Représentations formelles efficaces pour l'aide à la certification de contrôleurs logiques industriels

Vincent Gourcuff

N° ENSC-2007/xx

THESE DE DOCTORAT
DE L'ECOLE NORMALE SUPERIEURE DE CACHAN

Présentée par

Monsieur Vincent GOURCUFF

pour obtenir le grade de

DOCTEUR DE L'ECOLE NORMALE SUPERIEURE DE CACHAN

Domaine :
ELECTRONIQUE–ELECTROTECHNIQUE–AUTOMATIQUE

Sujet de la thèse :

Représentations formelles efficaces pour l'aide à la certification de contrôleurs logiques industriels

Document provisoire envoyé aux rapporteurs le 7 novembre 2007.
Soutenance de thèse prévue le 17 decembre 2007 à Cachan devant le jury composé de :

Alessandro GIUA	Professeur-Université de Cagliari (Italie)	Rapporteur
Jean-Pierre ELLOY	Professeur-Ecole Centrale de Nantes-IRCCyN	Rapporteur
Jean-Louis BOIMOND	Professeur-Université d'Angers-ISTIA	Examinateur
Olivier de SMET	Maitre de conférence-CNAM de Paris-LURPA	Encadrant
Jean-Marc FAURE	Professeur-SUPMECA Paris-LURPA	Directeur de thèse
Hervé SABOT	Responsable du service Engineering & Commissioning tool - Alstom Power	Invité

Laboratoire Universitaire de Recherche en Production Automatisée
(ENS CACHAN/EA 1385)
61, avenue du Président Wilson, 94235 CACHAN CEDEX (France)

Table des matières

Introduction

La complexité croissante des systèmes automatisés ainsi que la demande croissante, de la part de la société, pour des systèmes technologiques de plus en plus disponibles et qui ne soient pas sources potentielles de dangers, font que la **sûreté de fonctionnement des systèmes de contrôle-commande** est une préoccupation de plus en plus forte dans le monde industriel.

Dans cette mouvance, la norme CEI 61508 propose un ensemble de méthodes pour la conception sûre des systèmes électriques/électroniques/électroniques programmables assurant des fonctions de sécurité, ce qui englobe nombre de systèmes de contrôle-commande. Comme indiqué dans son intitulé, les méthodes préconisées par cette norme sont relatives tant aux composants matériels qu'aux logiciels. Lorsqu'un système a été développé selon les préconisations de cette norme, il peut être alors **certifié**, c'est-à-dire qu'un organisme habilité (le bureau Veritas ou le Technischer Überwachungs-Verein (TÜV) par exemple) peut garantir, après analyse des procédures de développement, le respect de ces préconisations. Il importe de souligner que les concepteurs de systèmes critiques portent un intérêt tout particulier à la certification, qui constitue bien souvent une obligation contractuelle à laquelle on ne peut déroger.

Cet intérêt est bien entendu partagé par l'entreprise Alstom Power. Le département Control Systems Engineering Tools de cette entreprise propose en effet un outil de développement de systèmes de contrôle-commande nommé Controcad, qui permet de concevoir des systèmes à base de contrôleurs logiques industriels, programmés par exemple dans un des langages de la norme CEI 61131-3. Afin d'aider les utilisateurs de cet outil à faire certifier les systèmes de contrôle-commande critiques qu'ils développent (systèmes de contrôle-commande pour la production d'énergie électrique), cette entreprise a eu la volonté d'inclure dans son outil certaines des méthodes préconisées par la norme CEI 61508.

Contrairement aux composants matériels dont les fautes sont **aléatoires**, un logiciel de contrôle-commande ne peut générer que des fautes **systématiques** ; la faute d'un composant logiciel provient en effet d'une mauvaise interprétation des spécifications ou d'une erreur de conception ou de codage, et se produit chaque fois que ses conditions d'apparition sont réunies. Afin d'éliminer ces fautes systématiques, la norme CEI 61508 préconise l'emploi de méthodes formelles de vérification. L'intégration de ces méthodes au sein d'un logiciel de développement de systèmes de contrôle-commande en facilitera donc la certification.

Les **méthodes formelles de vérification** ont fait l'objet de très nombreux travaux

scientifiques. Cependant, leur utilisation en milieu industriel, et en particulier dans le domaine de la conception des systèmes de contrôle-commande critiques, reste très marginale. Ceci est dû à plusieurs problèmes : les langages formels devant être mis en œuvre (langages de description de systèmes de transitions, logiques temporelles, ...) sont délicats à maîtriser, les résultats fournis en cas de preuve négative sont difficiles à interpréter, et, par-dessus tout, ces méthodes d'analyse exhaustive sont sujettes à des problèmes d'explosion combinatoire, même pour des systèmes de taille relativement modeste.

L'objectif de cette thèse est de contribuer à la résolution de ce dernier problème. Plus précisément, nous souhaitons proposer des représentations formelles des contrôleurs logiques industriels qui permettent le passage à l'échelle des méthodes de vérification. Ces représentations seront qualifiées d'efficaces dans la mesure où elles devront permettre à l'ingénieur automaticien d'accéder au monde de la preuve formelle tout en respectant des contraintes de temps de calcul et d'espace mémoire compatibles avec le processus de développement industriel des systèmes de contrôle-commande.

Ce mémoire de thèse est découpé en 6 chapitres qui nous permettent d'exposer successivement les problématiques industrielle et scientifique de nos travaux, nos contributions théoriques ainsi que l'utilisation de ces contributions dans le contexte industriel de ces travaux.

Plus précisément, le premier chapitre présente tout d'abord le contexte industriel de nos travaux. Cette présentation inclut une description des éléments de la norme CEI 61508 qui sont pertinents lorsque les logiciels de commande sont considérés, ainsi qu'une brève présentation du logiciel Controcad et des contrôleurs logiques qui sont étudiés dans ce mémoire. Le contexte étant posé, il est possible d'exprimer la problématique industrielle, puis les objectifs scientifiques qui en découlent.

Une analyse bibliographique sur la vérification formelle des contrôleurs logiques est contenue dans le deuxième chapitre. Après avoir rappelé les principes des méthodes de vérification formelle, et plus particulièrement des techniques de model-checking, une synthèse des travaux visant à l'utilisation de ces techniques pour l'analyse des contrôleurs logiques est effectuée. Ce chapitre s'intéresse également aux mécanismes d'abstraction qui ont été proposés pour améliorer les possibilités de passage à l'échelle de ces techniques, en dégageant leurs avantages et faiblesses.

Le troisième chapitre nous permet de présenter notre première contribution : une représentation formelle des contrôleurs logiques qui vise à améliorer les possibilités des techniques de model-checking non temporisé, lorsque la preuve de propriétés extrinsèques est recherchée. Cette représentation est basée sur deux abstractions originales : une abstraction d'interprétation, qui permet de ne garder que les états pertinents pour ce type de preuve, et une abstraction de données, qui permet de limiter le nombre de variables codant chaque état du modèle formel.

L'efficacité de ces deux abstractions et, par voie de conséquence, de la représentation proposée, est validée dans le quatrième chapitre. Après une discussion sur les critères d'évaluation de l'efficacité d'un modèle formel, la représentation proposée est comparée à

des propositions antérieures, sur la base de trois études de cas, ce qui permet de quantifier son efficacité.

Le cinquième chapitre propose une deuxième contribution : une représentation formelle des blocs fonctionnels, élément de langage largement utilisé pour encapsuler de la connaissance dans les contrôleurs logiques en langages CEI 61131-3. Cette représentation a un double but : fournir au concepteur de systèmes de contrôle-commande un moyen de définir formellement de nouveaux blocs fonctionnels et d'utiliser cette définition formelle dans le processus de vérification.

Le sixième chapitre traite de l'utilisation de nos résultats théoriques en contexte industriel. Après avoir présenté le mode d'intégration de ces résultats dans l'outil Controcad, les différents types de vérification qu'il est possible d'effectuer sont successivement décrits.

Finalement, après avoir rappelé de manière synthétique les résultats obtenus, nous concluons en proposant quelques pistes pour des travaux futurs prolongeant ce travail de thèse.

Chapitre 1

Contexte, problématique industrielle et objectifs scientifiques

Les travaux présentés dans ce mémoire ont été réalisés dans le cadre d'un projet de recherche financé par Alstom Power Control Systems, Engineering Tools Department. Cette entreprise a pour volonté de proposer à ses clients des systèmes de production d'énergie sûrs et fiables répondant aux préconisations de la norme CEI 61508.
Ce chapitre présente donc en premier lieu cette norme et ses préconisations. Le contexte industriel spécifique des travaux est ensuite détaillé afin de définir la problématique applicative. Afin d'apporter des solutions formelles aux problèmes industriels à résoudre, les objectifs scientifiques sont identifiés dans la dernière section.

Sommaire

1.1 Norme CEI 61508

La norme CEI 61508 [IEC00] présente un ensemble de méthodes pour la conception sûre des dispositifs Electrique/Electronique/Programmable Electronique (E/E/PE) assurant des fonctions relatives à la sécurité. Son objectif est la classification des systèmes critiques en niveaux, nommés niveaux d'intégrité de sécurité (Safety Integrated Level : SIL), au travers de la maîtrise des risques, du développement et des protections.

La norme CEI 61508 est composée de 7 parties :

1. Prescriptions générales
2. Prescriptions pour les systèmes électriques / électroniques / électroniques programmables relatifs à la sécurité
3. Prescriptions concernant les logiciels
4. Définitions et abréviations
5. Exemples de méthodes de détermination des niveaux d'intégrité de sécurité
6. Lignes directrices pour l'application de la CEI 61508-2 et de la CEI 61508-3
7. Présentation de techniques et mesures

Cette section introduit le concept de la sécurité fonctionnelle, mis en avant par cette norme, puis s'intéresse plus particulièrement à la sécurité fonctionnelle des logiciels relatifs à la sécurité, soit la troisième partie de la norme.

1.1.1 Sécurité fonctionnelle

La sécurité fonctionnelle correspond à l'absence de risques inacceptables, de blessures ou d'atteintes à la santé des personnes, directement ou indirectement, résultant d'un dommage au matériel ou à l'environnement. La sécurité fonctionnelle est le sous-ensemble de la sécurité globale qui correspond au bon fonctionnement d'un système ou d'un équipement en réponse à ses entrées. Un système est sûr fonctionnellement s'il répond de manière sûr à l'excitation de ses entrées.

Les risques significatifs pour les équipements et les éventuels systèmes de contrôle associés doivent être identifiés par le spécificateur ou le développeur au travers d'une analyse de risques. Cette analyse détermine si la sécurité fonctionnelle est nécessaire pour assurer une protection adéquate contre chaque risque significatif. Si c'est le cas, alors cela doit être pris en compte de manière appropriée lors de la conception.

Le terme *concerné par la sécurité* est utilisé pour décrire des systèmes qui doivent remplir une ou des fonctions spécifiques pour garantir que les risques sont maintenus à un niveau acceptable. Ces fonctions sont, par définition, des fonctions de sécurité. Tout système, réalisé dans une technologie quelconque (matérielle et/ou logicielle), qui remplit des fonctions de sécurité est un système concerné par la sécurité. Le système concerné par la sécurité peut être séparé d'un système de contrôle-commande ou peut être inclus dans ce dernier.

La sécurité fonctionnelle est simplement une méthode de prise en compte des risques qui vise à satisfaire deux types d'exigences auxquelles sont associés deux types d'analyse :

– exigences des fonctions de sécurité
  Ces exigences correspondent à ce que fait la fonction de sécurité. Elles sont dérivées de *l'analyse de risques*.
– exigences d'intégrité de la sécurité
  Ces exigences reflètent la probabilité que la fonction de sécurité soit réalisée correctement. Elles sont dérivées de *l'évaluation des risques*

La probabilité de défaillance d'un système se déduit de celles de ses composants. Le tableau 1.1 présente les quatre niveaux SIL définis par la norme en fonction de la probabilité de défaillance d'un système (pouvant contenir du matériel et du logiciel) concerné par la sécurité. Plus le niveau d'intégrité de la sécurité est élevé, plus la probabilité d'une panne dangereuse est faible.

Niveau d'intégrité de sécurité	Modes de fonctionnement	
	faible sollicitation (Probabilité de défaillance par sollicitation)	forte sollicitation ou continu (Probabilité de défaillance par heure)
4	$\geq 10^{-5}$ à $< 10^{-4}$	$\geq 10^{-9}$ à $< 10^{-8}$
3	$\geq 10^{-4}$ à $< 10^{-3}$	$\geq 10^{-8}$ à $< 10^{-7}$
2	$\geq 10^{-3}$ à $< 10^{-2}$	$\geq 10^{-7}$ à $< 10^{-6}$
1	$\geq 10^{-2}$ à $< 10^{-1}$	$\geq 10^{-6}$ à $< 10^{-5}$

Tab. 1.1 – Niveaux d'intégrité de la sécurité (SIL)

En conséquence, des niveaux d'intégrité de la sécurité élevés nécessitent une plus grande rigueur dans l'ingénierie d'un système concerné par la sécurité. Afin de réduire la probabilité d'une panne dangereuse, l'utilisation de méthodes rigoureuses est nécessaire pour éviter :

– les erreurs lors de la spécification ;
– l'introduction d'anomalies lors de la conception et du développement ;
– les anomalies lors de l'intégration ;
– les anomalies et les défaillances pendant les procédures d'exploitation et de maintenance ;
– les anomalies lors de la validation de sécurité.

La sécurité fonctionnelle présentée dans cette sous-section s'applique à tous les systèmes concernés par la sécurité (matériel et logiciel). Cependant les méthodes de réduction de risques permettant d'atteindre un niveau SIL élevé dépendent du type de système. Nous nous intéressons dans ce mémoire à la sécurité fonctionnelle relative aux logiciels, qui est présentée dans la sous-section suivante.

1.1.2 Sécurité fonctionnelle des logiciels

La partie 3 de la norme CEI 61508 [IEC00] décrit les prescriptions concernant les logiciels relatifs à la sécurité dans le cadre des systèmes électroniques programmables (PE). Tous les logiciels (systèmes d'exploitation, logiciel système, logiciels des réseaux de

communication, fonctions d'interface homme-machine, outils supports et micrologiciels (*firmware*)) sont concernés par la sécurité.

Il importe de souligner fortement que, contrairement aux systèmes matériels, qui donnent lieu à des défaillances aléatoires, les défaillances d'un logiciel sont seulement systématiques. La norme CEI 61508 définit les défaillances systématiques comme suit : *Une défaillance systématique est reliée de façon déterministe à une certaine cause, ne pouvant être éliminée que par une modification de la conception ou du processus de fabrication, des procédures d'exploitation de la documentation ou d'autres facteurs appropriés.*

Autrement dit, une défaillance systématique découle d'une faute non aléatoire. Cette faute se produira toutes les fois où les conditions (la cause) seront réunies. L'apparition de la cause mène donc systématiquement à la défaillance. Donc, aucun phénomène aléatoire ne rentre en compte dans l'apparition dans ce type de défaillance.

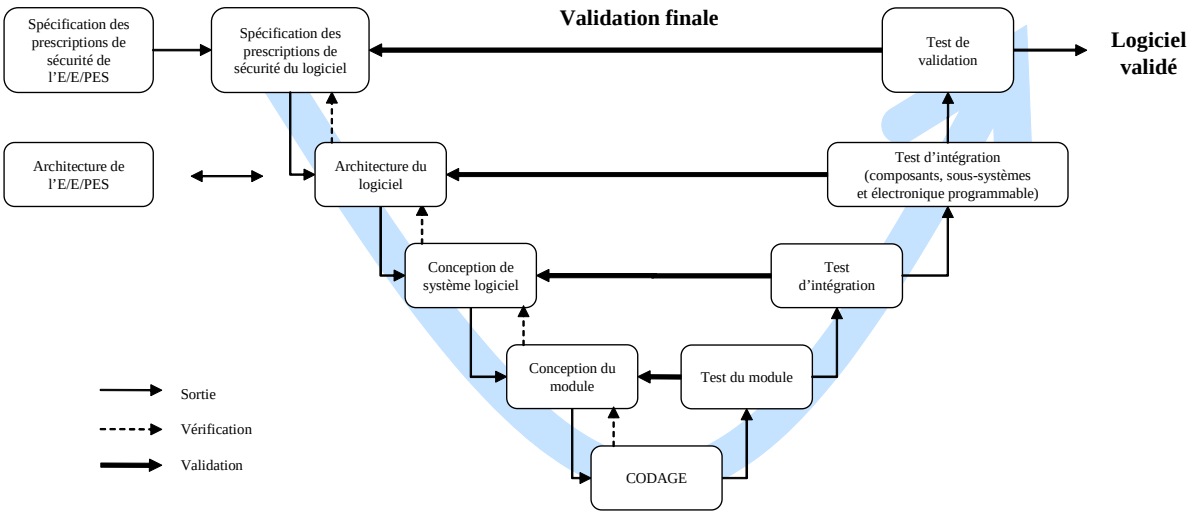


FIG. 1.1 – Cycle de vie d'un logiciel concerné par la sécurité

De ce fait, mis à part lors d'études statistiques portant sur l'exploitation, le calcul de la probabilité de défaillance d'un logiciel n'a pas de sens : le niveau d'intégrité d'un logiciel dépend seulement de la rigueur de son développement et des moyens employés pour éliminer les erreurs durant son cycle de vie. La figure 1.1, tirée de la norme CEI 61508, présente le cycle de vie d'un logiciel relatif à la sécurité. Entre chaque phase de ce cycle de vie, des vérifications et des validations sont effectuées. La norme définit ces deux termes comme suit :

vérification *confirmation, par examen et apport de preuves tangibles, que les exigences spécifiées ont été satisfaites* ;

validation *confirmation, par examen et apport de preuves tangibles, que les exigences particulières pour un usage spécifique prévu sont satisfaites.*

Autrement dit, la vérification s'assure que l'on a réalisé *UN* bon logiciel : aucune erreur n'a été introduite durant la phase précédente. La validation s'assure que l'on a réalisé *LE* bon logiciel : aucune information n'a été perdue durant cette phase.

Afin d'être *certifié* pour un niveau SIL donné, le développement d'un système concerné par la sécurité doit donc utiliser plusieurs méthodes de réduction et de prévention des risques par vérification et validation. La norme CEI 61508 indique la démarche à retenir pour l'application de ces méthodes :

1. Choix du niveau SIL
   Suivant la gravité des défaillances du système, un niveau de SIL est choisi.
2. Identification de méthodes de réduction de risques
   Parmi une liste de méthodes de vérification et validation (tableaux 1.2 et 1.3) et suivant le niveau SIL choisi, la liste des méthodes de réduction des risques est constituée.
3. Choix de la/des méthode(s)
   Parmi la liste de méthodes possibles, une ou plusieurs méthodes sont choisies en fonction de leur niveau de recommandation.
4. Evaluation des résultats
   Les méthodes retenues permettent d'identifier des défaillances systématiques possibles. Ces résultats sont analysés afin d'identifier la cause de ces défaillances.
5. Actions modificatrices à entreprendre
   Les causes des défaillances sont contournées, modifiées ou supprimées afin de ne plus apparaître, éliminant ainsi les possibilités de défaillances.

La *certification* d'un logiciel concerné par la sécurité est ensuite réalisée par un organisme de contrôle et de normalisation, comme le bureau Veritas ou le Technischer Überwachungs-Verein (TÜV), qui s'assure que la démarche de mise en place des méthodes de réduction des risques a bien été suivie et qu'elle est adaptée au niveau SIL désiré. Notamment, le type de méthode utilisée est comparé aux méthodes proposées (ou imposées) dans la norme.

Technique/Mesure	SIL1	SIL2	SIL3	SIL4
Test probabiliste	—	R	R	HR
Simulation/modélisation	R	R	HR	HR
Tests fonctionnel et boîte noire	HR	HR	HR	HR

Tab. 1.2 – Techniques de validation Recommandés (R) et Hautement Recommandé (HR) pour la sécurité du logiciel

En effet, la norme CEI 61508 propose un ensemble de techniques de vérification et de validation Recommandées (resp. Hautement Recommandées) pouvant (resp. devant) être utilisées en fonction du niveau d'intégrité visé (tableaux 1.2 et 1.3). Par exemple, pour un niveau SIL3, nous pouvons remarquer que la modélisation de fonctionnement et les automates à états finis sont hautement recommandés. Cependant la norme n'indique pas comment utiliser ces méthodes, seule une description sommaire est donnée en proposant des pistes d'action.

Cette thèse a pour but de fournir une méthode de réduction de risques pour des logiciels implantés sur des contrôleurs industriels en réduisant le nombre d'erreurs systématiques.

Technique/Mesure	SIL1	SIL2	SIL3	SIL4
Diagramme de flux de données	R	R	R	R
Automate à états finis	-	R	HR	HR
Méthodes formelles	-	R	R	HR
Modélisation du fonctionnement	-	HR	HR	HR
Réseaux de Petri temporels	-	R	HR	HR
Prototypage/animation	R	R	R	R
Diagrammes de structures	R	R	R	HR

TAB. 1.3 – Techniques de modélisation Recommandées (R) et Hautement Recommandées (HR) pour la sécurité du logiciel

Cette méthode permettra, comme décrit précédemment, de détecter une perte d'information (validation) et l'introduction d'erreur (vérification) entre les phases du cycle de vie.

Afin de développer cette méthode, la démarche de certification indiquée précédemment a été appliquée pour les logiciels de commande développés par l'entreprise Alstom Power. La section suivante présente donc la spécificité de la problématique industrielle au travers d'une présentation rapide de l'entreprise et de l'identification des verrous technologiques que doit lever cette démarche de certification.

1.2 Contexte industriel

1.2.1 Présentation générale d'Alstom

ALSTOM (anciennement GEC Alsthom, originellement Alsthom) est un groupe industriel français spécialisé dans les infrastructures d'énergie et de transport, présent dans deux grands secteurs : la construction ferroviaire et la production d'énergie.

Son offre inclut à la fois les systèmes, les équipements et les services. Son chiffre d'affaires, 13,5 milliards d'euros en 2005-2006, est réalisé à près de 90% hors de France. Le groupe emploie environ 60 000 personnes dans 70 pays. Certains des produits d'ALSTOM sont connus de tous : TGV, Queen Mary 2 (activité dorénavant vendue), d'autres, plus discrets, sont tout aussi innovants comme l'Alimentation Par le Sol (APS) des tramways.

Le groupe revendique des positions de numéro un mondial dans les centrales électriques clés en main, les turbines et alternateurs hydroélectriques, le service pour les sociétés d'électricité, les systèmes antipollution pour les centrales électriques, notamment à charbon, les trains à très grande vitesse, les trains à grande vitesse, les trains pendulaires, les systèmes de véhicules légers sur rail et les tramways, les trains de banlieue et régionaux, les services, la signalisation et les systèmes ferroviaires.

1.2.2 Présentation du système de contrôle-commande P320

ALSTOM fournit des centrales intégrées clés en mains et différents services associés pour différentes sources d'énergie, dont l'hydraulique, le gaz et le charbon.

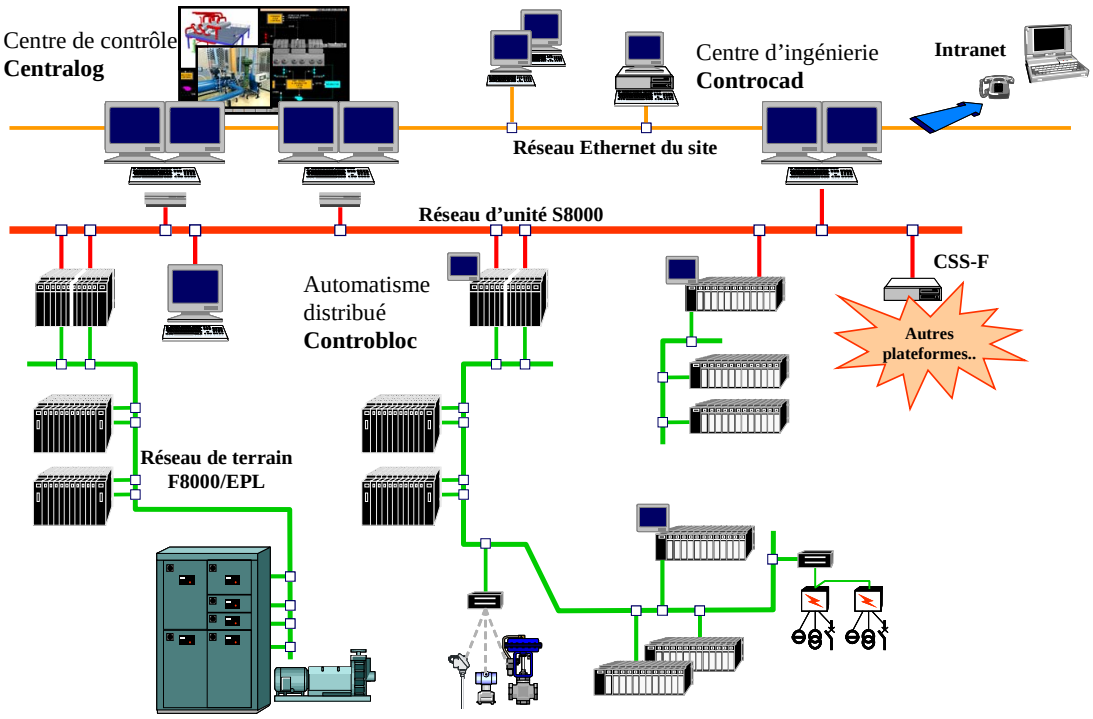


FIG. 1.2 – Vue d'ensemble du système P320

Le système ALSPA P320 (voir figure 1.2) est un système de contrôle-commande qui intègre l'expertise d'ALSTOM Power pour les centrales et le contrôle de machines à haute disponibilité. Le système P320 est un système d'automatisation de procédés rassemblant les fonctions traditionnelles d'un système de contrôle-commande et les fonctions de gestion du site pour l'aide à l'exploitation et à la maintenance.

Le système P320 est composé de trois sous-ensembles fonctionnels principaux :

Centralog , l'outil d'exploitation qui assure l'interface avec les opérateurs et l'environnement de la salle de commande avec une disponibilité élevée et de hautes performances en intégrant la conduite, la supervision sur écran et l'aide à l'opérateur pour l'expertise du procédé.

Controbloc , les cellules d'automatisme qui exécutent les fonctions de contrôle et de protection des machines, en utilisant des contrôleurs industriels, et réalisent l'interface avec le procédé. L'utilisation de réseaux de terrain autorise de plus la distribution géographique des équipements.

Controcad , l'atelier d'ingénierie qui offre une suite d'outils d'ingénierie performants aussi bien pour les bureaux d'étude que pour la documentation du site du système.

Nous nous intéressons dans ces travaux à l'outil Controcad. La figure 1.3 présente le cadre de son utilisation. Nous pouvons remarquer que Controcad apparaît à différentes

étapes de développement du système de contrôle commande : le développement logiciel au bureau d'étude, la simulation et les outils de vérification pour les plates-formes de test, la génération de documentation pour les consultants extérieurs (pour la certification par exemple) et enfin le chargement des contrôleurs industriels sur le site d'exploitation.

Plus précisément, notre étude porte sur le développement de logiciel d'automatisation pour contrôleurs logiques industriels, dans les bureaux d'étude. La sous-section suivante présentera les contrôleurs industriels et leurs caractéristiques, et sera suivie de deux sous-sections qui détailleront les possibilités de l'outil Controcad pour le développement de logiciel pour ces contrôleurs.

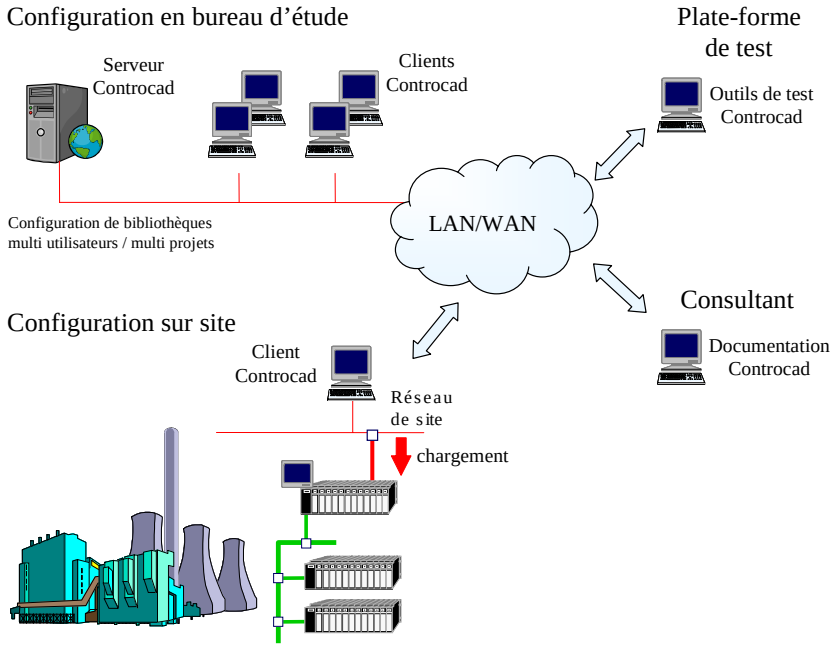


FIG. 1.3 – Vue d'ensemble de Controcad

1.3 Les contrôleurs logiques industriels

1.3.1 Principes généraux

Les contrôleurs logiques sont des composants d'automatisation qui reçoivent des informations du processus à contrôler par l'intermédiaire de leurs entrées et donnent des ordres aux actionneurs via leurs sorties. D'un point de vue logiciel, ils sont composés d'un programme et d'un moniteur d'exécution (figure 1.4). Les programmes peuvent contenir trois types de variables :

- **les variables d'entrée :** mémoires modifiées lors de la lecture des entrées du contrôleur.
- **les variables de sortie :** mémoires modifiées lors du traitement du programme utilisateur. Leurs valeurs sont recopiées sur les cartes de sortie du contrôleur lors de l'émission des sorties.

– **les variables internes :** mémoires modifiées lors du traitement du programme mais qui ne sont pas liées à une carte de sortie du contrôleur. Dans la suite de ce mémoire, ces variables seront considérées de la même manière que les variables de sortie, à savoir des variables modifiées par le programme.

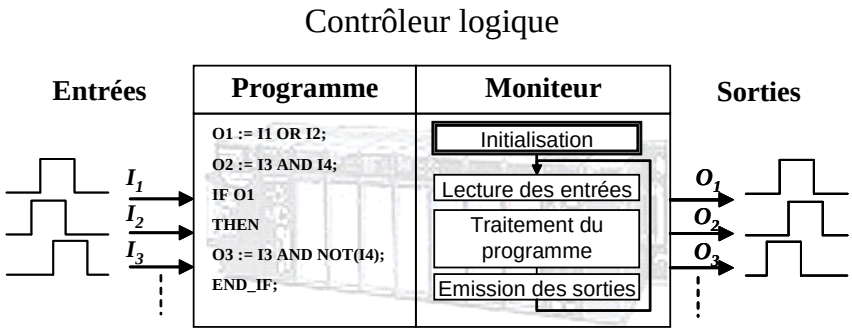


FIG. 1.4 – Un contrôleur industriel

Le programme indique le calcul des valeurs des variables de sortie et des variables internes en fonction des valeurs des variables d'entrée et des variables internes. Ces programmes peuvent être écrits dans l'un des langages préconisés par les normes suivantes : la norme CEI 61131-3 [IEC93], bien implantée dans le domaine de l'automatisme industriel, ou la norme CEI 61499 [IEC04], norme émergente, promue par certains offreurs de solutions d'automatisation, et qui fait l'objet de beaucoup de recherches universitaires.

Concernant la norme CEI 61131-3, plusieurs programmes peuvent être regroupés en *Program Organization Unit* (POU). Ces POU sont exécutés sur des moniteurs d'exécution cycliques ou périodiques, qui comportent une phase d'initialisation et trois phases principales :

Lecture des entrées Recopie de l'état des cartes d'entrées dans la mémoire des entrées du contrôleur industriel.

Traitement du programme Suite séquentielle d'opérations décrites dans un langage dédié.

Emission des sorties Recopie de l'état de la mémoire des sorties du contrôleur industriel dans les cartes de sortie.

La norme CEI 61131-3 définit cinq langages de programmation :

le Ladder Diagram (LD) Une représentation graphique d'équations booléennes combinant des contacts (en entrée) et des relais (en sortie). Il permet la manipulation de données booléennes, à l'aide de symboles graphiques organisés dans un diagramme comme les éléments d'un schéma électrique à contacts.

le Function Block Diagram (FBD) Un langage graphique permettant la construction d'équations complexes à partir des opérateurs standards, de fonctions ou de blocs fonctionnels.

le Sequential Function Chart (SFC) Un langage graphique utilisé pour décrire les opérations séquentielles. Le procédé est représenté comme une suite connue d'étapes

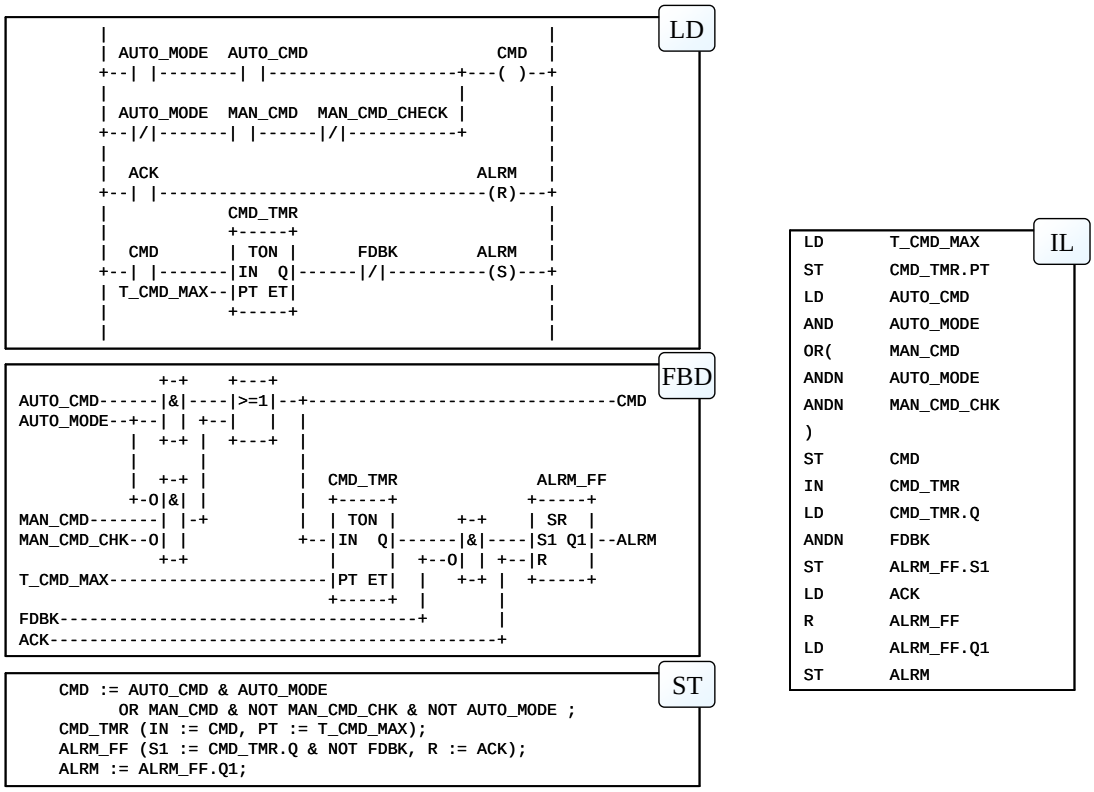


FIG. 1.5 – Quatres programmes équivalents écrits en LD, FBD, IL et ST

(états stables), reliées par des transitions. Une condition booléenne est attachée à chaque transition. Les actions dans les étapes sont décrites avec les langages ST, IL, LD ou FBD.

le Structured Text (ST) Un langage textuel de haut niveau dédié aux applications d'automatisation. Ce langage est principalement utilisé pour décrire les procédures complexes. C'est le langage par défaut pour la programmation des actions dans les étapes et des conditions associées aux transitions du langage SFC.

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

La figure 1.5 présente des exemples de programmes (supposés a priori équivalent, c'est-à-dire décrivant le même comportement) écrits dans les quatre premiers langages de la norme. Un exemple de programme écrit dans le cinquième langage, le SFC, est donné sur la figure 1.6.

La norme CEI 61131-3 [IEC93] définit aussi un ensemble de blocs fonctionnels pouvant être utilisés dans les différents langages de la norme (principalement dans les langages LD, ST et FBD). Ces blocs fonctionnels sont équivalents à des *Program Organization Units* (POU) qui retournent une ou plusieurs valeurs quand ils sont exécutés. Chaque instance

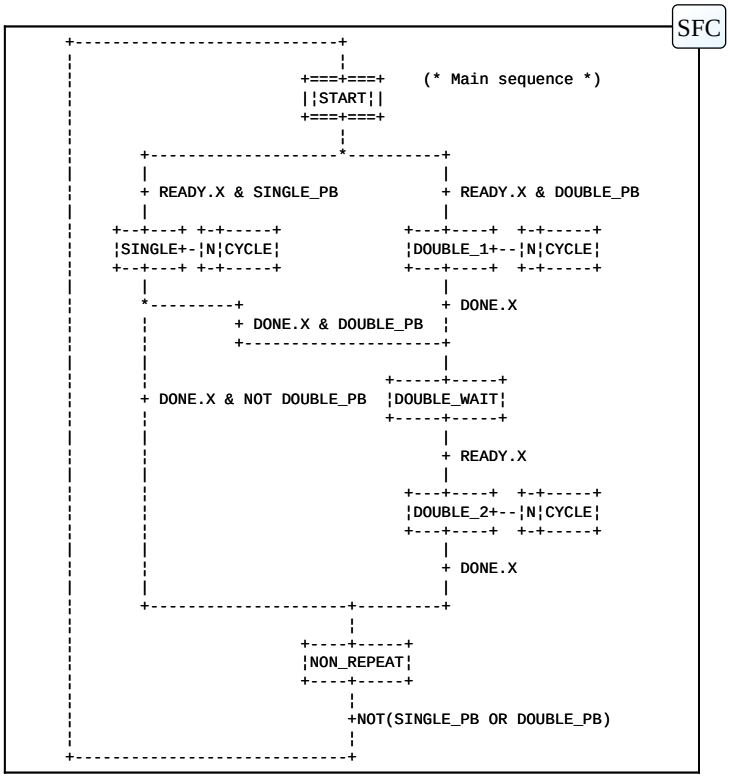


FIG. 1.6 – Exemple de programme écrit en langage SFC

(copie) d'un bloc fonctionnel possède son propre identifieur (nom du bloc fonctionnel) et sa propre structure contenant ses variables internes et de sortie.

La norme définit le comportement de plusieurs blocs fonctionnels :

- blocs fonctionnels bistables à activation prioritaire (`SR`) ou désactivation prioritaire (`RS`)
- blocs fonctionnels de détection de front montant (`R_TRIG`) ou descendant (`F_TRIG`)
- plusieurs blocs fonctionnels de comptage (`CTU, CTD, ...`)
- blocs fonctionnels de temporisation à l'activation (`TON`), à la déactivation (`TOF`) et à l'impulsion (`TP`).

Leurs comportements sont décrits, de manière informelle, avec des chronogrammes et du langage naturel. Nous verrons, dans le chapitre 5, que l'interprétation de ces blocs fonctionnels est souvent multiple et que le développement de nouveaux blocs fonctionnels se heurte à des problèmes de spécification de comportement.

Le deuxième standard de programmation des contrôleurs industriels est la norme CEI 61499 [IEC04] qui définit les blocs fonctionnels pour les processus industriels de mesure et de contrôle de système. Elle est principalement orientée vers la commande et l'automatisation distribuées. Un bloc fonctionnel (figure 1.7) dans la norme CEI 61499 est la base pour réaliser des applications. Deux types de blocs fonctionnels sont définis : les blocs fonctionnels de base et les blocs fonctionnels composés. Un bloc fonctionnel composé contient d'autres blocs fonctionnels composés et/ou d'autres blocs fonctionnels de base.

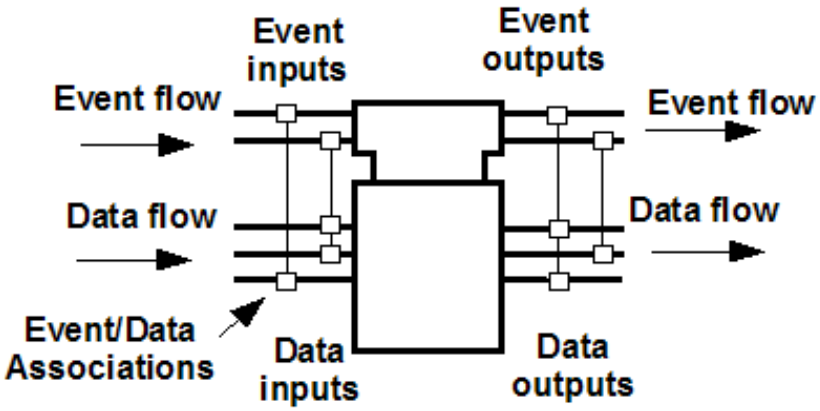


FIG. 1.7 – Un bloc fonctionnel de la norme CEI 61499

Un bloc fonctionnel de base contient des algorithmes et un diagramme de commande d'exécution (ECC pour *Execution Control Chart*), décrit dans une syntaxe proche de celle du SFC. Chaque bloc fonctionnel a des entrées et des sorties d'événements aussi bien que des entrées et des sorties de données. Dans un bloc fonctionnel de base, l'exécution d'un algorithme est déclenchée par l'occurrence d'un événement d'entrée. L'algorithme exécuté produit de nouvelles données de sortie à partir des données d'entrée. Quand l'algorithme a fini de s'exécuter, un événement de sortie est émis. Cet événement de sortie pourrait alors être l'événement d'entrée d'un autre bloc fonctionnel, et ainsi de suite.

