (S_{Act}(l_1), O_{Emit}(l_1)) \neq (S_{Act}(l_2), O_{Emit}(l_2)) \tag{2.8}$$

Propriété 2 : Unicité d'évolution entre deux localités :

$$\forall (e_1, e_2) \in Evol^2, \quad (l_U(e_1), l_D(e_1)) \neq (l_U(e_2), l_D(e_2)) \tag{2.9}$$

Propriété 3 : Absence d'évolution bouclée (self-loop). Les localités amont et aval de chaque évolution sont différentes. Toutes les évolutions représentent un changement de localité ; il n'y a pas d'évolution bouclant sur une même localité :

$$\forall e \in Evol, \quad l_U(e) \neq l_D(e) \tag{2.10}$$

Propriété 4 : Déterminisme d'évolution [5] :

$$\forall (e_1, e_2) \in Evol^2, \quad l_U(e_1) = l_U(e_2) \Rightarrow E_{Evol(I_G)}(e_1) \cdot E_{Evol(I_G)}(e_2) = 0 \tag{2.11}$$

Propriété 5 : Cohérence stabilité/évolution. Pour chaque localité, aucune combinaison des variables d'entrée ne satisfait à la fois la condition de stabilité de cette localité et une condition d'évolution depuis cette localité :

$$\forall l \in L \cup l_{Init}, \quad E_{Stab(I_G)}(l) \cdot \left(\sum_{\substack{e \in Evol \\ l_U(e)=l}} E_{Evol(I_G)}(e) \right) = 0 \tag{2.12}$$

Propriété 6 : Complétude. Pour chaque localité et pour chaque combinaison des variables d'entrée, le comportement est complètement défini (soit la localité est stable, soit il existe une évolution depuis cette localité) :

$$\forall l \in L \cup l_{Init}, \quad E_{Stab(I_G)}(l) + \left(\sum_{\substack{e \in Evol \\ l_U(e)=l}} E_{Evol(I_G)}(e) \right) = 1 \tag{2.13}$$

Cette définition de la complétude, permet également de définir formellement la condition de stabilité d'une localité comme étant la négation de la somme des évolutions depuis cette localité.

$$\forall l \in L \cup l_{Init}, \quad E_{Stab(I_G)}(l) = \overline{\sum_{\substack{e \in Evol \\ l_U(e)=l}} E_{Evol(I_G)}(e)} \tag{2.14}$$

Propriété 7 : Il n'y a pas d'évolution fugace :

$$\forall (e_1, e_2) \in Evol^2, \quad l_D(e_1) = l_U(e_2) \Rightarrow E_{Evol(I_G)}(e_1) \cdot E_{Evol(I_G)}(e_2) = 0 \tag{2.15}$$

Propriété 8 : La localité initiale n'est atteignable depuis aucune autre localité :

$$\forall e \in Evol, \quad l_D(e) \neq l_{Init} \tag{2.16}$$

Enfin, la définition formelle d'une localité stable doit être cohérente avec les règles d'évolution du Grafcet et les définitions des actions, c'est-à-dire :

5. Dans les équations booléennes, 0 signifie Faux et 1 Vrai.

– Lorsque la condition de stabilité est vraie, aucune transition validée pour la situation correspondante du Grafcet G ne peut être franchie (équation 2.17).

$$\forall l \in L, \quad \left(\prod_{s \in S_{Act}(l)} X(s) \right) \cdot \left(\prod_{\substack{s \in S_G \\ s \notin S_{Act}(l)}} \overline{X(s)} \right) \cdot E_{Stab(I_G)}(l) \cdot \left(\sum_{\substack{t \in T_G \\ S_U(t) \subseteq S_{Act}(l)}} E_{Cond(I_G,S_G)}(t) \right) = 0 \quad (2.17)$$

– Une sortie contrôlée par une action continue du Grafcet G appartient à l'ensemble des sorties émises si et seulement si elle est contrôlée par une action continue associée à une étape active pour la situation correspondante du Grafcet G et que la combinaison des variables d'entrée satisfaisant la condition de stabilité de l'ALS satisfait aussi la condition d'assignation associée à l'action (équation 2.18).

$$\forall l \in L, \quad \left(\prod_{s \in S_{Act}(l)} X(s) \right) \cdot \left(\prod_{\substack{s \in S_G \\ s \notin S_{Act}(l)}} \overline{X(s)} \right) \cdot E_{Stab(I_G)}(l) \cdot \left(\sum_{\substack{o \in O_{Emit}(l) \\ o \in O_C}} \overline{E_{Emit(I_G,S_G)}(o)} + \sum_{\substack{o \notin O_{Emit}(l) \\ o \in O_C}} E_{Emit(I_G,S_G)}(o) \right) = 0 \quad (2.18)$$

4.3 Construction de l'ALS d'une spécification Grafcet

Les définitions formelles d'une spécification Grafcet et d'un ALS ayant été posées, il est alors possible de définir les règles de construction d'un ALS à partir d'une spécification Grafcet. La construction de cet ALS repose sur la détermination de toutes les évolutions possibles depuis la localité initiale l_{Init}.

Le calcul des conditions d'évolution est effectué par un calcul symbolique sur les expressions booléennes dépendant des variables d'entrée. Par exemple, si deux ensembles S_1 et S_2 de combinaisons de variables d'entrée sont définis par les expressions Exp_1 et Exp_2, alors les ensembles $S_1 \cap S_2$, $S_1 \cup S_2$, et $S_1 \setminus S_2$ sont respectivement représentés par $(Exp_1 \cdot Exp_2)$, $(Exp_1 + Exp_2)$ et $(Exp_1 \cdot \overline{Exp_2})$. Cette solution limite l'explosion combinatoire et s'applique bien dans le cas des spécifications Grafcet pour lesquelles les réceptivités des transitions et les conditions d'assignation des actions sont définies par des expressions booléennes.

Comme indiqué précédemment, toutes les évolutions d'un ALS sont dues à une variation des variables d'entrée.

Les évolutions entre localités stables peuvent être scindées en deux parties :

- Les évolutions correspondant au franchissement d'une séquence d'ensembles de transitions simultanément franchissables de la spécification Grafcet.
- Les évolutions ne correspondant pas au franchissement de transitions de la spécification Grafcet mais uniquement à un changement des sorties émises.

Dans le premier cas, l'ensemble des étapes actives $S_{Act}(l)$ est changé, contrairement au second cas où seuls O_{Emit} et $E_{Stab(I_G)}$ sont modifiés.

4.3.1 Détermination des évolutions dues aux franchissements de transitions

Ces évolutions sont calculées depuis une localité stable $l = (S_{Act}, O_{Emit}, E_{Stab(I_G)})$, de la façon suivante :

Phase 1 : Détermination de l'ensemble des transitions de la spécification Grafcet franchissables, et de tous les ensembles de transitions simultanément franchissables, à partir de la situation active S_{Act}.

Phase 2 : Détermination des situations atteintes lors du franchissement d'un ensemble de transitions simultanément franchissables. Si la situation atteinte est instable, alors les phases 1 et 2 de la méthode doivent être réitérées jusqu'à atteindre une situation stable. Lorsque la séquence de franchissement est infinie, c'est-à-dire lorsqu'il existe un cycle d'évolution ne comportant que des situations instables, la spécification Grafcet est déclarée incohérente et doit être modifiée ; dans ce cas, l'ALS ne peut être calculé.

Phase 3 : Détermination de l'ensemble des sorties émises pour la situation stable atteinte par la séquence de franchissements d'ensembles de transitions simultanément franchissables.

Ces trois phases sont détaillées ci-dessous.

4.3.1.1 Phase 1

Une transition d'une spécification Grafcet G est franchissable lorsqu'elle est validée et que sa réceptivité est vraie. Ainsi, le calcul de l'ensemble des transitions validées t

depuis une situation $S_{Act}(l)$ d'une localité l est d'abord effectué :

$$T_{Enab}(l) = \{t \in T(G) \quad | \quad S_U(t) \subset S_{Act}(l)\} \tag{2.19}$$

Ainsi, pour l_{exB} on obtient $T_{Enab}(l_{exB}) = \{t2, t22, t32\}$

L'ensemble des transitions franchissables depuis une situation $S_{Act}(l)$ est le sous-ensemble de $T_{Enab}(l)$ pour lequel les variables d'entrée et d'activité d'étapes satisfont la réceptivité associée à une transition t. Cet ensemble $T_{Fire}(l)$ est défini comme suit :

$$T_{Fire}(l) = \Bigg\{ t \in T_{Enab}(l) \quad | \quad \left(\prod_{s \in S_{Act}(l)} X(s) \right) \cdot \left(\prod_{\substack{s \in S_G \\ s \notin S_{Act}(l)}} \overline{X(s)} \right) \cdot \left(E_{Cond(I_G, S_G)}(t) \right) \neq 0 \Bigg\} \tag{2.20}$$

Ainsi, pour l_{exB} on obtient $T_{Fire}(l_{exB}) = \{t2, t22, t32\}$. Toutes les transitions validées sont franchissables, car les réceptivités associées à ces transitions sont indépendantes des variables d'activité des étapes.

Afin de simplifier les formules, les notations suivantes seront utilisées pour la suite du mémoire :

- $E^l_{Fire(I_G)}(t)$ désignera l'expression booléenne dépendant des variables d'entrée représentant les combinaisons d'entrée pour lesquelles la transition t peut-être franchie depuis la situation $S_{Act}(l)$. Cette expression booléenne est obtenue à partir de la réceptivité associée à la transition t en substituant les variables d'activité d'étape par leur valeur correspondante (Vrai/Faux) en fonction de la situation active $S_{Act}(l)$.

$$E^l_{Fire(I_G)}(t) = E_{Cond(I_G, S_G)}(t) \quad \text{où} \begin{cases} \forall s \in S_{Act}(l) & X(s) = 1 \\ \forall s \notin S_{Act}(l) & X(s) = 0 \end{cases} \tag{2.21}$$

- Le terme I_G sera supprimé dans toutes les expressions notées $E_{...}$ puisque ces expressions dépendent uniquement des variables d'entrée de la spécification Grafcet. Il n'est donc plus nécessaire de le préciser.

En utilisant ces notations, l'ensemble $T_{Fire}(l)$ des transitions franchissables depuis

une situation $S_{Act}(l)$ d'un Grafcet, défini équation (2.20), peut être redéfini comme suit :

$$T_{Fire}(l) = \left\{ t \in T_{Enab}(l) \quad | \quad E^l_{Fire}(t) \neq 0 \right\} \tag{2.22}$$

Soit $ST_{SimFire}(l)$ l'ensemble des couples $(T_{SimFire}, E^l_{Fire}(T_{SimFire}))$, où $T_{SimFire}$ est un ensemble de transitions simultanément franchissables depuis $S_{Act}(l)$, et $E^l_{Fire}(T_{SimFire})$ est la condition booléenne permettant ce franchissement. L'ensemble $ST_{SimFire}(l)$ est défini par :

$$\begin{aligned} ST_{SimFire}(l) = \Big\{ & (T_{SimFire}, E^l_{Fire}(T_{SimFire})) \quad | \\ & [T_{SimFire} \subset T_{Fire}(l)] \wedge [E^l_{Fire}(T_{SimFire}) \neq 0] \Big\} \\ \text{avec } E^l_{Fire}(T_{SimFire}) = & \left(\prod_{t \in T_{SimFire}} E^l_{Fire}(t) \right) \cdot \left(\prod_{\substack{t \in T_{Fire}(l) \\ t \notin T_{SimFire}}} \overline{E^l_{Fire}(t)} \right) \end{aligned} \tag{2.23}$$

Ainsi, pour l_{exB} puisque $T_{Fire}(l_{exB}) = \{t2, t22, t32\}$, il est possible de construire 7 ensembles $T_{SimFire}$ de transitions simultanément franchissables depuis $S_{Act}(l_{exB})$: $\{T_{SimFire}\}(l_{exB}) = \{\{t2\}, \{t22\}, \{t32\}, \{t2, t22\}, \{t2, t32\}, \{t22, t32\}, \{t2, t22, t32\}\}$. En calculant les conditions booléennes permettant uniquement le franchissement des transitions de $T_{SimFire}$ pour chacun des 7 éléments de $\{T_{SimFire}\}(l_{exB})$, on obtient $ST_{SimFire}(l_{exB}) = \{(\{t2\}, \overline{on} \cdot \overline{dp} \cdot (\overline{z} + a + b)), (\{t22\}, on \cdot \overline{dp} \cdot z \cdot \overline{a} \cdot \overline{b}), (\{t32\}, on \cdot dp \cdot (\overline{z} + a + b)), (\{t2, t22\}, \overline{on} \cdot \overline{dp} \cdot z \cdot \overline{a} \cdot \overline{b}), (\{t2, t32\}, \overline{on} \cdot dp \cdot (a + b + \overline{z})), (\{t22, t32\}, on \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b}), (\{t2, t22, t32\}, \overline{on} \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b})\}$

4.3.1.2 Phase 2

La situation atteinte lorsqu'un ensemble de transitions $T_{SimFire}$ est simultanément franchi depuis $S_{Act}(l)$ est définie par :

$$S^l(T_{SimFire}) = \left(S_{Act}(l) \setminus \bigcup_{t \in T_{SimFire}} S_U(t) \right) \cup \bigcup_{t \in T_{SimFire}} S_D(t) \tag{2.24}$$

Soit $SS^l(T_{SimFire})$ l'ensemble des couples $(S^l(T_{SimFire}), E^l_{Fire}(T_{SimFire}))$, où $S^l(T_{SimFire})$ est la situation atteinte lorsqu'un ensemble de transitions $T_{SimFire}$ est simultanément franchi depuis $S_{Act}(l)$, et $E^l_{Fire}(T_{SimFire})$ est la condition booléenne permettant ce franchissement. La situation atteinte $S^l(T_{SimFire})$ peut être stable, partiellement instable ou totalement instable.

Ainsi, pour l_{exB}, on obtient $SS^{l_{exB}}(T_{SimFire}) = \{(\{A3, 22, 32\}, \overline{on} \cdot \overline{dp} \cdot (\overline{z} + a + b)), (\{F1, S20, 32\}, on \cdot \overline{dp} \cdot z \cdot \overline{a} \cdot \overline{b}), (\{F1, 22, S30\}, on \cdot dp \cdot (\overline{z} + a + b)), (\{A3, S20, 32\}, \overline{on} \cdot \overline{dp} \cdot z \cdot \overline{a} \cdot \overline{b}), (\{A3, 22, S30\}, \overline{on} \cdot dp \cdot (a + b + \overline{z})), (\{F1, S20, S30\}, on \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b}), (\{A3, S20, S30\}, \overline{on} \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b})\}$

Une situation partiellement instable est une situation pour laquelle un sous-ensemble des combinaisons d'entrées laisse cette situation stable et un autre sous-ensemble rend cette situation instable. Aussi, pour chaque situation $S_{Act}(l')$ atteignable depuis la localité l par le franchissement d'un ensemble de transitions simultanément franchissables $T_{SimFire}$, deux expressions booléennes définissant la condition de stabilité $E_{Stab}^{T_{SimFire}}(S_{Act}(l'))$ et la condition de non-stabilité $E_{Tran}^{T_{SimFire}}(S_{Act}(l'))$ de cette situation $S_{Act}(l')$, après franchissement de $T_{SimFire}$, peuvent être définies comme suit :

$$E_{Stab}^{T_{SimFire}}(S_{Act}(l')) = E_{Fire}^{l}(T_{SimFire}) \cdot \overline{\sum_{t \in T_{Fire}(l')} E_{Fire}^{l'}(t)} \tag{2.25}$$

$$E_{Tran}^{T_{SimFire}}(S_{Act}(l')) = E_{Fire}^{l}(T_{SimFire}) \cdot \sum_{t \in T_{Fire}(l')} E_{Fire}^{l'}(t) \tag{2.26}$$

La condition de stabilité est vraie si et seulement si la situation $S_{Act}(l')$ est atteinte depuis $S_{Act}(l)$ par le franchissement d'un ensemble $T_{SimFire}$, et qu'aucune transition validée n'est franchissable depuis cette situation $S_{Act}(l')$. La condition de non-stabilité est vraie si et seulement si la situation $S_{Act}(l')$ est atteinte depuis $S_{Act}(l)$ par le franchissement d'un ensemble $T_{SimFire}$, et qu'au moins une transition validée est franchissable depuis cette situation $S_{Act}(l')$.

Si l'expression $E_{Tran}^{T_{SimFire}}(S_{Act}(l'))$ est toujours fausse ($E_{Tran}^{T_{SimFire}}(S_{Act}(l')) = 0$), alors la situation est stable. À l'opposé, si l'expression $E_{Stab}^{T_{SimFire}}(S_{Act}(l'))$ est toujours fausse ($E_{Stab}^{T_{SimFire}}(S_{Act}(l')) = 0$), alors la situation est totalement instable. Sinon, si aucune des deux expressions n'est pas toujours fausse ($E_{Stab}^{T_{SimFire}}(S_{Act}(l')) \neq 0$ et $E_{Tran}^{T_{SimFire}}(S_{Act}(l')) \neq 0$), le couple $(S_{Act}(l'), E_{Fire}^{l}(T_{SimFire}))$ doit être séparée en deux couples $(S_{Act}(l'), E_{FireS}^{l}(T_{SimFire}))$ et $(S_{Act}(l'), E_{FireT}^{l}(T_{SimFire}))$, avec $E_{FireS}^{l}(T_{SimFire})$ respectant la condition de stabilité $E_{Stab}^{T_{SimFire}}(S_{Act}(l'))$, et $E_{FireT}^{l}(T_{SimFire})$ respectant la condition de non-stabilité $E_{Tran}^{T_{SimFire}}(S_{Act}(l'))$. Le calcul de recherche de stabilité doit alors être réitéré pour le couple $(S_{Act}(l'), E_{FireT}^{l}(T_{SimFire}))$, jusqu'à atteindre la stabilité.

Ainsi, pour l_{exB}, après recherche de la stabilité, on obtient $SS^{l_{exB}}(T_{SimFire}) =$

$\{(\{A3, 22, 32\}, \overline{on} \cdot \overline{dp} \cdot (\overline{z} + a + b)), (\{F1, S20, 32\}, on \cdot \overline{dp} \cdot z \cdot \overline{a} \cdot \overline{b}), (\{F1, 22, S30\}, on \cdot dp \cdot (\overline{z} + a + b)), (\{A3, S20, 32\}, \overline{on} \cdot \overline{dp} \cdot z \cdot \overline{a} \cdot \overline{b}), (\{A3, 22, S30\}, \overline{on} \cdot dp \cdot (a + b + \overline{z})), (\{F1, S20, S30\}, on \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b} \cdot v), (\{F1, E40\}, on \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b} \cdot \overline{v}), (\{A3, S20, S30\}, \overline{on} \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b} \cdot v), (\{A3, E40\}, \overline{on} \cdot dp \cdot z \cdot \overline{a} \cdot \overline{b} \cdot \overline{v})\}$

4.3.1.3 Phase 3

Pour chaque situation stable $S_{Act}(l)$ atteinte suite au franchissement de $T_{SimFire}$, il est nécessaire de déterminer les sorties émises. Une sortie continue o est émise pour une situation stable $S_{Act}(l)$ atteinte suite au franchissement de $T_{SimFire}$ ($S_{Act}(l)$ vérifiant $E_{Stab}^{T_{SimFire}}(S_{Act}(l)) \neq 0$) si et seulement si la condition d'assignation de cette sortie $E_{Emit}(o)$ vérifie :

$$E_{Emit}(o) \cdot \prod_{s \in S_{Act}(l)} X(s) \cdot \prod_{\substack{s \in S_G \\ s \notin S_{Act}(l)}} \overline{X(s)} \cdot E_{Stab}^{T_{SimFire}}(S_{Act}(l)) \neq 0 \tag{2.27}$$

Comme cela a été effectué pour les réceptivités associées aux transitions, $E_{Emit}^{l}(o)$ représente les combinaisons des variables d'entrée pour lesquelles la sortie o est émise pour la situation $S_{Act}(l)$. Cette expression booléenne est obtenue à partir de $E_{Emit}(o)$ en substituant les variables d'activité d'étapes par leur valeur correspondante (Vrai/Faux) en fonction de la situation $S_{Act}(l)$.

$$E_{Emit}^{l}(o) = E_{Emit}(o) \quad \text{où} \begin{cases} \forall s \in S_{Act}(l) & X(s) = 1 \\ \forall s \notin S_{Act}(l) & X(s) = 0 \end{cases} \tag{2.28}$$

Soit $SO_{SimEmit}(S_{Act}(l), E_{Stab}^{T_{SimFire}}(S_{Act}(l)))$ l'ensemble des couples ($O_{SimEmit}$, $E_{Emit}^{l}(O_{SimEmit})$), où $O_{SimEmit}$ est un ensemble de sorties continues simultanément émises pour la situation $S_{Act}(l)$ atteinte suite au franchissement de $T_{SimFire}$, et $E_{Emit}^{l}(O_{SimEmit})$ est la condition booléenne devant être satisfaite pour permettre l'émission de ces sorties.

L'ensemble $SO_{SimEmit}(S_{Act}(l), E_{Stab}^{T_{SimFire}}(S_{Act}(l)))$ est défini par :

$$SO_{SimEmit}(S_{Act}(l), E_{Stab}^{T_{SimFire}}(S_{Act}(l))) = \Big\{(O_{SimEmit}, E_{Emit}^{l}(O_{SimEmit})) \quad | \\ [O_{SimEmit} \subset O_C] \wedge [E_{Emit}^{l}(O_{SimEmit}) \neq 0]\Big\}$$
$$\text{avec } E_{Emit}^{l}(O_{SimEmit}) = \\ \left(\prod_{o \in O_{SimEmit}} E_{Emit}^{l}(o)\right) \cdot \left(\prod_{\substack{o \in O_C \\ o \notin O_{SimEmit}}} \overline{E_{Emit}^{l}(o)}\right) \cdot E_{Stab}^{T_{SimFire}}(S_{Act}(l)) \tag{2.29}$$

L'ensemble des sorties émises pilotées par des actions mémorisées est déterminé en analysant la séquence de franchissement entre deux situations stables. Lorsqu'une action mémorisée est associée à une étape qui appartient à une situation (stable ou instable) traversée par cette séquence, la sortie correspondante est affectée à sa valeur (0 ou 1), en fonction du type d'action (Set ou Reset). Lorsque plusieurs actions mémorisées affectant la même sortie sont exécutées durant une séquence, seul l'effet de la dernière action est retenu.

4.3.2 Détermination des évolutions sans franchissement de transition

Les évolutions sans franchissement de transition correspondent à un changement de l'ensemble des sorties émises tout en conservant la même situation active. Ces évolutions sont déterminées par le calcul des ensembles des sorties continues simultanément émises pour les combinaisons d'entrée satisfaisant l'équation (2.30) (aucune transition validée pour $S_{Act}(l)$ ne peut être franchie).

$$E_{Stab}(l) = \overline{\sum_{t \in T_{Fire}(l)} E_{Fire}^{l}(t)} \tag{2.30}$$

L'expression de $E_{Stab}(l)$ est légèrement différente de celle de $E_{Stab}^{T_{SimFire}}(S_{Act}(l))$, car E_{Stab} définit la condition de stabilité sans franchissement de transition alors que $E_{Stab}^{T_{SimFire}}$ définit la condition de stabilité après franchissement d'un ensemble $T_{SimFire}$ de transitions simultanément franchissables. De ce fait, la condition d'évolution $E_{Fire}^{l}(T_{SimFire})$ n'apparait pas dans l'expression de E_{Stab}.

4.4 Illustration sur les exemples

Un logiciel, appelé TELOCO (pour TEst of LOgic COntrollers), a été développé durant cette thèse et est disponible librement à l'adresse suivante : http://www.lurpa.ens-cachan.fr/isa/teloco/. Cet outil permet de construire automatiquement l'ALS à partir d'une description textuelle d'une spécification Grafcet, en utilisant les définitions présentées dans ce chapitre. Ce logiciel permet également de traduire un ALS en une machine de Mealy équivalente et de générer une séquence de test à partir de cette machine de Mealy. La méthode de transcription de l'ALS en une machine de Mealy est détaillée dans la section 5 ; la méthode de construction de la séquence de test est détaillée dans le chapitre 3.

Pour l'exemple présenté figure 2.6, l'ALS de cette spécification Grafcet contient 3 localités stables et une localité initiale instable :

$$
\begin{aligned}
&l_{Init} : (\{1\}, 0)\\
&l_1 : (\{1\}, \varnothing, po \cdot (tel + voit) + pf \cdot \overline{tel})\\
&l_2 : (\{2\}, \{Ouvrir\}, \overline{po})\\
&l_3 : (\{3\}, \{Fermer\}, \overline{pf} \cdot \overline{tel} \cdot \overline{voit})
\end{aligned}
$$

6 évolutions sont possibles entre localités stables et 3 évolutions depuis la localité initiale vers une localité stable ; aucune évolution n'est totalement instable. Le tableau 2.2 présente ces évolutions, en précisant, pour chaque évolution, sa condition d'évolution ainsi que la séquence des ensembles de transitions simultanément franchissables correspondante.

Pour l'exemple présenté figure 2.7 l'ALS de cette spécification Grafcet contient 64 localités stables et une localité initiale instable ainsi que 389 évolutions entre localités stables et 4 évolutions depuis la localité initiale vers une localité stable ; aucune évolution n'est totalement instable. La construction de cet ALS est effectuée en approximativement 800 ms, ce qui reste très raisonnable pour une spécification de cette taille.

Dans cette spécification Grafcet, la localité $l_{exB} = (\{F1, 22, 32\}, \{MT, VC, ILV\}, on \cdot \overline{dp} \cdot \overline{z})$ est atteignable depuis la localité initiale ; après calcul des évolutions sans franchissement de transition, 13 évolutions sont possibles depuis cette localité. Le tableau 2.3 présente ces évolutions, ligne par ligne. La première colonne donne la localité atteinte lorsque cette évolution est effectuée ; la seconde colonne indique la condition d'évolution (lorsque la localité source est active et que cette condition est vraie, alors l'évolution

Localité amont	Localité aval	Condition d'évolution	Séquence d'ensembles $T_{SimFire}$
l_{Init}	l_1	$po \cdot (tel + voit) + pf \cdot \overline{tel}$	$\varnothing$
l_{Init}	l_2	$\overline{po} \cdot (tel + \overline{pf} \cdot voit)$	$\langle\{t1\}\rangle$
l_{Init}	l_3	$\overline{pf} \cdot \overline{tel} \cdot \overline{voit}$	$\langle\{t2\}\rangle$
l_1	l_2	$\overline{po} \cdot (tel + \overline{pf} \cdot voit)$	$\langle\{t1\}\rangle$
l_1	l_3	$\overline{pf} \cdot \overline{tel} \cdot \overline{voit}$	$\langle\{t2\}\rangle$
l_2	l_1	$po \cdot (pf + tel + voit)$	$\langle\{t3\}\rangle$
l_2	l_3	$po \cdot \overline{pf} \cdot \overline{tel} \cdot \overline{voit}$	$\langle\{t3\}, \{t2\}\rangle$
l_3	l_2	$\overline{po} \cdot (tel + \overline{pf} \cdot voit)$	$\langle\{t4\}\rangle$
l_3	l_1	$po \cdot (tel + voit) + pf \cdot \overline{tel}$	$\langle\{t5\}\rangle$

Tableau 2.2 – Ensemble des évolutions pour la spécification Grafcet présentée figure 2.6

est effectuée) ; la troisième colonne indique la séquence des ensembles de transitions simultanément franchies depuis la localité source pour atteindre la localité d'arrivée ; la dernière colonne indique les actions continues dont la condition d'assignation est validée pour cette localité. Nous pouvons remarquer que la dernière ligne correspond à une évolution sans franchissement de transition ; seul l'ensemble des sorties émises est modifié (la sortie 'VC' n'est plus assignée). Les autres lignes décrivent des évolutions avec franchissement de transitions ; par exemple, pour la dixième évolution, deux ensembles de transitions sont successivement franchis : depuis la localité source, les transitions $t22$ et $t32$ sont d'abord simultanément franchies, conduisant à une localité non stable, ensuite la transition $t5$ est franchie et conduit à la localité stable $\{\{F1, E40\}, \{MR, ILV\}, (on \cdot \overline{v})\}$.

Localité atteinte	Condition d'évolution	Séquence d'ensembles $T_{SimFire}$	Actions continues assignées
$\big(\{A3,22,32\},\{MT,VC,ILO\},\overline{dp}\cdot\overline{z}\big)$	$\overline{on}\cdot\overline{dp}\cdot\overline{z}$	$\langle\{t2\}\rangle$	$\{(A3,ILO,1),(22,VC,\overline{z})\}$
$\big(\{A3,22,32\},\{MT,ILO\},\overline{dp}\cdot z\cdot(a+b)\big)$	$\overline{on}\cdot\overline{dp}\cdot z\cdot(a+b)$	$\langle\{t2\}\rangle$	$\{(A3,ILO,1)\}$
$\big(\{F1,S20,32\},\{MT,ILV\},on\cdot\overline{dp}\big)$	$on\cdot\overline{dp}\cdot z\cdot\overline{a}\cdot\overline{b}$	$\langle\{t22\}\rangle$	$\{(F1,ILV,\overline{X1})\}$
$\big(\{F1,22,S30\},\{VC,ILV\},on\cdot\overline{z}\big)$	$on\cdot dp\cdot\overline{z}$	$\langle\{t32\}\rangle$	$\{(F1,ILV,\overline{X1}),(22,VC,\overline{z})\}$
$\big(\{F1,22,S30\},\{ILV\},on\cdot z\cdot(a+b)\big)$	$on\cdot dp\cdot z\cdot(a+b)$	$\langle\{t32\}\rangle$	$\{(F1,ILV,\overline{X1})\}$
$\big(\{A3,S20,32\},\{MT,ILO\},\overline{dp}\big)$	$\overline{on}\cdot\overline{dp}\cdot z\cdot\overline{a}\cdot\overline{b}$	$\langle\{t2,t22\}\rangle$	$\{(A3,ILO,1)\}$
$\big(\{A3,22,S30\},\{VC,ILO\},\overline{z}\big)$	$\overline{on}\cdot dp\cdot\overline{z}$	$\langle\{t2,t32\}\rangle$	$\{(A3,ILO,1),(22,VC,\overline{z})\}$
$\big(\{A3,22,S30\},\{ILO\},(a+b)\cdot z\big)$	$\overline{on}\cdot dp\cdot z\cdot(a+b)$	$\langle\{t2,t32\}\rangle$	$\{(A3,ILO,1)\}$
$\big(\{F1,S20,S30\},\{ILV\},on\cdot v\big)$	$on\cdot dp\cdot z\cdot\overline{a}\cdot\overline{b}\cdot v$	$\langle\{t22,t32\}\rangle$	$\{(F1,ILV,\overline{X1})\}$
$\big(\{F1,E40\},\{MR,ILV\},on\cdot\overline{v}\big)$	$on\cdot dp\cdot z\cdot\overline{a}\cdot\overline{b}\cdot\overline{v}$	$\langle\{t22,t32\},\{t5\}\rangle$	$\{(F1,ILV,\overline{X1}),(E40,MR,1)\}$
$\big(\{A3,S20,S30\},\{ILO\},v\big)$	$\overline{on}\cdot dp\cdot z\cdot\overline{a}\cdot\overline{b}\cdot v$	$\langle\{t2,t22,t32\}\rangle$	$\{(A3,ILO,1)\}$
$\big(\{A3,E40\},\{MR,ILO\},\overline{v}\big)$	$\overline{on}\cdot dp\cdot z\cdot\overline{a}\cdot\overline{b}\cdot\overline{v}$	$\langle\{t2,t22,t32\},\{t5\}\rangle$	$\{(A3,ILO,1),(E40,MR,1)\}$
$\big(\{F1,22,32\},\{MT,ILV\},on\cdot\overline{dp}\cdot z\cdot(a+b)\big)$	$on\cdot\overline{dp}\cdot z\cdot(a+b)$	$\varnothing$	$\{(F1,ILV,\overline{X1})\}$

Tableau 2.3 – Ensemble des évolutions depuis la localité $l_{exB}=(\{F1,22,32\},\ \{MT,VC,ILV\},\ on\cdot\overline{dp}\cdot\overline{z})$ pour la spécification Grafcet présentée figure 2.7

5 Transcription de l'ALS en une machine de Mealy équivalente

Le but de cette section est de définir les règles de transcription d'un ALS construit à partir d'une spécification Grafcet en une machine de Mealy équivalente. Nous rappelons qu'une condition booléenne est associée à chaque évolution de l'ALS (Tableau 2.3). Par opposition, une machine de Mealy est un modèle à base d'événements, c'est-à-dire qu'une transition de la machine de Mealy est franchie si un événement d'entrée apparaît et non pas si une condition booléenne est vraie. Le problème scientifique à résoudre peut donc être énoncé comme suit : *"comment transcrire le comportement d'une machine à états dont les conditions d'évolution sont définies par des expressions booléennes en une machine à états à base d'événements, sans perte de sémantique ?"*.