1.3.2 Controcad : Un outil de développement de contrôleur industriel

Lors de la conception de la commande du procédé, Controcad permet de décrire indépendamment :

l'architecture matérielle qui comprend les contrôleurs, les réseaux, les pupitres, ...

l'architecture fonctionnelle qui comprend la description du comportement attendu de la commande, au travers des fonctions programmées en langage FBD ou SFC, et la déclaration de variables et de l'utilisation de blocs fonctionnels.

Le lien entre architectures matérielle et fonctionnelle est réalisé avec des structures de POU (Program Organization Unit) issues de la norme CEI 61131-3 [IEC93] (voir figure 1.8). Les POU contiennent différents programmes regroupés par fonctionnalité et chaque contrôleur industriel peut exécuter plusieurs POU.

Une fois la description de la commande réalisée, Controcad peut automatiquement générer des programmes pour chaque contrôleur en fonction de son type (fonction *génération du code* de la figure 1.8). Les échanges réseaux nécessaires sont *automatiquement* mis en place en identifiant les producteurs et consommateurs de données. Ceci permet une répartition des calculs non plus liés aux moyens d'obtention des informations mais en fonction des capacités de calcul du contrôleur et du temps de réponse voulu (plus ou moins loin de la source d'information). Ces programmes peuvent ensuite être chargés

sur les contrôleurs industriels (fonction *chargement* de la figure 1.8) ou simulés depuis Controcad (non représenté).

En parallèle au développement du programme de contrôle, des vues synoptiques sont mises en place afin de fournir les écrans de contrôle utilisables par le personnel de suivi du processus (fonction *HMI* (Human Machine Iterface) de la figure 1.8). Ils apparaissent sous deux formes : les vues indiquant l'état du processus et les interfaces homme - machine permettant le paramétrage du processus.

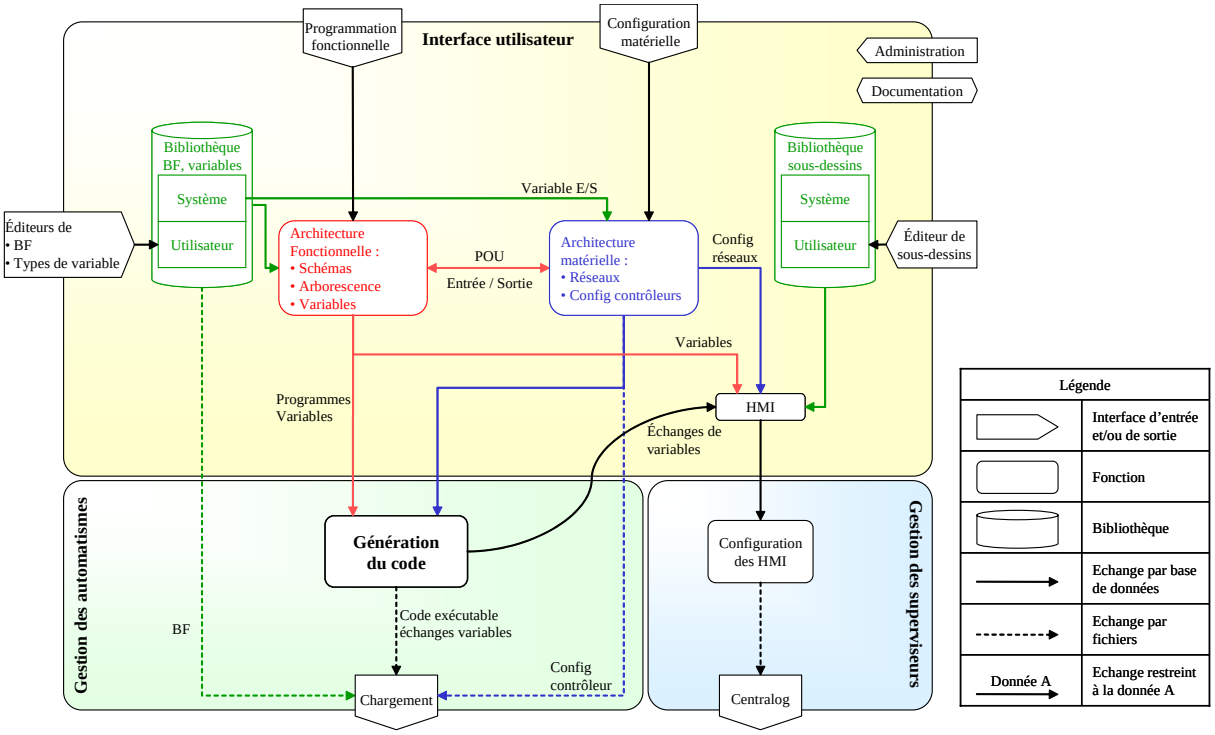


FIG. 1.8 – Échanges de données dans Controcad

A la vue de la structure de Controcad, nous pouvons faire plusieurs remarques :

- Controcad n'a pas été développé initialement dans l'objectif d'être un logiciel de niveau SIL3. Notamment le cycle de vie de cet outil n'est pas connu dans son ensemble car issu de plusieurs années de développement. Il n'est donc pas conforme à celui de la norme CEI 61508.
- Plusieurs outils externes sont utilisés dans Controcad, notamment :
  - L'outil logiciel Oracle, la base de données utilisée pour contenir et échanger la plupart des informations décrites sur la figure 1.8.
  - Les compilateurs fournis par les concepteurs de contrôleurs industriels (utilisés dans la fonction *generation de code*)
- Le point central de chaque développement de programme de contrôle est la génération du code du contrôleur.

Il ressort de ces constatations que la certification de l'outil Controcad en entier peut se révéler très difficile et demanderait un redéveloppement complet et l'utilisation d'outils

externes certifiés SIL3. À la vue du nombre d'heures de développement nécessaires pour atteindre cet objectif, il n'a pas été choisi.

Nous pouvons cependant constater que l'objectif primaire est d'améliorer la sûreté du processus et, dans notre cas, celui du programme de contrôle. Si nous ne pouvons pas certifier le générateur de ces programmes (ici, Controcad), la vérification du résultat de chaque génération vis-à-vis des spécifications de sécurité permettra de s'assurer de la sûreté des programmes de contrôle.

Finalement, l'objectif donc sera de s'assurer que les programmes générés par Controcad sont sûrs et corrects vis-à-vis du cahier des charges. Cette assurance permettra ensuite une certification plus aisée de ces programmes de contrôle car le développement suivra les préconisations de la norme en terme de vérification.

Cette aide à la certification de programmes de contrôle tournera donc autour de la vérification de la chaîne de génération de programmes de contrôle vis-à-vis des préconisations de la norme CEI 61508 comme proposé dans la section suivante.

1.3.3 Cycle de vie du programme généré par Controcad

Le schéma de la figure 1.9 présente la chaîne de développement depuis le cahier des charges du système à contrôler jusqu'à l'implantation des programmes d'automatisme dans le contrôleur industriel. Nous pouvons remarquer l'utilisation du langage *LEA*. Ce langage textuel, propre à l'entreprise Alstom et défini dans la documentation technique [P-T00], est proche du langage *ST* et est dédié à l'automatisation.

Une autre partie du développement est réalisée lors de la conception des blocs fonctionnels utilisables par l'utilisateur dans les langages SFC et FBD. Ceux-ci peuvent être fournis avec Controcad ou développés par les utilisateurs. Dans les deux cas, ces blocs fonctionnels sont issus d'une spécification qui est indépendante du système à contrôler.

En faisant le parallèle avec le cycle de vie préconisé dans la norme (figure 1.1), nous pouvons identifier différentes nécessités de vérification/validation dans Controcad :

- entre les étapes de développement, notamment entre les différentes transformations de langages, par exemple ici entre le LEA et le C ;
- validation des programmes obtenus par rapport au cahier des charges ;
- validation des blocs fonctionnels par rapport à leurs spécifications.

Afin de déterminer dans quelle mesure et avec quelles méthodes ces vérifications doivent être faites, la démarche de certification ci-dessous doit être suivie.

1.3.4 Démarche de certification

Dans le cadre de la certification des programmes générés par Controcad, nous nous intéresserons à la partie 3 de la norme CEI 61508 qui définit la sûreté des logiciels. Nous utiliserons donc la démarche de certification de logiciel définie dans cette norme en l'appliquant à notre contexte. La suite du document présentera plus précisément les points abordés et les méthodes utilisées.

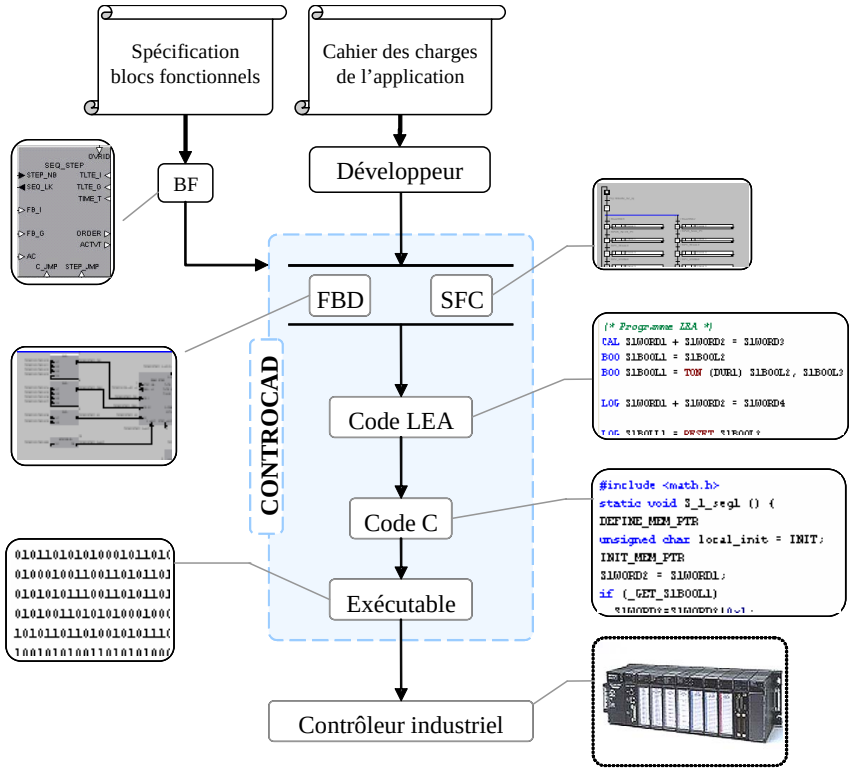


FIG. 1.9 – Chaîne de génération de programme dans Controcad

1. L'évaluation des prescriptions d'intégrité de sécurité.

   Le domaine de sécurité étudié est l'ensemble des programmes implantés sur des contrôleurs industriels mono-tâche pour la commande de systèmes de sécurité. Le niveau d'intégrité de sécurité, imposé par Alstom, pour ces programmes est SIL3.

   L'objectif est de s'assurer que les programmes implantés dans chaque contrôleur industriel :
   - sont conformes au cahier des charges initial ;
   - ne comportent pas d'erreur systématique.

   Pour réaliser cet objectif, une analyse du logiciel Controcad a été effectuée afin de déterminer les points pertinents à valider ou vérifier. Cette analyse est représentée sous la forme d'un schéma d'architecture permettant d'identifier les informations échangées entre les différents composants de Controcad, présenté dans la section 1.3.2.

2. La sélection et la documentation des stratégies, activités et techniques de vérification.

   Comme présenté dans la section 1.1.2, un ensemble de méthodes de réduction de risques est proposé par la partie 3 de la norme CEI 61508. Cependant la norme ne décrit pas comment utiliser ces méthodes. Afin d'effectuer une sélection de méthodes de vérification et validation, plusieurs critères ont été choisis ci-après par ordre de priorité :

   (a) le respect des préconisations de la norme CEI 61508 : la méthode de vérification

retenue doit être une des méthodes indiquées dans les tableaux 1.2 et 1.3.

(b) la possibilité d'application de méthodes académiques dans le domaine industriel.

(c) l'efficacité des méthodes possibles : le temps et la mémoire nécessaires à la vérification doivent être raisonnables.

(d) le coût de la méthode de vérification.

Les critères (b) et (c) sont détaillés dans le chapitre 2.

3. La sélection et l'utilisation des outils de vérification.
   Suite au choix de la méthode utilisée pour la vérification dans le chapitre 2, une méthode sera choisie. Les chapitres 3 et 5 indiqueront le moyen d'utiliser efficacement cette méthode.
4. L'évaluation des résultats de la vérification.
   Des exemples de résultats de vérification sont donnés dans les chapitres 4 et 6 en se focalisant, respectivement, sur l'efficacité de la représentation et sur les possibilités de vérification dans un contexte industriel.
5. Les actions modificatrices à entreprendre.
   Les actions correctives en cas de détection d'erreur dans le logiciel de commande sont traitées par le service de développement Controcad ou le bureau d'étude utilisant Controcad en fonction de la source des erreurs.

1.4 Objectifs scientifiques

Comme présenté dans la section 1.1.2, un ensemble de méthodes de vérification/validation est proposé par la partie 3 de la norme CEI 61508. Ceci explique l'intérêt porté à ces méthodes par les entreprises comme Alstom Power. Cependant la démarche à adopter pour mettre en œuvre ces méthodes n'est pas explicitée dans la norme.

Plusieurs recherches ont donc été menées depuis plusieurs années afin de fournir d'un coté des outils utilisant ces méthodes et d'un autre coté des démarches d'utilisation. Dans le domaine de la modélisation, méthode préconisée par la norme (voir tableau 1.2), nous pouvons identifier des recherches visant à obtenir des modèles formels, temporisés ([Zou04, BBG+05]) ou non temporisés ([Moo94, RK98, HLB03, dSR02, JFR01]), utilisables dans des outils de model-checking (pour *vérification de modèle*) comme UPPAAL [BLL+96] ou SMV [McM99], à partir de programmes de contrôleurs logiques industriels.

Cependant l'utilisation des méthodes de vérification formelle dans les entreprises concernées s'est peu (ou pas du tout) répandue [Joh07]. Plusieurs raisons sont la cause de cet état de fait :

- la difficulté pour les ingénieurs d'exprimer les propriétés formelles dans une logique temporelle,
- l'absence de traducteurs automatiques en langage formel dans les environnements de développement,
- des temps de vérification trop longs, voire infinis, en raison des problèmes bien connus d'explosion combinatoire,

– des contre-exemples difficiles à interpréter.

L'objectif de cette thèse est donc de fournir des représentations formelles de contrôleurs industriels et de blocs fonctionnels en vue d'une vérification efficace du respect du cahier des charges et de la conservation des informations entre les étapes du cycle de vie.

L'efficacité de cette représentation, caractérisée par le temps et la mémoire nécessaires aux vérifications, est essentielle dans notre contexte. En effet, le passage à l'échelle industrielle de ces méthodes de vérification est une obligation absolue pour ces travaux.

Ces représentations efficaces devront être, de plus, construites, à partir de modèles écrits dans des langages métiers industriels, tels que ceux manipulés par Controcad.

Les trois principaux apports escomptés de ces travaux de thèse sont décrits dans la figure 1.10, soit :

1. une modélisation des contrôleurs industriels efficace ;
2. une méthode de spécification formelle de blocs fonctionnels ;
3. l'utilisation des modèles pour la vérification de l'équivalence comportementale et la validation de propriétés issues de la spécification.

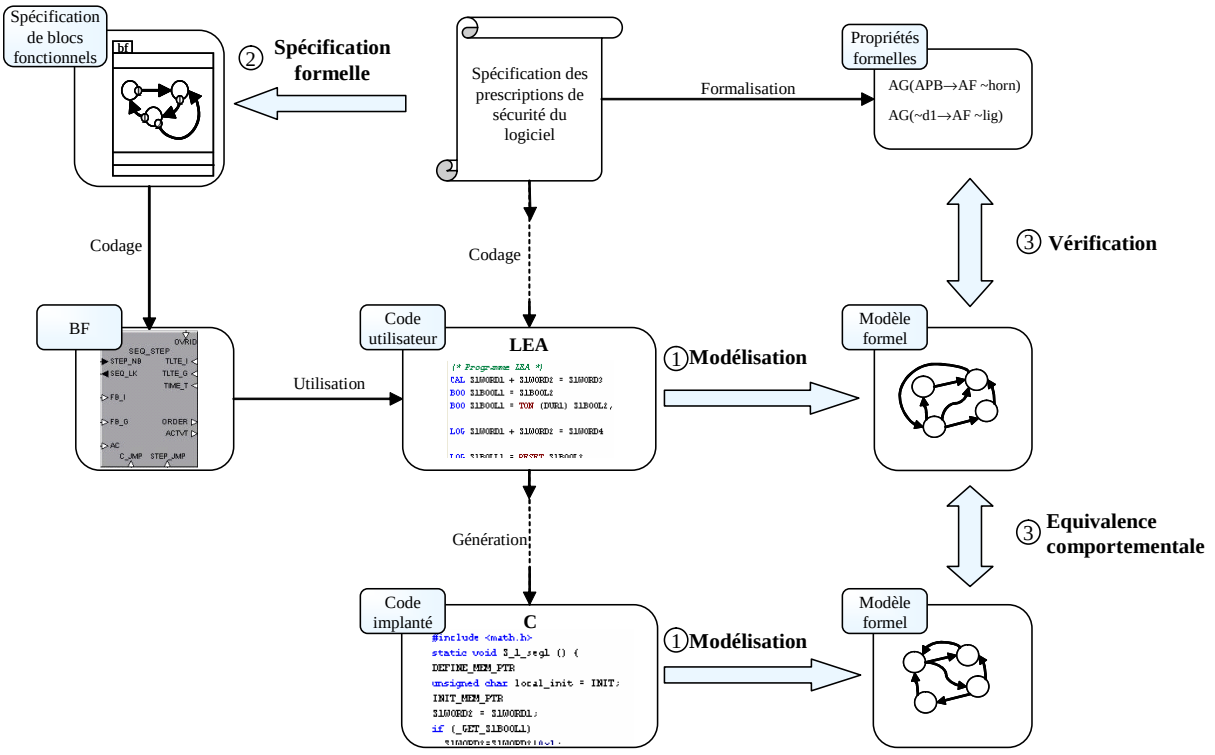


FIG. 1.10 – Principaux apports escomptés des travaux

Le prochain chapitre a donc pour but de déterminer le type de vérification formelle à mettre en œuvre ainsi que d'identifier les solutions existantes.

Chapitre 2

Etat de l'art sur la vérification formelle de contrôleurs logiques

Ce chapitre est consacré à une analyse bibliographique des travaux relatifs à la vérification formelle des contrôleurs logiques industriels, en se focalisant plus particulièrement sur les techniques de preuves par model-checking.
Afin de pallier au problème d'explosion combinatoire inhérent à ces techniques, deux classes d'abstraction (abstraction d'interprétation et abstraction de données) ont été proposées, qui sont également analysées.

Sommaire

2.1 Méthodes de vérification formelle

2.1.1 Introduction

L'importance croissante des systèmes automatisés dans la vie courante ainsi que le besoin accru de sûreté de fonctionnement pour ces systèmes impose le développement de systèmes de commande de plus en plus fiables pour lesquels les phases de tests sont de plus en plus lourdes. L'exigence de résultats impose en effet des tests de plus en plus longs sur des systèmes de taille de plus en plus importante.

Par exemple, la simulation est une des méthodes les plus utilisées industriellement lors des tests. Cependant, afin d'être exhaustive, son utilisation demande beaucoup de temps, surtout pour les systèmes à nombre d'entrés et sorties important. Pour cette raison, la plupart des simulations se restreignent à l'exécution de scénarios de tests, prédéfinis ou aléatoires. De ce fait, seules les défaillances se trouvant dans ces tests peuvent être trouvées. Cette méthode permet donc de vérifier de très grands systèmes sur des points particuliers mais pas de valider l'ensemble des exécutions possibles de ce système.

Afin de limiter la durée et le coût des tests, tout en garantissant le même niveau de sûreté, les méthodes formelles s'avèrent une perspective prometteuse. Ces méthodes permettent en effet de minimiser le risque de défauts avant même que le système ne soit implanté. Il existe deux grandes classes de méthodes formelles :

la synthèse formelle . Cette méthode vise à obtenir un système bon a priori ;

la vérification formelle . Cette méthode permet de s'assurer a posteriori et de manière exhaustive qu'un système est bon.

Comme indiqué dans le chapitre 1, la vérification formelle peut avoir deux buts :

la vérification , *confirmation, par examen et apport de preuves tangibles, que les exigences spécifiées ont été satisfaites* ;

la validation , *confirmation, par examen et apport de preuves tangibles que les exigences particulières pour un usage spécifique prévu sont satisfaites.*

Quatre principales classes de vérification formelle ont été proposées au cours des dernières années, répertoriées dans [FL00] : la simulation (présentée ci-dessus mais très difficilement formelle), l'analyse d'atteignabilité, le theorem proving et le model-checking.

Les méthodes basées sur l'analyse d'atteignabilité [KP96, FL98] construisent l'espace complet des états atteignables du système modélisé. Ces méthodes sont donc exhaustives mais sujettes à explosion combinatoire : le nombre d'états du système croît exponentiellement avec le nombre de variables discrètes.

La troisième classe de méthodes est le theorem-proving [RD02, VK02]. Le principe est d'utiliser les spécificités du formalisme pour prouver mathématiquement les propriétés demandées. La preuve de théorèmes nécessite une grande connaissance des systèmes formels (ce qui implique son utilisation par des personnels hautement qualifiés), et l'on n'a jamais la certitude de pouvoir mener à bien une démonstration. L'avantage principal de cette technique est d'éviter l'explosion combinatoire, problème récurrent en vérification formelle. Cependant, le theorem proving demande une forte expertise et les outils supportant ce type de méthode ne savent pour le moment que résoudre des cas simples.

Enfin, la dernière classe est la vérification par model-checking. L'outil de model-checking (dit *model-checker*) construit sa vérification à partir d'un modèle formel représentant le système étudié et d'un ensemble de propriétés à vérifier exprimées dans un formalisme adéquat. Le model-checker vérifie alors automatiquement les propriétés et détermine si elles sont vraies ou fausses, fournissant un diagnostic dans ce dernier cas.

Ce type de méthode présente l'intérêt de fournir une analyse exhaustive et de bénéficier du savoir-faire acquis en informatique. En effet, les model-checkers sont depuis longtemps déjà utilisés pour la vérification de logiciels, tels que les protocoles de communication, les logiciels de traitements de signaux, ...

Pour ces raisons, nous avons choisi pour ces travaux d'utiliser des méthodes de vérification formelle par model-checking comme une aide à la certification de contrôleurs logiques industriels. Nous allons dans les sections suivantes présenter les principes ainsi que les difficultés de mise en œuvre de ces techniques.

2.1.2 Démarche de vérification formelle par model-checking

La vérification formelle par model-checking d'un système (voir figure 2.1) nécessite :

1. de construire une modélisation formelle de ce système ;
   Ces modèles sont exprimés dans un langage formel, par exemple sous la forme d'un automate mais plus généralement comme un réseau de plusieurs automates synchronisés.
2. d'énoncer formellement les propriétés à vérifier ;
   On utilise alors un langage de spécification de propriétés, par exemple une logique temporelle.
3. de disposer d'un algorithme capable de déterminer si le système vérifie ou non les propriétés énoncées.
   Cet algorithme est implanté dans un outil issu de l'informatique, nommé model-checker.
4. le cas échéant, d'identifier la cause du non respect d'une propriété ;
   La plupart des model-checkers sont capables de fournir un diagnostic d'erreur complétant utilement la vérification qu'une propriété n'est pas satisfaite. Par exemple, dans le cas d'une propriété de sûreté que le système examiné ne vérifierait pas, le model-checker proposera un exemple d'exécution violant la propriété.

Sur chacun de ces points plusieurs problématiques se sont développées. Concernant le premier point, deux thèmes ont fait l'objet de recherches. Premièrement, plusieurs langages formels sont apparus, détaillés dans la section 2.1.3, chacun permettant une modélisation adaptée à une classe de système (temporisé, séquentiel, combinatoire, ...). Deuxièmement, des méthodes de modélisation ont été étudiées pour permettre de choisir les informations nécessaires à la vérification et qui doivent donc être présentes dans le modèle. En effet, la modélisation d'un système est fortement dépendante des propriétés à vérifier, comme identifié par [MDL06a], et implique une modélisation plus ou moins fine du comportement et une bonne définition de la frontière de modélisation (modélisation ou non des systèmes physiques,...).

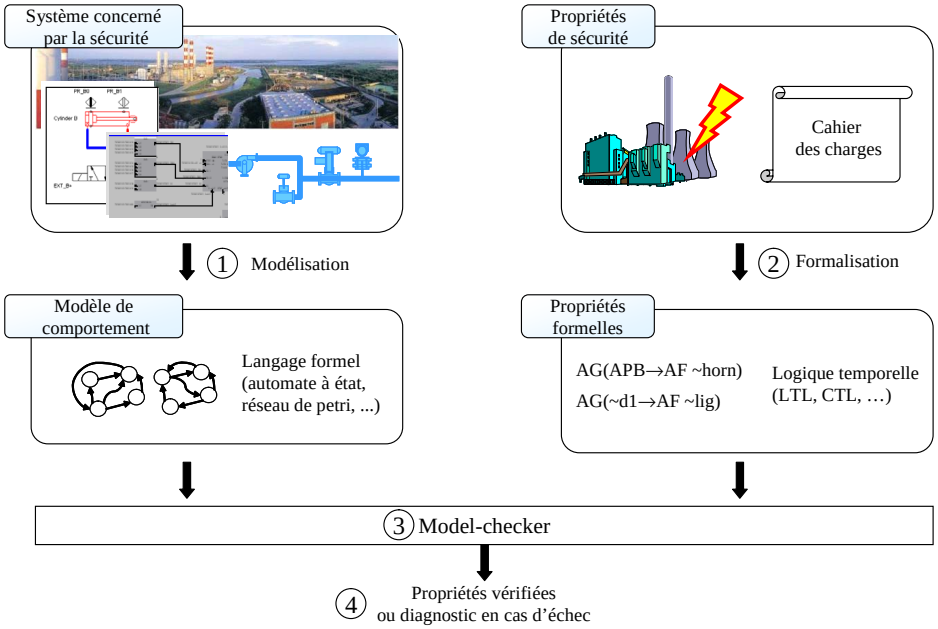


FIG. 2.1 – Démarche de model-checking

Le deuxième point, concernant l'expression de propriétés formelles, a été fortement travaillé dans le domaine informatique et a abouti à la définition de langages comme les logiques temporelles (voir section 2.1.3). Afin de transformer des propriétés issues du cahier des charges, souvent décrites en langage naturel ou semi-formel (arbre des défaillances, SADT, ...), des travaux ont été développés afin de fournir une formalisation de ces langages. Nous pouvons citer par exemple les travaux de [SRF06] qui propose une transformation d'arbres de défaillance en logique temporelle.

Beaucoup d'outils ont été proposés pour le troisième point, la plupart du temps liés au langage utilisé (voir la section 2.1.3). Les outils logiciels sont nécessaires à l'application des méthodes formelles en entreprise : sans outil, même une bonne méthode ne peut être applicable.

Le dernier point a été l'objet de recherches permettant de trouver les *meilleurs* contre-exemples suivant le type de propriété. Cela correspond généralement à trouver le plus court chemin (en terme de temps physique ou de nombre d'évolutions) menant à un état ne vérifiant pas la propriété.

La prochaine section présente les différents langages formels disponibles pour la modélisation des systèmes et l'écriture de propriétés formelles.

2.1.3 Langages de description de modèles formels

L'étape 1 consiste à décrire, dans un langage formel basé sur la théorie des systèmes à événements discrets, le comportement du système à analyser. Suivant le type de système à modéliser ou de propriétés à vérifier, plusieurs langages sont disponibles :

Automate à état Cette classe de langages couvre un champ très large, depuis la repré-

sentation de systèmes à événements discrets, avec les machines de Moore ([Moo56]), aux systèmes hybrides [Hen96] (voir figure 2.2). Cette classe est principalement utilisée pour le model-checking temporisé, avec l'outil UPPAAL [PL00], ou hybride, avec les outils HYTECH [HHWT97], Phaver [Fre05].

Réseau de Petri Présenté dans [DA92], ce langage se présente sous la forme d'un ensemble de transitions et de places pouvant contenir plusieurs jetons (voir figure 2.3). Il permet une représentation graphique compacte de systèmes complexes, surtout s'il est utilisé avec ses extensions colorées ou temporelles ([Jen97]). Son domaine d'application est principalement la simulation et l'analyse d'atteignabilité ([CGS07]) ainsi que quelques travaux de model-checking [HKG97].

Condition/Event Systems Proposé par [SK91], ce langage est décrit, sous sa forme graphique, par des diagrammes de blocs liés par des événements et des conditions. Il est destiné à représenter les systèmes à événements discrets et à temps continu et plus particulièrement les processus physiques. Ce langage est ensuite utilisé principalement pour des analyses d'atteignabilité [KP96] ou de la vérification par model-checking [RK98].

General Transition Systems Ce langage définit l'évolution du modèle comme un système de transitions global permettant, en fonction de l'état courant, de calculer l'état futur. Le système de transition est un ensemble de valeurs initiales $\mathcal{I}$ plus un ensemble de fonctions de transition $\mathcal{T}$ de la forme :

$$\begin{array}{lll} \mathcal{T}_j : & \mathcal{X}^{2n-1} & \to \mathcal{X} \\ & \forall k; k \neq j; (x_{j,i}, x_{k,i}, \ldots, x_{k,i+1}, \ldots) & \mapsto x_{j,i+1} \\ \text{Et} & x_{j,0} \in \mathcal{I}_j & \end{array}$$

Où :

$\mathcal{X}$ est l'ensemble des valeurs possibles d'une variable ;

$\mathcal{I}_j$ est l'ensemble des valeurs possibles d'initialisation de la variable x_j ;

x_j est une variable :
- $x_{j,i}$ indique son état courant,
- $x_{j,i+1}$ indique son état futur,

n est le nombre de variables.

Ce langage est principalement utilisé pour le model-checking, notamment grâce à l'outil NuSMV [CCG+02].

Langages synchrones Les langages synchrones considèrent les variables comme des signaux ; chaque signal est une suite de valeurs, cadencée par une horloge qui lui est propre. Les langages de programmation *Signal* [BLG90] et *ESTEREL* [Hal93] en sont les deux principaux représentants. Ils sont ensuite utilisés dans différentes méthodes de vérification [JFR01].

Équations algébriques La modélisation par équations algébriques permet une représentation mathématique d'un système et donc l'utilisation des méthodes de résolutions associées. De ce fait, elle est très bien adaptée pour la vérification par theorem

proving. Notamment, [Gun96] propose une méthode de résolution d'équations polynomiales modélisant les systèmes à événements discrets pour en effectuer la vérification. Cette classe de modèles contient aussi les modèles basés sur l'algèbre (Max,+) [BCOQ92].

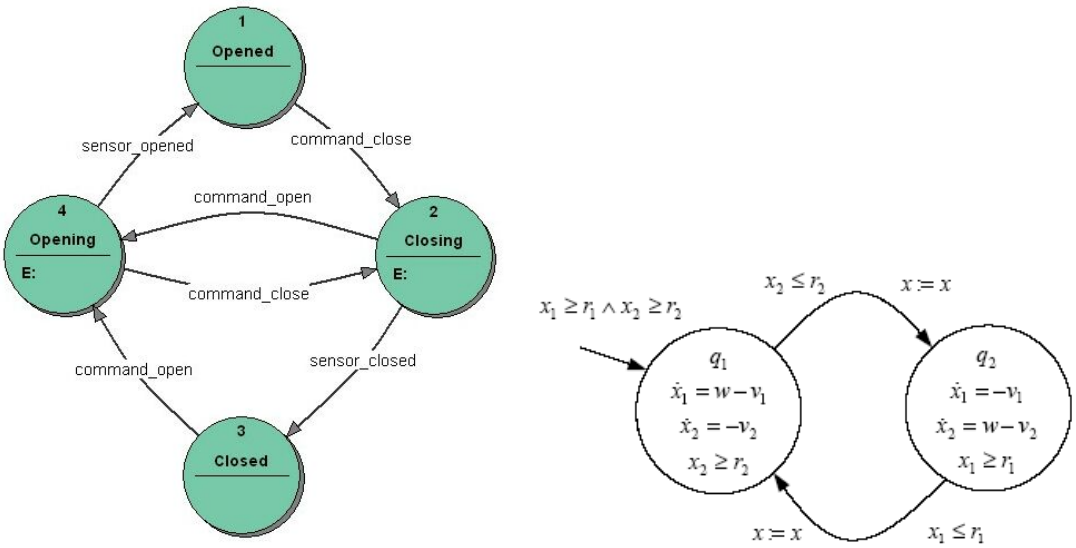


FIG. 2.2 – Exemples de machine de Moore et d'automate hybride

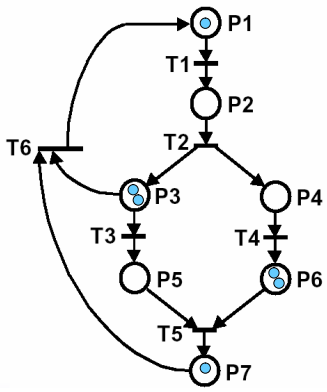


FIG. 2.3 – Exemple de réseau de Petri

L'étape 2 permet d'exprimer formellement les propriétés que doit respecter le système. Dans ce domaine, les logiques temporelles sont des formalismes adaptés pour énoncer des propriétés faisant intervenir la notion d'ordonnancement dans le temps, par exemple : *la barrière se ferme toujours avant le passage du train*. Ces logiques ont premièrement été proposées pour la spécification des systèmes réactifs par [Pnu77] qui a donné lieu à deux langages maintenant couramment utilisés en model-checking :

le langage CTL (*Computation Tree Logic*) Présenté par [QS82, CES86], les propriétés exprimées dans ce langage sont interprétées sur des automates finis dont

les états sont étiquetés par des propositions atomiques. Elle permet d'exprimer des propriétés sur ces états, faisant intervenir l'arbre des exécutions issues de cet état. Elle utilise des combinaisons booléennes, des modalités *next* (EX et AX) et des modalités *until* : E_U_ et A_U_. La figure 2.4 présente certaines modalités du langage CTL sur des arbres d'exécution.

le langage LTL (*Linear Temporal Logic*) Les propriétés exprimées dans ce langage, directement issu de [Pnu77], sont des traces d'exécution que doit respecter le système. Vérifier des propriétés de ce type correspond à s'assurer qu'il existe au moins une exécution du système qui respecte les traces exprimées dans ses propriétés.

Ces deux langages ont donné lieu à de nombreuses recherches, notamment sur le langage CTL en permettant par exemple la prise en compte du temps physique (TCTL [ACD93]) ou d'états fugaces (TCTLΔ [Bel06]).

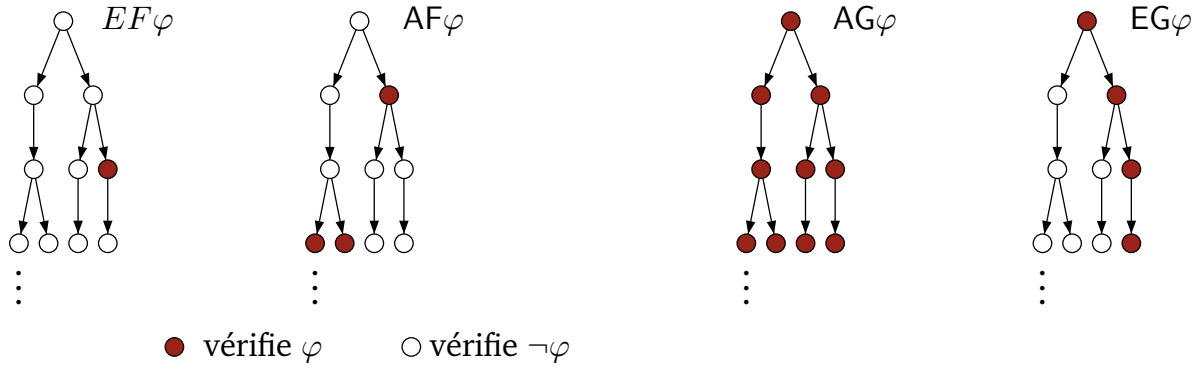


FIG. 2.4 – Illustration de certaines modalités de CTL

2.1.4 Difficultés relatives à la mise en œuvre du model-checking

Les deux principales difficultés de mise en œuvre du model-checking sont : d'un côté, comment modéliser un système et d'un autre côté, comment faire en sorte que ces modèles soient vérifiables dans un temps raisonnable.

Concernant la première difficulté, elle se retrouve sur chacun des points d'interface de la figure 2.1 : 1, 2 et 4. La première barrière vient principalement les connaissances nécessaires pour utiliser les langages formels, notamment dans le domaine industriel. Mais une deuxième barrière se trouve aussi dans la difficulté de trouver un bon compromis entre la finesse de représentation des systèmes et la complexité du modèle. Plus le modèle est fin plus les possibilités de vérification sont grandes mais plus le modèle est grand et complexe et donc difficile à réaliser. Nous pouvons citer dans ce domaine les travaux de[E.R05], qui propose une méthode et un langage de spécification qui permet d'écrire sous forme d'automates le comportement logique d'une représentation modale.

D'un point de vue industriel apparaissent ainsi des problèmes de choix de langage. En effet, l'utilisation de ces langages formels nécessite une grande expertise. Ainsi pour utiliser des méthodes formelles en entreprise, deux solutions doivent être proposées :

– Une méthode de modélisation automatisable (voir section 2.3).
– Une représentation de comportement adapté au domaine industriel et ayant une traduction possible en langage formel (voir chapitre 5).

Concernant l'expression de propriétés formelles, là encore, leur utilisation en industrie est difficile en raison des différences de langages. Il faut donc fournir des solutions, comme les travaux de [SRF06], permettant de transformer les langages industriels de spécification de faute en langages formels d'expression de propriété.

L'utilisation de model-checker en entreprise a aussi fait apparaître des problèmes de compréhension des contre-exemples notamment pour la détection de l'origine d'une défaillance. En effet, même si les contre-exemples sont les plus courts possibles, ils ne sont que des représentations de l'évolution du *modèle* qui peut être difficile à relier à l'évolution du système modélisé du fait de la simplification des modèles par exemple.

La deuxième principale difficulté est liée à l'augmentation de la taille des systèmes à modéliser. Plus le modèle est grand ou fin, plus le temps de vérification est long. En effet, le model-checking est confronté à un problème d'explosion combinatoire : les modèles de grande taille sont difficilement vérifiables notamment à cause de problèmes de temps de calcul et de mémoire nécessaire cette vérification. Cette complexité peut être caractérisée par trois critères :

– le nombre d'états atteignables ;
– le nombre de variables caractérisant chaque état ;
– le nombre de transitions entre les états.

Le rapport entre la complexité du système modélisé et la complexité du modèle obtenu définit donc l'efficacité des modèles. Afin d'améliorer l'efficacité, les modèles sont donc en général réalisés pour répondre à un seul type ou classe de vérification. Puis, par rapport à ce type de vérification, certaines abstractions sont disponibles afin de réduire la taille des modèles et donc d'améliorer leur efficacité.

Afin de vérifier les modèles de taille industrielle, il est donc nécessaire d'effectuer des abstractions permettant de limiter ces critères. Après avoir défini la limite de notre étude, nous détaillons, dans la section suivante, les abstractions possibles pour améliorer l'efficacité de ces modèles.

2.1.5 Limitations de l'étude

Nous nous intéressons dans ce mémoire au premier point de la méthode : la définition d'une méthode de modélisation. Pour ce faire, nous avons choisi d'utiliser la classe de langage GTS (*General Transition Systems*) pour la vérification par model-checking non-temporisé. Ce choix est fait par rapport aux antécédents du laboratoire ([Ros03, LCRRL99, BBG+05]).

En effet, notre principal objectif est de permettre la vérification de contrôleur industriel contenant des programmes de grande taille. Pour ce faire, il faut donc obtenir des modèles suffisamment efficaces pour être vérifiable dans un temps raisonnable. Afin d'atteindre cet objectif, nous cherchons des abstractions permettant de réduire la taille des modèles et donc leur temps de vérification. La section suivante propose donc un tour l'horizon des mécanismes d'abstraction possibles.

2.2 Mécanismes d'abstraction en model-checking

Suivant les domaines d'application le terme *abstraction* peut recouvrir plusieurs significations :

- Dans le domaine de la compilation informatique, ce terme est plutôt relié aux problèmes d'optimisation. Ces abstractions sont souvent issues d'une analyse de la sémantique statique d'un programme et permettent de calculer le résultat voulu au plus vite en réduisant la séquence de calcul et en la réorganisant. Comme les compilateurs doivent pouvoir opérer sur de très grands programmes, ils privilégient souvent la rapidité de l'analyse à la précision du résultat, la modification de la plage de calcul, la simplification des expressions ou la réorganisation des opérations modifiant d'autant la précision du calcul.
- Dans notre domaine de la vérification formelle, l'intérêt est porté sur la qualité de l'analyse en tenant compte du comportement dynamique du modèle (transitions entre les états). En effet, une abstraction peut ne pas conserver toutes les propriétés du programme original. Il est donc nécessaire de s'assurer qu'une abstraction est cohérente avec le type de propriétés à vérifier [LGS+95].

Nous nous intéresserons dans ce mémoire aux abstractions dites *sound*, c'est-à-dire aux abstractions qui conservent les propriétés qui doivent être vérifiées. Dans cette classe, [VA04] identifie deux types d'abstractions possibles :

- Les abstractions d'interprétation.
- Les abstractions de données

Les deux sous-sections suivantes présentent ces deux types d'abstraction ainsi que les travaux déjà effectués dans ces domaines.

2.2.1 Abstraction d'interprétation

Ce type d'abstraction vise à diminuer le nombre d'états du système ou le nombre d'interactions entre ceux-ci en ne gardant que les états et les interactions nécessaires à la vérification. La notion d'états *pertinents* et *non pertinents* est alors nécessaire :

- Les états *pertinents* correspondent à tous les états où la propriété à vérifier doit être satisfaite. Il faut donc que les modèles abstraits comprennent tous ces états pour que la vérification soit valide.
- Les états *non-pertinents* sont les états qui permettent d'atteindre les états *pertinents* mais qui ne sont pas concernés par la vérification. Ces états ne servent donc qu'à construire l'espace d'états sur lequel la vérification est effectuée.

En utilisant cette idée plusieurs abstractions d'interprétations se sont construites. Nous pouvons remarquer par exemple le principe de *cônes d'influence* proposé par [BCC98]. L'idée est d'identifier les données nécessaires à la vérification en utilisant la dépendance entre les données. Seules sont conservées les données qui sont directement reliées à la propriété concernée. **Si plusieurs propriétés doivent être vérifiées sur le système, alors il est réalisé autant de modèles que de propriétés.**

Une idée similaire, issue du domaine informatique, consiste en la découpe des programmes par analyse des flots de données (*program slicing*) [Tip94]. Cette méthode peut

être soit statique soit dynamique. D'après une comparaison effectuée dans [C+99], la découpe statique de programme (*static program slicing*) donne des résultats identiques à la méthode du cône d'influence, mis à part que le *static program slicing* opère sur un programme avant sa traduction en un modèle formel alors que la méthode du cône d'influence opère sur le modèle de ce programme.