La spécification Grafcet et son ALS sont définis en utilisant des expressions booléennes dépendant des variables logiques d'entrée. À chaque évolution de l'ALS est associée une condition d'évolution, cette condition est une expression booléenne construite à partir des n variables logiques d'entrée de l'ALS. Cette expression booléenne permet de représenter de façon symbolique (définition en compréhension) la condition que doivent satisfaire les n variables logiques d'entrée de l'ALS pour permettre d'effectuer cette évolution. Par opposition, les machines de Mealy utilisent une représentation énumérée (expression en extension) pour définir les transitions entre les différents états. Le comportement de cette machine de Mealy doit donc être défini pour chacune des 2^n combinaisons des entrées logiques.

5.1 Formalisation de la machine de Mealy équivalente

Une machine de Mealy $M : (I_M, O_M, S_M, s_{InitM}, \delta_M, \lambda_M)$ peut être construite depuis tout Automate des Localités Stables $ALS : (I_{ALS}, O_{ALS}, L, l_{Init}, Evol)$. Une machine de Mealy dont le comportement est strictement identique au comportement de l'ALS est définie comme suit :

- I_M : alphabet d'entrée. Cet alphabet contient $2^{|I_{ALS}|}$ éléments. Chaque élément i_i de cet alphabet représente une combinaison différente des variables d'entrée de I_{ALS}. Chaque élément i_i est associé à un minterme construit à partir des variables de I_{ALS}. Soit $m_I(i_i)$ le minterme associé à l'événement d'entrée i_i. Par construction,

les mintermes sont distincts et vérifient la propriété suivante :

$$\forall(i_i, i_j) \in I_M^2 \ [m_I(i_i) \cdot m_I(i_j) = 0] \tag{2.31}$$

Par exemple, dans le cas de la figure 2.7 :
$m_I(i_{256}) = on \cdot \overline{dcy} \cdot \overline{dp} \cdot \overline{mh} \cdot \overline{mb} \cdot \overline{z} \cdot \overline{a} \cdot \overline{b} \cdot \overline{v}$
$m_I(i_{323}) = on \cdot \overline{dcy} \cdot dp \cdot \overline{mh} \cdot \overline{mb} \cdot \overline{z} \cdot \overline{a} \cdot b \cdot v$

– O_M : alphabet de sortie. De façon similaire à I_M, cet alphabet contient $2^{|O_{ALS}|}$ éléments. Chaque élément o_i de cet alphabet représente une combinaison différente des variables de sortie O_{ALS}. Chaque élément o_i est associé à un minterme construit à partir des variables de O_{ALS}. Soit $m_O(o_i)$ le minterme associé à l'événement de sortie o_i.

– S_M : ensemble des états. Un état de la machine de Mealy M est associé à chaque localité de l'Automate des Localités Stables ALS. Afin de faciliter la compréhension de la méthode de transcription, les états de M et les localités associées de ALS seront notés de la même manière. Aussi, l'ensemble S_M est identique à l'ensemble L de ALS.

$$S_M \equiv L \tag{2.32}$$

– s_{InitM} : état initial. Cet état est associé à la localité initiale l_{Init} de ALS.

$$s_{InitM} \equiv l_{Init} \tag{2.33}$$

La fonction de transition δ_M et la fonction de sortie λ_M sont définies sur les ensembles de base I_{ALS}, O_{ALS}, L, $Evol$ et I_M, O_M, S_M. Elles doivent être définies de telle sorte que la machine de Mealy obtenue soit déterministe et complètement spécifiée, c'est-à-dire :

$$\forall(s, i) \in \{S_M \cup s_{InitM}\} \times I_M \quad \begin{cases} \exists!\delta_M(s, i) \in S_M \\ \exists!\lambda_M(s, i) \in O_M \end{cases} \tag{2.34}$$

– La fonction de transition δ_M de la machine M est définie comme suit :

$$\forall s \in S_M, \forall i \in I_M,$$
$$\left[\begin{array}{l} \textbf{si } \exists e \in Evol \left[\left\{ \begin{array}{l} l_U(e) = s \\ E_{Evol(I_G)}(e) \cdot m_I(i) = m_I(i) \end{array} \right. \right] \quad \textbf{alors } [\delta_M(s,i) = l_D(e)] \\ \textbf{sinon } [\delta_M(s,i) = s] \end{array} \right] \tag{2.35}$$

La fonction de transition δ_M est complètement spécifiée puisqu'une valeur est associée à chaque couple $(s,i) \in S_M \times I_M$. Cette valeur est unique car les évolutions partant d'une localité l sont mutuellement exclusives (voir équation 2.11).

La construction de la fonction de sortie λ_M repose sur l'hypothèse selon laquelle la valeur des sorties émises dépend uniquement de la localité active de l'ALS. Cette hypothèse est vérifiée, par définition de la localité, par opposition à la notion de situation d'une spécification Grafcet qui ne prend en compte que les étapes actives à un instant t.

– La fonction de sortie λ_M est définie comme suit :

$$\begin{array}{l} \forall s \in S_M \cup s_{InitM}, \forall i \in I_M \quad [\lambda_M(s,i) = o_j] \\ \textbf{avec } o_j \in O_M \textbf{ tel que :} \\ \left\{ \begin{array}{l} \forall v \in O_{Emit}(\delta_M(s,i))\, [m_O(o_j) \cdot v = m_O(o_j)] \\ \forall v \in \{O_{ALS} - O_{Emit}(\delta_M(s,i))\}\, [m_O(o_j) \cdot \bar{v} = m_O(o_j)] \end{array} \right. \end{array} \tag{2.36}$$

Les définitions proposées pour δ_M et λ_M assurent le déterminisme d'évolution ainsi que le déterminisme d'émission des sorties. La machine de Mealy obtenue est donc déterministe et complètement spécifiée. De plus, chaque état de cette machine de Mealy est atteignable depuis l'état initial, car cette propriété est vérifiée par construction sur l'ALS.

D'après ces définitions des fonctions $\delta_M(s,i)$ et $\lambda_M(s,i)$, chaque couple $(s,i) \in S_M \times I_M$ vérifie :

$$\forall ((s,i),(s',i')) \in (S_M \times I_M)^2, \quad \delta_M(s,i) = \delta_M(s',i') \Rightarrow \lambda_M(s,i) = \lambda_M(s',i') \tag{2.37}$$

Ce résultat assure la minimalité de la machine de Mealy obtenue. Les règles de transcription détaillées ci-dessus sont également implantées dans le logiciel TELOCO. L'algorithme 1 décrit l'implantation de ces règles.

Enfin, la taille de la machine de Mealy est facilement calculable. La machine de Mealy ainsi construite contient autant d'états qu'il y a de localités dans l'ALS. Le nombre de ses transitions dépend uniquement du nombre de localités et du nombre de variables d'entrée

Algorithm 1 Construction de la machine de Mealy

Inputs : SLA : (I_{SLA},O_{SLA},L,l_{Init},$Evol$)
Outputs : M : (I_M,O_M,S_M,s_{InitM},δ_M,λ_M)

```
/* Construction de I_M et O_M : */
for all j = 0 to 2^|I_SLA| do
    Construire le minterme m_I(i_j) et ajouter i_j to I_M
end for
for all k = 0 to 2^|O_SLA| do
    Construire le minterme m_O(o_k) et ajouter o_k to O_M
end for
/* Construction de S_M et s_initM */
for all l ∈ L do
    if l = l_Init then
        Associer s_InitM à l et ajouter s_InitM to S_M
    else
        Associer s à l et ajouter s to S_M
    end if
end for
/* Construction de δ_M et λ_M */
for all s ∈ S_M do
    for all j ∈ I_M do
        Affecter la valeur par défaut de δ_M(s,j) : δ_M(s,j) := s
        for all e ∈ Evol do
            if s = l_U(e) then
                if E_Evol(I_G)(e) · m_I(i_j) = m_I(i_j) then
                    δ_M(s,j) := l_D(e)
                    break /* Arrêter le calcul de δ_M(s,j) */
                end if
            end if
        end for
        λ_M(s,j) := o_k où m_O(o_k) satisfait et satisfait seulement la condition d'émission
        des sorties pour la localité l associée à s
    end for
end for
```

de l'ALS (voir relations (2.38)). Il est important de noter que le nombre d'évolutions de l'ALS n'a aucune influence sur la taille de la machine de Mealy obtenue.

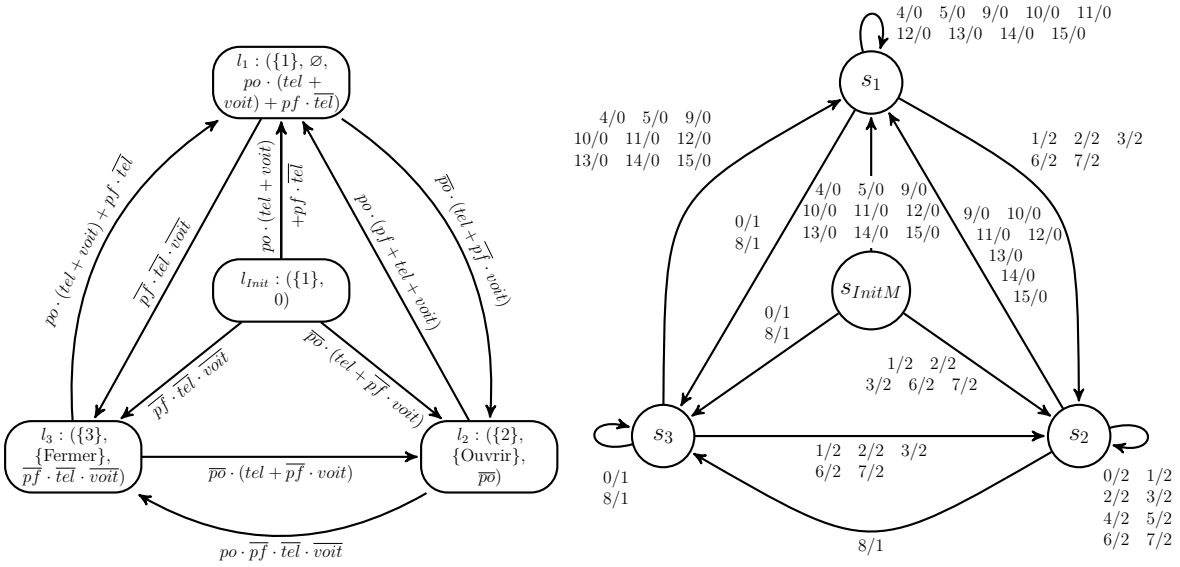
$$\begin{cases} |états| = |L| \\ |transitions| = |L| \cdot 2^{|I_G|} \end{cases} \tag{2.38}$$

5.2 Illustration sur les exemples

La machine de Mealy décrivant le comportement de la spécification Grafcet donnée figure 2.6 contient 3 états (plus 1 état initial) et 48 transitions entre états non initiaux,

car l'ALS contient 3 localités stables et possède 4 entrées logiques.

La figure 2.9 donne une présentation graphique de l'ALS et de la machine de Mealy équivalente pour l'exemple A (figure 2.6). Pour des raisons de lisibilité, la machine de Mealy est représentée de manière condensée : chaque arc représente plusieurs transitions entre deux états de la machine de Mealy, chaque étiquette (indice du symbole d'entrée/indice du symbole de sortie) associée à un arc représente une transition de la machine de Mealy.



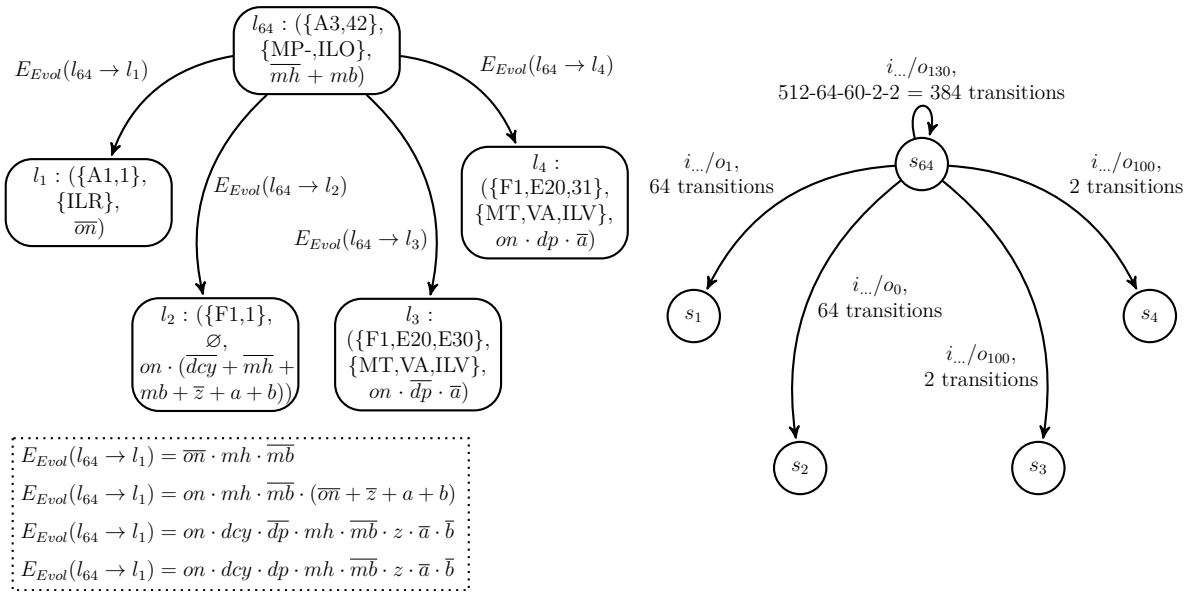
(a) Automate des Localités Stables (b) Machine de Mealy

Figure 2.9 – ALS et machine de Mealy pour l'exemple de la figure 2.6

La machine de Mealy décrivant le comportement de la spécification Grafcet donnée figure 2.7 contient 64 états (plus 1 état initial) et 32 768 transitions entre états non initiaux, car l'ALS contient 64 localités et possède 9 entrées logiques. Pour cet exemple, la transcription de l'ALS vers la machine de Mealy équivalente dure approximativement 300 ms, ce qui est très peu, en utilisant le logiciel TELOCO. La figure 2.10 présente une partie de l'ALS et une partie de la machine de Mealy obtenus à partir de cet exemple.

Synthèse

Les contributions apportées dans ce chapitre peuvent être séparées en deux parties : les contributions directes pour le test de conformité d'un contrôleur logique pilotant un SCC, et les contributions indirectes, utiles à d'autres phases de cycle de développement d'un SCC.



(a) Automate des Localités Stables (b) Machine de Mealy

Figure 2.10 – Extrait de l'ALS et la machine de Mealy pour l'exemple de la figure 2.7

Les contributions directes comprennent en premier lieu la définition formelle d'une spécification Grafcet par l'utilisation d'un modèle à états (l'Automate des Localités Stables) représentant tous les états stables et toutes les évolutions possibles dans une spécification Grafcet.

La deuxième contribution directe est la définition d'une méthode de transcription de l'ALS en une machine de Mealy, modèle ad-hoc pour le test de conformité réalisé en boîte noire.

De manière indirecte, la formalisation d'une spécification Grafcet par un ALS permet également de vérifier certaines propriétés fonctionnelles. Le recours à cette formalisation peut donc également être effectué plus tôt dans la phase de conception de la spécification afin de détecter les incohérences et les propriétés non vérifiées lors de différentes modifications faites sur la spécification Grafcet. Ainsi, cette formalisation contribue de manière certaine à l'amélioration de la sûreté de fonctionnement des contrôleurs logiques industriels.

La formalisation d'une spécification Grafcet par un ALS peut également être utilisée pour faciliter l'implantation de cette spécification dans un API. Ce dernier point sera discuté dans le chapitre suivant.

Enfin, il est important de souligner que les méthodes définies dans ce chapitre ont été implantées dans le logiciel TELOCO, disponible à l'adresse : `http://www.lurpa.ens-cachan.fr/isa/teloco/`. L'implantation des algorithmes conduit à des temps de

transformations de modèles très courts et donc compatibles avec une utilisation de ce formalisme aussi bien en phase de vérification d'une spécification par rapport à ses propriétés qu'en phase de validation par le test de conformité.

Chapitre 3

Génération de séquence de test et mise en œuvre sur banc de test

Sommaire

Introduction

La méthode de formalisation du langage de spécification présentée dans le chapitre précédent permet la génération automatique d'une machine de Mealy à partir d'une spécification Grafcet. En associant ce résultat aux travaux existants de la communauté sur le test de conformité des machines de Mealy, nous sommes en mesure d'effectuer un test de conformité d'un API spécifié en Grafcet.

Les travaux présentés dans ce chapitre ont fait l'objet de communications en conférences nationale et internationales (Provost *et al.* (2009a,c) ; Chériaux *et al.* (2010) ; Provost *et al.* (2011c)). Le logiciel et le banc de test développés ont également fait l'objet d'une démonstration lors des journées démonstrateurs de la section Automatique du Club EEA (Provost *et al.* (2010a)).

Ce chapitre est organisé comme suit :

- la première section présente les méthodes existantes permettant la génération de séquence de test à partir d'une machine de Mealy ;
- la deuxième section présente l'application de la méthode du Tour de Transition (méthode TT) à la machine de Mealy obtenue en appliquant le méthode de formalisation présentée dans le chapitre précédent ;
- la troisième section décrit le banc de test qui a été développé et la configuration expérimentale utilisée pour la réalisation des campagnes d'essais ;
- la quatrième section précise la mise en œuvre de l'exécution du test de conformité dans le cas d'un Automate Programmable Industriel (API) ;
- la cinquième partie détaille les résultats expérimentaux obtenus et présente une analyse de ces résultats, incluant une explication des erreurs de verdict constatées ;
- la sixième partie présente les contraintes induites par les séquences de test, au vu des résultats expérimentaux obtenus ;
- enfin, la dernière section décrit l'extension de ces résultats aux Automates Programmables Industriels dédiés à la Sécurité (APIdS).

1 Le test de conformité de machine de Mealy

Dans cette section, nous rappelons les techniques de base concernant la génération de séquence de test à partir d'une machine de Mealy. Pour plus de détails sur ces techniques,

le lecteur est invité à se reporter aux références suivantes : Lee et Yannakakis (1996) et Dorofeeva *et al.* (2010).

Nous rappelons qu'une machine de Mealy M est formellement définie par un 6-uplet $(S_M, s_{InitM}, I_M, O_M, \delta_M, \lambda_M)$ tel que :

- S_M est un ensemble fini d'états ;
- $s_{InitM} \in S_M$ est l'état initial ;
- I_M est un alphabet fini d'entrées ;
- O_M est un alphabet fini de sorties ;
- $\delta_M : S_M \times I_M \to S_M$ est la fonction de transfert ;
- $\lambda_M : S_M \times I_M \to O_M$ est la fonction de sortie.

1.1 Modèle de fautes retenu

Le problème du test de conformité basé sur le modèle des machines de Mealy peut être formulé de la façon suivante : étant donnée une machine M_{Spec} modélisant la spécification, et une machine M_{Impl} modélisant l'implantation à tester dont on ne peut contrôler et observer que les entrées et sorties, déterminer par une séquence finie de couples (entrée,sortie) si M_{Impl} est équivalente à M_{Spec}.

Pour que ce problème ait une solution, la spécification et l'implantation sont en général supposées minimales et fortement connexes ; le nombre d'états de l'implantation peut est également supposé identique au nombre d'états de la spécification. Avec ces hypothèses, l'équivalence entre une implantation M_{Impl} et sa spécification M_{Spec} revient alors à s'assurer que M_{Impl} n'a aucune faute des types suivants, illustrés figure 3.1 :

- fautes de sortie : depuis un état donné s_s, lors du franchissement d'une transition étiquetée i/o, la sortie produite par M_{Impl} est o' au lieu de la sortie attendue o ;
- fautes de transfert : depuis un état donné s_s, lors du franchissement d'une transition étiquetée i/o, l'état d'arrivée est s'_t au lieu de s_t.

Le niveau de confiance placée dans une technique de test de conformité réside dans sa capacité à détecter les implantations non conformes et à accepter les implantations conformes. Une technique de test est dite valide si et seulement si elle accepte toute implantation conforme, et non-biaisée si et seulement si elle ne rejette aucune implantation conforme (Goodenough et Gerhart (1975)). Une technique de test est dite fiable (ou pertinente) si elle est valide et non-biaisée.

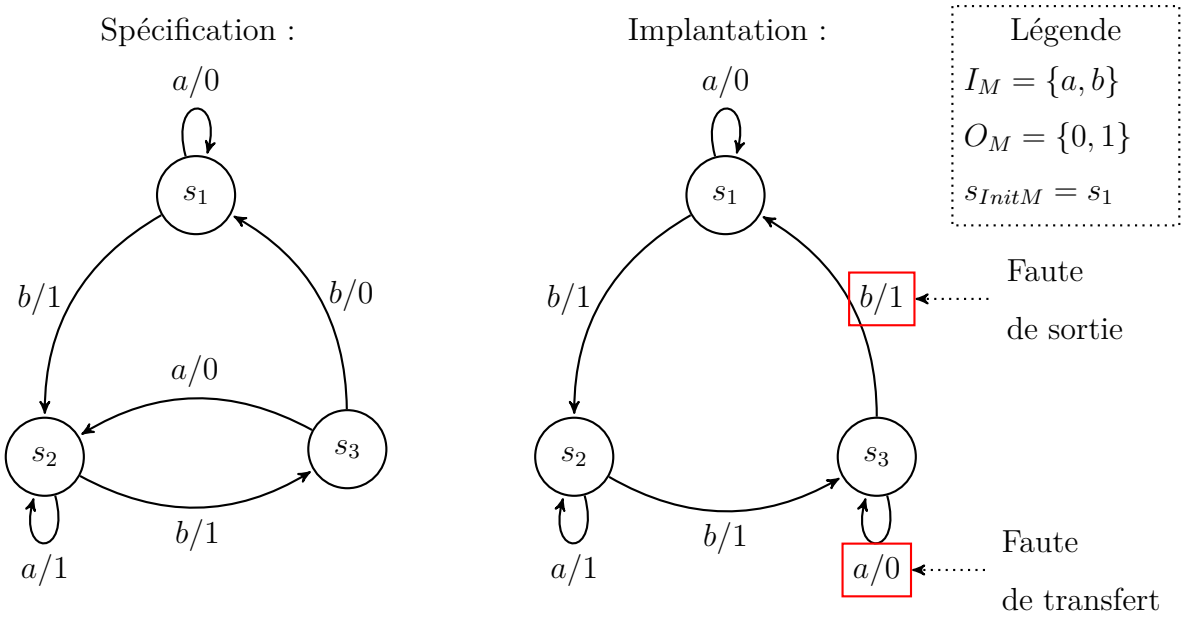


Figure 3.1 – Fautes de transfert et de sortie dans une machine de Mealy

Une séquence de test permettant de tester chacune des transitions d'une machine de Mealy sera dite complète.

1.2 Méthode de génération de séquence de test retenue

Une séquence de test est une suite de cas de test élémentaires. Chaque cas de test élémentaire a pour objectif de tester une transition $s \xrightarrow{i/o} s'$ de M_{Impl} et peut être décomposé de la façon suivante :

- un préambule, qui permet d'amener l'implantation dans un état de départ s_s ;
- un corps de test, qui sollicite l'implantation avec une entrée choisie i et observe la sortie émise o ;
- un postambule, qui identifie l'état d'arrivée s_t.

Le préambule de ces cas de test élémentaires est également appelé séquence de synchronisation. L'existence d'une séquence de synchronisation est garantie si les machines de Mealy sont minimales. La connaissance de l'arbre couvrant (ou "spanning tree") du graphe associé à la machine de Mealy permet de construire ces séquences de synchronisation depuis et vers chaque état de cette machine, ces séquences correspondent au plus court chemin entre deux nœuds du graphe associé. Parmi les différents types de séquences d'identification on peut citer les "homing sequences" et les "synchronizing

sequences" (Lee et Yannakakis (1994)).

Le postambule de ces cas de test élémentaires est également appelé séquence d'identification. Cette identification est effectuée par l'observation d'une séquence de sorties en réponse à une séquence d'entrées sollicitant l'implantation depuis son état d'arrivée s_t.

La manière la plus simple d'identifier l'état d'arrivée est de disposer d'une entrée spécifique pour laquelle le système émet une sortie permettant d'identifier directement son état courant. C'est l'hypothèse faite pour la méthode TT (Naito et Tsunoyama (1981)) ; dans le cas de cette méthode, soit les états sont directement discernables par observation des sorties émises lorsque cet état est actif, soit l'utilisation d'une entrée spécifique, appelée *status*, renvoie l'information permettant d'identifier l'état courant.

Lorsque l'implantation à tester ne dispose pas de cette entrée *status*, il est nécessaire d'utiliser une des méthodes permettant de faire la distinction des états par l'application d'une séquence de distinction. Les méthodes DS, UIO, W, UIOv, Wp, HSI et H sont les méthodes les plus connues. Ces méthodes sont présentées dans Hennie (1964) ; Sabnani et Dahbura (1988) ; Chow (1978) ; Vuong *et al.* (1989) ; Fujiwara *et al.* (1991), Yevtushenko et Petrenko (1990) et Petrenko et Yevtushenko (2005), Koufareva et Dorofeeva (2002) et Dorofeeva *et al.* (2005), respectivement. Les différences entre ces différentes méthodes portent sur les conditions d'existence des séquences à générer ainsi que sur la longueur des séquences générées.

Une synthèse de ces méthodes est présentée dans Lee et Yannakakis (1996) et Dorofeeva *et al.* (2010). Des résultats expérimentaux, obtenus par application de ces méthodes sur des protocoles réels, permettant de comparer ces différentes méthodes sont présentés dans Dorofeeva *et al.* (2010).

La méthode retenue dans le cadre des travaux présentés dans ce chapitre est la méthode TT. Cette méthode a été choisie pour sa simplicité de mise en œuvre pour cette première approche. De plus, il existe de nombreux travaux de la théorie des graphes permettant d'optimiser la longueur de cette séquence. En effet, la recherche d'une séquence de test de longueur minimale en appliquant la méthode TT revient à résoudre un problème bien connu de la théorie des graphes : le problème du postier chinois (Mei-Ko (1962) ; Edmonds et Johnson (1973) ; Thimbleby (2003)).

En contre-partie, l'utilisation de la méthode TT nécessite l'existence d'une entrée *status*. Lors de l'exécution du test de conformité d'un contrôleur logique en boîte noire, celui-ci ne peut être contrôlé que par ses entrées logiques et observé via ses sorties

logiques. Le contrôleur logique ne disposant donc pas d'une entrée spécifique *status*, une hypothèse doit donc être faite concernant la discernabilité des états de l'implantation par observation des sorties logiques émises. La discernabilité des états de l'implantation permet de s'affranchir de l'entrée spécifique *status*, l'identification de l'état courant pouvant être obtenue à tout instant par observation des sorties émises. Cette hypothèse est détaillée dans la section 3.1, équation 3.13.

Cette approche utilisant la méthode TT permet la génération de séquences de test hors ligne. L'approche proposée est donc très différente des techniques de génération de séquences de test "à la volée" utilisées par exemple dans l'outil TGV pour le test de systèmes de transitions étiquetées à entrée/sortie (Input/Output Labelled Transition Systems) (IOLTS) (Fernandez *et al.* (1997) ; Jard et Jéron (2005)). Une approche en ligne ou à la volée a pour avantage de ne pas nécessiter la représentation complète de l'espace d'états. Cette approche peut être utilisée pour générer une séquence de test en parallèle de la construction de l'IOLTS. Cette approche peut également être utilisée pour générer des séquences de test en les adaptant en fonction des sorties observées lors de l'exécution des pas de test précédents. À l'opposé, une approche hors ligne a pour avantage de permettre l'optimisation globale de la séquence, par exemple pour minimiser sa longueur et donc réduire le temps d'exécution de cette séquence.

2 Génération d'une séquence de test

La machine de Mealy construite à partir de l'Automate des Localités Stables (ALS) représente toutes les évolutions possibles du comportement décrit par la spécification Grafcet de départ. Il est possible de construire, à partir de cette machine de Mealy, une séquence de test permettant d'effectuer un test de conformité d'un contrôleur logique devant réaliser l'implantation du comportement décrit par la spécification Grafcet. Comme mentionné précédemment (section 1), plusieurs solutions sont possibles pour obtenir cette séquence. La méthode retenue pour obtenir une séquence de test complète de longueur minimale est la méthode TT (Naito et Tsunoyama (1981)). Le choix de cette méthode suppose l'existence d'une fonction *status*, mais est plus simple à mettre en œuvre car cette fonction *status* facilite la génération des séquences de test et réduit la longueur de ces séquences et donc la durée d'exécution du test. L'hypothèse permettant de vérifier l'existence d'une fonction *status* sera définie dans la section 3.1.

2.1 Définition formelle et propriétés d'une séquence de test

Cette section propose une définition formelle d'une séquence de test, puis définit les propriétés que peut satisfaire une séquence de test.

2.1.1 Définition formelle d'une séquence de test

Pendant l'exécution du test, une séquence de test est perçue comme une liste ordonnée de couples (valuations des entrées, valuations des sorties attendues) qui représente le comportement du contrôleur logique d'un point de vue externe :

$$\left[(v_I^0, v_O^0), (v_I^1, v_O^1), ..., (v_I^n, v_O^n)\right] \in (I_M \times O_M)^* \tag{3.1}$$

Le test de conformité implique que la valuation des sorties attendues soit obtenue à partir du modèle comportemental. Plus précisément, cette valuation des sorties est associée à l'état cible de l'évolution qui est provoquée par la valuation des entrées, depuis un état source donné. Pour obtenir cette valuation des sorties attendues, l'état source s_s et l'état cible s_t doivent obligatoirement être connus. Par conséquent, un pas de test d'une séquence de test de conformité *et* (elementary test) est défini par le 4-uplet suivant :

$$\begin{aligned} &et = (s_s, v_I, s_t, v_O) \in S_M \times I_M \times S_M \times O_M \\ &\text{avec} \begin{cases} s_t = \delta_M(s_s, v_I) \\ v_O = \lambda_M(s_s, v_I) \end{cases} \end{aligned} \tag{3.2}$$

Ainsi, une séquence de test correspond à une liste ordonnée et cohérente de cas de test élémentaires. Une séquence de test TS est dite cohérente si et seulement si l'état source du $k^{ème}$ cas de test élémentaire est égal à l'état cible du $(k-1)^{ème}$ cas de test.

$$\begin{aligned} TS = [(s^0, v_I^0, \delta_M(s^0, v_I^0), \lambda_M(s^0, v_I^0)), \cdots, (s^n, v_I^n, \delta_M(s^n, v_I^n), \lambda_M(s^n, v_I^n))] \mid \\ \forall k > 1, s^k = \delta_M(s^{k-1}, v_I^{k-1}) \end{aligned} \tag{3.3}$$

2.1.2 Propriétés des séquences de test

Une séquence de test peut être :

Propriété 1 : Initialisable, si et seulement si l'état source du premier pas de test est

atteignable lors de l'initialisation, et que la valuation d'entrée utilisée lors de l'initialisation laisse cet état stable.

$$s^0 \in \{s \in S_M \mid \exists v_I^0 \in I_M : s = \delta_M(s_{Init}, v_I^0)\} \tag{3.4}$$

Par définition de la machine de Mealy, l'état s^0 atteint lors de l'initialisation est stable pour la valuation d'entrée v_I^0.

$$\delta_M(s^0, v_I^0) = s^0 \tag{3.5}$$

Propriété 2 : Complète, si et seulement si il y a au moins un pas de test pour chaque élément de la fonction de transition.

$$\forall (s, v_I) \in (S_M \times I_M),\ (s, v_I, \delta_M(s, v_I), \lambda_M(s, v_I)) \in TS \tag{3.6}$$

2.2 Quelques définitions de la théorie des graphes

Les définitions suivantes sont tirées de Cabane (2000).