La figure 2.5 présente un exemple de découpe de programme. Dans cet exemple, le critère retenu est (10, product), soit *quelle est la valeur de product à la ligne 10 ?*. Tous les calculs annexes, comme celui de *sum*, sont *découpés* et enlevés car ils n'influent pas la valeur de *product* à la ligne 10.

```
(1)   read(n);
(2)   i := 1;
(3)   sum := 0;
(4)   product := 1;
(5)   while i <= n do
      begin
(6)     sum := sum + i;
(7)     product := product * i;
(8)     i := i + 1
      end;
(9)   write(sum);
(10)  write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

FIG. 2.5 – a) un exemple de programme et b) une découpe du programme avec le critère (10, product)

La découpe dynamique de programme (*dynamic program slicing*)[KL88] analyse le flot de données en fonction des données d'entrée du programme. A partir de ceci, la découpe s'effectue comme une méthode de débogage et permet d'aboutir à des découpes plus fines des programmes mais dépendantes des données d'entrée.

[ES96] propose une autre méthode de réduction basée sur l'analyse de la symétrie des programmes. Cette abstraction exploite le fait que les programmes sont souvent composés de beaucoup de parties identiques. Les parties identiques sont abstraites en une seule permettant un gain de temps de vérification.

Il convient de souligner que ces trois premières abstractions dépendent des propriétés à vérifier. Pour chaque propriété à vérifier, l'abstraction est différente et donc le modèle obtenu est différent. Dans le cas de multiples propriétés à vérifier, les modèles sont alors aussi multiples. Nous présenterons, dans le chapitre 3, un des apports important de nos travaux, à savoir deux abstractions dépendant du type de propriété et *non plus de la propriété elle-même.*

Les méthodes d'abstraction présentées dans les deux sous sections précédentes permettent une diminution du temps et de la mémoire nécessaires à la vérification. Ces abstractions sont possibles pour tout modèle utilisable en model-checking et certaines sont déjà implantées sur des outils de model-checking comme NuSMV (cônes d'influence, abstraction symbolique) ou Uppaal (abstraction de données).

Cependant devant des programmes industriels de grande taille, les abstractions présentées ici ne suffisent pas. La section suivante s'intéresse donc plus spécifiquement à la vérification de contrôleurs industriels et notamment aux nouvelles abstractions possibles, dues notamment à l'utilisation d'un moniteur cyclique.

2.2.2 Abstraction de données

L'abstraction de données vise à représenter de façon compacte les données utiles à la vérification mais sans modifier le comportement global du modèle. [CGL94] répertorie plusieurs abstractions de données couramment utilisées dans le domaine de la vérification formelle :

- **la congruence du modulo d'un entier, dans le cas d'opérations arithmétiques.**
  Cette abstraction utilise le fait que l'opérateur modulo respecte les propriétés suivantes :
  $$((i \bmod m) + (j \bmod m)) \bmod m \equiv i + j \pmod m$$
  $$((i \bmod m) - (j \bmod m)) \bmod m \equiv i - j \pmod m$$
  $$((i \bmod m).(j \bmod m)) \bmod m \equiv i \cdot j \pmod m$$
  Où i et j sont des entiers et m un entier positif.
  Grâce à ces propriétés, la vérification peut être effectuée sur des plages limitées des valeurs des variables i et j et en prenant plusieurs valeurs pour m, deux à deux premières (n'ayant pas de diviseur commun autre que 1). Cette réduction est basée sur le théorème du *reste chinois* :

  Théorème 1 *Soit m_1, m_2, ..., m_n des entiers positifs deux à deux premiers et soit b, i_1, i_2, ..., i_n des entiers. Définissons $m = m_1 \cdot m_2 \cdot \ldots \cdot m_n$. Alors il existe un entier unique i tel que :*
  $$b \leq i \leq b + m \text{ et } i \equiv (i_j \bmod m_j) \text{ pour } 1 \leq j \leq n$$

  Supposons que nous puissions vérifier qu'une variable x est égale à i_j modulo m_j pour chaque m_1, m_2, ..., m_n deux à deux premiers. De plus, supposons que la valeur de x est inférieure à m_1, m_2, ..., m_n. Alors, d'après le théorème précédent, [CGL94] a démontré que la valeur de x est déterminée de manière unique. La vérification peut donc se restreindre à plusieurs vérification sur de petites plages (limitées par m_1, m_2, ..., m_n), le nombre d'états s'en trouvant considérablement réduit.
- **l'abstraction sur un ou plusieurs bits pour les opérations logiques.**
  Dans le cas de programme utilisant des opérations logiques, la plage de représentation des entiers est rarement utilisée entièrement. Afin de réduire la plage de vérification, il faut donc se focaliser sur les seuls bits intéressants.
- **l'abstraction combinant les deux abstractions précédentes.**
  Dans le cas de programme complexe utilisant plusieurs types d'opération, [CGL94] a démontré que les deux abstractions précédentes peuvent être appliquées simultanément.

– **l'abstraction symbolique.**
Cette abstraction permet d'éviter le calcul explicite de toutes les valeurs des variables en les représentant symboliquement. Cette abstraction a donné lieu à des représentations symboliques comme les BDD [Bry86] ou ADD [BFG+97] utilisées notamment dans l'outil de model-checking NuSMV.

2.3 Vérification formelle de contrôleurs logiques industriels

Pour vérifier formellement si un contrôleur logique industriel satisfait les propriétés exigées dans le cahier des charges, il faut :
– modéliser le comportement de la commande à l'aide d'un langage formel ;
– exprimer chacune des propriétés à l'aide d'une formule de logique temporelle construite à partir de propositions élémentaires (état des variables du programme) et des opérateurs de logique temporelle ;
– caractériser les différents états du modèle en langage formel vis-à-vis des propositions élémentaires ;
– vérifier si le modèle satisfait chacune des formules de logique temporelle.

Nous allons nous intéresser dans ce mémoire aux méthodes de modélisation de contrôleurs logiques industriels en mettant l'accent sur leur efficacité. La prochaine section présente un état de l'art des différentes méthodes de vérification formelle, au travers de deux classifications. Certaines de ces méthodes sont ensuite détaillées dans la section suivante afin d'identifier leurs avantages et inconvénients, notamment en matière d'efficacité de représentation. Puis la dernière sous-section présentera quelques réponses à ce besoin d'efficacité au travers de méthodes d'abstraction spécifiques aux contrôleurs logiques industriels.

2.3.1 Classification des approches

La vérification formelle de contrôleurs logiques peut être réalisée par de nombreuses méthodes présentées et classifiées dans [FL00, Mad00].

[FL00] classe les méthodes par approche, langage utilisé et méthode. L'approche est liée à la modélisation ou non des processus physiques :

Model based . Dans cette approche, le processus contrôlé est inclus dans l'analyse. Les propriétés peuvent donc porter sur l'état du système *en boucle fermée*.

Non model based . Les approches de ce type ne prennent pas en compte le comportement du processus contrôlé ; seul le contrôleur industriel est inclus dans la vérification. L'environnement avec lequel communique le contrôleur est supposé complètement non-contraint : tous les événements peuvent arriver à tout instant.

Constrained based . Les approches basées sur des contraintes sont des approches sans prise en compte du processus mais avec des inclusions de connaissances limitées sur le comportement du processus. Par exemple, le fait que deux entrées booléennes ne peuvent être vraies en même temps.

La classification relative au langage utilisé est identique à celle donnée dans la section 2.1.3. De même, la classification par méthodes correspond à celle donnée dans la section 2.1.1, soit la simulation, l'analyse d'atteignabilité, le model-checking ou le theorem proving.

Suivant cette classification, nos travaux se positionnent dans le model-checking sans modélisation du processus contrôlé, en utilisant un langage GTS (Global Transition System).

La classification proposée par [Mad00] est basée sur trois autres critères :
- la manière de représenter le cycle du moniteur ;
- l'utilisation de temporisation ;
- le type de langage utilisé dans les contrôleurs industriels.

La modélisation de contrôleurs industriels ne se restreint pas à la modélisation du programme qu'il contient. Il faut aussi en effet modéliser le moniteur d'exécution. Le cycle d'un moniteur de contrôleurs industriels peut être pris en compte de quatre manières différentes :

- les modèles sans prise en compte du cycle.
  Ces modèles sont principalement destinés à l'analyse statique du programme, par exemple pour vérifier les propriétés de dépendance de données entre des composants parallèles, des codes programmes inatteignables. Ils sont souvent liés à la vérification du respect des règles de programmation, mais ne prennent pas en compte l'aspect dynamique du programme.
- les modèles avec prise en compte explicite du cycle.
  La modélisation du cycle est directement incluse dans le modèle et permet de reproduire fidèlement le comportement du contrôleur industriel. Cette modélisation inclut tous les temps élémentaires d'exécution de chaque élément de langage. Ces modèles sont utiles lorsque l'aspect temporel est nécessaire à la vérification de propriétés telles que le temps de réaction d'un système.
- les modèles avec prise en compte implicite du cycle.
  Dans ces modèles, le temps n'est pris en compte que de manière globale en indiquant le temps d'exécution d'un cycle. Le temps d'exécution de chaque élément langage est considéré comme nul et le temps s'écoule seulement avant le passage à un nouveau cycle. Ces modèles sont utiles pour la vérification de la bonne synchronisation de plusieurs systèmes indépendants par exemple.
- les modèles avec abstraction du cycle.
  Cette modélisation est réalisée en considérant que le temps de cycle d'un contrôleur industriel est infiniment plus petit que le temps de réaction du processus. De ce fait, le temps n'est pas pris en compte dans la vérification, seul l'ordre d'exécution est considéré. Les propriétés vérifiables sur ses modèles sont liées à l'état du processus ou du contrôleur. Par exemple, une sortie du contrôleur (*ouvrir vanne*) doit toujours être vraie lorsque qu'une entrée (*réservoir plein*) est vraie.

La norme CEI 61131-3 définit plusieurs temporisations, par exemple pour retarder l'émission d'une sortie. Le deuxième critère de classification est donc la prise en compte ou non des temporisations sachant que la modélisation de ces temporisations peut s'avérer difficile dans le cas de modèle abstrayant le temps.

Le dernier critère de classification est le type de langage modélisé. En plus des cinq langages de programmation possibles de la norme CEI 6113-3 (IL, ST, SFC, LD et FBD), les constructeurs de contrôleurs industriels ont ajouté plusieurs fonctionnalités indépendantes qui peuvent (ou doivent) être prises en compte lors de la modélisation. De plus ces langages ne sont pas forcément modélisés complètement, permettant ainsi des simplifications ou des abstractions.

Vis-à-vis de cette classification, nos travaux se limitent à des modèles non-temporisés en abstrayant le cycle du moniteur et en prenant en compte les temporisations (voir chapitre 5). Les langages considérés sont les deux langages textuels de la norme CEI 6113-3 (IL, ST). Cependant les programmes en langages graphiques (SFC, FBD et LD) peuvent facilement être traduits en langage textuel (voir [MDL+06b] pour une traduction algébrique du Grafcet, un équivalent du SFC, et le chapitre 5 pour une représentation de blocs fonctionnels utilisés en FBD). La prochaine sous-section présente certains travaux existants dans ce domaine.

Cette présentation ne prétend aucunement à l'exhaustivité et le lecteur intéressé pourra se référer à l'annexe bibliographique et aux deux références [FL00, Mad00].

2.3.2 Exemples de modélisation de contrôleurs industriels

Comme présenté dans [LCRRL99], plusieurs travaux ont permis la modélisation de contrôleurs industriels à des fins de vérification par model-checking. La principale différence entre ces modèles réside dans la manière d'interpréter la séquence d'exécution et le moniteur.

Par exemple, [Ros03] modélise les programmes de commande par un produit cartésien de l'état des variables du programme et de l'état d'évolution du programme en lui-même. L'idée principale de son travail, nommé *microstep*, a été de prendre en compte l'état du contrôleur industriel en plus de l'état des variables. Ainsi, en spécifiant à l'outil de model-checking à quel endroit du cycle il se trouve, il est contraint de n'effectuer qu'une opération par évolution. Chaque primitive du langage est exprimée par un automate à état. Un exemple est donné dans la figure 2.6 pour la primitive *contact*. Une composition d'automates permet ensuite de modéliser un programme entier.

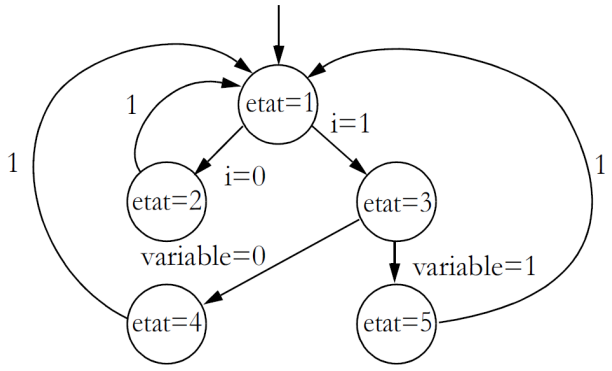


FIG. 2.6 – Représentation de la primitive *contact* dans [Ros03]

Cette méthode a été conçue pour la modélisation de programmes multi-langages. Elle utilise la modularité d'un langage constitué d'un nombre fini de primitives. Elle a de plus l'avantage de modéliser finement le comportement réel du contrôleur industriel. Toutefois, le model-checker doit exécuter un pas de calcul à chaque instruction traitée. De ce fait, le nombre d'états atteignables du modèle du contrôleur dans l'outil de model-checking augmente fortement avec le nombre de contacts, bobines ou blocs fonctionnels utilisés. Ce qui rend cette méthode inapplicable pour des programmes réels.

D'un autre coté, [RK98] présente une approche par modules SMV. Il propose de convertir le réseau Ladder en *function blocks* qui contiennent le nom des variables et l'ordre d'exécution du programme original et sont regroupés dans un *block diagram*. Par exemple, la figure 2.7 présente un exemple de programme et sa représentation en Condition/Event Systems. Ce modèle est ensuite transformé en code SMV en utilisant le principe de modules qui permettent de garder la hiérarchie des opérations grâce à la possibilité d'instanciations multiples et imbriquées des modules.

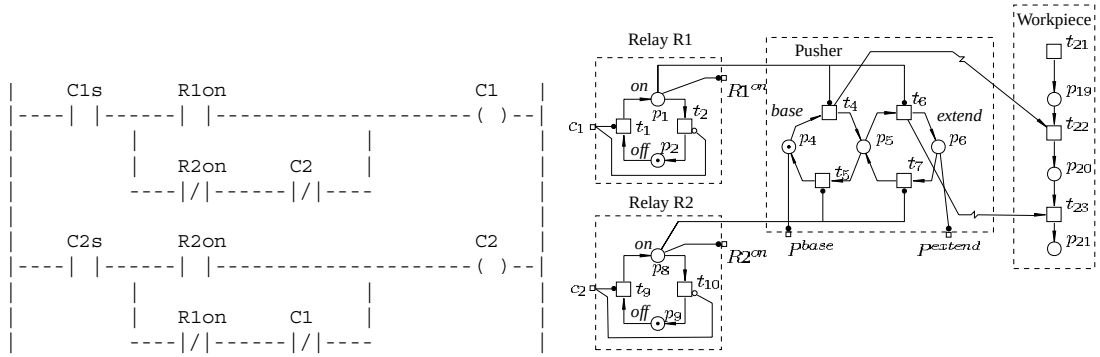


FIG. 2.7 – Exemple de programme et sa représentation en Condition/Event System

Toutefois, cette technique ne prend pas en compte les blocs fonctionnels, primitives du Ladder Diagram largement utilisées, et introduit un nombre important de variables, conduisant à des automates dont la taille rend difficile – voire impossible – l'analyse.

Afin de pouvoir vérifier des programmes industriels de grande taille, il est donc nécessaire de réaliser des abstractions. La section suivante présente des abstractions spécifiques à la modélisation des contrôleurs industriels.

2.3.3 Abstractions spécifiques aux contrôleurs industriels

Connaissant le comportement des contrôleurs industriels, présenté dans la section 1.3, la vérification et la validation de programmes exécutés sur ceux-ci peuvent être optimisées en abstrayant certaines parties. En effet, le cycle en trois phases présente l'intérêt de **bien séparer la communication avec le procédé et le calcul de la commande**.

Contrairement aux programmes informatiques conventionnels, le calcul de la commande sur un contrôleur industriel n'est pas réalisé après un événement extérieur mais cycliquement ou périodiquement. Ceci permet, notamment, d'avoir un temps de réponse borné quels que soient les événements puisque le calcul effectué est toujours le même.

De plus, les seuls moments où la commande est liée au procédé sont les phases de lecture des entrées et d'écriture des sorties. Vis-à-vis de cette exécution en trois phases la seule phase pouvant générer un état dangereux est l'émission des sorties, c'est-à-dire celle où le contrôleur industriel influence le processus. En effet, d'un point de vue extérieur, un contrôleur industriel ne fait que le lien entre les entrées et les sorties ; la méthode d'obtention de cette relation peut être abstraite car elle n'influe pas sur le processus à contrôler. De ce fait, hormis les problèmes de temps de réponse, non traités dans cette étude qui ne considère que le temps *logique*, le comportement du contrôleur pendant la phase de traitement importe peu du moment que le résultat, lors de l'émission des sorties, est correct par rapport à ce qui a été collecté lors de la première phase de lecture des entrées, et de l'historique de la commande.

Cette notion de point de vue extérieur se retrouve sur le type de propriété à vérifier et donc sur les abstractions possibles dans le cadre des contrôleurs industriels. En effet, deux types de propriétés peuvent être distinguées :

Les propriétés intrinsèques . Ces propriétés visent à caractériser le comportement interne de la commande comme proposé par [Huu05]. Ceci comprend notamment :
- les sauts conditionnels invariants : une condition de saut dans un programme est toujours vraie ou toujours fausse, rendant inutile ce saut conditionnel.
- les codes inatteignables : une partie du programme n'est jamais exécutée.
- les boucles infinies : la phase traitement de programme ne se termine jamais, menant à une absence de réaction de la partie commande.
- les calculs ou codes inutiles : des parties de code sont redondantes ou leur résultat n'est jamais utilisé. Ces calculs non nécessaires peuvent ensuite mener à une exécution erronée ou trop longue en temps d'exécution.

Les propriétés extrinsèques . Ces propriétés portent sur l'action de la commande sur le processus, et non sur le comportement interne de la commande. [BBF+01] identifie les propriétés suivantes :
- les propriétés de sûreté : un état dangereux ne sera jamais atteint.
  Par exemple : *Lorsque le capot de protection de la machine est ouvert, aucune commande de mouvement n'est possible.*
- les propriétés de vivacité : sous certaines conditions, un état sera toujours atteint.
  Par exemple : *En marche automatique, lorsqu'une pièce arrive au poste elle est chargée sur le centre d'usinage.*
- les propriétés d'équité : sous certaines conditions, un état apparaîtra une infinité de fois.
  Par exemple : *Une fois la machine démarrée, elle reviendra toujours dans l'état initial avant de poursuivre son cycle..*
- les propriétés d'atteignabilité : sous certaines conditions, un état pourra être atteint.
  Par exemple : *Quel que soit le mode de fonctionnement, il est possible de mettre la machine en mode manuel.*

La séquentialité du calcul, prise en compte dans les travaux de [Ros03, Zou04, dSR02, Huu05], peut être abstraite dans le cas de vérification de propriétés extrinsèques. En effet, le calcul étant identique à chaque cycle d'exécution et les propriétés extrinsèques

s'intéressant seulement au résultat de ce calcul, la séquence d'exécution peut être abstraite en une évolution unique dans le modèle, donnant directement la valeur des sorties[1] en fonction de la valeur des entrées.

Une première approche de réduction, utilisant cette idée, a été proposée par [Moo94]. Elle permet une modélisation d'un sous-ensemble du langage Ladder Diagram (bobine et contact seulement) de la norme CEI 61131-3 [IEC93]. [Moo94] présente de manière intuitive comment réduire le calcul séquentiel effectué avec un programme Ladder en un seul calcul d'un système d'équations récurrentes, en identifiant les variables déjà affectées. Cependant cette méthode est trop restrictive et trop peu formalisée pour être utilisable dans des cas industriels. Nous allons proposer donc une méthode générale et formalisée permettant de réaliser cette abstraction d'interprétation sur les langages textuels de la norme CEI 61131-3.

Sur la base de cette analyse bibliographique, le chapitre 3 va nous permettre de proposer deux nouvelles abstractions :

- Une abstraction d'interprétation, en identifiant les états pertinents dans l'évolution d'un contrôleur logique ;
- Une abstraction de données en identifiant les variables qu'il est nécessaire de mémoriser dans un modèle de contrôleur industriel.

[1] Nous rappelons que les variables internes sont aussi considérées comme des sorties

Chapitre 3

Modélisation des contrôleurs logiques industriels en vue de vérification par model-checking

Ce chapitre présente notre première contribution : une représentation formelle d'un contrôleur logique visant à améliorer l'efficacité de la vérification de propriétés extrinsèques. Cette représentation est basée sur deux abstractions originales : une abstraction d'interprétation et une abstraction de données.
Après avoir fixé les hypothèses de nos travaux, nous présenterons les principes de ces abstractions, puis détaillerons la démarche d'analyse qui permet d'obtenir, à partir d'un contrôleur industriel, un modèle formel, sous la forme d'un GTS compact, car utilisant ces abstractions.

Sommaire

3.1 Hypothèses pour la modélisation

3.1.1 Hypothèse sur les contrôleurs industriels

Nous rappelons que l'objectif de notre méthode est la vérification de contrôleur industriel logique mono-tâche à exécution cyclique ou périodique. De plus, dans la suite de ce mémoire, nous considérerons que les programmes de ces contrôleurs industriels respecteront les hypothèses suivantes :

1. l'action d'un programme se limite à la modification des valeurs de variables ;
2. seuls les langages textuels ST et IL de la norme CEI 61131-3 sont autorisés ;
3. pas d'utilisation de boucle non bornée (while, repeat) ni d'instruction de saut (goto, break, ...) ;
4. possibilité d'affectation multiple d'une variable.

Le premier point interdit tout effet annexe d'un programme (autre que l'affectation de variables) et interdit l'utilisation de fonctionnalités considérées comme sujettes à erreur dans la norme CEI 61508. Cela comprend l'interdiction :

- des pointeurs,
- de la modification de code programme,
- de lecture ou de modification directe des cartes d'entrée ou sortie du contrôleur,
- et toutes autres opérations qui modifient des éléments autres que des variables.

De ce fait, dans la suite de ce mémoire, les programmes de contrôleurs industriels seront considérés comme une suite séquentielle de modifications de variables, avec bien sûr les possibilités de structuration classiques de programme (if-then-else, switch-case, ...).

Deuxièmement, seuls les langages textuels sont étudiés afin de faciliter l'automatisation de la méthode. Cependant les programmes en langages graphiques (SFC, FBD et LD) peuvent facilement être traduits en langage textuel (voir [MDL+06b] pour une traduction algébrique du Grafcet, un équivalent du SFC, que nous avons adaptée à nos besoins dans le chapitre 6 et le chapitre 5 pour une représentation de blocs fonctionnels utilisables en FBD).

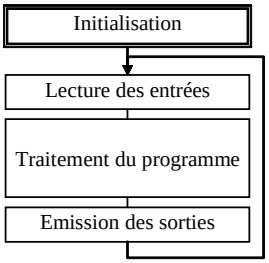
Le troisième point vient des bons usages de programmation pour les contrôleurs logiques. En effet, l'utilisation de boucles non bornées peut mener à des temps d'exécution trop longs, ce qui est incompatible avec des contraintes de temps de réponse court.

Le dernier point, bien que non conseillé dans certains guides de programmation, découle de la pratique industrielle de réutilisation de parties de codes, notamment pour la duplication de fonctions avec des variables intermédiaires de calcul. De plus, l'utilisation de structures conditionnelles conduit naturellement à de multiples affectations (une par condition). Par exemple, le programme suivant contient une double affectation de la variable logique de sortie O_1, en fonction des variables logique d'entrée I_1 et I_2 :

IF I_1
THEN
    $O_1 := 0$;
END_IF
IF I_2
THEN
    $O_1 := 1$;
END_IF

Lors d'une exécution, l'affectation devient unique si et seulement si les conditions sont exclusives. Dans l'exemple, ceci est vrai si I_1 et I_2 sont complémentaires.

Dans la suite de ce chapitre, tous les exemples, sauf mention contraire, respecteront ces hypothèses. Pour illustrer nos contributions, nous utiliserons deux exemples élémentaires (tableau 3.1) mais qui permettent d'illustrer clairement nos propositions.

	Variables	Initialisation	Programme	Moniteur
exemple a	Sorties booléennes : O_1, O_2, O_3, O_4, O_5. Entrées booléennes : I_1, I_2, I_3, I_4.	$O_1 := 0$; $O_2 := 0$; $O_3 := 0$; $O_4 := 0$; $O_5 := 0$;	**1** $O_1 := I_1$ OR I_2 ; **2** $O_2 := I_3$ AND I_4 ; **3** IF O_1 THEN **3-1** $O_3 := I_3$ AND NOT(I_4) ; END_IF ; **5** $O_4 := \mathrm{RS}(O_5, I_1)$ **6** $O_5 := O_2$ AND O_4 ; **7** $O_1 :=$ NOT(I_2 OR I_4) ;	
exemple b	Sorties entières : S_1, S_2, S_3. Entrée entière : E_1.	$S_1 := 0$; $S_2 := 0$; $S_3 := 1$;	**1** $S_1 := S_1 + S_3$; **2** $S_2 := S_1 + E_1$; **3** $S_3 := E_1 - 1$;	

TAB. 3.1 – Exemples utilisés dans ce chapitre (en gras, les numéros de ligne)

3.1.2 Hypothèse sur le model-checking

Pour nos travaux, nous avons retenu l'outil NuSMV [CCG+02], développé conjointement par les équipes du groupe Formal Methods à l'Institut Trentino de Culture (ITC - IRST), du groupe Model Checking à Carnegie Mellon University, du groupe Mechanized Reasoning à l'Université de Genova et du groupe Mechanized Reasoning à l'Université de Trento. Il s'agit d'un model-checker symbolique, c'est-à-dire qui travaille sur une représentation symbolique des états et des transitions de l'automate à vérifier. Cet automate est alors décrit à l'aide d'une expression combinatoire représentée elle-même par des diagrammes de décision binaires ordonnés (OBDD) [BCL+94] ce qui permet d'effectuer automatiquement une abstraction symbolique, présentée dans la sous-section 2.2.2.

Le langage utilisé pour décrire les modèles vérifiés par NuSMV est défini dans [McM99]. Ce langage est de type General Transitions System. De ce fait, le modèle est défini par un

système de transitions global permettant, en fonction de l'état *courant*, de calculer l'état *futur*. Nous utiliserons dans la suite de ce mémoire le terme *état courant* pour désigner l'état avant le calcul et *état futur* pour désigner l'état après le calcul. Dans la suite de ce mémoire l'*état courant* (i) d'une variable V_j sera noté $V_{j,i}$ et l'*état futur* $(i+1)$ sera noté $V_{j,i+1}$.

Dans la suite de ce mémoire, également, les états des automates utilisés dans les exemples ne contiennent que les valeurs après calcul des variables, les valeurs avant calcul pouvant être lues sur les états précédents.

3.2 Principes utilisés

Deux nouvelles abstractions de programme de contrôleur industriel sont présentées dans cette section : une abstraction de données et une abstraction d'interprétation.

Pour réaliser ces abstractions, nous identifions les seules informations nécessaires et suffisantes à la vérification des propriétés extrinsèques, définies dans la section 2.3.3. Ces abstractions ne sont valables que si nous nous intéressons à la vérification de propriétés extrinsèques. Elle sont par contre indépendantes de la propriété à vérifier : le *même* modèle abstrait sera utilisé pour la vérification de *toutes* les propriétés extrinsèques.

3.2.1 Abstraction d'interprétation : réduction du modèle aux seuls états *pertinents*

La première abstraction est de type abstraction d'interprétation et utilise le principe d'états *pertinents* et *non-pertinents* défini dans la section 2.2.1. Cette réduction vise donc à supprimer ces états *non-pertinents* tout en conservant les états *pertinents*. Il importe donc de déterminer quels sont les états pertinents lorsque l'on s'intéresse à la vérification de propriétés extrinsèques.

En considérant les trois phases du moniteur d'exécution des contrôleurs logiques, nous pouvons remarquer que les propriétés extrinsèques portent seulement sur les actions du contrôleur sur le système physique, lors de la phase d'émission des sorties (ES). Les états représentant cette phase sont donc *pertinents* car directement liés aux propriétés.

Par contre, les phases de lecture des entrées et de traitement du programme permettent de calculer la valeur des sorties mais la séquence suivie pour obtenir ces valeurs n'est pas importante : les états représentant ces phases sont donc *non-pertinents*. De plus, le traitement est aussi constitué d'une série de sous-étapes intermédiaires correspondant à l'exécution séquentielle des instructions du programme, entraînant d'autant plus d'états *non-pertinents* dans le modèle.

Par exemple, les figures 3.1a) et 3.1b) présente des traces d'exécution des *exemples a* et *b* en reprenant la représentation proposée dans [dSR02]. Sur ces traces, chaque transition correspond à une étape d'exécution du moniteur du contrôleur et un état représente l'ensemble des valeurs des variables après l'exécution de cette étape : après l'initialisation, après la lecture des entrés, après l'exécution de chaque ligne du programme et après l'émission des sorties.

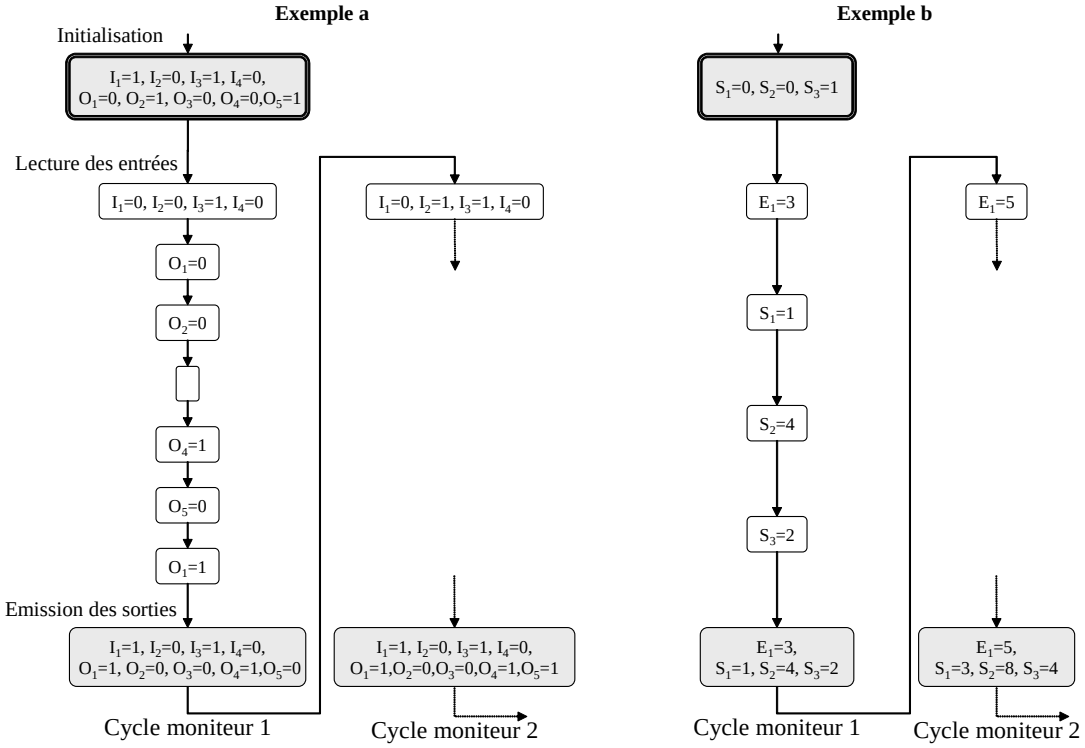


FIG. 3.1 – Traces avec tous les états des *exemples a* et *b* (seules les modifications des valeurs de variables sont indiquées)

Dans un but de lisibilité, seules les modifications sont représentées : sur l'*exemple a*, la lecture des entrées modifie seulement les entrées (I_1, I_2, I_3 et I_4), la ligne 1 modifie seulement la variable de sortie O_1, ... Nous pouvons remarquer que lors du premier cycle de l'*exemple a*, l'exécution de la ligne 3 (la structure conditionnelle) du programme ne modifie aucune variable.

Ces traces, si elles sont utilisées comme modèles, contiennent donc beaucoup d'états non-pertinents pour le type de propriété considérée. Il faut donc réaliser un modèle compact comportant, en plus de l'initialisation, seulement les états correspondant à la phase d'émission des sorties (ES) (grisés sur la figure 3.1) en éliminant les états intermédiaires de calcul, comme présenté sur la figure 3.2 pour les *exemples a* et *b*.

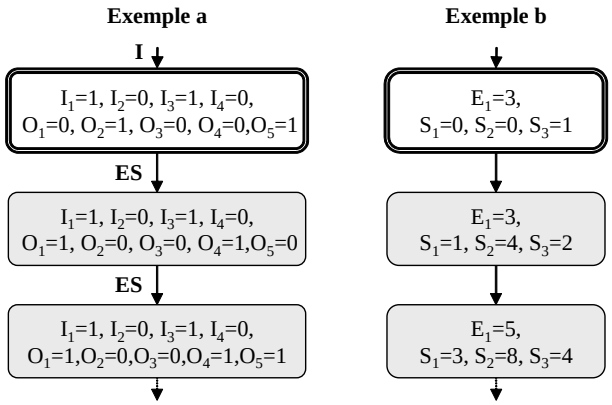


FIG. 3.2 – Traces des *exemples a* et *b* avec seulement les états pertinents

Plus formellement, il s'agit d'obtenir un modèle du contrôleur industriel sous la forme d'un GTS (*General Transition System*, défini dans la sous-section 2.1.3). Nous rappelons qu'un GTS est un ensemble de valeurs initiales $\mathcal{I}$ plus un ensemble de fonctions de transition $\mathcal{T}$ de la forme :

$$\begin{array}{lll} \mathcal{T}_j : & \mathcal{X}^{2n-1} & \rightarrow \mathcal{X} \\ & \forall k; k \neq j; (x_{j,i}, x_{k,i}, \ldots, x_{k,i+1}, \ldots) & \mapsto x_{j,i+1} \\ \text{Et} & x_{j,0} \in \mathcal{I}_j & \end{array}$$

Où :

$\mathcal{X}$ est l'ensemble des valeurs possibles d'une variable ;

$\mathcal{I}_j$ est l'ensemble des valeurs possibles d'initialisation de la variable x_j ;

n est le nombre de variables.

Comme indiqué dans la sous-section 3.1.1, nous considérons un programme de contrôleur industriel CI comme une suite séquentielle de modifications de variables de sortie O_j. Nous pouvons donc considérer un contrôleur industriel CI comme un système de transitions qui calcule la valeur des variables de sortie en fin de cycle $(i+1)$ à partir des variables de sortie en début (i) et fin de cycle $(i+1)$ et des variables d'entrée en fin de cycle $(i+1)$.

Le programme seul d'un contrôleur industriel est donc un système de transitions $\mathcal{F}$ qui se décompose, pour chaque variable de sortie O_j en une fonction de transition $\mathcal{F}j$:

$$\begin{array}{lll} \mathcal{F}_j : & \mathcal{O}^{2n-1} \times \mathcal{E}^m & \rightarrow \mathcal{O} \\ & \forall k, l; k \neq j; (O_{j,i}, O_{k,i}, \ldots, O_{k,i+1}, \ldots, E_{l,i+1}, \ldots) & \mapsto O_{j,i+1} \\ \text{Et} & O_{j,0} \in \mathcal{I}_j & \end{array}$$

Où :

$\mathcal{O}$ est l'ensemble des valeurs possibles d'une variable de sortie ;

$\mathcal{I}_j$ est l'ensemble des valeurs possibles d'initialisation d'une variable de sortie O_j ;

E_l est une variable d'entrée ;

$\mathcal{E}$ est l'ensemble des valeurs possibles d'une variable d'entrée ;

n est le nombre de variables de sortie ;

m est le nombre de variables d'entrée ;

Afin de prendre en compte le comportement du contrôleur dans son ensemble (programme + moniteur), nous considérons les variables de sortie ET les variable d'entrée du CI comme des variables du GTS. Il est alors possible d'établir une relation entre un cycle du CI et un pas de calcul du GTS. En effet, nous pouvons considérer qu'une entrée est équivalente, en comportement, à une sortie affectée de manière non-déterminisme (non contrôlée). Soit, il existe un système de transition GTS représentant le comportement de toutes les variables du contrôleur industriel CI :

$$\begin{array}{lll} & \forall O_j \in CI, f_j \in \mathcal{F}, & O_{j,i+1} = t_j \left(O_{j,i}, O_{k,i}, \ldots, O_{k,i+1}, \ldots, E_{l,i+1}, \ldots\right), k \neq j \\ \text{Et} & O_{j,0} \in \mathcal{I}_j & \\ \text{Et} & \forall E_l \in CI, & E_{l,i+1} \in \mathcal{E} \end{array}$$

Cette abstraction revient donc à faire correspondre un pas de calcul du model-checker avec un cycle du contrôleur industriel. Dans ce cas, l'*état courant* est l'état juste avant le début de la phase de lecture des entrées, pour ce cycle, l'*état futur* l'état au début de la phase d'émission des sorties (ou la fin du traitement. Bien sûr, une fois le cycle terminé, l'*état futur* devient l'*état courant*. Par exemple, la figure 3.3 présente une évolution possible des variables O_1 et O_2 de l'*exemple a* et les états courant et futurs de chaque variables. Seules les lignes L1, L2 et L6 sont indiquées pour l'exécution du programme ; ces lignes correspondent aux lignes de modification de ces variables.

Nous pouvons noter que le GTS obtenu permet des calculs avec un historique des variables de 1 : valeurs courante et future. Pour chaque pas de calcul du model-checker, ces deux valeurs peuvent être utilisées pour calculer le prochain état du système. Cependant, nous allons voir, dans la section suivante, qu'il n'est pas nécessaire de conserver ces deux valeurs pour certaines variables, ce qui permet une deuxième abstraction, de type abstraction de données.

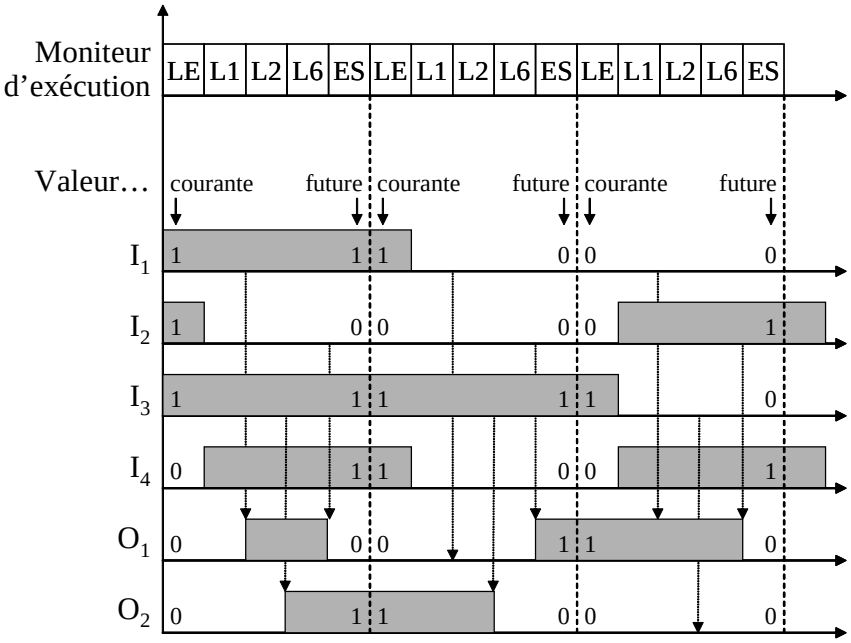


FIG. 3.3 – Chronogramme d'évolution des variables O_1 et O_2 de l'*exemple a*

3.2.2 Abstraction de données : les R-variables

L'abstraction détaillée dans cette sous-section a pour objectif la réduction du nombre de variables manipulées dans le modèle. En effet, la diminution de ce nombre permet de représenter chaque état pertinent de façon réduite.

Le nombre de variables caractérisant un état peut être réduit en limitant le modèle aux variables nécessaires à la vérification de la propriété. Cette réduction est réalisée en introduisant 2 types de variables :

Les R-variables. Ce sont les variables dont la représentation dans le modèle doit être double (valeur courante et valeur future). Leur valeur doit donc être mémorisée lors de la construction de l'espace d'états. Elles sont donc utilisées dans les formules de **R**écurrences définissant les valeurs futures des variables du modèle.

Les NR-variables. Ce sont les variables dont la représentation peut être la valeur future uniquement, la valeur courante n'étant jamais utilisée. La valeur courante n'a pas à être mémorisée pour la construction de l'espace d'états.

Nous pouvons donc définir formellement ces 2 types de variables par :

Définition 1 *Une variable V_i d'un modèle M sous forme GTS est une R-variable si et seulement si elle respecte la condition suivante :*

$$\exists V_j \in M, V_{j,k+1} = f(\ldots, V_{i,k}, \ldots)$$

Où :

- *$V_{j,k+1}$ est la valeur de la variable V_j dans l'état futur $(k+1)$,*
- *$V_{i,k}$ est la valeur de la variable V_i dans l'état courant (k),*
- *f est une fonction dont l'un des arguments est $V_{i,k}$.*

Cette définition formelle indique donc qu'une variable est une R-variable si sa valeur courante est utilisée pour calculer la valeur future d'une variable et qu'une NR-variable est une variable qui ne satisfait pas cette définition.

D'après la définition 1 des R-variables, nous pouvons constater, en analysant chaque ligne de l'*exemple b*, que :

1. La valeur de S_1 dans l'état futur $k+1$ est définie à partir de la valeur de S_1 dans l'état courant k et de la valeur de S_3 dans l'état courant k.
   $\longrightarrow$ S_1 et S_3 sont utilisées une fois avec leurs valeurs dans l'état courant k.
   $\longrightarrow$ S_1 et S_3 sont donc des R-variables.
2. La valeur de S_2 dans l'état futur $k+1$ est définie à partir de la valeur de S_1 dans l'état futur $k+1$ (définie à la ligne précédente) et de la valeur de E_1 dans l'état futur $k+1$ (définie lors de la lecture des entrées).
3. La valeur de S_3 dans l'état futur $k+1$ est définie à partir de la valeur de E_1 dans l'état futur $k+1$ (définie lors de la lecture des entrées) et de la valeur numérique 1.

$\longrightarrow$ S_2 n'est jamais utilisée avec sa valeur dans l'état courant k.
$\longrightarrow$ S_2 est donc une NR-variable.
L'*exemple b* contient donc une NR-variable et deux R-variables : les variables S_1 et S_3 sont des R-variables et la variable S_2 est une NR-variable. En effet, la valeur courante de S_2 ($S_{2,k}$), affectée au cycle précédent (k) ou à l'initialisation, est remplacée par $S_1 + E_1$. Donc la valeur courante de S_2, $S_{2,k}$ n'est pas nécessaire pour calculer l'état futur du modèle.

De la même manière pour chaque ligne de l'*exemple a* :

1. la valeur[2] O_1 est définie à partir des valeurs d'entrée $I_{1,k+1}$ et $I_{2,k+1}$ (définies lors de la lecture des entrées).
2. la valeur $O_{2,k+1}$ est définie à partir des valeurs d'entrée $I_{3,k+1}$ et $I_{4,k+1}$ (définies lors de la lecture des entrées).
3. la valeur $O_{3,k+1}$ est définie à partir des valeurs d'entrée $I_{3,k+1}$ et $I_{4,k+1}$ (définies lors de la lecture des entrées), de la valeur intermédiaire O_1 (définie à la première ligne) et de la valeur $O_{3,k}$ (dans le cas où la condition du IF (O_1) n'est pas remplie).
   $\longrightarrow$ O_3 est utilisée une fois avec sa valeur dans l'état courant k.
   $\longrightarrow$ O_3 est donc une R-variable.
4. la valeur $O_{4,k+1}$ est définie à partir de la valeur d'entrée $I_{1,k+1}$ (définies lors de la lecture des entrées) et des valeurs $O_{5,k}$ et $O_{4,k}$ (dû au bloc fonctionnel RS : *mémoire* à mise à zéro prioritaire).
   $\longrightarrow$ O_4 et O_5 sont utilisées une fois avec leur valeur dans l'état courant k.
   $\longrightarrow$ O_4 et O_5 sont donc des R-variables.

[2]Cette valeur n'est ni associée à l'état courant k ni à l'état futur $k+1$ car O_1 est redéfinie à la ligne 6 : c'est une valeur intermédiaire

5. la valeur $O_{5,k+1}$ est définie à partir de la valeur d'entrée $I_{1,k+1}$ (définies lors de la lecture des entrées) et des valeurs $O_{2,k+1}$ et $O_{4,k+1}$ (définies aux lignes 2 et 4).
6. la valeur $O_{1,k+1}$ est définie à partir des valeurs d'entrée $I_{2,k+1}$ et $I_{4,k+1}$ (définie lors de la lecture des entrées).

$\longrightarrow$ O_1 et O_2 ne sont jamais utilisées avec leur valeur l'état courant k.
$\longrightarrow$ O_1 et O_2 sont donc des NR-variables.
L'*exemple a* contient donc 3 R-variables (O_3, O_4 et O_5) et 2 NR-variables (O_1, O_2).

Afin de repérer les variables dans un programme de contrôleur industriel, nous pouvons donc donner une autre définition des R-variables :

Définition 2 *Une variable est une R-variable si elle est, dans le programme, utilisée avant d'être affectée.*

En effet, l'utilisation d'une variable avant son affectation dans le programme correspond, dans le modèle GTS, à l'utilisation de l'état courant de cette variable. Cela inclut bien sûr l'utilisation d'une variable dans sa propre affectation car l'affectation d'une variable se fait toujours *après* avoir calculé l'expression liée à cette affectation.

Une déduction logique est un calcul qui est effectué au même titre que celui d'une affectation dans le modèle mais la valeur qui en découle n'est pas mémorisé pour le calcul futur. Toute NR-variable peut être, si besoin, être évaluée de cette façon. Cependant l'utilisation d'une déduction logique n'est nécessaire que si la variable est utilisée dans une propriété. En effet, les expressions de ces déductions logiques peuvent avantageusement remplacer les appels de NR-variables. De ce fait, un modèle formel ne contient plus que les variables d'entrée et les R-variables.

L'abstraction de données consiste donc à ne caractériser chaque état pertinent du modèle formel que par R-variables. Les NR-variables ne figurent pas dans le modèle abstrait.

Par contre, les affectations de ces NR-variables sont transformées en *déductions logiques*, non incluses dans le modèle GTS qui représente le contrôleur. Ces déductions logiques, fonction uniquement des variables d'entrées et des R-variables, seront utilisées lorsque des propriétés à prouver feront intervenir des NR-variables.

Par exemple, si nous nous intéressons à la propriété suivante sur l'exemple a :
les variables de sortie O_1 et O_2 ne sont jamais vraies simultanément
Alors la preuve portera sur l'expression :

$$\text{NOT}(I_2 \text{ OR } I_4) \text{ AND } (I_3 \text{ AND } I_4) = 0$$

La figure 3.4 présente le résultat des deux abstractions pour les *exemples a* et *b*. Nous pouvons aussi remarquer que l'initialisation d'une NR-variable est inutile car non utilisée. Dans nos exemples, l'initialisation des variables O_1, O_2 de l'exemple a et de la variable S_2 de l'exemple b sont donc enlevées.

Une fois de plus, nous pouvons remarquer le GTS obtenu permet des calculs avec un historique des variables de 1 : valeurs courante et future. Cependant, comparé au GTS obtenue à la sous-section précédente, ces deux valeurs sont utilisées pour toutes

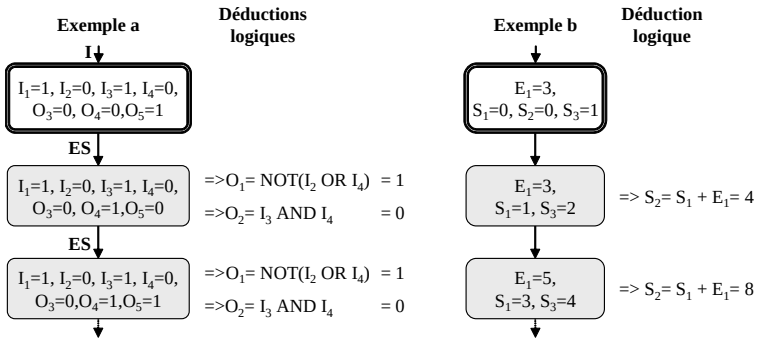


FIG. 3.4 – Traces des *exemples a* et *b* avec seulement les R-variables

les variables du *GTS*. Le *GTS* contient donc les seules informations nécessaires à la vérifications.

3.3 Démarche de modélisation

Cette section s'intéresse à la construction, sous la forme d'un GTS, d'un espace d'état discret qui conserve toute la sémantique du contrôleur, pour ce qui concerne les propriétés extrinsèques, tout en étant de taille suffisamment réduite pour être manipulable par des outils de model-checking sans donner lieu à une explosion combinatoire.

La démarche présentée dans cette section a donc pour but de transformer, selon les abstractions définies à la section 3.2, les affectations (conditionnelle ou non) du programme en un unique système d'équations GTS, utilisable par l'outil de model-checking. La figure 3.5 présente une vue globale des étapes de transformation qui seront détaillées dans les sous-sections suivantes.

Cette démarche nécessite deux données d'entrée : le programme du contrôleur et une bibliothèque de blocs fonctionnels. La bibliothèque contient des modèles formels de blocs fonctionnels issus de la méthode de spécification et de représentation proposée dans le chapitre 5.

Plus précisément, cette démarche comporte trois phases :

1. l'analyse des dépendances statiques.
   Cette phase permet, en partant du programme du contrôleur, d'obtenir les relations de dépendance statique entre les variables.
2. la détection des R-variables et la prise en compte de l'ordre d'exécution.
   Cette phase détecte, en suivant les définitions données dans la section 3.2.2, les R-variables puis transforme les relations de dépendance statique en relations de dépendance temporelle.
3. la génération du modèle.
   Le modèle NuSMV est obtenu dans cette phase en utilisant les informations contenues dans les relations de dépendance temporelle pour transformer le programme en modèle.

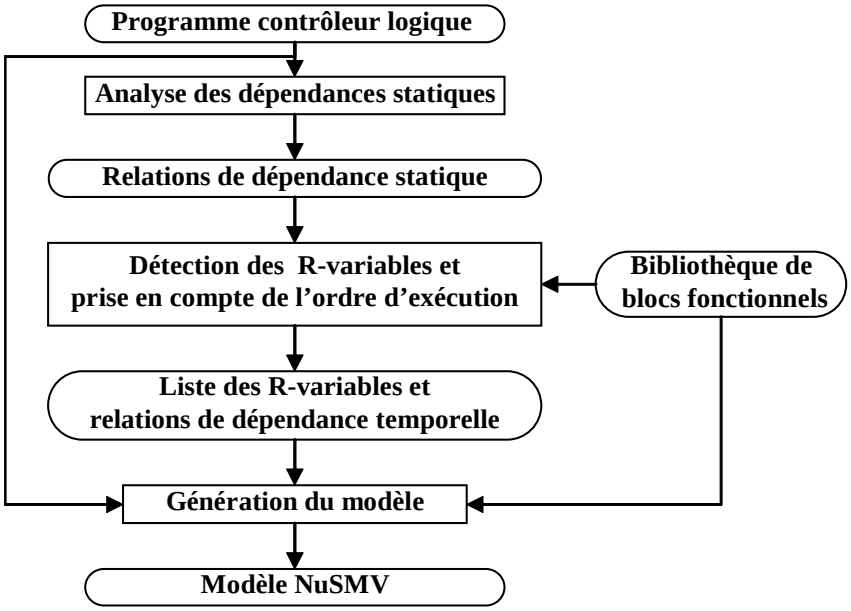


FIG. 3.5 – Vue globale de la méthode de construction du modèle

3.3.1 Analyse des dépendances statiques

L'analyse des dépendances statiques vise à obtenir, en partant du programme du contrôleur, les relations de dépendance statique entre les variables. Chaque relation de dépendance indique qu'une variable est calculée à partir d'une autre. Cet ensemble de relations est ordonné selon l'ordre de lecture du programme (de haut en bas).

La figure 3.6a) présente les dépendances statiques de l'*exemple a*. Sur cette figure, une flèche d'une variable X vers une variable Y indique que Y dépend de X, c'est à dire que X est présente dans l'expression permettant de calculer Y. Par exemple, nous pouvons voir que la variable O_5 dépend de O_2 et O_4. Soit, en reprenant le formalisme utilisé dans 3.2.1 : $O_5 = f_5(O_2, O_4)$.

Les variables utilisées dans les blocs fonctionnels, comme le bloc fonctionnel RS dans notre exemple, sont aussi prises en compte. Ceci comprend les variables utilisées comme arguments du bloc fonctionnel mais aussi, le cas échéant, ses variables internes. Ce bloc fonctionnel RS, par exemple, représente en effet une mémoire à un bit, prioritaire à la mise à zéro. Dans notre cas, la variable O_4 est utilisée comme mémoire du bloc RS donc O_4 dépend d'elle-même. La connaissance des variables internes est issue de la bibliothèque de blocs fonctionnels définie dans le chapitre 5.

Nous pouvons remarquer que l'évaluation des expressions n'est pas étudiée. De ce fait, si une variable est présente dans une expression mais qu'elle influe pas sur la valeur obtenue alors l'expression est tout de même considérée comme dépendante de cette variable. Par exemple, soit une variable booléenne A affectée par l'expression B OR 1 où B est une variable booléenne, alors A est dépendante de B même si la valeur affectée à A est toujours 1 quelle que soit la valeur de B.

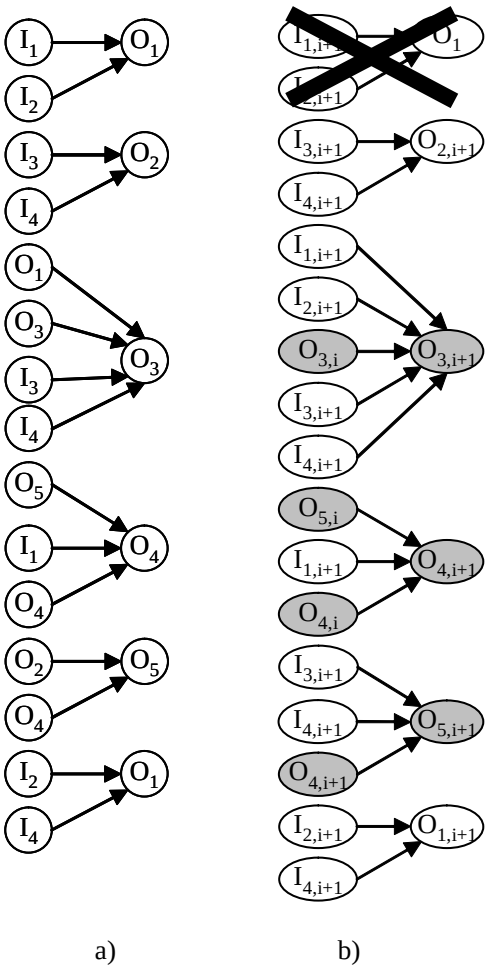


FIG. 3.6 – Dépendances a) statique et b) temporelle des variables du programme de l'*exemple a*

3.3.2 Détection des R-variables et prise en compte de l'ordre d'exécution

Dans un premier temps, les R-variables sont détectées selon la définition 2 donnée dans la section 3.2. Dans l'*exemple a*, les variables O_1 et O_2 sont affectées avant d'être utilisées, donc ce sont des NR-variables, alors que les variables O_3, O_4, O_5 (grisées sur la figure3.6b) sont des R-variables.

Deuxièmement, la prise en compte de l'ordre d'exécution permet d'obtenir, partant de l'ensemble de relations de dépendance statique, un ensemble de relations de dépendance non-ordonnée, en ajoutant à chacun des noms de variables un indicateur temporel caractérisant son état (courant ou futur). L'objectif est d'obtenir une dépendance temporelle qui aura les qualités suivantes :

- Chaque variable de sortie doit être définie par une seule relation de dépendance. Les affectations multiples sont supprimées.
- Seules les variables d'entrées et les R-variables sont utilisées pour définir les variables de sortie. Les relation de dépendance temporelle ne sont donc définies qu'à partir de ces deux types de variables (origines des flèches qui traduisent les relation de dépendance temporelle.
- L'état futur de chaque variable O_j , noté $O_{j,i+1}$, peut dépendre de l'état futur des variables d'entrée ($I_{k,i+1}$), de l'état futur ou courant des autres variables ($O_{l,i+1}$ ou $O_{l,i}$) et, s'il s'agit d'une R-variable, de son état courant ($O_{j,i}$).

Connaissant la liste des R-variables, la transformation des relations de dépendance est réalisée dans le sens d'exécution du programme en utilisant l'algorithme 3.7. Cet algorithme permet de remplacer chaque variable par sa valeur courante, future ou une relation de dépendance suivant son type (R-variable ou non) et sa dernière relation de dépendance rencontrée en suivant le sens d'exécution du programme.

Finalement, les relations correspondant aux affectations intermédiaires des variables sont enlevées. De ce fait, seule la dernière affectation de chaque variable est conservée. En effet, d'un point de vue du comportement externe du contrôleur industriel, seules les dernières affectations ont un effet. Cependant, les affectations intermédiaires peuvent influer sur le comportement externe si elle interviennent dans le calcul des affectations finales. Notre représentation prend en compte ce type de phénomène car l'algorithme 3.7 reporte le calcul de dépendance correspondant à chacune de ces affectations intermédiaires dans celles des affectations finales qui les utilisent. Dans notre exemple, seule la dernière relation de dépendance de la variable O_1 est conservée. La dépendance déduite de sa première affectation est reportée dans celle de la variable O_3 qui utilise cette valeur intermédiaire de O_1.

Chaque cycle étant représenté par un seul état, les variables d'entrée apparaissent seulement dans leur état pour ce cycle, soit $i + 1$. Chaque variable d'entrée I_k dans les relations de dépendance est donc remplacée par $I_{k,i+1}$.

La figure 3.6b) présente le résultat de l'application de cet algorithme sur l'ensemble de relations de dépendance statiques du programme de l'*exemple a*. Il importe enfin de souligner que :

- l'ensemble de relations obtenu n'est plus ordonné selon le sens d'exécution du pro-

for *Chaque relation de dépendance de la variable* O_k **do**
  for *Chaque variable* O_j *dont dépend* O_k **do**
    switch O_j *est* **do**
      case *une R-variable :*
        switch *dont l'affectation* **do**
          case *n'a pas encore eu lieu :*
            Remplacer O_j par sa valeur courante $O_{j,i}$;
          case *précédente était la dernière :*
            Remplacer O_j par sa valeur future $O_{j,i+1}$;
          otherwise
            Remplacer O_j par sa dernière relation de dépendance rencontrée;
      case *une NR-variable :*
        Remplacer O_j par sa dernière relation de dépendance rencontrée;

FIG. 3.7 – Transform_dependance(**Dépendances statiques**) → **Dépendances temporelles**

gramme. Cet *ordre* est contenu dans les indicateurs temporels de chaque variable.
– les variables O_3, O_4, O_5 (grisées sur la figure 3.6b)) sont les seules variable à respecter la définition 1 et donc les seules NR-variables : elles sont les seules variables utilisées dans leur état courant dans les relations de dépendance temporelle.
– seules les R-variables O_3, O_4 et O_5 et les entrées I_1, I_2, I_3 et I_4 apparaissent dans les relations de dépendance temporelle.

3.3.3 Génération du modèle

Une fois l'ensemble des relations de dépendance temporelle élaboré, le modèle NuSMV est obtenu à partir du programme du contrôleur et de la bibliothèque de blocs fonctionnels en réalisant les opérations suivantes :

1. Association des variables du programme aux variables du modèle.
   Seule les R-variables sont associées aux variables du modèle formel.
   L'expression des NR-variables est conservée pour les déductions logiques, comme expliqué dans la section 3.2.2.
   Dans le cadre du model-checker NuSMV, les R-variables seront donc associées à la catégorie VAR et les NR-variables à la catégorie $DEFINE$, si besoin.
2. Transformation des déclarations du programme en fonction des relations de dépendance temporelle.
   Ce deuxième point est réalisé en utilisant l'algorithme 3.8. Il permet la prise en compte des informations obtenues avec les relations de dépendance temporelle dans les affectations et de transformer les structures conditionnelles en de simples affectations.

for *chaque déclaration* S_i *de* Pr **do**
  switch *Si* S_i *est* **do**
    case *une affectation* ($V_i := expression_i$)
      for *chaque variable* V_k *dans l'expression*$_i$ **do**
        remplacer V_k par la variable indiquée ($V_{k,i}$ ou $V_{k,i+1}$) dans les dépendances temporelles Dp
    case *une structure conditionnelle (if cond ; then* $stmt_1$ *; else* $stmt_2$*)*
      for *chaque* V_k *dans cond* **do**
        remplacer $V_k$par la variable indiquée ($V_{k,i}$ ou $V_{k,i+1}$) dans les dépendances temporelles Dp
      for *chaque* V_m *affectée dans* S_i **do**
        Remplacer l'affectation de V_m par :
        "case
          cond : $\prec$ affectation de V_m dans prog_NuSMV_model($stmt_1$)$\succ$;
          !*cond* : $\prec$ affectation de V_m dans prog_NuSMV_model($stmt_2$)$\succ$;
        esac ;"

FIG. 3.8 – prog_NuSMV_model(Pr : **programme**, Dp : **dépendances temporelles**) → **modèle NuSMV**

3. Remplacement des blocs fonctionnels par leur modèle formel.
   Les appels de blocs fonctionnels sont remplacés par des modèles issus d'une bibliothèque préalablement remplie avec les méthodes présentés dans le chapitre 5. Dans le cas du bloc fonctionnel *RS* dans l'*exemple a*, la forme générique du modèle, avec deux entrées notées *Set* et *Reset* et une sortie notée Q, est :

$$
\begin{aligned}
&Next(Q) := case\\
&\qquad\qquad Reset\ :\ 0;\\
&\qquad\qquad Set\ :\ 1;\\
&\qquad\qquad 1\ :\ Q;\\
&\qquad esac;
\end{aligned}
$$

   Ce bloc fonctionnel *RS* est une mémoire à mise à 0 prioritaire. Nous pouvons effectivement constater dans cette modélisation que si l'argument *Reset* est vrai durant l'exécution de ce bloc fonctionnel alors cela entraîne forcement la mise à 0 de Q (premier choix de la structure conditionnelle case).

La figure 3.9 présente le résultat de l'application de ces étapes sur l'*exemple a*. Ce modèle présente seulement les affectations de chaque variable. La version complète de ce modèle, avec les déclarations de variables, les initialisations, ... est donnée en annexe B.

Mis à part l'efficacité de cette représentation formelle en terme de vérification, qui est quantifiée dans le chapitre 4, nous pouvons remarquer que le nombre d'équations pour décrire l'*exemple a* est faible : seulement 3 équations pour représenter un programme qui contient 6 affectations alors que le modèle se basant sur la représentation de [dSR02] en contient 11.

Toute la démarche de modélisation présentée dans cette section peut être synthétisée en un algorithme général, effectuant une analyse à la volée des relations de dépendances.

– modèle minimal nécessaire :

$Next(O_3)$:=*case*
$Next(I_1) \mid Next(I_2)$: $Next(I_3)$ & !($Next(I_4)$);
!($Next(I_1) \mid Next(I_2)$) : O_3;
esac;
$Next(O_4)$:= *case*
$Next(I_1)$: 0;
O_5 : 1;
1 : O_4;
esac;
$Next(O_5) := Next(I_3)$ & $Next(I_4)$ & $Next(O_4)$;

– Si besoin dans les propriétés :

$DEFINE\ O_2 := Next(I_3)$ & $Next(I_4)$;
$DEFINE\ O_1$:=!($Next(I_2) \mid Next(I_4)$);

FIG. 3.9 – Modèle NuSMV du programme de l'*exemple a*

Cet algorithme est présenté dans l'annexe C avec un exemple d'application.

Chapitre 4

Validation et quantification expérimentales de l'efficacité des abstractions

Après avoir proposé, dans le chapitre précédent, deux abstractions visant à améliorer l'efficacité des modèles pour la vérification, nous allons dans ce chapitre chercher à quantifier de cette efficacité.
Nous introduirons dans la première section des critères permettant d'évaluer ou estimer l'efficacité d'une représentation formelle. Dans les trois sections suivantes, nous quantifierons notre proposition, tout d'abord en la comparant à des représentations antérieures, puis en la confrontant à un réel problème de taille et de complexité industrielle.

Sommaire

4.1 Discussion sur l'efficacité des modèles pour la vérification par model-checking

Tout au long de ce mémoire, nous avons eu pour objectif de fournir des modèles de contrôleurs industriels utilisables dans des outils de model-checking pour vérifier des propriétés extrinsèques. Ceci implique bien sûr de conserver toute la sémantique des contrôleurs industriels nécessaire à la vérification de cette classe de propriétés, mais surtout de pouvoir réaliser cette vérification dans un temps raisonnable.

Pour cette raison, nous avons proposé dans le chapitre 3 deux abstractions visant à améliorer l'efficacité des modèles pour la vérification. Cependant, cette efficacité peut être évaluée selon plusieurs critères. Ceux-ci sont détaillés dans les deux sous-sections suivantes.

4.1.1 Efficacité théorie des graphes

Premièrement, l'efficacité est reliée aux *dimensions* du modèle. Comme nous nous intéressons ici à l'efficacité de la vérification, ces dimensions sont celle du modèle que *manipule* le model-checker. Seul comptent donc les *dimensions* de l'espace d'états atteignables du modèle, exprimé dans notre cas comme un Global Transition System (GTS) (voir chapitre 2).

Il existe plusieurs dimensions caractérisant un espace d'états atteignables, issues de la théorie des graphes [BM76] :

L'ordre *L'ordre d'un graphe est le nombre de ses sommets.*
Dans le cas d'un espace d'états issu d'un GTS, cela correspond au nombre d'états atteignables avec ce GTS.

Le degré *Le degré d'un sommet est la taille de son voisinage. Le degré d'un graphe est le degré maximum de tous ses sommets.*
Cela correspond au nombre maximum d'évolutions possibles à partir d'un état.

La taille *La taille d'un graphe est le nombre de ses arêtes.*
Dans le cas d'un espace d'états issu d'un GTS, cela correspond au nombre de transitions possibles entre les états.

Le diamètre *Le diamètre d'un graphe est la plus longue distance entre deux sommets de ce graphe, où la distance entre deux sommets est la longueur de la plus courte chaîne entre eux.*
Autrement dit, c'est le nombre maximum d'évolutions nécessaires pour aller d'un état vers un autre.

La figure 4.1 présente un exemple élémentaire de programme de contrôleur industriel ayant 2 entrées (I_1 et I_2) et 2 sorties (O_1 et O_2) et son modèle sous forme de GTS, dans le langage NuSMV, obtenu avec la représentation que nous avons proposé dans le chapitre 3.

La figure 4.2 présente espace d'états atteignables par le modèle GTS présenté sur la figure 4.1. Dans un état, le premier chiffre indique le numéro de l'état, la série de 4 chiffres en dessous indique l'état en fin de cycle des deux entrées et des deux sorties sous la forme

Programme de contrôleur :	Modèle NuSMV :
$O_1 := not(I_2)\ and\ (O_1\ or\ I_1);$	$next(O_1) := !next(I_2)\ \&\ (O_1 \mid next(I_1));$
$O_2 := O_1\ and\ not(O_2);$	$next(O_2) := next(O_1)\ \&\ !O_2;$

FIG. 4.1 – Exemple de programme et son modèle en NuSMV

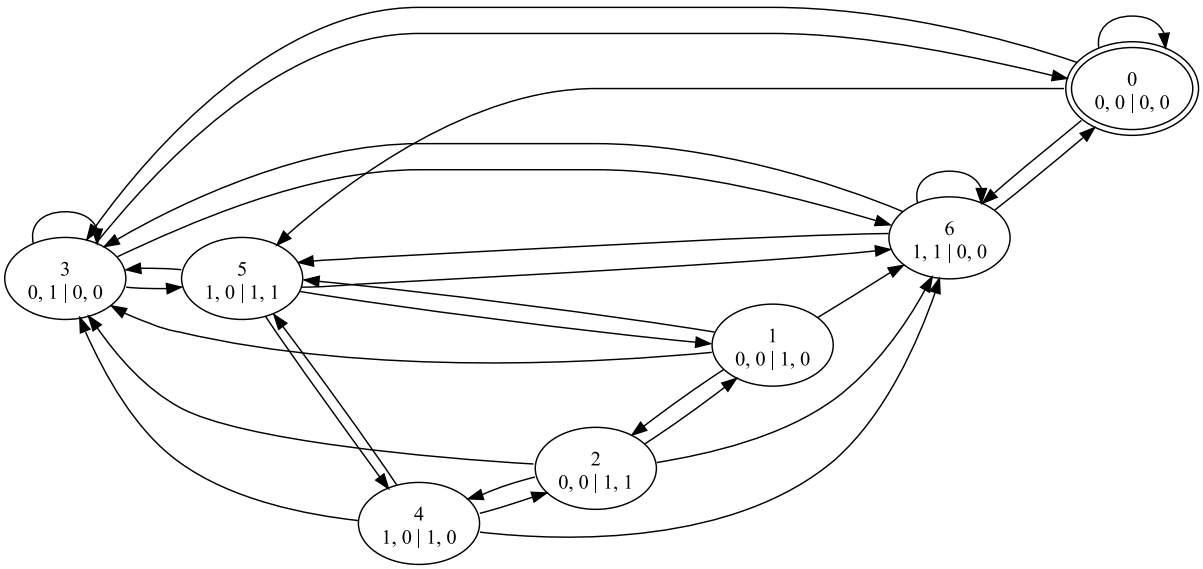


FIG. 4.2 – Espace d'états atteignables par le programme de la figure 4.1

"I_1, I_2 | O_1, O_2". L'état initial , état *0*, est considéré unique et correspondant à une valeur nulle de chaque variable.

Cet espace d'états atteignables possède donc les dimensions suivantes :
- Un ordre de 7, soit 7 états atteignables.
- Un degré de 4, soit 4 possibilités maximum d'évolutions par état, ce qui correspond aux 2 entrées (2^2).
- Une taille de 28, soit 28 transitions.
- Un diamètre de 3. Depuis tout état, on peut atteindre n'importe quelque autre état en au plus 3 évolutions. La distance maximum est atteinte entre l'état 6 et l'état 2.

Cet exemple simple nous permet d'illustrer le fait que, pour l'espace d'états représentatif d'un contrôleur logique :
- degré et taille dépendent principalement du nombre d'entrées,
- alors qu'ordre et diamètre dépendent de la nature des variables de sortie (R ou NR-variables).

En effet, les possibilités de transition à partir d'un état donné (degré de l'état) et le nombre total de transitions (taille de l'espace) représentent l'indéterminisme du modèle. Or dans le cas de modèles de contrôleurs industriels, seules les entrées[3] sont indéterministes.

A contrario, le nombre total d'états (ordre) et la longueur du plus long chemin entre deux états (diamètre) dépendent du nombre de variables qu'il est nécessaire de mémoriser ou, en d'autres termes, de la nature plus ou moins combinatoire (à l'opposé, séquentielle) du contrôleur. A titre d'exemple, un contrôleur à deux sorties combinatoires donne lieu à un espace d'états d'ordre 4 alors que l'exemple de la figure 4.2, où les deux variables de sorties sont des R-variables, conduit à un espace d'états d'ordre 7.

Concernant la vérification, seules 3 dimensions sont vraiment importante : l'ordre, la taille et le diamètre. Plus l'ordre est élevé, plus le model-checker devra explorer d'états lors de la vérification. Plus la taille est élevée plus le model-checker devra tester d'évolutions, à chaque état, pour identifier si un nouvel état est possible. Enfin, plus le diamètre est important plus le model-checker devra faire d'itérations pour *construire* l'espace d'états, sachant qu'à chaque itération le model-checker teste toutes les transitions disponibles.

4.1.2 Efficacité et model-checking symbolique

Un model-checker symbolique est un model-checker qui utilise une ou plusieurs abstractions symboliques, comme le BDD [Bry86] ou l'ADD [BFG+97], automatiquement et sans intervention de l'utilisateur. Un de ses principaux représentants est l'outil NuSMV, utilisé dans nos travaux.

L'abstraction symbolique des données permet une représentation extrêmement compacte d'un espace d'états en ne faisant que le caractériser sans l'évaluer. Autrement dit, chaque état n'est pas connu explicitement mais est contenu dans la structure qui le représente : le BDD, pour les systèmes logiques.

[3] L'indéterminisme est aussi dû au temps, mais nous verrons dans le chapitre 5 que chaque référence à un intervalle de temps peut être ramenée à l'introduction d'une variable d'entrée supplémentaire.

Par exemple, la figure 4.3 présente une représentation sous forme de BDD du modèle donné figure 4.1. Sur cette figure, chaque *étage* correspond à une variable qui influence le calcul de la variable représentée. A chaque étage, plusieurs nœuds sont possibles, chacun correspondant à un choix. Le choix est binaire, 0 ou 1, représenté respectivement par un trait pointillé ou non.

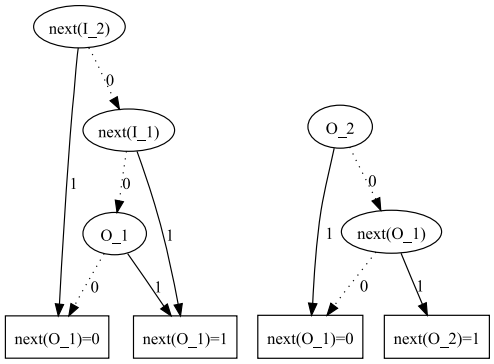


FIG. 4.3 – BDD représentant les variables O_1 et O_2 du modèle donné figure 4.1

La vérification sur un model-checker symbolique se décompose en deux étapes : la construction des BDD puis l'utilisation de ces BDD pour la vérification. Les BDD sont obtenus à partir de l'expression de chaque variable ce qui peut prendre un temps important dans le cas d'expressions complexes notamment si l'ordre des étages est *mal* choisi (comme l'exemple figure 4.4). En effet, suivant l'ordre des étages pour représenter un BDD, le nombre de choix et de noeuds peut beaucoup varier. Ce concept est propre à la représentation en BDD et a donné lieu à de nombreuses recherches [RAB+95] pour déterminer, a priori, quel est le meilleur ordre des tests des variables intervenant dans l'expression.

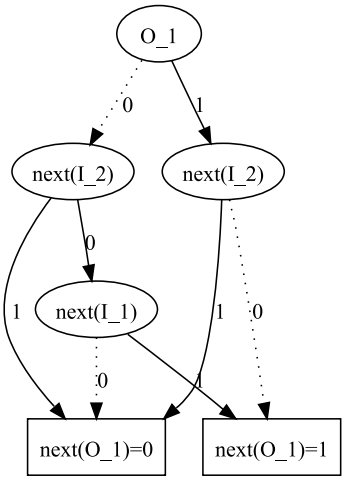


FIG. 4.4 – Exemple d'ordre d'étage mal choisi pour la variable O_1

L'utilisation des BDD pour la vérification se fait de manière itérative à partir de l'état initial. A chaque pas de calcul, tout les BDD sont évalués pour caractériser (et non évaluer)

tous les nouveaux états accessibles à partir de l'ensemble d'états précédents et les analyser selon la prorpiété à prouver.

L'efficacité en vérification d'un modèle formel utilisé par un model-checker symbolique est difficile à quantifier car elle ne dépend plus directement des dimensions données dans la sous-section précédente. L'exemple simple suivant suffit à nous en convaincre. Un programme avec seulement 500 entrées logiques et aucune sortie correspond à un espace d'états du modèle le représentant de 2^{500} états et $(2^{500}-1)*2^{500}$ transitions. Cependant, tous les états sont seulement un produit de toutes les évolutions *indépendantes* des entrées. De ce fait, la représentation sous forme de BDD se réduit à 500 arbres à deux choix, soit un modèle assez simple.

L'efficacité d'une représentation analysé par un model-checker symbolique dépend donc :

- de son diamètre, nombre itérations maximum du model-checker pour caractériser tout les états, sachant qu'a chaque itération le model-checker évalue tous les BDD.
- et les trois caractéristiques du mode de résolution :
  - nombre de BDD nécessaires pour représenter l'espace d'état.
    Il est lié au nombre de variables dans le modèle et donc au nombre d'états maximum atteingnables par le modèle. Dans notre exemple, figure 4.3, il est de 2.
  - nombre d'étages de choix du BDD.
    C'est le nombre maximum de choix pour obtenir le résultat. Il est donc directement lié à l'expression et à la relation de dépendance (comme celles présentées dans la section 3.3) de chaque variable. Sur figure 4.3, il est de 3 et 2 pour les variables O_1 et O_2.
  - nombre de noeud.
    Suivant l'ordre des étages, le nombre de noeud peut beaucoup varier, et donc influencent l'efficacité

Les critères présentés ci-dessus influent tous sur l'efficacité d'une représentation. Cependant, il n'existe pas à notre connaissance de formule et/ou méthode permettant de tous les obtenir à partir de la donnée du GTS, et d'en déduire une grandeur représentative de l'efficacité du GTS lors de la vérification. De ce fait, l'efficacité d'un modèle peut seulement être connue grâce à l'utilisation de ce modèle avec un model-checker symbolique.

Les sections suivantes présentent 3 cas d'étude visant à quantifier l'efficacité de notre représentation, basée sur deux abstractions. Les critères utilisés pour ce faire sont :

- l'ordre et le diamètre de l'espace d'états, ainsi que le nombre de variables à mémoriser, pour le premier cas d'étude.
- les performances en termes de temps d'obtention d'un résultat de preuve et de mémoire nécessaire à cette preuve, pour le second cas d'étude ;
- l'ordre de l'espace d'états, le nombre de variables à mémoriser, ainsi que le temps d'exploration de l'espace d'états, pour le dernier cas d'étude.

4.2 Premier cas d'étude

Ce cas d'étude reprend l'*exemple a* donné dans la section 3.1 afin de comparer la représentation décrite dans ce mémoire à celles proposées dans [dSR02] et dans [GdSF06b],

cette dernière utilisant seulement seulement l'abstractions d'interprétation. Le tableau 4.1 présente les résultats obtenus avec ces trois représentations.

Représentation proposée dans	États atteignables	Variables à mémoriser	Diamètre de l'espace d'états
[dSR02]	314 sur 14336	7	22
[GdSF06b]	21 sur 512	5	3
ce mémoire	20 sur 128	3	3

TAB. 4.1 – Comparaison de l'efficacité sur la base de l'*exemple a*

Afin de vérifier l'équivalence de ces trois représentations une analyse de l'équivalence de comportement a été faite en adoptant la méthodologie décrite dans [GdSF06a] et qui sera détaillé dans la section 6.3 : quelle que soit la séquence d'entrée, les deux modèles émettent la même séquence de sortie. Les critères retenus pour la comparaison sont :

le nombre d'états atteignables
: X sur Y correspond à X états réellement atteignables sur les Y possibles (combinaison de toutes les valeurs possibles des variables) ;

le nombre de variables à mémoriser
: indique le nombre de variables que le model-checker doit mémoriser à chaque évolution, sans compter les variables d'entrée ;

le diamètre du système
: Comme auparavant, ce diamètre indique le nombre maximum d'itération que doit effectuer le model-checker pour aller d'un état à un autre. Ce critère permet également d'estimer la taille maximum des contre-exemples fournis en cas de preuve négative. Un contre-exemple sera plus court avec notre représentation, ce qui en facilitera l'analyse.

Ces résultats montrent clairement que la représentation proposée permet de réduire l'espace d'états, le nombre de variables à mémoriser et le diamètre du système. Il est importe de souligner que, par comparaison avec une représentation usuelle incluant tout les états intermédiaires liés au traitement ainsi que toutes les variables de sorties, notre proposition permet sur cet *exemple simple* :

- une diminution de l'ordre de l'espace d'états de plus d'un ordre de grandeur (1,2 environ),
- une diminution du diamètre d'un facteur 7 environ.

Il est intéressant de remarquer également que la représentation de [dSR02] oblige à utiliser des variables représentant le cycle du contrôleur et l'avancement dans le programme et entraîne une modélisation par sept variables alors que le programme ne compte que cinq variables de sortie. Notre représentation ne nécessite pas ces variables. De plus, le modèle basé sur les 2 abstractions ne nécessite de mémoriser que trois variables (les R-variables) contre 5 dans [GdSF06b]. La diminution seule du nombre de variables entraîne aussi une diminution faible du nombre d'états atteignables. En effet, la diminution du nombre de variables permet de réduire le nombre d'états initiaux, lors de la vérification.

De plus, certaines évolutions du modèle proposé dans [GdSF06b] sont alors transformées en déductions logiques.

4.3 Deuxième cas d'étude

Ce deuxième cas d'étude a pour but de déterminer les performances en terme de mémoire et temps dans le cas de vérification de propriétés de sûreté et de vivacité. Cette expérimentation est basée sur le système présenté et vérifié dans [dSR02] : un contrôleur de système Fischertechnik. Les mêmes vérifications ont été réalisées afin de comparer, dans le tableau 4.2, les performances des trois représentations en utilisant le model-checker NuSMV, version 2.3.1, sur un PC P4 3.2 GHz, avec 1 Go de RAM, sous Windows XP.

Représentation proposé dans	Propriétés de vivacité	Propriétés de sûreté
[dSR02]	5h / 526Mo	20min / 200Mo
[GdSF06b]	2s / 8Mo	2s / 8Mo
ce mémoire	0.3s / 8Mo	0.3s / 8Mo

TAB. 4.2 – Temps et mémoire requis pour la vérification de propriétés

Les gains réalisés sont extrêmement significatifs : un facteur de 4 000 à 60 000 pour le temps de vérification et de 40 à 60 pour la mémoire utilisée par rapport à la représentation de [dSR02] et suivant le type de propriété vérifiée. La diminution seule du nombre de variables à mémoriser par le model-checker, par rapport à [GdSF06b], permet une amélioration du temps de vérification d'un ordre de grandeur environ, le minimum de mémoire mesurable étant déjà atteint avec la représentation précédente.

4.4 Troisième cas d'étude

Le troisième cas d'étude se place dans le contexte industriel d'Alstom Power et concerne le contrôle d'une centrale électrique thermique. Ce contrôle est réalisé avec 175 contrôleurs logiques connectés en réseau. L'objectif de cette expérimentation est de déterminer, pour chaque programme utilisé dans les contrôleurs logiques, l'espace d'états atteignables (ordre) et donc de quantifier la représentation proposée pour des cas industriels. En effet, s'il est possible de déterminer cet espace d'états alors il est aussi possible de vérifier toutes les propriétés sur cette représentation sans rencontrer de problèmes d'explosion combinatoire.

Les statistiques de représentation pour cette modélisation sont données dans le tableau 4.3. Trois points méritent d'être soulignés :

– Les programmes sont de complexité importante, comme l'indique la quatrième ligne du tableau. Cependant, la somme des temps de détermination de tous les états des programmes, donnée en ligne 6, est très raisonnable et compatible avec des contraintes industrielles.

Nombre de programmes	175
Nombre de variables de sortie	max : 47 min : 1 somme : 1822
Nombre de R-variables	max : 18 min : 1 somme : 238
Nombre de variables d'entrée	max : 50 min : 2 somme : 2317
Ordre de l'espace d'états d'un programme	max : 8.10^{28} min : 10^5 moyenne : 5.10^{26}
Temps de détermination de l'espace d'états de tous les programmes	1 sec
Temps total de modélisation	50 sec

TAB. 4.3 – Statistiques de représentation de programmes industriels

– L'abstraction de données permet une forte diminution (d'un facteur 8 si l'on considère la somme) du nombre de variables nécessaires à la vérification, comme le montrent les lignes 2 et 3.
– La dernière ligne indique le temps de modélisation, du programme industriel vers le modèle NuSMV, par l'outil logiciel développé dans le cadre de nos travaux et décrit dans la section 6.1.

4.5 Conclusion

Nous avons proposé dans le chapitre 3 deux abstractions visant à améliorer l'efficacité des modèles formels des contrôleurs logiques où l'efficacité est la possibilité de vérifier des programmes de grande taille et complexité dans des temps raisonnables.

Nous avons montrés dans la section 4.1 que cette complexité était fonction des caractéristiques de l'espace d'états du modèle formel, ainsi que de celles des BDD le codant, lors qu'un model-checker symbolique est utilisé. Nous avons montré que seule l'expérimentation permettait d'évaluer l'efficacité.

Les cas d'étude présentés dans les trois sections précédentes montrent clairement que l'utilisation de nos deux abstractions permet de réaliser des modèles beaucoup plus efficaces quelles que soient les caractéristiques considérées (avec un facteur allant de 7 à 60000).

De ce fait, les modèles utilisant nos deux abstractions peuvent être qualifiés d'*utilisables par l'industrie*. En effet, maintenant que les contrôleurs logiques contenant des programmes de taille industrielle sont vérifiables sur les outils de model-checking actuel (comme présenté dans la section 4.4), ces modèles peuvent contribuer à la certification de ces contrôleurs.

De plus, dans le cadre de nos travaux, nous avons développé un traducteur permettant de générer automatiquement ces modèles. Cet outil sera détaillé au chapitre 6.

Seul un point reste à traiter avant de pouvoir utiliser nos propositions pour la vérification à des fins de certification : la définition de la bibliothèque de modèles des blocs

fonctionnels utilisés dans les algorithmes du chapitre 3. Ceci est donc l'objet du chapitre 5.

Chapitre 5

Représentation et formalisation de blocs fonctionnels

La programmation de contrôleur industriel dans un des langages de la norme CEI 61131-3 est fortement dépendante de l'utilisation de blocs fonctionnels. Ceux-ci permettent une écriture hiérarchique du programme en y plaçant une partie du comportement attendu.
Ceci est d'autant plus vrai lorsque le contrôleur comporte des blocs fonctionnels *utilisateurs* encapsulant un savoir-faire métier complexe. La vérification formelle d'un contrôleur requiert cependant que le comportement de tous les blocs fonctionnels qu'il intègre soit formellement défini.
Ce chapitre propose donc une représentation formelle pour des blocs fonctionnels devant être intégrés dans des contrôleurs en langages CEI 61113-3.
Il comporte quatre sections : la première présente le contexte et les besoins industriels en termes de représentation de blocs fonctionnels. La seconde section brosse un état de l'art des propositions industrielles et universitaires. A l'issue de cette analyse, nous proposerons, dans la troisième section, un langage de description de blocs fonctionnels adapté à nos besoins. Enfin, la quatrième section permet de doter ce langage d'une sémantique formelle, ce qui autorise la vérification des blocs fonctionnels qui l'utilisent.

Sommaire

5.1 Besoins et pratiques industriels

Un Bloc Fonctionnel (BF) est un élément de programme prenant en entrée des arguments et émettant des sorties. Le but d'un BF est de :

- encapsuler de la connaissance métier (régulation, commande d'actionneur complexe tel qu'un alternateur, ...) ;
- permettre une programmation hiérarchisée ;
- faciliter la maintenance du code ;
- permettre la réutilisation.

Comme présenté dans le chapitre 1, la norme CEI 61131-3 prend en compte ce besoin en proposant plusieurs BF *basiques* ou standard. Leur définition est faite sous la forme de chronogrammes et d'énoncés en langage naturel. Ces définitions manquent donc de formalisme car le comportement attendu est rarement complètement donné. Un exemple simple suffira à nous convaincre : la définition du bloc fonctionnel TON (temporisation à l'activation) n'indique pas le comportement en cas de modification du délai de temporisation pendant le décompte du temps. De plus, la norme CEI 61131-3 ne propose aucune méthode rigoureuse pour définir de *nouveaux* BF.

Le premier but d'un BF est lié au processus à contrôler alors que les trois derniers buts concernent la programmation de fonctions de contrôle. Nous pouvons, en effet, identifier deux publics qui peuvent être amenés à définir et implanter des BF qualifiés d'*utilisateur*, car ils encapsulent de la connaissance métier spécifique à un domaine d'utilisation (contrôle de turbine, démarrage de moteur, ...) :

- les développeurs d'applications d'automatismes.
- les développeurs d'outils informatiques pour le développement d'applications d'automatismes.

Les premiers ont une vue *extérieure* du comportement des BF mais manquent beaucoup de rigueur. La spécification est alors souvent floue, définie en langage naturel et en omettant des comportements aux limites. Notamment, les défaillances du logiciel de contrôle-commande sont rarement prises en compte.

Les deuxièmes sont fortement dépendants du matériel et du langage et ont une vue *interne* du comportement des BF. Ce comportement est alors spécifié par rapport à la séquence de traitements à réaliser plutôt que par rapport aux résultats voulus.

Nous pouvons donc identifier un besoin essentiel : la définition d'une sémantique *formelle* des BF, commune aux différents publics. Cette préoccupation a fait l'objet de plusieurs propositions antérieures. La section suivante présente un état de l'art de ces propositions industrielles et universitaires.

5.2 Etat de l'art sur la représentation et la formalisation de blocs fonctionnels

5.2.1 Travaux autour de la norme CEI 61499

Ce problème de définition formelle du comportement est récurrent pour les langages à blocs fonctionnels et en particulier pour celui de la norme CEI 61499 [IEC04]. Cette norme définit des blocs fonctionnels pour les processus industriels de mesure et de contrôle de systèmes.

Même si elle répond à de nombreuses demandes industrielles pour la prise en compte simultanée de signaux et d'évènements, cette norme ne définit pas correctement le fonctionnement interne des blocs fonctionnels et les comportements aux limites, comme présenté par [CLA06, FV04]. Pour répondre à ce besoin de formalisation, plusieurs travaux ont permis d'aboutir à des représentations formelles de ces blocs fonctionnels afin de permettre :

- leur traduction dans des langages industriels existants [FRV06] ;
- leur vérification par model-checking [VH99, FLS02] ;
- de nouvelles méthodes de développement de logiciels [VKP05] ;
- la génération automatique de scénarios de tests [HF06] ;
- une conception générique de blocs fonctionnels [PF05] ;
- le diagnostic de fautes [KJH06].

Nous pouvons remarquer que tous les travaux effectués autour des BF de la norme CEI 61499 se font dans le sens de garder la syntaxe des blocs fonctionnels et d'apporter une sémantique plus formelle que celle donnée dans la norme. En effet, la syntaxe des BF dans la norme CEI 61499 a été définie pour répondre aux attentes industrielles ; il est donc inutile, voire maladroit, de la redéfinir. Par contre, cette norme ne propose qu'une sémantique vague pour chaque BF et pour chaque réseau à base de BF.

Comparée à la norme CEI 61499, la norme CEI 61131-3 propose une meilleure définition du comportement des blocs fonctionnels grâce à l'introduction de moniteurs d'exécution. Cependant, le comportement interne des blocs fonctionnels, basiques et utilisateurs, est seulement défini à l'aide de chronogrammes et d'énoncés en langage naturel. Deux difficultés apparaissent alors lors de l'utilisation de ces blocs fonctionnels :

- La spécification de comportement global d'un programme contenant des BF est difficile. Même si chaque élément de langage est bien défini, les programmes complexes sont difficiles à comprendre.
- La définition de nouveaux blocs fonctionnels utilisateurs, surtout dans un cadre industriel, est rarement formelle et donc sujette à erreurs.

Les deux sous-sections suivantes présentent deux travaux visant à répondre à ces difficultés.

5.2.2 La proposition NuSCR

La principale initiative universitaire relative à la spécification de programme utilisant des BF est la représentation en langage NuSCR (New Software Cost Reduction). NuSCR

est un langage de spécification formelle, initié par [Lev95], pour spécifier les exigences des logiciels de commande embarqués, notamment dans le domaine du nucléaire. Par exemple, la figure 5.1 présente une spécification NuSCR pour le RPS (Reactor Protection System) dans DPPS (Digital Plant Protection System) pour un réacteur développé en Corée.