Dans un graphe orienté, un *chemin* est une suite finie de nœuds $[s_0, s_1, \cdots, s_n]$ telle que, pour tout $0 < i < (n-1)$, il existe un arc de s_i vers s_{i+1}.

Dans un graphe orienté, un *circuit* est un chemin bouclé, c'est-à-dire un chemin qui part d'un nœud et aboutit à ce même nœud.

Un graphe orienté est *fortement connexe* si, de tout nœud, on peut trouver un chemin vers n'importe quel autre nœud.

Dans un graphe orienté, un *circuit eulérien* est un circuit passant exactement une fois par chaque arc du graphe.

Un graphe orienté est *eulérien* s'il admet un circuit eulérien.

Dans un graphe orienté, un *circuit hamiltonien* est un circuit passant exactement une fois par chaque nœud du graphe.

Un graphe orienté est *hamiltonien* s'il contient un circuit hamiltonien.

Dans un graphe orienté, un *circuit pré-eulérien* est un circuit passant au moins une fois par chaque arc du graphe.

Dans un graphe orienté, un *circuit pré-hamiltonien* est un circuit passant au moins une fois par chaque nœud du graphe.

2.3 Brève présentation du problème du postier chinois

Les algorithmes nécessaires à l'implantation de cette méthode sont très répandus dans la communauté de la théorie des graphes, ce qui nous permettra de nous appuyer sur un savoir-faire existant reposant sur un cadre mathématique éprouvé. En effet, la solution à l'optimisation de la méthode TT est une solution particulière, pour un graphe qui représente la structure de la machine de Mealy, d'un problème bien connu de la théorie des graphes : le problème du postier Chinois (Mei-Ko (1962)). La formulation générale de ce problème est la suivante : *"déterminer un circuit de longueur minimale traversant chaque arc du graphe au moins une fois"*.

Un graphe orienté pondéré peut être défini par un couple (V, A) tel que :

- V est l'ensemble des nœuds v.
- A est l'ensemble des arcs a.

Un nœud v peut être défini par un couple $(\Delta_d, label_v)$ tel que :

- $\Delta_d = d^+ - d^-$ qualifie l'équilibre du nœud. Si $\Delta_d = 0$, le nœud est dit équilibré.
  - d^- est le demi-degré intérieur du nœud, soit le nombre d'arcs entrants du nœud.
  - d^+ est le demi-degré extérieur du nœud, soit le nombre d'arcs sortants du nœud.
- $label_v$ est le nom (optionnel) associé au nœud.

Un arc a est défini par un 4-uplet $(v_U, v_D, c_{v_U v_D}, label_{v_U v_D})$ tel que :

- v_U est le nœud origine de l'arc.
- v_D est le nœud destination de l'arc.
- $c_{v_U v_D}$ est le poids de l'arc allant du nœud v_U au nœud v_D.
- $label_{v_U v_D}$ est l'étiquette (optionnelle) associée à l'arc.

Les ensembles D^- et D^+ désignent les ensembles des nœuds non équilibrés, en excès d'arcs sortants et d'arcs entrants, respectivement. D^- et D^+ sont définis comme suit :

$$\begin{aligned} D^- &= \{v | d(v) = d^+(v) - d^-(v) < 0\} \\ D^+ &= \{v | d(v) = d^+(v) - d^-(v) > 0\} \end{aligned} \tag{3.7}$$

Le poids d'un chemin dans ce graphe est défini par la somme des poids des arcs traversés c_{ij} multipliés par le nombre de fois f_{ij} où ces arcs sont traversés, soit :

$$\sum c_{ij} \cdot f_{ij} \tag{3.8}$$

La matrice C, matrice carrée de dimension $|V| \times |V|$, définit l'ensemble des poids c_{ij}

des plus courts chemins. Elle est définie comme suit :

$$C[i][j] = c_{ij} \tag{3.9}$$

De même, l'ensemble des coefficients f_{ij} peut être défini par une matrice carrée F de dimension $|V| \times |V|$ définie comme suit :

$$F[i][j] = f_{ij} \tag{3.10}$$

La formulation mathématique définissant la recherche d'une solution au problème du postier chinois est la suivante :

$$\begin{aligned} &\textbf{à partir des données du graphe :} && C, D^+, D^-, \{\Delta_d\} \\ &\textbf{déterminer } F \textbf{ minimisant :} && \sum_{(i,j)\in V^2} c_{ij} \cdot f_{ij} \\ &\textbf{et vérifiant :} && \begin{cases} f_{ij} & \in \mathbb{N}^+ \\ \sum_{j\in D^+} f_{ij} & = -\Delta_d(i) \\ \sum_{i\in D^-} f_{ij} & = \Delta_d(j) \end{cases} \end{aligned} \tag{3.11}$$

La résolution de ce problème consiste à transformer le graphe de la machine de Mealy en un graphe eulérien, en dupliquant certains arcs, permettant ainsi d'équilibrer les nœuds non équilibrés (Edmonds et Johnson (1973)). En effet, un graphe est eulérien si et seulement si il est fortement connexe et que tous ses nœuds sont équilibrés ($\forall v \in V, d^+(v) = d^-(v)$). Le problème du postier chinois sur un graphe orienté peut alors être résolu en temps polynomial (Papadimitriou (1976)). L'algorithme implanté dans le logiciel TELOCO, reprend celui proposé dans Thimbleby (2003). Cet algorithme utilise une méthode gloutonne pour déterminer, par itération à partir d'une solution initiale, un circuit de longueur minimale.

2.4 Application à une machine de Mealy

Le graphe décrivant la structure (états et transitions entre les états) d'une machine de Mealy transcrite depuis un ALS est construit comme suit :

- Un nœud est associé à chaque état $s \in S_M$ de la machine de Mealy.
- Un arc est associé à chaque couple $(s, i) \in S_M \times I_M$ de la machine de Mealy. Cet

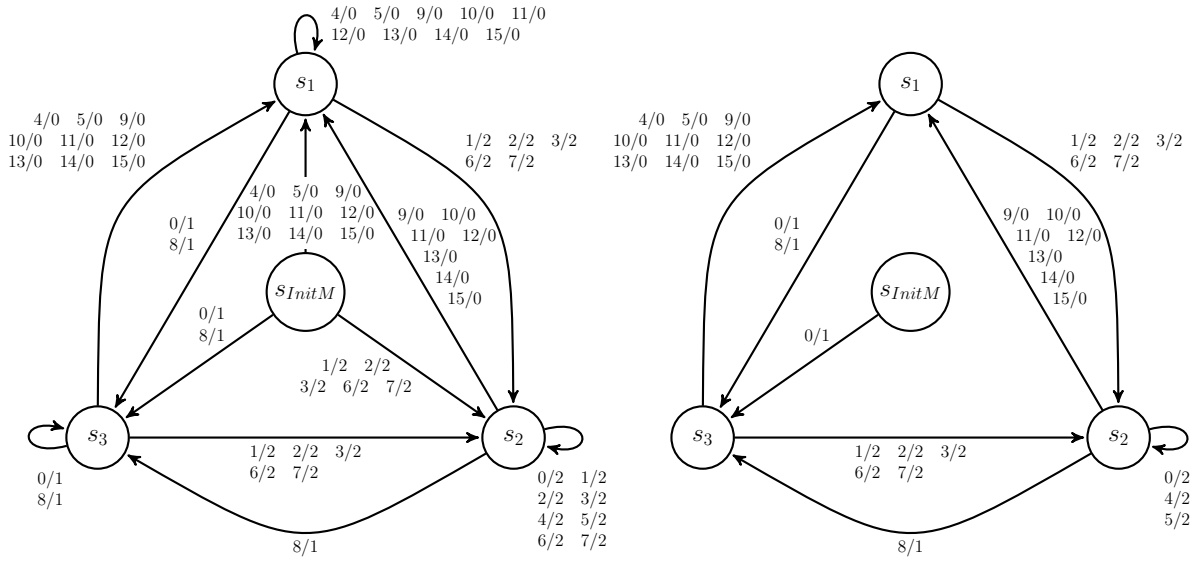
arc a pour origine le nœud correspondant à l'état s et pour destination le nœud correspondant à l'état $\delta_M(s, i)$.

- Aucun nœud n'est associé à l'état s_{InitM} et aucun arc n'est associé aux couples $(s, i) \in s_{InitM} \times I_M$. En effet, l'état s_{InitM} ne doit pas être pris en compte lors du calcul du chemin bouclé car il est impossible de l'atteindre depuis un autre état de la machine de Mealy (il est atteignable uniquement lors de la (ré)initialisation de l'implantation).
- Un arc *init* permet de désigner l'état de départ du chemin bouclé. Cet arc a pour destination un nœud associé à un état s atteignable lors de la (ré)initialisation de l'implantation ($\exists i \in I_M \ : \ \delta_M(s_{InitM}, i) = s$).
- Une étiquette $i/\lambda_M(s, i)$ est associée à chaque arc (arc *init* compris). Le symbole i de l'étiquette associée à l'arc *init* définit la valeur des entrées lors de l'initialisation de l'implantation.

En pratique, lors de l'exécution de la séquence, chaque pas de test permet de tester deux cas de test élémentaires correspondants aux deux arcs du graphe décrivant la structure de la machine de Mealy : l'arc étiqueté i_i/o_j allant de l'état s_s vers l'état s_t et l'arc étiqueté i_i/o_j bouclant sur l'état s_t. Cela est dû au fait que les signaux d'entrée sollicitant l'API sont lus plusieurs fois pendant la durée d'un pas de test, si cette durée est correctement choisie. Ce point sera détaillé dans la section suivante. Afin de prendre en compte cet aspect et ainsi simplifier la structure du graphe, les arcs correspondant aux transitions bouclant sur un même état peuvent être supprimés si et seulement si il existe une transition menant à cet état pour la même combinaison d'entrée.

$$\begin{aligned} &\text{Suppression de l'arc associé au couple } (s, i) \Leftrightarrow \\ &\forall (s, i) \in S_M \times I_M : \begin{cases} \delta_M(s, i) = s \\ \exists s' \in S_M \mid \delta_M(s', i) = s \end{cases} \end{aligned} \tag{3.12}$$

La figure 3.2 illustre le graphe obtenu après simplification de la machine de Mealy. Pour l'initialisation, par défaut seul l'arc associé à la valuation d'entrée i_0 est conservé. En effet, il est plus simple d'effectuer la (ré)initialisation de l'implantation avec cette valuation d'entrée pour laquelle tous les signaux logiques d'entrées sont au niveau bas (0V).



(a) Machine de Mealy

(b) Graphe simplifié

Figure 3.2 – Graphes de la machine de Mealy et graphe simplifié

Il est préférable de supprimer ces arcs dans le graphe décrivant la structure de la machine de Mealy, plutôt que de supprimer les pas de test inutiles dans la séquence de test générée, car cela permet de réduire la taille du graphe et donc de réduire le temps de calcul relatif à la minimisation de la séquence de test. Le logiciel TELOCO effectue automatiquement ces suppressions d'arcs.

La fonction de transition δ_M de la machine de Mealy n'étant par définition pas symétrique[1], le graphe ainsi construit est orienté. Ce graphe est non pondéré car toutes les transitions ont la même priorité, un poids $\in \mathbb{N}^{+*}$ identique peut donc être affecté à chaque arc. Cette définition des poids permet de respecter l'inégalité triangulaire (le chemin allant directement de A vers C est plus court que la somme des chemins allant de A vers B et de B vers C) et est compatible avec la définition du problème de minimisation.

Le calcul d'un chemin bouclé de longueur minimale traversant chaque arc du graphe au moins une fois nécessite la connaissance des plus courts chemins entre toutes les paires de nœuds d'un graphe. Lors du calcul de la matrice des plus courts chemins entre les nœuds d'un graphe, l'algorithme de Floyd-Warshall (Floyd (1962) et Warshall (1962)) détermine et enregistre, sous forme d'une matrice, le premier arc du plus court chemin permettant d'atteindre un nœud depuis un autre. Cet algorithme permet donc de stocker

1. $\delta_M(s_s, i) = s_t \nLeftrightarrow \delta_M(s_t, i) = s_s$

seulement le poids total de chacun des plus courts chemins entre deux nœuds du graphe et le premier arc de chacun de ces plus courts chemins. En effet, seuls les poids des plus courts chemins sont utiles lors des calculs d'optimisation de séquences ; l'ensemble des arcs constituant ces plus courts chemins est uniquement calculé après l'optimisation de la séquence, lors de l'enregistrement de cette séquence dans un format adapté à l'exécution du test. Le logiciel TELOCO est utilisé pour exécuter l'algorithme de minimisation présenté par Thimbleby (2003).

Pour l'exemple de la spécification Grafcet présenté figure 2.6, la machine de Mealy contient 3 états (plus un état initial) et 48 transitions entre états non-initiaux. Le graphe épuré issu de cette machine de Mealy contient 3 nœuds et 32 arcs (figure 3.2). La séquence permettant de tester toutes les transitions de la machine de Mealy contient 43 pas de test. Cette séquence est donnée tableau 3.1.

pas :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
s_s :	s_1	s_3	s_2	s_3	s_2	s_3	s_2	s_2	s_2	s_2	s_3	s_2	s_1	s_3	s_2	s_1	s_3	s_1	s_3	s_1	s_3	s_1
s_t :	s_3	s_2	s_3	s_2	s_3	s_2	s_2	s_2	s_2	s_3	s_2	s_1	s_3	s_2	s_1	s_3	s_1	s_3	s_1	s_3	s_1	s_3
po :	0	0	1	0	1	0	0	0	0	1	0	1	0	0	1	0	1	0	1	0	1	0
pf :	0	1	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	0	0	0	1	0
tel :	0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	0	1	0	0	0
$voit$:	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1	0	1	0	1	0	1	0

23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
s_3	s_1	s_3	s_1	s_3	s_1	s_3	s_1	s_3	s_1	s_3	s_1	s_2	s_1	s_2	s_1	s_2	s_1	s_2	s_1	s_2
s_1	s_3	s_1	s_3	s_1	s_3	s_1	s_3	s_1	s_3	s_1	s_2	s_1	s_2	s_1	s_2	s_1	s_2	s_1	s_2	s_1
0	0	1	0	1	0	1	1	1	0	0	0	1	0	1	0	1	0	1	0	1
1	0	0	0	1	0	0	0	1	0	1	1	1	0	0	0	1	1	0	0	1
0	0	0	0	1	0	1	0	0	0	0	1	0	1	0	0	1	1	1	1	0
1	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0

Tableau 3.1 – Séquence d'entrée de la séquence de test complète pour l'exemple présenté figure 2.6

Pour l'exemple de la spécification Grafcet présenté figure 2.7, la machine de Mealy contient 64 états (plus un état initial) et 32 768 transitions entre états non-initiaux. Le graphe épuré issu de cette machine de Mealy contient 64 nœuds et 24 932 arcs. La séquence de test construite à partir du graphe épuré issue de cette machine de Mealy

contient 73 528 pas de test. Cette séquence est obtenue en *approximativement 1 s* avec le logiciel TELOCO et satisfait, par construction, l'objectif de test (taux de couverture des transitions de la machine de Mealy égale à 100%).

3 Configuration expérimentale

Afin de pouvoir juger de l'applicabilité de la méthode proposée pour des contrôleurs logiques réels, un banc de test spécifique a été développé. Après une description de ce banc, les conditions expérimentales choisies pour effectuer le test de conformité ainsi que la mise en œuvre du test seront détaillées.

3.1 Présentation du banc de test développé

Le banc de test utilisé pour ces travaux a été conçu à partir des retours d'expérience d'une précédente plate-forme développée au LURPA pour l'identification des systèmes logiques (De Smet *et al.* (1999) ; Denis *et al.* (2001)).

Comme indiqué précédemment, dans le cas d'un test non invasif, la connaissance de l'état courant d'une implantation ne peut être effectuée que par observation des sorties émises. L'utilisation de la méthode du tour de transition, utilisée pour la génération de la séquence de test repose sur l'hypothèse de l'existence d'une fonction *status* permettant de déterminer à tout instant l'état courant de l'implantation par observation des sorties. Dans notre cas, cette hypothèse est obtenue en imposant que l'état global de l'implantation (localité) soit discernable par observation de ses sorties.

HYPOTHÈSE 1 : L'état global de l'implantation (localité) est discernable par observation des sorties émises.

$$\forall (l_1, l_2) \in L^2, \quad O_{Emit}(l_1) \neq O_{Emit}(l_2) \tag{3.13}$$

En s'appuyant sur la définition de la machine de Mealy équivalent à l'ALS, cette hypothèse peut être définie comme suit :

$$\forall ((s,i),(s',i')) \in (S_M \times I_M)^2, \quad \delta_M(s,i) \neq \delta_M(s',i') \Rightarrow \lambda_M(s,i) \neq \lambda_M(s',i') \tag{3.14}$$

Nous rappelons que d'après les définitions des fonctions $\delta_M(s,i)$ et $\lambda_M(s,i)$, chaque couple $(s,i) \in S_M \times I_M$ vérifie :

$$\forall((s,i),(s',i')) \in (S_M \times I_M)^2, \quad \delta_M(s,i) = \delta_M(s',i') \Rightarrow \lambda_M(s,i) = \lambda_M(s',i') \tag{3.15}$$

Pour l'exemple A, cette hypothèse est vérifiée. Pour d'autres spécifications Grafcet, et donc a fortiori pour leurs implantations, comme pour l'exemple B, il est possible (et fortement probable) que cette hypothèse ne soit pas vérifiée. Dans ce cas, la spécification Grafcet doit être modifiée en ajoutant des sorties supplémentaires permettant de discerner l'ensemble des localités. Cette hypothèse pouvant être vérifiée dès la construction de l'ALS, le concepteur peut remédier au plus tôt à ce problème. La vérification de cette hypothèse s'inscrit dans une démarche de conception de type "design for test" (Joint Test Action Group (JTAG) (2010)). L'ajout de sorties supplémentaires est comparable à l'ajout de Points de Contrôle et d'Observation (PCO) pour les circuits intégrés (voir chapitre 1, section 3.3).

Pour pouvoir tester la conformité d'un contrôleur en fonctionnement par la seule observation du comportement de ses sorties à des sollicitations de ses entrées, le banc de test doit permettre de :

- Solliciter chacune des entrées logiques du contrôleur en fonction de la séquence de test pré-établie ;
- Observer chacune des sorties logiques ;
- Communiquer avec un ordinateur de bureau, afin de faciliter l'exécution de la séquence de test, l'élaboration du verdict, ainsi que l'enregistrement des résultats.

Le banc de test utilisé actuellement est construit autour d'un module d'entrées/sorties déportées. Ce module industriel est composé d'une embase d'entrées/sorties Tout Ou Rien (TOR) (Schneider Electric, TSX Momentum, 170 ADM 350 11) et d'un communicateur Ethernet (Schneider Electric, TSX Momentum, 170 ENT 110 01). Ce module dispose de 16 entrées et 16 sorties logiques contrôlables et observables à distance à partir d'un ordinateur de bureau via le protocole Modbus TCP/IP. Chaque sortie de ce module d'entrées/sorties déportées est connectée en amont d'une entrée du contrôleur logique à tester tandis que chacune de ses entrées est connectée en aval d'une sortie de ce contrôleur logique (cf. figure 3.3). Cette solution ouverte offre ainsi une grande flexibilité en ce qui

concerne les possibilités de sollicitation et de traitement des observations effectuées, et un moindre coût.

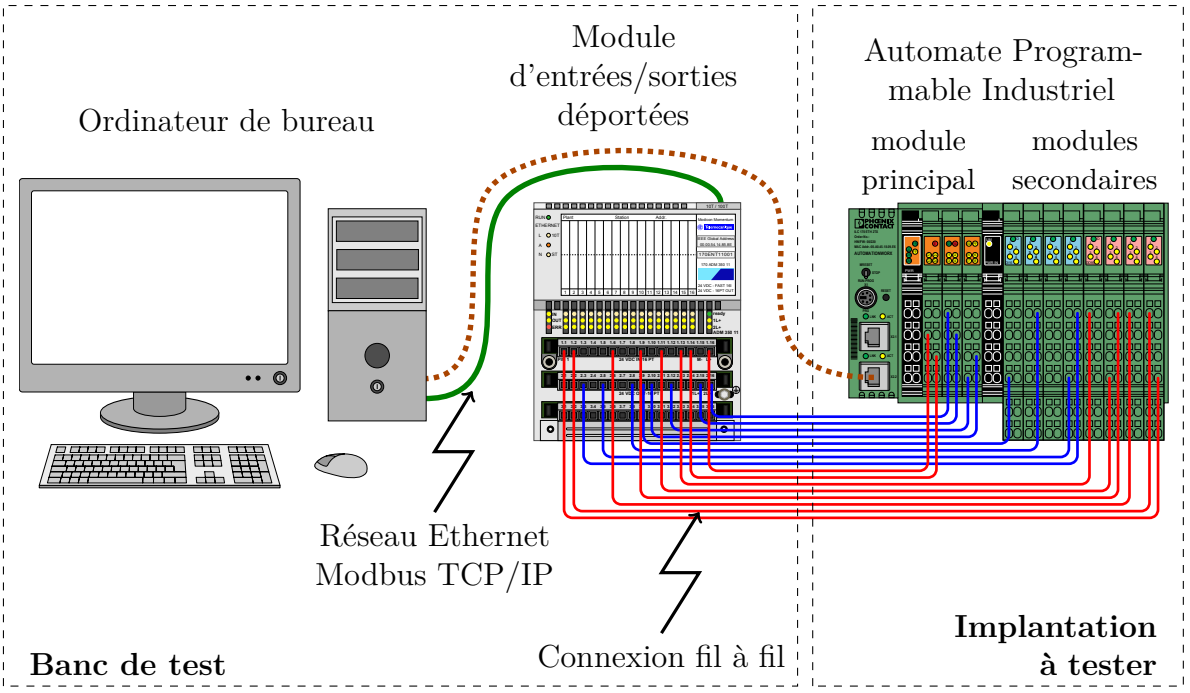


Figure 3.3 – Dispositif expérimental - $1^{ère}$ validation expérimentale

Concernant le contrôleur à tester, un API modulaire (Phœnix Contact ILC 170 ETH 2TX) a été retenu, en raison de sa réactivité (temps de cycle minimum : 1 ms) et de la conformité de son atelier de programmation (PC Worx) à la norme IEC 61131-3 (2003). Le mode d'exécution des tâches de cet API peut être cyclique ou périodique.

3.2 Implantation d'une spécification Grafcet dans un API

Avant de s'intéresser à la génération et à l'exécution de séquence de test à partir d'une spécification Grafcet, cette section a pour objectif de présenter quelques méthodes d'implantation de spécifications Grafcet dans un API, puis l'intérêt de la formalisation de cette spécification Grafcet par un ALS pour cette implantation du programme.

3.2.1 Méthodes d'implantation de spécifications Grafcet dans un API

Plusieurs méthodes existantes permettent l'implantation d'une spécification Grafcet dans un API. Certaines méthodes font appel à un atelier de programmation spécifique permettant de représenter de manière graphique la spécification Grafcet, puis utilise

un interpréteur pour générer automatiquement le code API exécutant la spécification Grafcet. C'est, par exemple, le cas de PL7 qui permet la programmation d'API Schneider Electric. L'avantage de ces méthodes est sans conteste l'interface graphique qui permet de retranscrire la spécification Grafcet sous forme graphique, et la possibilité d'animer cette représentation lorsque l'API est en fonctionnement. Cependant, cette interface présente des limitations à l'utilisation (taille des feuilles de dessin, et navigation entre les différentes parties de la spécification). De plus, la syntaxe et la sémantique utilisées ne respectent pas toujours le standard Grafcet, et font parfois un mélange entre Grafcet (IEC 60848) et SFC (IEC 16131) ; ceci implique que le code généré ne respecte pas toujours le comportement de la spécification Grafcet, tel que décrit par cette même norme (IEC 60848).

D'autres méthodes ont également été développées afin de permettre la génération de code pour API à partir d'une spécification Grafcet, sans avoir recours à un atelier de programmation propriétaire. Les langages les plus utilisés dans ce cas, pour la programmation du code pour l'API, sont les langages ST et LADDER de la norme IEC 61131-3. En effet, ces deux langages permettent une exécution séquentielle du code (de la première ligne à la dernière ligne), sans prendre en compte des priorités d'exécution en fonction du placement de blocs sur une représentation graphique, comme c'est le cas pour le langage FBD. Ces méthodes présentées dans Roussel *et al.* (2004) et Machado *et al.* (2006), présentent l'avantage d'être simples à appliquer : en général la génération peut être automatisée, par exemple depuis les équations récurrentes de la spécification Grafcet. Mais cette simplicité d'application présente un inconvénient : la recherche de stabilité n'est pas prise en compte, donc des sorties affectées à une étape du Grafcet instable lors d'une évolution fugace peuvent être émises.

3.2.2 Intérêt de l'ALS pour l'implantation de spécifications Grafcet

La formalisation de la spécification par un ALS permet de générer une machine à état sans évolution fugace. Une solution pour implanter la spécification Grafcet est donc d'appliquer la méthode proposée par Machado *et al.* (2006), non plus à la spécification Grafcet, mais à l'ALS de cette spécification Grafcet ; la recherche de stabilité n'a pas besoin d'être implantée dans le code pour l'API car celle-ci a déjà été effectuée lors de la formalisation de l'ALS. Cependant, le nombres de localités et d'évolutions étant supérieur au nombre d'étapes et de transitions, cette méthode augmente considérablement la taille

mémoire nécessaire ainsi que le nombre de calculs à effectuer par l'API (un calcul par transition ou évolution, ainsi qu'un calcul par étape ou localité, et un calcul par sortie).

En revanche, la formalisation de la spécification par un ALS permet de générer une machine à état dont l'état actif est, par construction, unique à tout instant. En prenant en compte cette propriété, il apparait que la connaissance de l'état courant permet de réduire le nombre de calculs à effectuer par l'API. En utilisant le langage ST de la norme IEC 61131-3, une structure conditionnelle *CASE* permet de prendre en compte cette propriété ; ainsi seules les expressions booléennes associées à la localité active doivent être évaluées afin de connaitre la localité atteinte ainsi que les sorties émises. Cette méthode permet donc de calculer, en un seul cycle API, la localité stable atteinte et les sorties stables qui doivent être émises, sans accroitre de manière trop importante le nombre de calculs à effectuer à chaque cycle API (un calcul pour la condition *CASE*, ainsi qu'un calcul par évolution partant de la localité active, et un calcul pour déterminer les sorties modifiées suite au changement de localité active).

Malheureusement, les logiciels et outils de compilation de code utilisés pour la génération de code embarqué dans un API ou autre contrôleur n'intègrent pas cette phase de formalisation. Le comportement du code généré, exécuté par une implantation, ne peut donc par être garanti, par construction, conforme au comportement décrit par la spécification Grafcet. De plus, peu de compilateurs de code étant certifiés, le recours à une phase de test de conformité est nécessaire à la certification de la conformité d'une implantation.

4 Réalisation d'un test de conformité

Il a été choisi de réaliser le test de conformité du contrôleur dans les conditions les plus proches de son fonctionnement réel. Le programme implanté dans l'API, au format texte structuré (ST), est obtenu automatiquement depuis la spécification Grafcet en utilisant la méthode proposée par Machado *et al.* (2006). Cette solution permet, soit de générer un code *a priori* conforme à la spécification, soit d'introduire artificiellement des erreurs de programmation. Une fois programmé, le contrôleur logique à tester est déconnecté de son atelier de programmation pendant l'exécution de l'intégralité du test.

Afin de vérifier l'hypothèse 1 (relation 3.13, page 99), une analyse statique de ce code permet de vérifier l'association variables d'état/variables de sortie. Cette phase de

vérification est rapide à mettre en œuvre car la partie du code à analyser ne concerne que quelques lignes (1 ligne par sortie).

4.1 Exécution d'une séquence de test

Le déroulement d'un test de conformité est le suivant :

- Connexion physique du contrôleur à tester au banc de test ;
- Initialisation du contrôleur à tester ;
- Exécution de la séquence de test et élaboration du verdict.

Comme mentionné dans le chapitre précédent, lors de l'initialisation d'un API, les cartes d'entrée sont déjà sollicitées par des signaux logiques ; les valeurs de ces entrées logiques sont donc utilisées dès le premier cycle de calcul de l'API. De ce fait, l'initialisation ou la réinitialisation d'un API, devra être conforme à la définition de l'initialisation donnée par la formalisation de la spécification Grafcet définie par l'ALS. Cette (ré)initialisation doit donc conduire l'implantation dans son état initial, et (ré)initialiser les variables internes du programme API, notamment les sorties affectées, à leur valeur initiale. La norme Grafcet précise qu'à l'initialisation, la valeur des sorties affectées est nulle. Pour effectuer correctement cette (ré)initialisation des variables, l'API doit soit être arrêté, réinitialisé puis redémarré (à chaud ou à froid), soit redémarré à froid (voir *section 2.4.2 Initialization*, dans la norme IEC 61131-3)

Une fois le contrôleur initialisé, l'exécution du test de conformité consiste, pour chaque pas de la séquence de test, à :

- Émettre les signaux d'entrée logiques correspondant à la sollicitation mentionnée par ce pas de test ;
- Attendre le temps nécessaire à la stabilisation des sorties émises ;
- Observer les signaux de sortie logiques émis par le contrôleur à tester ;
- Comparer les valeurs des signaux de sortie observées à celles attendues, et rendre un verdict ;
- Enregistrer les résultats.

Le temps d'attente nécessaire à la stabilisation des sorties émises doit être choisi de telle sorte qu'il prenne en compte les deux paramètres suivants (cf. figure 3.4) :

- Retard de causalité minimal ;
- Nombre de pas de calcul maximum nécessaire à l'obtention de variables de sortie

stables.

Le retard de causalité minimal dépend du mode de fonctionnement de l'API ; il est égal, dans le pire des cas, à un temps de cycle API. Le nombre de pas de calcul maximum dépend de la méthode utilisée pour programmer l'implantation. Dans le cas de la méthode proposée par Machado *et al.* (2006), plusieurs cycles API peuvent être nécessaires pour atteindre la stabilité. Le nombre maximal de cycles API nécessaires pour atteindre la stabilité est égal à la longueur de la plus longue séquence d'ensembles de transitions simultanément franchissables ; cette valeur peut donc être déterminée lors de la construction de l'ALS de la spécification Grafcet. Cette valeur reste une valeur théorique, elle devra donc être majorée afin de prendre en compte de potentielles erreurs présentes dans l'implantation. Nous supposerons de plus que l'implantation ne comporte pas de cycle d'évolutions fugaces de longueur supérieure au nombre de cycles API effectués pendant l'attente de la stabilisation.

HYPOTHÈSE 2 : L'implantation ne comporte pas de cycle d'évolutions fugaces de longueur supérieure au nombre de cycles API effectués pendant l'attente de la stabilisation des sorties.