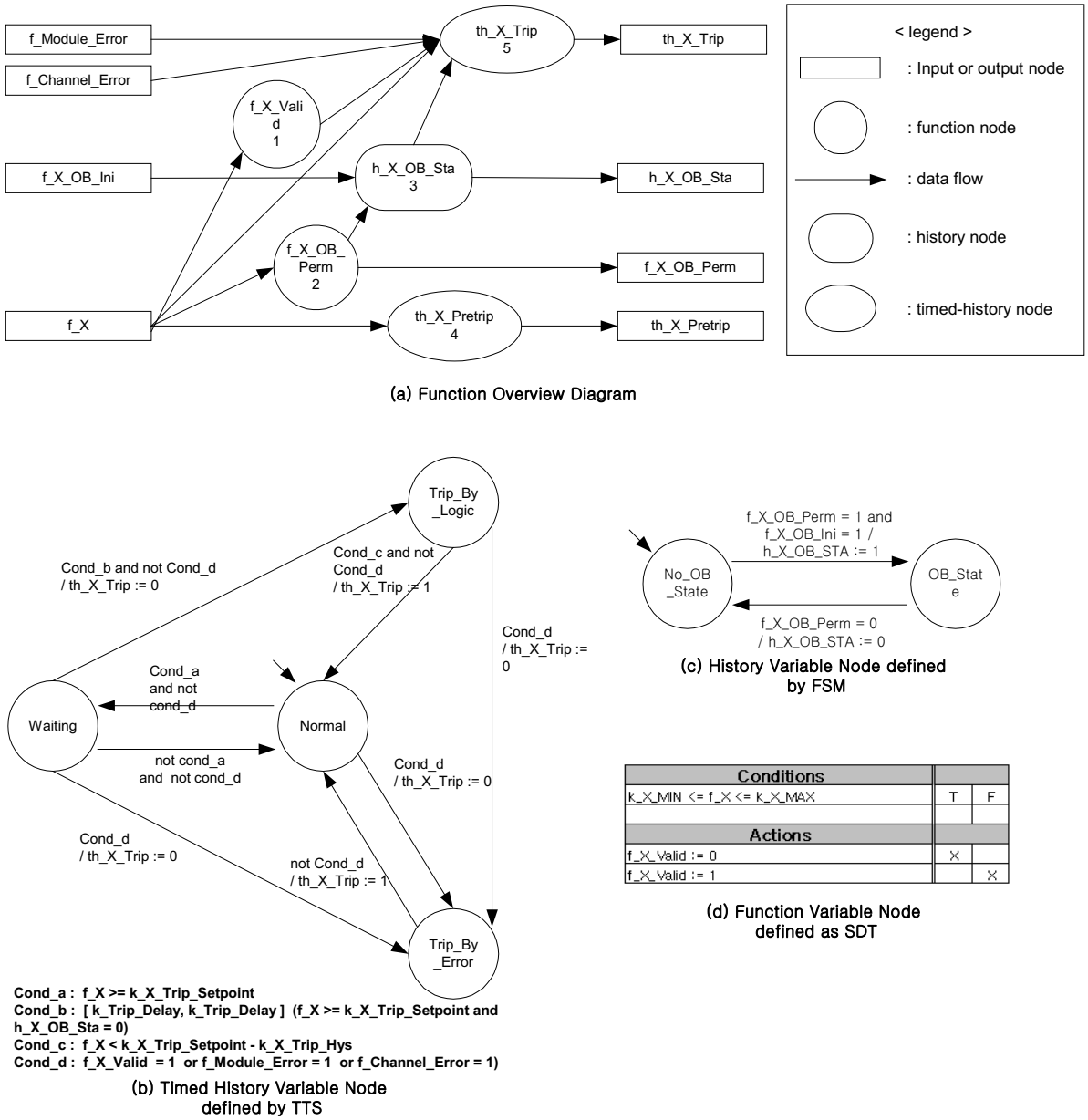


FIG. 5.1 – Exemple de modèles en langage NuSCR

Ce langage est composé de trois parties :

- une définition du comportement global avec un diagramme FOD (Function Overview Diagram), figure 5.1a) ;
- une notation tabulaire pour définir les opérations désirées (Structured Decision Table (SDT), figure 5.1d) ;

– des automates à états finis pour spécifier les comportements séquentiels (Finite State Machine (FSM) figure 5.1c) ou temporisés (Timed Transition System (TTS) figure 5.1b).

Ce langage est supporté par l'outil NuEditor [CYC04] qui permet de faire le lien avec d'autres langages formels et outils de vérification comme SMV.

Il a été utilisé par plusieurs chercheurs pour la norme CEI 61131-3. Notamment, [YCKS05] propose, à partir d'une spécification écrite en langage NuSCR, d'obtenir automatiquement un schéma (ou un ensemble de schémas dans un *POU*) correspondant en FBD, en utilisant uniquement des blocs fonctionnels basiques de la norme CEI 61131-3. L'objectif de ces recherches est d'obtenir le minimum de blocs fonctionnels pour remplir la fonction spécifiée en langage NuSCR.

Ce langage est particulièrement adapté pour spécifier le comportement de programme utilisant des blocs fonctionnels mais n'a pas été développé dans le but de développer de nouveaux blocs utilisateurs. Notamment, il manque de solution pour définir les interfaces des blocs fonctionnels (arguments, sorties, ...). Nous allons donc présenter maintenant une deuxième proposition dédiée à la définition de BF utilisateurs : le langage PLCopen.

5.2.3 La proposition PLCopen

Pour ce qui concerne la deuxième difficulté, la définition de nouveaux blocs fonctionnels utilisateur, peu de recherches universitaires ont été menées mais des initiatives industrielles ont vu le jour. C'est le cas du langage proposé par l'association PLCopen [vdW99].

PLCopen est une association internationale d'industriels offreurs de solutions d'automatismes. Son but est d'harmoniser la spécification et la conception des programmes industriels de contrôle-commande en standardisant les interfaces de programmation. Ces interfaces de programmation standardisées permettent la même interprétation quelle que soit la personne concernée. Ceci est d'autant plus important lors d'échanges entre les différentes phases du cycle de vie du programme : spécification, conception, implantation, tests, installation et maintenance.

PLCopen propose une représentation du comportement des blocs fonctionnels par des diagrammes à états. La figure 5.2 présente le diagramme à états du bloc fonctionnel *équivalent* S_EquivalentOut. Ce bloc a deux arguments d'entrée booléens (S_chanelA et S_chanelB et une sortie booléenne. Si les deux arguments sont équivalents, la sortie est vraie. Si l'un des arguments change, la sortie est fausse jusqu'à ce que l'autre argument fasse de même. Si un temps trop important sépare les deux variations un signal d'erreur (*ERROR*) est généré.

La syntaxe utilisée pour les diagrammes d'états est proche des automates à états de type machine de Moore avec quelques particularités :

– Une priorité est associée à chaque transition, indiquée à l'origine de chacune d'elle (chiffres cerclés sur la figure 5.2).
– Le temps physique est déclaré implicitement (*Wait for ...*).
– Trois zones de comportement définissent l'état global du bloc, séparées par des traits forts pointillés sur la figure 5.2. Elles correspondent à : *non-disponible* (Ready=False),

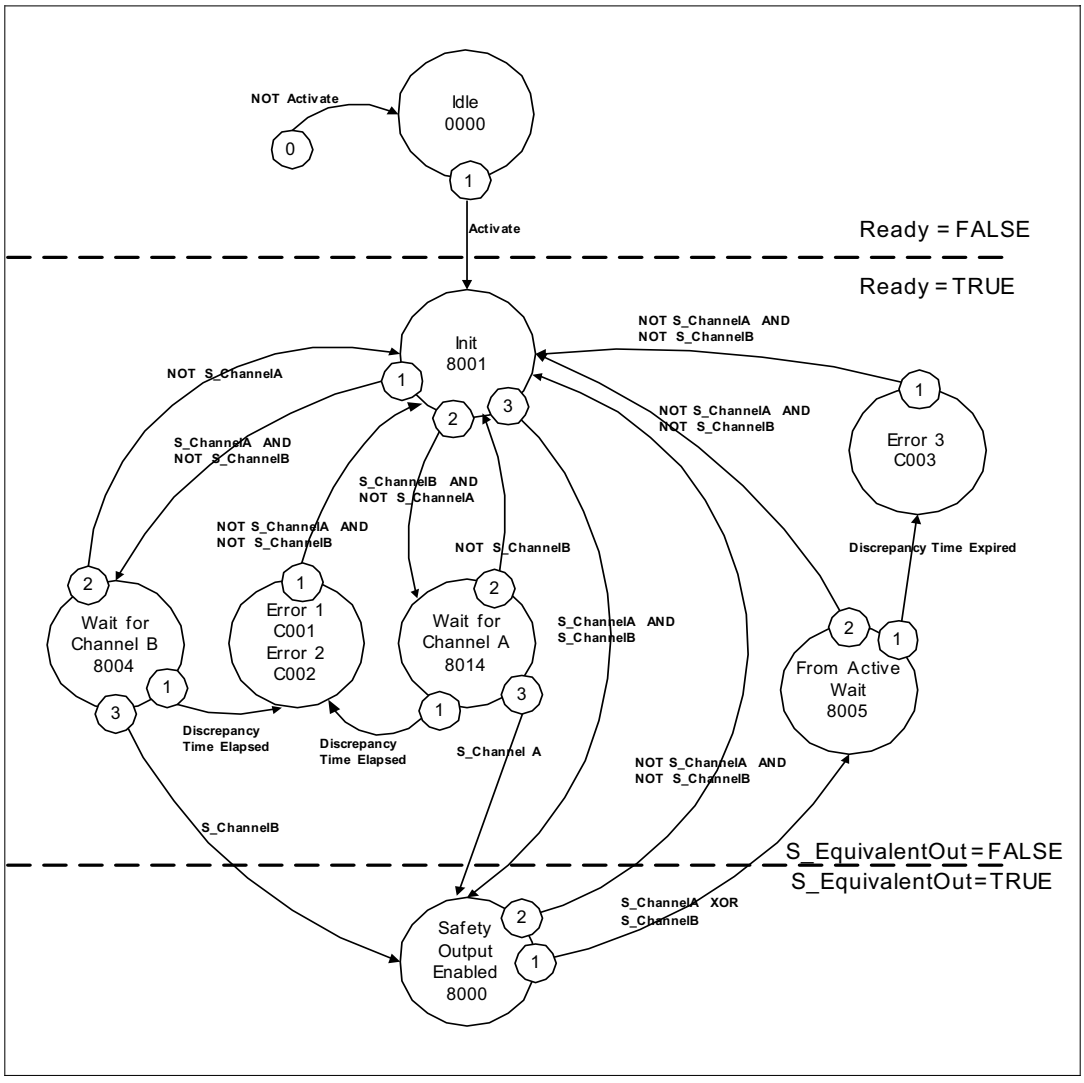


FIG. 5.2 – Un diagramme à états PLCopen

disponible et sortie non-valide (Ready=True) et *disponible* (Ready=True) et *sortie valide* (S_Equivalent=True).
- Chaque état est associé à un code (nombre indiqué dans les états) qui permet de le repérer.

La sémantique n'est pas définie ; seuls des exemples illustrent le comportement, comme celui de la figure 5.2. En particulier :
- Il n'est pas indiqué comment évolue ce diagramme (recherche de stabilité ou non, ordre de calcul des transitions et affectation, ...).
- Les affectations et mémoires nécessaires ne sont pas définies.
- L'initialisation des horloges est définie en langage naturel (*Wait for*)dans le diagramme.
- L'interface avec le programme et le moniteur d'exécution n'est pas indiquée (exécution parallèle, séquentielle, ...).

Ce manque de formalisme n'est pas gênant vis-à-vis de l'objectif du langage PLCopen : clarifier[4] le comportement voulu par les automaticiens. Cependant, si l'objectif est d'utiliser ces blocs fonctionnels sur des systèmes critiques, il devient nécessaire d'aller plus loin que la clarification et de viser la formalisation[5] du comportement.

Nous allons donc proposer dans ce qui suit un langage nommé AFBL (Alstom Function Block Language), s'inspirant de la proposition PLCopen, mais doté d'une syntaxe et d'une sémantique plus rigoureuses. Cette définition formelle de comportement permettra ensuite la vérification de ces blocs fonctionnels et des programmes qui les utilisent.

5.3 Représentation proposée

5.3.1 But de la représentation

La représentation AFBL présentée dans cette section permet de spécifier des Blocs Fonctionnels (BF) qui peuvent être par la suite utilisés dans Controcad. Cette spécification consiste à indiquer, suivant un formalisme prédéfini, des informations sur le BF, à savoir :
- son nom ;
- le type de variables ;
- le comportement normal attendu ;
- les différentes protections mises en place pour éviter ou limiter l'apparition d'erreurs dues à :
  - de mauvais arguments ;
  - des erreurs de calculs dans le BF ;
  - des débordements (par rapport à une norme ou à une capacité de calcul du contrôleur).
- d'autres conditions à respecter (sur les variables, calculs, ...) ;
- le comportement en mode dégradé et le retour vers un comportement normal ;
- les erreurs connues (absence de protection) dont l'évitement est laissé à la charge du concepteur d'application d'automatisme utilisant ce BF.

[4] par définition : qui vise à *RÉDUIRE* les ambiguïtés.
[5] par définition : qui est *SANS* ambiguïtés.

La spécification obtenue avec la représentation AFBL est ensuite utilisée pour concevoir le code du BF (phase de conception). Le cadre formel que nous proposons permettra donc (figure 5.3) :

- de doter chaque spécification d'un bloc fonctionnel d'un modèle formel ;
- de prouver l'équivalence comportementale entre le code et la spécification, la traduction du code du bloc fonctionnel se faisant selon la méthode développée au chapitre 3 ;
- de vérifier, par des techniques classiques, que cette spécification correspond bien au cahier des charges ;
- de vérifier des programmes de contrôleurs industriels utilisant ce BF.

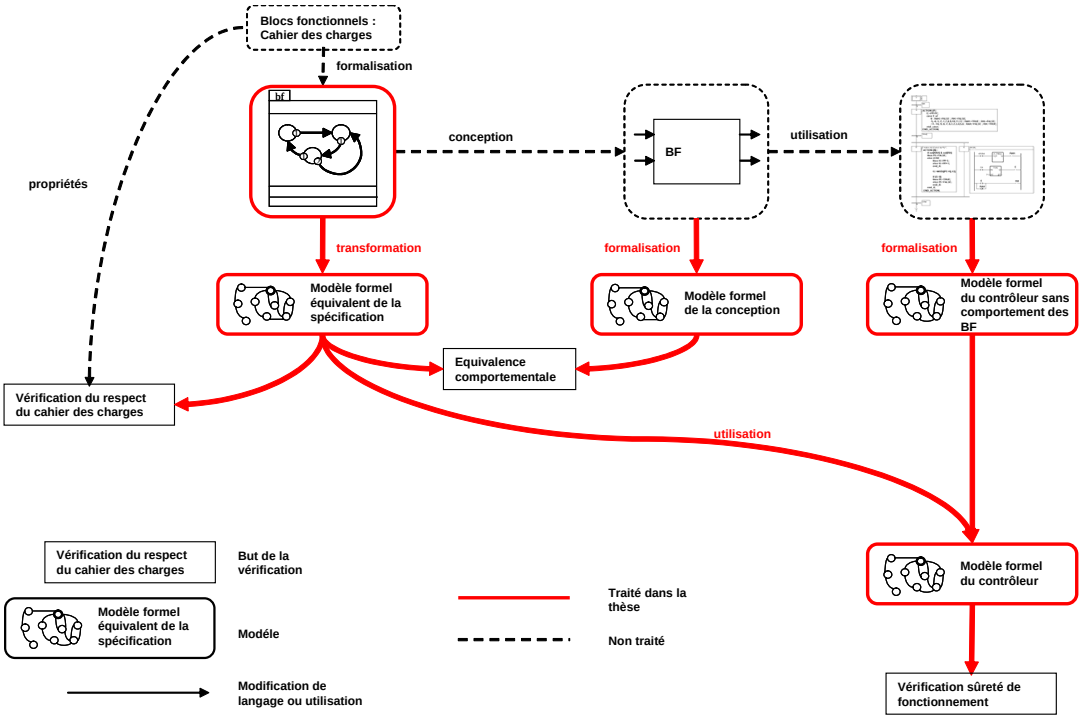


FIG. 5.3 – Utilisation d'un modèle formel de bloc fonctionnel

5.3.2 Syntaxe

La syntaxe du langage AFBL proposée dans cette section est inspirée du langage PLCopen original. Elle sera illustrée sur la base d'un BF présenté sur la figure 5.4 : un *Voteur 2 mesures haut niveau* (MES_2H) dont la définition globale donnée dans la spécification initiale est :
Ce bloc fonctionnel élabore une mesure au moyen d'un voteur 1/2 sur 2 mesures haut-niveau. Cette mesure est ensuite filtrée à l'aide d'un filtre exponentiel du 1er ordre. La mesure est ensuite mise à l'échelle Normée [0 - 10000] avant d'être délivrée.

Le tableau 5.1 présente les arguments d'entrée, les sorties de ce BF. Il a 11 arguments, dont les deux mesures analogiques à comparer (M1 et M2) et leur invalidant booléen (V1

et V2). Le reste des arguments permet la configuration des différentes fonctions. Ce BF retourne trois sorties : une sortie analogique R, issue des deux mesures, et deux sorties booléennes (1D et 2D) indiquant les défauts.

Ce BF réalise donc trois fonctions d'automatisme :

Un voteur 1/2 En mode de fonctionnement normal, le voteur retourne la valeur moyenne, le maximum ou le minimum (en fonction de la valeur de la variable TYP) des deux mesures (M1 et M2).
Trois modes dégradés sont prévus :
- Deux modes dégradés correspondant à une seule des deux mesures valides (reflété par la variables V1 ou V2). Dans ce cas, seule la mesure valide est retournée.
- Si l'écart entre ces deux mesures est supérieur à EM ou si les deux mesures sont invalides, la valeur précédente de la sortie R est retournée.

Le retour d'un mode dégradé vers le mode normal est temporisé : les causes du mode dégradé doivent disparaître pendant un temps minimum (TMP1 et TMP2) avant que le BF ne repasse en mode normal.

Un filtre du 1er ordre La sortie du voteur est filtrée avec un filtre d'ordre 1 et de constante Tf.

Une mise à l'échelle La valeur de la sortie est adaptée à la plage [0-10000].

Interface	Type	Commentaires	Symbole
Argument	Booléen	Invalidant mesure 1	V1
Argument	Booléen	Invalidant mesure 2	V2
Argument	Booléen	Initialisation du système	SYS_Init
Argument	Entier	Cadence TRX	SYS_Cad_TRX
Argument	Entier	Valeur initiale de la temporisation V1	TMP1
Argument	Entier	Valeur initiale de la temporisation V2	TMP2
Argument	Entier	Écart maximum entre 2 mesures	EM
Argument	Entier	Choix de fonctionnement	TYP
Argument	Entier	Constante du filtre	Tf
Argument	Entier	Mesure 1	M1
Argument	Entier	Mesure 2	M2
Sortie	Booléen	Premier défaut	1D
Sortie	Booléen	Deuxième défaut	2D
Sortie	Entier	Résultat du vote filtré et normalisé	R

Tab. 5.1 – Liste des arguments, sorties du BF MEM_2H

La spécification initiale, donnée dans [Y3-04], ne peut pas être présentée dans ce mémoire pour des raisons de confidentialité. Mais celle-ci est principalement le reflet du code de ce BF et a été réalisée par les développeurs de Controcad. Cette spécification est donc le reflet du code et non de son comportement en tant que fonction d'automatisme. Nous allons présenter dans la suite de cette section les éléments d'une syntaxe plus adaptée aux automaticiens : celle du langage AFBL.

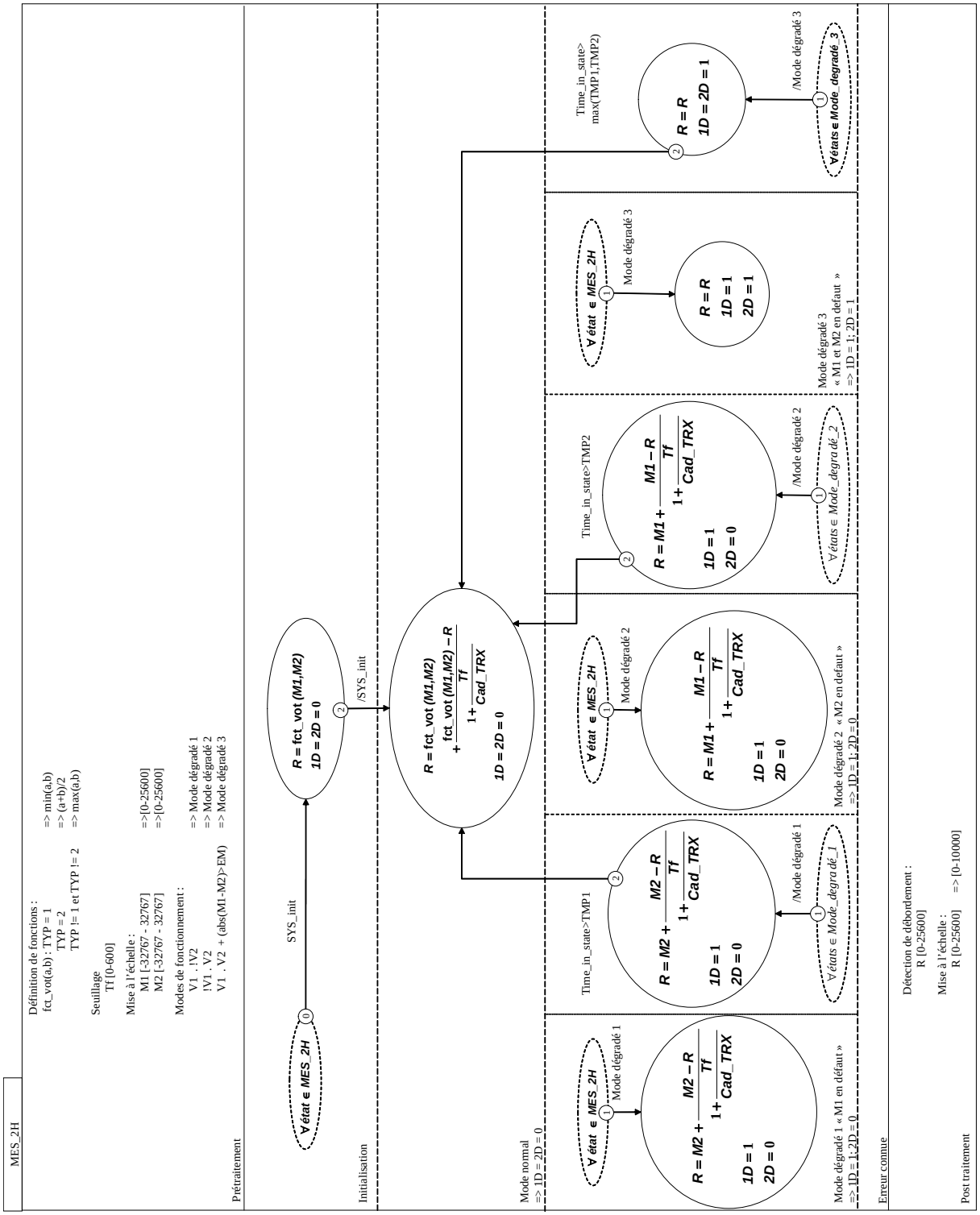


FIG. 5.4 – Vue d'ensemble du bloc fonctionnel MES_2H

5.3.2.1 Organisation d'un BF

La syntaxe d'un BF (figure 5.5) est organisée en trois parties :

le pré-traitement. Il permet la mise en forme des variables d'entrée et le choix des modes de marche. Sa syntaxe est présentée dans la section 5.3.2.2 et sa sémantique dans la section 5.3.3.2.

le comportement. Il définit le comportement normal et dégradé du BF avec un diagramme à état. Cette section se subdivise elle-même en trois zones : initialisation, modes nominaux (un seul dans l'exemple) et modes dégradés. Sa syntaxe globale est présentée dans la section 5.3.2.3 et sa sémantique dans la section 5.3.3.3.

le post-traitement. Il décrit la mise en forme des variables de sortie et les protections liées aux calculs. Sa syntaxe est présentée dans la section 5.3.2.4 et sa sémantique dans la section 5.3.3.4.

Ces parties[6] sont exécutées séquentiellement lors de l'appel du BF et ont chacune une syntaxe propre, présentée dans ce qui suit.

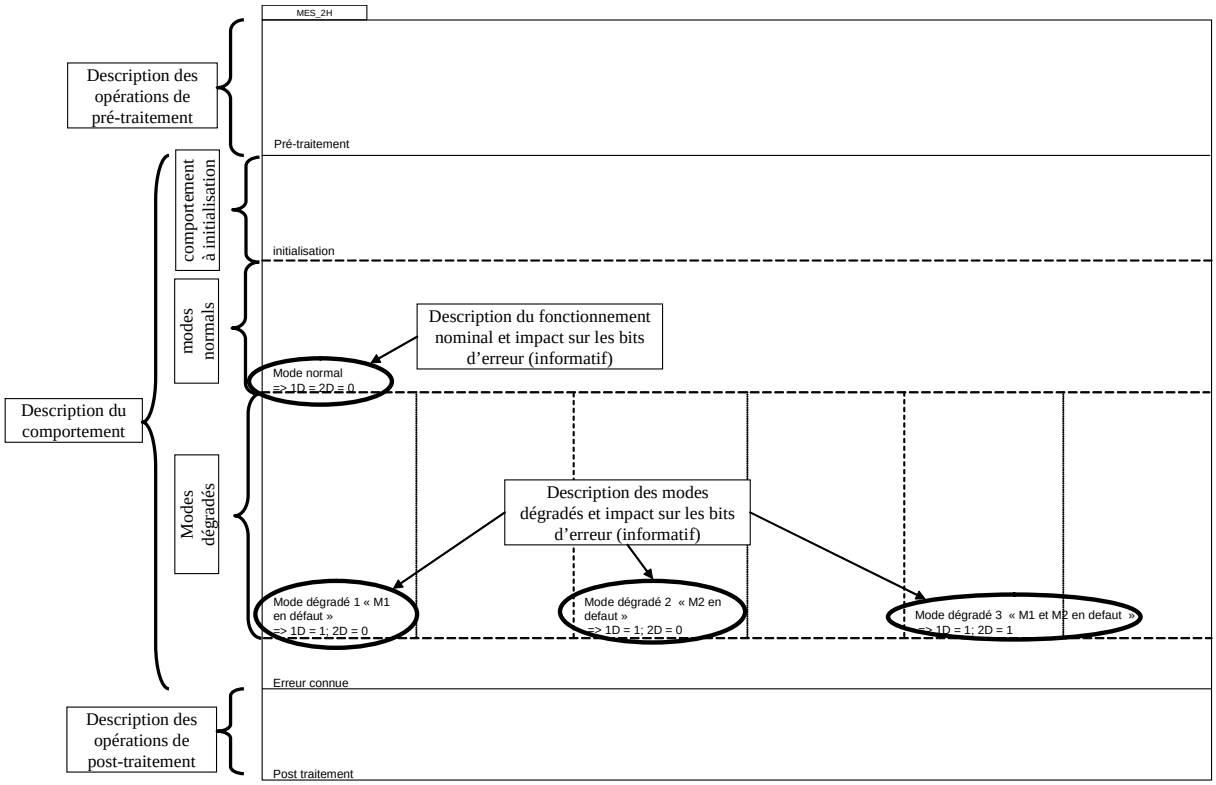


FIG. 5.5 – Structure générale d'un bloc fonctionnel AFBL

5.3.2.2 Pré-traitement

La syntaxe du pré-traitement est présentée sur la figure 5.6. Elle contient :

[6]La zone *Erreur connue* sert à renseigner l'utilisateur du BF sur des absences de protection contre les arguments erronés ou les débordements lors des calculs. Elle ne sera pas détaillée dans ce qui suit

Définition de fonctions. Afin d'alléger la représentation, des fonctions peuvent être définies à ce niveau. Le résultat d'une fonction peut dépendre d'une condition booléenne, comme c'est le cas dans la figure 5.6.

Mise en forme des variables d'entrée. Deux types de mises en forme de variables d'entrée sont possibles : le seuillage et la mise à l'échelle (dont la sémantique est définie dans la section 5.3.3). Ces mises en forme dépendent de plages de valeur dont la syntaxe comporte 2 entiers séparés par un tiret indiquant le minimum (à gauche) et le maximum (à droite) de la plage.

Choix des modes de fonctionnement. En fonction d'une condition exprimée sous la forme d'une expression booléenne, plusieurs modes dégradés peuvent être définis. A chaque mode dégradé est associé un nom (ici, mode dégradé 1, mode dégradé 2, ...) qui sera utilisé dans l'automate à états de la partie *Comportement*.

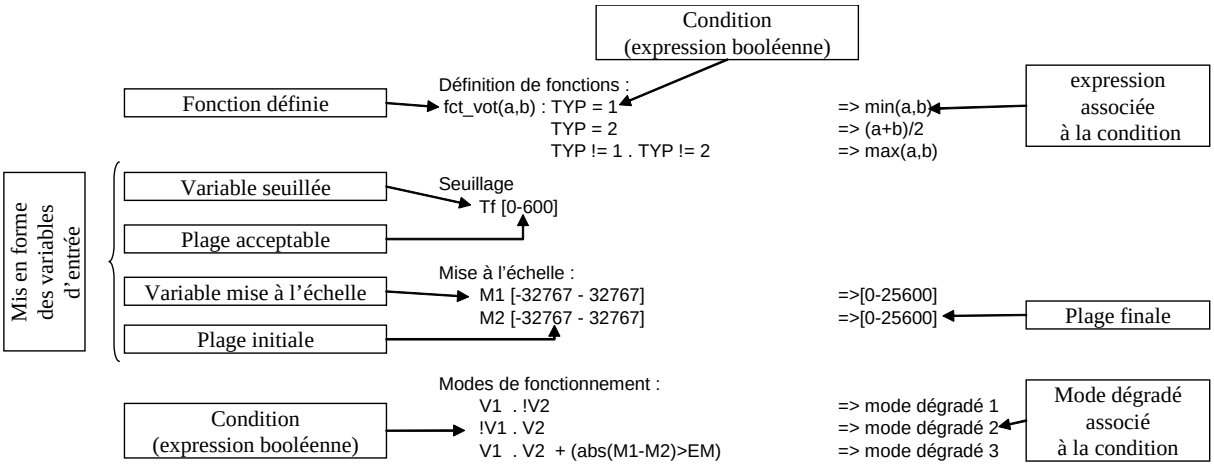


FIG. 5.6 – Syntaxe du pré-traitement

5.3.2.3 Description du comportement par automates à états

La partie *Comportement* est le point central du BF, à la fois en terme de place et d'importance. Dans une syntaxe proche de celle de PLCopen, cette partie décrit le comportement attendu du BF (figure 5.7). Elle se représente sous la forme d'un automate à états. Dans chaque état, une action de type calcul peut être effectuée (sans obligation). Le passage d'un état à l'autre est réalisé par des transitions caractérisées par leurs gardes et leurs priorités.

Les gardes sont des conditions booléennes pouvant faire intervenir :
- toutes les variables d'interface du BF (arguments d'entrée et sorties) (ici, R, SYS_INIT, ...) ;
- les variables de choix de mode de fonctionnement ou de mode dégradé (ici *Mode dégradé 1*,) ;
- le temps d'activation de l'état amont, sous le nom $Time\_in\_state$.

La principale différence entre le langage PLCopen et notre représentation est l'apparition de groupes d'états. Les groupes d'états permettent de simplifier la représentation

de multiples transitions ayant le même état aval, la même garde et la même priorité. Ils sont représentés par des ellipses en pointillés contenant une description du groupe d'états concerné. Cette description doit être exprimée dans un des langages ensemblistes (comme la Théorie naïve des ensembles proposée par [H+74]).

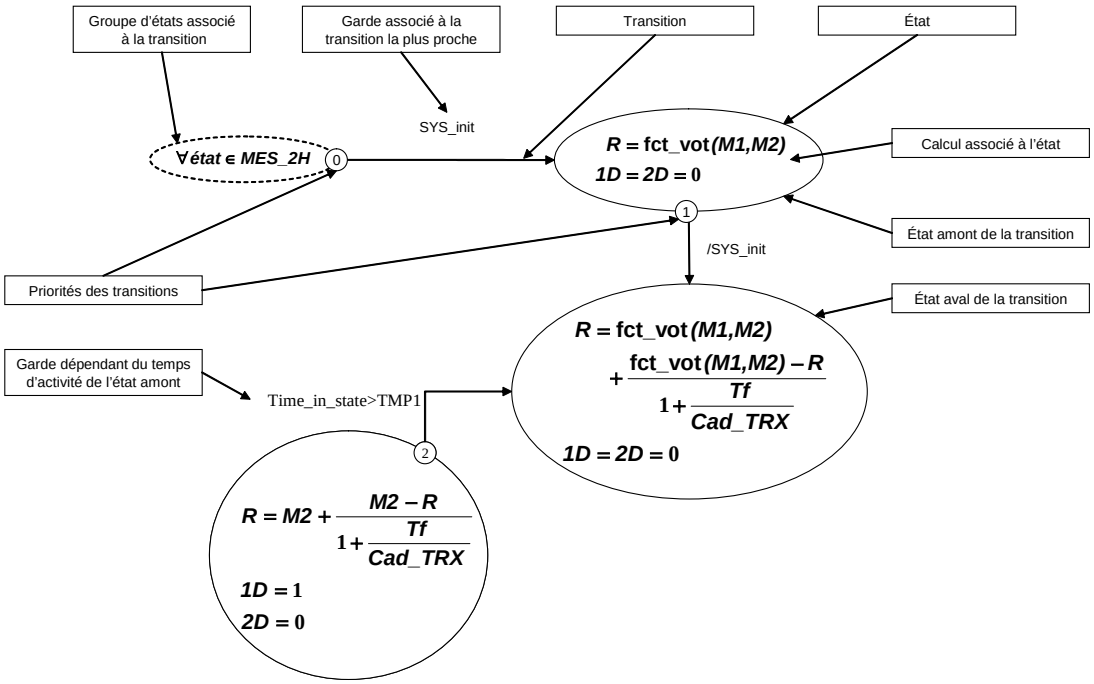


FIG. 5.7 – Syntaxe de l'automate à états

Ce concept de groupe d'états permet de définir des modes de fonctionnement (figure 5.8) : chaque état de l'automate à états est classifié dans un mode de fonctionnement. Trois grandes classes de modes de fonctionnement sont possibles :

- les initialisations ;
- les modes de fonctionnement nominaux ;
- les modes de fonctionnement dégradés.

Dans chaque classe, plusieurs modes de fonctionnement peuvent être définis et ces modes peuvent donner lieu à des sous-modes de fonctionnement, fonctions par exemple du type d'erreur rencontrée. Dans le cas des modes de fonctionnement dégradés, la description du comportement doit comprendre :

- La spécification des actions à effectuer pour assurer le fonctionnement, malgré la présence d'une erreur.
- La spécification des actions à effectuer pour revenir à un état de fonctionnement normal.

Le choix du mode de fonctionnement est déterminé dans la phase de pré-traitement (voir section 5.3.2.2) en associant une variable à chaque mode (ici *mode dégradé 1*, ...). Ces variables sont ensuite utilisées dans les transitions menant aux et sortant des modes dégradés (voir figure 5.8).

Chaque mode dégradé peut être relié à des variables d'erreur (ici $1D$ et $2D$). A titre d'information, l'association mode dégradé / variable d'erreur ainsi qu'un court descriptif

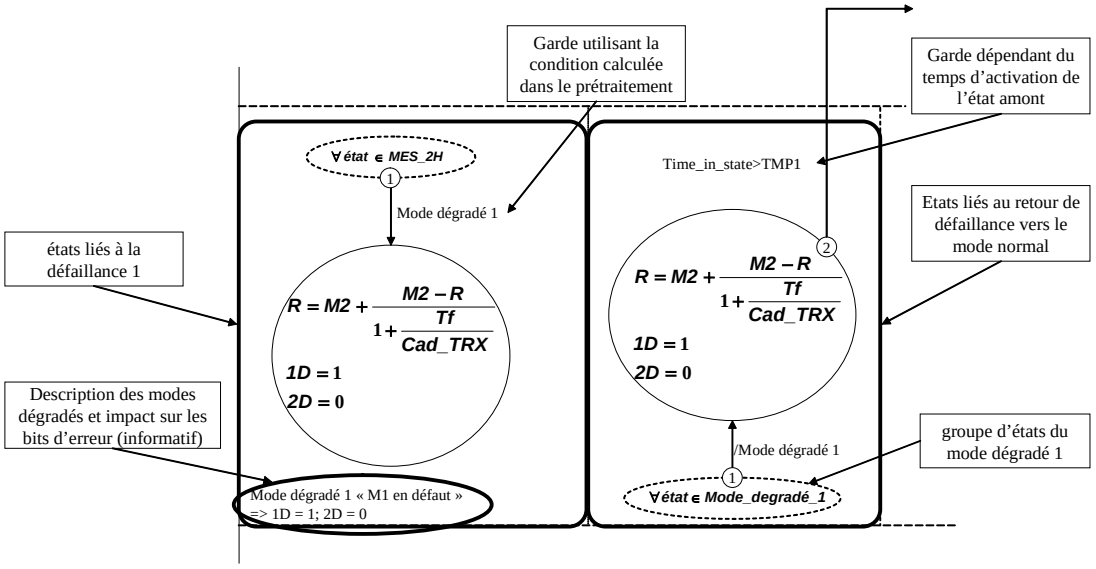


FIG. 5.8 – Syntaxe du mode dégradé

du mode dégradé peut être indiqué dans le cadre contenant le mode dégradé. Cependant, en cas de différence entre ces informations et les valeurs affectés dans les états, les valeurs affectées prévalent.

5.3.2.4 Post-traitement

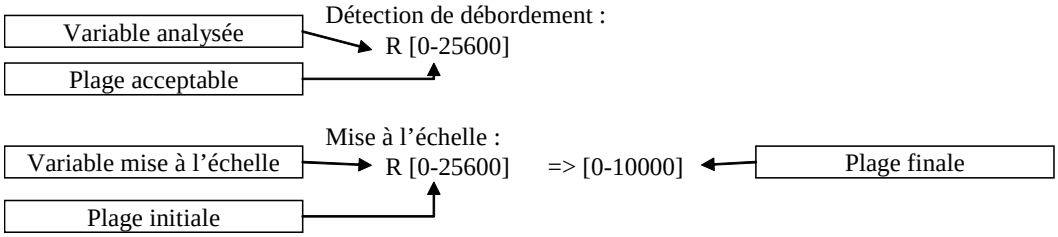


FIG. 5.9 – Syntaxe du post-traitement

La figure 5.9 présente le dernier élément de syntaxe : le post-traitement des données. Celui-ci est réalisé en deux étapes, dont le comportement est définie dans la section suivante :

Validation des variables de sortie. Si la valeur de la variable de sortie est en-dehors de la plage indiquée (débordement), elle est ramenée à la valeur inférieure ou supérieure de cette plage, selon le type de débordement (en-dessous de la valeur inférieure ou au-dessus de la valeur supérieure).

Mise à l'échelle des variables de sortie. La plage de valeurs possibles d'une variable est modifiée. Sa nouvelle valeur est obtenue en appliquant un coefficient de proportionnalité et un décalage d'origine.

5.3.3 Sémantique

5.3.3.1 Comportement externe

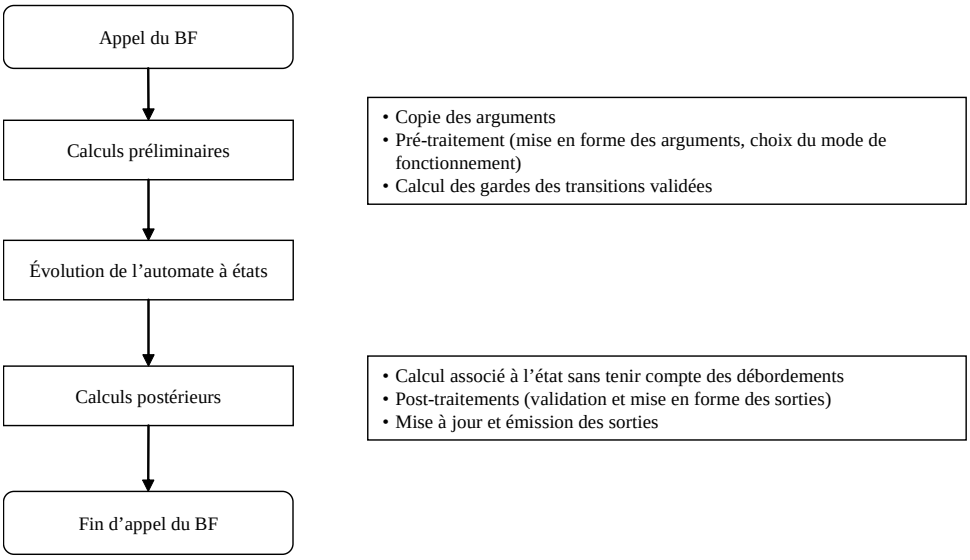


FIG. 5.10 – Moniteur de blocs fonctionnels AFBL

Le comportement global du bloc fonctionnel est décrit par le moniteur donné sur la figure 5.10. Ce moniteur décrit le comportement du BF lors de son appel, c'est-à-dire :

Calculs préliminaires

Cette étape assure les fonctions suivantes :
- copie des arguments d'entrée dans la mémoire interne du BF.
- opérations définies dans la partie pré-traitement, ce qui comprend :
  la mise en forme des argument d'entrée ;
  la choix du mode de fonctionnement.
- calcul des gardes des transitions activées.

Les détails de cette étape sont indiqués dans le paragraphe 5.3.3.2.

Évolution de l'automate à états

Ce comportement est décrit dans le paragraphe 5.3.3.3.

Calculs postérieurs

Cette étape assure les fonctions suivantes :
- calcul associé à l'état actif sans tenir compte des débordements ;
- opérations définies dans la partie post-traitement, soit :
  la validation des sorties ;
  la mise en forme des sorties.
- mise à jour et émission des sorties du BF.

Les détails de cette étape sont indiqués dans le paragraphe 5.3.3.4.

Il convient de noter que chaque variable utilisée ou affectée est une copie des arguments d'entrée ou des sorties. Donc toutes les actions de mise à l'échelle, seuillage et protection sont locales au BF.

5.3.3.2 Calculs préliminaires

Cette étape est réalisée en premier lors de l'appel du BF. Elle a pour but d'identifier en amont les modes de fonctionnement (normaux ou dégradés) et de mettre en forme les arguments. Les opérations suivantes sont exécutées séquentiellement :

Copie des arguments Chaque argument d'entrée est recopié dans la mémoire interne du BF. Toutes les opérations futures seront réalisées sur cette copie.

Mise en forme des arguments

Seuillage

Si la valeur de l'argument est en dehors de la plage de valeur indiquée entre crochets, elle est ramenée à la valeur la plus proche (minimale ou maximale) de la plage.

Mise à l'échelle

L'étendue de valeurs possibles d'un argument est modifiée. Sa nouvelle valeur est obtenue en appliquant un coefficient de proportionnalité et un décalage d'origine.

Choix du mode de fonctionnement Chacune des conditions est testée, séquentiellement. La première condition vraie détermine le mode dégradé correspondant. Ce choix de mode sera ensuite utilisé lors de l'évolution de l'automate à états

Calcul des gardes des transitions La garde de chaque transition validée (en amont de l'état actif) est calculée. Cette garde peut dépendre de la variable $Time\_in\_state$ qui représente le temps d'activation de l'état amont.

5.3.3.3 Évolution de l'automate à états

La sémantique de l'automate à états est inspirée de la proposition de l'association PLC Open. Cependant, notre proposition inclut plusieurs possibilités supplémentaires :

- plusieurs modes dégradés ;
- des transitions associées à un groupe d'état ;
- possibilité d'utilisation de variables entières dans les calculs associés aux états ;
- priorité obligatoire, afin d'assurer le déterminisme.

Les automates à états AFBL évoluent donc selon les règles suivantes :

1. Un seul état est actif à chaque instant dans l'automate. Au premier lancement du BF, l'état actif n'est pas défini (indéterministe). Pour cette raison, ce premier lancement doit obligatoirement activer une transition d'initialisation (réalisée avec la variable d'initialisation SYS_INIT mise à 1 dans l'exemple figure 5.4). L'état pointé par cette transition d'initialisation sera par la suite considéré comme l'état initial.
2. La priorité associée à une transition est un nombre entier positif ou nul. Plus ce nombre est faible, plus la priorité est élevée. Les valeurs 0 et 1 sont réservées, respectivement, à l'initialisation et au choix de mode de fonctionnement.
3. Une transition associée à un groupe d'états est équivalente à un ensemble de transitions partant de chaque état du groupe avec la même garde et pointant sur le même état aval. Un groupe d'états peut seulement être utilisé en amont d'une transition.

4. Une transition est validée si son état amont est actif. Dans le cas d'une transition partant d'un groupe d'états, elle est validée si l'état actif fait parti du groupe.
5. Une transition est franchissable si elle est validée et que sa garde est vraie.

L'évolution de l'automate est réalisée en trois temps :

1. Calcul des gardes associées aux transitions validées (effectué dans le pré-traitement du BF).
2. Franchissement de la transition franchissable dont la priorité est la plus haute (zéro étant la plus haute priorité)
3. Désactivation de l'état amont et activation de l'état aval de la transition franchie. Puis mise à 0 de la variable temps *Time_in_state* (correspondant au temps d'activation de l'état actif).

Nous pouvons remarquer que :
– une seule transition peut être franchie par appel du BF (pas de recherche de stabilité).
– si aucune transition n'est franchissable, l'automate n'évolue pas.

5.3.3.4 Calculs postérieurs

Cette étape permet d'effectuer tout d'abord les calculs associés à l'état actif ainsi que de mettre en forme les résultats de ces calculs. Les calculs indiqués dans les états sont en effet des affectations de variables avec des expressions arithmétiques ou booléennes et sont réalisés sans tenir compte de la plage de valeur des variables ni de la capacité de la cible.

Suite au calcul, deux opérations sont effectuées :

Mise en forme des variables de sortie. Pour les variables dont la plage de validité est indiquée, un seuillage est réalisé : si la valeur de la variable est en dehors de la plage de valeur, elle est ramenée à la valeur la plus proche dans la plage.

Pour les variables dont l'échelle est indiquée, une mise à l'échelle est réalisée ; l'étendue de valeurs possibles d'une variable est modifiée. Sa nouvelle valeur est obtenue en appliquant un coefficient de proportionnalité et un décalage d'origine.

Mise à jour des sorties. Les sorties sont affectées avec les valeurs de leurs images internes.

5.4 Formalisation des blocs fonctionnels

Il importe à présent de doter la représentation sous forme d'automates à état d'une sémantique formelle, ceci en particulier afin de permettre la vérification formelle.

Les trois sous-sections suivantes exposent notre contribution sur ce sujet. Une attention particulière est portée à l'efficacité du modèle obtenu afin de pouvoir les vérifier dans des temps raisonnables. Ceci est obtenu en utilisant une représentation sous forme de machine de Moore (sous-section 5.4.1) en codant cette machine sous la forme d'un Global

Transition System (GTS) construit de manière adéquate (sous-section 5.4.2). La dernière sous-section s'intéresse à la représentation des BF utilisant le temps.

Pour illustrer nos propos, nous nous appuierons sur le BF *CB* représenté à la figure 5.11. Ce BF admet 4 arguments et émet une sortie (présentés sur le tableau 5.2). Il permet de détecter le dépassement d'un seuil bas (SB) par l'argument d'entrée E. En cas de dépassement, l'argument doit repasser au dessus de SB+H avant d'être considéré comme *non-dépassant* (phénomène d'hystérésis).

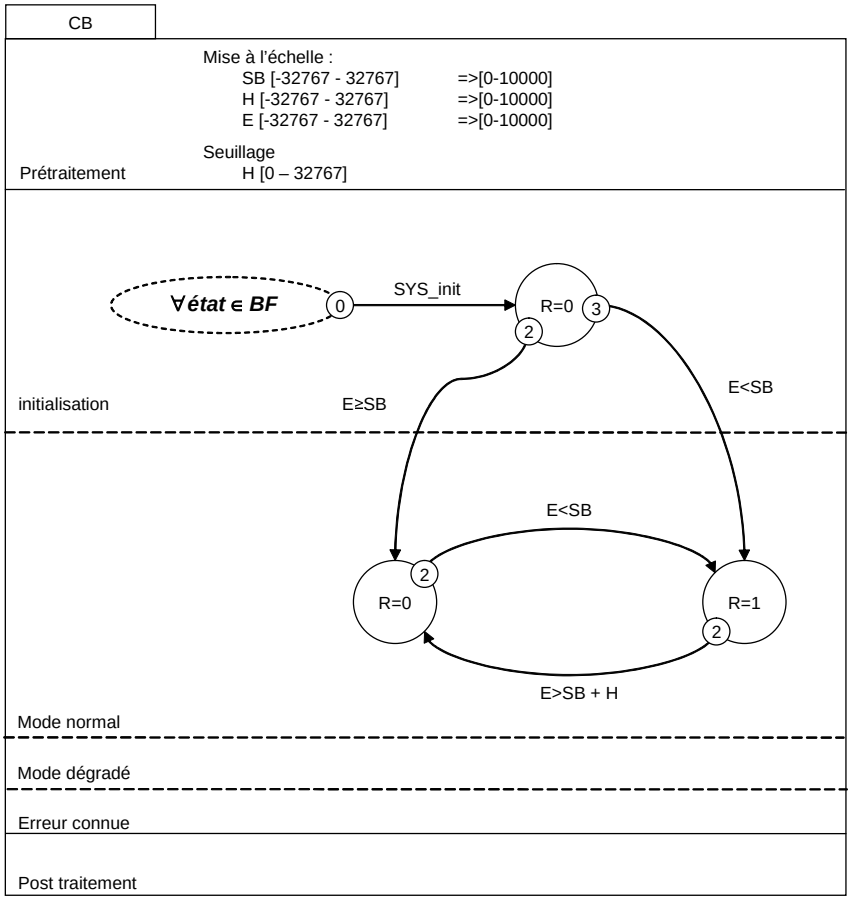


FIG. 5.11 – Représentation AFBL du BF détection de dépassement de seuil bas (CB)

Interface	Type	Commentaires	Symbole
Argument	Booléen	Initialisation du système	SYS_Init
Argument	entier	Entrée à comparer	E
Argument	entier	Paramètre Seuil Bas	SB
Argument	entier	Paramètre Hystérésis	H
Sortie	booléen	Résultat dépassement de seuil	R

TAB. 5.2 – Liste des variables du BF CB

5.4.1 Formalisation sous forme de machine de Moore

Seul l'automate à états sera traduit en machine de Moore. Le moniteur et les calculs préliminaires et postérieurs ne correspondent pas à des états de la partie commande, ils sont juste une aide à l'écriture des opérations effectuées dans l'automate à états. Ces opérations peuvent être donc placées dans le calcul des gardes des transitions ou des variables de sortie.

Afin de formaliser cet automate sous la forme d'une machine de Moore, les spécificités de notre représentation doivent être traduites. Soit, séquentiellement :

Les groupes d'états

Les références aux groupes d'états sont remplacées par autant de transitions que nécessaires. Ces transitions partent de chaque état du groupe avec la même garde et pointent sur le même état aval, avec la même priorité.
De plus, l'état pointé par la transition d'initialisation (liée à la variable SYS_INIT) partant du groupe d'état représentant tous les états, est considéré comme l'état initial.

Les priorités

L'adaptation des priorités est réalisée en modifiant les gardes. Pour chaque garde g_n associée à une transition de priorité n, toutes les gardes g_i associées à des transitions de priorités i inférieures et ayant le même état amont sont modifiées en g_n & $!g_i$. Plus formellement :

$$\forall g_n \in \mathcal{G}, \exists\ f_n \in \mathcal{F} \text{ tel que } f_n = g_n . \prod_{j \in \mathcal{P}_j} \overline{g_j}$$

Où :

$\mathcal{G}$ est l'ensemble des gardes de la représentation AFBL.

$\mathcal{P}_j$ est l'ensemble des gardes de cette représentation ayant un priorité inférieure à la transition j et partant du même état.

$\mathcal{F}$ est l'ensemble des gardes de la machine de Moore équivalente.

La figure 5.12 indique l'application de ces deux traductions à notre exemple de BF CB [7]. Nous pouvons remarquer que le groupe de transition '$\forall états \in BF$' est transformé en trois transitions partant de chaque état de la machine de Moore et ayant pour garde SYS_init et pour priorité 0. Puis les gardes sont modifiées en fonction des priorités :

- Les transitions associées à la garde SYS_init (venant du groupe de transitions) ont la plus haute priorité (0). Leur garde n'est donc pas modifiée.
- Les transitions de l'état 1 à 2, 2 à 3 et de 3 à 2 sont de priorité 2 et sont associées aux gardes $E \geq SB$, $E < SB$ et $(E > (SB+H))$. Donc, les transitions de priorité 0 sont plus prioritaires. Les gardes de ces transitons sont donc modifiées en $\overline{SYS\_init}.(E \geq SB)$, $\overline{SYS\_init}.(E > (SB+H))$ et $\overline{SYS\_init}.(E < SB)$ afin que les gardes associées aux transitions de priorité 0 et 2 soit exclusives[8].
- De la même manière, les gardes de transitions de priorité de 3 sont modifiées de façon à ce qu'elle soit exclusives par rapport aux autres gardes.

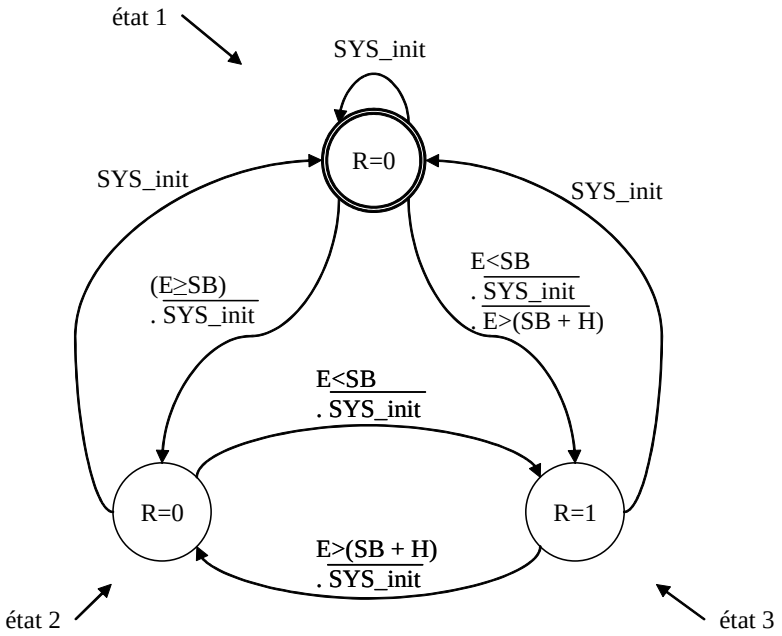


FIG. 5.12 – Machine de Moore du BF CB

Les actions de la machine de Moore équivalente correspondent aux calculs de l'état correspondant dans la représentation AFBL ainsi qu'aux calculs préliminaires et postérieurs. Dans le cas du BF CB, les variables SB, E et H sont remplacées par leurs valeurs mises à l'échelle et seuillées.

5.4.2 Traduction en GTS

L'objectif de cette étape est de pouvoir vérifier les BF spécifiés dans notre représentation et de construire la bibliothèque de modèles formels de BF utilisée dans les algorithmes du chapitre 3. Pour atteindre cet objectif, les machines de Moore obtenues précédemment sont transformées en des modèles en langage GTS utilisables par NuSMV.

Cette étape est réalisée en deux phases. Premièrement, les R-variables nécessaires aux calculs dans les états de la machine de Moore sont identifiées. Puis, la structure de la machine de Moore est représentée en GTS à l'aide de R-variables supplémentaires.

5.4.2.1 Détermination des R-variables nécessaires aux calculs

Afin de minimiser le nombre de variables utilisées dans NuSMV pour représenter le BF, il faut identifier tout d'abord les R-variables intervenant dans les calculs associés aux états de l'automate. Nous rappelons qu'une R-variable est définie comme suit :
Une variable V_i d'un modèle M sous forme GTS est une R-variable si et seulement si elle

[7]Dans un but de compréhension, des numéros ont été ajoutés aux états (1, 2 et 3)
[8]La priorité 1 n'est pas présente dans ce BF car il ne contient qu'un seul mode.

respecte la condition suivante :

$$\exists V_j \in M, V_{j,k+1} = f(\ldots, V_{i,k}, \ldots)$$

Où :
- $V_{j,k+1}$ est la valeur de la variable V_j dans l'état futur $(k+1)$,
- $V_{i,k}$ est la valeur de la variable V_i dans l'état courant (k),
- f est une fonction dont l'un des arguments est $V_{i,k}$.

Dans le cas des blocs fonctionnels, une variable est une R-variable si son état précédent est nécessaire, ce qui se produit lorsque l'une des deux conditions suivantes est remplie :

1. Si la variable est utilisée avant d'être affectée, comme énoncé dans la définition 2 du chapitre 3. Dans le cas d'un BF, si cette condition est remplie localement pour au moins un état de la machine de Moore, cette variable est une R-variable globalement pour le BF.
2. S'il existe un état où cette variable n'est pas affectée.

Nous pouvons remarquer que ces deux conditions sont liées ; d'après la sémantique des calculs présentée dans la sous-section 5.3.3.4, ne pas affecter une variable dans un état (condition 2) correspond à lui affecter sa valeur précédente. Elle est donc utilisée avant d'être affectée (condition 1) et est donc une R-variable. Définir deux conditions a principalement un rôle utilitaire : repérer graphiquement les R-variables en s'affranchissant de la sémantique.

Dans le cas du BF CB, R n'est pas une R-variable car elle ne répond à aucune de ces deux conditions. Par contre, la transformation de l'automate du BF MES_2H (figure 5.4) nécessite l'introduction d'une R-variable : R.

Une fois que les R-variables nécessaires aux calculs sont identifiées, il importe de coder le comportement de la machine de Moore, ce qui peut nécessiter l'introduction d'autres R-variables. Ce point est l'objet de la sous-section suivante.

5.4.2.2 Possibilités de représentation d'un état

Trois solutions sont possibles pour définir les R-variables du GTS modélisant les évolutions de la machine de Moore :

Une variable par état

Une variable booléenne X_i est associée à chaque état i de la machine de Moore. Cette solution suit les mêmes règles que la représentation algébrique du Grafcet proposée par [MDL+06b]. Le calcul de chaque variable X_i est donc réalisé comme suit :

$$X_{i,k+1} = \sum_{j \in \mathcal{P}_i} (X_{j,k} \cdot g_{j,k+1}) + X_{i,k} \cdot \overline{\sum_{l \in \mathcal{N}_i} g_{l,k+1}}$$

Où :
- i, j, k et l sont des entiers positifs,
- X_i est la variable représentant l'activité de l'état i avec :
  - $X_{i,k}$ sa valeur avant exécution du BF,

  - $X_{i,k+1}$ sa valeur après exécution du BF.
- $\mathcal{P}_i$ est l'ensemble des états amont de l'état i et :
  - $X_{j,k}$ représente l'activité avant l'exécution du BF d'un état amont $j \in \mathcal{P}_i$.
  - $g_{j,k+1}$ est la valeur de la garde, durant l'exécution du BF, de la transition entre les états j et i.
- $\mathcal{N}_i$ est l'ensemble des transitions aval de l'état i et :
  - $g_{l,k+1}$ est la valeur de la garde, durant l'exécution du BF, de la transition $l \in \mathcal{N}_i$.

Dans le cas du BF CB, le GTS représentant le comportement de la machine de Moore est donc[9] :

$$\begin{aligned}
X_{1,k+1} &= (X_{1,k} + X_{2,k} + X_{3,k}) \cdot SYS_init_{k+1} \\
&\quad + X_{1,k} \cdot \overline{(\overline{SYS_init_{k+1}} \cdot ((E_{k+1} \geq SB_{k+1}) + (E_{k+1} < SB_{k+1})))} \\
X_{2,k+1} &= [(X_{1,k} \cdot (E_{k+1} \geq SB_{k+1})) + (X_{3,k} \cdot (E_{k+1} > (SB_{k+1} + H_{k+1})))] \cdot \overline{SYS_init_{k+1}} \\
&\quad + X_{2,k} \cdot \overline{SYS_init_{k+1} + (E_{k+1} < SB_{k+1})} \\
X_{3,k+1} &= (X_{1,k} + X_{2,k}) \cdot (\overline{SYS_init_{k+1}} \cdot (E_{k+1} < SB_{k+1})) \\
&\quad + X_{3,k} \cdot \overline{SYS_init_{k+1} + (E_{k+1} > (SB_{k+1} + H_{k+1}))}
\end{aligned}$$

Une variable entière pour tous les états

Chaque valeur d'une variable entière X est associée à chaque état (comme proposé dans [LYF05]). Le comportement est alors décrit en modifiant la valeur de la variable X avec une structure conditionnelle de type *case*.

Une condition Ci par état i de la machine de Moore est présente dans la structure conditionnelle. Si la condition Ci est vraie, alors la prochaine valeur de X est i, ce qui signifie que l'état de la machine de Moore après l'exécution du BF sera i. Chaque condition Ci est de la forme :

$$\sum_{j \in \mathcal{P}i} ((X_k = j) \cdot g_{j,k+1})$$

- i, j et k sont des entiers positifs,
- X_k est la valeur de la variable indiquant l'état actif avant l'exécution du BF,
- $\mathcal{P}_i$ est l'ensemble des états amont à l'état i et, si $j \in \mathcal{P}i$, alors :
  - $g_{j,k+1}$ est la valeur de la garde de la transition entre les états j et i durant l'exécution du BF.

Le cas *par défaut* de la structure conditionnelle *case* correspond au fait qu'aucune transition n'est franchissable. Elle mène donc à la conservation de l'état précédent (avant l'exécution du BF) soit X_k.

Dans le cas de la figure 5.12, le GTS résultant est donc :

$$\begin{aligned}
X_{k+1} = case \\
&((X_k = 1) + (X_k = 2) + (X_k = 3)) \cdot SYS_init_{k+1} \quad : 1;
\end{aligned}$$

[9] Dans un but de compacité, les expressions ont été simplifiées.

$$\begin{array}{rl} ((X_k = 1) \cdot (E_{k+1} \geq SB_{k+1}) \cdot \overline{SYS\_init_{k+1}} & \\ +(X_k = 3) \cdot (E_{k+1} > (SB_{k+1} + H_{k+1})) \cdot \overline{SYS\_init_{k+1}}) & : 2; \\ ((X_k = 1) + (X_k = 2)) \cdot \overline{SYS\_init_{k+1}} \cdot (E_{k+1} < SB_{k+1}) & : 3; \\ 1 & : X_k; \end{array}$$

esac;

Aucune variable supplémentaire

Aucune R-variable supplémentaire n'est alors utilisée car le comportement est directement associé aux variables de sortie. Par exemple, la variable R du BF CB peut être déclarée comme une R-variable puis sa formule de récurrence est utilisée pour représenter le comportement du BF. En effet, nous pouvons observer que :

$$R_{k+1} = \overline{(SYS\_init_{k+1} \cdot (E_{k+1} > (SB_{k+1} + H_{k+1})))} \cdot (R_k + (E_{k+1} > SB_{k+1}))$$

Cependant, il n'existe pas à notre connaissance de démarche systématique pour obtenir cette relation. Cette approche repose essentiellement sur l'expérience du modélisateur et ne peut s'appliquer qu'à des cas simples.

Bien que la dernière possibilité soit prometteuse en terme de performances elle n'est pas généralisable à tous les cas de représentation et, surtout, est difficile à mettre en œuvre pour des cas plus complexes. De ce fait, il a été choisi de retenir la deuxième possibilité de représentation, qui constitue un bon compromis entre la compacité de la représentation (nombre de variables, ...) et l'automatisation de la transformation d'une machine de Moore en GTS.

5.4.2.3 Modélisation des actions

Les calculs de chaque état sont transformés en affectations de R-variables ou en déductions logiques des NR-variables (voir chapitre 3). Les expressions associées sont sous la forme d'une structure conditionnelle *case* où chaque condition est l'activité d'un état de la machine de Moore et la conséquence correspondante est le calcul de la variable dans cet état.

Dans le cas du BF CB, avec une représentation des états basée sur une variable entière (deuxième possibilité ci-dessus), l'expression de la déduction logique liée à R est donc :

$$\begin{array}{l} case \\ \quad \begin{array}{ll} X = 1 & : 0; \\ X = 2 & : 0; \\ X = 3 & : 1; \end{array} \\ esac \end{array}$$

5.4.3 Abstraction du temps

Nous ne considérons dans nos travaux que le *temps logique*. Cette abstraction, qui est celle de la logique temporelle CTL, consiste à ne considérer que l'ordre relatif des

évènements (a avant b, b après c ...) et non les valeurs des durées entre les occurrences évènements ou leurs dates en temps absolu. Les représentations formelles de blocs fonctionnels faisant référence au temps devront donc s'appuyer sur cette abstraction.

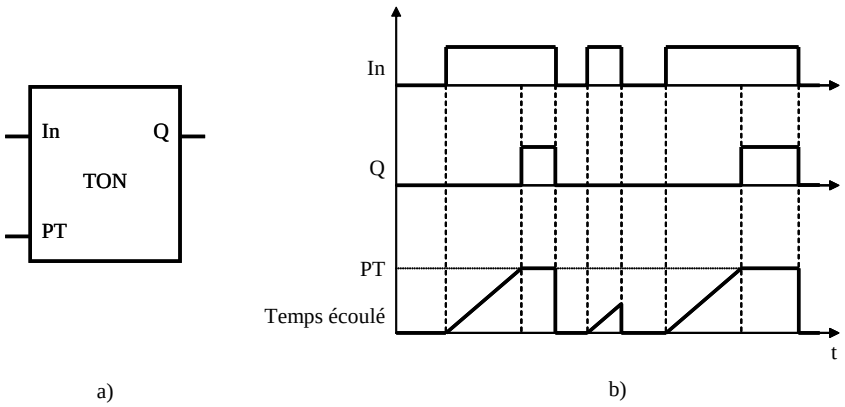


FIG. 5.13 – a) le bloc fonctionnel TON et b) son comportement indiqué dans la norme CEI 61131-3

Nous pouvons illustrer ceci sur l'exemple figure 5.13 qui est une temporisation à l'enclenchement, ou Time ON delay (TON). Son comportement est décrit de manière informelle dans la norme CEI 61131-3 [IEC93] et permet de retarder l'enclenchement d'une sortie Q d'une durée PT par rapport à son entrée In.

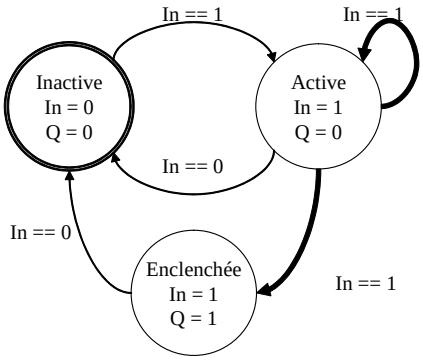


FIG. 5.14 – Modèle à trois états du bloc fonctionnel TON donnée dans [Ros03]

Nous pouvons remarquer que la temporisation à enclenchement comporte trois états logiques :

- la temporisation est inactive ($In = 0$ et $Q = 0$), c'est-à-dire que l'entrée In est à 0 et que donc la sortie Q l'est aussi ;
- la temporisation est active ($In = 1$ et $Q = 0$), ce qui signifie que In est passée à 1 depuis moins de PT unités de temps (on compte PT unités de temps depuis le front montant de In) ;
- la temporisation est enclenchée ($In = 1$ et $Q = 1$) ce qui signifie que In est passée à 1 depuis plus de PT unités de temps et donc que la sortie Q a été positionnée à 1.

Avec l'abstraction du temps choisie, un modèle proposé par [Ros03] est présenté sur la figure 5.14. Nous pouvons remarquer l'introduction d'une évolution non déterministe à la sortie de l'état *actif* (transitions en trait fort sur la figure 5.14) qui représente le temps logique relatif au délai (le délai peut être écoulé ou non). Cet indéterminisme peut être traduit sous la forme d'un GTS caractérisant toutes les évolutions possibles du modèle, comme présenté ci-dessous :

$$
\begin{aligned}
Q_{i+1} := case \\
&\overline{In_{i+1}} && : 0; \\
&In_{i+1} \ \& \ \overline{Q_i} && : \{0,1\}; \\
&In_{i+1} \ \& \ Q_i && : 1; \\
esac;
\end{aligned}
$$

Où :

- In_{i+1} est la valeur de l'argument d'entrée In pendant l'exécution du bloc TON,
- Q_i est la valeur de Q avant l'exécution du bloc TON,
- Q_{i+1} est la valeur de Q après l'exécution du bloc TON.

Cette abstraction de temps logique peut être étendue à tous les blocs fonctionnels dont la modélisation sous forme d'automates fait appel à des gardes comportant des tests fonctions de temps physique. Pour ce faire, chaque test du temps physique dans une garde est transformé en une représentation logique.

Par exemple, le test visant à déterminer si le temps physique est supérieur à une valeur donnée est remplacé dans son ensemble par un indéterminisme traduisant : *Le temps physique est-il supérieur à cette valeur ou non ?*. Pour chaque transition dont la garde comporte un test fonction du temps physique, un comportement non-déterministe, sous la forme de deux transitions ayant des gardes identiques, est donc introduit.

Nous préférons pour notre part une modélisation différente, dans laquelle l'indéterminisme provenant de l'abstraction du temps physique en un temps logique est traduit non pas par deux transitions concurrentes mais par une entrée (forcément non déterministe) supplémentaire.

Le modèle du bloc fonctionnel TON avec notre représentation[10] est donné sur la figure 5.15. La figure 5.16 présente sa traduction en machine de Moore utilisant l'abstraction temporelle.

L'automate AFBL comporte une transition dont le garde $((Time\_in\_state > PT)$ & $In)$ fait référence à un temps physique T correspondant au temps d'activité de l'état amont. La machine de Moore déduite de cet automate (figure 5.16) utilise une variable $Time\_elapsed$, nouvelle entrée du modèle, qui traduit le non-déterminisme dû à l'abstraction du temps. Cette variable traduit le fait que le délai s'est écoulé ou non. La garde devient alors : $Time\_elapsed$ & In.

[10] Cette représentation ne dispose pas de transition de priorité 1 car un seul mode de marche est considéré

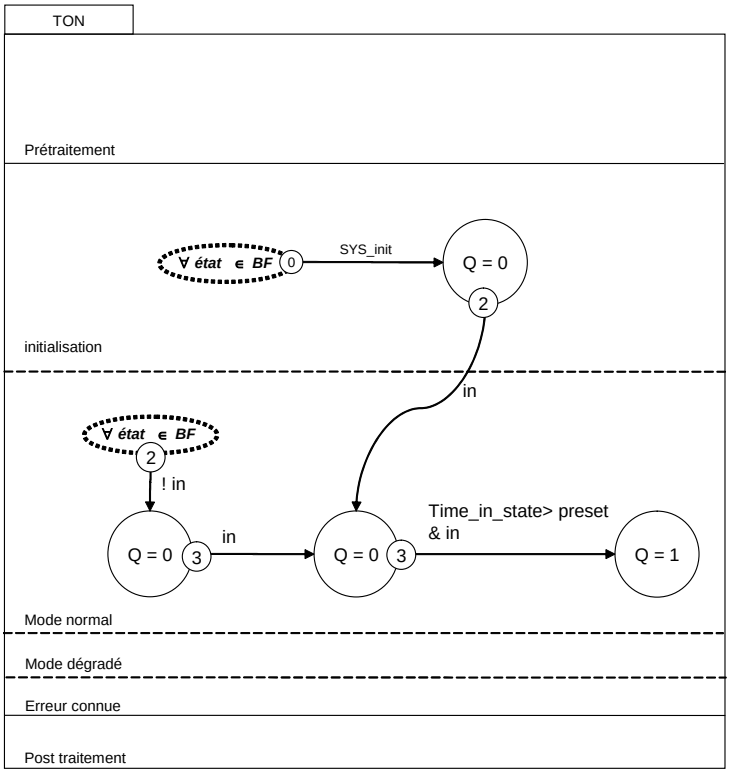


FIG. 5.15 – Représentation AFBL du bloc fonctionnel TON

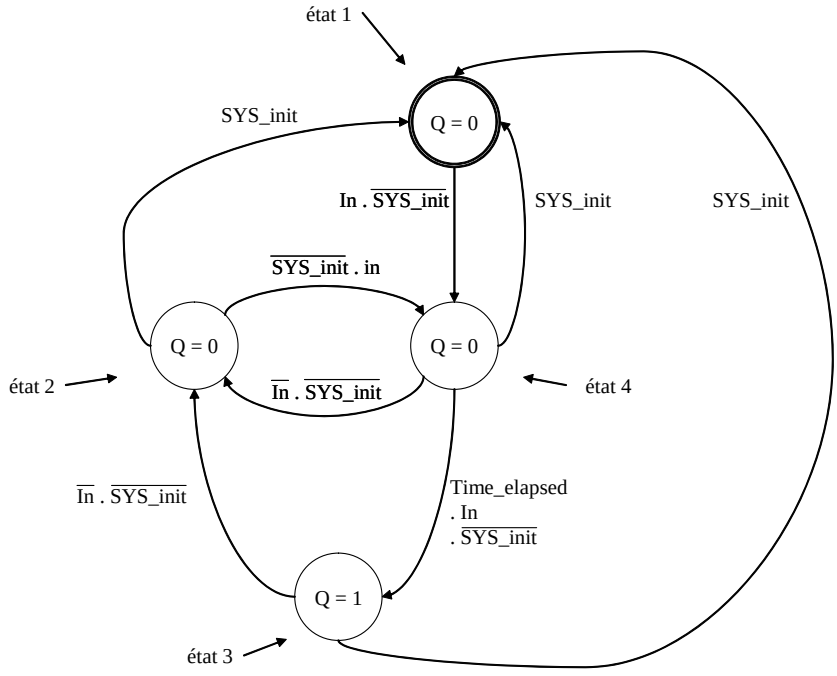


FIG. 5.16 – Représentation en machine de Moore du bloc fonctionnel TON avec abstraction du temps

```
next(state):=
 case
  SYS_init                                                  : next(state)=0;
  (state=0 & next(in) & !next(SYS_init))
    |(state=2 & !next(in) & !next(SYS_init))                : next(state)=1;
  (state=3 & !next(in) & !next(SYS_init))
    |(state=1 & !next(in) & !next(SYS_init))                : next(state)=2;
  (state=1 & next(in) & Time_elapsed & !next(SYS_init))     : next(state)=3;
 esac;
DEFINE
  Q := next(state)=3 ;
```

FIG. 5.17 – Modèle NuSMV du bloc fonctionnel TON

Si un automate comporte plusieurs transitions dont les gardes comportent un test sur une durée, et si ces tests sont différents, il est alors rajouté autant d'arguments d'entrée que de tests.

Finalement, le modèle NuSMV est obtenu (figure 5.17) en suivant la démarche indiquée dans la section précédente (une variable entière pour tous les états).

5.5 Conclusion

Ce chapitre nous a permis de présenter le langage AFBL pour la modélisation de blocs fonctionnels. Cette représentation a pour but de fournir un langage commun aux automaticiens et aux développeurs d'outil de développement d'automatismes. Nous avons défini tout d'abord de manière textuelle et graphique la syntaxe de ce langage, puis l'avons doté d'une sémantique formelle à l'aide de machines de Moore. Nous avons également montré qu'il était possible de traduire un modèle formel en GTS.

Les conséquences de ces travaux sont les suivants

- Tout bloc fonctionnel AFBL possède un comportement sans ambiguïté.
- Il est possible de vérifier tout BF.
  Notamment, le lien avec le langage GTS permet la vérification d'un BF avec NuSMV :
  - soit en vérifiant des propriétés formelles sur le modèle GTS de ce BF ;
  - soit en vérifiant l'équivalence comportementale entre le code du BF (obtenu avec la démarche présentée dans le chapitre 3) et le modèle formel de sa représentation. Cette équivalence est vérifiée en utilisant la démarche proposée dans le chapitre 6.
- Il est possible de vérifier tout programme comportant de tels BF.
  En effet, les modèles en GTS obtenus sont compatibles avec la démarche de modélisation du chapitre 3.

Une fois tous les BF définis dans ce langage et traduits en langage GTS, la bibliothèque de modèles formels utilisée dans les algorithmes du chapitre 3 peut être constituée. Notre méthode de vérification de contrôleurs logiques est donc maintenant complète. Nous allons

donc monter dans le chapitre suivant comment l'ensemble de cette méthode a été utilisé dans un contexte industriel.

Chapitre 6

Utilisation des représentations formelles dans un contexte industriel

RÉALISER des modèles de contrôleurs industriels, comme proposé dans le chapitre 3, n'est qu'un préliminaire au processus de vérification. Ces modèles doivent ensuite être utilisés dans un model-checker pour vérifier les différentes propriétés.
Ce chapitre présente donc *l'utilisation* de ces modèles, dans un contexte industriel. Nous montrerons dans la première section comment les résultats théoriques obtenus lors de nos travaux ont été intégrés au logiciel Controcad afin de rendre accessibles les techniques de vérification formelle.
Les deux sections suivantes présentent respectivement les possibilités en matière de vérification de propriétés extrinsèques et d'équivalence comportementale, objectifs premiers de ces recherches.
Enfin, les deux dernières sections sont consacrées à la présentation des possibilités en matière de vérification de propriété intrinsèques, l'outil de preuves développé possédant des caractéristiques qui dépassent nos objectifs initiaux.

Sommaire

6.1 Intégration des résultats théoriques dans le logiciel Controcad

Les travaux présentés dans ce mémoire ont été réalisés dans l'optique d'être appliqués dans l'entreprise Alstom Power et plus précisément d'être intégrés dans logiciel Controcad, présenté dans le chapitre 1. Pour ce faire, les représentations formelles et algorithmes développés ont permis la création d'une spécification [GCV07] pour un *outil de model-checking pour la contribution à la vérification de la sûreté de fonctionnement logicielle du programme applicatif généré.*

L'objectif de cet outil est de fournir une interface entre Controcad et le model-checker NuSMV afin de permettre à un utilisateur automaticien, non spécialiste des techniques de preuves, d'accéder à ces techniques sans difficultés particulières. Cet objectif s'inscrit dans le cadre de l'amélioration de la sûreté des programmes générés par Controcad, afin de faciliter la certification de ces programmes au niveau SIL 3 de la norme CEI 61508.

Cet outil a deux champs d'application, présentés sur la figure 6.1 dans le chapitre 1 :
- la vérification de la génération des programmes pour le contrôleur, comme préconisé dans la norme CEI 61508, c'est-à-dire la comparaison du comportement du programme LEA et de celui du programme C.
- la vérification de propriétés relative au comportement attendu du programme ou aux préconisations de la norme CEI 61131-3.

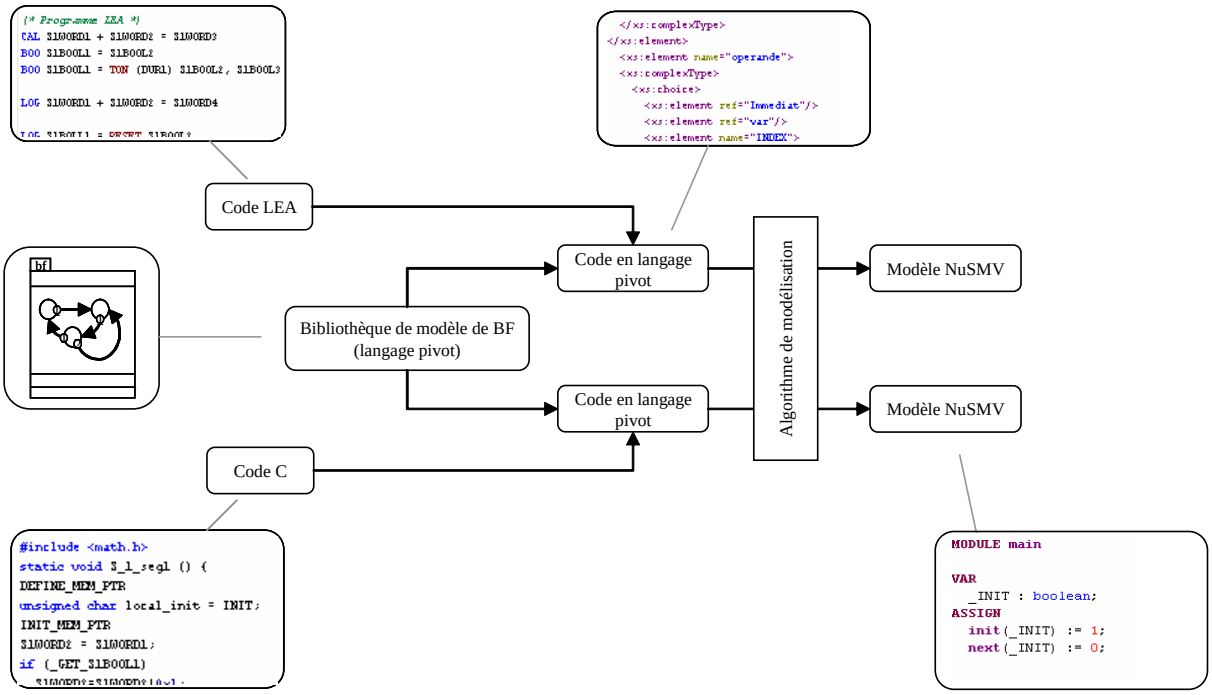


FIG. 6.1 – Interfaçage LEA/C vers NuSMV

La structure de l'interface LEA/C-NuSMV est donnée sur la figure 6.1. Plusieurs points méritent d'être soulignés :

- l'utilisation d'un langage pivot permet une modélisation équivalente des programmes écrits en C et LEA ;
- l'utilisation d'une bibliothèque de BF décrits en langage pivot ;
- la traduction des codes LEA et C en modèles formels NuSMV se fait *sans intervention de l'utilisateur, de même que toutes les vérifications décrites dans la suite de ce chapitre.*

Le premier point découle d'une préoccupation technique non abordée auparavant dans ce mémoire mais pourtant essentielle à l'automatisation des méthodes proposées : les analyses syntaxique et sémantique d'un programme. L'analyse syntaxique est réalisée par un module communément appelé *parser*. Il a la charge de lire le programme textuel et d'analyser sa syntaxe et sa structure et d'en retourner une représentation structurée utilisable par un outil de traduction automatique.

La deuxième phase de l'analyse du programme textuel, l'analyse sémantique, utilise les données retournées pas le parser et associe à chaque structure un comportement. Ceci est en fait la première partie de la modélisation. Dans notre méthode, nous ne considérons que le remplacement de blocs fonctionnels mais cette analyse peut être étendue pour caractériser tous les éléments de langage qui n'ont pas d'équivalents dans le model-checker.

Cette caractérisation de comportement doit être effectuée dans un langage adéquat, faisant le lien entre le comportement du langage du programme (LEA ou C) et celui du model-checker. Le langage pivot développé ici répond à ce besoin. Il est proche, en terme de comportement, des langages ST et LEA. Cependant, il ne comporte ni mémoires internes (implicites), ni structures conditionnelles, ni instruction fonction de temps physique. Les instructions des langages LEA et C des types ci-dessus sont donc remplacées, dans le modèle en langage pivot, par des mémoires et structures conditionnelles explicites dans le programme et par des expressions fonctions du temps logique (comme indiqué dans la section 5.4.3).

Le deuxième point découle des résultats présentés dans le chapitre 5.

Enfin, nous tenons à souligner que tant l'obtention des modèles formels (figure 6.1) que les vérifications qu'il est possible de réaliser sur ces modèles se font directement à partir du logiciel Controcad.

La figure 6.2 présente l'interface utilisateur que nous avons développée, dans le cadre de nos travaux. Cette interface permet à un utilisateur de vérifier les propriétés suivantes qui seront développées dans la suite de ce chapitre :

No limite overflow : vérification de non dépassement de borne de chaque entier du contrôleur.

No division by zero : vérification de non division par zéro pour chaque opération de division.

Liveness : chaque variable du contrôleur peut toujours évoluer (et ne reste pas bloquée à une valeur).

Without bell effect : sans variation des variables d'entrée, les sorties finissent toujours par se stabiliser.

Not unreachable : le SFC considéré ne contient pas de branche morte (voir section 6.5).

Not unsafe : le SFC considéré n'a pas de possibilité de multiplication incontrôlée des jetons (voir section 6.5).

Reinitialisable : le SFC considéré peut toujours revenir dans son état initial (voir section 6.5).

State machine : le SFC considéré a toujours une et une seule étape active (voir section 6.5).

Property check : vérification de propriété extrinsèque, exprimée en langage CTL (voir section 6.2).

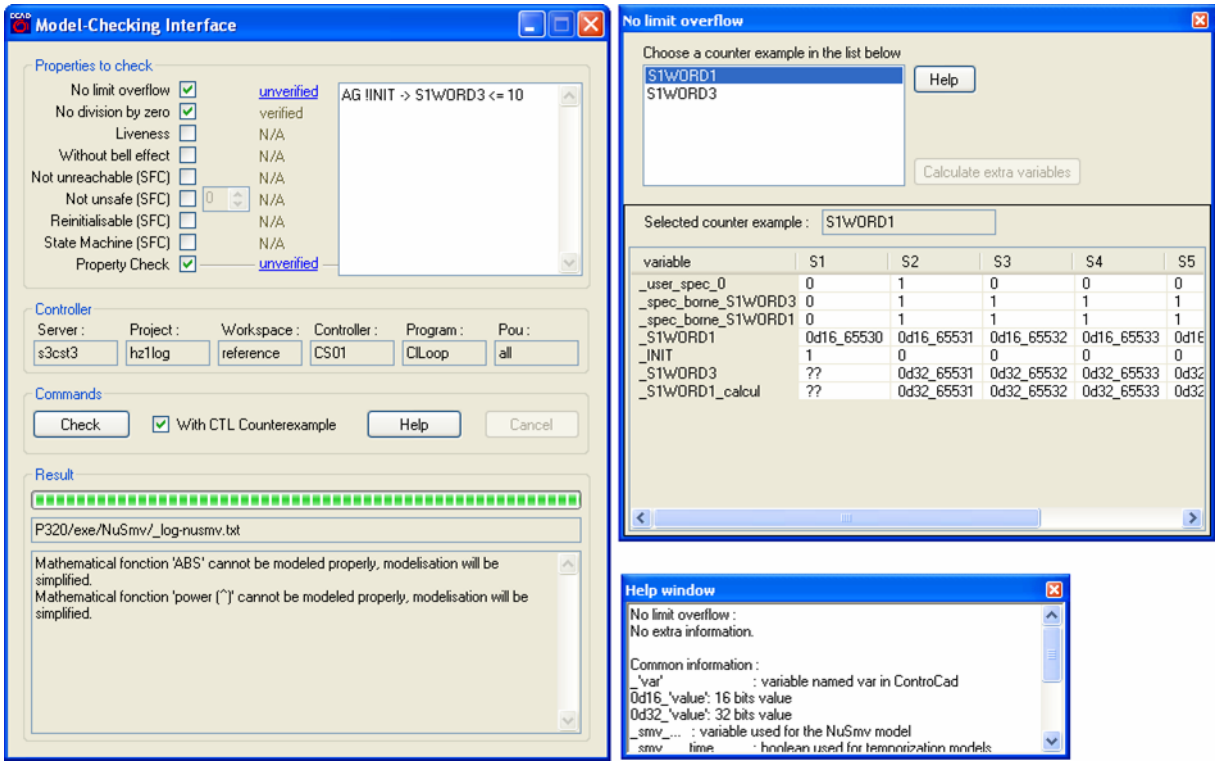


FIG. 6.2 – Interface utilisateur de l'outil de model-checking automatique

La fenêtre en haut, à droite sur la figure 6.2 présente l'interface de représentation des contre-exemples. Il importe de souligner nettement que les 8 premiers types de propriétés font référence à des propriétés intrinsèques, ce qui est tout à fait surprenant, étant donné l'objectif initial de nos travaux. En fait, les représentations formelles que nous avons développées permettent de vérifier, *en sus des propriétés extrinsèques du contrôleur*, certaine de ses propriétés intrinsèques, comme il le sera montré aux sections 6.4 et 6.5.

Dans sa version actuelle, les contre-exemples sont donnés sous la forme d'une liste de valeurs successives (trace d'exécution) mais des améliorations, par exemple des simulations d'évolution de programmes SFC ou FBD, sont envisagées.

L'outil de model-checking automatique présenté dans cette section a été conçu, développé et testé lors de nos travaux. Il a permis de valider le bien-fondé de nos propositions sur des cas industriels. Il devrait, dans les mois qui suivent, être disponible pour une utilisation dans les bureaux d'études d'automatismes.

Nous allons maintenant détailler, dans les sections suivantes, les propriétés qu'il est possible de vérifier avec cet outil intégré à Controcad.

6.2 Vérification de propriétés extrinsèques

Les modèles obtenus avec la méthode présentée dans le chapitre 3 et reposant sur l'utilisation de blocs fonctionnels décrits dans le chapitre 5 ont été conçus pour permettre la vérification de propriétés extrinsèques (définies dans le chapitre 2). Cette focalisation sur les propriétés extrinsèques permet deux abstractions (restriction aux états pertinents et aux R-variables) qui conduisent à améliorer l'efficacité des modèles (quantifiée dans le chapitre 4).

En plus, la représentation proposée permet de simplifier l'expression des propriétés extrinsèques.

En effet, la vérification de propriétés extrinsèques sur des modèles de type [dSR02] nécessite l'utilisation d'une variable indiquant la fin de cycle du moniteur dans les propriétés. Ces propriétés extrinsèques ne concernant que l'état en fin de cycle, il ne faut pas considérer bien sûr les états intermédiaires de calcul. Ceci est réalisé en modifiant la propriété dans le but d'inhiber la vérification lors de ces états intermédiaires. Ceci devient plus difficile pour les propriétés de vivacité car celles-ci nécessitent de mémoriser la valeur de chaque variable pour la comparer à la valeur au prochain cycle du contrôleur, impliquant d'autant plus de variables dans le modèle.

La représentation présentée dans ce mémoire ne nécessite plus ce type de considération : le modèle est directement le reflet de l'état du contrôleur industriel en fin de cycle. Tous les états sont donc concernés par les propriétés extrinsèques. Les types usuels de propriétés qu'il est possible de vérifier, ainsi que leurs formules CTL, sont donnés dans le tableau 6.1

De ce fait les propriétés CTL découlent naturellement pour chaque type de propriétés extrinsèques donné dans la section 2.3.3. Le tableau 6.1 donne la forme des quatre types de propriétés utilisables sur les modèles issus des méthodes présentées dans ce mémoire.

Dans le cas de l'application industrielle que nous avons développée, les propriétés de sûretés et d'atteignabilité ont été conservées inchangées, tandis que les propriétés de vivacité et d'équité ont été modifiées de façon à les exprimer en fonction des valeurs des variables de sortie du contrôleur (tableau 6.2).

La propriété de vivacité des variables de sortie indique que la valeur d'une variable ne reste pas indéfiniment la même au cours du fonctionnement. La propriété de stabilisation

Dénomination	Description	Propriété CTL
Propriété de sûreté	Un état dangereux q ne sera jamais atteint	$AG\ !q$
Propriété de vivacité	Un état q sera toujours atteint.	$AF\ q$
Propriété d'équité	Un état q apparaîtra une infinité de fois.	$AG\ EF\ q$
Propriété d'atteignabilité	Un état q pourra être atteint.	$EF\ q$

TAB. 6.1 – Forme des propriétés extrinsèques utilisables sur les modèles formels

des sorties spécifie que si les entrées n'évoluent plus (variable $freeze$ vraie) alors l'état où toutes les sorties seront et resteront fixes sera atteint ($change\_state$ sera fausse, indiquant qu'aucune sortie n'évolue).

Dénomination	Description	Propriété CTL
Propriété de sûreté	Un état dangereux q ne sera jamais atteint	$AG\ !q$
Propriété de vivacité des sorties	Aucune variable de sortie ne reste bloquée	pour chaque variable $< var >$: $AG(< var > \rightarrow EF(! < var >))$ $AG(! < var > \rightarrow EF(< var >))$
Propriété de stabilisation des sorties	Les variables de sortie finissent toujours par se stabiliser si les entrées n'évoluent plus.	$AG(freeze \rightarrow AF(AG!change\_state))$
Propriété d'atteignabilité	Sous certaines conditions, un état q pourra être atteint.	EFq

TAB. 6.2 – Propriétés extrinsèques retenues pour l'utilisation industrielle

6.3 Vérification de l'équivalence comportementale

La deuxième préconisation de la norme CEI 61508 concernant les programmes informatiques est la vérification de la cohérence des modèles produits lors des différentes étapes de leur cycle de vie. Pour ce faire, la vérification doit porter sur l'absence de perte d'information et d'introduction d'erreurs (voir chapitre 1 pour plus de détails).

Dans ce domaine, [Riv04] propose de certifier la compilation et les programmes compilés lors du développement de contrôle en aéronautique. Ces travaux sont issus de préoccupations similaires aux nôtres, les contraintes provenant de normes aéronautiques et non de la CEI 61508. Cette approche utilise une représentation des programmes basée sur les fonctions de transfert symboliques pour vérifier que la source et le programme compilé ont

le même comportement. Ces fonctions de transfert sont soit concrètes (Translation Validation) soit abstraites (Invariant Translation) et permettent de vérifier des invariants précis au niveau du langage assembleur. Cependant, cette méthode s'adresse aux programmes implantés sur des contrôleurs non-cycliques. Les modèles obtenus ne peuvent faire l'objet des deux abstractions détaillées dans le chapitre 3. De plus, la certification proposée par [Riv04] n'est effectuée que sur les invariants indiqués et non sur l'ensemble du programme.

Notre méthode, présentée chapitre 3, permet de modéliser les différents programmes en prenant en compte le comportement cyclique du contrôleur et s'appuie sur deux abstractions issues de ce comportement. Les modèles obtenus permettent de vérifier l'équivalence comportementale des programmes de contrôleur logiques. La figure 6.3 présente une vue globale de la méthode retenue.

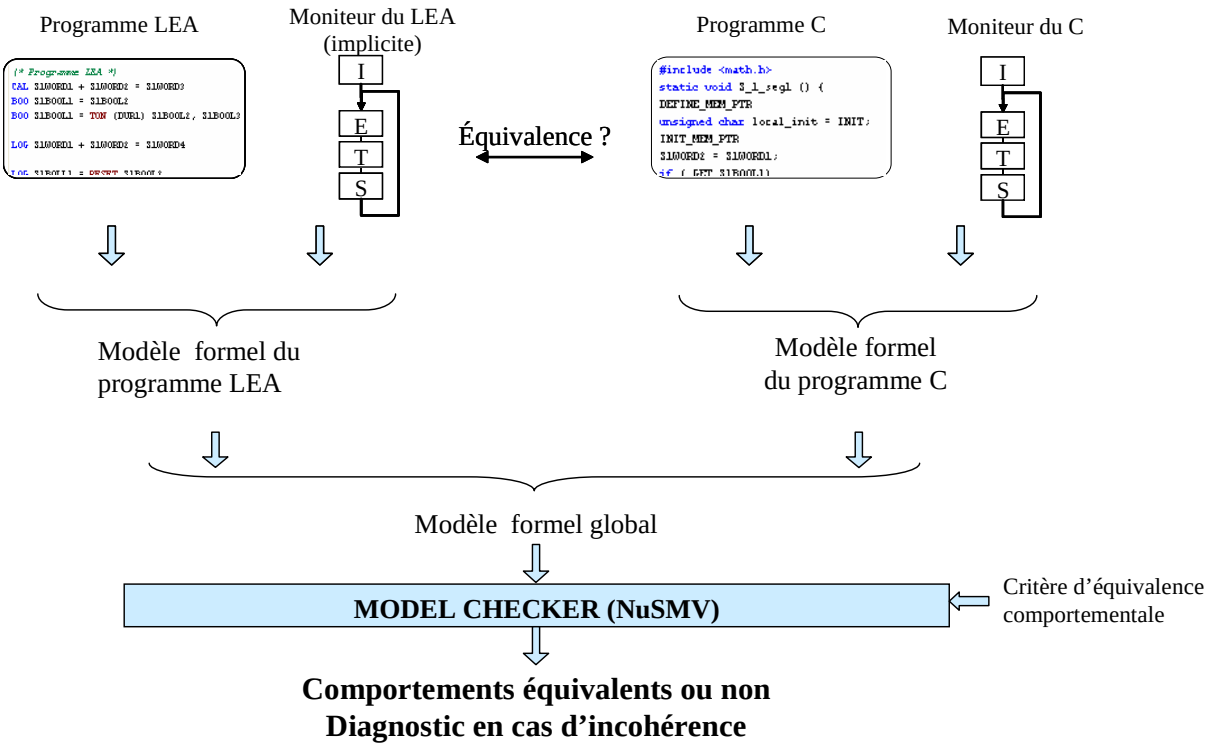


FIG. 6.3 – Méthode de vérification de l'équivalence comportementale

L'équivalence comportementale de deux programmes est définie par une propriété à vérifier pour chaque variable de sortie :
Le long de tous les chemins d'exécution, tous les états vérifient l'égalité des valeurs de la variable de sortie calculée par chacun des programmes.
Si $sortie1\_LEA$ et $sortie1\_C$ sont les deux valeurs d'une variable $sortie1$ calculées respectivement par les programmes LEA et C, la traduction en logique temporelle CTL de la propriété pour cette variable est :

$$AG(sortie1\_LEA = sortie1\_C)$$

Deux conditions sont nécessaires pour vérifier si deux modèles (et donc les programmes

qu'ils représentent) sont équivalents :

- Les indéterminismes doivent être communs.
  Ceci implique que les deux modèles doivent avoir les mêmes variables d'entrée (e_1, e_2, ...) et utiliser le même temps logique (synchronisation par la variable Time_elapsed sur la figure 6.4)
- Les valeurs initiales des variables de sortie des deux modèles doivent être identiques. Ceci est représenté par les égalités $init(si\_LEA) = init(si\_C)$ sur la figure 6.4.

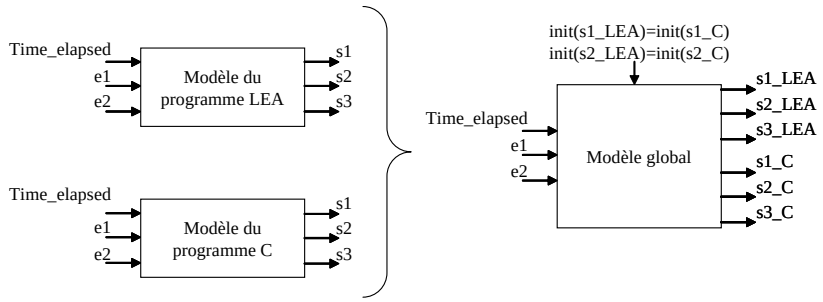


FIG. 6.4 – Synchronisation des évolutions de deux modèles pour la vérification de l'équivalence comportementale

6.4 Vérification de propriétés intrinsèques générales

Nous avons vu dans les deux sections précédentes que les possibilités de vérification à l'aide de nos modèles répondent aux objectifs énoncés au début de ce mémoire, dans le chapitre 1. Cependant le champ d'application de ces modèles s'est révélé plus large.

En effet, les abstractions de conservent certaines informations nécessaires à la vérification de propriétés autres qu'extrinsèques. Notamment, les modèles obtenus permettent la vérification de certaines propriétés intrinsèques générales (développées dans cette section), ainsi que de propriétés intrinsèques propres aux SFC (développées dans la section suivante).

L'abstraction d'interprétation conduit à supprimer les états correspondant à des affectations intermédiaires de variables. Ceci pourrait laisser penser qu'il n'est pas possible de vérifier des propriétés intrinsèques relatives à ces affectations, telles que "*Pas de division par zéro*" ou "*Pas de dépassement de borne (overflow)*". Il n'en est rien. En effet, si l'abstraction d'interprétation supprime les états non pertinents, elle ne supprime pas les affectations correspondantes mais les reporte dans l'affectation finale des variables de sortie. Il est donc possible de vérifier les deux propriétés intrinsèques ci-dessus, comme indiqué dans le tableau 6.3, en utilisant les affectations finales des variables de sortie.

La seule difficulté réside dans la présence éventuelle d'affectations de variables qui ne sont pas des R-variables et qui n'interviennent pas dans le calcul de R-variables. Il faut dans ce cas effectuer la vérification sur l'expression correspondant à chacune de ces variables.

Dénomination	Description	Propriété CTL
Pas de division par zéro (No division by zero)	Une division par zéro est-elle possible ?	Pour toutes les divisions de type <exp1> DIV <exp2> : AG (!(< $exp1 > DIV < exp2 >= 0$))
Pas de dépassement de borne (No limit overflow)	Existe-t-il un dépassement de borne d'un entier ?	Pour toutes les opérations susceptibles de déborder de type $< exp1 > OP < exp2 >$ AG (!($< exp1 > OP < exp2 >$) > N)

Tab. 6.3 – Possibilité de vérification de propriétés intrinsèques

6.5 Vérification de propriétés intrinsèques de SFC

La norme CEI 61131-3 indique deux propriétés inhérentes à la qualité de la commande d'un programme écrit en SFC :

Code mort (*Unreachable*)
Il ne doit pas exister d'étapes non utilisées ou non atteignables.

Non sûr (*Unsafe*)
La multiplication incontrôlée des jetons d'activité d'étape doit être évitée.

La première propriété est une notion assez courante dans le domaine de la programmation qui vise à éliminer les *codes morts*. Ces portions de programme jamais utilisées posent deux problèmes majeurs. Premièrement, elles nuisent à la compréhension du programme. Deuxièmement, en cas de modification du programme, ces portions peuvent être réactivées et produire des effets de bord non désirés. La figure 6.5 présente un exemple de programme SFC ayant ce type de comportement.

La deuxième propriété est liée à l'interprétation du langage SFC. Lors de l'utilisation de divergence en *ET* dans un SFC, le nombre de jetons d'activité des étapes est augmenté. Si la structure du SFC fait que le nombre de jetons n'est jamais diminué par la suite, par exemple par manque de convergence en *ET* systématique, alors le nombre de jetons est continuellement augmenté. Le SFC peut alors être *rempli* de jeton et de plus réaliser la fonction attendue. La figure 6.6 présente un exemple de programme écrit en langage SFC ayant un comportement *non-sûr*.

Les méthodes présentées dans ce mémoire sont dédiées aux langages textuels. Cependant, pour vérifier un SFC, il suffit de le transformer en sa représentation algébrique. [MDL+06b] propose une représentation algébrique d'un langage équivalent au SFC : le Grafcet. L'évolution du SFC diffère principalement par l'utilisation de priorité des transitions. Nous allons donc adapter cette représentation au langage SFC.

La représentation algébrique proposée, découpée en trois étapes, est présentée ci-dessous. Pour des questions de lisibilité et comme cette transformation n'est pas le but premier de ce mémoire, nous nous limitons aux structures *simples* du SFC (pas de temporisation, d'action conditionnelle, de macro-étapes, ...). Nous utiliserons dans ce cadre les abréviations ci-dessous.
Concernant une étape i :

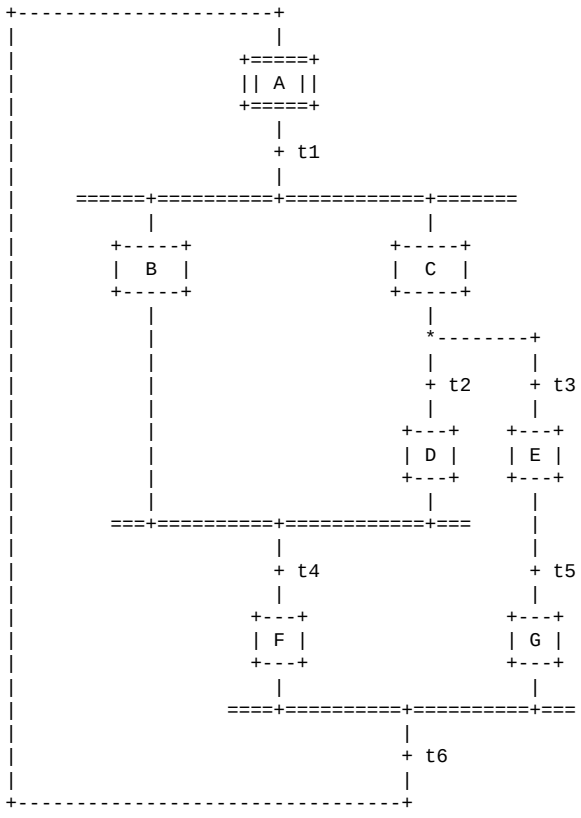


FIG. 6.5 – Programme SFC ayant des étapes non atteignables (*unreachable*)

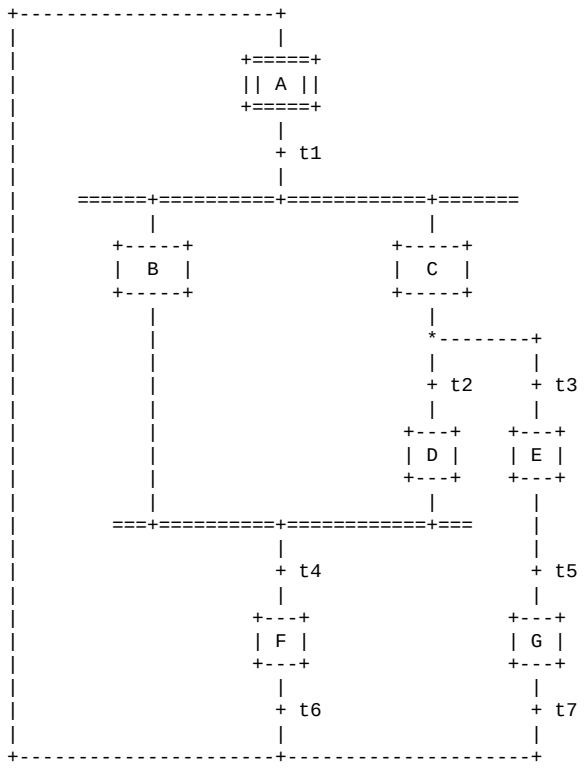


FIG. 6.6 – Programme SFC ayant un comportement non-sûr (*unsafe*)

- X_i : la variable booléenne représentant l'activité de étape i,
- $\mathcal{N}_i$: l'ensemble des transitions qui suivent l'étape i,
- $\mathcal{P}_i$: l'ensemble des transitions qui précèdent l'étape i.

Concernant une transition q :

- Cf_q : la variable booléenne représentant la possibilité de franchissement de la transition q,
- Tc_q : la condition associée à la transition q (réceptivité),
- $\mathcal{A}_q$: l'ensemble des étapes qui précédent la transition q,
- $\mathcal{D}_q$: l'ensemble des transitions qui appartiennent à la même (aux mêmes) structure(s) de type divergence en OU (sélection de séquence) que la transition q et qui ont une priorité supérieure à celle de la transition q.

Concernant une action l :

- Ac_l : la variable booléenne représentant l'émission de l'action l,
- $\mathcal{M}_l$: l'ensemble des étapes où l'action l est émise.

Etape 1 : calcul des conditions de franchissabilité

Il est possible de franchir une transition q si : toutes les étapes immédiatement précédentes sont actives, sa condition associée Tc_q vraie et que la priorité est respectée par rapport aux autres transitions de l'ensemble Dq. Donc Cf_q peut être calculée

avec la formule :

$$Cf_q = (\prod_{j \in \mathcal{A}_i} X_j).Tc_q.\prod_{k \in \mathcal{D}_q} \overline{Cf_k}$$

Etape 2 : calcul des variables d'étapes

Une étape devient active si une des transitions amont est franchissable. Elle devient inactive si une transition aval est franchissable. L'activation est prioritaire sur la désactivation.En cas d'absence de conditions d'activation et de désactivation, l'étape concerne son état. Donc la valeur de X_i peut être calculée avec la formule :

$$X_i = \sum_{j \in \mathcal{P}_i} Cf_j + X_i.\prod_{k \in \mathcal{N}_i} \overline{Cf_k}$$

Etape 3 : calcul des actions

Une action l est émise si l'une des étapes auxquelles elle est associée est active. Ac_l peut donc être calculée comme suit :

$$Ac_l = \sum_{j \in \mathcal{M}_l} X_j$$

Sur la base de cette représentation algébrique, il est alors possible de traduire tout SFC en un langage textuel, tel que le LEA ; cette possibilité est présente dans l'outil Controcad [CH07b, CH07a].

A partir du modèle formel d'un SFC, déduit de sa traduction en langage textuel selon les principes exposés au chapitre 3, les propriétés intrinsèques indiquées dans le tableau 6.4 peuvent être vérifiées.

6.6 Conclusion

Les résultats théoriques obtenus lors de nos travaux ont été intégrés à l'outil Controcad, afin de permettre à l'ingénieur automaticien d'accéder aux techniques de preuve.

Le prototype de logiciel développé durant cette thèse permet non seulement la vérification de propriétés extrinsèques et de l'équivalence comportementale de contrôleurs, objectifs initiaux de ces travaux, mais encore la vérification de certaines propriétés intrinsèques, en particulier pour des contrôleurs décrits en SFC.

Même si certaines améliorations, notamment en termes d'interprétation des contre-exemples, sont souhaitables, ce prototype doit permettre dans un futur proche l'utilisation de techniques de vérification formelle en bureau d'étude d'automatismes.

Dénomination	Description	Propriété CTL
Not unsafe	Le nombre de jetons d'activité d'étapes reste toujours inférieur à n (donné par l'utilisateur).	$AG\ \left(\left(\sum_{algébrique} X_i\right) < n\right)$
Not unreachable	Toutes les étapes du SFC peuvent devenir actives.	pour chaque étape i $EF\ X_i$
Reinitialisable	Le SFC peut toujours revenir dans son état initial.	$AG\ EF\ \left(\prod_{i\in\mathcal{I}} X_i . \prod_{k\in\mathcal{N}} \overline{X_k}\right)$ $\mathcal{I}$ ensemble des étapes initiales $\mathcal{N}$ ensemble des étapes non-initiales
State machine	Une et une seule étape est active à tout instant	$AG\ \left(\left(\sum_{algébrique} X_i\right) = 1\right)$

Tab. 6.4 – Possibilités de vérification de propriétés intrinsèques de SFC

Conclusions et Perspectives

L'objectif scientifique de cette thèse était de fournir des représentations formelles des contrôleurs logiques industriels qui permettent le passage à l'échelle des méthodes de vérification de type model-checking non temporisé. Ceci afin de permettre, en contexte industriel, l'utilisation de ces techniques de preuves lors du développement des systèmes de contrôle-commande critiques, dans la perspective de leur certification.

Pour atteindre cet objectif scientifique, nous avons en premier lieu proposé les contributions suivantes :

- une **abstraction d'interprétation**, qui conserve uniquement les états pertinents pour la vérification de propriétés extrinsèques ;
- une **abstraction de données**, qui ne conserve que les variables qu'il est nécessaire de mémoriser (R-variables) dans chaque état du modèle formel.

Sur la base de ces propositions, nous avons développé un algorithme générique permettant de construire, à partir d'un contrôleur logique décrit dans un langage de type ST (*Structured Text*), **une représentation formelle efficace** de ce contrôleur.

L'efficacité de cette représentation a été **validée et quantifiée**. Vis-à-vis des représentations proposées antérieurement, les gains sont très importants pour tous les critères qui permettent d'évaluer l'efficacité (nombre d'états atteignables, diamètre du système, temps de vérification, mémoire nécessaire à la preuve).

En outre, afin de permettre aux concepteurs de systèmes de contrôle-commande critiques de définir sans ambiguïté le savoir-faire encapsulé dans les blocs fonctionnels, nous avons proposé :

- une représentation graphique de ces blocs fonctionnels adaptée aux besoins ;
- une définition formelle de la **sémantique des blocs fonctionnels**, à l'aide de machines de Moore et de systèmes de transition.

Ces deux propositions permettent la constitution d'une bibliothèque de modèles formels de blocs fonctionnels, à laquelle il est possible de faire appel lors de la vérification de contrôleurs qui incluent ces blocs.

Tous ces résultats théoriques ont été intégrés dans l'outil de développement d'automatismes industriels Controcad. Le prototype industriel réalisé lors de ces travaux permet la vérification des propriétés extrinsèques de contrôleurs industriels, ainsi que la vérification de l'équivalence comportementale de deux contrôleurs, ces deux types de vérification étant ceux initialement considérés dans l'optique d'une aide à la certification. De plus, certaines propriétés intrinsèques, relatives en particulier au langage SFC, sont également vérifiables avec ce prototype. Ceci atteste de la richesse des représentations formelles proposées.

Nous estimons donc que ces travaux ont atteint leur objectif initial et l'ont même dépassé sur certains points.

Il n'en demeure pas moins vrai que plusieurs thèmes de recherche peuvent être développés sur la base de ces résultats, avec l'objectif d'améliorer la diffusion industrielle des méthodes de vérification formelle.

En se remémorant les difficultés relatives à la mise en oeuvre de ces méthodes (Cf. chapitre 2, section 2.1.4), et même si cette recherche a contribué à résoudre les difficultés relatives à l'expression du comportement du contrôleur dans un langage formel et à l'explosion combinatoire, nous pensons qu'il convient dans un futur proche de lever les verrous technologiques concernant l'expression des propriétés formelles et l'interprétation des contre-exemples, en cas de preuve négative. Pour ce faire, il nous paraît utile de développer des recherches sur :

- la définition de bibliothèques de propriétés formelles métier. En model-checking, les propriétés formelles sont en général classées en grandes catégories, fonctions d'objectifs généraux (ce qu'il faut faire, ce qu'il ne faut pas faire, ce qui peut se produire, ...) exprimables facilement à l'aide d'associations d'opérateurs de la logique temporelle retenue. Nous pensons qu'il serait intéressant de s'intéresser à la définition de propriétés plus précises, fonction du processus à contrôler. Par exemple, la conception d'un bloc fonctionnel de démarrage de turbine devrait impliquer la définition de propriétés permettant de vérifier le bon contrôle de cette turbine lors de son démarrage.

- l'interprétation des contre-exemples à l'aide de simulateur. Un outil de model-checking génère, en cas de preuve négative, un contre-exemple ; cependant, ce contre-exemple est donné sous la forme d'une trace d'exécution difficile à interpréter. Même si la représentation formelle proposée dans cette thèse permet de réduire la taille des contre-exemples, conséquence de la diminution du diamètre des modèles formels manipulés, et donc facilite leur compréhension, des recherches méritent d'être entreprises afin d'améliorer la présentation des exécutions conduisant au non-respect des propriétés. Nous pensons que l'utilisation d'un outil de simulation des comportements du contrôleur et du processus contrôlé est une voie prometteuse pour ce faire.

A plus long terme, les travaux rapportés dans ce mémoire, relatifs à la vérification formelle de systèmes non temporisés devront être étendus aux systèmes temporisés, en considérant des outils de preuve comme TSMV (extension temporisée de NuSMV) ou UPPAAL (model-checker à base d'automates à états temporisés et synchronisés). Ceci conduira à évoluer d'une modélisation logique du temps vers la prise en compte du temps physique.

La vérification de modèles hybrides, avec une approche model-based, est également une piste de recherche prometteuse. Il conviendra dans ce cas d'étudier le couplage entre les représentations formelles de contrôleurs proposées et des modèles hybrides représentatifs des processus contrôlés.

Enfin il conviendrait d'étudier les possibilités d'extensions des abstractions proposées dans ce mémoire. Nous pensons en particulier que l'abstraction de données, qui nous a conduit à la définition des R-variables, variables mémorisant l'historique du contrôleur sur un horizon temporel d'un cycle, doit pouvoir être étendue afin de considérer des horizons temporels plus larges.

Bibliographie

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1) :2–34, 1993.

[BBF+01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[BBG+05] H. Bel Mokadem, B. Bérard, V. Gourcuff, J.-M. Roussel, and O. de Smet. Verification of a timed multitask system with Uppaal. In *ETFA'05*, pages 347–354, Catania, Italy, September 2005.

[BCC98] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. *LNCS*, 1536 :81–102, 1998.

[BCL+94] J.-R. Burch, E.-M. Clarke, D.-E. Long, K.-L. MacMillan, and D.-L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4) :401–424, 1994.

[BCOQ92] F.L. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat. *Synchronization and Linearity : An Algebra for Discrete Event Systems*. Wiley Series on Probability and Mathematical Statistics : Probability and Mathematical Statistics, 1992.

[Bel06] Houda Bel Mokadem. *Vérification des propriétés temporisées des automates programmables industriels*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS de Cachan, France, September 2006.

[BFG+97] RI Bahar, EA Frohm, CM Gaona, GD Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebric Decision Diagrams and Their Applications. *Formal Methods in System Design*, 10(2) :171–206, 1997.

[BLG90] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the Signal language. *IEEE Transactions on Automatic Control*, 35(5) :535–546, 1990.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *LNCS*, volume 1066, pages 232–243. SV, 1996.

[BM76] J.A. Bondy and U.S.R. Murty. *Graph theory with applications*. Macmillan London, 1976.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[C+99] E.M. Clarke et al. *Program Slicing of Hardware Description Languages*. Springer, 1999.

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2 : An OpenSource Tool for Symbolic Model Checking. In *Proc. CAV 2002*, volume 2004 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[CGS07] M.P. Cabasino, A. Giua, and C. Seatzu. Marking estimation of Petri nets with arbitrary transition labeling. In *1st IFAC workshop on Dependable Control of Discrete Systems*, pages 109–114, Paris, France, 2007.

[CLA06] G. Cengic, O. Ljungkrantz, and K. Akesson. Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime. In *IEEE Conference on Emerging Technologies and Factory Automation (ETFA'06)*, pages 1269–1276, 2006.

[CYC04] J. Cho, J. Yoo, and S. Cha. NuEditor - a tool suite for specification and verification of NuSCR. In *Second ACIS international conference on software engineering research, management and applications (SERA2004)*, pages 298–304, 2004.

[DA92] R. David and H. Alla. *Petri Nets and Grafcet : Tools for Modelling Discrete Event Systems*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[dSR02] O. de Smet and O. Rossi. Verification of a controller for a flexible manufacturing line written in ladder diagram via model-checking. In *ACC'02*, pages 4147–4152, Anchorage (USA), May 2002.

[E.R05] E.Rakotomalala. *Spécifications robustes du système de pilotage d'une fonction électronique automobile*. PhD thesis, École centrale de Nantes et Université de Nantes, décembre 2005.

[ES96] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design : An International Journal*, 9(1/2) :105–131, August 1996.

[FL98] G. Frey and L. Litz. Verification and validation of control algorithms by coupling of interpreted Petri nets. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, 1998.

[FL00] G. Frey and L. Litz. Formal methods in PLC programming. In *Proceedings of the IEEE SMC 2000*, pages 2431–2436, October 2000.

[FLS02] J.M. Faure, J.J. Lesage, and C. Schnakenbourg. Towards IEC 61499 function blocks diagrams verification. In *IEEE Int. Conference on Systems, Man and Cybernetics (SMC02)*, October 2002.

[Fre05] Goran Frehse. Phaver : Algorithmic verification of hybrid systems past hytech. In *HSCC*, pages 258–273, 2005.

[FRV06] L. Ferrarini, M. Romano, and C. Veber. Automatic Generation of AWL Code from IEC 61499 Applications. In *IEEE International Conference on Industrial Informatics*, pages 25–30, 2006.

[FV04] L. Ferrarini and C. Veber. Implementation approaches for the execution model of IEC 61499 applications. In *2nd IEEE International Conference on Industrial Informatics (INDIN'04)*, pages 612–617, 2004.

[GdSF06a] V. Gourcuff, O. de Smet, and J.-M. Faure. Détermination de l'équivalence comportementale d'algorithmes de contrôle - commande. In *AFADL'06*, pages 111–125, Paris (France), mars 2006.

[GdSF06b] V. Gourcuff, O. de Smet, and J.-M. Faure. Efficient representation for formal verification of PLC programs. In *WODES'06*, pages 182–187, Ann Arbor, USA, July 2006.

[Gun96] J. Gunnarsson. Algebraic methods for discrete event systems : a tutorial. In *Workshop On Discrete Event Systems (WODES'96)*, 1996.

[H+74] P.R. Halmos et al. *Naive Set Theory*. Springer, 1974.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Springer, 1993.

[Hen96] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[HF06] T. Hussain and G. Frey. UML-based Development Process for IEC 61499 with Automatic Test-case Generation. In *IEEE Conference on Emerging Technologies and Factory Automation (ETFA'06)*, pages 1277–1284, 2006.

[HHWT97] T.A. Henzinger, P.H. Ho, and H. Wong-Toi. HYTECH : a model-checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1) :110–122, 1997.

[HKG97] L.E. Holloway, BH Krogh, and A. Giua. A Survey of Petri Net Methods for Controlled Discrete Event Systems. *Discrete Event Dynamic Systems*, 7(2) :151–190, 1997.

[HLB03] R. Huuck, B. Lukoschus, and N. Bauer. A model-checking approach to safe SFCs. In *Proc. of CESA 2003*, Lille (France), July 2003.

[Huu05] R. Huuck. Semantics and Analysis of Instruction List Programs. In *SFEDL'2004*, pages 3–18, January 2005.

[IEC93] IEC (International Electrotechnical Commission). *IEC Standard 61131-3 : Programmable controllers - Part 3*, 1993.

[IEC00] IEC (International Electrotechnical Commission). *IEC Standard 61508 : Functional safety of electrical/electronic/ programmable electronic safety-related systems*, 2000.

[IEC04] IEC (International Electrotechnical Commission). *IEC Standard 61499 : Function blocks for industrial-process measurement and control systems*, 2004.

[Jen97] K. Jensen. Coloured Petri Nets–Basic Concepts, Analysis Methods and Practical Use, Vol. 1 : Basic Concepts. *Monographs in Theoretical Computer Science. Springer-Verlag*, 1997.

[JFR01] F. Jimenez-Fraustro and E. Rutten. A Synchronous Model of IEC 61131 PLC Languages in SIGNAL. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 135. IEEE Computer Society Washington, DC, USA, 2001.

[Joh07] T.L. Johnson. Improving automation software dependability : A role for formal methods ? *Control Engineering Practice*, 15(11) :1403–1415, 2007.

[KJH06] H. Kaghazchi, R. Joyce, and D. Heffernan. Function Blocks for Fieldbus Diagnostics. In *IEEE Conference on Emerging Technologies and Factory Automation (ETFA'06)*, pages 322–327, 2006.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3) :155–163, 1988.

[KP96] S. Kowalewski and J. Preußig. Verification of sequential controllers with timing functions for chemical processes. In *IFAC 13th World Congress*, volume J, pages 419–424, San Francisco, USA, 1996.

[LCRRL99] S. Lampérière-Couffin, O. Rossi, J.-M. Roussel, and J.-J. Lesage. Formal validation of PLC programs : a survey. In *European Control Conference 1999 (ECC'99), Karlsruhe, Germany, Aug.-Sep. 1999*, 1999. Proc. on CD-ROM, communication 741.

[Lev95] N.G. Leveson. *Safeware : system safety and computers*. ACM Press, New York, NY, USA, 1995.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, and D. Probst. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1) :11–44, 1995.

[LYF05] K. Loeis, M.B. Younis, and G. Frey. Application of Symbolic and Bounded Model Checking to the Verification of Logic Control Systems. In *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA'05)*, volume 1, pages 247–250, Sept. 2005.

[Mad00] A. Mader. A classification of PLC models and applications. In *WODES'2000*, pages 239–247, August 21-23 2000.

[McM99] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, 1999.

[MDL06a] J. Machado, B. Denis, and J.-J. Lesage. A generic approach to build plant models for DES verification purposes. In *WODES'06*, pages 407–412, Ann Arbor (USA), July 2006.

[MDL+06b] J. Machado, B. Denis, J.-J. Lesage, J.-M. Faure, and J. C. L. Ferreira Da Silva. Logic controllers dependability verification using a plant model. In *DESDes'06*, pages 37–42, Rydzyna (Poland), September 2006.

[Moo56] E.F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, 34 :129–153, 1956.

[Moo94] I. Moon. Modeling programmable logic controllers for logic verification. In *Control Systems Magazine, IEEE*, pages 53–59. IEEE Comp. Soc. Press, 1994.

[PF05] S. Panjaitan and G. Frey. Functional Design for IEC 61499 Distributed Control Systems using UML Activity Diagram. In *International Conference Instrumentation, Communication and Information Technology (ICICI)*, pages 64–70, 2005.

[PL00] P. Pettersson and K.G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70(40-44) :2, 2000.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, volume 46-57, 1977.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351. Springer-Verlag London, UK, 1982.

[RAB+95] R.K. Ranjan, A. Aziz, R.K. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM International Workshop on Logic Synthesis (IWLS'95)*, page 254, May 1995.

[RD02] J.M. Roussel and B. Denis. Safety properties verification of Ladder Diagram programs. *Journal européen des systèmes automatisés*, 36(7) :905–917, 2002.

[Riv04] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *Principles of Programming Languages (POPL 2004)*, 2004.

[RK98] M. Rausch and B. Krogh. Formal verification of PLC programs. In *American Control Conference*, pages 234–238, PA, USA, June 1998.

[Ros03] O. Rossi. *Validation formelle de programmes Ladder Diagram pour Automates Programmables Industriels*. PhD thesis, ENS de Cachan, 2003.

[SK91] RS Sreenivas and BH Krogh. On condition/event systems with discrete state realizations. *Discrete Event Dynamic Systems*, 1(2) :209–236, 1991.

[SRF06] I.S. Barragan Santiago, M. Roth, and J.M. Faure. Obtaining temporal and timed properties of logic controllers from fault tree analysis. In *INCOM 2006*, pages 243–248, Saint-Etienne, France, May 2006.

[Tip94] F. Tip. *A Survey of Program Slicing Techniques*. Centrum voor Wiskunde en Informatica, 1994.

[VA04] Shobha Vasudevan and Jacob A. Abraham. Static program transformations for efficient software model-checking. In *IFIP Congress Topical Sessions*, pages 257–282, 2004.

[vdW99] E. van der Wal. Introduction into IEC 1131-3 and PLCopen. In *IEEE Colloquium on The Application of IEC 61131 to Industrial Control : Improve Your Bottom Line Through High Value Industrial Control Systems (Ref. No. 1999/076)*, page 2, 1999.

[VH99] V. Vyatkin and H.M. Hanisch. A modeling approach for verification of IEC1499 function blocks using net condition/event systems. In *7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'99)*, volume 1, 1999.

[VK02] N. Völker and B.J. Krämer. Automated verification of function block-based industrial control systems. *Science of Computer Programming*, 42(1) :101–113, 2002.

[VKP05] V. Vyatkin, S. Karras, and T. Pfeiffer. Architecture for automation system development based on IEC 61499 standard. In *3rd IEEE International Conference on Industrial Informatics (INDIN'05)*, pages 13–18, 2005.

[YCKS05] J. Yoo, S. Cha, C.H. Kim, and D.Y. Song. Synthesis of FBD-based PLC design from NuSCR formal specification. *Reliability Engineering and System Safety*, 87(2) :287–294, 2005.

[Zou04] B. Zoubek. *Automatic verification of temporal and timed properties of control programs.* PhD thesis, University of Birmingham, Sept. 2004.

Bibliographie technique

[CH07a] Sandrine Couffin and Laurent Holz. *CONTROCAD - Principales fonctionnalités de l'éditeur SFC*, 2007. réf. P-TP21-A40041-FC.

[CH07b] Sandrine Couffin and Laurent Holz. *SFC - Règles d'évolution*, 2007. réf. P-TP21-A40041-FC.

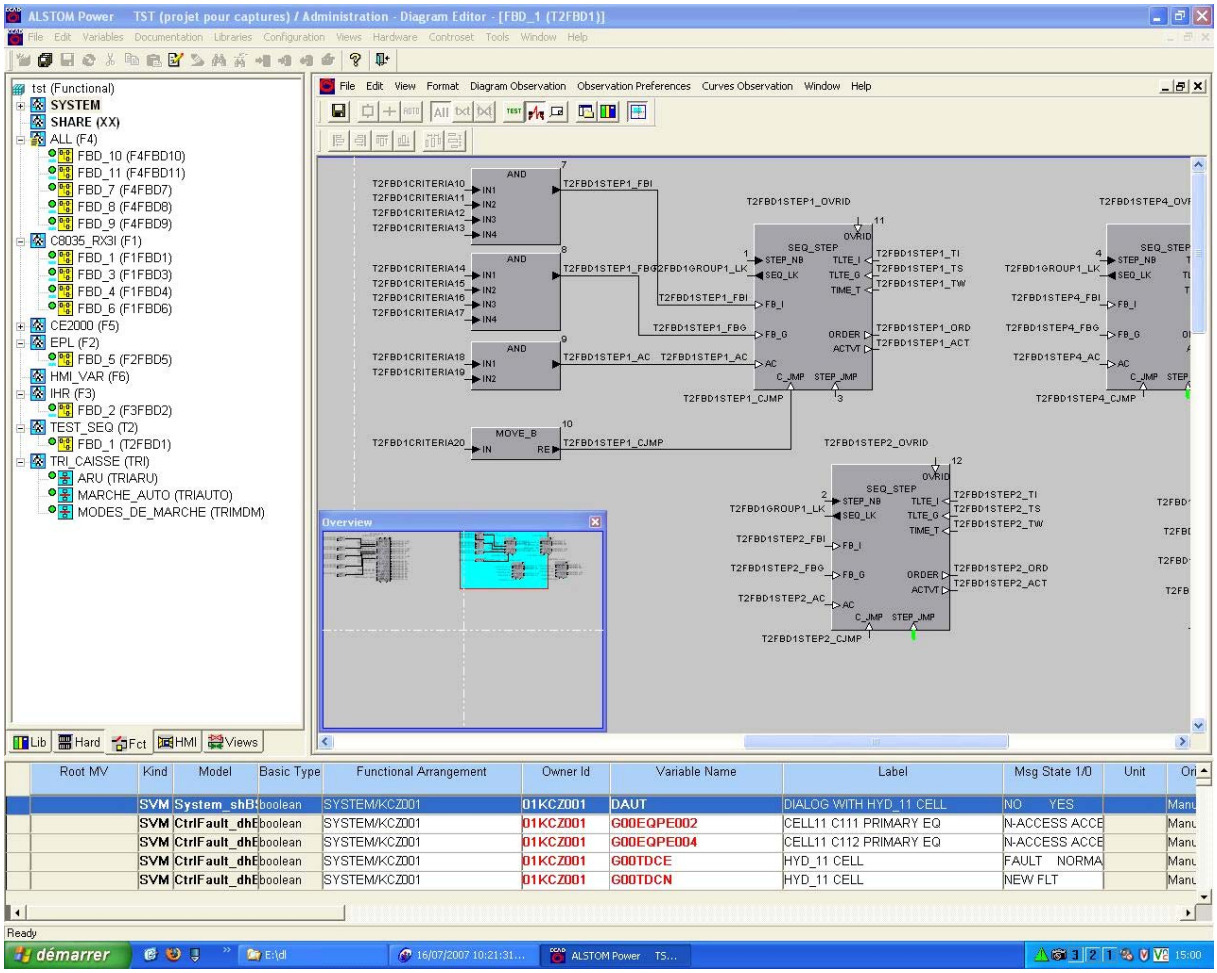
[GCV07] Vincent Gourcuff, Sandrine Couffin, and François Vallernaud. *Spécification de l'outil de model-checking automatique - Contribution à la vérification de la sûreté de fonctionnement logicielle du programme applicatif généré*, 2007. réf. P-TP21-A41103-FB.

[P-T00] *Langage littéral de CONTROCAD - Manuel de Référence*, 2000. réf. P-TP21-A40004-AA.

[Y3-04] *Description applicative des BF pour applications de sécurite*, 2004. réf. Y3-32 A423581-C.

Annexe A

Capture d'écran Controcad



Annexe B

Modèles NuSMV complets de l'exemple a) du chapitre 3

B.1 Méthode présentée dans [dSR02]