Il est important de souligner que **le banc de test n'est pas synchronisé avec l'API**, ni de manière directe, via l'atelier de programmation, ni de manière indirecte, par synchronisation des cycles de sollicitation du banc de test et d'exécution de l'API. Cela assure que les conditions dans lesquelles le test de conformité est effectué soient les plus proches possibles des conditions d'opération d'un système à événements discrets réel en boucle fermée, l'API pouvant avoir un mode de fonctionnement cyclique ou bien périodique et le comportement d'une partie opérative n'étant en pratique pas synchronisé avec celui du contrôleur.

4.2 Illustration sur un exemple

Les premiers essais ont été réalisés avec un API en fonctionnement périodique réglé à 20 ms (le code est exécuté toutes les 20 ms). La lecture des sorties de l'API est faite après un temps d'attente de 250 ms, soit 10 fois le temps de cycle API plus 50 ms.

Pour le Grafcet proposé sur la figure 2.6, la séquence comporte 1 966 pas, conduisant à la réalisation du test de conformité couvrant tous les couples (états,valuations d'entrée)

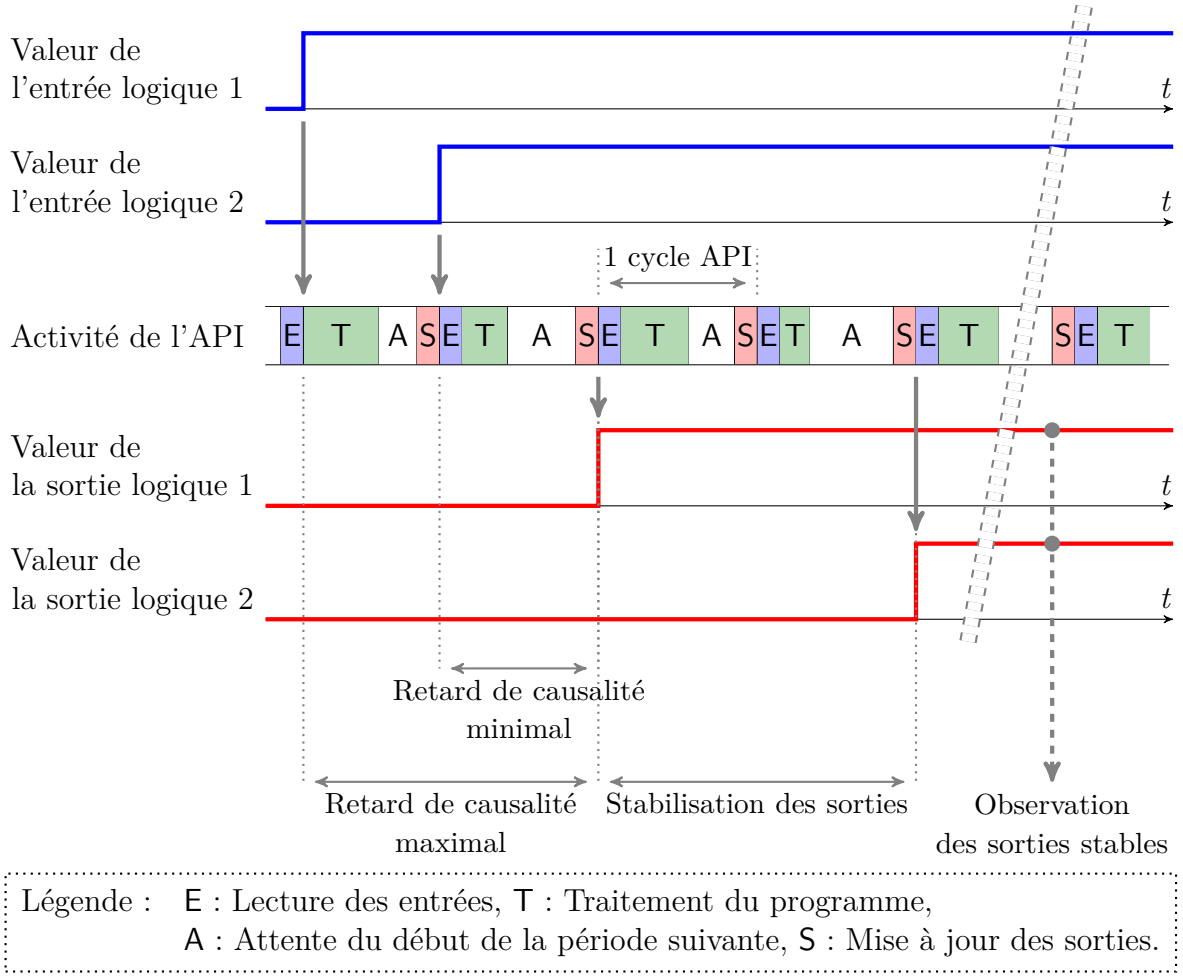


Figure 3.4 – Chronogramme illustrant le temps d'attente avant observation des sorties (cas d'un fonctionnement périodique)

de l'implantation en moins de 9 minutes.

Pour le Grafcet proposé sur la figure 2.7, la séquence comporte 73 528 pas, la réalisation du test de conformité est effectuée en environ 5 h. Dans le cas d'un API en mode d'exécution cyclique (cycle de 4 ms), ce temps de réalisation peut être réduit à moins de deux heures, à raison d'un pas de test toutes les 90 ms.

5 Étude des verdicts de la méthode de test

Lors de la réalisation du test de conformité de l'API en fonctionnement cyclique (mode de fonctionnement le plus couramment utilisé car maximisant la réactivité de la commande), des détections erronées de non-conformité conduisant au rejet par le banc de test d'un programme conforme ont été relevées. Une campagne d'essais a donc été réalisée

afin de quantifier ce taux d'erreur. Les expérimentations menées ont pour objectif l'étude des verdicts obtenus en utilisant une séquence de test générée avec l'approche proposée dans la section 2. Ces expérimentations s'intéressent plus particulièrement à l'influence des caractéristiques techniques du contrôleur à tester (mode de fonctionnement : cyclique ou périodique, et répartition des entrées) sur les verdicts obtenus lors de la réalisation du test de conformité de l'API. L'objectif de ces expérimentations est de qualifier et de quantifier la capacité d'une technique de test à fournir des verdicts non-biaisés et valides en fonction de la configuration expérimentale retenue.

Une analyse détaillée des fichiers de suivis des verdicts a mis en avant que ces problèmes sont toujours intervenus pour des pas de test où plusieurs valeurs des entrées logiques étaient changées simultanément par le banc de test, par exemple, pour le passage du pas de test 2 au pas 3 du tableau 3.1 (page 98). Les pas de test où une seule valeur des entrées logiques était changée n'ont jamais causé de tels problèmes. Une explication a alors été avancée : les changements d'entrées synchrones générées par le banc de test peuvent être perçus comme asynchrones par le contrôleur logique car, du fait du fonctionnement cyclique (ou périodique) de l'implantation, toutes les entrées peuvent ne pas être lues simultanément. Les valeurs d'entrée utilisées par l'implantation pour calculer ses sorties sont alors différentes de celles prévues. Ces erreurs d'interprétation peuvent alors conduire au rejet d'une implantation correcte (verdict biaisé), mais également à l'acceptation d'une implantation comportant des erreurs (verdict non valide). Des comportements similaires ayant déjà été observés lors d'une étude préliminaire (Roussel et Faure (2003)), ces observations confirment les résultats de la précédente étude et méritent d'être quantifiés plus précisément.

La figure 3.5 illustre une portion d'une séquence de test émise par le banc de test ainsi que les perceptions possibles de cette séquence de test par le contrôleur logique à tester. Dans le cas de cette séquence, faisant varier simultanément deux variables d'entrées logiques a et b, le contrôleur logique peut percevoir un changement d'entrée synchrone de ces deux entrées logiques selon les trois manières suivantes :

- Changements simultanés de a et b ;
- Changement de a suivi du changement de b ;
- Changement de b suivi du changement de a.

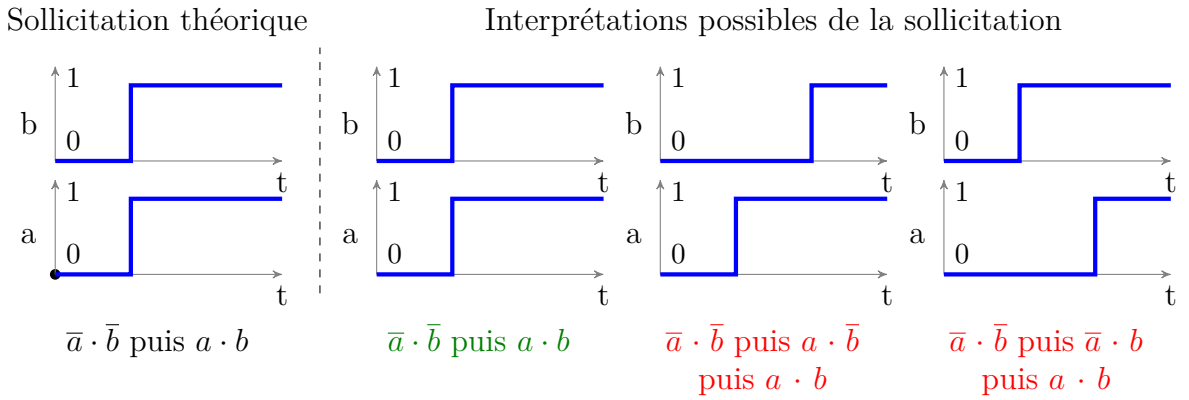


Figure 3.5 – Interprétations possibles d'une variation simultanée de 2 entrées logiques

5.1 1ère validation expérimentale de la perception asynchrone de signaux synchrones

L'étude théorique qui a été conduite repose sur l'hypothèse implicite que toute sollicitation émise par le banc de test est perçue par le contrôleur à tester sans erreur d'interprétation. Dans la pratique, il s'avère que cette hypothèse peut ne pas être vérifiée lorsque deux sollicitations successives nécessitent de faire varier simultanément plusieurs entrées logiques.

Pour analyser la sensibilité du contrôleur vis-à-vis de cet aléa d'interprétation, l'expérimentation suivante a été conduite :

- Le banc de test sollicite les entrées de l'automate programmable industriel avec 8 signaux logiques variant simultanément de 0 à 1, puis de 1 à 0. La période de la sollicitation $T_{Sollicitation}$ est fixée à une valeur grande devant le temps de cycle API T_{API} : $T_{Sollicitation} \simeq 2 \times (10 \times T_{API} + 50\ ms)$. Ainsi, les variations de ces 8 signaux (fronts montants et descendants des signaux logiques) sont synchrones, et l'API peut effectuer plusieurs cycles de calcul entre deux sollicitations.
- Le programme implanté dans l'API fonctionne de la façon suivante : lorsque l'API détecte le premier changement de l'une de ses 8 entrées logiques (front montant ou front descendant), la valeur de chacune de ces 8 entrées est recopiée sur la sortie correspondante. Ainsi, la valeur du vecteur de sortie représente la valeur du vecteur d'entrée lors de la détection du premier front (montant ou descendant). La recopie des entrées sur leur sortie associée se faisant lors de la détection du premier front, les valeurs des sorties logiques restent donc invariablement stables jusqu'au

prochain front de type inverse détecté.

– Après un temps d'attente égal à $\frac{T_{Sollicitation}}{2}$, le banc de test lit les valeurs des sorties émises. Les valeurs de ces sorties doivent être identiques aux valeurs des entrées envoyées si aucune erreur de lecture des entrées (changements synchrones perçus comme asynchrones) n'est apparue. Un compteur est incrémenté dès lors qu'une erreur est détectée, i.e. quand le vecteur de sortie diffère de celui d'entrée.

– À la fin de la période $\frac{T_{Sollicitation}}{2}$, avant d'effectuer une nouvelle sollicitation, le banc de test attend un temps aléatoire, compris entre 0 et 3 ms, afin d'éviter tout phénomène de synchronisation entre le banc de test et l'API.

La figure 3.6 illustre le fonctionnement de ce protocole expérimental.

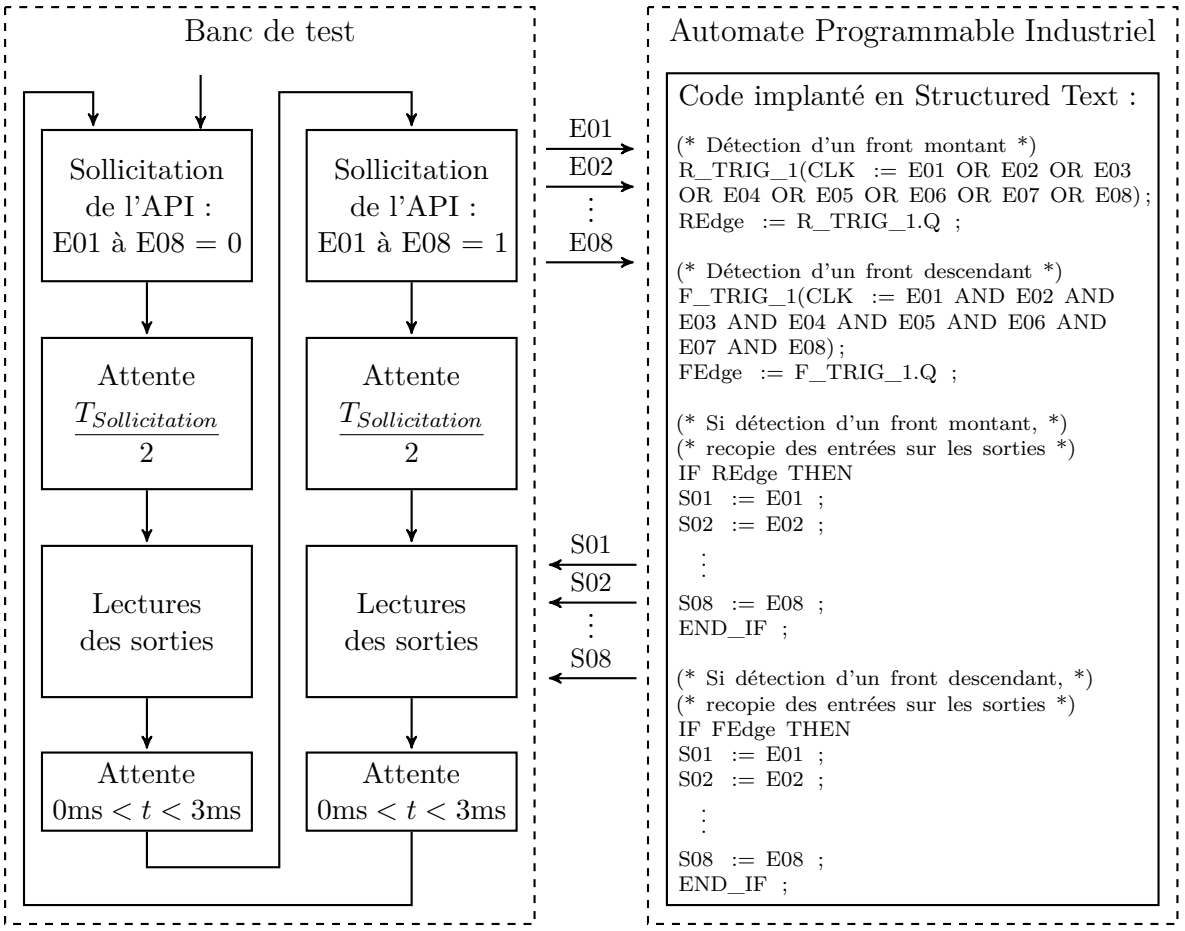


Figure 3.6 – Descriptif du protocole expérimental

La période de sollicitation de $T_{Sollicitation}$ est volontairement choisie grande, afin de prendre en compte les différentes durées d'immunité et de filtrage des composants d'entrée et de sortie, et de permettre l'observation de signaux de sortie stables. Pour information, le temps de réponse du module d'entrées/sorties déportées utilisé pour la lecture des sorties émises par l'API est de 0,06 ms pour une variation du signal de 0 à 1, et de

0,08 ms pour une variation du signal de 1 à 0.

Le nombre de signaux choisis pour cette expérience a été fixé à 8 pour deux raisons :

- Le module de base de l'API ne dispose que de 8 entrées logiques, le module d'entrées logiques associées en comporte 16. En choisissant 8 entrées, les résultats obtenus peuvent être facilement comparés.
- Le module d'entrées/sorties déportées pilote ses sorties (signaux sollicitant les entrées de l'API) en deux groupes de 8 sorties. La variation de ces 8 sorties est donc effectuée simultanément tandis qu'il peut exister un délai entre les deux groupes de sorties. Les sorties utilisées pour les expérimentations sont donc toutes reliées à un même groupe.

Le tableau 3.2 regroupe les taux d'erreur d'interprétation des signaux sollicitant l'API, en fonction de son mode de fonctionnement (exécution cyclique ou périodique à une période fixée) et de la répartition des entrées sur les modules de l'API (toutes les entrées connectées au module de base, connectées au module d'entrées, ou bien réparties à la fois sur le module de base et sur le module d'entrées). Les résultats obtenus permettent de quantifier le taux d'erreurs commises dans les conditions expérimentales décrites figure 3.3. Ces résultats ont été obtenus à partir de séries d'essais de 200 000 expérimentations pour chacun des couples (répartition,mode). La réalisation de cette campagne de 9 séries d'essais de 200 000 expérimentations a été effectuée en une semaine.

Répartition \ Mode		Cyclique	Périodique 10 ms	Périodique 20 ms
sur module de base	Passage de 0 à 1	6,20 ‰	2,15 ‰	1,35 ‰
	Passage de 1 à 0	191,80 ‰	88,60 ‰	45,45 ‰
sur module d'entrées	Passage de 0 à 1	10,40 ‰	4,00 ‰	2,00 ‰
	Passage de 1 à 0	253,60 ‰	99,30 ‰	55,00 ‰
entre les deux modules	Passage de 0 à 1	29,17 %	39,62 %	20,23 %
	Passage de 1 à 0	38,55 %	43,46 %	21,89 %

Tableau 3.2 – Taux d'erreur d'interprétation par l'API - 1ère validation expérimentale

Les résultats obtenus valident l'explication proposée. Étant donné que le cycle de scrutation des entrées de l'API n'est pas synchronisé avec le cycle d'exécution du banc de test, *les événements synchrones envoyés par le banc de test peuvent être perçus de manière*

asynchrone par l'API. Dans le cas où les entrées sont distribuées sur plusieurs modules (les deux dernières lignes du tableau 3.2), le taux d'erreur augmente fortement à cause de la communication interne entre les modules qui induit des retards supplémentaires. Cette communication entre les différents modules, conçue pour diminuer les temps d'attente du processeur de l'API, n'est pas prévue pour garantir la simultanéité de lecture des entrées réparties entre plusieurs modules.

L'analyse des valeurs de ce tableau montre que plus le temps de cycle API est élevé, plus le taux d'erreur est faible. En effet, lorsque le temps de cycle API augmente, la probabilité que le changement des entrées par le banc puisse s'effectuer lors de la phase de scrutation des entrées par l'API est plus faible. Ce tableau montre également que le comportement n'est pas le même lors de la mise à 1 et de la mise à 0, plus d'erreurs de perception ayant lieu lors de la lecture à 0. Une des raisons à cette différence peut être la différence entre les temps de réponse de ce module pour une variation de 0 à 1 et de 1 à 0 : les valeurs annoncées pour ces temps de réponse sont respectivement de 0,06 ms et 0,08 ms, mais la différence réelle entre ces deux temps de réponse peut être plus grande. L'influence de ces temps de réponse sera présentée dans la section 5.3.

5.2 2ème validation expérimentale de la perception asynchrone de signaux synchrones

Une réponse rapide à ce problème eut été de considérer que l'ensemble de ces erreurs provenait uniquement de l'émission des entrées par le module déporté, et aucunement de la lecture de ces signaux par l'API. En effet, la notice technique du module d'entrées/sorties déportées (Schneider Electric, TSX Momentum, 170 ADM 350 11) indique un temps de réponse de ses sorties inférieur à 0,1 ms, aussi bien pour la mise à 1 qu'à 0. Ce temps de réponse, bien que très inférieur au temps de cycle API, reste non nul, comme pour tout composant physique réel. Afin d'obtenir plus de précisions quant à l'influence de ce paramètre, une seconde campagne d'essais à été réalisée, en conservant le même programme API, mais en modifiant la manière dont les 8 signaux logiques sollicitant les entrées logiques de l'API sont émis. Lors de la précédente expérimentation, chaque signal sollicitant l'API est émis par une sortie distincte du module déporté ; pour cette nouvelle expérimentation, un seul signal de sortie du module déporté est utilisé, ce signal est ensuite *dupliqué électriquement* en 8 signaux logiques identiques reliés aux 8 entrées

de l'API (voir le nouveau dispositif expérimental présenté figure 3.7). Le tableau 3.3 donne les taux d'erreur obtenus pour cette nouvelle expérimentation, ces résultats ayant été obtenus à partir de séries d'essais de 500 000 expérimentations pour chacun des couples (répartition,mode). La réalisation de cette campagne de 9 séries d'essais de 500 000 expérimentations a été effectuée en un peu plus de deux semaines.

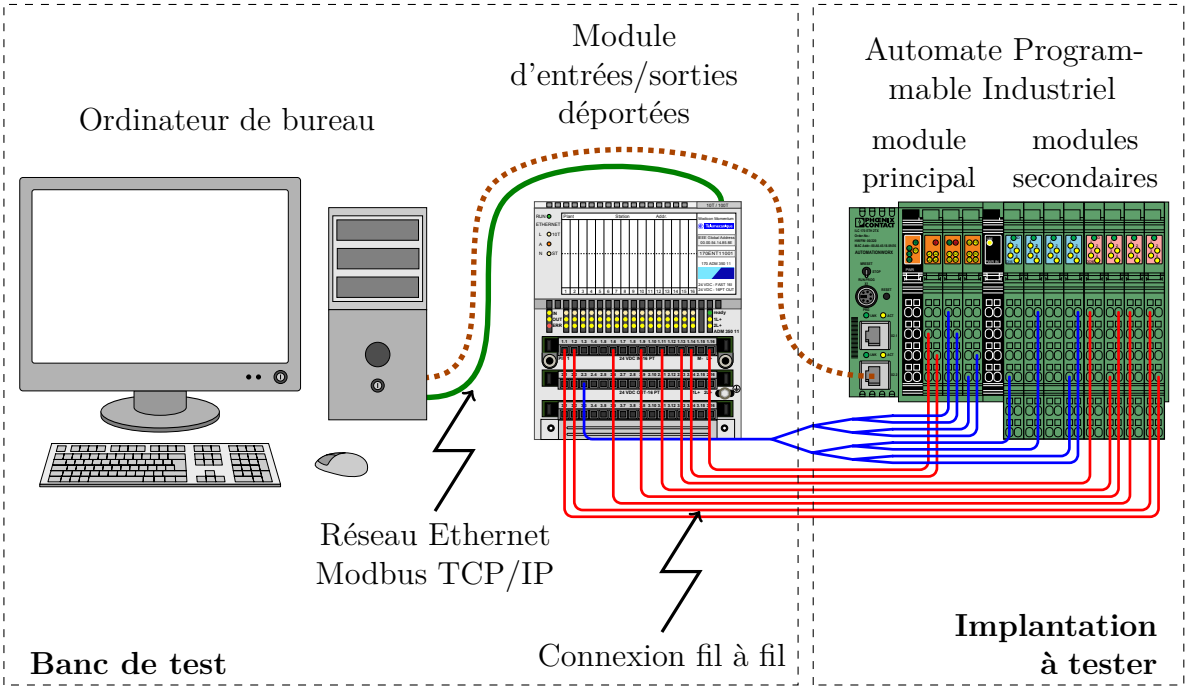


Figure 3.7 – Dispositif expérimental - $2^{ème}$ validation expérimentale

Répartition \ Mode		Cyclique	Périodique 10 ms	Périodique 20 ms
sur module de base	Passage de 0 à 1	5,94 ‰	2,64 ‰	1,22 ‰
	Passage de 1 à 0	23,66 ‰	10,46 ‰	4,84 ‰
sur module d'entrées	Passage de 0 à 1	8,62 ‰	3,60 ‰	1,78 ‰
	Passage de 1 à 0	42,92 ‰	7,98 ‰	9,46 ‰
entre les deux modules	Passage de 0 à 1	30,22 %	39,34 %	19,86 %
	Passage de 1 à 0	36,63 %	42,21 %	21,20 %

Tableau 3.3 – Taux d'erreur d'interprétation par l'API - $2^{ème}$ validation expérimentale

Les résultats obtenus montrent que le taux d'erreur diminue fortement dans le cas d'un passage de 1 à 0 (dans un facteur de 5,7 et 8,7 en moyenne, pour le module principal

et secondaire, respectivement), mais plus légèrement dans le cas d'un passage de 0 à 1 (dans un facteur de 0,9 et 1,1 en moyenne, pour le module principal et secondaire, respectivement) ; les taux d'erreur observés confirment la difficulté à percevoir de manière synchrone des signaux synchrones.

Les taux d'erreur observés restent faibles, mais non nuls. Ceci explique pourquoi ce phénomène a été peu observé auparavant par d'autres personnes, car difficilement observable sur des petites séries, mais qu'il reste primordial de le considérer dans le cas d'applications critiques.

5.3 Origine de la perception asynchrone de signaux synchrones

Dans le cas de signaux logiques réels, la variation de 0 à 1 (respectivement de 1 à 0) de ces signaux ne peut être faite à temps nul. De plus, les seuils de détection à 0 et à 1 de ces signaux, ainsi que les durées de filtrage peuvent être différents entre deux voies d'une même carte d'entrées car les composants électroniques (résistance, condensateur, transistor, ...) qui constituent ces cartes ne sont pas parfaits ; des variations sont donc possibles autour d'une valeur moyenne. Cet écart à la moyenne peut être réduit mais ne pourra jamais être garanti comme nul. Il est donc, dans l'absolu, impossible de toujours percevoir ces signaux logiques de manière synchrone.

La figure 3.8 illustre les différences entre deux signaux d'entrée variant théoriquement simultanément et les interprétations de ces signaux par l'API, après seuillage et filtrage. Sur cette figure, les signaux d'entrée théoriques sont supposés strictement identiques (cas le plus proche de la $2^{ème}$ série d'expérimentations). Les variations des caractéristiques des composants électroniques des cartes d'entrées impliquent que deux signaux électriques identiques avant filtrage peuvent être différents après seuillage et filtrage ; l'écart temporel entre ces deux signaux, noté εt, est en général petit devant le temps de cycle API. Pour le banc de test qui a été développé, le temps de réponse d'un module d'entrées/sorties déportées est de est de 0,06 ms pour une variation du signal de 0 à 1, et de 0,08 ms pour une variation du signal de 1 à 0 ; l'écart temporel entre deux signaux logiques émis par ce module est donc borné par ces valeurs. Cependant, plus que la valeur de cet écart, c'est l'instant où apparait cet écart εt qui implique les différences de perceptions possibles. Sur cette figure, dans le premier cas (à gauche), les deux signaux filtrés présentent un écart εt, mais le changement intervient pendant la phase d'attente de l'API : les deux signaux

seront donc lus lors du prochain cycle de scrutation, et seront perçus comme synchrones. Dans le second cas (à droite), le changement intervient juste avant un cycle de scrutation. Le premier signal filtré est donc lu lors du cycle de scrutation des entrées tandis que le second sera lu au cycle suivant : les deux signaux sont perçus comme asynchrones, avec un écart temporel, noté Δt, égal à un temps de cycle API.

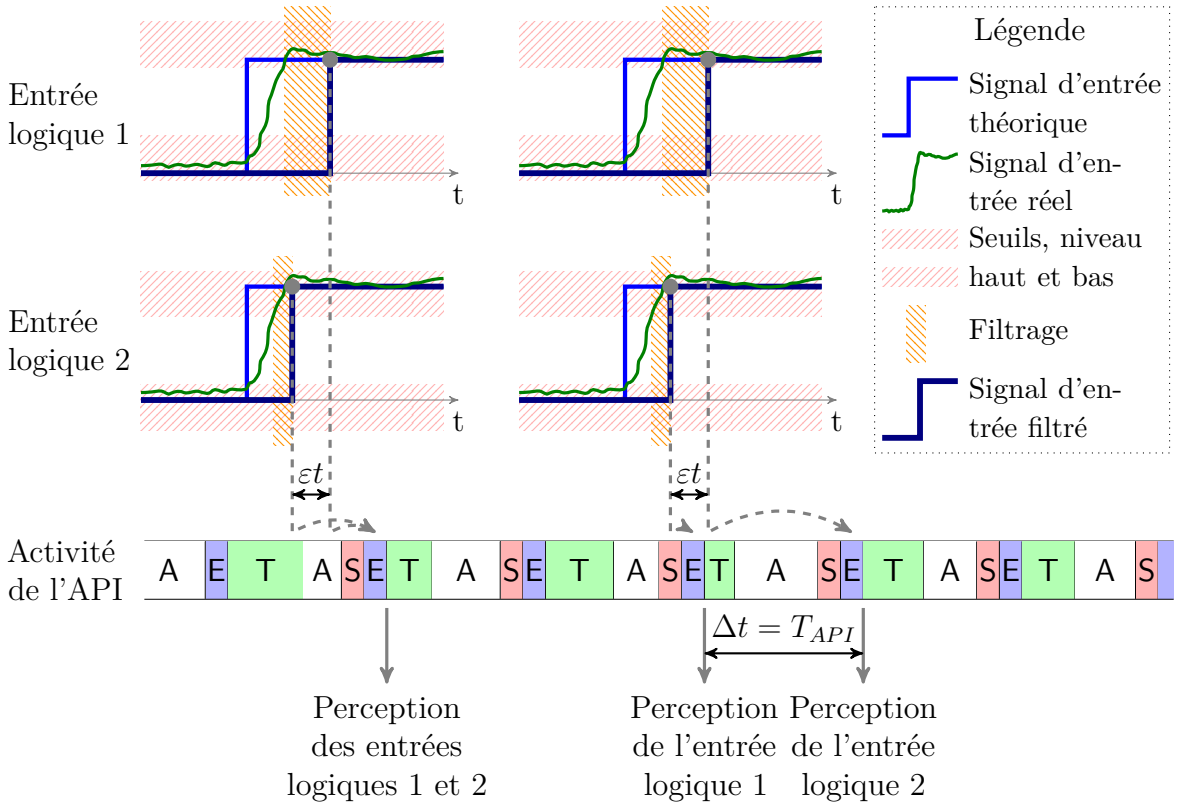


Figure 3.8 – Illustration de l'écart entre signaux théoriques et réels perçus par l'API

La figure 3.9 donne deux exemples élémentaires de machines de Mealy. Dans le premier cas (machine de gauche), une erreur de perception du changement simultané des deux entrées logiques a et b conduit au rejet d'une implantation correcte, le verdict est biaisé. Dans le second cas (machine de droite), une erreur de perception de ce changement simultané conduit à l'acceptation d'une implantation qui peut être incorrecte, le verdict est non valide.

En effet, pour la machine de gauche, lorsque le testeur souhaite tester la transition étiquetée $\overline{a} \cdot \overline{b}/o$ depuis l'état s_2, il est nécessaire de changer simultanément les valeurs des deux entrées logiques a et b car s_2 est stable uniquement pour les valuations $a \cdot b$ et $\overline{a} \cdot \overline{b}$. Une erreur de perception de ce changement synchrone peut conduire au franchissement de la transition étiquetée $\overline{a} \cdot b/\overline{o}$ depuis l'état s_2 puis de la transition étiquetée $\overline{a} \cdot \overline{b}/\overline{o}$

bouclant sur l'état s_1, ou bien au franchissement de la transition étiquetée $\overline{a} \cdot \overline{b}/\overline{o}$ depuis l'état s_2 puis de la transition étiquetée $\overline{a} \cdot b/\overline{o}$ bouclant sur l'état s_1. L'état atteint ainsi que la sortie logique émise sont différents de ceux correspondant au franchissement de la transition étiquetée $\overline{a} \cdot \overline{b}/o$ depuis l'état s_2, le verdict rejette donc l'implantation qui est correcte ; ce verdict est donc biaisé.

Pour la machine de droite, lorsque le testeur souhaite tester la transition étiquetée $\overline{a} \cdot \overline{b}/\overline{o}$ depuis l'état s_2, il est nécessaire de changer simultanément les valeurs des deux entrées logiques a et b car s_2 est stable uniquement pour la valuation $a \cdot b$. Une erreur de perception de ce changement synchrone peut conduire au franchissement la transition étiquetée $\overline{a} \cdot b/\overline{o}$ depuis l'état s_2 puis de la transition étiquetée $\overline{a} \cdot \overline{b}/\overline{o}$ bouclant sur l'état s_1, ou bien au franchissement de la transition étiquetée $\overline{a} \cdot \overline{b}/\overline{o}$ depuis l'état s_2 puis de la transition étiquetée $\overline{a} \cdot b/\overline{o}$ bouclant sur l'état s_1. L'état atteint et la sortie logique émise sont les mêmes que pour le franchissement de la transition étiquetée $\overline{a} \cdot \overline{b}/\overline{o}$ depuis l'état s_2, mais la transition testée n'est pas nécessairement celle prévue ; le verdict est donc non valide.

6 Contraintes induites sur la séquence de test

Les expériences qui ont été conduites montrent que l'utilisation d'une séquence de test générée suivant l'objectif de longueur minimale peut conduire à des résultats erronés lors de l'exécution de cette séquence, et par conséquent à l'acceptation d'une implantation non conforme ou au rejet d'une implantation conforme à sa spécification. En effet, cette approche repose sur l'hypothèse implicite que toutes les variations des signaux logiques d'entrée générés par le banc de test sont perçues correctement par l'API, même si plusieurs de ces signaux logiques varient simultanément ; ce qui n'est pas toujours vérifié, comme l'illustre les tableaux 3.2 et 3.3.

Pour éviter cet aléa d'interprétation, plusieurs stratégies sont envisageables :

- Synchronisation du banc de test avec le contrôleur à tester.
- Exécution des séquences de test de conformité pour une configuration de l'API à tester telle que le taux d'erreur de perception soit le plus faible possible (cycle d'exécution lent, et entrées reliées à une seule carte).
- Exécution à plusieurs reprises du même test de conformité, et analyse statistique des résultats.

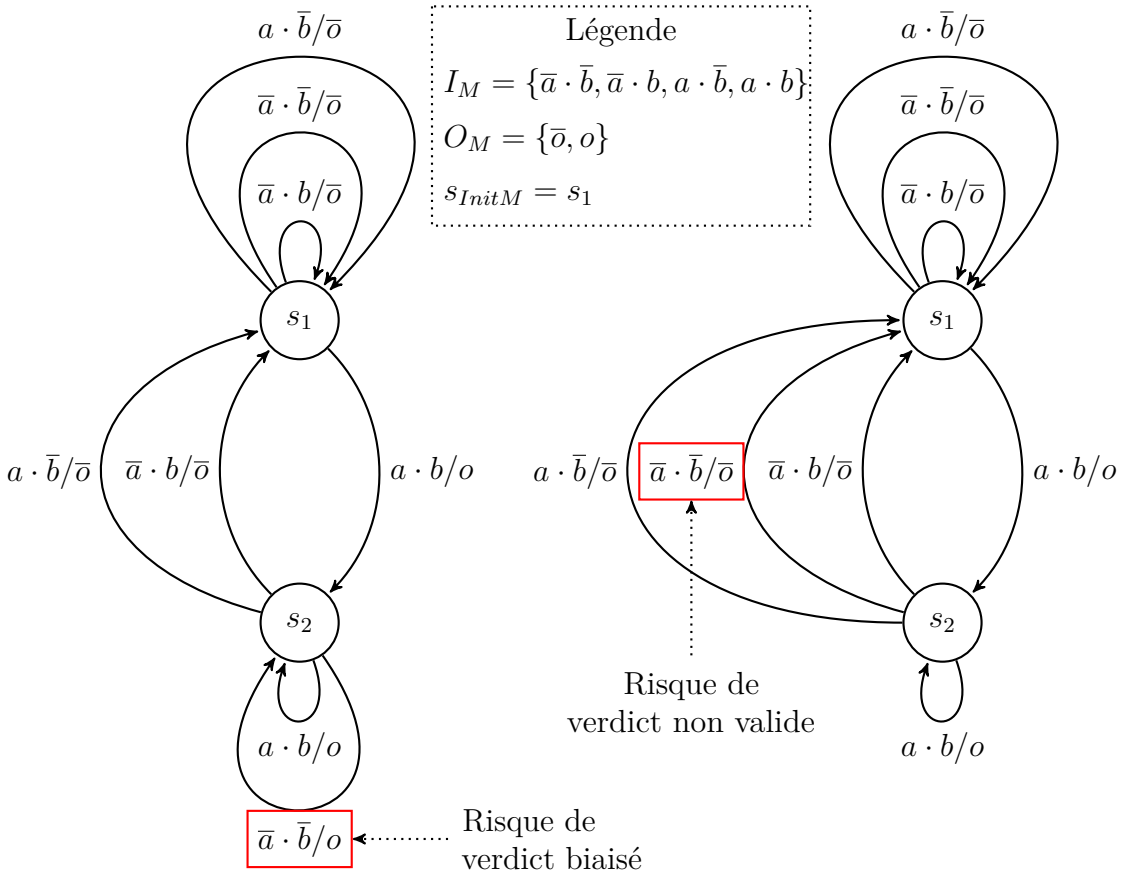


Figure 3.9 – Machines de Mealy pouvant conduire à des verdicts biaisé (à gauche) ou non valide (à droite)

– Intégration de ce risque d'aléa lors de la constitution d'une séquence de test.

La première solution est rejetée car elle nécessite qu'une communication additionnelle entre le banc de test et l'API à tester soit mise en place, et ne représente pas les conditions d'opération d'un système à évènements discrets réel en boucle fermée. Les seconde et troisième solutions doivent être évitées dans le cas du test de contrôleurs destinés à piloter des systèmes critiques car ils impliquent des analyses statistiques des résultats.

En raison de la forte criticité des systèmes pour lesquels cette méthode doit être mise au point, c'est la dernière stratégie qui a été retenue dans nos travaux. Il est donc nécessaire de s'assurer que la séquence de sollicitations utilisée pour le test de conformité ne puisse pas être à l'origine d'erreur de perception. Une condition suffisante pour garantir ceci est d'imposer qu'une seule entrée logique varie entre deux pas consécutifs de la séquence de test. La construction d'une séquence respectant cette condition fait l'objet du chapitre suivant.

7 Discussion sur l'utilisation d'APIdS

L'étude expérimentale menée pour étudier les verdicts obtenus pour différentes configurations de l'implantation a été réalisée sur un API standard et non sur un APIdS. Cette section a pour objectif de présenter brièvement la composition des APIdS et d'esquisser l'application de l'étude présentée dans ce chapitre aux APIdS.

Les API standards effectuant peu de test internes, il est très difficile d'assurer un niveau de sécurité élevé en utilisant uniquement des API standards. Les APIdS ont été conçus pour traiter des fonctions de sécurité de personnes en intégrant une architecture redondante ainsi que des procédures de test automatiques (Built-In Self Test (BIST) des composants électroniques, vote k parmi n sur les variables des entrées observées, variables internes et sorties émises, ...). Ainsi, en cas de défaillance d'un des éléments d'un APIdS, celui-ci passe dans une position de repli sûre.

D'après les appareils dédiés à la sécurité soumis par plusieurs fabricants à des organismes de contrôle, deux architectures types ont été identifiées par l'INRS. Ces deux architectures, A et B, sont illustrées en annexe (figures B.1 et B.2).

L'étude expérimentale ayant montré l'influence de répartition des entrées logiques, la présentation qui va suivre s'intéresse plus particulièrement à la connexion des entrées logiques sur ces deux architectures.

L'architecture A comporte un seul module d'entrées ; les entrées sont ensuite découplées dans des canaux différents reliés chacun à un microprocesseur distinct. La comparaison est faite à l'issue des calculs avant l'émission des sorties réelles. L'architecture B comporte deux sous-unités identiques, reliées entre elles par fibre optique, fonctionnant en parallèle. Chaque unité possède son propre module d'entrées, la comparaison entre les deux sous-unités est faite lors d'étapes intermédiaires de calcul, ou avant l'émission des sorties.

L'architecture A possédant un seul module d'entrées, le problème d'introduction d'asynchronisme lors de la lecture des entrées reste le même. La réduction du taux d'erreur ne peut être obtenue que par l'utilisation de modules d'entrées ayant un temps de réponse plus court, mais ce taux d'erreur ne sera cependant pas nul. L'architecture B possédant deux modules d'entrées séparés, le comportement global obtenu sera donc différent. En effet, chacune des sous-unités possédant son propre module d'entrées, chacune peut percevoir, ou non, un asynchronisme entre deux évènements synchrones ; lors de la

comparaison des calculs intermédiaires ou des sorties à émettre, si une seule des deux sous-unités détecte un asynchronisme ou si les deux sous-unités détectent un asynchronisme différent, l'APIdS ira vers sa situation de repli sûre. En utilisant deux modules différents, la probabilité de détection de la présence d'asynchronisme d'interprétation (ou d'interprétations différentes, quoique asynchrones) est presque doublée (la probabilité d'avoir une erreur sur le module d'entrées $B1$ ou sur le module d'entrées $B2$ est égale à la somme des probabilités d'avoir une erreur sur chacun des modules $B1$ et $B2$, moins la probabilité d'avoir une erreur sur les modules d'entrées $B1$ et $B2$ en même temps : $p(B1 \cup B2) = p(B1) + p(B2) - p(B1 \cap B2)$).

En conclusion, si l'utilisation d'APIdS permet d'éviter des erreurs dues à la défaillance de l'un de leurs composants, elle ne permet pas de passer outre l'asynchronisme introduit par la lecture des signaux logiques d'entrées.

Synthèse

La première contribution apportée dans ce chapitre est la définition d'une méthode de génération automatique de séquence de test à partir d'une spécification Grafcet. L'approche présentée est basée sur la méthode du Tour de Transition (méthode TT) et permet la génération automatique d'une séquence de test monolithique complète et de longueur minimale.

Les temps de calcul restant raisonnables pour des contrôleurs logiques critiques, l'intégration de cette méthode dans un contexte industriel a été étudiée avec la société Geensoft pour leur outil logiciel ControlBuild.

La seconde contribution est la mise en évidence d'erreurs de verdict lors de l'utilisation de séquences de test basées sur une séquence de vecteurs d'entrées non-adjacents. Une campagne d'essais, réalisée pour plusieurs configurations (mode d'exécution et répartition des entrées logiques sur les modules), a permis de quantifier les taux d'erreur de verdicts en fonction de la configuration de l'implantation à tester.

Cette approche expérimentale met en évidence la nécessité de la prise en compte du comportement détaillé de l'implantation vis-à-vis de la perception des signaux d'entrées logiques.

La génération de séquence ne doit donc pas être basée uniquement sur le comportement de la spécification fonctionnelle, mais également prendre en compte les caractéristiques

de l'implantation matérielle afin d'éviter que des changements d'entrées logiques générés de manière synchrone par le banc de test ne soient perçus comme asynchrones par l'implantation à tester. Les résultats de cette réflexion sont présentés dans le chapitre suivant.

Chapitre 4

Génération de séquences adaptées au test de conformité d'API

Sommaire

Introduction

Comme indiqué dans le précédent chapitre, il existe de nombreuses méthodes permettant de générer une séquence de test à partir d'une spécification formelle sous forme de machine de Mealy. La méthode choisie (méthode du Tour de Transition (méthode TT)) permet d'optimiser la longueur des séquences de test utilisées. En revanche, ni cette méthode, ni aucune des autres méthodes appliquées à une machine de Mealy ne permettent de prendre en compte le nombre de changements d'entrées logiques entre deux pas de test consécutifs. Les expérimentations présentées dans le précédent chapitre ont montré que le changement synchrone de plusieurs entrées logiques peut être perçu de manière asynchrone par l'Automate Programmable Industriel (API) testé, ce qui entraine un risque de verdict biaisé ou non-valide.

Ce chapitre présente notre dernière contribution : la génération automatique de séquence de test complète ne comportant qu'un seul changement d'entrées logiques entre deux pas de test consécutifs. Dans un premier temps, nous définirons un critère de SIC-testabilité puis vérifierons l'existence d'une solution à ce problème. Pour les spécifications où cette solution existe, nous étudierons ensuite la génération de séquence de test Single Input Change (SIC). Dans le cas contraire, nous montrerons qu'il est possible de construire une séquence de test complète minimisant le nombre de pas de test nécessitant un changement de plusieurs variables d'entrées logiques.

Les travaux présentés dans ce chapitre on fait l'objet de communications en conférences internationales (Provost *et al.* (2010b, 2011c)).

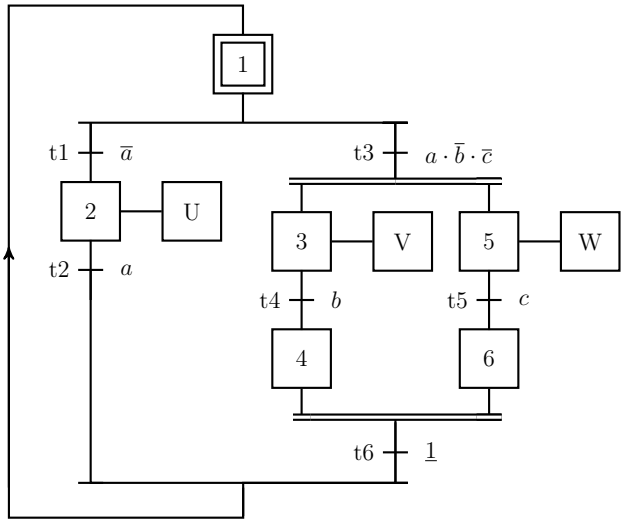
Ce chapitre est organisé comme suit :

- la première section précise les définitions des séquences MIC et SIC ;
- la deuxième section est consacrée à la définition et la vérification de la SIC-testabilité d'un contrôleur logique ;
- la troisième section propose une méthode permettant la génération automatique d'une séquence de test SIC pour la partie SIC-testable du comportement ;
- la quatrième section présente la méthode proposée pour la génération automatique d'une séquence de test Maximum Consecutive Single Input Change (MaxC-SIC) complète maximisant le nombre de pas de test SIC consécutifs ;
- la cinquième partie propose une méthode permettant la génération automatique d'une séquence de test Minimum Multiple Input Change (min-MIC) complète

minimisant le nombre de pas de test Multiple Input Change (MIC).

1 Présentation de l'exemple utilisé

La démarche proposée dans ce chapitre est illustrée sur un exemple simple, dont la spécification Grafcet est donnée figure 4.1 ; il sera appelé exemple C par la suite. Cet exemple comporte 3 entrées (a, b et c) et 3 sorties (U, V et W) ; ainsi, 8 valuations d'entrée (v_I) et 8 valuations de sortie (v_O) peuvent être définies pour cet exemple.



Entrées				Sorties			
a	Bac vide	b	Niveau B atteint	U	Vider le bac	V	Verser B
c	Niveau C atteint			W	Verser C		

Figure 4.1 – Spécification Grafcet utilisée pour l'illustration de la SIC-testabilité (Exemple C)

Le calcul de l'Automate des Localités Stables (ALS) de cet exemple ainsi que sa transcription en une machine de Mealy permettent de représenter le comportement formalisé de cette spécification Grafcet sous différentes formes. La définition de l'ALS par le 5-uplet $(I_{ALS}, O_{ALS}, L, l_{Init}, Evol)$ est donnée figure 4.2, tandis que les tableaux 4.1a et 4.1b, similaires aux tables de Huffman (Huffman (1954)), illustrent les définitions des fonctions de transition δ_M de sortie λ_M de la machine de Mealy équivalente. L'espace d'état de cet exemple étant composé de 5 localités ou états, les fonctions δ_M et λ_M sont définies pour chaque valeur des 40 couples $(s, v_I) \in S_M \times I_M$.

$$
\begin{aligned}
I_{ALS} &= \{a, b, c\},\\
O_{ALS} &= \{U, V, W\},\\
L &= \{l_1 : (\{1\}, \varnothing, a \cdot b + a \cdot c),\ l_2 : (\{3,5\}, \{V,W\}, \overline{b} \cdot \overline{c}),\ l_3 : (\{2\}, \{U\}, \overline{a}),\\
&\quad\ l_4 : (\{4,5\}, \{W\}, \overline{c}),\ l_5 : (\{3,6\}, \{V\}, \overline{b}),\},\\
l_{Init} &= \ l_0 : (\{1\}, 0),\\
Evol &= \{(l_0, l_1, a \cdot b + a \cdot c),\ (l_0, l_2, a \cdot \overline{b} \cdot \overline{c}),\ (l_0, l_3, \overline{a}),\ (l_1, l_2, a \cdot \overline{b} \cdot \overline{c}), (l_1, l_3, \overline{a}),\\
&\quad\ (l_2, l_1, a \cdot b \cdot c),\ (l_2, l_3, \overline{a} \cdot b \cdot c),\ (l_2, l_4, b \cdot \overline{c}),\ (l_2, l_5, \overline{b} \cdot c),\ (l_3, l_1, a \cdot b + a \cdot c),\\
&\quad\ (l_3, l_2, a \cdot \overline{b} \cdot \overline{c}),\ (l_4, l_1, a \cdot c),\ (l_4, l_3, \overline{a} \cdot c),\ (l_5, l_1, a \cdot b),\ (l_5, l_3, \overline{a} \cdot b)\}
\end{aligned}
$$

Figure 4.2 – Définition de l'ALS pour le Grafcet donné figure 4.1

Dans le tableau 4.1a, le contenu de chaque case indique l'état atteint s_t depuis l'état s_s pour la valuation d'entrée v_I ; dans le tableau 4.1b, le contenu de chaque case indique la valuation de sortie émise v_O lors du franchissement d'une transition depuis l'état s_s vers l'état s_t pour la valuation d'entrée v_I. L'agencement des valuations dans ces tableaux est fait suivant un code Gray, afin de faciliter la compréhension des valuations adjacentes [1] (au sens logique).

Les cases marquées d'un cercle représentent les couples (s, v_I) stables, c'est-à-dire les couples (s, v_I) pour lesquels $\delta_M(s, v_I) = s$; seules les sorties associées à ces couples stables sont observables. Les cases marquées du chiffre 0 dans le coin en haut à gauche représentent les couples (s, v_I) atteignables depuis s_{Init} lors de l'initialisation, c'est-à-dire les couples (s, v_I) pour lesquels $\delta_M(s_{Init}, v_I) = s$.

Comme le comportement défini est déterministe et complètement spécifié, chaque case des tableaux 4.1a et 4.1b contient une et une seule valeur. De plus, comme ce comportement ne contient pas d'évolution fugace (évolution passant par un état intermédiaire s instable), chaque case du tableau 4.1a représente soit un couple (s, v_I) stable, soit une évolution vers un couple (s, v_I) stable. Par exemple, lorsque l'état actif est s_1, le changement de la valuation d'entrée i_7 vers i_0, provoque une évolution vers l'état s_3 : le couple (s_1, i_7) est stable car $\delta_M(s_1, i_7) = s_1$, le changement de la valuation d'entrée i_7

1. Deux valuations d'entrées sont adjacentes si elles diffèrent d'une seule entrée logique. Une définition formelle est donnée équation 4.3

	i_0 ou $\overline{a}\cdot\overline{b}\cdot\overline{c}$ ou $\{\}$	i_1 ou $\overline{a}\cdot\overline{b}\cdot c$ ou $\{c\}$	i_3 ou $\overline{a}\cdot b\cdot c$ ou $\{b,c\}$	i_2 ou $\overline{a}\cdot b\cdot\overline{c}$ ou $\{b\}$	i_6 ou $a\cdot b\cdot\overline{c}$ ou $\{a,b\}$	i_7 ou $a\cdot b\cdot c$ ou $\{a,b,c\}$	i_5 ou $a\cdot\overline{b}\cdot c$ ou $\{a,c\}$	i_4 ou $a\cdot\overline{b}\cdot\overline{c}$ ou $\{a\}$
s_1	s_3	s_3	s_3	s_3	$^0(s_1)$	$^0(s_1)$	$^0(s_1)$	s_2
s_2	(s_2)	s_5	s_3	s_4	s_4	s_1	s_5	$^0(s_2)$
s_3	$^0(s_3)$	$^0(s_3)$	$^0(s_3)$	$^0(s_3)$	s_1	s_1	s_1	s_2
s_4	(s_4)	s_3	s_3	(s_4)	(s_4)	s_1	s_1	(s_4)
s_5	(s_5)	(s_5)	s_3	s_3	s_1	s_1	(s_5)	(s_5)

(a) Fonction de transition $\delta_M(s, v_I)$

	i_0 ou $\overline{a}\cdot\overline{b}\cdot\overline{c}$ ou $\{\}$	i_1 ou $\overline{a}\cdot\overline{b}\cdot c$ ou $\{c\}$	i_3 ou $\overline{a}\cdot b\cdot c$ ou $\{b,c\}$	i_2 ou $\overline{a}\cdot b\cdot\overline{c}$ ou $\{b\}$	i_6 ou $a\cdot b\cdot\overline{c}$ ou $\{a,b\}$	i_7 ou $a\cdot b\cdot c$ ou $\{a,b,c\}$	i_5 ou $a\cdot\overline{b}\cdot c$ ou $\{a,c\}$	i_4 ou $a\cdot\overline{b}\cdot\overline{c}$ ou $\{a\}$
s_1	o_4	o_4	o_4	o_4	(o_0)	(o_0)	(o_0)	o_3
s_2	(o_3)	o_2	o_4	o_1	o_1	o_0	o_2	(o_3)
s_3	(o_4)	(o_4)	(o_4)	(o_4)	o_0	o_0	o_0	o_3
s_4	(o_1)	o_4	o_4	(o_1)	(o_1)	o_0	o_0	(o_1)
s_5	(o_2)	(o_2)	o_4	o_4	o_0	o_0	(o_2)	(o_2)

(b) Fonction de sortie $\lambda_M(s, v_I)$

avec : $m(o_0) = \overline{U}\cdot\overline{V}\cdot\overline{W},\ m(o_1) = \overline{U}\cdot\overline{V}\cdot W,\ m(o_2) = \overline{U}\cdot V\cdot\overline{W},$
$m(o_3) = \overline{U}\cdot V\cdot W,\ m(o_4) = U\cdot\overline{V}\cdot\overline{W}$

Tableau 4.1 – Représentation tabulaire du comportement du Grafcet donné figure 4.1

vers i_0 rend l'état s_1 instable et provoque l'évolution vers l'état s_3 car $\delta_M(s_1, i_0) = s_3$, le couple atteint (s_3, i_0) est stable car $\delta_M(s_3, i_0) = s_3$.

Ces représentations tabulaires sont ici données uniquement dans un but d'illustration de la méthode. La méthode proposée reposant sur des calculs effectués de manière symbolique sur des expressions booléennes, seule la définition ensembliste de l'ALS est nécessaire. Ces représentations tabulaires représentent en extension le comportement décrit par la spécification Grafcet et permettent ainsi d'identifier rapidement "à la main" les vecteurs d'entrées adjacents, ce qui facilite de notre point de vue la compréhension de la méthode appliquée à l'exemple.

2 Définition des séquences de test SIC et MIC

Cette section présente tout d'abord la différence entre une séquence de test SIC et une séquence de test MIC (figure 4.3), puis propose une définition formelle d'une séquence

de test SIC.

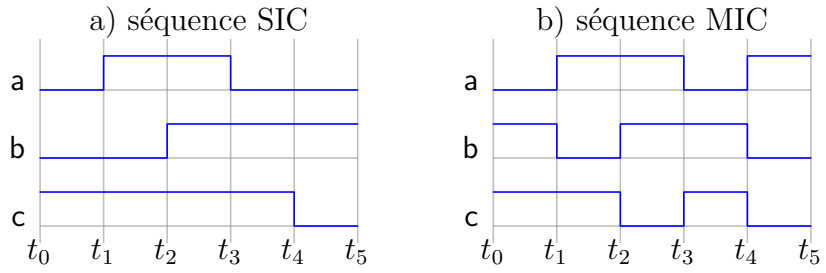


Figure 4.3 – Exemples de séquences SIC et MIC

Dans une séquence de test SIC, la séquence d'entrée est telle qu'une seule entrée logique puisse changer à un seul et même moment ; les changements synchrones de plusieurs entrées ne sont pas permis. À l'inverse, une séquence de test MIC peut comprendre des changements synchrones de plusieurs entrées. Si l'usage de séquence de test SIC a déjà été étudié, dans le cadre du test de circuits intégrés, les préoccupations des auteurs de ces travaux sont différentes des nôtres. En effet, la réduction de la consommation électrique (Yi *et al.* (2008)) d'un API durant sa phase de test n'est pas un problème prioritaire ; et l'étude de l'efficacité des techniques de test temporisé (Virazel *et al.* (2001)) ne sera pas non plus étudiée puisque nous nous limitons à l'étude de spécifications Grafcet non temporisées.

2.1 Influence d'une séquence de test MIC sur les erreurs d'interprétation

En revanche, la génération de séquences de test SIC pour le test de conformité d'API mérite d'être étudiée car l'utilisation de telles séquences permet d'éviter l'obtention des verdicts erronés, lorsque le banc de test et le contrôleur logique à tester ne sont pas synchronisés. En effet, lorsqu'une séquence de test MIC est émise par le banc de test, les changements d'entrées émis de manière synchrone peuvent être vus de manière asynchrone par le contrôleur logique à tester. Comme indiqué dans le chapitre précédent, le fonctionnement cyclique (ou périodique) de l'implantation implique qu'un changement simultané de plusieurs entrées logiques puisse être perçu de plusieurs manières différentes (voir chapitre 3, section 5 et figure 3.5).

Une solution évidente permettant de remédier à ce problème eut été de synchroniser le banc de test et le contrôleur à tester ; cette solution est souvent impossible à mettre

en œuvre lorsqu'un test non invasif est requis. C'est pourquoi la génération de séquence de test SIC doit être étudiée.

Pour un système logique combinatoire, il est toujours possible de construire une séquence de test SIC, en utilisant un arrangement des variables d'entrée suivant le code Gray. Pour des systèmes logiques séquentiels, l'existence d'une séquence de test SIC ne peut être garantie. En effet, le parcours de l'ensemble de l'espace d'états ainsi que de toutes les évolutions possibles, peut nécessiter des changements simultanés d'entrées logiques. La génération d'une séquence de test SIC couvrant la totalité de l'espace d'états et des évolutions ne peut donc être effectuée qu'après avoir vérifié l'existence de cette solution. Si cette solution n'existe pas, il faut déterminer la partie de la spécification qui peut être testée par une séquence de test SIC.

La vérification de l'existence de cette solution ainsi que la détermination de la partie de la spécification Grafcet qui est SIC-testable est étudiée dans la section 3. La génération d'une séquence de test SIC pour la partie de la spécification Grafcet testable est étudiée dans la section 4. La section 5 traite de la génération d'une séquence de test maximisant le nombre de pas de test SIC consécutifs ; tandis que la section 6 propose une optimisation de la méthode de génération d'une séquence de test minimisant le nombre de pas de test MIC.

L'analyse de la SIC-testabilité d'une spécification Grafcet nécessite une définition formelle d'une séquence de test qui repose sur des variables d'entrée/sortie.

2.2 Définition d'une séquence de test SIC

Une séquence de test SIC repose sur la définition mathématique donnée équation 3.3 (page 92). Une séquence de test SIC doit également vérifier les deux propriétés données équations 3.4 et 3.6, que nous rappelons ici, ainsi qu'une troisième propriété liée au caractère SIC de cette séquence.

Une séquence de test SIC doit être :

Propriété 1 : Initialisable, si et seulement si l'état source du premier pas de test est atteignable lors de l'initialisation, et que la valuation d'entrée v_I^0 utilisée lors de l'initialisation laisse cet état stable.

$$s^0 \in \{s \in S \mid \exists v_I^0 \in I : s = \delta_M(s_{Init}, v_I^0)\} \tag{4.1}$$

Propriété 2 : Complète, si et seulement si il y a au moins un pas de test pour chaque élément de la fonction de transition.

$$\forall (s, v_I) \in (S \times I),\ (s, v_I, \delta_M(s, v_I), \lambda_M(s, v_I)) \in TS \tag{4.2}$$

Propriété 3 : Basée sur une séquence d'entrée SIC

Pour exprimer de façon formelle cette dernière propriété, la relation SIC entre deux valuations d'entrée doit être tout d'abord définie. La définition exprimée ci-dessous est basée sur la représentation d'une valuation d'entrée par le sous-ensemble des variables d'entrées I qui contient uniquement les variables qui sont vraies pour cette valuation. Ainsi, deux valuations d'entrée v_I et v'_I satisfont une relation SIC si et seulement si :

$$dim((\mathbb{1}(v_I) \backslash \mathbb{1}(v'_I)) \cup (\mathbb{1}(v'_I) \backslash \mathbb{1}(v_I))) = 1 \tag{4.3}$$

Pour un contrôleur logique à n entrées logiques, chaque valuation d'entrée v_I satisfait n relations SIC avec une autre valuation d'entrée.

Par exemple, la valuation d'entrée représentée par le minterme $\overline{a} \cdot \overline{b} \cdot c$ satisfait une relation SIC avec les mintermes $a \cdot \overline{b} \cdot c$, $\overline{a} \cdot b \cdot c$ et $\overline{a} \cdot \overline{b} \cdot \overline{c}$ car :

$$dim((\{c\} \backslash \{a, c\}) \cup (\{a, c\} \backslash \{c\})) = dim(\{\} \cup \{a\}) = 1$$
$$dim((\{c\} \backslash \{b, c\}) \cup (\{b, c\} \backslash \{c\})) = dim(\{\} \cup \{b\}) = 1$$
$$dim((\{c\} \backslash \{\}) \cup (\{\} \backslash \{c\})) = dim(\{c\} \cup \{\}) = 1$$

Dans la suite de ce chapitre, cette relation symétrique sera notée : $v_I\ R_{Gray}\ v'_I$.

$$v_I\ R_{Gray}\ v'_I \Leftrightarrow dim((\mathbb{1}(v_I) \backslash \mathbb{1}(v'_I)) \cup (\mathbb{1}(v'_I) \backslash \mathbb{1}(v_I))) = 1 \tag{4.4}$$

Ainsi, une séquence de test TS est basée sur une séquence d'entrée SIC et appelée séquence de test SIC si et seulement si :

$$TS = [(s^0, v_I^0, \delta_M(s^0, v_I^0), \lambda_M(s^0, v_I^0)), \cdots, (s^n, v_I^n, \delta_M(s^n, v_I^n), \lambda_M(s^n, v_I^n))] \mid \forall k > 1,\ v_I^k\ R_{Gray}\ v_I^{k-1} \tag{4.5}$$

Il n'est malheureusement pas possible de construire, pour toute spécification, une

séquence de test cohérente qui puisse satisfaire à la fois les propriétés 2 et 3 énoncées ci-dessus ; la vérification de l'aptitude d'une spécification à permettre la construction d'une séquence respectant ces deux propriétés, appelée SIC-testabilité, est détaillée dans la section 3.

3 Vérification de la SIC-testabilité d'un contrôleur logique

Cette section a pour but de montrer que la SIC-testabilité d'un contrôleur logique peut être vérifiée par un calcul de point fixe sur des cas de test élémentaires.

3.1 Règles d'évolution entre les couples (s, v_I)

Une séquence de test initialisable et complète doit commencer depuis un couple (s, v_I) initialisable (indiqué par une case marquée du chiffre 0 dans le tableau 4.1a) et comprendre tous les couples $(s, v_I) \in S_M \times I_M$ (toutes les cases du tableau).

Une séquence de test cohérente (équation 3.3) autorise tous les changements des valuations d'entrée entre deux pas de test successifs. Cependant, l'état source de chaque pas de test doit être identique à l'état cible du pas de test précédent.