```
MODULE main
VAR
O_1 : boolean;
O_2 : boolean;
O_3 : boolean;
O_4 : boolean;
O_5 : boolean;
I_1 : boolean;
I_2 : boolean;
I_3 : boolean;
I_4 : boolean;
  PLC : { system, read, execute, write };
  PLC_cp_line : 0 .. 6 ;

ASSIGN
  init(PLC) := system;
  init(PLC_cp_line) := 1;
--  init(O_1 ) := 0;
--  init(O_2 ) := 0;
--  init(O_3 ) := 0;
--  init(O_4 ) := 0;
--  init(O_5 ) := 0;
init(I_1 ) := 0;
init(I_2 ) := 0;
init(I_3 ) := 0;
init(I_4 ) := 0;

  next(PLC) := case
    (PLC=system) : read ;
    (PLC=read) : execute ;
    (PLC=execute) & (PLC_cp_line=6) : write ;
    (PLC=execute) & ! (PLC_cp_line=6) : execute ;
    (PLC=write) : system ;
```

```
  esac;

  next(PLC_cp_line) := case
    (PLC=execute) & (PLC_cp_line<6) : PLC_cp_line+1 ;
    (PLC=execute) & (PLC_cp_line=6) : 0 ;
    (PLC=read) : 1;
    1 : PLC_cp_line;
  esac;

  next(I_1) := case
    (PLC=read) : {0,1};
    1 : I_1 ;
  esac;
  next(I_2) := case
    (PLC=read) : {0,1};
    1 : I_2 ;
  esac;
  next(I_3) := case
    (PLC=read) : {0,1};
    1 : I_3 ;
  esac;
  next(I_4) := case
    (PLC=read) : {0,1};
    1 : I_4 ;
  esac;

next(O_1) := case
(PLC_cp_line=1) :  I_1 | I_2;
(PLC_cp_line=6) :  !(I_2 | I_4);
1:O_1;
esac;

next(O_2) := case
(PLC_cp_line=2) :  I_3 & I_4;
1:O_2;
esac;

next(O_3) := case
(PLC_cp_line=3) :  case
           O_1: I_3 & !(I_4);
           1 : O_3;
           esac;
1:O_3;
esac;

next(O_4) := case
(PLC_cp_line=4) :  case
I_1 : 0;
O_5 : 1;
1 : O_4;
esac;
1:O_4;
esac;
```

```
next(O_5) := case
(PLC_cp_line=5) :  O_2 & O_4;
1:O_5;
esac;
```

B.2 Méthode présentée dans [GdSF06b]

```
MODULE main
VAR
  O_1 : boolean;
  O_2 : boolean;
  O_3 : boolean;
  O_4 : boolean;
  O_5 : boolean;
  I_1 : boolean;
  I_2 : boolean;
  I_3 : boolean;
  I_4 : boolean;

ASSIGN
  init(O_1 ) := 0;
  init(O_2 ) := 0;
  init(O_3 ) := 0;
  init(O_4 ) := 0;
  init(O_5 ) := 0;
  init(I_1 ) := 0;
  init(I_2 ) := 0;
  init(I_3 ) := 0;
  init(I_4 ) := 0;

  next(O_2) := next(I_3) & next(I_4);
  next(O_3) := case
                  next(I_1) | next(I_2) : next(I_3) & !next(I_4);
                !(next(I_1) | next(I_2)) : O_3;
               esac;
  next(O_4):= case
                 O_5 : 1;
                 next(I_1) : 0;
                 1 : O_4;
              esac;
  next(O_5) := next(O_2) & next(O_4);
  next(O_1) := !(next(I_2) | next(I_4));
```

B.3 Méthode présentée dans ce mémoire

```
MODULE main
VAR
  O_3 : boolean;
  O_4 : boolean;
```

```
  O_5 : boolean;
  I_1 : boolean;
  I_2 : boolean;
  I_3 : boolean;
  I_4 : boolean;

ASSIGN
--   init(O_3 ) := 0;
--   init(O_4 ) := 0;
--   init(O_5 ) := 0;
  init(I_1 ) := 0;
  init(I_2 ) := 0;
  init(I_3 ) := 0;
  init(I_4 ) := 0;

  next(O_3) := case
                  next(I_1) | next(I_2) : next(I_3) & !next(I_4);
                !(next(I_1) | next(I_2)) : O_3;
               esac;
  next(O_4):= case
                 next(I_1) : 0;
                 O_5 : 1;
                 1 : O_4;
              esac;
  next(O_5) := next(I_3) & next(I_4) & next(O_4);
```

Annexe C

Algorithme générique et application

la section 3.3 du chapitre 3 nous a permis de présenter la démarche pas-à-pas de modélisation d'un contrôleur industriel en modèle NuSMV en utilisant les abstractions décrites dans la section 3.2. Cette annexe propose un algorithme qui inclut toutes les étapes décrites dans ces sections.

L'algorithme C.1 prend pour entrée le programme du contrôleur industriel logique et permet de retourner le modèle NuSMV correspondant codé comme indiqué auparavant.

Au départ,

- **les variables sont considérées comme des NR-variables**
  Puis, s'il est trouvé durant l'exécution de l'algorithme que la variable est utilisée dans son état courant, alors elle sera marquée comme R-variable.
- **sauf mention contraire la valeur par défaut d'une variable est sa valeur courante k sauf mention contraire, les variables sont considérées comme utilisées dans leur état courant k**
  Tout les cas où une variable est utilisée dans son état futur $k+1$ sont indiqués dans l'algorithme.

Les dépendances entre variables sont analysées à la volée. En effet, si le programme est analysé dans le sens de son exécution, alors les informations collectées sur les affectations amont permettent de modéliser les affectations aval.

Cependant une information initiale supplémentaire doit être connue : **Quelle est la dernière affectation de chaque variable ?**. Vu l'aspect trivial de cette information, elle est considérée comme connue et n'est pas identifiée par l'algorithme. Elle permet de savoir si l'affectation traitée fournit la valeur finale (en fin de cycle) ou temporaire (entre deux affectations) de la variable.

Le modèle, sous forme de GTS utilisable sur NuSMV, obtenu avec cet algorithme retourne les affectations de chaque R-variable (les équations du modèle GTS) et les expressions temporaires des NR-variables (les déductions logiques).

Afin de mieux comprendre l'algorithme C.1, voici l'application détaillée de celui ci à l'*exemple a*. Sur cette application, chaque étape est numérotée ainsi :
<numéro de ligne dans l'algorithme>.<numéro de la ligne de programme actuellement traitée>

1 Le bloc fonctionnel *RS* est remplacé par sa représentation générique. Donc la quatrième déclaration est remplacée par :
O_4 := case
I_1 : 0;
O_5 : 1;
1 : O_4;
esac ;

3.1 La première déclaration du programme est la ligne 1 du programme.

4.1 C'est une affectation : celle de O_1 par une expression S_1 contenant deux variables d'entrée (I_1 et I_2).

5.1 Il n'y a pas de variables de sortie dans l'expression S_1.

13.1 Deux variables d'entrée I_1 et I_2 sont présentes dans l'expression S_1.

14.1 Les deux variables d'entrée I_1 et I_2 sont donc remplacées par leur valeur future $I_{1,i+1}$ et $I_{2,i+1}$.

15.1 Ce n'est pas la dernière affectation de O_1 et O_1 est une NR-variable

16.1 Cette expression ($I_{1,i+1}$ OR $I_{2,i+1}$) est donc mémorisée comme temporaire pour la variable O_1 et cette affectation est supprimée.

3.2 La deuxième déclaration est la ligne 2 du programme.

4.2 à 14.2 C'est l'affectation de O_2 par une expression S_2 contenant deux variables d'entrée I_3 et I_4, qui sont donc remplacées par leur état futur $I_{3,i+1}$ et $I_{4,i+1}$.

15.2 C'est la dernière affectation de O_2 et O_2 est une NR-variable.

16.2 L'expression ($I_{3,i+1}$ AND $I_{4,i+1}$) est donc mémorisée comme temporaire pour la variable O_2 et cette affectation est supprimée.

3.3 La troisième déclaration est la ligne 3.

17.3 C'est une structure conditionnelle avec une condition *cond*, une déclaration dans le *then* et aucune déclaration dans le *else*.

18.3 La condition *cond* est une expression contenant O_1.

19.3 O_1 à déjà été affectée avant (ligne 1)

20.3 O_1 est une NR-variable et sa dernière affectation n'est pas passée.

23.3 O_1 est donc remplacée par son expression temporaire, soit $I_{1,i+1}$ OR $I_{2,i+1}$.

26.3 O_3 est affectée dans la structure conditionnelle

27.3 O_3 est affectée la partie *else*.

29.3 O_3 n'est pas affectée la partie *then*.

30.3 L'affectation $O_3 := O_3$; est donc ajoutée dans la partie *else*.

31.3 Model_Auto de la partie *then* de la structure conditionnelle

1 aucun bloc fonctionnel n'est utilisé ici

3.3-1 la première déclaration de cette partie est la ligne 3-1

4.3-1 à 14.3-1 c'est l'affectation de O_3 par une expression S_{3-1} contenant les deux entrée I_3 et I_4, qui sont donc remplacées par leur état futur $I_{3,i+1}$ et $I_{4,i+1}$.

15.3-1 C'est la dernière affectation de O_3 et O_3 est une NR-variable.

16.3-1 L'expression S_{3-1} ($I_{3,i+1}$ AND NOT($I_{4,i+1}$)) est donc mémorisée comme temporaire pour la variable O_3.

43.3-1 Retourner que l'expression temporaire O_3 est $I_{3,i+1}$ AND NOT($I_{4,i+1}$) et que O_3 est une NR-variable.

32.3 Model_Auto de la partie *else* de la structure conditionnelle

1 aucun bloc fonctionnel n'est utilisé ici

3.3-2 La première déclaration de cette partie est la ligne 3-2 (non présente dans le programme original mais qui correspond explicitement à $O_3 := O_3$;)

4.3-2 C'est l'affectation de O_3 par une expression S_{3-2} contenant une variable de sortie O_3

5.3-2 La variable de sortie O_3 est présente dans l'expression S_{3-2}.

5.3-2 O_3 n'a jamais été affectée auparavant.

12.3-2 O_3 est donc une R-variable.

13.3-2 Il n'y a pas de variable de entrée dans l'expression S_{3-2}.

15.3-2 O_3 est une R-variable.

43.3-2 Retourner que l'expression de O_3 est $O_{3,i}$ et que O_3 est une R-variable.

33.3 Seule O_3 est affecté dans la structure conditionnelle.

34.3 O_3 est une NR-variable dans la partie *then* et une R-variable dans la partie *else* donc O_3 est globalement une R-variable.

36.3 Les deux expressions de O_3 sont remplacées par :
$O_{3,i+1}$ = case
$I_{1,i+1}$ OR $I_{2,i+1}$: $I_{3,i+1}$ AND NOT($I_{4,i+1}$)
!$I_{1,i+1}$ OR $I_{2,i+1}$: $O_{3,i}$
esac

41.3 C'est la dernière affectation de O_3 et O_3 est une R-variable
Son affectation est donc conservée et correspond à son état futur $O_{3,i+1}$.

3.4 La quatrième déclaration est la ligne 4 du programme.

4.4 C'est l'affectation de O_4 par une expression S_4 (représentant le bloc fonctionel RS)

5.4 S_4 contient une variables de sortie : O_4

6.4 à 12.4 O_5 n'a pas encore été affectée et est donc une R-variable. Elle est considérée dans son état courant $O_{5,i}$ (par défaut).

6.4 à 12.4 O_4 n'a pas encore été affectée et est donc une R-variable. Elle est considérée dans son état courant $O_{4,i}$ (par défaut).

13.4 S_4 contient une variable de entrée I_1

14.4 I_1 est donc remplacée par son état futur $I_{1,k+1}$

15.4 C'est la dernière affectation de O_4 et elle est une R-variable ; son affectation (ci-dessous) est donc conservée et correspond à son état futur $O_{4,i+1}$:
$O_{4,i+1}$ = case
$I_{1,i+1}$: 0
$O_{5,i}$: 1
1 : $O_{4,i}$
esac

3.5 La cinquième déclaration est la ligne 5 du programme.

4.5 C'est l'affectation de O_5 par une expression S_5.

5.5 S_5 contient 2 variables de sortie O_2 et O_4.

6.5 à 12.5 O_2 a déjà été affectée, est une NR-variable et sa dernière affectation est passée. Elle est donc remplacée par son expression temporaire S_2 : $I_{3,i+1}$ AND $I_{4,i+1}$.

6.5 à 12.5 O_4 a déjà été affectée, est une R-variable et sa dernière affectation est passée. Elle est donc remplacée dans son état futur $O_{4,i+1}$.

13.5 S_5 ne contient aucune variable d'entrée.

15.5 C'est la dernière affectation de O_5 et elle est une R-variable. Son affectation avec l'expression S_4 (($I_{3,i+1}$ AND $I_{4,i+1}$) AND $O_{4,i+1}$) est donc conservée par défaut et correspond à sa valeur future $O_{5,i+1}$.

3.6 La sixième et dernière déclaration est la ligne 6.

4.6 C'est la deuxième affectation de O_1 par un expression S_6.

5.6 S_6 ne contient aucune variable de sortie.

13.6 et 14.6 S_6 contient les variables d'entrée I_2 et I_4 qui sont remplacées par leur état futur $I_{3,i+1}$ et $I_{4,i+1}$.

15.6 C'est la dernière affectation de O_1 et elle est une NR-variable.

16.6 Son affectation avec l'expression S_6 ($I_{2,i+1}$ AND NOT($I_{4,i+1}$)) est donc supprimée et mémorisée comme expression temporaire.

---

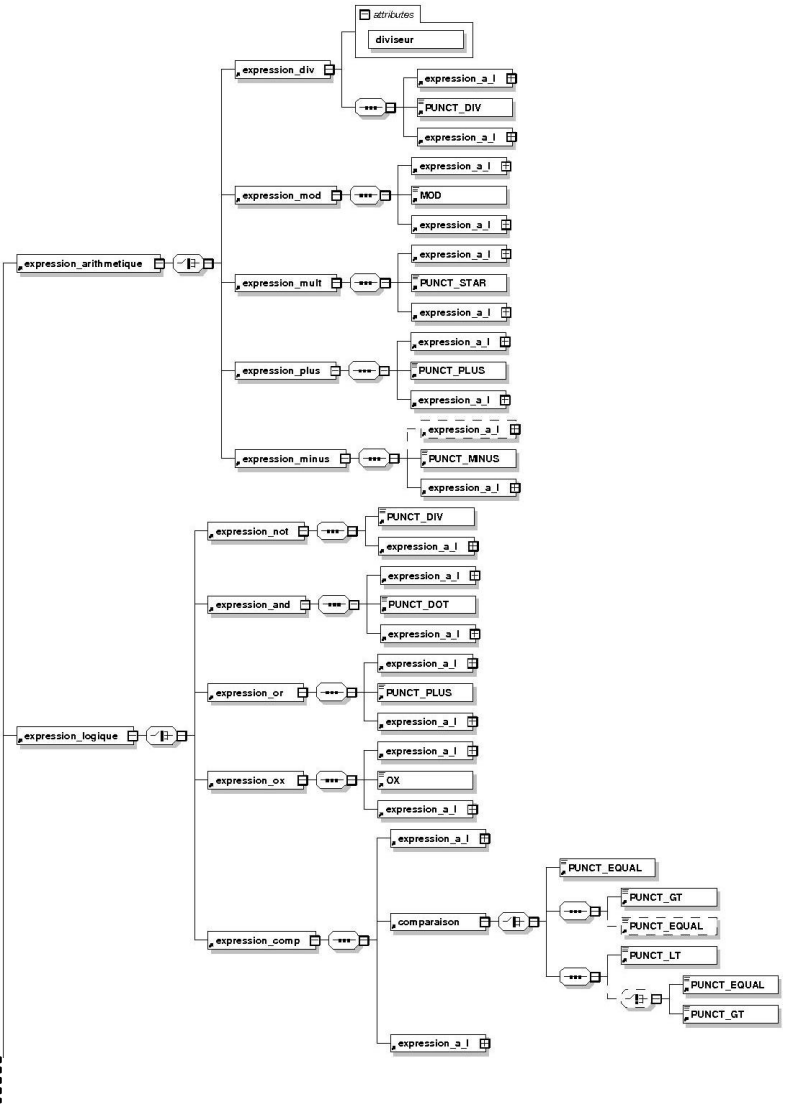
43 Le modèle GTS est retourné (voir figure 3.9) avec l'information que les variables O_1 et O_2 sont des NR-variables et que les variables O_3,O_4 et O_5 sont des R-variables.

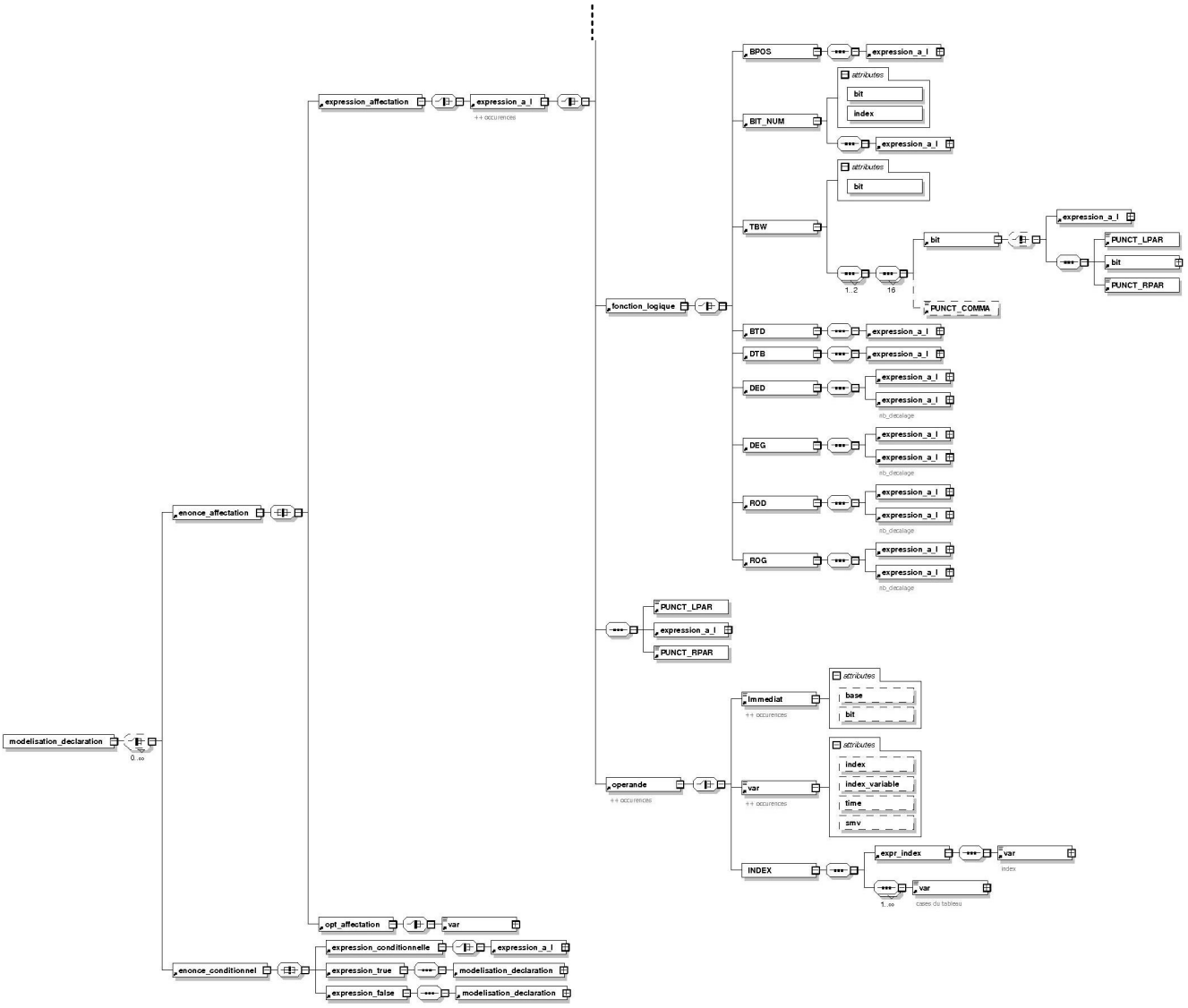
```
1  for chaque bloc fonctionnel BF_i dans Pr do
2      Remplacer BF_i par le modèle issue de la librairie.
3  for chaque déclaration S_i du programme Pr do
4      if cette déclaration S_i est une affectation (V_i := expression_i) then
5          for chaque variable de sortie V_k dans l'affectation do
6              if V_k a été affectée auparavant then
7                  if V_k est une R-variable et sa dernière affectation est passée then
8                      Remplacer V_k par next(V_k) ;
9                  else
10                     Remplacer V_k par son expression temporaire ;
11             else
12                 Marquer V_k comme R-variable ;
13         for chaque variable d'entrée I_k dans l'affectation do
14             Remplacer I_k par I_{k,i+1} ;
15         if ce n'est pas la dernière affectation de V_i ou que V_i est une NR-variable then
16             Mémoriser l'expression temporaire de V_i et supprimer l'affectation;
17     if la déclaration est une structure conditionnelle (if cond ; then stmt_1 ; else stmt_2) then
18         for chaque variable de sortie V_k dans cond do
19             if V_k a été affectée auparavant then
20                 if V_k est une R-variable et sa dernière affectation est passée then
21                     Remplacer V_k par next(V_k) ;
22                 else
23                     Remplacer V_k par son expression temporaire ;
24             else
25                 Marquer V_k comme R-variable ;
26         for chaque variable V_m affectée dans cette structure conditionnelle do
27             if V_m n'est pas affectée dans stmt_1 then
28                 Ajouter l'affectation " V_m = V_m " dans stmt_1 ;
29             if V_m n'est pas affectée dans stmt_2 then
30                 Ajouter l'affectation " V_m = V_m " dans stmt_2 ;
31         Lancer Model_Auto(stmt_1) ;
32         Lancer Model_Auto(stmt_2) ;
33         for chaque variable V_m affectée dans cette structure conditionnelle do
34             if V_m est une R-variable dans Model_Auto(stmt_1) ou Model_Auto(stmt_2) then
35                 Marquer V_m comme R-variable ;
36             Remplacer les expressions associées à V_m par :
37               "case
38                  cond : ≺ affectation de V_m dans Model_auto(stmt_1)≻ ;
39                  !cond : ≺ affectation de V_m dans Model_auto(stmt_2)≻ ;
40               esac"
41             if ce n'est pas la dernière affectation de V_m ou V_m est une NR-variable then
42                 Mémoriser l'expression temporaire de V_m ;
43 return chaque variable avec son affectation (le modèle GTS) et ses caractéristiques (R ou NR,
   expression temporaire, ...) ;
```

FIG. C.1 – Model_auto(Pr : **programme**) → **modèle NuSMV**

Annexe D

Grammaire du langage pivot




modelisation_declaration
0..∞
enonce_affectation
expression_affectation
expression_a_l
++ occurences
opt_affectation
var
enonce_conditionnel
expression_conditionnelle
expression_a_l
expression_true
modelisation_declaration
expression_false
modelisation_declaration
fonction_logique
BPOS
expression_a_l
BIT_NUM
attributes
bit
index
expression_a_l
TBW
attributes
bit
1..2
16
bit
expression_a_l
PUNCT_LPAR
bit
PUNCT_RPAR
PUNCT_COMMA
BTD
expression_a_l
DTB
expression_a_l
DED
expression_a_l
expression_a_l
nb_decalage
DEG
expression_a_l
expression_a_l
nb_decalage
ROD
expression_a_l
expression_a_l
nb_decalage
ROG
expression_a_l
expression_a_l
nb_decalage
PUNCT_LPAR
expression_a_l
PUNCT_RPAR
operande
++ occurences
immediat
++ occurences
attributes
base
bit
var
++ occurences
attributes
index
index_variable
time
smv
INDEX
expr_index
var
index
var
1..∞
cases du tableau