Ainsi, du point de vue du tableau 4.1a, une séquence de test cohérente contient uniquement des changements de cases horizontaux et verticaux en accord avec les règles suivantes :

Règle 1 : Depuis les cases qui contiennent un nom d'état entouré (les états source et cible du pas de test associé sont identiques), seuls les changements horizontaux sont possibles.

Règle 2 : Depuis les cases qui contiennent un nom d'état non entouré (les états source et cible du pas de test associé sont différents), seul le changement vertical indiqué par le contenu de la case est possible.

Depuis la case associée au couple (s_1, i_7), les changements horizontaux vers les 7 cases associées aux couples (s_1, i_0) à (s_1, i_6) sont possibles. Depuis, la case associée au couple (s_1, i_0), seule le changement vertical vers la case associée au couple (s_3, i_0) est possible.

Afin de construire une séquence de test SIC, une condition restrictive doit être ajoutée

à la règle 1. Comme ce type de séquence exclut tout changement simultané de plusieurs valeurs d'entrées, un changement horizontal doit correspondre à un couple de valuations d'entrée qui satisfont une relation SIC. Dans le cas d'un contrôleur avec n entrées logiques, seules n cases parmi $(2^n - 1)$ peuvent être atteintes depuis une case contenant un nom d'état entouré, d'où :

Règle 3 : Depuis les cases qui contiennent un nom d'état entouré, seuls les changements horizontaux entre deux valuations d'entrée satisfaisant une relation SIC sont possibles.

Seule la règle 3 nécessite d'être prise en compte pour déterminer si une séquence de test est une séquence de test SIC, c'est-à-dire si les valuations d'entrée v_I et v'_I de tous les couples des pas de test successifs satisfont $v_I \; R_{Gray} \; v'_I$.

Par exemple C, pour le comportement décrit dans le tableau 4.1a, depuis la case associée au couple (s_1, i_7), les changements horizontaux vers les 7 cases associées aux couples (s_1, i_0) à (s_1, i_6) sont possibles, mais seules les 3 cases associées aux couples (s_1, i_3), (s_1, i_5) et (s_1, i_6) respectent la règle 3. Par conséquent, en appliquant les trois règles, depuis le couple (s_1, i_7) les 3 couples (s_3, i_3), (s_1, i_5) et (s_1, i_6) sont atteignables en satisfaisant une relation SIC.

3.2 Principe de la méthode de vérification de la SIC-testabilité

La méthode de vérification de la SIC-testabilité d'un contrôleur logique proposée est basée sur les deux observations suivantes :

- Un cas de test élémentaire (s_s, v_I, s_t, v_O) d'un comportement donné est SIC-testable s'il peut être inclus dans une séquence de test SIC initialisable pour ce comportement. En raison des caractéristiques d'exécution des tests, le cas de test élémentaire (s_t, v_I, s_t, v_O) est également SIC-testable (cf. section 2.4, 95).
- Si le cas de test élémentaire (s_t, v_I, s_t, v_O) est SIC-testable, il est toujours possible d'ajouter à la séquence de test un cas de test élémentaire $(s_t, v'_I, \delta_M(s_t, v'_I), \lambda_M(s_t, v'_I))$ où v'_I satisfait : $v'_I \; R_{Gray} \; v_I$.

L'ensemble des cas de test élémentaires SIC-testables peut être obtenu par un calcul de point fixe. Un calcul de point fixe est une méthode itérative de calcul qui peut être appliquée sur une fonction monotone, et a pour objectif la détection d'extremum de cette fonction. Ce type de calcul consiste à itérer la fonction étudiée à partir d'une valeur

ou d'un ensemble donné, ce qui permet d'obtenir une suite croissante convergeant vers l'extremum recherché, appelé point fixe. Dans le cas de la méthode présentée, le calcul de point fixe est initialisé par un ensemble de couples (s, v_I), puis cet ensemble est étendu en y ajoutant d'autres couples (s, v_I) ; la fonction est donc strictement monotone. Le calcul de point fixe s'arrête lorsque la dimension de cet ensemble reste identique entre deux itérations, l'ensemble maximum étant atteint.

En prenant en compte les deux observations précédentes, l'ensemble des cas de test élémentaires SIC-testables peut être obtenu par le calcul de point fixe suivant :

$$R_{SIC}(0) = \left\{(s_{Init}, v_I^0, s, \lambda_M(s_{Init}, v_I^0)) \mid v_I^0 \in I_M,\ \delta_M(s_{Init}, v_I^0) = s\right\} \tag{4.6}$$

$$\begin{aligned} R_{SIC}(n+1) = R_{SIC}(n) \cup \\ \Bigg\{ \left\{(s_k, v_I^{k+1}, \delta_M(s_k, v_I^{k+1}), \lambda_M(\delta_M(s_k, v_I^{k+1})))\right\} \cup \\ \left\{(\delta_M(s_k, v_I^{k+1}), v_I^{k+1}, \delta_M(s_k, v_I^{k+1}), \lambda_M(\delta_M(s_k, v_I^{k+1})))\right\} \mid \\ \exists (s_k, v_I^k) \in R_{SIC}(n) \mid \begin{cases} \delta_M(s_k, v_I^k) = s_k \\ v_I^{k+1}\ R_{Gray}\ v_I^k \end{cases} \Bigg\} \end{aligned} \tag{4.7}$$

À la fin de ce calcul itératif, le contrôleur est SIC-testable si et seulement si l'ensemble final contient tous les cas de test élémentaires qui peuvent être définis à partir de sa description comportementale. Sinon, cet ensemble final définit la partie SIC-testable du comportement étudié.

La connaissance de cet ensemble, noté R_{SIC}^{Maxi}, représentant la partie SIC-testable du comportement permet de définir un taux de couverture de l'implantation, appelé *taux de couverture SIC*, pour lequel le test de conformité de l'implantation est garanti sans erreur de verdict. Le taux de couverture SIC définit le rapport entre le nombre de transitions de la machine de Mealy pouvant être testées sans erreurs de verdict due à une mauvaise interprétation par l'API d'un changement synchrone de plusieurs entrées logiques et le nombre de transitions de la machine de Mealy ; il est défini comme suit :

$$\text{Taux de couverture SIC } = \frac{|R_{SIC}^{Maxi}|}{|S_M| \times |I_M|} \tag{4.8}$$

Le tableau 4.2 présente les résultats de ce calcul pour l'exemple C présenté figure 4.1. L'indice k de l'itération à laquelle le cas de test a été déterminé SIC-testable est indiqué

dans le coin supérieur gauche de chaque case. Par exemple, le cas de test élémentaire associé à la case (s_2, i_4) est obtenu à l'itération 0. Les cas de test élémentaires associés aux cases (s_2, i_6) et (s_4, i_6) sont obtenus à l'itération 1, car $(a \cdot \overline{b} \cdot \overline{c})\ R_{Gray}\ (a \cdot b \cdot \overline{c})$, et ainsi de suite. Le calcul s'arrête à la cinquième itération, initialisation exclue.

L'ensemble final R_{SIC}^{Maxi} contient seulement 37 cas de test ; les cas qui n'appartiennent pas à cet ensemble sont représentés par des cases grisées. Par conséquent, le comportement de ce contrôleur *n'est pas complètement SIC-testable* ; sa partie SIC-testable est définie par l'ensemble des cases non grisées. Le taux de couverture SIC de la séquence couvrant cet ensemble est donc égal à $\frac{37}{40}$.

	i_0 ou $\overline{a} \cdot \overline{b} \cdot \overline{c}$ ou $\{\}$	i_1 ou $\overline{a} \cdot \overline{b} \cdot c$ ou $\{c\}$	i_3 ou $\overline{a} \cdot b \cdot c$ ou $\{b,c\}$	i_2 ou $\overline{a} \cdot b \cdot \overline{c}$ ou $\{b\}$	i_6 ou $a \cdot b \cdot \overline{c}$ ou $\{a,b\}$	i_7 ou $a \cdot b \cdot c$ ou $\{a,b,c\}$	i_5 ou $a \cdot \overline{b} \cdot c$ ou $\{a,c\}$	i_4 ou $a \cdot \overline{b} \cdot \overline{c}$ ou $\{a\}$
s_1	s_3	${}^{1}s_3$	${}^{1}s_3$	${}^{1}s_3$	${}^{0}(s_1)$	${}^{0}(s_1)$	${}^{0}(s_1)$	${}^{1}s_2$
s_2	${}^{1}(s_2)$	${}^{2}s_5$	s_3	${}^{2}s_4$	${}^{1}s_4$	s_1	${}^{1}s_5$	${}^{0}(s_2)$
s_3	${}^{0}(s_3)$	${}^{0}(s_3)$	${}^{0}(s_3)$	${}^{0}(s_3)$	${}^{1}s_1$	${}^{1}s_1$	${}^{1}s_1$	${}^{1}s_2$
s_4	${}^{3}(s_4)$	${}^{4}s_3$	${}^{3}s_3$	${}^{2}(s_4)$	${}^{1}(s_4)$	${}^{2}s_1$	${}^{3}s_1$	${}^{2}(s_4)$
s_5	${}^{3}(s_5)$	${}^{2}(s_5)$	${}^{3}s_3$	${}^{4}s_3$	${}^{3}s_1$	${}^{2}s_1$	${}^{1}(s_5)$	${}^{2}(s_5)$

Tableau 4.2 – Illustration des étapes du calcul de la partie SIC-testable, par calcul de point fixe

3.3 Application du calcul de point fixe à partir de l'ALS

Comme mentionné dans la section 4, la représentation énumérée du comportement, sous forme tabulaire, ne peut être utilisée pour la description de comportements plus complexes. Cette sous-section décrit l'application de la méthode du calcul de point fixe à partir d'un comportement formalisé par un ALS.

La base de cette application repose sur la manipulation des valuations d'entrée au travers d'expressions booléennes et non de façon énumérée.

La vérification de la SIC-testabilité et le calcul de l'ensemble R_{SIC}^{Maxi} sont obtenus simultanément, tandis que la vérification de l'existence d'une séquence de test SIC monolithique couvrant l'ensemble R_{SIC}^{Maxi} nécessite le calcul de la matrice des plus courts chemins sur l'ensemble R_{SIC}^{Maxi}. Ces deux phases de calcul sont définies comme suit :

Phase 1 : Calcul de la SIC-testabilité au meilleur cas. La région initiale est composée de l'ensemble de tous les couples (s_{Init}, v_I). La région obtenue à la fin du calcul de point fixe est appelée R_{SIC}^{Maxi}.

Phase 2 : Calcul de la matrice des plus courts chemins. Cette matrice carrée, de dimension $|S_M \times I_M|^2$, est initialisée avec les arcs du graphe représentant la région SIC, puis complétée en utilisant l'algorithme de Floyd-Warshall.

Selon la connexité de la région R_{SIC}^{Maxi} obtenue, différents cas sont possibles. L'algorigramme présenté figure 4.4 décrit ces différents cas. Dans le meilleur des cas, la région R_{SIC}^{Maxi} est fortement connexe ; une séquence de test SIC monolithique pourra alors être générée pour cette région. Si la région n'est pas fortement connexe, mais qu'elle ne possède qu'un unique couple (s, v_I) non réatteignable, une séquence de test SIC monolithique pourra également être générée pour cette région, mais le couple (s, v_I) appliqué lors de l'initialisation est imposé. Dans les autres cas, si plusieurs couples (s, v_I) sont non réatteignables, il n'est pas possible de générer une séquence de test SIC monolithique pour l'ensemble de la région R_{SIC}^{Maxi} ; plusieurs séquences de test devront être générées pour les sous-régions connexes, ces séquences devront ensuite être raccordées par des séquences de test MIC ou par une réinitialisation de l'implantation.

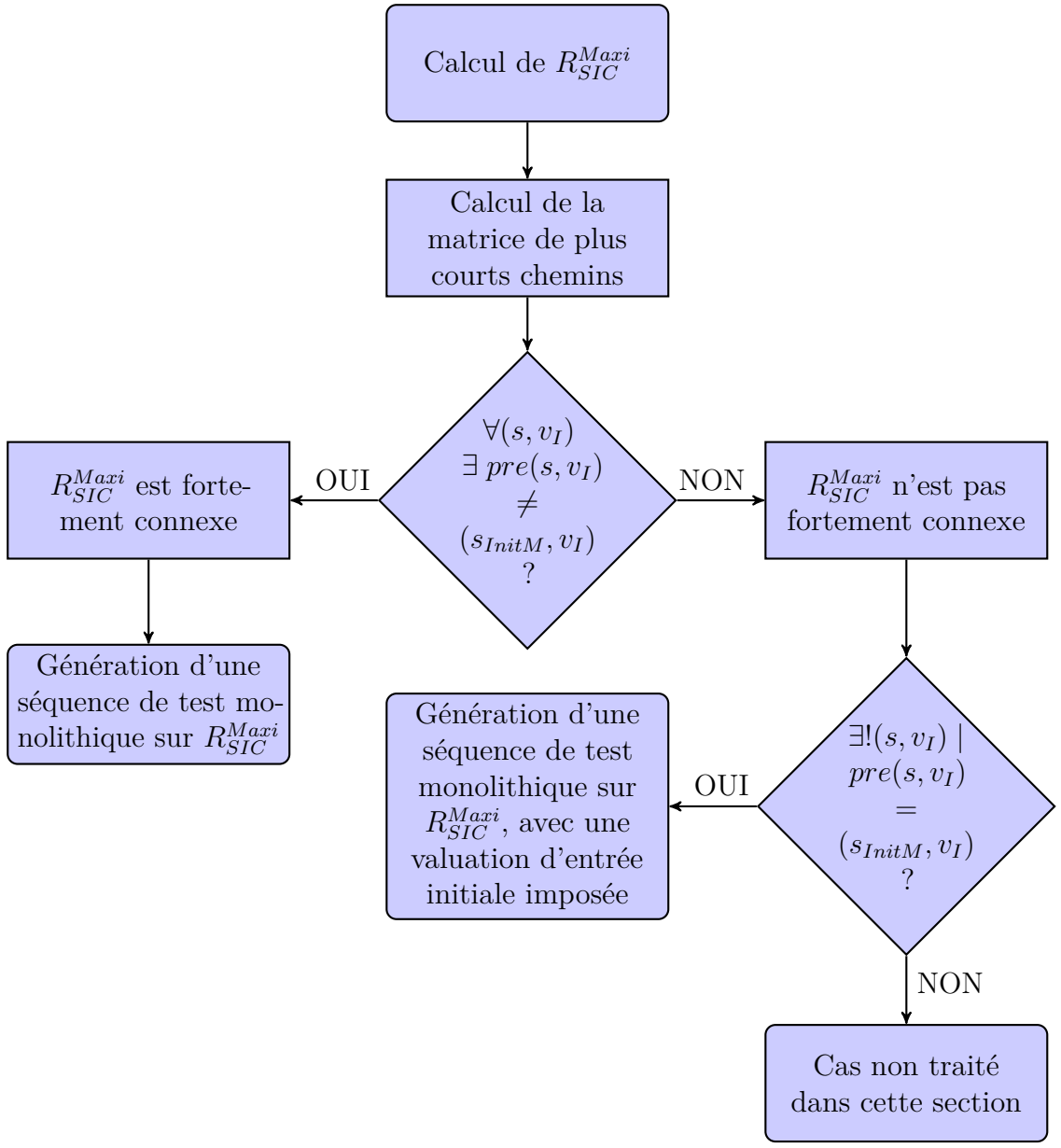
Le calcul de la région R_{SIC}^{Maxi} est donc initialisé à partir de l'ensemble des couples (s, v_I) initialisable :

$$R_{SIC}(0) = \left\{(s_{Init}, v_I^0, s, \lambda_M(s_{Init}, v_I^0)) \mid v_I^0 \in I_M,\ \delta_M(s_{Init}, v_I^0) = s\right\} \tag{4.9}$$

L'algorithme 2 permet d'illustrer l'automatisation de l'analyse de la SIC-testabilité.

Cet algorithme repose principalement sur une opération, notée $Expansion_{Gray}$, basée sur des calculs symboliques et permettant d'étendre un ensemble donné I^A de valuations d'entrée v_I avec l'ensemble des valuations d'entrée v'_I qui satisfont $v'_I\ R_{Gray}\ v_I$. L'ensemble étendu, noté I^B, est défini comme suit :

$$I^B = I^A \cup \left\{v'_I \mid \exists v_I \in I^A :\ v'_I\ R_{Gray}\ v_I\right\}, \tag{4.10}$$



Notation :

$pre(s, v_I) = \{(s_s, v_I) \in S_M \cup s_{InitM} \times I_M \mid \delta_M(s_s, v_I) = s\}$

Figure 4.4 – Algorigramme du calcul de R_{SIC} et de la génération d'une séquence de test SIC

Algorithm 2 Calcul de la partie SIC-testable du comportement

Inputs : Comportement $\mathcal{B} : (S, s_{Init}, \delta_M, \lambda_M)$
Outputs : Partie SIC-testable du comportement $\mathcal{B}$

```
/* Initialisation : */
for all s_i ∈ S do
  T_SIC(s_i) := Exp_AlwaysFalse /* Valuations d'entrée SIC-testable de s_I */
  Acc(s_i) := Exp_AlwaysFalse /* Valuations d'entrée de s_i atteignable par une séquence
  SIC */ /* Exp_AlwaysFalse est l'expression booléenne toujours fausse */
end for
T_SIC(s_Init) := E_11 /* Puisque s_Init = s_1 */
Acc(s_Init) := E_11 /* Puisque s_Init = s_1 */
/* Calcul itératif : */
Improvements := True
while Improvements do
  Improvements := False
  for all s_i in S do
    New_SIC(s_i) := Expansion_Gray(Acc(s_i)) · not(T_SIC(s_i))
    if New_SIC(s_i) ≠ Exp_AlwaysFalse then
      Improvements := True
      T_SIC(s_i) := T_SIC(s_i) + New_SIC(s_i)
      for all evolution from s_i to s_j do
        new := New_SIC(s_i) · E_ij
        if new ≠ Exp_AlwaysFalse /* Condition facultative ... */ then
          Acc(s_j) := Acc(s_j)+new /* ... car si new = Exp_AlwaysFalse alors Acc(s_j) :=
          Acc(s_j) + Exp_AlwaysFalse = Acc(s_j) */
        end if
      end for
    end if
  end for
end while
/* Affichage des résultats : */
for s_i in S do
  SICTestablePart := T_SIC(s_i)
  print "Partie SIC-testable ",s_i," : ",SICTestablePart
end for
```

Afin d'appliquer cette méthode à la manipulation d'expressions booléennes, cet ensemble peut être reformulé de la façon suivante : soit Exp_A l'expression booléenne qui représente l'ensemble I^A, l'expression booléenne qui représente l'ensemble I^B est défini comme suit :

$$\begin{aligned} Exp_B &= Expansion_{Gray}(Exp_A) \\ &= \sum_{i \in I} \left(Exp_{A_{|i \leftarrow Faux}} + Exp_{A_{|i \leftarrow Vrai}} \right) \end{aligned} \tag{4.11}$$

$Exp_{A_{|i \leftarrow Faux}}$ désigne l'expression booléenne Exp_A simplifiée après remplacement de la variable logique i par la variable logique $Faux$, tandis que $Exp_{A_{|i \leftarrow Vrai}}$ représente

l'expression booléenne Exp_A simplifiée après remplacement de la variable logique i par la variable logique $Vrai$.

L'exemple ci-dessous illustre cette opération :

$$\begin{aligned}
I^A =& \{\overline{a}\cdot\overline{b}\cdot c\,,\ \overline{a}\cdot b\cdot c\,,\ a\cdot\overline{b}\cdot\overline{c}\}\\
Exp_A =& \overline{a}\cdot c + a\cdot\overline{b}\cdot\overline{c}\\
Exp_B =& \Big((\overline{0}\cdot c + 0\cdot\overline{b}\cdot\overline{c}) + (\overline{1}\cdot c + 1\cdot\overline{b}\cdot\overline{c})\Big)\\
&+\Big((\overline{a}\cdot c + a\cdot\overline{0}\cdot\overline{c}) + (\overline{a}\cdot c + a\cdot\overline{1}\cdot\overline{c})\Big)\\
&+\Big((\overline{a}\cdot 0 + a\cdot\overline{b}\cdot\overline{0}) + (\overline{a}\cdot 1 + a\cdot\overline{b}\cdot\overline{1})\Big)\\
&=(c+\overline{b}\cdot\overline{c}) + (\overline{a}\cdot c + a\cdot\overline{c} + \overline{a}\cdot c) + (a\cdot\overline{b} + \overline{a} + \overline{b}\cdot\overline{c})\\
&=1\\
I^B =& \{\overline{a}\cdot\overline{b}\cdot\overline{c}\ ,\ \overline{a}\cdot\overline{b}\cdot c\ ,\ \overline{a}\cdot b\cdot\overline{c}\ ,\ \overline{a}\cdot b\cdot c\ ,\\
& a\cdot\overline{b}\cdot\overline{c}\ ,\ a\cdot\overline{b}\cdot c\ ,\ a\cdot b\cdot\overline{c}\ ,\ a\cdot b\cdot c\}
\end{aligned} \tag{4.12}$$

Pour l'exemple C présenté figure 4.1, le calcul de la matrice des plus courts chemins nous indique que l'ensemble R_{SIC}^{Maxi} est connexe ; une séquence de test SIC monolithique peut donc être générée pour la partie SIC-testable de la spécification.

4 Génération automatique d'une séquence de test SIC

4.1 Définition du graphe pour le calcul d'une séquence de test SIC

La séquence de test SIC de longueur minimale peut être obtenue en utilisant les résultats de la théorie des graphes pour un problème bien connu : le problème du voyageur de commerce (Dantzig *et al.* (1954)). La formulation générale de ce problème est la suivante : *"déterminer un circuit de longueur minimale passant, au moins une fois, par chaque nœud"*.

La génération automatique d'une séquence de test SIC est basée sur une représentation du comportement par un graphe orienté dont les nœuds représentent tous les couples (s, v_I) qui peuvent être définis pour la partie SIC-testable du comportement.

Le graphe utilisé pour la génération de la séquence SIC pour un contrôleur comprenant n entrées logiques est construit comme suit :

– Un nœud est associé à chaque couple $(s, v_I) \in S_M \times I_M$ de la machine de Mealy ;
– Un seul arc part d'un nœud correspondant à un couple (s, v_I) tel que $\delta_M(s, v_I) \neq s$; le nœud cible de cet arc est le nœud correspondant à $(\delta_M(s, v_I), v_I)$;
– n arcs partent d'un nœud correspondant à un couple (s, v_I) tel que $\delta_M(s, v_I) = s$; le nœud cible de chaque arc correspond au couple (s, v'_I) tel que v_I et v'_I satisfont une relation SIC $(v'_I \; R_{Gray} \; v_I)$.

L'affectation des poids associés aux arcs est effectuée de la manière suivante :

– Poids égal à 1 pour les arcs correspondant aux changements de vecteurs d'entrée entre deux vecteurs satisfaisant une relation SIC, depuis une localité stable ;
– Poids égal à 0 pour les arcs correspondant aux changements de localités suite à un changement du vecteur d'entrée.

La figure 4.5 illustre une partie de ce graphe pour l'exemple C présenté figure 4.1. L'exemple suivant explique la construction de ce graphe à partir de la représentation tabulaire du comportement décrit dans le tableau 4.1a.

En utilisant les notations données précédemment (chapitre 3, section 2.3), la formulation mathématique définissant la recherche d'une solution au problème du voyageur de commerce est la suivante :

$$\begin{aligned}
&\textbf{à partir des données du graphe :} \; V, A, C \\
&\quad \textbf{déterminer } \{x_{ij}\} \textbf{ minimisant :} \; \sum_{i<j} c_{ij} \cdot x_{ij} \\
&\quad\quad \textbf{et vérifiant :} \; \begin{cases}
\displaystyle\sum_{i<k} x_{ik} + \sum_{j>k} x_{kj} = 2 \quad (i, j, k \in V) \\
\displaystyle\sum_{i,j \in S, i<j} x_{ij} \leq |S| - 1 \quad (S \subset V, 3 \leq |S| \leq |V|) \\
\begin{cases}
x_{ij} = 1 \textbf{ si l'arc } (i, j, c_{ij}, label_{ij}) \in A \\
\qquad\qquad \text{est dans la solution optimale,} \\
x_{ij} = 0 \textbf{ sinon}
\end{cases}
\end{cases}
\end{aligned} \tag{4.13}$$

4.2 Génération d'une séquence de test SIC

Une séquence de test SIC pour la partie SIC-testable de l'exemple C présenté figure 4.1 est donnée tableau 4.3. Les deux premières lignes (s_s et s_t) du tableau sont mentionnées afin de faciliter la compréhension de cette séquence en regard à la figure 4.2 et au

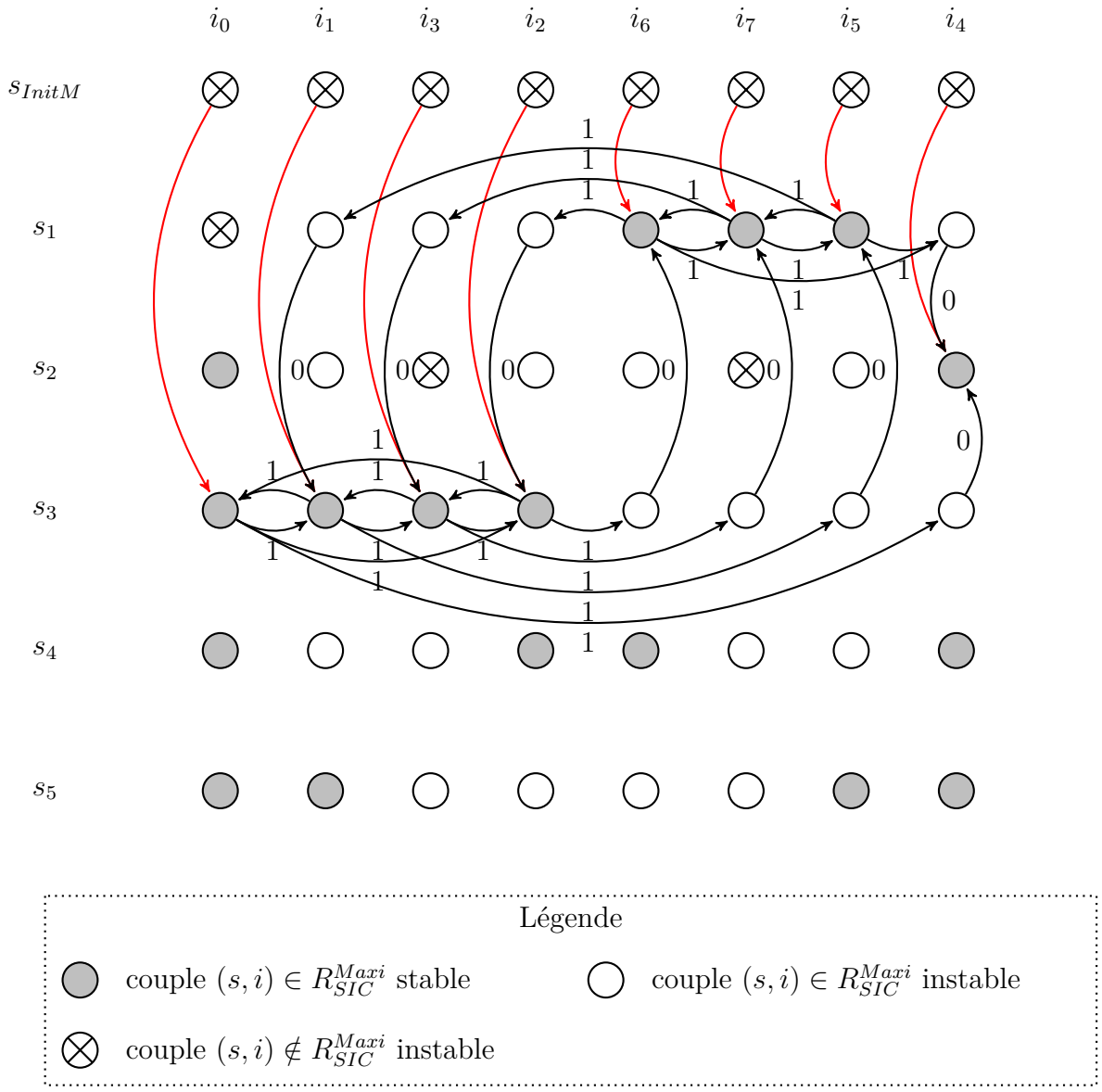


Figure 4.5 – Extrait du graphe utilisé pour le génération de séquence de test SIC

tableau 4.1a. Cette séquence de test contient 45 pas de test et permet de tester les 37 couples (s, v_I) de la partie SIC-testable de la spécification ; certains pas de test permettent de tester les deux couples (s, v_I) et $(\delta_M(s, v_I), v_I)$. Par exemple, le premier pas de test permet de tester à la fois $(s_1, a \cdot \overline{b} \cdot \overline{c})$ et $(s_2, a \cdot \overline{b} \cdot \overline{c})$, et ainsi de suite pour tous les pas de test dont l'état de la source s_s et de la cible s_t sont différents. Pour cet exemple, le calcul de la partie SIC-testable est effectué de manière symbolique et dure moins de 50 ms. La séquence de test donnée dans le tableau 4.3 est obtenue en environ 4 s, ce calcul dure plus longtemps que le calcul d'une séquence MIC de longueur minimale car la taille du graphe manipulé n'est plus la même : le graphe utilisé pour le calcul d'une séquence MIC contient $|S_M|$ nœuds et $|S_M \times I_M|$ arcs, tandis que le graphe utilisé pour le calcul

d'une séquence SIC contient $|S_M \times I_M|$ nœuds et $|S_M \times I_M \times I_G|$ arcs au maximum. La génération de la séquence de test SIC n'est donc pas effectuée par un algorithme mais par une heuristique utilisant la méthode du recuit simulé (Kirkpatrick *et al.* (1983)). La solution obtenue n'est donc pas toujours de longueur minimale, mais l'utilisation d'une heuristique permet d'obtenir une solution approchée en un temps raisonnable. En effet, pour des exemples de plus grande taille, la recherche d'une solution optimale peut conduire à un temps de calcul largement supérieur au temps gagné sur la durée d'exécution du test.

Une version prototype de ces heuristiques a été implantée en langage Python, d'après les sources proposés dans Montgomery (2007). Cette version prototype a été utilisée afin de générer les séquences données en illustration.

pas :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s_s :	s_1	s_2	s_2	s_4	s_4	s_3	s_3	s_2	s_5	s_1	s_1	s_3	s_3	s_2	s_5	s_5	s_1	s_2	s_5	s_5	s_5	s_3	s_1
s_t :	s_2	s_2	s_4	s_4	s_3	s_3	s_2	s_5	s_1	s_1	s_3	s_3	s_2	s_5	s_5	s_1	s_2	s_5	s_5	s_5	s_3	s_1	s_2
a :	1	0	0	0	0	0	1	1	1	1	0	0	1	1	1	1	1	1	1	0	0	1	1
b :	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	0
c :	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0

	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
	s_2	s_4	s_1	s_3	s_3	s_1	s_3	s_3	s_2	s_4	s_4	s_1	s_2	s_2	s_4	s_3	s_1	s_1	s_2	s_2	s_5	s_3
	s_4	s_1	s_3	s_3	s_1	s_3	s_3	s_2	s_4	s_4	s_1	s_2	s_2	s_4	s_3	s_1	s_1	s_2	s_2	s_5	s_3	s_1
	1	1	0	0	1	0	0	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1
	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	1	1
	0	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	1	1	1

Tableau 4.3 – Séquence d'entrée de la séquence de test SIC pour la partie SIC-testable de l'exemple C

5 Construction d'une séquence de test MaxC-SIC

Quand la spécification n'est pas SIC-testable, une séquence de test composée d'un maximum de pas de test consécutifs SIC, appelée séquence de test MaxC-SIC, doit être construite pour limiter le risque de verdicts erronés lors de l'exécution du test. Cette séquence est composée d'une séquence de test SIC de longueur minimale, calculée pour

la partie SIC-testable de la spécification, suivie d'une séquence de test MIC de longueur minimale, calculée pour la partie non-SIC-testable de la spécification. Il s'agit donc d'une combinaison des méthodes présentées dans la section précédente (pour la partie SIC) et dans le chapitre précédent (pour la partie MIC).

Une séquence de test MaxC-SIC pour l'exemple C présenté figure 4.1 est donnée tableau 4.4. Cette séquence de test contient 45 pas de test SIC, permettant de tester les 37 couples (s, v_I) de la partie SIC-testable, et 5 pas de test MIC, permettant de tester les 3 couples (s, v_I) de la partie non-SIC-testable. La comparaison avec le tableau 4.5 montre que l'utilisation d'une séquence de test MaxC-SIC au lieu d'une séquence de test MIC augmente la longueur de la séquence de test (50 pas de test au lieu de 40) mais réduit fortement le nombre de pas de test pour lesquels des changements simultanés de plusieurs entrées sont nécessaires (seulement 3 couples (s, v_I) sont testés dans ces conditions avec la séquence MaxC-SIC). Ainsi, l'exécution du test sur un API durera un peu plus longtemps, mais les résultats contiendront beaucoup moins de verdicts erronés.

pas :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
s_s :	s_1	s_2	s_2	s_4	s_4	s_3	s_3	s_2	s_5	s_1	s_1	s_3	s_3	s_2	s_5	s_5	s_1	s_2	s_5	s_5	s_5	s_3	s_1	s_2	s_4
s_t :	s_2	s_2	s_4	s_4	s_3	s_3	s_2	s_5	s_1	s_1	s_3	s_3	s_2	s_5	s_5	s_1	s_2	s_5	s_5	s_5	s_3	s_1	s_2	s_4	s_1
a :	1	0	0	0	0	0	1	1	1	1	0	0	1	1	1	1	1	1	1	0	0	1	1	1	1
b :	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	0	1	1
c :	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1

	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
	s_1	s_3	s_3	s_1	s_3	s_3	s_2	s_4	s_4	s_1	s_2	s_2	s_4	s_3	s_1	s_1	s_2	s_2	s_5	s_3	s_1	s_3	s_2	s_3	s_2
	s_3	s_3	s_1	s_3	s_3	s_2	s_4	s_4	s_1	s_2	s_2	s_4	s_3	s_1	s_1	s_2	s_2	s_5	s_3	s_1	s_3	s_2	s_3	s_2	s_1
	0	0	1	0	0	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	0	1	0	1	1
	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	1	1	0	0	1	0	1
	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	1	1	1	0	0	1	0	1

Tableau 4.4 – Séquence d'entrée de la séquence MaxC-SIC pour l'exemple C

pas :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
s_s :	s_1	s_2	s_3	s_2	s_2	s_4	s_3	s_2	s_4	s_3	s_2	s_4	s_4	s_4	s_1	s_2	s_4	s_1	s_2	s_5
s_t :	s_2	s_3	s_2	s_2	s_4	s_3	s_2	s_4	s_3	s_2	s_4	s_4	s_4	s_1	s_2	s_4	s_1	s_2	s_5	s_3
a :	1	0	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	1	1	0
b :	0	1	0	0	1	1	0	1	0	0	1	0	0	1	0	1	0	0	0	1
c :	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	1

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
s_3	s_2	s_5	s_3	s_2	s_5	s_5	s_5	s_1	s_2	s_5	s_1	s_3	s_2	s_1	s_3	s_1	s_3	s_1	s_3
s_2	s_5	s_3	s_2	s_5	s_5	s_5	s_1	s_2	s_5	s_1	s_3	s_2	s_1	s_3	s_1	s_3	s_1	s_3	s_1
1	0	0	1	0	1	0	1	1	0	1	0	1	1	0	1	0	1	0	1
0	0	1	0	0	0	0	1	0	0	1	1	0	1	0	1	1	0	0	1
0	1	0	0	1	0	0	1	0	1	0	1	0	1	1	1	0	1	0	0

Tableau 4.5 – Séquence d'entrée de la séquence MIC pour l'exemple C

6 Construction d'une séquence de test min-MIC

La génération automatique d'une séquence de test de longueur minimale contenant un maximum de pas de test SIC, peut également être traitée par la recherche d'une séquence de test de longueur minimale contenant un minimum de pas de test MIC.

Cette section présente la généralisation de la méthode de génération de séquence MaxC-SIC à la génération de séquence min-MIC. Le principe de cette méthode est de permettre la prise en compte de différents paramètres technologiques liés à l'implantation et ainsi de traiter la génération de la séquence de test en une seule phase au lieu de deux pour une séquence MaxC-SIC.

En effet, comme mentionné dans la section 3, pour certaines spécifications, la génération de la séquence de test SIC pour la partie SIC-testable nécessite de ramener l'implantation dans son état initial. Ce retour à l'état initial, peut être effectué de deux manières : soit par la réinitialisation de l'implantation, soit par l'utilisation d'une séquence MIC permettant d'atteindre l'état initial ou un autre état depuis lequel une nouvelle séquence SIC peut être exécutée. La génération automatique d'une séquence MaxC-SIC permet de prendre en compte ces possibilités, en y affectant un poids spécifique.

6.1 Définition du graphe pour le calcul d'une séquence de test min-MIC

La génération d'une séquence min-MIC est effectuée à partir d'un graphe représentant l'ensemble du comportement de la spécification, et non pas seulement le comportement de la partie SIC-testable de la spécification. Dans le graphe utilisé pour représenter le comportement de la partie SIC-testable, seuls les arcs correspondant à des évolutions SIC étaient représentés, et un poids de 1 leur était associé. Le graphe utilisé pour représenter l'ensemble de la spécification reprend ce graphe auquel sont ajoutés les arcs correspondant à des évolutions MIC ou à une réinitialisation manuelle de l'implantation.

Le poids affecté aux arcs correspondant à des évolutions MIC ou à une réinitialisation manuelle de l'implantation est supérieur à 1 afin de privilégier le passage par des arcs correspondant à des évolutions SIC. Dans une première approche, l'affectation des poids associés aux arcs est effectuée de la manière suivante :

- Poids égal à 1 pour les arcs correspondants aux changements de vecteurs d'entrée SIC, depuis une localité stable ;
- Poids égal à 0 pour les arcs correspondants aux changements de localité suite à un changement du vecteur d'entrée ;
- Poids égal à $10^n \times |S_M|$ pour les arcs correspondants aux changements de vecteurs d'entrée faisant varier simultanément n entrées logiques ($n > 1$), depuis une localité stable ;
- Poids égal à $w_{max} \gg 10^{|I_G|} \times |S_M|$ pour les arcs correspondants à une réinitialisation de l'implantation.

La figure 4.6 illustre une partie de ce graphe pour l'exemple C présenté figure 4.1. Pour des raisons de lisibilité seuls les arcs ayant pour origine les nœuds correspondants aux couples (s_1, i_7) et (s_2, i_4) ainsi que les arcs représentant les évolutions vers les couples stables sont représentés ; les arcs entre les nœuds correspondants aux couples (s_1, i_7) et (s_2, i_4) et l'ensemble des nœuds correspondant aux couples $(s_{InitM}, i_{...})$ ne sont pas représentés.

Le calcul de la matrice des plus courts chemins est effectué à partir de ce graphe et permet de supprimer les arcs pour lesquels il existe un chemin entre le nœud de départ et le nœud d'arrivée dont le poids total est plus faible que celui de l'arc correspondant à un changement de vecteurs d'entrée faisant varier simultanément plusieurs entrées

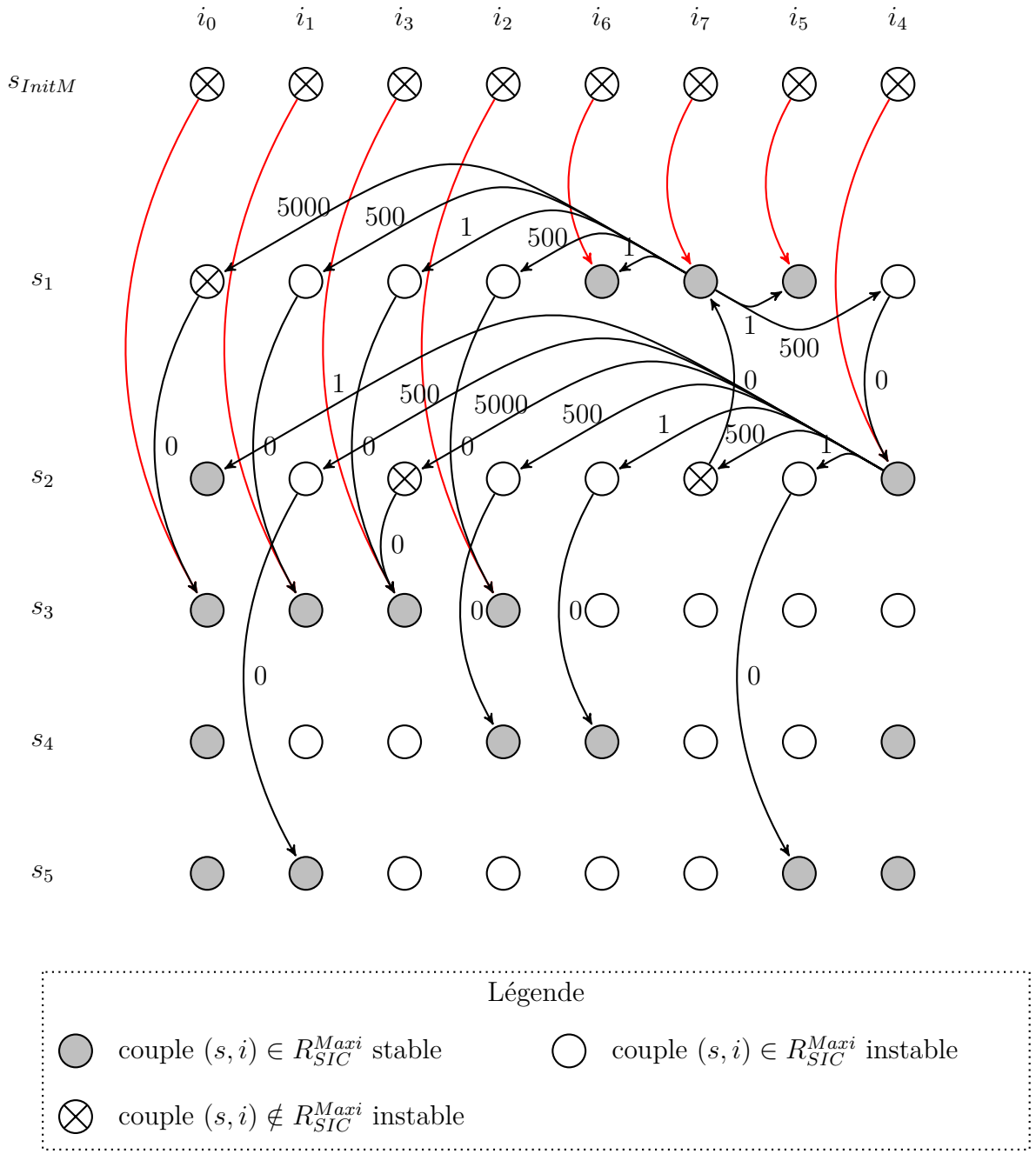


Figure 4.6 – Extrait du graphe utilisé pour le génération de séquence de test min-MIC

logiques. Par exemple, le calcul des plus courts chemins permet de déterminer que le (un des) plus court chemin entre le nœud (s_1, i_7) et le nœud (s_1, i_0) est le chemin $(s_1, i_7) \rightarrow (s_1, i_6) \rightarrow (s_1, i_0)$; le poids total de ce chemin est égal à $1 + 10^2 \times 5$ alors que le poids de l'arc entre le nœud (s_1, i_7) et le nœud (s_1, i_0) est égal à $10^3 \times 5$. Le calcul de la matrice des plus courts chemins permet d'appliquer les algorithmes de parcours de graphes sur une structure de graphe respectant l'inégalité triangulaire.

Le poids affecté à la réinitialisation est le plus important car il nécessite une interven-

tion manuelle lors de l'exécution du test, ce qui nuit à l'exécution automatique du test, en temps masqué. Ce choix est discutable et peut être réduit si la personne en charge du test accepte d'effectuer des réinitialisations de l'implantation plus fréquentes afin de réduire le nombre de pas de test MIC.

La règle d'affectation du poids aux arcs correspondants aux changements simultanés de plusieurs entrées assure que d'un point de vue "minimisation du poids", il est préférable de changer n variables d'entrées séquentiellement que simultanément. La valeur choisie croît rapidement, et permet ainsi lors du calcul d'optimisation, d'écarter rapidement l'utilisation de ces arcs pour la construction du circuit de poids minimal.

D'autres règles d'affectation dépendantes de l'implantation matérielle sont également possible, certaines d'entre elles sont détaillés dans la section 6.2.

Le graphe ainsi défini peut servir de donnée d'entrée au problème du voyageur de commerce. Ainsi, la génération automatique d'une séquence de test min-MIC utilise les mêmes outils que ceux permettant de générer une séquence de test SIC. La taille de ce graphe augmente modérément en nombre de nœuds, puisque le graphe contient tous les couples (s, v_I) alors que le graphe utilisé pour le calcul de la séquence SIC contient uniquement les couples (s, v_I) contenus dans la région SIC-testable. En ce qui concerne les arcs de ce graphe, leur nombre augmente de manière plus importante car $(2^n - 1)$ arcs partent depuis chaque nœud représentant un couple (s, v_I) contre n pour le graphe représentant la région SIC-testable. Cependant, l'augmentation de ce nombre d'arcs n'a d'influence que pour le calcul de la matrice des plus courts chemins, mais pas pour le calcul d'optimisation de la séquence où la donnée de cette dernière matrice suffit.

6.2 Autres règles d'affectation

L'affectation des poids définie dans la section précédente permet d'établir des règles génériques applicables dans la majorité des cas. Cependant, afin d'améliorer la qualité des séquences de test générées et de faciliter la minimisation de la longueur de ces séquences, d'autres règles d'affectation des poids peuvent être définies. Contrairement aux règles préalablement définies, ces règles nécessitent une connaissance experte de l'implantation, et plus particulièrement de l'agencement des entrées logiques sur cette implantation.

Comme le montrent les résultats présentés dans le chapitre précédent, l'agencement des entrées sur les différentes cartes de l'API impacte fortement le taux d'erreurs d'inter-

prétation d'un changement simultané de plusieurs entrées logiques. En effet, les valeurs des taux d'erreurs présentées dans le tableau 3.3 montrent que si deux entrées reliées chacune sur une carte d'entrée différente (principale et secondaire) varient simultanément, le risque d'erreur d'interprétation est beaucoup plus grand que si ces deux entrées sont toutes les deux reliées sur une même carte d'entrée (facteur compris entre 350 et 1700). Les changements multiples impliquant des entrées connectées à des cartes d'entrées différentes doivent donc avoir un effet plus important sur le poids associé à une séquence de test. Par exemple, le poids associé aux arcs correspondant à un changement simultané de plusieurs entrées connectées à des cartes d'entrées différentes peut être égal à $1000 \times 10^n \times |S|$ contre $10^n \times |S|$ pour les arcs correspondants à un changement simultané de plusieurs entrées connectées à une même carte.

De plus, comme le montre le tableau 3.2, la composition du banc de test conduit à un taux d'erreurs d'interprétation plus important lors du changement simultané de plusieurs entrées logiques de 1 à 0 (facteur compris entre 25 et 35). Il est donc préférable de pénaliser davantage les changements simultanés de plusieurs entrées de 1 à 0. Par exemple, le poids associé aux arcs correspondant à un changement simultané de plusieurs entrées de 1 à 0 peut être égal à $50 \times 10^n \times |S|$ contre $10^n \times |S|$ pour les arcs correspondant à un changement simultané de plusieurs entrées de 0 à 1.

Ces dernières règles étant dépendantes de l'implantation et de la composition du banc de test, elles ne peuvent être appliquées de façon générique pour toute implantation. En revanche, elles permettent d'illustrer la difficulté à définir une "bonne" séquence de test. En effet, une "bonne" séquence de test est le résultat d'un compromis entre différents critères faisant intervenir le taux de couverture de cette séquence, la confiance que l'on peut avoir en cette séquence, le temps de calcul pour obtenir cette séquence ainsi que son temps d'exécution, critères dont la prévalence dépendra du point de vue (concepteur, testeur, fiabiliste...).

Synthèse

La première contribution présentée dans ce chapitre est la définition du concept de SIC-testabilité. Ce concept permet de qualifier l'aptitude d'une spécification à permettre la construction d'une séquence de test initialisable, complète et ne contenant que des vecteurs d'entrées logiques adjacents.

La deuxième contribution est le développement d'une méthode permettant la génération automatique d'une séquence de test SIC monolithique permettant de tester sans erreur de verdict la partie SIC-testable de l'implantation.

Enfin, la dernière contribution concerne l'extension de la méthode de génération automatique de séquences de test SIC à la génération de séquences de test min-MIC permettant de tester sans erreur de verdict la partie SIC-testable de l'implantation et de tester la partie non SIC-testable en minimisant les changement simultanées de plusieurs entrées logiques.

Nous rappelons également que les méthodes définies dans ce chapitre sont implantées dans une version prototype du logiciel TELOCO, développé en langage Python.

Conclusions et perspectives

Conclusions

Les différentes contributions apportées durant ces travaux d'études doctorales ont été exposées dans ce mémoire.

La première contribution est la définition d'une *formalisation du comportement d'une spécification Grafcet*. Cette formalisation permet ensuite de transcrire une spécification Grafcet en une machine de Mealy afin d'y appliquer des techniques de génération de séquences de test. La méthode proposée a été automatisée via le développement du logiciel TELOCO, disponible à l'adresse `http://www.lurpa.ens-cachan.fr/isa/teloco`. Ce logiciel permet la construction, depuis une spécification Grafcet, de l'*automate des localités stables* associé à ce Grafcet, de la *machine de Mealy équivalente* ainsi que la *génération automatique d'une séquence de test* permettant de tester le comportement de l'implantation, réalisée par un contrôleur logique, pour toutes les combinaisons d'entrées logiques depuis chaque état de cette implantation défini dans sa spécification. Cette première contribution a pour objectif de rendre accessible la pratique du test de conformité au milieu industriel.

La deuxième contribution a une coloration *fortement expérimentale*. La confrontation des séquences de test obtenues en utilisant les techniques de génération courantes aux verdicts délivrés lors de l'exécution de ces séquences a mis en évidence la présence de verdicts erronés. Une approche expérimentale a donc été mise en œuvre afin d'analyser la source des verdicts erronés et de quantifier le taux d'erreur de ces verdicts. Ces expérimentations ont permis d'illustrer la *perception asynchrone de signaux d'entrées logiques variant théoriquement de manière synchrone*.

Enfin, la dernière contribution propose une solution pour évaluer la capacité d'une implantation à être testée sans erreur de verdict. Les concepts de *SIC-testabilité* et de *taux de couverture SIC* ont été introduits à cet effet. Des techniques de génération de séquences de test basées sur ces critères ont également été proposées. Les méthodes de

vérification de la SIC-testabilité d'une implantation et les techniques de génération de séquences de test sont actuellement implantées dans une version prototype de TELOCO, développée en langage Python.

Perspectives

Afin d'élargir le spectre d'application des méthodes proposées, plusieurs perspectives peuvent être indiquées. Ces perspectives peuvent être classées en quatre grandes catégories :

- levée de l'hypothèse de discernabilité des états de l'implantation par observation des sorties ;
- étude de systèmes de grande taille ;
- étude de l'applicabilité de la méthode de calcul de l'ALS à d'autre langages de spécification ;
- étude de systèmes temporisés.

La première perspective proposée, la levée de l'hypothèse de discernabilité des états de l'implantation par observations des sorties, est une perspective importante pour la généralisation des travaux présentés à l'ensemble des spécifications Grafcet respectant les hypothèses de travail énoncées dans le chapitre 2 (section 3.1, page 55). Le développement de cette perspective nécessite surtout un travail de développement logiciel mais peu d'avancée scientifique. En effet, pour les travaux présentés dans ce mémoire, la méthode TT a été retenue pour sa simplicité de mise en œuvre car, du fait de la discernabilité des états, elle ne nécessite ni séquence de synchronisation, ni séquence de distinction. Si cette hypothèse de discernabilité est levée, la séquence de test doit alors comprendre les séquences de synchronisation et de distinction pour chaque transition de la machine de Mealy testée (chapitre 3, section 1.2, page 89). La construction des séquences de synchronisation est directe par utilisation de la matrice des plus courts chemins ; seule la construction des séquences de distinction nécessite une phase de calcul supplémentaire. Ensuite, la principale difficulté réside dans l'optimisation de la longueur de la séquence de test, qui repose à nouveau sur un problème de la théorie des graphes.

La deuxième perspective, l'étude des systèmes de grande taille, regroupe plusieurs pistes de réflexions visant à permettre la génération de séquences de test pour des systèmes de plus grande taille (plusieurs dizaines d'entrées logiques, et plusieurs centaines

voire quelques milliers de localités). Si la méthode de calcul de l'ALS et de transcription vers la machine de Mealy équivalente restent viables pour des systèmes de cette taille, la principale difficulté réside à nouveau dans la génération de la séquence de test. En effet, le nombre de transitions de la machine de Mealy augmentant proportionnellement au nombre de localités et exponentiellement en fonction du nombre d'entrées logiques de la spécification, la longueur de la séquence de test de longueur minimale croît de la même manière, et le temps de calcul de cette séquence suit une tendance encore plus accentuée. Il n'est donc pas envisageable d'utiliser les techniques proposées, permettant un test complet, pour des systèmes de grande taille. Par conséquent, la taille du comportement à tester de la spécification Grafcet doit être réduite.

Une des solutions permettant de réduire la taille du comportement à tester est d'avoir recours aux objectifs de test. Un objectif de test permet de ne conserver que la partie de la spécification concernée par cet objectif. Une définition formelle d'un objectif de test défini par un Grafcet doit donc être proposée, ainsi qu'un ensemble de "patterns" afin de faciliter la définition de ces objectifs de test par un Grafcet à partir de propriétés données en langage naturel.

Une autre solution pour réduire la taille du comportement à tester est de ne pas tester la totalité des combinaisons d'entrées. En effet, lors du test de toutes les transitions de la machine de Mealy, toutes les combinaisons d'entrées sont testées, mêmes celles mécaniquement impossible du fait de l'implantation matérielle de la partie opérative. Le test ainsi effectué permet donc de tester le comportement du contrôleur pour une partie opérative sans et avec défauts de la partie opérative. Aussi, afin de réduire la taille du comportement à tester, il pourrait être intéressant d'utiliser un modèle de partie opérative sans faute afin de ne conserver que les combinaisons d'entrées possibles sans défaut de la partie opérative. Ensuite, une fois le test de conformité effectué pour le modèle de partie opérative sans faute, celui-ci peut être effectué avec un modèle de partie opérative intégrant certains défauts sélectionnés par une approche experte ou aléatoire.

Ces deux premières perspectives montrent la nécessité de disposer d'outils logiciels performants pour le calcul de parcours dans un graphe. Au vu de la taille des graphes manipulés et des complexités des algorithmes et heuristiques utilisés, l'adage "diviser pour mieux régner" prend ici tout son sens ; car si la recherche de séquences de longueurs minimales permet de réduire la durée d'exécution de ces derniers, il ne faut pas que le temps de calcul de ces séquences devienne prépondérant devant leur temps d'exécution.

La troisième perspective, l'étude de l'applicabilité d'une méthode similaire au calcul de l'ALS à d'autre modèles de spécification, a pour objectif d'étendre la formalisation et les résultats obtenus pour la génération de séquences de test à d'autres langages métier de spécification, tels que les DFL ou les StateCharts UML ou Simulink. L'utilisation de l'ALS comme modèle pivot permet de disposer d'un modèle formel intégrant les comportements intrinsèque et extrinsèque d'une spécification, ainsi que d'une définition du comportement avec recherche de stabilité.

Enfin, la dernière perspective, l'étude de systèmes temporisés, nécessite la définition d'un modèle étendu de l'ALS permettant de prendre en compte les évolutions dues au temps. Cependant, outre cette prise en compte du temps dans le calcul des conditions d'évolutions, la prise en compte du temps lors de la génération et de l'exécution d'une séquence de test est un point plus délicat. En effet, une temporisation peut être déclenchée dans une localité, s'écouler lors du changement de localité active, puis provoquer une évolution vers une nouvelle localité sans variation des entrées. Des critères doivent donc être définis afin de préciser ce qui doit être testé lors du test de conformité temporisé d'un contrôleur logique : la durée d'une temporisation, la précision de la durée d'une temporisation, les sorties émises à tout instant lors de l'écoulement d'une temporisation...

Annexes

Sommaire

Annexe A. Utilisation de macro-étape et d'étape encapsulante

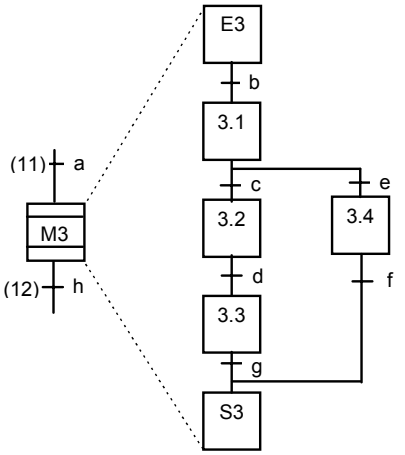


Figure A.1 – Utilisation d'une macro-étape (IEC 60848 (2002))

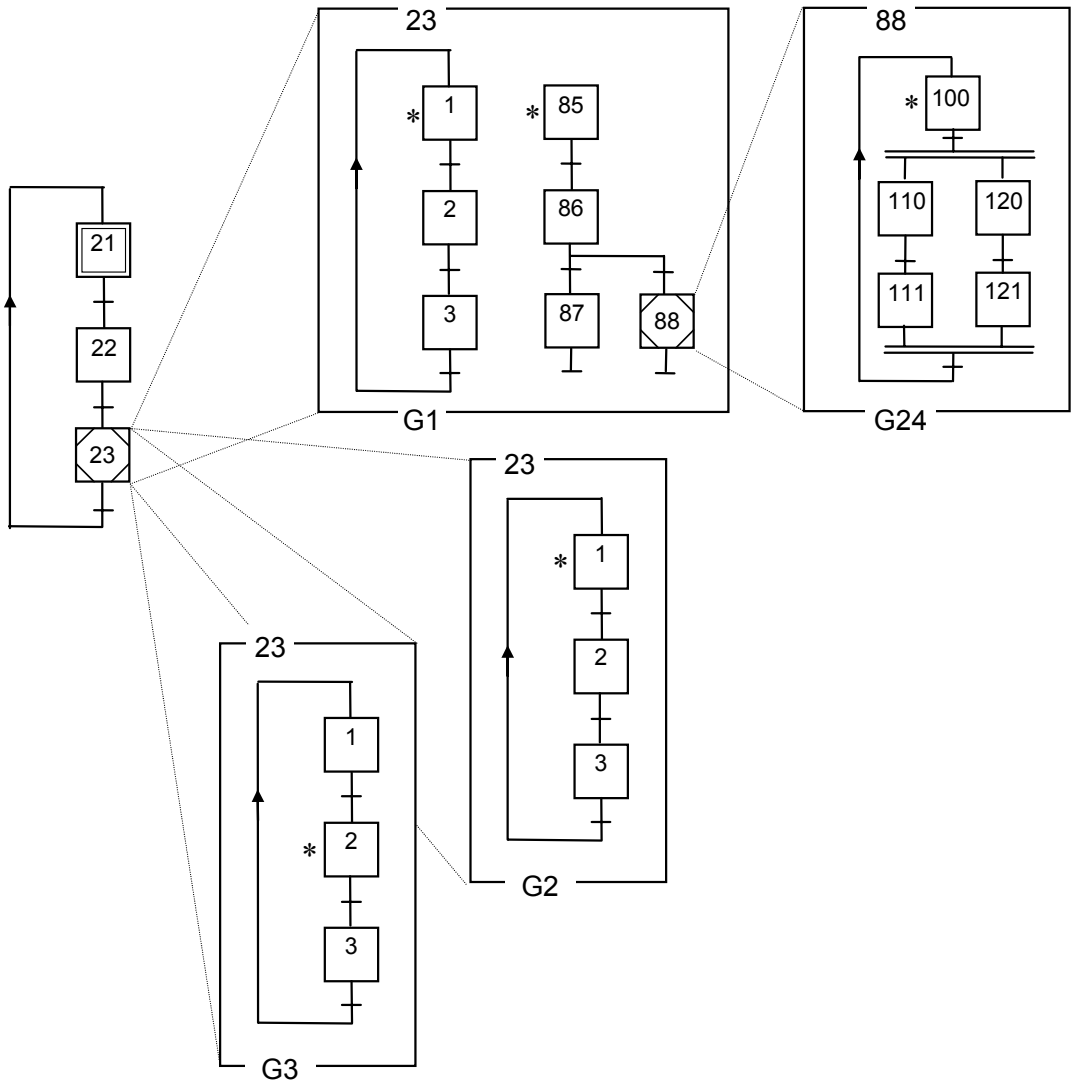


Figure A.2 – Utilisation d'une étape encapsulante (IEC 60848 (2002))

Annexe B. Architecture des APIdS

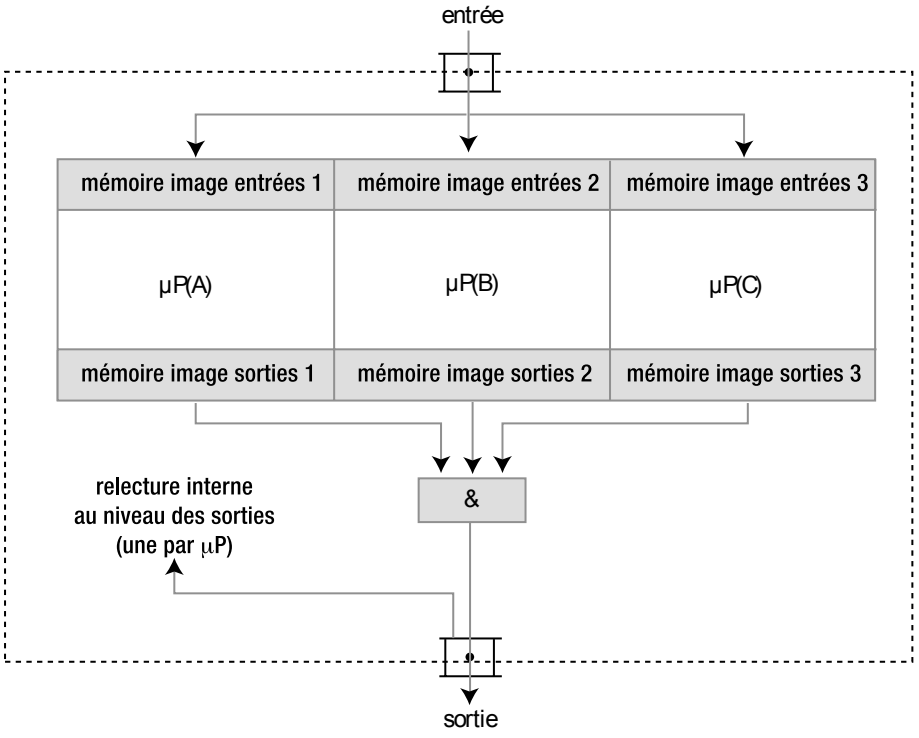


Figure B.1 – Architecture A (INRS (Institut National de Recherche et de Sécurité) (2003))

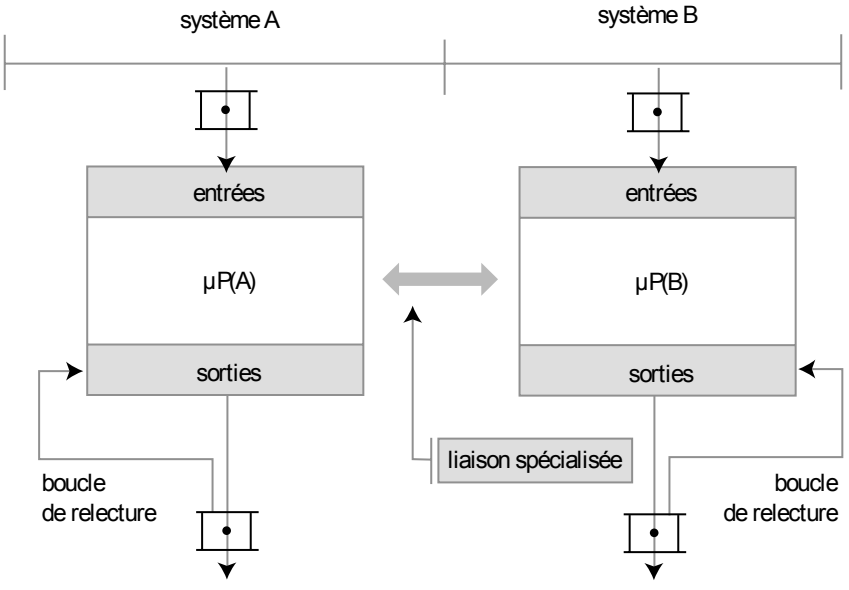


Figure B.2 – Architecture B (INRS (Institut National de Recherche et de Sécurité) (2003))

Références bibliographiques scientifiques

[Arnold et Nivat (1982)] : A. Arnold et M. Nivat. Comportement de processus. Dans *Colloque AFCET "Les Mathématiques de l'Informatique"*, pages 35–68, 1982. *Cité page 33*

[Awedikian (2009)] : R. Awedikian. *Quality of the design of test cases for automotive software : design platform and testing proces*. Thèse de doctorat, Ecole Centrale Paris, 2009. *Cité page 23*

[Barragan Santiago (2007)] : I. Barragan Santiago. *Elaboration de propriétés formelles de contrôleurs logiques à partir d'analyse prévisionnelle par Arbre des Défaillances*. Thèse de doctorat, École Normale Supérieure de Cachan, 2007. URL http://tel.archives-ouvertes.fr/tel-00348404. *Cité page 24*

[Bierel *et al.* (1997)] : E. Bierel, O. Douchin, et P. Lhoste. Grafcet : from theory to implementation. *Journal Européen des Systèmes Automatisés*, volume 31, numéro 3, pages 543–559, (1997). *Cité page 61*

[Blanchard (1979)] : M. Blanchard. *Comprendre maîtriser et appliquer le GRAFCET*. 1979. 174 pages. *Cité page 60*

[Boehm (1979)] : B. Boehm. Software engineering : R&D trends and defense needs. *Research directions in software technology*, (1979). *Cité page 21*

[Brinksma et Tretmans (2000)] : E. Brinksma et J. Tretmans. Testing transition systems : An annotated bibliography. Dans F. Cassez, C. Jard, B. Rozoy, et M. D. Ryan, editeurs, *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000*, volume 2067 de *Lecture Notes in Computer Science*, pages 187–195, 2000. *Cité page 34*

[Brookes *et al.* (1984)] : S. Brookes, C. Hoare, et A. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, volume 31, numéro 3, pages 560–599, (1984). ISSN 0004-5411. *Cité page 33*

[Broy *et al.* (2005)] : M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, et A. Pretschner, editeurs. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of Lecture Notes in Computer Science. Springer, 2005. ISBN 978-3-540-26278-7. DOI 10.1007/b137241. *Cité page 33*

[Cabane (2000)] : R. Cabane. *Théorie des graphes.* Techniques de l'ingénieur, 2000. *Cité page 93*

[Cassez (1997)] : F. Cassez. Formal semantics for reactive GRAFCET. *European Journal of Automation*, volume 31, numéro 3, pages 581–603, (1997). *Cité page 61*

[Chow (1978)] : T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, volume 4, numéro 3, pages 178–187, (1978). ISSN 0098-5589. DOI 10.1109/TSE.1978.231496. *Cité page 90*

[Chériaux *et al.* (2010)] : F. Chériaux, L. Picci, J. Provost, et J.-M. Faure. Conformance test of logic controllers of critical systems from industrial specifications. Dans *Proceedings of ESREL 2010*, pages 1569–1576, 2010. URL http://hal.archives-ouvertes.fr/hal-00483215. *Cité page 87*

[Cohen *et al.* (1996)] : D. M. Cohen, S. R. Dalal, J. Parelius, et G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, volume 13, numéro 5, pages 83–88, (1996). ISSN 0740-7459. DOI 10.1109/52.536462. *Cité page 32*

[Cohen *et al.* (1984)] : G. Cohen, P. Godlewski, et M. Karpovsky. Test exhaustif de circuit combinatoire. *Traitement du signal*, volume 1, pages 223–226, (1984). *Cité page 32*

[Dantzig *et al.* (1954)] : G. Dantzig, R. Fulkerson, et S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, volume 2, numéro 4, pages 393–410, (1954). *Cité page 136*

[Daran (1996)] : M. Daran. *Modélisation des comportements erronés du logiciel et application à la validation des tests par injection de fautes.* Thèse de doctorat, Institut National Polytechnique de Toulouse, 1996. *Cité page 29*

[David (1995)] : R. David. Grafcet : a powerful tool for specification of logic controllers. *IEEE Transaction on Control Systems Technology*, volume 3, numéro 3, pages 253–268, (1995). *Cité page 46*

[David et Alla (1992a)] : R. David et H. Alla. *Petri Nets and Grafcet : Tools for modelling discrete event systems.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992. ISBN 013327537X. *Cité page 47*

[David et Alla (1992b)] : R. David et H. Alla. *Du Grafcet aux réseaux de Petri.* Hermès, 1992. ISBN 2866013255. *Cité page 47*

[David *et al.* (2002)] : R. David, P. Girard, C. Landrault, S. Pravossoudovitch, et A. Virazel. Random adjacent sequences : An efficient solution for logic bist. *SoC Design Methodology*, pages 413–424, (2002). *Cité pages 29 et 32*

[De Smet *et al.* (1999)] : O. De Smet, J.-M. Roussel, et N. Hevin. Identification de machine séquentielle binaire : application à un système réactif. Dans *2ème congrès sur la modélisation des systèmes réactifs, MSR'99*, pages 351–360, 1999. *Cité page 99*

[Denis (1994)] : B. Denis. *Aide à la conception d'architectures de conduite des systèmes de production.* Thèse de doctorat, Université de Nancy 1, 1994. URL `http://tel.archives-ouvertes.fr/tel-00173900`. *Cité pages xi et 18*

[Denis *et al.* (2001)] : B. Denis, O. De Smet, J.-J. Lesage, et J.-M. Roussel. Dispositif et procédé d'analyse de performances et d'identification comportementale d'un système en tant qu'automate à évènements discrets et finis. Dans *French Patent N° 01 110 933*, 2001. *Cité page 99*

[Dorofeeva *et al.* (2005)] : R. Dorofeeva, K. El-Fakih, et N. Yevtushenko. An improved fsm-based conformance testing method. Dans *Proc. of the IFIP 25th International Conference on Formal Methods for Networked and Distributed Systems*, volume 3731 de *Lecture Notes in Computer Science*, pages 204–218, 2005. *Cité page 90*

[Dorofeeva *et al.* (2010)] : R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, et N. Yevtushenko. FSM-based conformance testing methods : A survey annotated with experimental evaluation. *Information and Software Technology*, volume 52, numéro 12, pages 1286 – 1297, (2010). DOI 10.1016/j.infsof.2010.07.001. *Cité pages 88 et 90*

[Edmonds et Johnson (1973)] : J. Edmonds et E. L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, volume 5, pages 88–124, (1973). *Cité pages 90 et 95*

[Faure et Lesage (2001)] : J. Faure et J. Lesage. Methods for safe control systems design and implementation. Dans *Proceedings of 10th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'2001)*, 2001. URL http://hal.archives-ouvertes.fr/hal-00361647. *Cité pages xi et 16*

[Fernandez *et al.* (1997)] : J. Fernandez, C. Jard, T. Jéron, et C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, volume 29, numéro 1-2, pages 123–146, (1997). ISSN 0167-6423. *Cité page 91*

[Floyd (1962)] : R. Floyd. Algorithm 97 : shortest path. *Communications of the ACM*, volume 5, numéro 6, page 345, (1962). ISSN 0001-0782. *Cité page 97*

[Fujiwara *et al.* (1991)] : S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, et A. Ghedamsi. Test selection based on finite state models. Dans *IEEE Transactions on Software Engineering*, volume 17, pages 591–603, 1991. DOI 10.1109/32.87284. *Cité page 90*

[Galiay *et al.* (1980)] : J. Galiay, Y. Crouzet, et M. Vergniault. Physical versus logical fault models MOS LSI circuits : Impact on their testability. *IEEE Transactions on Computers*, volume 100, numéro 6, pages 527–531, (1980). ISSN 0018-9340. *Cité page 28*

[Goodenough et Gerhart (1975)] : J. Goodenough et S. Gerhart. Toward a theory of test data selection. Dans *Proceedings of the International Conference on Reliable software*, volume 10, pages 493–510, 1975. *Cité page 88*

[Gourcuff (2007)] : V. Gourcuff. *Représentations formelles efficaces pour l'aide à la certification de contrôleurs logiques industriels.* Thèse de doctorat, École Normale Su-

périeure de Cachan, 2007. URL http://tel.archives-ouvertes.fr/tel-00202652. *Cité page 24*

[Guéguen et Bouteille (2001)] : H. Guéguen et N. Bouteille. Extensions of Grafcet to structure behavioural specifications. *Control Engineering Practice*, volume 9, numéro 7, pages 743 – 756, (2001). ISSN 0967-0661. DOI 10.1016/S0967-0661(01)00033-8. *Cité page 46*

[Gundlach et Muller-Glaser (1990)] : H. Gundlach et K.-D. Muller-Glaser. On automatic testpoint insertion in sequential circuits. Dans *Proceedings of the International Test Conference 1990*, 1990. *Cité page 31*

[Hayes et Friedman (1974)] : J. Hayes et A. Friedman. Test point placement to simplify fault detection. *IEEE Transactions on Computers*, volume 100, numéro 7, pages 727–735, (1974). ISSN 0018-9340. *Cité page 31*

[Hennie (1964)] : F. C. Hennie. Fault detecting experiments for sequential circuits. Dans *5th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, 1964. *Cité page 90*

[Hietter (2009)] : Y. Hietter. *Synthèse algébrique de lois de commande pour les systèmes à évènements discrets logiques*. Thèse de doctorat, École Normale Supérieure de Cachan, 2009. URL http://tel.archives-ouvertes.fr/tel-00402699. *Cité page 24*

[Hopcroft (1971)] : J. Hopcroft. An N log N algorithm for minimizing states in a finite automaton. Rapport technique, Standford University - Computer Science Department, 1971. *Cité page 37*

[Huffman (1954)] : D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, volume 257, numéro 3, pages 161–190, (1954). ISSN 0016-0032. DOI 10.1016/0016-0032(54)90574-8. *Cité page 123*

[Hulgaard *et al.* (1995)] : H. Hulgaard, S. M. Burns, et G. Borriello. Testing asynchronous circuits - a survey. *Integration, the VLSI Journal*, volume 19, numéro 3, pages 111–131, (1995). ISSN 0167-9260. DOI 10.1016/0167-9260(95)00012-5. *Cité page 28*

[Jard et Jéron (2005)] : C. Jard et T. Jéron. TGV : theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, volume 7, numéro 4, pages 297–315, (2005). ISSN 1433-2779. *Cité page 91*

[Jéron (2004)] : T. Jéron. *Contribution à la génération automatique de tests pour les systèmes réactifs*. Habilitation à diriger des recherches, Université de Rennes 1, France, 2004. *Cité pages xi, 33 et 35*

[Juárez-Orozco (2008)] : Z. Juárez-Orozco. *Vérification de propriétés quantitatives des systèmes logiques par model-checking hybride*. Thèse de doctorat, École Normale Supérieure de Cachan, 2008. URL http://tel.archives-ouvertes.fr/tel-00341969. *Cité page 24*

[Kirkpatrick *et al.* (1983)] : S. Kirkpatrick, J. C. D. Gelatt, et M. P. Vecchi. Optimization by simulated annealing. *Science*, volume 220, numéro 4598, pages 671–680, (1983). ISSN 1095-9203. *Cité page 139*

[Koufareva et Dorofeeva (2002)] : I. Koufareva et R. Dorofeeva. A novel modification of w-method. *Joint bulletin of the Novosibirsk computing center and A.P. Ershov institute of informatics systems*, volume vol 18, series Computer Science, pages 69–81, (2002). *Cité page 90*

[Lee et Yannakakis (1994)] : D. Lee et M. Yannakakis. Testing finite state machines : State identification and verification. *IEEE Transaction on Computers*, volume 43, numéro 3, pages 306–320, (1994). *Cité page 90*

[Lee et Yannakakis (1996)] : D. Lee et M. Yannakakis. Principles and methods of testing finite state machines - a survey. Dans *Proceedings of the IEEE*, volume 84, pages 1090–1123, 1996. *Cité pages xi, 36, 37, 88 et 90*

[Lhoste *et al.* (1993)] : P. Lhoste, H. Panetto, et M. Roesch. Grafcet : from syntax to semantics. *Automatique Productique Informatique Industrielle*, volume 27, numéro 1, pages 127–141, (1993). *Cité page 61*

[Limal (2009)] : S. Limal. *Architectures de contrôle-commande redondantes à base d'Ethernet Industriel : Modélisation et validation par model-checking temporisé*. Thèse de doctorat, École Normale Supérieure de Cachan, 2009. URL http://tel.archives-ouvertes.fr/tel-00468531. *Cité page 24*

[Machado *et al.* (2006)] : J. Machado, B. Denis, J.-J. Lesage, J.-M. Faure, et J. Ferreira Da Silva. Logic controllers dependability verification using a plant model. Dans *3rd IFAC Workshop on Discrete-Event System Design, DES-Des'06*, pages 37–42, 2006. URL http://hal.archives-ouvertes.fr/hal-00361815. *Cité pages 102, 103 et 105*

[Massink *et al.* (2006)] : M. Massink, D. Latella, et S. Gnesi. On testing UML statecharts. *Journal of Logic and Algebraic Programming*, volume 69, numéro 1-2, pages 1–74, (2006). DOI 10.1016/j.jlap.2006.03.001. *Cité page 33*

[McConnell (1997)] : S. McConnell. *Software Project - Survival Guide*. Microsoft Press, 1997. *Cité pages xi et 17*

[Mei-Ko (1962)] : K. Mei-Ko. Graphic programming using odd or even points. *Chinese Mathematics*, volume 1, pages 273–277, (1962). *Cité pages 90 et 94*

[Merle (2010)] : G. Merle. *Algebraic modelling of Dynamic Fault Trees, contribution to qualitative and quantitative analysis*. Thèse de doctorat, École Normale Supérieure de Cachan, 2010. URL http://tel.archives-ouvertes.fr/tel-00502012. *Cité page 24*

[Moalla (1981)] : M. Moalla. *Spécification et conception sûre d'automatismes discrets complexes, basées sur l'utilisation du GRAFCET et des réseaux de PETRI*. Thèse de doctorat, Université Scientifique et Médicale, 1981. *Cité page 47*

[Naito et Tsunoyama (1981)] : S. Naito et M. Tsunoyama. Fault detection for sequential machines by transitions tours. Dans *Proceedings of the IEEE Fault Tolerant Computer Symposium*, pages 238–243, 1981. *Cité pages 90 et 91*

[Papadimitriou (1976)] : C. Papadimitriou. On the complexity of edge traversing. *Journal of the ACM*, volume 23, pages 544–554, (1976). *Cité page 95*

[Patterson, Jr. (2009)] : F. G. Patterson, Jr.. Systems engineering life cycles : Life cycles for research, development, test, and evaluation ; acquisition ; and planning and marketing. *Handbook of systems engineering and management*, page 65, (2009). *Cité pages xi, 16, 18 et 22*

[Petrenko (2000)] : A. Petrenko. Fault model-driven test derivation from finite state models : Annotated bibliography. Dans F. Cassez, C. Jard, B. Rozoy, et M. D. Ryan, editeurs, *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000*, volume 2067 de *Lecture Notes in Computer Science*, pages 187–195, 2000. *Cité page 36*

[Petrenko et Yevtushenko (2005)] : A. Petrenko et N. Yevtushenko. Testing from partial deterministic FSM specification. *IEEE Trans. Comput.*, volume 54, numéro 9, pages 1154–1165, (2005). *Cité page 90*

[Provost *et al.* (2009a)] : J. Provost, J.-M. Roussel, et J.-M. Faure. Test sequence construction from SFC specification. Dans *Proceedings of 2nd IFAC Workshop on Dependable Control of Discrete Systems (DCDS'09)*, pages 341–346, 2009. URL http://hal.archives-ouvertes.fr/hal-00394454. *Cité pages 43 et 87*

[Provost *et al.* (2009b)] : J. Provost, J.-M. Roussel, et J.-M. Faure. Construction d'une séquence de test minimale à partir d'une spécification grafcet. Dans *Actes des 3èmes Journées Doctorales / Journées Nationales MACS (JD-JN-MACS'09)*, 2009. URL http://hal.archives-ouvertes.fr/hal-00369851. *Cité page 43*

[Provost *et al.* (2009c)] : J. Provost, J.-M. Roussel, et J.-M. Faure. Test exhaustif de contrôleurs logiques spécifiés en grafcet : apports et limites d'une modélisation par machines de mealy. Dans *7ième colloque francophone sur la Modélisation des Systèmes Réactifs (MSR'09)*, pages 889–904, 2009. URL http://hal.archives-ouvertes.fr/hal-00440343. *Cité page 87*

[Provost *et al.* (2010a)] : J. Provost, J.-M. Roussel, et J.-M. Faure. Un démonstrateur pour le test de conformité de contrôleurs logiques. Dans *3èmes Journées Démonstrateurs*, 2010. URL http://hal.archives-ouvertes.fr/hal-00585235. *Cité page 87*

[Provost *et al.* (2010b)] : J. Provost, J.-M. Roussel, et J.-M. Faure. SIC-testability of sequential logic controllers. Dans *Proceedings of 10th International Workshop on Discrete Event Systems (WODES 2010)*, pages 203–208, 2010. URL http://hal.archives-ouvertes.fr/hal-00512767. *Cité page 122*

[Provost *et al.* (2011a)] : J. Provost, J.-M. Roussel, et J.-M. Faure. A formal semantics for grafcet specifications. Dans *Proceedings of the IEEE 7th International*

Conference on Automation Science and Engineering (CASE 2011), 2011. URL http://hal.archives-ouvertes.fr/hal-00603189. *Cité page 43*

[Provost *et al.* (2011b)] : J. Provost, J.-M. Roussel, et J.-M. Faure. Translating grafcet specifications into mealy machines for conformance test purposes. *Control Engineering Practice*, volume 19, numéro 9, pages 947–957, (2011). DOI 10.1016/j.conengprac.2010.10.001. URL http://hal.archives-ouvertes.fr/hal-00547891. *Cité page 43*

[Provost *et al.* (2011c)] : J. Provost, J.-M. Roussel, et J.-M. Faure. Testing programmable logic controllers from finite state machines specification. Dans *Proceedings of the 3rd IFAC Workshop on Dependable Control of Discrete Systems (DCDS'11)*, 2011. URL http://hal.archives-ouvertes.fr/hal-00585242. *Cité pages 87 et 122*

[Rossi (2003)] : O. Rossi. *Validation formelle de programmes Ladder Diagram pour Automates Programmables Industriels*. Thèse de doctorat, École Normale Supérieure de Cachan, 2003. *Cité page 24*

[Roussel et Denis (2002)] : J. Roussel et B. Denis. Safety properties verification of ladder diagram programs. *Journal Européen des systemes automatisés*, volume 36, numéro 7, pages 905–917, (2002). URL http://hal.archives-ouvertes.fr/hal-00356881. *Cité pages xi et 21*

[Roussel *et al.* (2004)] : J. Roussel, J. Faure, J. Lesage, et A. Medina. Algebraic approach for dependable logic control systems design. *International journal of production research*, volume 42, numéro 14, pages 2859–2876, (2004). ISSN 0020-7543. URL http://hal.archives-ouvertes.fr/hal-00344923. *Cité page 102*

[Roussel (1994)] : J.-M. Roussel. *Analyse De Grafcets Par Génération Logique De l'Automate Equivalent*. Thèse de doctorat, Ecole Normale Supérieure de Cachan, France, 1994. URL http://tel.archives-ouvertes.fr/tel-00340842. *Cité page 60*

[Roussel et Faure (2003)] : J.-M. Roussel et J.-M. Faure. Étude de la démarche, des méthodes et outils pour réaliser le portage d'un automatisme à relais de contrale nucléaire vers un automate programmable industriel : Tests de comportements combinatoires, séquentiels et temporisés. Rapport n°3 du contrat edf r&d - lurpa n°p11/f01377/0, LURPA, 2003. *Cité page 107*

[Roussel et Faure (2006)] : J.-M. Roussel et J.-M. Faure. Designing dependable controllers using algebraic specifications. *Control Engineering Practice*, volume 14, numéro 10, pages 1143–1155, (2006). URL http://hal.archives-ouvertes.fr/hal-00340844. *Cité page 56*

[Ruel (2009)] : S. Ruel. *Evaluation des bornes des performances temporelles des Architectures d'Automatisation en Réseau par preuves itératives de propriétés logiques.* Thèse de doctorat, École Normale Supérieure de Cachan, 2009. URL http://tel.archives-ouvertes.fr/tel-00405783. *Cité page 24*

[Sabnani et Dahbura (1988)] : K. K. Sabnani et A. T. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, volume 15, numéro 4, pages 285–297, (1988). *Cité page 90*

[Schotten et Meyr (1995)] : C. Schotten et H. Meyr. Test point insertion for an area efficient BIST. Dans *Proceedings of 1995 IEEE International Test Conference (ITC)*, pages 515–523, 1995. ISBN 0780329929. *Cité page 31*

[Thimbleby (2003)] : H. Thimbleby. The directed Chinese postman problem. *Software – Practice & Experience*, volume 33, numéro 11, pages 1081–1096, (2003). DOI 10.1002/spe.540. *Cité pages 90, 95 et 98*

[Tretmans (1996)] : J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software — Concepts and Tools*, volume 17, numéro 3, pages 103–120, (1996). *Cité pages 33 et 35*

[Tretmans (2008)] : J. Tretmans. Model based testing with labelled transition systems. Dans R. M. Hierons, J. P. Bowen, et M. Harman, editeurs, *Formal Methods and Testing*, volume 4949 de *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. ISBN 978-3-540-78916-1. DOI 10.1007/978-3-540-78917-8_1. *Cité page 35*

[Tretmans (2010)] : J. Tretmans. Model-based testing with labelled transition systems. 6th TAROT Summer School on Software Testing, 2010. *Cité pages xi et 25*

[Virazel *et al.* (2001)] : A. Virazel, R. David, P. Girard, C. Landrault, et S. Pravossoudovitch. Delay fault testing : Choosing between random sic and random mic test sequences. *Journal of Electronic Testing : Theory and Applications*, volume 17, numéro 3-4, pages 233–241, (2001). ISSN 0923-8174. *Cité page 126*

[von Bochmann et Jourdan (2009)] : G. von Bochmann et G.-V. Jourdan. Testing k-safe Petri nets. Dans M. Núñez, P. Baker, et M. G. Merayo, editeurs, *TestCom/FATES - Testing of Software and Communication Systems*, volume 5826 of Lecture Notes in Computer Science, pages 33–48. Springer, 2009. DOI 10.1007/978-3-642-05031-2_3. *Cité page 33*

[Vuong *et al.* (1989)] : S. T. Vuong, W. W. L. Chan, et M. R. Ito. The uiov-method for protocol test sequence. Dans *Proc. of the IFIP TC6 2nd IWPTS*, pages 161–175, 1989. *Cité page 90*

[Warshall (1962)] : S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, volume 9, numéro 1, pages 11–12, (1962). ISSN 0004-5411. *Cité page 97*

[Williams et Angell (1973)] : M. Williams et J. Angell. Enhancing testability of large-scale integrated circuits via test points and additional logic. *IEEE Transactions on Computers*, volume 100, numéro 1, pages 46–60, (1973). ISSN 0018-9340. *Cité page 31*

[Yevtushenko et Petrenko (1990)] : N. Yevtushenko et A. Petrenko. Test derivation method for an arbitrary deterministic automaton. *Automatic Control and Computer Sciences*, volume 5, (1990). *Cité page 90*

[Yi *et al.* (2008)] : W. Yi, F. Xing-hua, et W. Dai-qiang. An implementation of random single input change technique for low-power test. Dans *Proceeding of the 2nd International Conference on Anti-counterfeiting, Security and Identification*, pages 352–355, 2008. *Cité page 126*

[Zaytoon (1996)] : J. Zaytoon. Specification and design of logic controllers for automated manufacturing systems. *Robotics and computer-integrated manufacturing*, volume 12, numéro 4, pages 353–366, (1996). *Cité page 17*

Références bibliographiques techniques

[Autorité de Sûreté Nucléaire (2000)] : Autorité de Sûreté Nucléaire. *Règle Fondamentale de Sûreté II.4.1.a relative aux logiciels des systèmes électriques classés de sûreté*. 2000. URL http://www.asn.fr/index.php/content/download/25211/150867/file/RFS-II-4-1.pdf. *Cité page 24*

[Cnomo (2003)] : Cnomo. *E03.65.036.G Règles de conception et de réalisation des logiciels d'automatismes*. Comité de normalisation des moyens de production, 2003. *Cité page 19*

[IEC 60848 (1988)] : IEC 60848. *Preparation of function charts for control systems*. International Electrotechnical Commission, 1 édition, 1988. Annulée (2002-02-01). *Cité page 46*

[IEC 60848 (2002)] : IEC 60848. *GRAFCET specification language for sequential function charts*. International Electrotechnical Commission, 2 édition, 2002. *Cité pages iii, v, xi, xii, xiii, 19, 39, 46, 51, 52, 53, 56, 152, 153 et 170*

[IEC 61131-1 (2003)] : IEC 61131-1. *Programmable controllers - Part 1 : General information*. International Electrotechnical Commission, 2 édition, 2003. *Cité page 10*

[IEC 61131-2 (2007)] : IEC 61131-2. *Programmable controllers - Part 2 : Equipment requirements and tests*. International Electrotechnical Commission, 3 édition, 2007. *Cité pages xi, 10 et 11*

[IEC 61131-3 (2003)] : IEC 61131-3. *Programmable controllers - Part 3 : Programming languages*. International Electrotechnical Commission, 2 édition, 2003. *Cité pages 10, 19, 53, 56 et 101*

[IEC 61131-4 (2004)] : IEC 61131-4. *Programmable controllers - Part 4 : User guidelines*. International Electrotechnical Commission, 2 édition, 2004. *Cité page 10*

[IEC 61131-5 (2000)] : IEC 61131-5. *Programmable controllers - Part 5 : Communications*. International Electrotechnical Commission, 1 édition, 2000. *Cité page 10*

[IEC 61131-6 (2010)] : IEC 61131-6. *Programmable controllers - Part 6 : Part 6 : Functional safety (IEC 65B/742/CD :2010)*. International Electrotechnical Commission, committee draft édition, 2010. *Cité page 10*

[IEC 61131-7 (2000)] : IEC 61131-7. *Programmable controllers - Part 7 : Fuzzy control programming*. International Electrotechnical Commission, 1 édition, 2000. *Cité page 10*

[IEC 61131-8 (2000)] : IEC 61131-8. *Programmable controllers - Part 8 : Guidelines for the application and implementation of programming languages*. International Electrotechnical Commission, 1 édition, 2000. *Cité page 10*

[IEEE 1012 (2004)] : IEEE 1012. *Standard for Software Verification and Validation*. IEEE Computer Society, 2004. *Cité page 20*

[INRS (Institut National de Recherche et de Sécurité) (2003)] : INRS (Institut National de Recherche et de Sécurité). *Cablage des entrées et des sorties des automates programmables dédiés à la sécurité*. 2003. *Cité pages xiii et 154*

[Joint Test Action Group (JTAG) (2010)] : Joint Test Action Group (JTAG). Design for testability guidelines. Rapport technique, http ://www.xjtag.com/support-jtag/dft-guidelines.php, 2010. Version 3.2. *Cité page 100*

[Montgomery (2007)] : J. Montgomery. Tackling the travelling salesman problem : prologue, 2007. URL http://www.psychicorigami.com/2007/04/10/tackling-the-travelling-salesman-problem-prologue. Visité le 05/05/2011. *Cité page 139*

[NF C03-190 (1982)] : NF C03-190. *Schémas, diagrammes, tableaux : diagramme fonctionnel GRAFCET pour la description des systèmes logiques de commande - Diagrams, charts, tables. Control system function chart GRAFCET*. AFNOR, 1982. Annulée (1995-09-20). *Cité page 46*

[NF EN 60880 (2010)] : NF EN 60880. *Centrales nucléaires de puissance - Instrumentation et contrôle-commande importants pour la sûreté - Aspects logiciels des*

systèmes programmés réalisant des fonctions de catégorie A. AFNOR, 2 édition, 2010.

Cité page 25

Résumé : Les travaux présentés dans ce mémoire de thèse s'intéressent à la génération et à la mise en œuvre de séquences de test pour le test de conformité de contrôleurs logiques. Dans le cadre de ces travaux, le Grafcet (IEC 60848 (2002)), langage de spécification graphique utilisé dans un contexte industriel, a été retenu comme modèle de spécification. Les contrôleurs logiques principalement considérés dans ces travaux sont les automates programmables industriels (API). Afin de valider la mise en œuvre du test de conformité pour des systèmes de contrôle/commande critiques, les travaux présentés proposent :

- Une formalisation du langage de spécification Grafcet. En effet, l'application des méthodes usuelles de vérification et de validation nécessitent la connaissance du comportement à partir de modèles formels. Cependant, dans un contexte industriel, les modèles utilisés pour la description des spécifications fonctionnelles sont choisis en fonction de leur pouvoir d'expression et de leur facilité d'utilisation, mais ne disposent que rarement d'une sémantique formelle.
- Une étude de la mise en œuvre de séquences de test et l'analyse des verdicts obtenus lors du changement simultané de plusieurs entrées logiques. Une campagne d'expérimentation a permis de quantifier, pour différentes configurations de l'implantation, le taux de verdicts erronés dus à ces changements simultanés.
- Une définition du critère de SIC-testabilité d'une implantation. Ce critère, déterminé à partir de la spécification Grafcet, définit l'aptitude d'une implantation à être testée sans erreur de verdict. La génération automatique de séquences de test minimisant le risque de verdict erroné est ensuite étudiée.

Mots-clés : Test de conformité, Grafcet, Contrôleurs logiques, Automate Programmable Industriel, Transformation de modèle, SIC-testabilité

Abstract : The works presented in this PhD thesis deal with the generation and implementation of test sequences for conformance test of logic controllers. Within these works, Grafcet (IEC 60848 (2002)), graphical specification language used in industry, has been selected as the specification model. Logic controllers mainly considered in these works are Programmable Logic Controllers (PLC). In order to validate the carrying out of conformance test of critical control systems, this thesis presents :

- A formalization of the Grafcet specification language. Indeed, to apply usual verification and validation methods, the behavior is required to be expressed through formal models. However, in industry, the models used to describe functional specifications are chosen for their expression power and usability, but these models rarely have a formal semantics.
- A study of test sequences execution and analysis of obtained verdicts when several logical inputs are changed simultaneously. Series of experimentation have permitted to quantify, for different configurations of the implantation under test, the rate of erroneous verdicts due to these simultaneous changes.
- A definition of the SIC-testability criterion for an implantation. This criterion, determined on the Grafect specification defines the ability of an implementation to be tested without any erroneous verdict. Automatic generation of test sequences that minimize the risk of erroneous verdict is then studied.

Keywords : Conformance test, Grafcet, Logic controllers, Programmable Logic Controller, Model transformation, SIC-testability